Distributed cost-optimal planning

Loïg Jezequel

THÈSE / ENS CACHAN - BRETAGNE

sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de
DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN
Mention : Informatique
École doctorale MATISSE

présentée par

Loïg Jezequel

Préparée à l'Unité Mixte de Recherche 6074
Institut de recherche en informatique
et systèmes aléatoires

Distributed Cost-Optimal Planning

Thèse soutenue le 13 novembre 2012
devant le jury composé de :

Sophie Pinchinat,
Professeure, Université Rennes 1 / *présidente du jury*

Malik Ghallab,
Directeur de Recherche, LAAS-CNRS / *rapporteur*

Marc Zeitoun,
Professeur, Université Bordeaux 1 / *rapporteur*

Javier Esparza,
Professeur, Technische Universität München / *examinateur*

Serge Haddad,
Professeur, ENS Cachan / *examinateur*

Patrik Haslum,
Chargé de recherche contractuel, Australian National University / *examinateur*

Éric Fabre,
Directeur de Recherche, INRIA Rennes Bretagne Atlantique / *directeur de thèse*

Remerciements

Je tiens en premier lieu à remercier les membres de mon jury : Sophie Pinchinat, qui m'a fait l'honneur de le présider ; Malik Ghallab et Marc Zeitoun, rapporteurs, pour leur relecture attentive du présent manuscrit ainsi que leurs remarques constructives avant et pendant la soutenance ; Javier Esparza, Serge Haddad et Patrik Haslum, examinateurs, pour avoir bien voulu s'intéresser à mon travail et participer à ce jury. De manière générale je les remercie tous pour le temps qu'ils ont consacré à ma thèse et pour leurs questions pertinentes qui m'ont ouvert de nombreuses pistes de réflexion.

Je remercie aussi Sylvie Thiébaux, mes collègues de l'IRISA – et en particulier les membres des équipes DistribCom, S4 et VerTeCS –, ainsi que l'ensemble des chercheurs avec qui j'ai pu avoir des discussions enrichissantes lors de séjours dans d'autres laboratoires, lors de conférences, ou lors de leurs visites à l'IRISA.

Enfin, je voudrais remercier tout particulièrement Eric Fabre, mon directeur de thèse, pour m'avoir donné l'opportunité de travailler sur ce sujet passionnant, pour ses conseils avisés lors de la rédaction de ce manuscrit et en préparation de la soutenance mais aussi tout au long des trois ans et demi que nous avons passés à travailler ensemble depuis le début de mon stage de master. J'espère sincèrement que nous pourrons continuer à collaborer à l'avenir.

Contents

Résumé long en français

LA PLANIFICATION EST un domaine de l'intelligence artificielle où l'objectif est de permettre à un système d'atteindre un état particulier (appelé but) au moyen d'actions qu'il faut choisir et ordonner. Les problèmes de planifications sont en fait fortement liés à ceux de recherche de chemins. En effet, il est possible de représenter les états du système considérés par les sommets d'un graphe et les actions modifiant ces états par les arcs (orientés) de ce graphe. Trouver une solution (généralement appelée plan) à un problème de planification revient alors à trouver un chemin dans le graphe correspondant. On peut donc résoudre ces problèmes en utilisant des algorithmes classiques de recherche de chemins. Cependant on souhaite fréquemment, plutôt que simplement trouver un plan, en trouver un qui soit de bonne qualité. Cette notion de qualité peut être définie en associant à chaque action un coût (généralement un nombre réel positif) permettant de définir la qualité d'un plan à partir de la somme de coûts des actions le constituant : plus cette somme sera petite, plus le plan sera de bonne qualité. On cherche alors à trouver des plans du plus faible coût possible, c'est-à-dire des chemins de coût minimal dans un graphe valué. En planification ceci se fait traditionnellement à l'aide de l'algorithme A* (ou d'une de ses variantes), qui a été proposé en 1968 par Hart, Nilsson et Raphael [37]. Cet algorithme consiste en une recherche de chemins dans un graphe où le choix des directions à explorer en priorité est biaisé par des informations sur le coût maximal des chemins prenant ces directions. Toute la difficulté de l'utilisation de cet algorithme réside dans la recherche de ces informations biaisant la recherche, c'est à dire dans la construction de ce qu'on appelle une heuristique. De nombreuses méthodes existent pour calculer des heuristiques, aussi bien pour des problèmes particuliers [20] que sans faire d'hypothèses sur les problèmes (par exemple en utilisant des méthodes d'abstraction [46] ou de relaxation [43] des problèmes).

Plus récemment de nouvelles méthodes de planification sont apparues, tirant avantage de relations d'indépendance existant entre certaines actions au sein d'un problème. En effet, il arrive que deux actions n'aient pas d'influence l'une sur l'autre, par exemple (dans le cas où un état du système est représenté par une valuation d'un ensemble de variables) lorsqu'elles ne modifient pas les mêmes variables ni ne touchent aux variables lues par l'autre. Dans ce cas, l'ordre dans lequel ces actions sont exécutées n'a pas d'importance. Ceci permet de représenter les plans non plus comme des séquences d'actions mais comme des ordres partiels d'actions, ce qui peut réduire grandement le nombre de plans potentiels à considérer pour en trouver un qui soit correct. Un des premiers exemples d'algorithme de planification utilisant à son avantage l'indépendance entre actions au sein d'un problème est GRAPHPLAN [8]. Cet algorithme se base sur une représentation des plans par une structure de données, appelée graphe de planification, qui a quelques défauts, notamment en ce qui concerne la détection de conflits

entre actions : seuls les conflits binaires (entre deux actions) sont pris en compte alors qu'il peut exister des conflits plus complexes. Une façon de prendre en compte plus que les conflits binaires est d'utiliser des représentations différentes des plans, comme par exemple les dépliages de réseaux de Petri [47].

Enfin, durant les dix dernières années, une autre façon d'utiliser l'indépendance entre actions, mais aussi potentiellement entre variables, dans un problème de planification est apparue. Cette approche, appelée planification factorisée, a été initialement proposée par Amir et Engelhardt dans l'article éponyme *Factored Planning* [1]. Le principe de cette méthode est de décomposer un problème de planification en plusieurs sous-problèmes (ou facteurs, ou composants) le plus indépendants possible, qui sont plus simples à résoudre que le problème d'origine puisque de taille potentiellement grandement inférieure. L'idée est alors de trouver une solution locale à chacun des sous-problèmes de telle sorte que ces solutions puissent-être assemblées en un plan pour le problème considéré (ou solution globale). Plusieurs façons d'exploiter cette idée ont été proposées (notamment dans [12] et [13]) mais toutes ont pour défaut de se baser sur des bornes (sur la longueur des plans globaux ou sur le nombre de synchronisations entre plans locaux) incrémentées progressivement au cours de la recherche de solutions. Ceci empêche de garantir l'optimalité des solutions trouvées car il n'y a pas a priori de corrélation entre la longueur des plans (en nombre d'actions) et leur coût : un plan optimal peut être très long.

Une description plus précise de l'évolution des méthodes de planification depuis l'algorithme A* jusqu'aux approches factorisées est présentée au chapitre 1 du présent document.

Dans cette thèse nous proposons deux nouvelles approches de la planification factorisée dont le principal intérêt est de permettre la recherche de plans optimaux. La première est basée sur le calcul sur les automates à poids pour représenter l'ensemble des solutions d'un facteur et sur une famille d'algorithmes, bien connue en satisfaction de contraintes, pour raffiner ces ensembles de solutions locales afin de trouver des solutions globales. Le seconde approche consiste en une version distribuée de l'algorithme A* où un agent est responsable de chaque sous-problème et cherche une solution localement en dirigeant sa recherche grâce à un biais local (c'est-à-dire grâce à une heuristique standard pour le facteur considéré) mais aussi grâce à un biais global obtenu des autres agents.

Planification factorisée par passage de messages

Une première approche de la planification factorisée permettant la recherche de solutions optimales aux problèmes considérés est proposée dans les chapitres 2 à 5 de ce document. Le principe de cette approche est de représenter un problème de planification comme un réseau d'automates à poids, c'est à dire un ensemble de tels automates se synchronisant par une version pondérée du produit synchrone. On peut alors prouver qu'utiliser une famille d'algorithmes (dits à passage de messages) bien connus dans le domaine de la satisfaction de contraintes permet de résoudre de façon modulaire ces problèmes.

Passage de messages et planification

Formellement, un problème de planification factorisée, est représenté par un ensemble $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ d'automates à poids $\mathcal{A}_i = (S_i, S_i^I, S_i^F, \Sigma_i, T_i, c_i, c_i^i, c_i^f)$ où S_i est un

ensemble fini d'états, $S_i^I \subseteq S_i$ est un ensemble d'états initiaux, $S_i^F \subseteq S_i$ est un ensemble d'états finaux, Σ_i est un alphabet d'actions, $T_i \subseteq S_i \times \Sigma_i \times S_i$ est un ensemble de transitions, $c_i : T_i \to \mathbb{R}_+$ associe un coût à chaque transition, $c_i^i : S_i^I \to \mathbb{R}_+$ associe un coût à chaque état initial et $c_i^f : S^F \to \mathbb{R}_+$ associe un coût à chaque état final. Les notions de chemin et de mot sont définies de manière standard, par contre on peut leur associer un coût : pour un chemin il s'agira de la somme des coûts des transitions le constituant, et pour un mot il s'agira du coût du plus petit chemin le réalisant (en prenant en compte le coût de l'état initial et de l'état final utilisés pour accepter le mot). Le langage $\mathcal{L}(\mathcal{A}_i)$ d'un tel automate est alors l'ensemble des mots acceptés par l'automate associés à leurs coûts. Résoudre le problème de planification factorisée défini par $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ consiste à trouver un mot de coût minimal dans $\mathcal{L} = \mathcal{L}(\mathcal{A}_1) \times_{\mathcal{L}} \cdots \times_{\mathcal{L}} \mathcal{L}(\mathcal{A}_n)$ où le produit de langages ($\times_{\mathcal{L}}$) est défini de la manière suivante : $\mathcal{L}_i \times_{\mathcal{L}} \mathcal{L}_j = \{(w, c) \ : \ (w_{|\Sigma_i}, c_i) \in \mathcal{L}_i \wedge (w_{|\Sigma_j}, c_j) \in \mathcal{L}_j \wedge c = c_i + c_j\}$, avec $w_{|\Sigma}$ le mot obtenu à partir de w en supprimant les actions ne faisant pas partie de l'alphabet Σ.

Étant donné un problème $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ on définit alors son graphe d'interaction comme le graphe $G = (V, E)$ tel que $V = \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ et il y a un arc de E entre $\mathcal{A}_i$ et $\mathcal{A}_j$ si et seulement si ils partagent des actions : $\Sigma_i \cap \Sigma_j \neq \emptyset$. Dans un tel graphe certains arcs sont dits redondants, il s'agit des arcs $(\mathcal{A}_i, \mathcal{A}_j) \in E$ tels qu'il existe un chemin dans G entre $\mathcal{A}_i$ et $\mathcal{A}_j$ n'utilisant pas l'arc $(\mathcal{A}_i, \mathcal{A}_j)$ et ne passant que par des automates $\mathcal{A}_k$ tels que $\Sigma_k \supseteq \Sigma_i \cap \Sigma_j$. Un graphe obtenu à partir du graphe d'interaction d'un problème en retirant de manière récursive des arcs redondants jusqu'à ce que ce ne soit plus possible est appelé graphe de communication (pour un problème donné il peut en exister plusieurs différents, cependant si l'un d'entre eux est un arbre alors tous le sont).

L'algorithme 1 a pour entrée un problème de planification factorisée $\{\mathcal{A}_1, \ldots, \mathcal{A}_n\}$ et l'un de ses graphes de communication $G = (V, E)$. Il retourne, pour chaque composant $\mathcal{A}_i$, une version mise à jour $\mathcal{L}(\mathcal{A}_i)'$ du langage de ce composant. La notation $\mathcal{N}(\mathcal{A}_i)$ représente l'ensemble des voisins de $\mathcal{A}_i$ dans G. Le langage $\mathcal{L}_{\mathbb{I}}$ a pour alphabet l'ensemble vide et pour seul mot ε. Les $\mathcal{M}_{i,j}$ sont des messages transmis entre composants. Enfin on utilise une notion de projection pour réduire le plus possible la quantité d'information transmise entre composants : $\Pi'_{\Sigma}(\mathcal{L}) = \{(w_{|\Sigma'}, c) \ : \ (w, c) \in \mathcal{L} \wedge c = \min_{(w', c') \in \mathcal{L}, w'_{|\Sigma'} = w_{|\Sigma'}}(c')\}$.

Algorithme 1 Passage de messages dans les problèmes de planification factorisée.

pour tout $(\mathcal{A}_i, \mathcal{A}_j) \in E$ **faire**
  $\mathcal{M}_{i,j} \leftarrow \mathcal{L}_{\mathbb{I}}$
fait
jusqu'à stabilité des messages **faire**
  sélectionner $(\mathcal{A}_i, \mathcal{A}_j) \in E$
  $\mathcal{M}_{i,j} \leftarrow \Pi_{\Sigma_i \cap \Sigma_j}(\mathcal{L}(\mathcal{A}_i) \times_{\mathcal{L}} (\prod_{\mathcal{A}_k \in \mathcal{N}(\mathcal{A}_i) \setminus \{\mathcal{A}_j\}} \mathcal{M}_{k,i}))$
fait
pour tout $\mathcal{A}_i \in V$ **faire**
  $\mathcal{L}(\mathcal{A}_i)' = \mathcal{L}(\mathcal{A}_i) \times_{\mathcal{L}} (\prod_{\mathcal{A}_k \in \mathcal{N}(\mathcal{A}_i)} \mathcal{M}_{k,i})$
faire

Théorème 0.1. *Si G est un arbre, alors l'algorithme 1 converge et*

$$\forall i, \mathcal{L}(\mathcal{A}_i)' = \Pi_{\Sigma_i}(\mathcal{L})$$

.

À partir de ce théorème et en remarquant que:

1. quelque soit w un mot de $\mathcal{L}$ de coût c minimal, le mot $w_{|\Sigma_i}$ est un mot de coût minimal de $\Pi_{\Sigma_i}(\mathcal{L})$ et a lui aussi pour coût c ;
2. de même quelque soit w_i un mot de $\Pi_{\Sigma_i}(\mathcal{L})$ de coût c_i minimal, il existe un mot w de $\mathcal{L}$ de coût c minimal et tel que $w_{|\Sigma_i} = w_i$;

on obtient une méthode de résolution des problèmes de planification factorisée basée uniquement sur des calculs locaux (c'est à dire impliquant des voisins dans le graphe de communication considéré). Cette méthode consiste à utiliser l'algorithme 1 pour calculer, pour chaque composant $\mathcal{A}_i$ du problème, la valeur de $\mathcal{L}(\mathcal{A}_i)'$, puis à sélectionner dans l'un de ces langages $\mathcal{L}(\mathcal{A}_k)'$ un mot w_k de coût minimal, puis dans chacun des langages $L(\mathcal{A}_\ell)'$ des voisins de $\mathcal{A}_k$ un mot w_ℓ compatible avec w_k (c'est à dire, en étendant légèrement la notion de produit, tel que $w_\ell \times_{\mathcal{L}} w_k$ est non vide) et de coût minimal (ce qui est possible d'après la remarque précédente), puis dans chacun de leurs voisins un mot de coût minimal compatible avec w_k et tous les w_ℓ, et ainsi de suite. On obtient alors un mot w_i par automate $\mathcal{A}_i$ tels que ces mots peuvent être entrelacés en un mot de $\mathcal{L}$.

Ce formalisme pour la planification factorisée, ainsi que l'utilisation des algorithmes par passage de message pour résoudre ces problèmes (et notamment la preuve de la validité de cette approche) sont présentés en détails dans le chapitre 2 de ce document.

Mise en œuvre

Cependant, on ne peut pas travailler directement sur les langages car ce sont des objets potentiellement infinis. Il est toutefois possible de définir les opérations nécessaires à l'algorithme 1 (produit et projection) directement sur les automates, qui sont des objets finis, et donc d'utiliser cet algorithme en pratique. Nous avons implanté cette méthode dans un programme du nom de Distoplan en nous basant notamment sur le travail de Mohri [76] pour ce qui est des opérations sur les automates à poids (déterminisation, minimisation, ε-reduction). En utilisant notre implantation nous avons pu comparer notre méthode à d'autres approches (A* avec une heuristique efficace et SATPLAN) sur des jeux de tests classiques de planification (utilisés lors de compétitions internationales). Les résultats obtenus se sont révélés très positifs, même si nous avons trouvé peu de problèmes adaptés à la planification factorisée. En particulier, Distoplan a obtenu des résultats meilleurs que A* sur tous les problèmes, tout en passant à l'échelle de façon comparable à SATPLAN (les différences d'ordre de grandeur des temps de calcul entre Distoplan et SATPLAN pourraient s'expliquer par le fait que SATPLAN ne garantit pas l'optimalité des plans trouvés).

Notre implémentation ainsi que les résultats expérimentaux obtenus sont décrits au chapitre 3 de ce document.

Utilisation d'algorithmes turbo

La principale faiblesse de notre approche de la planification factorisée est qu'elle ne fonctionne que lorsque les graphes de communication du problème considéré sont des arbres. Pour contourner ce problème il est possible d'utiliser des méthodes de décomposition de graphes. Ces méthodes ont pour objectif de modifier un graphe de

communication en réunissant certains composants, de façon à en faire un arbre constitué de sous-problèmes de taille la plus petite possible. Or, dans notre cas, d'autres paramètres que la taille des composants entrent en jeu et il vaut peut-être mieux, par exemple, minimiser les interactions entre composants qu'en réunir le moins possible. Nous nous sommes donc intéressés à d'autres manières de traiter les problèmes ayant des cycles d'interactions.

Une voie qui a retenu notre attention est l'utilisation de méthodes dites turbo. Il s'agit en fait simplement d'exécuter l'algorithme 1 sur des graphes de communication contenant des cycles, en utilisant des méthodes un peu plus évoluées que la stabilité des messages pour décider de l'arrêt d'une exécution. Concrètement, celui-ci sera décidé à l'aide d'une notion de distance entre automates. Ainsi, quand la distance entre l'automate représentant un message avant mise à jour et l'automate représentant le même message après mise à jour est inférieur à une borne fixée au préalable, on peut considérer que ce message est stable. La distance que nous avons choisie est la suivante :

$$d(\mathcal{A}_1, \mathcal{A}_2) = \sum_{n=0}^{\infty} \frac{1}{2^n} 1_{\mathcal{L}_n(\mathcal{A}_1) \neq \mathcal{L}_n(\mathcal{A}_2)},$$

où $1_{\mathcal{L} \neq \mathcal{L}'}$ vaut 1 quand $\mathcal{L} \neq \mathcal{L}'$ et 0 sinon, et $\mathcal{L}_n(\mathcal{A})$ est le langage contenant les mots de longueur n de $\mathcal{A}$. Ceci revient à considérer un priorité les mots les plus courts.

On remarque aussi que, si l'on ne souhaite pas que les coûts des mots augmentent après chaque produit, il est nécessaire de pouvoir normaliser les automates. Nous avons choisi pour cela de réduire le coût de chaque chemin de l'automate à normaliser par le coût du plus court chemin dans cet automate moins 1. Ceci fixe le coût du plus court chemin à 1. L'avantage qu'a l'utilisation d'une normalisation additive par rapport à une normalisation multiplicative est qu'elle préserve l'écart entre les coûts des différents chemins. De plus une telle normalisation peut même aider à trouver les meilleurs chemins plus rapidement : certains chemins sont écartés car ils ne satisfont pas les contraintes imposées par le systèmes, mais d'autre le sont car leur coût s'écartent trop fortement de celui du chemin de plus faible coût.

Une présentation plus détaillée de cette approche ainsi qu'une étude expérimentale montrant l'intérêt de l'utilisation des algorithmes turbo pour la planification sont décrits au chapitre 4.

Arcs de lecture

On peut remarquer que, parfois, dans un problème de planification factorisée, une action se contente de lire l'état d'un composant sans le modifier. C'est-à-dire que les variables décrivant l'état de ce composant apparaissent dans la précondition de cette action sans pour autant que ses effets modifient les valeurs de ces variables. Dans l'automate représentant un composant une action ne faisant que lire dans ce composant est représentée par une auto-boucle.

Par exemple dans le problème représenté à la figure 1 les actions α, β et γ ont des effets sur les automates $\mathcal{A}_1$ et $\mathcal{A}_2$ mais ne font que lire dans l'automate $\mathcal{A}_3$. On peut remarquer que les mots de $\mathcal{A}_3$ correspondant à des solutions sont γ, $\alpha\alpha\beta$, $\alpha\beta\alpha$, et $\beta\alpha\alpha$. Il semble donc que l'automate $\mathcal{A}_3$ doive savoir par quelles actions et dans quel ordre son état est lu. Ceci ne semble pas raisonnable et peut engendrer des automates inutilement grands lors d'exécutions de l'algorithme 1 sur des problèmes impliquant des lectures.

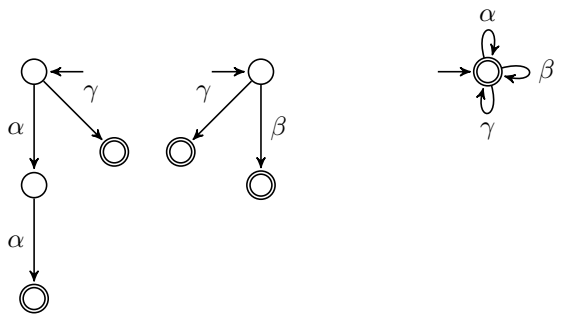


Figure 1: Un problème de planification factorisée avec trois composants, représentés par les automates: $\mathcal{A}_1, \mathcal{A}_2$ et $\mathcal{A}_3$ (de gauche à droite).

Pour éviter ce phénomène nous nous sommes inspirées de la notion d'arcs de lecture dans les réseaux de Petri afin de proposer un mécanisme de lecture d'états lors du produit d'automates. Nous avons donc modifié notre modèle d'automates afin d'y ajouter la possibilité de lire l'état de voisins (au sein d'un problème) lors de l'utilisation d'une action. Pour mettre en place ce mécanisme en utilisant uniquement les transitions nous avons aussi ajouté un mécanisme d'écriture sur celles-ci. Cette approche permet de définir un produit et une projection très proches de ceux que nous utilisions précédemment. En fait, la différence principale est dans la notion de langage d'un automate : on considère ensemble de mots cohérents, c'est à dire tels que les lectures d'une action doivent être en accord avec les écritures de la précédente.

Dans ce formalisme on remarque que la projection et le produit n'ont pas exactement les propriétés voulues (ceci étant du en particulier à la notion de langage cohérent). On peut cependant, en utilisant ce formalisme, définir une nouvelle version de l'algorithme par passage de messages (où la projection à lieu un peu moins souvent) qui permet de faire de la planification dans des réseaux d'automates avec arcs de lecture.

Ce nouveau formalisme, les produit et projection correspondants ainsi que la version adaptée de l'algorithme 1 utilisée dans ce contexte sont décrits au chapitre 5.

Planification factorisée par une version distribuée de A*

Nous avons aussi travaillé sur une version distribuée de A* pour la planification optimale factorisée, présentée dans les chapitres 6 et 7. Les différences entre cette approche et la précédente sont les suivantes :

1. il s'agit vraiment d'une approche distribuée (l'algorithme par passage de message peut être vu comme un algorithme distribué si chaque composant fait la mise à jour de ses messages sans se préoccuper des autres, mais il existe un ordre de mise à jour des messages qui assure un nombre minimal de mises à jour, ce qui tend à faire préférer une utilisation non distribuée de cette méthode),

2. elle ne nécessite pas de chercher toutes les solutions au problème considéré, au contraire, comme A*, notre algorithme (appelé A#) construit pas à pas une unique solution de coût minimal.

Afin de concevoir A# nous avons considéré des problèmes de difficulté croissante menant au problème de planification. L'étude de ces problèmes nous a permis de nous focaliser indépendamment sur chacune des difficultés induites par une version distribuée de A*.

Sommets de même couleur

Le premier problème que nous avons considéré est le suivant : on se donne deux graphes G_1 et G_2 (où $G_k = (V_k, E_k)$) et pour chacun d'eux une fonction de coloration de certains sommets : $\gamma_k : F_k \to \Gamma$ où $F_k \subseteq V_k$ et Γ est un ensemble de couleur commun à tous les graphes. L'objectif est alors, étant donné pour chaque graphe G_k un sommet initial $i_k \in V_k$ et une fonction de coût sur les arcs $c_k : E_k \to \mathbb{R}_+$, de trouver un couple (p_1, p_2) de chemins tel que $\forall k, p_k^- = i_k, p_k^+ \in F_k$, $\gamma_1(p_1^+) = \gamma_2(p_2^+)$ et tel que la somme des coûts de ces chemins soit minimale. Ce problème peut être vu comme un problème de planification particulier où l'on peut savoir en connaissant uniquement les états finaux atteints dans les différents composants si les plans locaux utilisés pour atteindre ses états sont compatibles ou non.

Nous souhaitons résoudre ce problème de manière distribuée : un agent φ_k sera chargé de trouver le chemin p_k (pour simplifier on considère qu'il n'existe qu'une solution optimale) dans le graphe G_k. Pour cela il n'utilisera que la connaissance qu'il a du graphe G_k et des informations transmises par l'autre agent sur le reste du système (on considérera que chaque agent partage une zone mémoire, dans laquelle il ne peut que lire, avec l'autre agent qui lui peut y écrire). Chaque agent va en fait exécuter sur son graphe une version légèrement modifiée de A* afin de chercher un solution locale dont on peut prouver qu'elle fait partie d'une solution globale de coût minimal.

Les deux principales différences entre l'algorithme exécuté par chaque agent et A* concernent le choix des sommets à traiter en priorité et la condition d'arrêt de l'algorithme (décision de l'optimalité d'une solution ou de l'absence de solutions). Nous en donnons ici une brève description.

Fonction d'évaluation des sommets. Dans A* on choisit, sur la base d'une connaissance évolutive du problème considéré, les sommets les plus prometteurs d'un graphe pour la recherche de plus court chemin. Cette connaissance du problème est représentée par deux fonctions : l'une indiquant le meilleur coût connu pour atteindre les sommets déjà traités au moins une fois, l'autre, appelée heuristique, étant une borne inférieur sur le coût pour atteindre un but depuis chaque sommet. C'est cette heuristique qui est construite différemment dans notre cas : elle représente une borne inférieure sur le coût pour atteindre des sommets compatibles dans les deux graphes. Elle est calculée comme un minimum sur toutes les couleurs possibles de la somme entre des heuristiques classiques (une pour chaque couleur dans chaque graphe). Par exemple pour le sommet v du graphe G_1, l'agent φ_1 considérera la valeur suivante pour cette fonction :

$$\min_{\gamma \in \Gamma}(h_1(v, \gamma), h_2(i_2, \gamma)),$$

où h_1 est une heuristique locale à G_1 et h_2 est une heuristique pour G_2.

Condition d'arrêt. Pour décider localement de la découverte d'une solution globale optimale il faut, une fois un candidat trouvé, attendre confirmation de l'autre agent que la couleur de ce candidat est en accord avec sa propre recherche. Pour cela, au lieu de

simplement considérer des sommets ouverts ou fermés, comme c'est le cas dans A*, nous utilisons trois types de sommets : ouverts, fermés (avec le même sens que pour A*) et candidats. Les sommets candidats ne peuvent être que des sommets colorés. Ils sont mis en attente par l'agent φ_k jusqu'à ce que l'autre agent puissent soit affirmer que leur couleur ne peut en aucun cas donner une solution optimale, soit indiquer le meilleur coût possible pour atteindre un sommet de cette couleur de son coté. Un sommet candidat tel que la somme du coût pour l'atteindre dans G_k et du meilleur coût pour atteindre sa couleur dans l'autre graphe est inférieure à la valeur de la fonction d'évaluation de tous les autres sommets ouverts ou candidats est alors nécessairement d'une couleur permettant de trouver une solution globale de coût minimal.

Chemins de même couleur

Le deuxième problème que nous avons considéré est le suivant : on se donne deux graphes G_1 et G_2 (où $G_k = (V_k, E_k)$) et pour chacun d'eux une fonction de coloration des arcs : $\gamma_k : E_k \to \Gamma$ où Γ est un ensemble de couleur commun à tous les graphes, ainsi qu'un ensemble de sommets $F_k \subseteq V_k$. L'objectif est alors, étant donné pour chaque graphe G_k un sommet initial $i_k \in V_k$ et une fonction de coût sur les arcs $c_k : E_k \to \mathbb{R}_+$, de trouver un couple (p_1, p_2) de chemins $p_k = e_k^1 \dots e_k^{n_k}$ tel que $\forall k, p_k^- = i_k, p_k^+ \in F_k$, $\cup\gamma_1(e_1^\ell) = \cup\gamma_2(e_2^\ell)$ et tel que la somme des coûts de ces chemins soit minimale.

Ce problème est plus proche d'un vrai problème de planification : la « couleur » d'un sommet de F_k est dynamique, elle dépend du chemin utilisé pour l'atteindre. Afin de résoudre ce type de problèmes nous réutilisons les résultats précédents en remarquant que ces problèmes peuvent être ramenés à des problèmes de recherche de sommets de même couleur. Ceci peut se faire par la transformation suivante : pour modifier G_k on considère comme ensemble de sommets $V_k \times 2^\Gamma$ et l'ensemble de couleurs 2^Γ. Le sommet initial est alors $(i_k, \emptyset)$. Les sommets à atteindre sont ceux de $F_k \times 2^\Gamma$ et tout sommet (v, Γ_v) parmi eux a pour couleur Γ_v. Enfin il y a un arc entre (v, Γ_v) et (v', Γ'_v) si et seulement si $(v, v') \in E_k$ et $\gamma_k((v, v')) = \gamma$ tel que $\Gamma'_v \setminus \Gamma_v = \{\gamma\}$. On peut donc utiliser le même algorithme que pour le problème précédent. En fait on remarque qu'il est possible d'adapter cet algorithme pour effectuer le passage d'un problème à l'autre au fur et à mesure de la recherche de solutions.

Ces deux premiers problèmes ainsi que les méthodes utilisées pour les résoudre de manière distribuée sont décrits en détails au chapitre 6 de ce document.

Planification distribuée

Finalement, on peut identifier un problème de planification factorisée impliquant deux composants à un problème de recherches de sommets de même couleur. La principale différence avec la transformation précédente étant que les graphes transformés peuvent être infinis. Cependant nous avons montré qu'il est possible de calculer ces graphes dynamiquement lors de la recherche de solutions et, lorsqu'une solution globale existe, que la recherche de solution terminera toujours en temps fini. Cette approche ne permet cependant pas de détecter l'absence de solution dans un problème.

Nous avons aussi montré qu'il est possible de calculer les heuristiques utilisées pour cette recherche à l'aide d'un nombre fini de valeurs « intéressantes », ce qui permet d'utiliser notre algorithme en pratique.

Finalement nous avons généralisé cette approche à n'importe quel problème de planification factorisée pour lequel les graphes de communication sont des arbres. Ceci

implique de transmettre des informations un peu plus complexes que précédemment entre voisins dans le graphe de communication considéré. Ces informations concernant non seulement le composant de l'agent φ_k qui les envoie à φ_ℓ mais aussi tous les composants qui sont séparés de G_ℓ par G_k.

Le chapitre 7 décrit en détail notre algorithme pour la planification factorisée avec deux composants (comme une extension des résultats du chapitre précédent) ainsi que sa généralisation à des problèmes impliquant plus de composants.

Conclusion

Pour résumer, dans ce document deux nouvelles approches de la planification factorisée sont présentées, toutes les deux permettant de trouver des solutions de coût minimal aux problèmes considérés et nécessitant une structure particulière des interactions entre composants pour fonctionner.

La première approche est basée sur la représentation sous forme d'automates à poids de l'ensemble des solutions locales à chaque composant (qui est intéressante lorsque les composants sont petits en regard du problème global). Ces ensembles de solutions sont ensuite affinés à l'aide d'un algorithme par passage de messages (traditionnellement utilisé dans le domaine de la résolution de contraintes) afin de ne garder que les solutions locales faisant partie d'une solution globale. Cette approche a été implantée et testée sur des problèmes utilisés lors de compétitions de planification. Finalement nous avons proposé deux extensions à cette approche : l'une basée sur l'utilisation de méthodes dites turbo permet, dans une certaine mesure, de traiter des problèmes pour lesquels les graphes de communication ne sont pas des arbres (sans garantir toutefois l'optimalité des solutions trouvées) et l'autre permettant de réduire la taille des automates dans les problèmes impliquant des lectures sans modification d'état dans certains composants.

La deuxième approche est basée sur un algorithme très proche de A* exécuté par un agent dans chaque composant. Chacun de ces agents utilise son information locale ainsi que des informations sur les coûts dans le reste du système (transmises pas ses voisins) afin de diriger sa recherche d'une solution de coût minimal.

Description des contributions et publications

Voici un descriptif des contributions principales de cette thèse.

Dans le chapitre 2 la contribution est un algorithme de planification factorisée permettant de trouver des plans de coût minimal. À notre connaissance aucune approche de la planification factorisée antérieure à notre travail ne permettait cela. Ces résultats ont été présentés à la conférence CDC en 2009 [29].

Dans le chapitre 3 la contribution consiste en une implantation de nos résultats (utilisant diverses techniques pour réduire la taille des automates considérés) et la comparaison de cette implémentation avec d'autres algorithmes de planification connus pour être efficaces. Ces résultats ont été présentés en partie à la conférence ICAPS 2010 [31].

Dans le chapitre 4 nos contributions sont la proposition d'utiliser les méthodes turbos dans le cadre de la planification ainsi qu'une étude expérimentale de cette idée

montrant son intérêt. Ces résultats ont été présentés au workshop WODES en 2012 [53].

Dans le chapitre 5 notre contribution est la proposition d'un nouveau formalisme pour décrire les problèmes de planification : les automates avec arcs de lecture. Ces résultats ont été présentés en partie au congrès mondial IFAC en 2011 [50].

Dans les chapitres 6 et 7 la contribution est une version distribuée de A* permettant de trouver des solutions de coût minimal de façon distribuée dans des problèmes de planification. Ces résultats ont été, en partie, présentés à la conférence CDC en 2012 [51] et une partie des preuves apparaissent dans un rapport de recherche INRIA [52].

Nous avons aussi présenté des résultats (sans rapport direct avec ce travail de thèse) sur le diagnostic probabiliste au workshop WODES en 2010 [30].

Travaux futurs

Notre travail reste améliorable, notamment sur les points suivants : la décomposition automatique de problèmes de planification (soit pour assurer que les graphes de communication soient des arbres, soit pour permettre une résolution efficace par des méthodes turbos), l'utilisation de représentations différentes des plans locaux (par exemple en utilisant des modèles prenant en compte la concurrence), la possibilité de préserver plus que les langages lors de l'utilisation de l'algorithme par passage de messages, l'étude de métriques sur les automates à poids et les réseaux d'automates à poids (afin d'avoir de nouvelles conditions pour la terminaison des algorithmes turbo, mais aussi pour pouvoir mesurer la quantité d'interactions entre deux automates au sein d'un réseau et ainsi avoir des outils pour décider du bien fondé d'utiliser les méthodes turbo sur un problème en particulier), et enfin la construction de jeux de tests spécifiques à la planification factorisée (ce qui permettrait de s'abstraire de la décomposition des problèmes pour tester l'efficacité des différents algorithmes).

Introduction

PLANNING CONSISTS IN choosing and ordering a set of state modifying actions with the objective of reaching a goal in a given state-space. This problem has been widely studied during the last 50 years and many approaches have been proposed for solving it. Current resolution methods for planning problems are mainly based on guided searches in the considered state-spaces. They traditionally rely on (or are variations of) the famous A* algorithm originally proposed by Hart, Nilsson, and Raphael in 1968. This algorithm guarantees that a solution (usually called plan) to the considered planning problem will be found as soon as there exists one. Moreover, when the actions have associated costs and under some assumptions on the functions used for driving the search, it guarantees the cost-optimality of the plan found.

Other approaches to planning exist however. They take advantage of the specific properties of some planning problems to solve them more efficiently. These methods have the same worst case complexity as A* in general. However, they have the interest of being extremely efficient on the classes of problems for which they are developed. Among these approaches the ones using independence properties between actions of a problem are of particular interest in our opinion. They are based on the remark that in many planning problems some actions can be used in any order without changing the final state they jointly reach. These actions thus do not need to be ordered in a plan. This reveals a notion of concurrency in planning. One can then represent plans as partial orders of actions rather than sequences of actions. And thus, it renders possible a representation of problems based on data-structures well suited for handling concurrency, such as Petri nets for example. This can significantly reduce the complexity of the search.

Going further in this direction it is possible to exploit the independency of some parts of a planning problem in order to split it into small subproblems (or factors or components). These components are themselves planning problems potentially exponentially smaller than the original one. If these components are sufficiently independent (in other words if their interaction is sufficiently sparse) it is possible to solve each of them taking into account minimal information about the others. The local plans obtained in each subproblem can then be merged into a global plan for the original planning problem. This kind of approach is generally referred as factored planning and has been studied for the last 10 years.

However – while being of great interest for the huge gain of efficiency they can potentially bring in solving planning problems with few interaction – current factored planning methods suffer from several weaknesses. First of all, to our knowledge, no factored planning algorithm was able to perform cost-optimal planning prior to this thesis. This is due to the fact that existing approaches were all based on parameters fixing a bound on the length of local plans to consider. Another side effect of such bounds is that it prevents detecting the absence of a global plan. The second weakness of ex-

isting approaches to factored planning is that they are not easily distributable. They are essentially centralized and hierarchical. In our opinion many planning problems are well suited to distributed solving and this approach has to be explored because of the potential efficiency gain it can bring in the resolution of these problems. The third weakness concerns the use of approximate methods. There exists approximate versions of A* (in the centralized case) that provide close-to-optimal plans instead of cost-optimal ones, in order to gain efficiency. However, approximate methods have surprisingly not appeared in factored planning. Our goal in this work was to address these weaknesses.

In order to achieve this objective we first use the fact that factored planning can be seen, in some sense, as a constraint solving problem. This is what was done in several previous contributions in factored planning. However, the novelty of our approach is that we generalize constraint solving methods to the context of planning. Previously, planning problems where simplified (by bounding lengths of plans) and then recast as constraint solving problems. More precisely we show that message passing algorithms can be used for solving planning problems. This allows us to avoid the use of bounds on the length of plans, making possible the cost-optimal planning and the detection of the absence of plan. Moreover, the sparse interaction in some factored planning problems enables the use of the famous turbo algorithms, mainly known for their use in the domain of digital communication and signal processing. This opens the way to approximate resolution methods in factored planning. We also propose a second approach to cost-optimal factored planning. The first approach is top-down: in each component a set containing all local plans is refined until it contains only the local plans giving global ones. This second approach is bottom-up: an empty set of local plans in each component is extended until a cost-optimal global plan is found. It uses a truly distributed algorithm based on A*.

Overview of the approaches developed in this thesis

Consider the factored planning problem given in Figure 2. It is represented as a network of weighted automata. Each automaton (or component) $\mathcal{A}_i$ is constituted of states (the circles) and actions (the arrows between the states and the corresponding greek letters). The upper left state of each automaton is initial and the states represented by double circles are finals. For example, in $\mathcal{A}_1$, the action β permits to go from the initial state to a final state. The cost of this move is 1. A local plan in one of these automata is a sequence of actions going from the initial state to a final state. For example $\beta\gamma\beta$ is one of the local plans of $\mathcal{A}_1$ and its cost is 3. As another example, the empty sequence of actions ε is a local plan in $\mathcal{A}_3$. The set of the local plans of each automaton $\mathcal{A}_i$ is called the language of $\mathcal{A}_i$. For example, the language of $\mathcal{A}_2$ is $(\delta\beta)^*(\delta + \alpha\alpha)$. In other words, it contains all the local plans starting by any number (including 0) of repetitions of the sequence $\delta\beta$, followed by a single action δ or one occurrence of the sequence $\alpha\alpha$. These automata also share some of their actions. For example, the action α is shared between the automata $\mathcal{A}_1$ and $\mathcal{A}_2$. These shared actions give the constraints imposed between the different components. Indeed, in a component $\mathcal{A}_i$ one will only be allowed to consider the local plans that use the shared actions between $\mathcal{A}_i$ and some other component $\mathcal{A}_j$ in the same order as in some local plan in $\mathcal{A}_j$. In our example, among the local plans of $\mathcal{A}_1$, $\alpha\omega\alpha$ can be preserved due to the existence of $\alpha\alpha$ in the language of $\mathcal{A}_2$ and ω in the language $\mathcal{A}_3$. A global plan in such a problem is a tuple (p_1, p_2, p_3) where each p_i is a local plan in the corresponding $\mathcal{A}_i$.

It must be possible to interleave these three local plans into a sequence p of actions so that the restriction of this sequence to the actions existing in $\mathcal{A}_i$ is exactly p_i. For example, the tuple $(\alpha\omega\alpha, \alpha\alpha, \omega)$ is a global plan for the particular problem of Figure 2, a corresponding interleaving is $\alpha\omega\alpha$. Its cost is 4, that is the sum of the costs of the local plans constituting it. In this problem the only other global plan is $(\alpha\alpha, \alpha\alpha, \varepsilon)$ with cost 3. An example of a local plan in $\mathcal{A}_1$ which can not be part of a global plan is $\alpha\alpha\omega\alpha$ because it is not possible to fire three α in $\mathcal{A}_2$. Less evident, $\beta\gamma\beta$ must not be preserved because it can only be associated with $\delta\beta\delta\beta\delta$ in the language of $\mathcal{A}_2$ and with $\gamma\delta$ in the language of $\mathcal{A}_3$, and they do not contain the same number of δ. More precisely, any local plan in $\mathcal{A}_1$ using k times β an no α must not be preserved because it requires using $k+1$ times δ in $\mathcal{A}_2$ and thus $k+1$ times γ in $\mathcal{A}_3$ which implies using $k+2$ times β in $\mathcal{A}_1$.

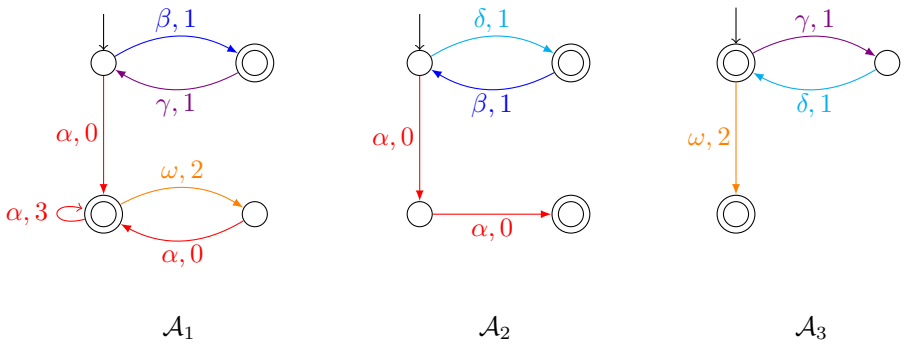


Figure 2: A factored planning problem with 3 components represented by weighted automata.

Top-down resolution. In the case of Figure 2 we give an overview of what should be the behavior of an ideal factored cost-optimal planner based on message passing and using turbo methods.

The principle of message passing is that each component sends information about the constraints it imposes to the other components that share actions with it. The message sent from $\mathcal{A}_i$ to $\mathcal{A}_j$ states which sequences of shared actions of $\mathcal{A}_j$ can fit with local plans of $\mathcal{A}_i$. It also contains information on the relative costs that can be achieved for each of these sequences of shared actions. For example in the case of Figure 2 component $\mathcal{A}_2$ will send a message $\mathcal{M}_{2,1}$ to $\mathcal{A}_1$. This message will in fact be the language $\beta^* + \beta^*\alpha\alpha$, that is the set of acceptable synchronization sequences from the point of view of $\mathcal{A}_1$. The message will also state that the best possible cost of any of this synchronization sequence p is $2.|p|_\beta$ (where $|p|_\beta$ is the number of occurrences of β in p). Using all the messages it receives each component can be updated by removing its local plans that do not match the received constraints. In each component the costs of the preserved local plans can also be updated by summing them with the costs of the constraints they match.

In our example, the language of $\mathcal{A}_1$ is initially $\mathcal{L}_1 = (\beta\gamma)^*(\beta + \alpha(\alpha + \omega\alpha)^*)$ and the cost of a local plan p in this language is $|p|_\beta + |p|_\gamma + 3.(|p|_\alpha - 1 - |p|_\omega) + 2.|p|_\omega$. Taking into account the constraints from $\mathcal{A}_2$ this language can be restricted to $\mathcal{L}'_1 = (\beta\gamma)^*(\beta + \alpha(\alpha + \omega\alpha))$ because no more than two α are possible in a local plan from the language of $\mathcal{A}_2$. The cost of a local plan p in this new language $\mathcal{L}'_1$

is $3.|p|_\beta + |p|_\gamma + 3.(|p|_\alpha - 1 - |p|_\omega) + 2.|p|_\omega$ taking into account the costs in $\mathcal{A}_2$ which increases the cost of each β by 2. The constraints from $\mathcal{A}_3$ do not change this language but only the costs of its local plans. The new cost of a plan p will be $3.|p|_\beta + 3.|p|_\gamma + 3.(|p|_\alpha - 1 - |p|_\omega) + 4.|p|_\omega$. Similarly the language of $\mathcal{A}_2$ will be updated from $\mathcal{L}_2 = (\delta\beta)^*(\delta + \alpha\alpha)$ to $\mathcal{L}'_2 = (\delta\beta)^*(\delta\beta\delta + \alpha\alpha)$. In this language the cost of a local plan p is $3.|p|_\beta + 3.|p|_\delta + |p|_\alpha$. And the language of $\mathcal{A}_3$ is updated from $\mathcal{L}_3 = (\gamma\delta)^* + (\gamma\delta)^*\omega$ to $\mathcal{L}'_3 = (\gamma\delta)^*(\gamma\delta + \omega + \varepsilon)$. The cost of a local plan p in this language is $3.|p|_\gamma + 3.|p|_\delta + 4.|p|_\omega$. It is then possible to update $\mathcal{L}'_1$ from $\mathcal{L}'_2$ and $\mathcal{L}'_3$ (and similarly update $\mathcal{L}'_2$ and $\mathcal{L}'_3$). This will lead to the new languages $\mathcal{L}''_1 = (\beta\gamma)^*(\beta\gamma\beta+\alpha(\alpha+\omega\alpha)^*)$, $\mathcal{L}''_2 = (\delta\beta)^*(\delta\beta\delta\beta\delta+\alpha\alpha)$, and $\mathcal{L}''_3 = (\gamma\delta)^*(\gamma\delta\gamma\delta+\omega+\varepsilon)$.

When updating languages one can deduce that some local plans can not be part of a global plan: the local plans belonging to a set of plans for which the smallest element is larger after each update, and the local plans p for which there exists p' such that the difference between the cost of p and the cost of p' grows after each update. For example, one will notice that in $\mathcal{A}_1$ no local plan without α can be part of a global plan: the minimal length of such a local plan grows after each update (initially the smallest one was β, then it was $\beta\gamma\beta$, after that it would be $\beta\gamma\beta\gamma\beta$, and so on). Moreover among the local plans containing α only those starting by α will be acceptable because for any local plans $w_1 = u$ and $w_2 = (\beta\gamma)^k u$ with $k \geq 1$ and $u \in \alpha(\alpha + \omega\alpha)$ the difference between the costs of w_1 and w_2 is the cost of $(\beta\gamma)^k$, which grows after each update.

From that one will be able to deduce that among the local plans of $\mathcal{A}_1$ only $\alpha\alpha$ and $\alpha\omega\alpha$ can reasonably be part of a global plan. Similarly, in $\mathcal{A}_2$ only $\alpha\alpha$ will be kept while in $\mathcal{A}_3$ the candidates will be ω and ε. Then, as the cost difference between $\alpha\alpha$ and $\alpha\omega\alpha$ in $\mathcal{A}_1$ grows after each update only $\alpha\alpha$ will be kept. Similarly, ε will be the only remaining local plan in $\mathcal{A}_3$. This will give as global plan $(\alpha\alpha, \alpha\alpha, \varepsilon)$, which is the optimal global plan as stated above.

In order to generalize these simple ideas, several questions have to be answered. How to formalize this approach? Which problems can it solve? Is it of interest in practice? Can one derive approximate methods yielding suboptimal plans with costs close to optimal?

Bottom-up resolution. Still relying on Figure 2 we give an example of how a distributed A* algorithm should behave.

The idea is to consider an agent φ_i for each automaton $\mathcal{A}_i$. Each agent will run an extended version of A* on her automaton. The search she performs being driven by both information local to her automaton and information coming from the other agents. For example, in the problem of Figure 2, agent φ_1 could initially consider the local plans α and β. Using information obtained from agent φ_2 she will remark that α is not a good candidate because no local plan with a single α exists in $\mathcal{A}_2$. Her information on local costs combined with information on costs in the rest of the system can let her conclude that a more interesting plan than β potentially exists. Indeed, a global plan involving a β would cost at least 4: a local cost of 1 and a cost of 3 from φ_2 (any local plan involving β and no α costs at least 3). And a global plan involving two α may exists and cost less than 4: a local cost of at least 2, a cost of at least 0 from φ_2 (the local plan $\alpha\alpha$ costs 0 in $\mathcal{A}_2$), and a cost of at least 0 from φ_3 (the local plan ϵ costs 0 in $\mathcal{A}_3$). Agent φ_1 will thus consider the local plans $\alpha\alpha$ and $\alpha\omega\alpha$. Local information will give her a cost of 3 for $\alpha\alpha$ and a cost of 2 for $\alpha\omega\alpha$. Information from φ_2 will consist in a cost of 0 in both cases. Information from φ_3 will give a cost of 0 for $\alpha\alpha$ and a cost of 2 for $\alpha\omega\alpha$. Using this agent φ_1 will conclude that $\alpha\omega\alpha$ is the

only reasonable local plan, and thus that it must be part of a global plan. With similar reasonings the other agents will find their respective parts of the cost-optimal global plan.

In this case as well several questions have to be answered. What should be the information transmitted between agents? Is it always computable? How can one decide that a cost-optimal global plan has been found?

Structure of this document

Chapter 1 is an overview of the evolution of planning methods from best-first search algorithms such as A* to methods taking advantage of intrinsic concurrency of planning problems and then factored planning. The objective of this chapter is to justify the interest of factored approaches to planning but also to show the weaknesses of current factored planners.

The next two chapters present the basis of a first approach to factored planning. The reader may at first glance skip the proofs while reading these chapters. These results have been published in [29] and [31].

Chapter 2 first presents the formalism we use for representing factored planning problems, that is networks of weighted automata. Then message passing algorithms are presented and it is shown how and when they can be plugged into this formalism for providing cost-optimal solutions to factored planning problems by the mean of local computations only (i.e. computations involving only a component and its neighbors).

Chapter 3 explains how it is possible to reduce the size of the objects involved in message passing algorithms using standard weighted automata algorithms. Indeed, the results of previous chapter, even if based on local computations only, involve weighted automata of important size with possibly useless information. Then this chapter presents a concrete implementation we made of this approach, as well as experimental results obtained with this factored planner.

The next two chapters describe two independent extensions of the planning method described in Chapters 2 and 3. Parts of these results have been published in [53] and [50] respectively.

Chapter 4 mainly presents an experimental study of turbo algorithms for solving factored planning problems. It reveals that these approximate methods are of interest for planning, in particular in cases where the results of previous chapters are not supposed to be applicable (that is in cases where the interaction between components has a complex structure).

Chapter 5 presents a slightly different formalism for factored planning problems than network of automata. This formalism is proposed for more efficiently dealing with factored planning problems where some actions of a component only read the variables of another component without modifying them.

Finally, the last two chapters are almost independent from the previous ones as they present a different approach to factored planning. These results were partly published in [51] and some of the proofs appear in [52].

Chapter 6 describes two simple distributed problems related to planning and explains how a distributed version of A* allows one to solve them. This chapter is a first step towards a truly distributed algorithm for finding cost-optimal solutions to factored planning problems.

Chapter 7 presents A#, a distributed version of A* for factored planning. It is done by presenting factored planning as a generalization of the problems of Chapter 6. This algorithm allows one to solve cost-optimal factored planning problems in a distributed manner: in each component an agent progressively builds a local plan using minimal information sent by the other agents on their own progress.

Chapter 1

From Planning to Factored Planning

chapter abstract: *This chapter presents the formalism of planning problems considered in this thesis as well as various approaches currently used for solving planning problems. These approaches are of three types: centralized approaches based on "best first" search, centralized approaches exploiting intrinsic concurrency of problems, and modular (or factored) approaches. We also give some insight about the theoretical complexity of planning.*

PLANNING CONSISTS IN finding a sequence of actions driving a system from an initial state to a final one (Section 1.1). It amounts to finding a path in a generally huge graph. Planning problems are traditionally solved using centralized approaches where this search is driven by heuristic functions indicating which search directions are the most promising. These approaches rely on the famous "best first" search A* algorithm [37] where heuristics are (hopefully tight) lower bounds on cost of paths reaching the goal from any state. The main challenge when using the A* algorithm is in the design of the heuristic function. It has to be accurate, but also must not be too expensive to compute. Indeed, to make it of interest in practice the cost of computing a heuristic and doing a search with it should be smaller than the cost of doing the search without this heuristic. This approach of planning is presented in Section 1.2, and the notion of heuristic is discussed in more details in Section 1.3.

Recently, new approaches were proposed for planning, taking profit of the intrinsic concurrency of planning problems. These approaches propose to exploit the fact that some, and sometimes many, actions of a planning problem are independent in order to represent solutions by partial orders of actions rather than sequences of actions. This representation of solutions reduces the state-space to explore for solving a problem. A first and famous planner using this kind of approach is Graphplan [8]. The notion of independence of actions and its exploitation are the topics of Section 1.4.

Finally, in the last 10 years the notion of factored planning appeared [1]. The idea is to use a decomposition of a planning problem into smaller planning problems with sparse interaction for solving it by parts. The subproblems can be exponentially smaller than the original problem, which is the main reason making factored planning of interest. Planning problems are usually decomposed by separating their state variables

(then subproblems interact by shared actions) or by separating their actions (then subproblems interact by shared variables). Factored planning is presented in Section 1.5.

In practice some approach can be much more efficient than another one on some particular problem, nevertheless these three approaches have the same worst-case complexity. The complexity of planning is discussed more precisely in Section 1.6

1.1 Planning problems

In the domain of artificial intelligence planning refers to a *state transformation* problem. It consists of a set of states with the objective of finding a sequence of state transformations allowing to change an *initial* state into a *goal* state. This has been described, for example, in [36]. Currently, a model of such problems widely used in the planning community is propositional STRIPS (other models exist however, such as SAS and SAS$^+$ [3]). This model was originally introduced in [32] as the representation of planning problems for the STanford Research Institute Problem Solver. In this section we present a modern formalism of propositional STRIPS.

1.1.1 Formalism of planning problems

Definition 1.1. *A planning problem is a tuple* $\mathcal{P} = (A, O, I, G)$ *where* A *is a finite set of* atoms *(or* variables*),* $O \subseteq 2^A \times 2^A \times 2^A$ *is a set of* operators*,* $I \subseteq A$ *is an* initial state*, and* $G \subseteq A$ *defines a set of* goal states*.*

The atoms are the state variables of the problem: a state is given by assigning a truth value to each atom of A. One usually represents a state by the subset of atoms that take value *true*. The initial state is a particular state: $I \subseteq A$ giving the initial truth value of all atoms. $G \subseteq A$ defines a set of states: all the states such that the truth value of each atom from G is true (in other words, all states which contain G).

Any operator $o \in O \subseteq 2^A \times 2^A \times 2^A$ is a triple $o = (\mathbf{pre}_o, \mathbf{del}_o, \mathbf{add}_o)$. We call $\mathbf{pre}_o \subseteq A$ its *precondition*, $\mathbf{del}_o \subseteq A$ its *negative effect*, and $\mathbf{add}_o \subseteq A$ its *positive effect*. An operator o is always such that $\mathbf{del}_o \cap \mathbf{add}_o = \emptyset$. Usually, an operator is instantiated as several *actions*, one for each possible firing of this operator as described below. This is however a technical detail. We thus do not distinguish operators and actions in this document.

The semantic of operators is as follows. From any state $s \subseteq A$ the operator $o = (\mathbf{pre}_o, \mathbf{del}_o, \mathbf{add}_o)$ can be *fired* if and only if $\mathbf{pre}_o \subseteq s$. In this case the system changes its state to $s]o = s \setminus \mathbf{del}_o \cup \mathbf{add}_o$. A sequence of operators $o_1 \dots o_n$ is said to be *firable* from a state s if and only if, for any of these operators o_i, one has o_i is firable from $s]o_1] \dots]o_{i-1}$. In this case, the (unique) state $s]o_1] \dots]o_n$, is called the state *reached* by $o_1 \dots o_n$ from s.

A solution to a planning problem $\mathcal{P} = (A, O, I, G)$, generally called a *plan*, is a sequence p of actions (or operators) which is firable from I and such that the state reached by this sequence from I belongs to the set of states defined by G.

1.1.2 An example

As an example we model the famous Tower of Hanoi problem (with 3 rods and 3 disks) as a planning problem. This problem is as follows. Three disks of different diameters (a small one $d1$, a medium one $d2$, and a large one $d3$) are initially stacked on a rod

$r1$ in ascending order of diameter. One has to transfer all these disks to another rod $r2$ using a third rod $r3$ and obeying to the following simple rules defining the possible moves of the disks from rod to rod:

1. a disk can never be placed on top of a smaller disk,
2. only the upper disk on a rod can be moved.

In order to model this we use the 9 following atoms: `(at-r1-d1)`, `(at-r2-d1)`, `(at-r3-d1)`, `(at-r1-d2)`, `(at-r2-d2)`, `(at-r3-d2)`, `(at-r1-d3)`, `(at-r2-d3)`, and `(at-r3-d3)`, where `(at-rx-dy)` means that disk dy is on rod rx. Notice that these atoms only define the rod on which each disk is. They do not describe in which order the disks are on a given rod. This is however sufficient to fully describe the *valid* states of this problem: for a set of disks and a rod only one ordering of these disks on the rod is possible. The actions considered will be used to ensure that only valid states are reachable: an action will never allow to place a disk on a smaller one.

With this representation of the problem the initial state would be defined as the set of atoms $I = \{$`(at-r1-d1)`, `(at-r1-d2)`, `(at-r1-d3)`$\}$ and the set of goal states by the set of atoms $G = \{$`(at-r2-d1)`, `(at-r2-d2)`, `(at-r2-d3)`$\}$. Notice that, in fact the set G will define only one reachable state (as a given disk can not be on several rods in a given state).

It is then possible to define the operators for moving the disks (that is for changing the state of the system). For disk $d1$ one will have the operators

$$(\{\texttt{(at-rx-d1)}\}, \{\texttt{(at-rx-d1)}\}, \{\texttt{(at-ry-d1)}\})$$

for all `rx` $\neq$ `ry` (this gives 6 operators). Such an operator moves disk `d1` from rod `rx` to a different rod `ry`. For disk $d2$ the operators will be

$$(\{\texttt{(at-rx-d2)}, \texttt{(at-ry-d1)}\}, \{\texttt{(at-rx-d2)}\}, \{\texttt{(at-rz-d2)}\})$$

for all `rx` $\neq$ `ry` $\neq$ `rz` (this gives 6 operators). It corresponds to a move of disk `d2` from rod `rx` to a different rod `rz` with the requirement that disk `d1` is on a third rod `ry` (that is, not above `d2` before the move and not under it after the move). Finally, for disk $d3$ the operators will be

$$(\{\texttt{(at-rx-d3)}, \texttt{(at-ry-d1)}, \texttt{(at-ry-d2)}\}, \{\texttt{(at-rx-d3)}\}, \{\texttt{(at-rz-d3)}\})$$

for all `rx` $\neq$ `ry` $\neq$ `rz` (this gives 6 operators as well). It allows to move `d3` from `rx` to `rz` as soon as the two other disks do not make this move illegal.

This modeling is only an example and many others are possible in propositional STRIPS (for example with more complex atoms and simpler operators) and in other formalisms such as SAS and SAS+ which allow multi-valued variables.

1.1.3 Cost-optimal planning

In general one is not only interested in finding a plan for a given planning problem but rather in finding the best possible plan solving this problem. This notion of best plan has to be formally defined. Usually one would have some resource consumption to minimize. A standard way to model that is to consider that firing an action uses some resource, or incurs some cost.

In this case, a planning problem $\mathcal{P} = (A, O, I, G)$ will be provided with a cost function $c : O \to \mathbb{R}_+$, associating to each operator o a cost $c(o)$. From that the

cost function can be extended to plans, associating a cost to any (firable) sequence of operators $o_1 \dots o_n$ in the following way: $c(o_1 \dots o_n) = c(o_1) + \dots + c(o_n)$. The objective is then to find a minimal cost plan (that is a plan p such that no plan p' exists with $c(p') < c(p)$).

1.2 The A* algorithm

The A* algorithm has been introduced in [37, 38]. It is an algorithm for cost-optimal planning performing a search in the state-space of a planning problem. Its principle is to drive the search by using a heuristic giving for each state s a lower bound on the cost of reaching a goal state from s (that is a lower bound on the cost of the sequences of operators firable from s that reach a goal state). As this algorithm is the basis for the results of Chapters 6 and 7, in the following we give a full presentation of A* and intuitions on the proof of its validity. Then we briefly describe some heuristics.

1.2.1 Planning problems as search in a graph

For simplicity, we consider a representation of planning problems as reachability problems on graphs. In practice however, one will prefer using A* directly on planning problems (or equivalently build the graph online) rather than re-encode them as graphs.

Definition 1.2. *A reachability problem on a graph is a tuple $\mathcal{P} = (V, E, \Lambda, \lambda, c, i, F)$ where the finite set of vertices V and the finite set of directed edges $E \subseteq V \times V$ define a (finite) directed graph $\mathcal{G} = (V, E)$, Λ is a set of labels, $\lambda : E \to \Lambda$ is a labeling function for edges, $c : E \to \mathbb{R}_+$ is a cost function associating a cost to edges, $i \in V$ is an initial vertex, and $F \subseteq V$ is a set of final vertices.*

A *path* in $\mathcal{G}$ is a sequence of edges $p = e_1 \dots e_n$ such that, for any $1 \leq i < n$, one has $e_i = (v_i, v_{i+1})$. p is said to be a path from $v_1 = p^-$ to $v_{n+1} = p^+$. The labeling and the cost function extend to paths $p = e_1 \dots e_n$ by $\lambda(p) = \lambda(e_1) \dots \lambda(e_n) \in \Lambda^*$ and $c(p) = c(e_1) + \dots + c(e_n)$. The labeling is said to be *deterministic* if and only if for every pair of edges (v, v') and (v, v''), $\lambda(v, v') = \lambda(v, v'')$ entails $v' = v''$. In this case one can extend the cost function to sequences of labels $w = w_1 \dots w_n$ in the following way: $c(w) = c(p)$ for p the unique path such that $\lambda(p) = w$ (if no such path exists then $c(w) = +\infty$). The objective is to find a path p such that $p^- = i,\ p^+ \in F$ and $c(p)$ is minimal among such paths.

Any planning problem $\mathcal{P} = (A, O, I, G)$ with cost function c can be represented as a reachability problem on graph $\mathcal{P} = (V, E, \Lambda, \lambda, c', i, F)$ as follows:

- $V = 2^A$ (vertices thus correspond to states of the planning problem),
- $E = \{(v, v') \ : \ \exists o \in O, \mathbf{pre}_o \subseteq v \wedge v]o = v'\}$ (edges correspond to actions),
- $\Lambda = O$,
- λ is such that $\lambda((v, v')) = o$ with o an operator such that $\mathbf{pre}_o \subseteq v$ and $v]o = v'$ (without loss of generality we suppose unicity of this o),
- c' is such that $c'(e) = c(\lambda(e))$,
- $i = I$, and
- $F = \{v \ : \ v \in 2^A, v \supseteq G\}$

Notice that any reachability problem obtained in this way always has a deterministic labeling because any operator o in any planning problem is by definition such that for any state s from which it can be fired $s]o$ is unique. So, one can identify paths in the graph with plans. Notice also that for a given problem some vertices and edges may not be of interest (because they can never be accessed from the initial vertex) and can thus be removed from the graph. In the following we consider that these vertices and edges were removed. We recall however that, in practice, the A* algorithm will be run directly on planning problems, without building the full corresponding graphs.

1.2.2 Presentation of A*

The idea of A* is to run a search algorithm on a problem $\mathcal{P} = (V, E, \Lambda, \lambda, c, i, F)$, using a time-varying *evaluation function* $f : V \to \mathbb{R}_+ \cup \{+\infty\}$ which allows one to decide at any moment of the execution of the algorithm which vertex should be expanded next. This is what Algorithm 2 does (expand function is computed by Algorithm 3). In this algorithm two types of vertices are considered: $open$ ones and $closed$ ones. Initially all vertices v are $closed$ and such that $f(v) = +\infty$.

Algorithm 2 A search algorithm

```
1:  mark i open
2:  calculate f(i)
3:  while there exists open vertices do
4:     let v be the open vertex with minimal f(v)
5:     if v ∈ F then
6:        return v and terminate
7:     else
8:        expand(v)
9:     end if
10: end while
```

Algorithm 3 A generic expand function

```
1: mark v closed
2: for all v' such that (v, v') ∈ E do
3:    calculate f(v')
4:    if f(v') strictly decreased then
5:       mark v' open
6:       pred(v') ← v
7:    end if
8: end for
```

The $pred$ relation built along execution of the algorithm allows one, after termination (by fulfilling the condition at line 5 of Algorithm 2), to build a path p from i to the final vertex v obtained. This path is defined as follows: $p = e_1 \dots e_n$ with $p^- = i$ and $e_k = (pred^{n-k+1}(v), pred^{n-k}(v))$ where computing $pred^k(v)$ consists in applying k times $pred$ to v.

The interest of A* is to propose a concrete manner of computing the evaluation function f. The idea is to use two functions: g and h with some properties such that taking $f(v) = g(v) + h(v)$ will ensure the search algorithm to find cost-optimal paths.

The function g will in fact be such that $g(v)$ is the best cost known (when this function is used) for reaching v from i. And the function h (usually called the *heuristic function*) will be such that $h(v)$ is a lower bound on the optimal cost over the paths reaching a state in F from v.

The proof of validity of A* is mainly based on the following lemma.

Lemma 1.1 (Lemma 1 of [37]). *For any vertex v such that v is open or has not yet been opened (i.e. $g(v) = +\infty$), and for any cost-optimal path p from i to v, there exists an open vertex v' belonging to p such that $g(v')$ is the optimal cost from i to v'.*

A corollary of this lemma is the following:

Lemma 1.2 (Corollary of [37]). *For any cost-optimal path p from i to some vertex $v \in F$, as long as Algorithm 2 has not terminated there exists an open vertex v' belonging to p such that $f(v') \leq c(p)$.*

To prove the validity of Algorithm 2 one then notices that if it terminates by returning a vertex v, then $v \in F$ (see Line 5) and the path obtained from the $pred$ relation is a cost-optimal path from i to v (this comes from Lemma 1.1 and Lemma 1.2).

The proof of termination of A* is due to two facts. The number of vertices in the considered graph is finite. And for any real value c the number of different real values $c' < c$ such that c' is the cost of a path in a given graph is finite, which implies that any vertex can only be opened a finite number of times.

It is possible to use a slightly different expand function for running Algorithm 2. This function (Algorithm 4) has two interests. First, it highlights a possible computation of g. And then, it allows one to safely use heuristics h for which $h(v)$ can be modified along the search. The interest is to be able to refine h along an execution of A* using the knowledge obtained on the explored part of the graph.

Algorithm 4 An expand function specific to A*

1: mark v closed
2: **for all** v' such that $(v, v') \in E$ **do**
3: $\quad g(v') \leftarrow \min(g(v'), g(v) + c((v, v')))$
4: $\quad$ **if** $g(v')$ strictly decreased **then**
5: $\quad\quad$ mark v' open
6: $\quad\quad$ $pred(v') \leftarrow v$
7: $\quad\quad$ calculate $f(v') = g(v') + h(v')$
8: $\quad$ **end if**
9: **end for**

Other algorithms also exist for heuristic search based planning, most of them being variations around A*. For example let us mention IDA* (which stores less information than A* thanks to the use of iterative deepening – a search method combining the advantages of both depth-first search and breadth-first search – see [66]), RTA* and LRTA* (which trade guarantees on optimality for efficiency of the search, see [67]), or WA*(which associates a weight – that is a multiplicative factor – to heuristics and guarantees optimality up to a constant only, see Chapter 3 of [79]). Moreover, search is not always done from initial state to goal states but sometimes from goal to initial states (it allows easier computations of particular heuristics), this is called regression planning (see Chapter 1 in [40]).

1.3 About heuristics

In planning a heuristic is an estimate of the cost of a path. Heuristics are usually requested to be *admissible*, that is to give lower bounds on costs of paths. A typical example is distance in a beeline which is an admissible heuristic for searching shortest paths between two points in a street network.

When using the A* algorithm, or any other heuristic based search algorithm, for planning the main difficulty is to find a "good" admissible heuristic. Such a heuristic must have two main characteristics:

1. be accurate, that is provide a lower bound as tight as possible,
2. be cheap to compute, the time needed for computing h and solving the problem using it should be smaller than the time needed for solving the problem without using a heuristic.

Remark that a heuristic always exists: one can take h equal to 0 in every point. This is however, in general, not very accurate as in this case A* corresponds in some sense to a breadth-first search.

Notice that it is possible in practice to use heuristics that are not admissible. It still guarantees termination of A*, and finding of a solution when it exists, but no longer guarantees the cost-optimality of the paths found. This is however of interest as it may be much easier to compute a good heuristic with no guarantee of admissibility than a good admissible heuristic, while still allowing to find solutions which are almost cost-optimal (see for example h_{add} in [10]). In this section we focus on admissible heuristics as we are mostly interested in cost-optimal planning.

Originally, heuristics were *domain specific*: a given heuristic was defined for a given (type of) planning problem (as for example Sokoban [54], Rubik's Cube [68] or 15-Puzzle [20]). This allows one to develop very efficient heuristics. However, it enforces the development of a new heuristic for each new problem. Nowadays the objective is rather to develop more general heuristics which address any problem in a given formalism (e.g. STRIPS). It is hopeless to reach the level of efficiency of domain specific heuristics as for a given heuristic h it is always possible to build a problem such that, by using h, A* will fall in its worst case and explore all the states before finding a plan. But general heuristics still give promising results.

According to [45] current heuristics are mainly based on four ideas: delete relaxation, critical path, abstraction, and landmark.

1.3.1 Delete relaxation heuristics

The principle of these heuristics is, for a given planning problem $\mathcal{P} = (A, O, I, G)$, to consider a relaxed problem $\mathcal{P}^+ = (A, O^+, I, G)$ where O^+ is obtained from O by removing the negative effects of each operator: $O^+ = \{(\mathbf{pre}_o, \emptyset, \mathbf{add}_o) \ : \ o \in O\}$. For any state $s \subseteq 2^A$ the cost of reaching G in $\mathcal{P}^+$ from s gives a lower bound on the cost of reaching G in $\mathcal{P}$ from s. This gives an accurate heuristic but which is difficult to construct (deciding the existence of a solution in $\mathcal{P}^+$ is NP-complete, as stated by Lemma 1.3 in Section 1.6).

In the example of Section 1.1.2 using this approach would give operators which never remove disks from rods. For example the operators moving disk `d1` will become the $(\{$`(at-rx-d1)`$\}, \emptyset, \{$`(at-ry-d1)`$\})$ for all `rx` $\neq$ `ry`. One can remark that any sequence of operators moving disk `d1` more than two times is never of interest in this

context: it would always be such that at some time an atom already appearing in the current state is added. Table 1.1 presents the value of the heuristic obtained (considering that each operator has cost 1) from some particular states.

s	$h(s)$	$opt(s)$	h-plan	opt-plan
I	5	7	$\tilde{o}_1^{1,2}\tilde{o}_1^{1,3}\tilde{o}_2^{1,2}\tilde{o}_2^{1,3}\tilde{o}_3^{1,2}$	$o_1^{1,2}o_2^{1,3}o_1^{2,3}o_3^{1,2}o_1^{3,1}o_2^{3,2}o_1^{1,2}$
$I]o_1^{1,2}$	4	6	$\tilde{o}_1^{2,3}\tilde{o}_2^{1,2}\tilde{o}_2^{1,3}\tilde{o}_3^{1,2}$	$o_2^{1,3}o_1^{2,3}o_3^{1,2}o_1^{3,1}o_2^{3,2}o_1^{1,2}$
$I]o_1^{1,2}]o_2^{1,3}$	4	5	$\tilde{o}_1^{2,3}\tilde{o}_1^{2,1}\tilde{o}_2^{3,2}\tilde{o}_3^{1,2}$	$o_1^{2,3}o_3^{1,2}o_1^{3,1}o_2^{3,2}o_1^{1,2}$

Table 1.1: Delete-relaxation heuristic for the Tower of Hanoi. Value $h(s)$ of this heuristic from state s as well as value $opt(s)$ of an optimal plan from s in the original problem are depicted. In each case h-plan is an optimal plan in the relaxed problem used for computing h while opt-plan is an optimal plan in the original problem. $o_k^{i,j}$ is the only operator moving disk k from rod i to rod j and $\tilde{o}_k^{i,j}$ is its relaxed counterpart.

Due to the complexity of computing these delete-relaxation heuristics in general, one will prefer to use estimates of them. An example is the h_{max} heuristic from [10]. The idea for building $h_{max}(s)$ is to compute for each atom $a \in G$ an estimate $g_s(a)$ of the cost for achieving a from s and combine these estimates. This g_s can be computed by a simple procedure. First $g_s(a)$ is initialized to 0 for all $a \in s$ and to $+\infty$ for all $a \notin s$. Then an operator o firable from s is selected and for each $a \in \mathbf{add}_o$ the value $g_s(a)$ is updated to $\min(g_s(a), c(o) + g_s(\mathbf{pre}_o))$ and a is added to s. This is repeated until no operator selection can change the values of g_s. h_{max} is then defined as follows: $h_{max}(s) = g_s(G)$. The only remaining part is to define the meaning of $g_s(S)$ for some set S of atoms. One could consider that $g_s(S) = \sum_{a \in S} g_s(a)$. This would however result in a heuristic which is in general not admissible (the heuristic obtained in this case is the h_{add} cited previously) because some atoms can be achieved as a side effect of achieving a particular one. It is thus suggested in [10] to use $g_s(S) = \max_{a \in S} g_s(a)$ for building an admissible heuristic.

1.3.2 Critical path heuristics

Critical path heuristics are a family of admissible heuristics denoted by h^m ($m \in \mathbb{N}^*$). They were introduced in [42] and are presented in details in Chapter 3 of [40] for regression planning. These heuristics are such that $h^m(s) \geq h^{m-1}(s)$ for all s. In other words h^m is more accurate than h^{m-1}. As the complexity of computing h^m is exponential in m (but polynomial for a fixed m) there is a tradeoff between accuracy of h^m and efficiency of its computation. Moreover, one has that h^1 corresponds to h_{max} and h^2 corresponds to the lower bound function computed by GRAPHPLAN using the notion of planning graph (a brief description of GRAPHPLAN principles is given in Section 1.4.1). h^m is thus in some sense a generalization of previous planning approaches.

The idea for computing h^m is that instead of taking as estimate the maximum cost over all atoms of achieving one of them (as it was the case for h_{max}), $h^m(s)$ will be the maximum cost over all subsets of m atoms of achieving one of these subsets. This can be computed for a given m using similar methods as for h_{max}.

1.3.3 Abstraction heuristics

The principle for building abstraction heuristics is to map each state s of a planning problem $\mathcal{P} = (A, O, I, G)$ to an *abstract state* $\alpha(s)$. One can then use as estimate of the cost from s to G the cost from $\alpha(s)$ to an abstract state in $\alpha(G)$ (where $\alpha(G) = \{\alpha(s) \ : \ s \supseteq G\}$) in the transition system induced by α. Each possible abstraction leads to a heuristic.

For example in the case of the Tower of Hanoi (Section 1.1.2) one could decide to abstract the position of disk `d1`. It corresponds to considering that for any given state s, all states s' reachable from s using operators modifying the position of `d1` only are such that $\alpha(s') = \alpha(s)$. Table 1.2 presents the value of the heuristic obtained (considering that each operator has cost 1) from some particular states.

s	$h(s)$	$opt(s)$	h-plan	opt-plan
I	3	7	$\tilde{o}_2^{1,3}\tilde{o}_3^{1,2}\tilde{o}_2^{3,2}$	$o_1^{1,2}o_2^{1,3}o_1^{2,3}o_3^{1,2}o_1^{3,1}o_2^{3,2}o_1^{1,2}$
$I]o_1^{1,2}$	3	6	$\tilde{o}_2^{1,3}\tilde{o}_3^{1,2}\tilde{o}_2^{3,2}$	$o_2^{1,3}o_1^{2,3}o_3^{1,2}o_1^{3,1}o_2^{3,2}o_1^{1,2}$
$I]o_1^{1,2}]o_2^{1,3}$	2	5	$\tilde{o}_3^{1,2}\tilde{o}_2^{3,2}$	$o_1^{2,3}o_3^{1,2}o_1^{3,1}o_2^{3,2}o_1^{1,2}$

Table 1.2: An abstraction heuristic for the Tower of Hanoi problem. Value $h(s)$ of this heuristic from state s as well as value $opt(s)$ of an optimal plan from s in the original problem are depicted. In each case h-plan is an optimal plan in the abstract problem used for computing h while opt-plan is an optimal plan in the original problem. $o_k^{i,j}$ is the only operator moving disk k from rod i to rod j and $\tilde{o}_k^{i,j}$ is its counterpart in the abstract problem.

Many abstractions have been considered for building heuristics such as:

- merge and shrink abstractions [46]: starting from a collection of abstractions (which are actually projections on each variable) a single abstraction is built using two operations: merging (which builds a single abstraction from two abstractions using a product) and shrinking (which abstracts more), or
- pattern databases based abstractions: pattern databases are an efficient data-structure for storing and accessing costs of cost optimal paths into an abstraction of a planning problem. They were introduced in [20] for solving 15-puzzle problem (as well as several other problems in various papers), and then applied to the building of domain independent heuristics [24].

1.3.4 Landmark heuristics

A landmark in a planning problem is a propositional formula over truth values of atoms that is true at some time in any plan. This notion (and the notion of ordering of landmarks) can be used to build heuristics. It has been suggested first in [80] were the number of landmarks yet to achieve from a given state s is used as a heuristic. This heuristic depends on the path used to reach s and thus can in no way be admissible. In [55] however admissible heuristics based on landmarks have been proposed. As an example, in the Tower of Hanoi problem of Section 1.1.2 the fact that, at some time, the disk `d3` needs to be alone on rod `r2` can be used as a landmark ℓm_1. Another example of possible landmark ℓm_2 is the fact that at some time disks `d2` and `d3` need to be the

only two on rod r2. Moreover in any plan ℓm_2 must occurs after ℓm_1. This information can also be used.

1.4 Exploiting concurrency

There exists a notion of independence (or concurrency) between operators of a planning problem. The idea is that two actions o_1 and o_2 are said to be independent if they can be fired in any order and yield the same result. In other words, for any state s from which o_1 and o_2 are firable one has $s]o_1]o_2 = s]o_2]o_1$. This notion of independence clearly generalizes to any number of actions. In presence of independent actions, considering that plans are sequences of actions artificially increases the number of possible plans. In Figure 1.1 for example the possible paths between two states with 2 to 4 independent operators are represented . With n independent operators there exists $n!$ such paths. This suggests to consider representations of planning problems taking into account this notion of independence and to search for plans as partial orders of actions rather than sequences of actions. Notice that building a sequence of operators from a partially ordered plan, for example in order to execute the plan, is straightforward. However, another interest of partially ordered plans is that their actual execution may allow to perform some actions in parallel [64]. This – compared to a sequential execution of the actions of a plan – can reduce the execution time of this plan.

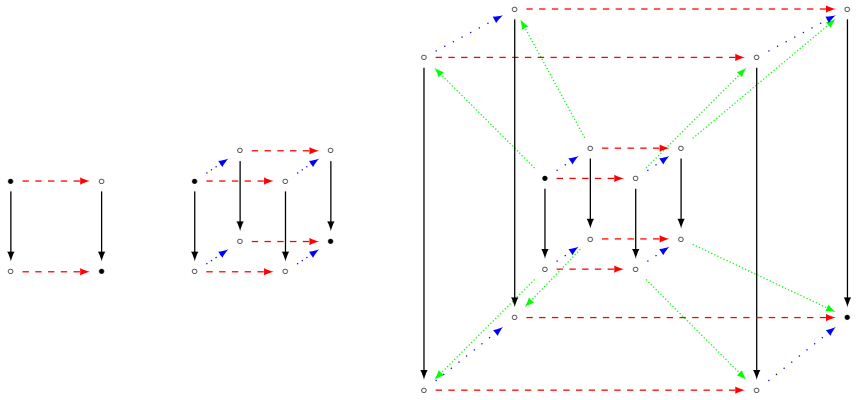


Figure 1.1: From left to rigth: paths with 2 independent operators (concurrency diamond), 3 independent operators (concurrency cube), and 4 independent operators (concurrency hypercube). States are represented by dots and each operator by a type of arrow (plain, dashed, dotted. . .).

1.4.1 Graphplan

A famous planner exploiting the independence of actions is GRAPHPLAN [8]. It is based on a notion of interference between actions. Two actions are said to *interfere* with one another if one deletes a pre-condition or an add effect of the other. Formally, two actions $o_1 = (\mathbf{pre}_{o_1}, \mathbf{del}_{o_1}, \mathbf{add}_{o_1})$ and $o_2 = (\mathbf{pre}_{o_2}, \mathbf{del}_{o_2}, \mathbf{add}_{o_2})$ interfere if and only if $\mathbf{del}_{o_1} \cap (\mathbf{pre}_{o_2} \cup \mathbf{add}_{o_2}) \neq \emptyset$ or $\mathbf{del}_{o_2} \cap (\mathbf{pre}_{o_1} \cup \mathbf{add}_{o_1}) \neq \emptyset$. Interference implies a notion of independence (actions which do not interfere are independent). The interesting fact about this notion of independence is that a set of independent actions

can all be executed together, this is called an execution step. GRAPHPLAN allows one to find plans as partial orders of actions ensuring that the number of execution steps of the plans found is minimal. This is achieved by searching plans in what is called a planning graph.

Planning graph

In this part we consider that each planning problem $\mathcal{P} = (A, O, I, G)$ is such that for every atom $a \in A$ there exists an operator $noop_a = (\{a\}, \emptyset, \{a\}) \in O$. Notice that these operators actually do nothing and can be added to any planning problem without changing the states that can be reached.

Definition 1.3. *A* planning graph *for a planning problem $\mathcal{P} = (A, O, I, G)$ is a graph $\mathcal{G}_\mathcal{P} = (V, E)$ such that $V = V_A \dot\cup V_O$ consists of two disjoint subsets of* proposition vertices *($V_A \subseteq A \times \mathbb{N}^*_+$) and* action vertices *($V_O \subseteq O \times \mathbb{N}^*_+$), and $E = E_{\mathbf{pre}} \dot\cup E_{\mathbf{del}} \dot\cup E_{\mathbf{add}}$ consists of three disjoint subsets of oriented edges called* precondition edges *($E_{\mathbf{pre}} \subseteq V_A \times V_O$),* delete edges *($E_{\mathbf{del}} \subseteq V_O \times V_A$), and* add edges *($E_{\mathbf{add}} \subseteq V_O \times V_A$). This graph is also leveled: its vertices are partitioned into disjoints subsets $V_1, \ldots, V_n$ such that $(v, v') \in E$ only if $v \in V_i$ and $v' \in V_{i-1} \cup V_{i+1}$. Moreover its levels are of two types:* proposition levels $V_i = V_{A,(i+1)/2} \subseteq \{(a, (i+1)/2) : a \in A\} \subseteq V_A$ *for all odd values of i and* action levels $V_i = V_{O,i/2} \subseteq \{(o, i/2) : o \in O\} \subseteq V_O$ *for all even values of i. Finally precondition edges are such that $E_{\mathbf{pre}} = \{((a, i), (o, i)) : a \in \mathbf{pre}_o\}$, delete edges are such that $E_{\mathbf{del}} = \{((o, i), (a, i)) : a \in \mathbf{del}_o\}$, and add edges are such that $E_{\mathbf{add}} = \{((o, i), (a, i)) : a \in \mathbf{add}_o\}$.*

Figure 1.2 presents an example of planning graph corresponding to the Tower of Hanoi problem of Section 1.1.2.

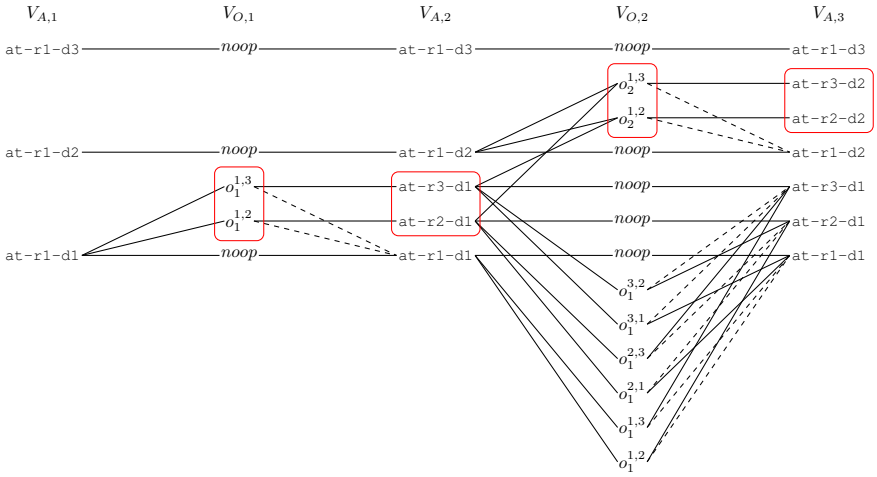


Figure 1.2: 3 firsts proposition levels and 2 firsts action levels of the planning graph constructed by GRAPHPLAN's algorithm for the Tower of Hanoi. Precondition and add edges are represented as plain lines, delete edges are represented as dashed lines. The boxed proposition vertices and action vertices represent a part of the "propagation" of mutual exclusion among levels of the graph.

Graphlan's algorithm

The principle of GRAPHPLAN is to build a particular planning graph starting from $V_{A,1} = \{(a, 1) \ : \ a \in I\}$ (which is a representation of the initial state of the problem). This graph is constructed level by level until a plan is found (a search for a plan happens after the construction of each proposition level). Each action level $V_{O,i}$ represents the actions that may be fired at time step i and each proposition level $V_{A,i}$ represents the propositions that may be true just before firing these actions. To build an action level of the graph one takes into account a notion of *mutual exclusion* between vertices of the previous proposition level. An action level $V_{O,i}$ is constructed by adding a vertex (o, i) for each action o for which for all $a \in \mathbf{pre}_o$ there is a proposition vertex (a, i) at previous proposition level $V_{A,i}$ and there exists no $a_1, a_2 \in \mathbf{pre}_o$ such that a_1 and a_2 are mutually exclusive at level $V_{A,i}$. To build a proposition level $V_{A,i+1}$ one just adds a vertex $(a, i + 1)$ for each atom a such that there is an action vertex (o, i) at previous action level $V_{O,i}$ with $a \in \mathbf{add}_o$. An example of planning graph as built by GRAPHPLAN is given in Figure 1.2.

The mutual exclusion relation is propagated along the construction of the planning graph by stating that any two actions o_1 and o_2 at a given action level are mutually exclusive if one of these actions deletes a precondition or an add-effect of the other, or if a precondition of action o_1 and a precondition of action o_2 are mutually exclusive at the previous proposition level. Moreover, two atoms a_1 and a_2 at a given proposition level are said to be mutually exclusive when they can only come from mutually exclusive actions: they can not be added by a single action, and any two actions o_1 and o_2 such that $a_1 \in \mathbf{add}_{o_1}$ and $a_2 \in \mathbf{add}_{o_2}$ are mutually exclusive actions. In Figure 1.2 a part of this propagation of mutual exclusion is highlighted: at action level $V_{O,1}$ actions $o_1^{1,2}$ and $o_1^{1,3}$ are mutually exclusive because each of them deletes a precondition of the other, from that atoms `at-r2-d1` and `at-r3-d1` are mutually exclusive at proposition level $V_{A,2}$ because they can only be added by $o_1^{1,2}$ and $o_1^{1,3}$ respectively, this implies mutual exclusion of $o_2^{1,2}$ and $o_2^{1,3}$ at level $V_{A,2}$ for having mutually exclusive preconditions, and finally `at-r2-d2` and `at-r3-d2` are mutually exclusive at level $V_{A,3}$.

The search for a plan after construction of proposition level $V_{A,i}$ is done by backward search. One first checks that the atoms defining the goals of the problem are all present at proposition level $V_{A,i}$ ($\forall a \in G, (a, i) \in V_{A,i}$) and that these atoms are not mutually exclusive. Then a set O_{i-1} of not mutually exclusive actions from action level $V_{O,i-1}$ such that $G \subseteq \cup_{o \in O_{i-1}} \mathbf{add}_o$ is searched. If one is found the same process is repeated from proposition level $V_{A,i-1}$ using $\cup_{o \in O_{i-1}} \mathbf{pre}_o$ as a goal. As soon as level $V_{A,1}$ is reached, a plan has been found. If the search from some proposition level $V_{A,k}$ (resp. action level $V_{O,\ell}$) does not allow to find a plan, then one backtracks and a different set of actions (resp. propositions) is searched in action level $V_{O,k}$ (resp. proposition level $V_{A,\ell+1}$). If all possible set have been considered at proposition level $V_{A,i}$ without finding a plan it means that no plan can be found in the current planning graph. This graph is thus extended with one more action level and one more proposition level.

Other features and comments

The algorithm presented above can be implemented efficiently using well chosen data structures for storing mutual exclusion information. Moreover, the information about goals that can not be achieved from proposition level $V_{A,i}$ can be stored to avoid repeated searches. One may also have noticed that when considering a planning problem with no solution the algorithm may not halt. In GRAPHPLAN however, a method has

been implemented to detect the absence of solution.

It is also guaranteed that the levels of the planning graphs created by GRAPHPLAN for any problem are small and quickly constructed. Consider a planning problem with n_A atoms, n_I atoms defining the initial state, n_O operators, and at most $n_{\mathbf{add}}$ atoms added by a single operator. A planning graph of depth i is a planning graph where the last proposition level is $V_{A,i}$. In this case the size of the planning graph of depth i constructed by GRAPHPLAN is polynomial in n_A, n_I, n_O, $n_{\mathbf{add}}$, and i.

In fact, GRAPHPLAN does not fully handle mutual exclusion. In particular one should have that two proposition vertices (resp. action vertices) at the same level are mutually exclusive if no plan could possibly have both true (resp. contain both) at the corresponding time step. This is not ensured by the propagation of mutual exclusion in planning graphs as done by GRAPHPLAN. Moreover, at some time step it is possible that more than two actions are mutually exclusive while not being pairwise mutually exclusive. These facts imply a necessity to check for correctness of plans with respect to mutual exclusion during the search. But to fully handle mutual exclusion would increase significantly the cost of building GRAPHPLAN's planning graph. However, there exists ways to better handle mutual exclusion such as for example Petri nets and their unfoldings.

1.4.2 Planning via Petri net unfolding

Petri nets are a well known model for systems with concurrency. Relations between Petri nets and planning were studied by various authors. We focus here on an approach to planning based on Petri net unfolding (a technique for reachability analysis preserving concurrency). This approach, presented in [47], uses a representation of planning problems as Petri nets in order to benefit of unfoldings for providing plans as partial orders of actions.

Planning problems and Petri nets

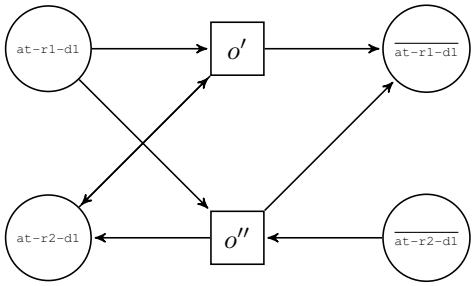
Definition 1.4. *A* Petri net *is a tuple* $PN = (P, T, F, M_0)$ *where* P *is a set of* places*,* T *is a set of* transitions *(*$P \cap T = \emptyset$*),* $F : (P \times T) \cup (T \times P) \to \mathbb{N}_+$ *is a* flow relation*, and* M_0 *is an* initial marking

The places represent the state variables of the system, while the transitions represent its actions. A *marking* $M : P \to \mathbb{N}_+$ associates a certain number of tokens to each place, it represents a state of the system. The flow relation relates the places and the transitions, defining the behavior of the system. From a certain marking M a transition t can be fired if and only if $\forall p \in P, F(p,t) \leq M(p)$. The firing of this transition leads to the new marking M' such that $\forall p \in P, M'(p) = M(p) - F(p,t) + F(t,p)$. Planning problems will be encoded as *1-safe* Petri nets, that is Petri nets such that $\forall p \in P, M_0(p) \leq 1$ and after any sequence of transitions fired from M_0 the marking M reached is such that $\forall p \in P, M(p) \leq 1$. In other words: a place will never contain more than 1 token (intuitively, a place will represent the truth value of an atom, that is a binary variable).

A planning problem $\mathcal{P} = (A, O, I, G)$ can then be represented as the Petri net $PN_{\mathcal{P}} = (P, T, F, M_0)$ formally defined as follows (an example is also given in Figure 1.3):

- $P = A \cup \overline{A}$, where $\overline{A}$ is a set of variables such that $\overline{a} \in \overline{A}$ if and only if $a \in A$, the idea being that $\overline{a}$ is true if and only if a is false.

- $T = \cup_{o \in O} S(o)$, where one has $S(o) = \{(\mathbf{pre}, \mathbf{del}, \mathbf{add}) \ : \ \exists \mathbf{del}' \subseteq \mathbf{del}_o \setminus \mathbf{pre}_o, \exists \mathbf{add}' \subseteq \mathbf{add}_o, \mathbf{pre} = \mathbf{pre}_o \cup \mathbf{del}' \cup \overline{\mathbf{add}'} \cup \overline{(\mathbf{del}_o \setminus \mathbf{pre}_o) \setminus \mathbf{del}'}, \mathbf{del} = \mathbf{del}' \cup (\mathbf{del}_o \cap \mathbf{pre}_o) \cup \overline{\mathbf{add}'}, \mathbf{add} = \mathbf{add}' \cup \overline{\mathbf{del}' \cup (\mathbf{del}_o \cap \mathbf{pre}_o)}\}$.
- For $p \in P$ and $t = (\mathbf{pre}_t, \mathbf{del}_t, \mathbf{add}_t) \in T$, $F(p,t) = 1$ if and only if $p \in \mathbf{pre}_t$, and $F(t,p) = 1$ if and only if $p \in \mathbf{add}_t$ or $p \in \mathbf{pre}_t \wedge p \notin \mathbf{del}_t$.
- For $a \in A, M_0(a) = 1$ if $a \in I$ and $M_0(a) = 0$ else, and for $\overline{a} \in \overline{A}, M_0(\overline{a}) = 1 - M_0(a)$.



$$o' = (\{\texttt{at-r1-d1}, \texttt{at-r2-d1}\}, \{\texttt{at-r1-d1}\}, \{\overline{\texttt{at-r1-d1}}\})$$

$$o'' = (\{\texttt{at-r1-d1}, \overline{\texttt{at-r2-d1}}\}, \{\texttt{at-r1-d1}, \overline{\texttt{at-r2-d1}}\}, \{\texttt{at-r2-d1}, \overline{\texttt{at-r1-d1}}\})$$

Figure 1.3: Mapping of the action $o = (\{\texttt{at-r1-d1}\}, \{\texttt{at-r1-d1}\}, \{\texttt{at-r2-d1}\})$ (from the Tower of Hanoi problem of Section 1.1.2) into a Petri net. The two transitions o' and o'' correspond to the elements of $S(o)$. The marking is not represented because only one action is considered, not a full planning problem. The places are depicted as circles and the transitions as squares. The arrows represent the flow relation.

One can then show that solving $\mathcal{P}$ is equivalent to finding way to reach a marking corresponding to G in $PN_{\mathcal{P}}$. The markings corresponding to G being the markings M_G such that for $a \in A, M_G(a) = 1$ if $a \in G$ and $M_G(a)$ can take any value otherwise, and for $\overline{a} \in \overline{A}, M_G(\overline{a}) = 1 - M_G(a)$.

Theorem 1.1 (Corollary of Theorem 1 of [47])**.** *Finding a sequence of firing of transitions in $PN_{\mathcal{P}}$ allowing to reach a marking M_G from the original marking M_0 is equivalent to finding a plan in the planning problem $\mathcal{P}$.*

Unfolding Petri nets for solving planning problems

In fact, as suggested above, one will prefer to search for a way to reach a marking M_G in the form of a partial order of transitions. Unfolding is a method for performing reachability analysis in Petri nets while preserving concurrency [71, 26]. Thus it permits to obtain plans as partial orders of actions from a Petri net. Figure 1.4 represents a Petri net and (a part of) its unfolding. Intuitively it corresponds to the possible runs in the net. The unfolding of a Petri net gives a partially ordered plan for achieving each reachable marking of this net. In planning however, one only needs to find one plan,

thus unfolding can be stopped as soon as a marking M_G has been reached. This has been done for example in Figure 1.4.

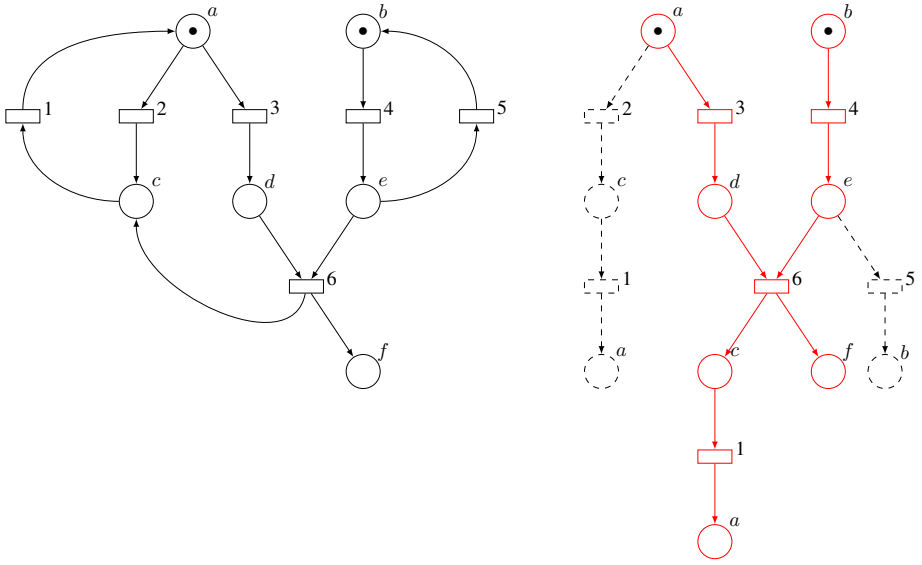


Figure 1.4: A Petri net (left) and a prefix of its unfolding (right) sufficient for finding how to reach a marking M_G such that $M_G(a) = M_G(f) = 1$, and $M_G(b) = M_G(c) = M_G(d) = M_G(e) = 0$. In the unfolding plain lines represent a partially ordered plan reaching this marking.

Moreover, one can use standard heuristics to "drive" the unfolder and try to reach a marking M_G without considering too many other markings. Indeed, unfoldings are built by adding transitions (usually called events in the unfolding) one by one. One can use different policies for defining the order in which these events are added. In particular it is possible to add in priority the events that are the most promising for reaching a goal marking. The notion of promising event can be defined by the mean of heuristics, as in A*. In Figure 1.4 this could avoid the unfolding of the dashed parts (or at least part of them). Driven unfolding was first suggested in [47] and then studied in details in [11].

1.5 Exploiting modularity

Beyond concurrency is the notion of modularity. Some planning problems are intrinsically modular: they are formed by several subproblems (or *components*, or *factors*) which can be solved almost independently. This can be the case for example in planning problems implying several agents which do not interact much (i.e. are highly concurrent). In this case each agent and the actions and variables related to her will define a subproblem. However, the components allowing the efficient modular resolution of a planning problem are not always as meaningful as in the case of multi-agent problems: they can simply come from independence relations between variables or actions of a problem.

The fact that some parts of a planning problem can be loosely interacting was first used in what is called *hierarchical planning* [63]. The idea is to solve a problem using a hierarchy of abstractions: first the most abstract version of the problem is solved, the result is then used for solving a finer version of the problem, and so on and so forth until the non-abstracted problem is solved. After that appeared the notion of *factored planning* which consists in decomposing a problem into loosely interacting *factors*, in solving these factors, and finally in gathering their solutions into a plan for the original problem. The factors are frequently defined by a partition of the atoms or by a partition of the actions of the original planning problem. Such a partition simply defines a set of planning problems as shown in Section 1.5.2 for the case of actions and in Chapter 2 for the case of atoms.

The main challenges in factored planning are the following: to identify the planning problems for which using factored planning makes sense, to decompose the problems into factors, and to solve the factored problems. One may think that these challenges are almost independent. However even if some general guidelines can be used for decomposing a problem (limit the interactions between factors as most as possible) the definition of a good decomposition highly depends on the method used for solving factored problems. Accordingly, depending on the decomposition method (and thus depending on the resolution method), the planning problems which can be efficiently solved will not always be the same.

1.5.1 A first approach to factored planning

The first true factored planner (by planner we refer to a program solving planning problem) was presented in [1]. The approach uses a specific decomposition of planning problems as factors organized into a tree. Factors are treated from leaves to root to propagate constraints and then from root to leaves to build a plan. In this section we describe this decomposition and explain how it is used for solving factored planning problems.

Factored planning formalism of [1]

The formalism for factored planning problem used in this first approach is to represent a problem as a tree $\mathcal{T} = (V, E, \ell)$, an initial state $I_{\mathcal{T}}$ and a set of goal states $G_{\mathcal{T}}$ where:

- $V = \{D_i \ : \ i \leq m\}$ is a set of *planning domains*, that is a set of couples $D_i = (A_i, O_i)$ with A_i a set of atoms and O_i a set of operators over A_i, these D_i are the factors;
- $E \subseteq V \times V$ is a set of edges;
- $\ell : E \to \cup_{i \leq m} A_i$ is a labeling of the edges;
- $I_{\mathcal{T}} \subseteq \cup_{i \leq m} A_i$;
- $\exists t \leq m, G_{\mathcal{T}} \subseteq A_t$, the corresponding domain D_t is the *root* of $\mathcal{T}$.

This tree should also have the following three properties:

1. for any $e = (D_i, D_j) \in E$, $\ell(e) \supseteq A_i \cap A_j$;
2. for any $i \neq j$ such that $A_i \cap A_j \neq \emptyset$ a path should exist between D_i and D_j;

3. any atom appearing in both A_i and A_j for $i \neq j$ should also appear in the label of each edge of the shortest path between D_i and D_j.

The objective in such a factored planning problem is to find a plan for the corresponding planning problem $\mathcal{P}_\mathcal{T} = (\cup_{i \leq m} A_i, \cup_{i \leq m} O_i, I, G)$ (without considering this problem directly but using its factored version instead).

It is possible to transform any planning problem $\mathcal{P} = (A, O, I, G)$ into a factored planning problem with this formalism (consider the tree $\mathcal{T} = ((A, O), \emptyset, \ell)$, the initial state $I_\mathcal{T} = I$, and the set of goal states $G_\mathcal{T} = G$). However any tree $\mathcal{T} = (V, E, \ell)$ respecting the above definition and such that $\cup_{i \leq m} A_i = A$ and $\cup_{i \leq m} O_i = O$ (with $I_\mathcal{T} = I$, and $G_\mathcal{T} = G$) would be acceptable. In fact, in order to really benefit from factored planning one has to have many small factors rather than few big ones. For a planning problem $\mathcal{P}$, consider the graph $\mathcal{G}_\mathcal{P}$ which vertices are the elements of A and where there is an edge between a_1 and a_2 if and only if there exists an action $o \in O$ such that $a_1 \in \mathbf{pre}_o \cup \mathbf{del}_o \cup \mathbf{add}_o$ and $a_2 \in \mathbf{pre}_o \cup \mathbf{del}_o \cup \mathbf{add}_o$ (an example is given in Figure 1.5 in the case of the Tower of Hanoi problem of Section 1.1.2). According to [1], a possible way to automatically build a tree from a planning problem is to use tree decomposition techniques (see for example [9]) on this graph. These techniques will return a tree $\mathcal{T}_\mathcal{P} = (\{A_i\}_i, E_\mathcal{P})$ verifying the following properties: $\cup_i A_i = A$, for every edge (a_1, a_2) of $\mathcal{G}_\mathcal{P}$ there exists i such that $a_1 \in A_i$ and $a_2 \in A_i$, and for every i, j, k, if A_j is on the path from A_i to A_k in $\mathcal{T}_\mathcal{P}$ then $A_i \cap A_k \subseteq A_j$. These subsets A_i are then used to build the planning domains $D_i = (A_i, O_i)$ for the factored representation, where O_i is the set of all operators o_i such that $\mathbf{pre}_{o_i} \cup \mathbf{del}_{o_i} \cup \mathbf{add}_{o_i} \subseteq A_i$.

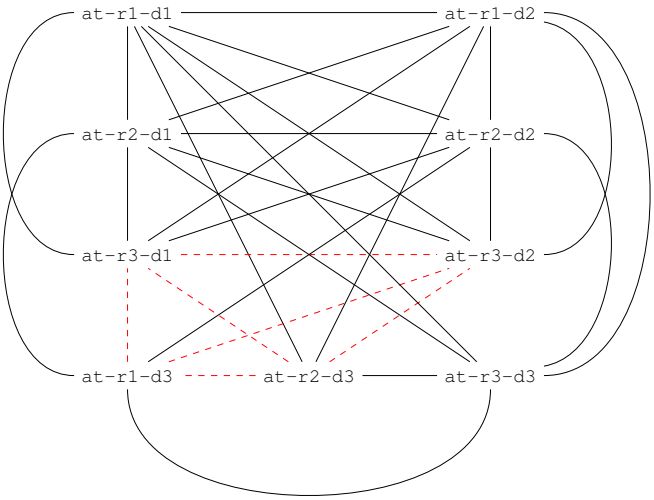


Figure 1.5: Interactions by actions between the atoms of the Tower of Hanoi problem. The dashed edges correspond to the action moving `d3` from `r1` to `r2`.

Planning algorithm of [1]

A schematic presentation of the algorithm proposed for solving factored planning problems as presented above is given in Algorithm 5. This algorithm takes as input a factored planning problem $\mathcal{T} = (V, E, \ell)$, an initial state $I_\mathcal{T}$, a set of goal states $G_\mathcal{T}$, and

two parameters k and d which will bound the lengths of plans considered in the factors. The principle of this algorithm is to select a leaf of $\mathcal{T}$ and search for plans between various valuations of the variables shared with its parent. These plans are then sent to the parent in the form of a set of actions (called macro actions) reflecting their effects and preconditions. The leaf is then removed from $\mathcal{T}$ and the same process is repeated until only the root remains in $\mathcal{T}$. After that, a plan p is searched in the root of $\mathcal{T}$ and sent to its children in the original $\mathcal{T}$. For each of these children, the macro actions in p are then replaced by the corresponding local plans and an updated p is sent to their own children. The same process is repeated until all the gaps in p have been filled, that is until p has been updated using the leaves of the original $\mathcal{T}$.

Three parts of Algorithm 5 have to be explained: the definition of $\mathcal{CO}_k$ and the way it is taken into account in the research for a plan (lines 5 and 6), the meaning of an update of D_j (line 8), and the meaning of expanding a plan (line 16). We only give intuitions on these parts of the algorithm, however, detailed implementations can be found in [1].

Algorithm 5 A first algorithm for factored planning

1: $(V', E') = (V, E)$
2: **while** $V' \neq \{D_t\}$ **do**
3: let D_i be a leaf of (V', E')
4: let D_j be the parent of D_i in (V', E')
5: **for all** $\mathcal{CO} \in \mathcal{CO}_k(\ell((D_i, D_j)))$ **do**
6: search a plan in D_i according to $\mathcal{CO}$
7: **end for**
8: update D_j with all the plans found
9: $V' \leftarrow V' \setminus \{D_i\}$
10: $E' \leftarrow E' \setminus \{(D_j, D_i)\}$
11: **end while**
12: search a plan in D_t achieving $G_{\mathcal{T}}$
13: **if** a plan p has been found **then**
14: **while** $V' \neq V$ **do**
15: let $D_i \notin V'$ and $D_j \in V'$ be two neighbors in $\mathcal{T}$
16: expand p in D_i from D_j
17: $V' \leftarrow V' \cup \{D_i\}$
18: $E' \leftarrow E' \cup \{(D_j, D_i)\}$
19: **end while**
20: return p
21: **else**
22: no solution exists for parameters d and k
23: **end if**

For any $(D_i, D_j) \in E$, the set $\mathcal{CO}_k(\ell((D_i, D_j)))$ contains all possible tuples $(\{\mathbf{pre}_i : i < k\}, \{\mathbf{eff}_i : i \leq k\})$ with $\mathbf{pre}_i \subseteq \ell((D_i, D_j))$ for all $i < k$, and $\mathbf{eff}_i \subseteq \ell((D_i, D_j))$ for all $i \leq k$. Using these tuples a search is done in D_i (line 6). This search has for objective to find a path p (using at most d operators) which reaches a state s such that $\mathbf{eff}_k \cap s = \mathbf{eff}_k$. Moreover, new operators are added to D_i in order to reach this state, each of these operators o_i can only be fired from states s such that $s \cap \mathbf{eff}_i = \mathbf{eff}_i$ and from s reaches the new state $s' = s \setminus \mathbf{eff}_i \cup \mathbf{pre}_i$. Moreover, operator $o_i, i \geq 1$ can only be fired once and if o_{i-1} has already be fired before. This can be

implemented using only operators as defined in propositional STRIPS by adding atoms to D_i.

The paths found at line 6 are then used at line 8 to modify D_j. The idea is to add new operators in D_j for each path p found. These operators enforce to fill the blanks (i.e. provide a path from the precondition of an o_i in D_i to its effects) in p if one needs to use some state s in D_j such that $s \cap \mathbf{eff}_k = \mathbf{eff}_k$ (for the value of $\mathbf{eff}_k$ corresponding to p). Intuitively, at line 6 it has been stated that some valuations of the atoms shared between D_i and D_j can be achieved by the operators of D_i if operators in D_j allows to reach some other valuations of these shared atoms.

Finally, when a path has been found at the root to reach a goal of the planning problem considered, it is needed to expand it into a real path for the planning problem: the operators added during the first part of the algorithm have to be replaced by operators of the original planning problem. This is done at line 16 where the operators added to D_j using the paths in D_i are replaced by relevant parts of these paths. This is always possible as the added operators all correspond to actual paths by construction (line 8).

1.5.2 Factored planning using constraint solving

Another factored planning algorithm of interest was proposed in [13]. This algorithm works for factored planning problems defined by a partition of the actions of a standard planning problem. These problems are recast as constraint solving problems (CSP) by focusing on coordination points between the local plans of the factors (i.e. shared actions). This allows one to use standard constraint solving methods for planning. In particular methods based on message passing algorithms can be used [22].

Factored planning formalism of [13]

A factored planning problem (called multi-agent problem in [13]) is defined as a tuple $\mathcal{P} = (A, \{O_i \ : \ i \leq k\}, I, G)$ such that $\mathcal{P}' = (A, \cup_{i \leq k} O_i, I, G)$ is a standard planning problem. Each of the k sets of operators O_i defines a factor $\mathcal{P}_i = (A_i, O_i, I_i, G_i)$ which is itself a planning problem. In such a factor A_i is the set of atoms involved in the operators from O_i, $I_i = I \cap A_i$,and $G_i = G \cap A_i$. The operators of a factor can be separated in two sets: the set of *public operators* and the set of *private operators*. Public operators are the ones with preconditions or effects including atoms from several factors. Private operators are the ones with preconditions and effects using atoms from only one factor.

An important notion in such a factored planning problem is the notion of interaction graph of the factors. This graph has factors as vertices and an edge between two factors $\mathcal{P}_i$ and $\mathcal{P}_j$ if and only if they share atoms ($A_i \cap A_j \neq \emptyset$). The complexity of solving a factored planning problem is highly related with the tree-width of this interaction graph due to the constraint satisfaction methods used which require it to be a tree [22]. Section 1.6.2 gives more details about this complexity.

Planning algorithm of [13]

Given a factored planning problem $\mathcal{P} = (A, \{O_i \ : \ i \leq k\}, I, G)$ one prefers to solve $\mathcal{P}$ by searching plans in its factors rather than directly solve its counterpart $\mathcal{P}' = (A, \cup_{i \leq k} O_i, I, G)$. The factored planning algorithm proposed consists in assuming that a plan p exists such that no more than d public operators will be used in any factor. This implies that no more than $d \times k$ public operators will appear in p (there is k factors).

Then one can recast the factored planning problem as a CSP in which the objective is to find the sequence of public operators used in each factor (and a "time" in $\{1..dk\}$ for the occurrence of each of these operators, this time being in increasing order along the sequence). In other words one has to find a tuple $(\theta_1, \ldots, \theta_k)$ of sequences of couples of public operators and times of length at most d with constraints ensuring that these sequences correspond to an actual plan in $\mathcal{P}$.

The first constraint is called *coordination constraint*. Intuitively it ensures that the sequences of public operators in all the factors can be assembled into a single coherent sequence. This constraint can be expressed as follows: $(\theta_1, \ldots, \theta_k)$ should be such that for any (o, t) in θ_i, for any $a \in \mathbf{pre}_o \cap (\cup_{j \neq i} A_j)$ one has:

1. $\exists (o', t')$ in some θ_j such that $a \in \mathbf{add}_{o'}$ and $t' < t$ or $a \in I$ (in this case $t' = 0$): a is supplied before being needed;

2. $\nexists (o'', t'')$ in some θ_ℓ such that $a \in \mathbf{del}_{o''}$ and $t' \leq t'' \leq t$: a is not destroyed before having been used.

Moreover, for any $a \in G \cap (\cup_{i \neq j}(A_{O_i} \cap A_{O_j}))$, 1. and 2. should also hold for (a, t) with $t = dk + 1$.

The second constraint is called *internal-planning constraint*. It ensures that in each factor $\mathcal{P}_i$ it is possible to find a plan which allows to fire the public operators in the order requested by θ_i. For O_i a set of operators, consider the planning problem $\mathcal{P}'_i = (A_i^{int}, O_i^{int}, I_i^{int}, G_i^{int})$ where A_i^{int} is the set of atoms appearing only in the operators from O_i, $I_i^{int} = I \cap A_i^{int}$, and $G_i^{int} = G \cap G_i^{int}$. The set of operators is $O_i^{int} = \{o' : o \in O_i \wedge \mathbf{pre}_{o'} = \mathbf{pre}_o \cap A_i^{int}, \mathbf{del}_{o'} = \mathbf{del}_o \cap A_i^{int}, \mathbf{add}_{o'} = \mathbf{add}_o \cap A_i^{int}\}$, for simplicity of presentation we consider that there are no two operators o_1 and o_2 in O_i such that $\mathbf{pre}_{o_1} \cap A_i^{int} = \mathbf{pre}_{o_2} \cap A_i^{int}$, $\mathbf{del}_{o_1} \cap A_i^{int} = \mathbf{del}_{o_2} \cap A_i^{int}$, and $\mathbf{add}_{o_1} \cap A_i^{int} = \mathbf{add}_{o_2} \cap A_i^{int}$. This second constraint can be expressed as follows: $(\theta_1, \ldots, \theta_k)$ should be such that for any $\theta_i = (o_1, t_1) \ldots (o_\ell, t_\ell)$, there exists a plan $p = o''_1 \ldots o''_n$ in $\mathcal{P}'_i$ such that there exists a subplan $p' = o''_{m_1} \ldots o''_{m_\ell}$ with $o''_{m_q} = o'_q$ (o'_q is the operator built from o_q in $\mathcal{P}'_i$) for all $1 \leq q \leq \ell$ and $m_q < m_r$ for all $q < r$. Moreover no o'_q should appear in p outside of p'.

If the CSP considered has no solution, one just increases the value of d and tries to solve the new CSP it defines. As in previous section, this method does not allow cost-optimal planning: one should test all d (up to the length of the longest possible plan, which is exponential in the size of $\mathcal{P}$) for ensuring cost-optimality of plans found.

1.6 Complexity of planning

We conclude this section by giving some known results about the complexity of planning in STRIPS domains. We consider in fact a slightly different version of STRIPS, where preconditions of operators are defined by two sets of atoms: a set called positive precondition (atoms for which the truth value must be true for firing the operator) and a set called negative precondition (atoms for which the truth value must be false for firing the operator). In our definition of STRIPS, only positive preconditions exist. It is in fact possible to build a STRIPS domain with only positive preconditions from a STRIPS domain with both positive and negative preconditions by adding new atoms representing negation of original atoms [33]. The interest of considering negative preconditions is thus not to increase the expressivity of STRIPS but to allow a more accurate complexity analysis.

1.6.1 Complexity in general

A large collection of complexity results for planning in STRIPS domains is given in [15]. The decision problem considered for studying complexity is PLANSAT: decide whether or not there exists a plan for a given STRIPS planning problem. As we will see the complexity of this decision problem depends on the number and types of preconditions and effects in the operators of the STRIPS domain considered. We thus adopt the following notation: PLANSAT(k, ℓ, m, n) with $k, m \in \mathbb{N} \cup \{*\}$ and $\ell, n \in \{+, -, *\}$ denotes the decision problem of the existence of a plan in a STRIPS domain where some restrictions are imposed on operators. The value $k \neq *$ (resp. $m \neq *$) bounds the maximum number of atoms in the precondition (resp. effects) of any operator. The value $\ell \neq *$ (resp. $n \neq *$) specifies that for any operator only positive $(+)$ or negative $(-)$ preconditions (resp. effects) exist. The symbol $*$ means that nothing is specified: PLANSAT=PLANSAT$(*, *, *, *)$. The main result about complexity of planning in STRIPS domains is the following:

Theorem 1.2 (Theorem 1 in [15])**.** *PLANSAT is PSPACE-complete.*

The fact that PLANSAT is in PSPACE is due to the observation that with n atoms the number of possible states is 2^n. Moreover, if a plan exists, a plan with no loops exists. Thus, if a plan exists, a plan of length less than 2^n exists. Such a plan can be found doing at most 2^n non-deterministic choices, so PLANSAT is in NPSPACE=PSPACE. The PSPACE-hardness of PLANSAT can be proved by giving a polynomial reduction of any turing machine using only polynomial space to an instance of PLANSAT. As this reduction only implies operators with positive precondition, no more than two atoms in precondition, and no more than two atoms in effects, one gets the following result:

Proposition 1.1 (Corollary 1 in [15])**.** *PLANSAT*$(2, +, 2, *)$ *is PSPACE-complete.*

This implies, in particular, that the version of STRIPS we consider (with no negative precondition) is such that PLANSAT is PSPACE-complete. Finally, even with only one effect in each operator PLANSAT remains in PSPACE:

Proposition 1.2 (Theorem 3 in [15])**.** *PLANSAT*$(*, *, 1, *)$ *is PSPACE-complete.*

Some particular planning domains however are NP-complete, or even polynomial. These are generally very restrictive on the operators allowed. The following lemmas give some examples.

Proposition 1.3 (Theorem 4 in [15])**.** *PLANSAT*$(*, *, *, +)$ *is NP-complete.*

Proposition 1.4 (Corollary 5 in [15])**.** *PLANSAT*$(1, *, 1, +)$ *is NP-complete.*

Proposition 1.5 (Theorem 7 in [15])**.** *PLANSAT*$(*, +, 1, *)$ *is polynomial.*

Proposition 1.6 (Theorem 9 in [15])**.** *PLANSAT*$(0, *, *, *)$ *is polynomial.*

Among other works of interest on complexity of planning one can notice [25]. This paper presents complexity results for PLANSAT and for k-PLANSAT (decide whether or not there exists a plan of length smaller than some bound k), these results being presented for a similar STRIPS formalism than in [15] and for various other formalism generalizing STRIPS. One can also notice [3] for results on complexity of planning in SAS (allowing multi-valued variables instead of STRIPS atoms) and SAS+ (SAS with multiple initial states) domains. PLANSAT reveals to have the same complexity for SAS and SAS+ domains than for STRIPS domains.

1.6.2 The case of factored planning

The first important fact to notice is that there is no hope to reduce the complexity of planning by using factored methods. It is always possible that a planning problem can not be factorized in more than one factor. However, when it is possible to factorize a planning problem in many small factors with few interaction, one can really benefit from factored planning. In fact, when factorization is possible, complexity of factored planning algorithms is highly related to the tree-width of the graph of interaction between variables or actions of the planning problem considered. The tree-width of a graph being the minimum number of vertices which have to be aggregated in a single vertex in order to make a tree from this graph.

For example, in [1], the complexity of the algorithm presented is exponential in the number of atoms in the largest factor, which is in fact the tree-width of the graph of interaction between atoms used for automated factorization of planning problems. This complexity is also linear in the cost of searching a plan in any factor.

Similarly, in [13], the complexity of the presented algorithm is exponential in the tree-width of the graph of interaction between the (pre-defined) factors, and linear in the cost of searching a plan in a factor. The relation with tree-width is due to the constraint satisfaction methods used [22].

Theorem 1.3 (Theorem 2.4 in [1], Equation 2 in [13])**.** *Complexity of factored planning is linear in the cost of searching a plan in a factor and exponential in the tree-width of the graph of interaction between factors.*

This theorem strongly relates the complexity of factored planning with the complexity of classical planning. As soon as no decomposition of a planning problem can be found (i.e. when tree width corresponds to the number of factors) factored planning corresponds to classical planning. It also reveals that tree-width is an important parameter for factored planning, even if it is not the only one. Indeed, complexity also depends of the degree of interaction between factors: the more shared atoms or actions exist between any two factors, the more factored planning is complex. In [1] the complexity of the algorithm presented is exponential in the maximum number of atoms shared between two factors. And in [13] the complexity is polynomial in the number of public actions (actions using a shared atom between two factors), but the degree of the polynomial is the tree-width of the graph of interaction between factors.

Conclusion

In this chapter we gave a small overview of planning. This view is far from being exhaustive due to the vast literature brought to us by more than 40 years of research in planning since publication of the original paper on A* in 1968. This prevented us from presenting more “exotic” planners, such as SATPLAN [56] for example, which exploits a SAT-solver for solving planning problems, obtaining extremely good results in planning competitions. Notice that domain specific planning was originally the norm. But, recently lots of works have been done on domain independent planning. Clearly, domain independence limits the efficiency of plan search. However it avoids the tedious work of designing new planners for each family of problems.

We focused on the evolution from planning as heuristic search to factored planning (and in some sense back to planning as heuristic search since recent heuristics benefit from the techniques developed for exploiting concurrency and modularity of

problems). This relatively new approach to planning can be located between domain specific and domain independent planning. Indeed, is uses some characteristics of problems (that is independence of subproblems) for solving them efficiently, while allowing to deal with a large range of problems. In fact, factored planning generally uses standard planning methods for solving subproblems (which are themselves smaller planning problems). This makes it a method having some of the advantages of domain specific planning (when a good decomposition exists it is very efficient) while not having its drawbacks (in the worst case there exists only one factor and factored planning corresponds to standard planning).

However, it is not possible to find cost-optimal plans with the current factored planning algorithms. They are all based on some bounds on the length of plans, increased along the execution of the algorithm. Proposing solutions for avoiding the use of such bounds thus looks to be a topic of interest and an important gap to fill. Moreover very few implementations of factored planners exist, and none were compared with implementations of other approaches. In fact, the only implementation we are aware of is the one presented in [1].

Another topic of interest is the use of approximate methods for factored planning. Indeed, such methods revealed interest for planning, allowing to significantly reduce the time needed to find plans by allowing them to be close to optimal instead of optimal. This is due to the fact that the difficulty in the search for an optimal solution is not in finding the solution but rather in checking its optimality. However, these kinds of approaches – despite their noted interest – were almost not considered in the case of factored planning.

Finally, an important fact about planning is that comparing planners is a difficult task. Few theoretical studies exist on this topic. Currently the main methods for planning algorithms comparison consists in the use of benchmarks. The planners which have to be evaluated are run on the same set of problems and their performances are compared (usually one is interested in run time). This is in particular what happens during international planning competitions. The choice of benchmarks has however a huge impact on this evaluation method. Moreover, due to this manner of comparing planners there may exist a tendency to avoid theoretical analysis of algorithms and a focus on experimental results only. In our opinion it is necessary to have a minimal theoretical understanding of algorithms, at least by proving their validity. For this reason, in this thesis we always start by giving formal descriptions of our results before doing empirical optimizations on top of this formal basis.

Chapter 2

Planning in Networks of Weighted Automata

chapter abstract: *In this chapter we present the classical message passing algorithm and show how it can be used for factored cost-optimal planning. This is done by encoding planning problems in terms of weighted automata and showing that the basic operations (product and projection) needed for message passing can be implemented directly on weighted automata. This work is a basis for the results of Chapters 3, 4, and 5.*

AS PRESENTED IN Chapter 1 current approaches to cost-optimal planning are mainly based on A^* like algorithms, using heuristic functions to drive the graph exploration. If heuristics are accurate, these approaches allow one to avoid looking at cost expensive paths. Thus, they reduce the explored part of the state-space. However, driving the search using heuristics is not the only possible method for reducing the cost of finding a path. In particular, the relatively new approach called factored planning (see Section 1.5) is of interest. Nevertheless, current factored planning approaches do not allow one to perform cost-optimal planning. Indeed, they usually rely on incrementally adjusted bounds on length of the paths (or more precisely on the number of synchronization actions in a path). The aim of this chapter is to propose a general framework for factored planning not relying on such bounds, and to show how it can be used in order to perform factored cost-optimal planning.

The approach presented in this Chapter is derived from a family of algorithms called message passing algorithms [28]. It allows one to permanently handle all the valid plans in each component of a factored planning problem by representing sets of plans as weighted regular languages. From now on we prefer speaking about components rather than factors because we consider that problems are given decomposed. The idea of our approach is to begin with the languages (one per component) containing all local plans of each subproblem and then refine these sets of plans to conserve only the valid ones, that is the local plans corresponding to global solutions. In this chapter we focus on the theoretical aspects of our method. We show that factored cost-optimal planning is possible but we do not consider its efficiency. The practical aspects of our approach are presented in Chapter 3.

This chapter is structured as follows. First we describe how planning problems

and factored planning problems can be encoded in terms of weighted automata (Section 2.1). Then we present message passing algorithms in their generality (Section 2.2) and show how they can be applied to find cost-optimal plans in factored planning problems using weighted languages theory (Section 2.3). Finally we show that this method works directly with weighted automata (Section 2.4) by proposing weighted automata implementations of the operations on weighted languages.

2.1 Automata and (factored) planning

In Chapter 1 we desrcribed various representations of planning problems, such as graphs or Petri nets. In this chapter we consider a different representation using weighted automata [16, 81]. The interest of this representation is to enable the use of known results from weighted automata theory and regular weighted languages theory for planning.

2.1.1 Planning problems in terms of automata

Definition 2.1. *An* automaton *is a tuple* $\mathcal{A} = (S, S^I, S^F, \Sigma, T)$ *where* S *is a finite set of* states*,* $S^I \subseteq S$ *is a set of* initial states*,* S^F *is a set of* final states*,* Σ *is a finite set of* actions *(also called alphabet), and* $T \subseteq S \times \Sigma \times S$ *is a set of* transitions*.*

Any transition $t \in T$ is denoted as $t = (t^-, \sigma_t, t^+)$. In such an automaton a *path* is a sequence $\pi = t_1 \dots t_n$ of transitions such that $\forall 1 < i \leq n, t_i^- = t_{i-1}^+$. For any such path we denote by π^- its first state t_1^- and by π^+ its last state t_n^+. An *accepted path* is a path π such that $\pi^- \in S^I$ and $\pi^+ \in S^F$. A *word* in $\mathcal{A}$ is the sequence of actions $\sigma(\pi) = \sigma_{t_1} \dots \sigma_{t_n}$ corresponding to an accepted path $\pi = t_1 \dots t_n$. Finally, the *language* $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of all its words: $\mathcal{L}(\mathcal{A}) = \{\sigma(\pi) \ : \ \pi \text{ an accepted path in } \mathcal{A}\}$.

The set of solutions of a planning problem $\mathcal{P} = (A, O, I, G)$ given in STRIPS formalism corresponds to the language of the automaton $\mathcal{A}_\mathcal{P} = (S, S^I, S^F, \Sigma, T)$ such that:

- $S = 2^A$: the set of states of the automaton corresponds to the states of the planning problem;
- $S^I = \{I\}$: the only initial state of the automaton corresponds to the initial state of the planning problem;
- $S^F = \{s \in S \ : \ s \supseteq G\}$: the final states of the automaton are the goal states of the planning problem;
- $\Sigma = O$: the actions of the automaton are the operators of the planning problem;
- $T = \{(s^-, o, s^+) \ : \ o \text{ is firable from } s^- \text{ and } s^-]o = s^+\}$.

Solving $\mathcal{P}$ and finding a word in $\mathcal{A}_\mathcal{P}$ are thus two instances of the same problem. As we are interested in cost-optimal planning a slightly different model is however needed to integrate the notion of cost in automata. This model is called weighted automaton.

Definition 2.2. *A* weighted automaton *is a tuple* $\mathcal{A} = (S, S^I, S^F, \Sigma, T, c, c^i, c^f)$ *such that* (S, S^I, S^F, Σ, T) *is an automaton,* $c : T \to \mathbb{R}_+$ *is a cost function on transitions,* $c^i : S^I \to \mathbb{R}_+$ *is a cost function on initial states, and* $c^f : S^F \to \mathbb{R}_+$ *is a cost function on final states.*

The notions of path, accepted path, and word are defined similarly as for automata. The novelty is that each path $\pi = t_1 \dots t_n$ is associated to a cost

$$c(\pi) = \sum_{1 \leq i \leq n} c(t_i).$$

And from that each word w is associated to a cost $c(w)$ defined as the minimal cost over all accepted paths giving that word, taking into account the initial and the final costs:

$$c(w) = \min_{\pi, \sigma(\pi)=w} (c^i(\pi^-) + c(\pi) + c^f(\pi^+)).$$

Finally, the language $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is defined as the set of all words in $\mathcal{A}$ associated to their cost:

$$\mathcal{L}(\mathcal{A}) = \{(w, c(w)) \ : \ \exists \pi \text{ an accepted path in } \mathcal{A}, \sigma(\pi) = w\}.$$

Given a cost-optimal planning problem $\mathcal{P} = (A, O, I, G)$ with cost function $c_{\mathcal{P}}$, consider the automaton $\mathcal{A} = (S, S^I, S^F, \Sigma, T, c, c^i, c^f)$ such that $(S, S^I, S^F, \Sigma, T) = \mathcal{A}_{\mathcal{P}}$ as defined above, $\forall t \in T, c(t) = c_{\mathcal{P}}(\sigma_t)$, $\forall s \in S^I, c^i(s) = 0$, and $\forall s \in S^F, c^f(s) = 0$. One gets that $\mathcal{L}(\mathcal{A})$ is exactly the set of plans in $\mathcal{P}$ associated with their minimal cost. Thus, finding a minimal cost word in $\mathcal{A}$ corresponds to finding a cost-optimal plan in $\mathcal{P}$. Notice that planning problems are deterministic by definition. Thus, the automata representing them are deterministic as well and paths and words are in a one to one correspondence. However, in general the manipulations of automata described at the end of this chapter destroy determinism. A plan in one of these automata will then be a word rather than a path.

2.1.2 Factored representation of planning problems

The factorization of planning problems we consider is given by a partition of the atoms of these planning problems. This means that factors will synchronize by their operators rather than by truth values of their atoms as it was the case with the factored representations of planning problems in Section 1.5.

Consider a planning problem $\mathcal{P} = (A, O, I, G)$ and a partition $A = A_1 \dot\cup \dots \dot\cup A_n$ of its set of atoms. This defines a factor $\mathcal{P}_k$ per set A_k of the partition as follows. $\mathcal{P}_k = (A_k, O_k, I_k, G_k)$ is such that $I_k = I \cap A_k$ is the restriction of the initial state to A_k, and $G_k = G \cap A_k$ is the restriction of the goal states to A_k. The operators in O_k are defined with names. In other words they are of the form $o = (\mathbf{id}_o, \mathbf{pre}_o, \mathbf{del}_o, \mathbf{add}_o)$ where $(\mathbf{pre}_o, \mathbf{del}_o, \mathbf{add}_o)$ is a standard operator and $\mathbf{id}_o$ is a unique name. This avoids unnecessary technical details for identifying correspondences between operators from different subproblems, in particular when several operators have the same preconditions and effects on a given component. The operators in O_k correspond to the operators in O having precondition or effects on A_k. The name of an operator in this set is the corresponding operator in O. So, several operators can have the same preconditions and effects but different names. Formally: $O_k = \{(o, \mathbf{pre}_o \cap A_k, \mathbf{del}_o \cap A_k, \mathbf{add}_o \cap A_k) \ : \ o \in O, (\mathbf{pre}_o \cup \mathbf{del}_o \cup \mathbf{add}_o) \cap A_k \neq \emptyset\}$.

Let us denote by $\mathbf{id}(p_k) = \mathbf{id}_{o_1} \dots \mathbf{id}_{o_\ell}$ the sequence of operator names corresponding to a plan $p_k = o_1 \dots o_\ell$ in $\mathcal{P}_k$. Let us also denote by $p_{|O'}$ the restriction of a plan $p \in O^*$ to $O' \subseteq O$, that is the sequence obtained from p by removing all operators not in O'. Finally, let us denote by O'_k the set of operators from O having preconditions or effects in A_k, that is $O'_k = \{o \ : \ (\mathbf{pre}_o \cup \mathbf{del}_o \cup \mathbf{add}_o) \cap A_k \neq \emptyset\}$,

one can notice this corresponds exactly to the names of the operators in O_k. One can then immediately remark the two following facts.

Property 2.1. *For any plan p in $\mathcal{P}$ there exists a unique plan p_k in each $\mathcal{P}_k$ such that $\mathbf{id}(p_k) = p_{|O'_k}$.*

Property 2.2. *For any sequence p' of operators from O which is not a plan in $\mathcal{P}$ there exists at least one component $\mathcal{P}_k$ such that no plan p_k verifying $\mathbf{id}(p_k) = p'_{|O'_k}$ exists.*

This allows a definition of compatibility between plans of different components, highly related to the notion of solution of factored planning problems.

Definition 2.3. *A tuple of plans $(p_1, \dots, p_n)$ for subproblems $\mathcal{P}_1, \dots, \mathcal{P}_n$ are said to be* compatible *if and only if there exists a sequence p of operators from O such that $\forall k, \mathbf{id}(p_k) = p_{|O'_k}$.*

For solving $\mathcal{P}$ in its factored form one then has to find a plan p_k in each $\mathcal{P}_k$ such that these plans are compatible. Any sequence p of operators such that $\forall k, \mathbf{id}(p_k) = p_{|O'_k}$ will necessarily be a plan in $\mathcal{P}$ from Properties 2.1 and 2.2. Figure 2.1 shows examples of compatible and non-compatible plans.

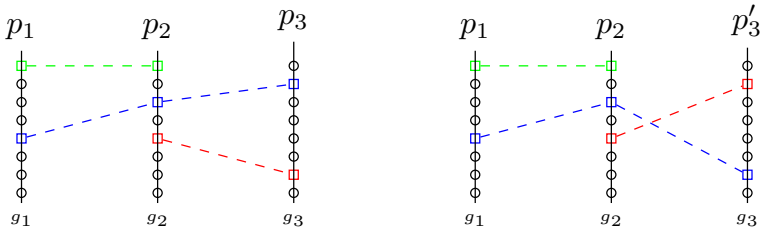


Figure 2.1: p_k are plans, circles are names of operators appearing only in one component, squares are names of operators appearing in two or three components. Dashed lines join names of operators which are the same. Left: the three plans are compatible. Right: p_2 and p'_3 are not compatible (the name of operators they share are not used in the same order).

If the considered planning problem $\mathcal{P}$ is provided with a cost function c, one will need to find a tuple of compatible plans minimizing the cost of the plans corresponding to this tuple (they all have the same cost as they are different orderings of the same operators). In other words, one has to find $(p_1, \dots, p_n)$ such that for any p verifying $\forall k, \mathbf{id}(p_k) = p_{|O'_k}$ the cost $c(p)$ is minimal. For achieving that one can split the cost $c(o)$ of each operator $o \in O$ between all the factors $\mathcal{P}_k$ containing an operator o_k with name o in any way ensuring that $\sum_k c_k(o_k) = c(o)$ (where c_k is the cost function of planning problem $\mathcal{P}_k$). Doing that one ensures that any tuple $(p_1, \dots, p_n)$ of compatible plans minimizing $\sum_k (c_k(p_k))$ gives a cost-optimal solution to $\mathcal{P}$ (because $\sum_k (c_k(p_k)) = c(p)$ for any p such that $\forall k, \mathbf{id}(p_k) = p_{|O'_k}$).

Finally, components are particular planning problems. So, they can be represented by weighted automata. In this case it is possible to simplify a bit the components by considering that the action on a transition corresponding to an operator $o = (\mathbf{id}_o, \mathbf{pre}_o, \mathbf{del}_o, \mathbf{add}_o)$ is $\mathbf{id}_o$ rather than o. This allow the following definition of factored planning problems in terms of weighted automata. This definition is the one we will be using in most of this document.

Definition 2.4 (factored cost-optimal planning problem)**.** *Given a collection of weighted automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$, find a tuple $((w_1, c_1), \ldots, (w_n, c_n)) \in \mathcal{L}(\mathcal{A}_1) \times \cdots \times \mathcal{L}(\mathcal{A}_n)$ such that there exists a word w in $(\cup_{1 \leq i \leq n} \Sigma_i)^*$ verifying $\forall 1 \leq k \leq n, w_{|\Sigma_k} = w_k$ and minimizing $\sum_{1 \leq i \leq n} c_i$.*

The first method we suggest for solving cost-optimal planning problems as defined above is based on the well-known message passing algorithms. In the next section we formally describe these algorithms in a general context. The remaining of this chapter is then dedicated to the instantiation of these generic algorithms to the particular case of planning.

2.2 Basics of message passing algorithms

Message passing algorithms are a family of algorithms first presented by Pearl in 1982 [77] under the name *belief propagation* algorithms. They consist in propagating information between levels of a hierarchical system with purpose of updating these levels and reach a fixpoint where each level says more about the system than previously. These algorithms are well known, in particular, in the domain of constraint satisfaction. In this section we describe a general framework for message passing algorithms which has been proposed before in Chapter 2 of [28].

Consider a finite set V_{max} of variables. Each variable $\mathrm{v} \in V_{max}$ taking values in a domain $\mathcal{D}_{\mathrm{v}}$. Also consider abstract systems defined with these variables. We usually denote them by $\mathcal{S}$. These systems are provided with two operations, called *composition* and *reduction*. The composition $\wedge$ is associative and commutative. The reduction of a system $\mathcal{S}$ to the subset $V \subseteq V_{max}$ of variables is denoted by $\Pi_V(\mathcal{S})$. These operations have to verify three axioms:

$$\forall V_1, V_2 \subseteq V_{max}, \forall \mathcal{S},\ \Pi_{V_1}(\Pi_{V_2}(\mathcal{S})) = \Pi_{V_1 \cap V_2}(\mathcal{S}), \tag{2.1}$$

which states that the reduction is in fact a *projection*.

$$\forall \mathcal{S}, \exists V \subseteq V_{max},\ \Pi_V(\mathcal{S}) = \mathcal{S}, \tag{2.2}$$

which permits to consider that the subset of variables over which $\mathcal{S}$ is defined is the smallest subset V of V_{max} such that $\Pi_V(\mathcal{S}) = \mathcal{S}$. We denote by V_i the subset of variables over which a system $\mathcal{S}_i$ is defined.

$$\forall \mathcal{S}_1, \mathcal{S}_2, \forall V_3 \supseteq V_1 \cap V_2,\ \Pi_{V_3}(\mathcal{S}_1 \wedge \mathcal{S}_2) = \Pi_{V_3}(\mathcal{S}_1) \wedge \Pi_{V_3}(\mathcal{S}_2), \tag{2.3}$$

which states that the interaction between two systems is fully captured by their shared variables. Moreover, we assume that composition admits an identity element $\mathbb{I}$:

$$\forall \mathcal{S},\ \mathcal{S} \wedge \mathbb{I} = \mathcal{S}. \tag{2.4}$$

As examples of such systems one can think to the following.

- Constraint systems, where each system $\mathcal{S}_i$ is a set of local states, that is a set of valuations of all the variables in V_i. Any local state in $\mathcal{S}_i$ defines a set of global states: one global state for each possible valuation of all the variables in $V_{max} \setminus V_i$. Using a representation of each system as the induced set of global states it is possible to define projection as a relaxation of constraints and composition as an intersection of systems.

- Constraint systems with costs, with the same definition for constraints but with a cost on each local state. These costs can be defined over any commutative semiring. As an example consider the $(\min, +)$ semiring. The cost of a state after projection will be the minimum over the costs of all states projecting in it. The cost of a state obtained by composition will be the sum of the costs of the two states composing it.
- Systems defined by regular languages and their alphabets. Projection will be natural projection and composition will be synchronous product. This is the setting used – without explicitly stating it – in [83] for example.

When dealing with systems defined by composition of smaller systems an important notion is the one of interaction graph. This graph describes how systems are related together.

Definition 2.5. *Given a compound system $\mathcal{S} = \mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n$ the (non-directed)* interaction graph *of this system is defined as: $G = (V, E)$, were $V = \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}$ and $E = \{(\mathcal{S}_i, \mathcal{S}_j) \mid i < j \wedge V_i \cap V_j \neq \emptyset\}$.*

In this graph an edge $(\mathcal{S}_i, \mathcal{S}_j)$ is said to be *redundant* if and only if there is a path $\mathcal{S}_i \mathcal{S}_{k_1} \ldots \mathcal{S}_{k_L} \mathcal{S}_j$ in G such that $V_i \cap V_j \subseteq V_{k_\ell}$ and $k_L \notin \{i, j\}$ for $1 \leq \ell \leq L$. By recursively removing *redundant* edges from G until no redundant edge remains one construct *communication graphs* of compound systems. Figure 2.2 shows an interaction graph and the corresponding communication graphs.

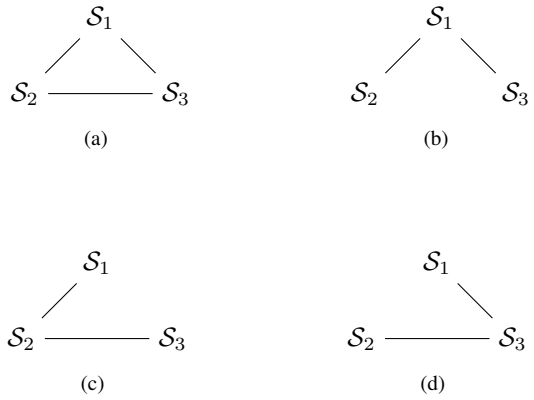


Figure 2.2: interaction graph of a system $\mathcal{S} = \mathcal{S}_1 \wedge \mathcal{S}_2 \wedge \mathcal{S}_3$ such that $V_1 \cap V_2 = V_2 \cap V_3 = V_3 \cap V_1 \neq \emptyset$ (a), and the three corresponding communication graphs (b) (c) (d).

Proposition 2.1 (Proposition 1 in [28])**.** *If any communication graph of a compound system is a tree then all communication graphs of this system are trees. In this case the system is said to* live on a tree.

On communication graphs it is possible to define Algorithm 6 which is of interest in particular when communication graphs are tree, as stated in Theorem 2.1. This algorithm is called a *message passing algorithm* (MPA). It works on any communication graph $G = (V, E)$ of a compound system $\mathcal{S} = \mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n$. In this algorithm,

$\mathcal{N}(\mathcal{S}_i)$ denotes the set containing all neighbors of $\mathcal{S}_i$ in G. The first loop is just an initialization step. The second loop is the core of the algorithm. The idea is that each sub-system $\mathcal{S}_i$ propagates its knowledge of the compound system $\mathcal{S}$ to all its neighbors $\mathcal{S}_j$ using messages $\mathcal{M}_{i,j}$. When the messages no longer contain new information the algorithm ends. The third loop computes the new systems $\mathcal{S}'_i$ using the messages computed before.

Algorithm 6 message passing algorithm

1: **for all** $(\mathcal{S}_i, \mathcal{S}_j) \in E$ **do**
2: $\quad \mathcal{M}_{i,j} \leftarrow \mathbb{I}$
3: **end for**
4: **repeat**
5: $\quad$ select $(\mathcal{S}_i, \mathcal{S}_j) \in E$
6: $\quad \mathcal{M}_{i,j} \leftarrow \Pi_{V_i \cap V_j}(\mathcal{S}_i \wedge (\wedge_{\mathcal{S}_k \in \mathcal{N}(\mathcal{S}_i) \setminus \{\mathcal{S}_j\}} \mathcal{M}_{k,i}))$
7: **until** stability of messages
8: **for all** $\mathcal{S}_i \in V$ **do**
9: $\quad \mathcal{S}'_i = \mathcal{S}_i \wedge (\wedge_{\mathcal{S}_k \in \mathcal{N}(\mathcal{S}_i)} \mathcal{M}_{k,i})$
10: **end for**

Theorem 2.1 (Theorem 1 in [28]). *If $\mathcal{S} = \mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n$ lives on a tree, then Algorithm 6 converges in finitely many steps on any communication graph of $\mathcal{S}$, and at convergence $\mathcal{S}'_i = \Pi_{V_i}(\mathcal{S})$, $\forall 1 \leq i \leq n$.*

Intuitively $\Pi_{V_i}(\mathcal{S})$ is a refinement of the system $\mathcal{S}_i$ which exactly describes the behavior of this system inside the compound system $\mathcal{S} = \mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n$. This makes this object of great interest for understanding the behavior of $\mathcal{S}$ without computing it (in other words, by performing only computations local to components). Back to some of the previous examples one would have the following.

- For constraint systems, $\Pi_{V_i}(\mathcal{S})$ gives exactly the local states of $\mathcal{S}_i$ which are part of a global state of $\mathcal{S}$.
- For constraint systems with costs over $(\min, +)$ semiring, $\Pi_{V_i}(\mathcal{S})$ gives the local states of $\mathcal{S}_i$ which are part of a global state of $\mathcal{S}$. Each of these local states has for cost the minimal cost among the global states in which it takes part.

Notice that, when systems live on trees, it is possible to ensure convergence with exactly one update of each message. The idea is to schedule the message updates: a message should be updated only when all the other messages taking part in its computation were already updated and no message should be updated twice (it is always possible to find an updatable message while the algorithm did not converge). This is what Algorithm 7 does. A possible scheduling of message updates obtained with Algorithm 7 is depicted in Figure 2.3.

To summarize, the message passing algorithms allow, given a compound system $\mathcal{S} = \mathcal{S}_1 \wedge \ldots \wedge \mathcal{S}_n$ living on a tree, to compute the projection of this system on the variables of each of its component $\Pi_{V_i}(S)$, without computing the whole system. Moreover, this algorithm only requires two operations which follow the axioms presented in Equations 2.1, 2.2, and 2.3 [28]. In the following we relate this notion of compound system to factored planning and show how this algorithm can be used to perform factored cost-optimal planning by instantiating systems, composition, and projection.

Algorithm 7 MPA with efficient message update scheduling

for all $(\mathcal{S}_i, \mathcal{S}_j) \in E$ **do**
  $\mathcal{M}_{i,j} \leftarrow \mathbb{I}$
end for
repeat
  select $(\mathcal{S}_i, \mathcal{S}_j) \in E$ such that $\mathcal{M}_{i,j}$ not updated and $\forall \mathcal{S}_k \in \mathcal{N}(\mathcal{S}_i) \setminus \{\mathcal{S}_j\}, \mathcal{M}_{k,i}$ was updated before
  $\mathcal{M}_{i,j} \leftarrow \Pi_{V_i \cap V_j}(\mathcal{S}_i \wedge (\wedge_{\mathcal{S}_k \in \mathcal{N}(\mathcal{S}_i) \setminus \{\mathcal{S}_j\}} \mathcal{M}_{k,i}))$
until all messages were updated exactly once
for all $\mathcal{S}_i \in V$ **do**
  $\mathcal{S}'_i = \mathcal{S}_i \wedge (\wedge_{\mathcal{S}_k \in \mathcal{N}(\mathcal{S}_i)} \mathcal{M}_{k,i})$
end for

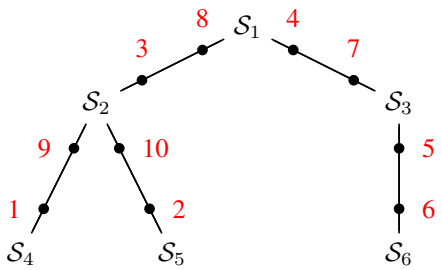


Figure 2.3: A communication graph which is a tree, the dots represent messages (for example the dot numbered 1 is message $\mathcal{M}_{4,2}$) and the numbers correspond to an ordering of updates which ensures stability in one update per message.

2.3 Message passing for cost-optimal planning

Consider a planning problem in factored form given as in Definition 2.4. The set of plans in each component $\mathcal{A}_k$ is the weighted language $\mathcal{L}_k = \mathcal{L}(\mathcal{A}_k)$. In this section we denote by $\mathcal{L}$ the weighted language containing all words corresponding to tuples of compatible words solution of the factored planning problem associated with their cost: $\mathcal{L} = \{(w, c) \ : \ \forall k, (w_{|\Sigma_k}, c_k) \in \mathcal{L}_k \wedge c = \sum_k c_k\}$. We consider that a weighted language and its set of actions is a system. For example $(\mathcal{L}, \cup_k \Sigma_k)$ is a system, as well as $(\mathcal{L}_k, \Sigma_k)$ for each k. One can then define a notion of composition of weighted languages ($\times_{\mathcal{L}}$) allowing to define $(\mathcal{L}, \cup_k \Sigma_k)$ as the compound system $(\mathcal{L}_1, \Sigma_1) \times_{\mathcal{L}} \ldots \times_{\mathcal{L}} (\mathcal{L}_n, \Sigma_n)$.

2.3.1 Composition: synchronous product

The composition we consider is in fact the standard synchronous product of languages, taking costs into account by summation.

Definition 2.6. *The* product $(\mathcal{L}_{1,2}, \Sigma_{1,2}) = (\mathcal{L}_1, \Sigma_1) \times_{\mathcal{L}} (\mathcal{L}_2, \Sigma_2)$ *of two weighted languages* $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *with set of actions* Σ_1 *and* Σ_2 *is defined as follows:* $\Sigma_{1,2} = \Sigma_1 \cup \Sigma_2$*, and* $\mathcal{L}_{1,2} = \{(w, c) \ : \ (w_{|\Sigma_1}, c_1) \in \mathcal{L}_1 \wedge (w_{|\Sigma_2}, c_2) \in \mathcal{L}_2 \wedge c = c_1 + c_2\}$.

As we want to use this product in the setting defined in Section 2.2, we have to ensure that it is associative and commutative.

Proposition 2.2. $\times_{\mathcal{L}}$ *is associative:* $((\mathcal{L}_1, \Sigma_1) \times_{\mathcal{L}} (\mathcal{L}_2, \Sigma_2)) \times_{\mathcal{L}} (\mathcal{L}_3, \Sigma_3) = (\mathcal{L}_1, \Sigma_1) \times_{\mathcal{L}} ((\mathcal{L}_2, \Sigma_2) \times_{\mathcal{L}} (\mathcal{L}_3, \Sigma_3))$.

Proof. By definition one has $(\mathcal{L}_1, \Sigma_1) \times_{\mathcal{L}} (\mathcal{L}_2, \Sigma_2) = (\{(w, c) \ : \ (w_{|\Sigma_1}, c_1) \in \mathcal{L}_1 \wedge (w_{|\Sigma_2}, c_2) \in \mathcal{L}_2 \wedge c = c_1 + c_2\}, \Sigma_1 \cup \Sigma_2) = (\mathcal{L}_{1,2}, \Sigma_{1,2})$. Thus, applying a second time the definition, $((\mathcal{L}_1, \Sigma_1) \times_{\mathcal{L}} (\mathcal{L}_2, \Sigma_2)) \times_{\mathcal{L}} (\mathcal{L}_3, \Sigma_3) = (\{(w, c) \ : \ (w_{|\Sigma_{1,2}}, c_{1,2}) \in \mathcal{L}_{1,2} \wedge (w_{|\Sigma_3}, c_3) \in \mathcal{L}_3 \wedge c = c_{1,2} + c_3\}, \Sigma_{1,2} \cup \Sigma_3) = (\{(w, c) \ : \ (w_{|\Sigma_1}, c_1) \in \mathcal{L}_1 \wedge (w_{|\Sigma_2}, c_2) \in \mathcal{L}_2 \wedge (w_{|\Sigma_3}, c_3) \in \mathcal{L}_3 \wedge c = c_1 + c_2 + c_3\}, \Sigma_1 \cup \Sigma_2 \cup \Sigma_3)$. Using a similar reasoning one finds the same result for $(\mathcal{L}_1, \Sigma_1) \times_{\mathcal{L}} ((\mathcal{L}_2, \Sigma_2) \times_{\mathcal{L}} (\mathcal{L}_3, \Sigma_3))$. Which proves associativity of $\times_{\mathcal{L}}$. □

Proposition 2.3. $\times_{\mathcal{L}}$ *is commutative:* $(\mathcal{L}_1, \Sigma_1) \times_{\mathcal{L}} (\mathcal{L}_2, \Sigma_2) = (\mathcal{L}_2, \Sigma_2) \times_{\mathcal{L}} (\mathcal{L}_1, \Sigma_1)$.

Proof. Proof is a direct application of the definition of $\times_{\mathcal{L}}$. □

One also needs an identity element for product (axiom 2.4). $\mathcal{L}_{\mathbb{I}} = (\{(\epsilon, 0)\}, \emptyset)$ is an acceptable identity element.

Remark 2.1. *From this definition of product, one immediately has for any factored planning problem that* $(\mathcal{L}, \cup_k \Sigma_k) = (\mathcal{L}_1, \Sigma_1) \times_{\mathcal{L}} \ldots \times_{\mathcal{L}} (\mathcal{L}_n, \Sigma_n)$*. Where the* $\mathcal{L}_k$ *are the weighted languages giving the plans in each component with their minimal cost. And* $\mathcal{L}$*, as defined above contains all words obtained from tuples of compatible words in the* $\mathcal{L}_k$*, with their cost.*

2.3.2 Projection: natural projection

In order to fit with the setting of Section 2.2, one also needs a notion of projection for weighted languages. The operation we suggest is in fact the natural projection of languages associated with a cost minimization.

Definition 2.7. *The* projection $\Pi_{\Sigma'}((\mathcal{L},\Sigma))$ *of a weighted language* $(\mathcal{L},\Sigma)$ *over an alphabet* Σ' *is the language* $(\mathcal{L}',\Sigma')$ *such that:* $\mathcal{L}' = \{(w_{|\Sigma'}, c) \ : \ (w,c) \in \mathcal{L} \wedge c = \min_{(w',c')\in\mathcal{L}, w'_{|\Sigma'}=w_{|\Sigma'}} c'\}$.

One can then prove that this projection is effectively a projection as defined in Section 2.2 by showing that is satisfies the two required axioms.

Proposition 2.4. $\forall(\mathcal{L},\Sigma),\Sigma_1,\Sigma_2, \Pi_{\Sigma_1}(\Pi_{\Sigma_2}((\mathcal{L},\Sigma))) = \Pi_{\Sigma_1\cap\Sigma_2}((\mathcal{L},\Sigma))$*, that is* Π *verifies axiom 2.1.*

Proof. Let $(\mathcal{L},\Sigma)$ be a weighted language with its alphabet and let Σ_1 and Σ_2 be two alphabets. Note $\Pi_{\Sigma_1}(\Pi_{\Sigma_2}((\mathcal{L},\Sigma))) = (\mathcal{L}',\Sigma')$ and $\Pi_{\Sigma_1\cap\Sigma_2}((\mathcal{L},\Sigma)) = (\mathcal{L}'',\Sigma'')$. One has $\Sigma' = \Sigma'' = (\Sigma_1\cap\Sigma_2)\cap\Sigma$. Moreover, for a given $(w,c)\in\mathcal{L}$, one has $(w_{|\Sigma_1})_{|\Sigma_2} = w_{|\Sigma_1\cap\Sigma_2}$. This shows that $\mathcal{L}'$ and $\mathcal{L}''$ contain the same words. The remaining is to proove that, for any word w', if $(w',c')\in\mathcal{L}'$ and $(w',c'')\in\mathcal{L}''$, then $c' = c''$. First remark that, because $(w_{|\Sigma_1})_{|\Sigma_2} = w_{|\Sigma_1\cap\Sigma_2}$ for any $(w,c)\in\mathcal{L}$, the sets of words giving (w',c') and (w',c'') are the same. Denote this set by W. To conclude the proof, it is sufficient to notice that:

$$c' = \min_{(w,c)\in\mathcal{L}, w\in W} c = c''.$$

□

Proposition 2.5. $\forall(\mathcal{L},\Sigma),\exists\Sigma', \Pi_{\Sigma'}((\mathcal{L},\Sigma)) = (\mathcal{L},\Sigma)$*, that is* Π *verifies axiom 2.2.*

Proof. Let $(\mathcal{L},\Sigma)$ be a weighted language. Take $\Sigma' = \Sigma$. Note $\Pi_{\Sigma'}((\mathcal{L},\Sigma)) = (\mathcal{L}',\Sigma')$ In this case, $\mathcal{L}$ and $\mathcal{L}'$ contain the same words as, for any word w defined over $\Sigma = \Sigma'$ it is clear that $w_{|\Sigma} = w$. From that one also knows that for $(w,c)\in\mathcal{L}$ and $(w,c')\in\mathcal{L}'$, $c = c'$ as each word in $\mathcal{L}'$ corresponds to a single word in $\mathcal{L}$. □

2.3.3 Relation between product and projection

We finally show that the product and the projection defined above verify the axiom 2.3. After that we relate these product and projection to the message passing algorithm of Section 2.2, and to factored planning problems.

Proposition 2.6. $\forall(\mathcal{L}_1,\Sigma_1),\forall(\mathcal{L}_2,\Sigma_2),\forall\Sigma_3\supseteq\Sigma_1\cap\Sigma_2, \Pi_{\Sigma_3}((\mathcal{L}_1,\Sigma_1)\times_{\mathcal{L}}(\mathcal{L}_2,\Sigma_2)) = \Pi_{\Sigma_3}((\mathcal{L}_1,\Sigma_1))\times_{\mathcal{L}}\Pi_{\Sigma_3}((\mathcal{L}_2,\Sigma_2))$*, that is* $\times_{\mathcal{L}}$ *and* Π *verify axiom 2.3.*

Proof. Let $(\mathcal{L}_1,\Sigma_1)$ and $(\mathcal{L}_2,\Sigma_2)$ be two weighted languages with their alphabets, and let Σ_3 be an alphabet such that $\Sigma_3\supseteq\Sigma_1\cap\Sigma_2$. Note $\Pi_{\Sigma_3}((\mathcal{L}_1,\Sigma_1)\times_{\mathcal{L}}(\mathcal{L}_2,\Sigma_2)) = (\mathcal{L},\Sigma)$ and $\Pi_{\Sigma_3}((\mathcal{L}_1,\Sigma_1))\times_{\mathcal{L}}\Pi_{\Sigma_3}((\mathcal{L}_2,\Sigma_2)) = (\mathcal{L}',\Sigma')$. It is clear that $\Sigma = \Sigma' = \Sigma_3\cap(\Sigma_1\cup\Sigma_2)$, by definition of product and projection. Also note $\Pi_{\Sigma_3}((\mathcal{L}_1,\Sigma_1)) = (\mathcal{L}_1^3,\Sigma_1^3)$ and $\Pi_{\Sigma_3}((\mathcal{L}_2,\Sigma_2)) = (\mathcal{L}_2^3,\Sigma_2^3)$. And note $(\mathcal{L}_1,\Sigma_1)\times_{\mathcal{L}}(\mathcal{L}_2,\Sigma_2) = (\mathcal{L}_{1,2},\Sigma_{1,2})$.

First prove that $\mathcal{L}$ and $\mathcal{L}'$ contain the same words: $\forall(w,c)\in\mathcal{L},\exists c',(w,c')\in\mathcal{L}'$ and $\forall(w,c')\in\mathcal{L}',\exists c,(w,c)\in\mathcal{L}$. An intuition of the proof is given in Figure 2.4. This figure shows two words w_1 belonging to $\mathcal{L}_1$ and w_2 belonging to $\mathcal{L}_2$ and the part of the words of $\mathcal{L}$ and $\mathcal{L}'$ they generate. Squares depict elements of $\Sigma_1\cap\Sigma_2$. Notice that all these elements are in Σ_3. Dashed lines depict synchronizations between these elements. White circles depict the other elements of Σ_3 and black circles the elements of Σ_1 and Σ_2 which are not in Σ_3. The sign $\|$ between two parts of words means "all possible interleavings of these parts".

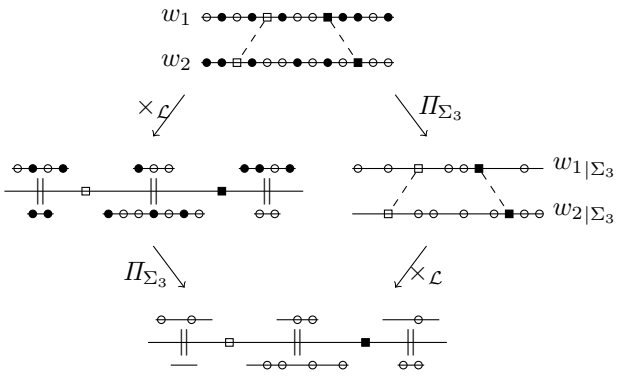


Figure 2.4: Product and projection of words. Top are two words. Middle left is a representation of their product (which is in fact a set of words). Middle right are the restrictions of these words to Σ_3. Down is a representation of the restriction to Σ_3 of their product, which is also the product of their restrictions to Σ_3.

Let $(w, c) \in \mathcal{L}$, one has by definition of projection that $\exists (w_{1,2}, c) \in \mathcal{L}_{1,2}$ such that $w_{1,2|\Sigma_3} = w$. Thus, by definition of product, $\exists (w_1, c_1) \in \mathcal{L}_1$ and $\exists (w_2, c_2) \in \mathcal{L}_2$ such that $w_{1,2|\Sigma_1} = w_1$ and $w_{1,2|\Sigma_2} = w_2$. Then, by definition of projection, $\exists c, (w_{1|\Sigma_3}, c) \in \mathcal{L}_1^3$ and $\exists c, (w_{2|\Sigma_3}, c) \in \mathcal{L}_2^3$. Then remark that $w_{1|\Sigma_3} = w_{1,2|\Sigma_1|\Sigma_3} = w_{1,2|\Sigma_3|\Sigma_1} = w_{1,2|\Sigma_3|\Sigma_1\cap\Sigma_3} = w_{|\Sigma_1\cap\Sigma_3}$ and similarly $w_{2|\Sigma_3} = w_{|\Sigma_2\cap\Sigma_3}$. By definition of product this proves that $\exists c', (w, c') \in \mathcal{L}'$. It concludes the proof that $\forall (w, c) \in \mathcal{L}, \exists c', (w, c') \in \mathcal{L}'$.

Let $(w, c') \in \mathcal{L}'$, one has, by definition of $\mathcal{L}'$, $w = w_1^1 w_2^1 \sigma_3^1 \dots \sigma_3^{n-1} w_1^n w_2^n$ with $w_1^i \in ((\Sigma_1 \setminus \Sigma_2) \cap \Sigma_3)^*$, $w_2^i \in ((\Sigma_2 \setminus \Sigma_1) \cap \Sigma_3)^*$, and $\sigma_3^i \in \Sigma_1 \cap \Sigma_2 \cup \{\varepsilon\}$ for all i. By definition of the product one has $w_{|\Sigma_1\cap\Sigma_3} = w_1^1 \sigma_3^1 w_1^2 \sigma_3^2 \dots w_1^n$ is a word in $\mathcal{L}_1^3$. Similarly $w_{|\Sigma_2\cap\Sigma_3} = w_2^1 \sigma_3^1 w_2^2 \sigma_3^2 \dots w_2^n$ is a word in $\mathcal{L}_2^3$. Thus, by definition of the projection, there exists a word $w_1 = w_1^{1'} \sigma_3^1 w_1^{2'} \sigma_3^2 \dots w_1^{n'}$ in $\mathcal{L}_1$ such that $w_{1|\Sigma_3}^{i'} = w_1^i$ for all i. Similarly there exists a word $w_2 = w_2^{1'} \sigma_3^1 w_2^{2'} \sigma_3^2 \dots w_2^{n'}$ in $\mathcal{L}_2$ such that $w_{2|\Sigma_3}^{i'} = w_2^i$ for all i. As $\Sigma_3 \supseteq \Sigma_1 \cap \Sigma_2$, there is a word $w' = w_1^{1'} w_2^{1'} \sigma_3^1 \dots \sigma_3^{n-1} w_1^{n'} w_2^{n'}$ in $\mathcal{L}_{1,2}$. Thus $w'_{|\Sigma_3}$ is a word in $\mathcal{L}$. And $w'_{|\Sigma_3} = w_{1|\Sigma_3}^{1'} w_{2|\Sigma_3}^{1'} \sigma_3^1 \dots \sigma_3^{n-1} w_{1|\Sigma_3}^{n'} w_{2|\Sigma_3}^{n'} = w_1^1 w_2^1 \sigma_3^1 \dots \sigma_3^{n-1} w_1^n w_2^n = w$. This proves that $\forall (w, c') \in \mathcal{L}', \exists c, (w, c) \in \mathcal{L}$.

Now prove that, for $(w, c) \in \mathcal{L}$ and $(w, c') \in \mathcal{L}'$, $c = c'$. The first step is to show $c \leq c'$. By definition of product, it is known that $c' = c_1^3 + c_2^3$ for some $(w_1^3, c_1^3) \in \mathcal{L}_1^3$ and some $(w_2^3, c_2^3) \in \mathcal{L}_2^3$. By definition of projection it is known that $c_1^3 = c_1$ for some $(w_1, c_1) \in \mathcal{L}_1$ and $c_2^3 = c_2$ for some $(w_2, c_2) \in \mathcal{L}_2$. Moreover, $w_{1|\Sigma_3} = w_1^3$ and $w_{2|\Sigma_3} = w_2^3$. As $\Sigma_3 \supseteq \Sigma_1 \cap \Sigma_2$, it is known that there is $(w_{1,2}, c_{1,2}) \in \mathcal{L}_{1,2}$ such that $w_{1,2|\Sigma_1} = w_1$ and $w_{1,2|\Sigma_2} = w_2$. Thus, by definition of projection, it is known that $c \leq c_{1,2}$. Moreover, by definition of product, $c_{1,2} = c_1 + c_2 = c_1^3 + c_2^3 = c'$. Finally, $c \leq c'$.

The remaining is to prove that $c' \leq c$. It is known, by definition of product and projection, that there is $(w_1, c_1) \in \mathcal{L}_1$ and $(w_2, c_2) \in \mathcal{L}_2$ such that $c = c_1 + c_2$. Moreover, it is known that $(w, c') \in \mathcal{L}'$, so, $(w_{|\Sigma_1}, c_1^3), \in \mathcal{L}_1^3$ and $(w_{|\Sigma_2}, c_2^3) \in \mathcal{L}_2^3$, with $c' = c_1^3 + c_2^3$. One has, in particular, that $w_{1|\Sigma_3} = w_{|\Sigma_1}$ and $w_{2|\Sigma_3} = w_{|\Sigma_2}$. By

definition of projection, $c_1 \geq c_1^3$ and $c_2 \geq c_2^3$. Hence, $c' = c_1^3 + c_2^3 \leq c_1 + c_2 = c$. Finally $c' \leq c$. It has been proved that $c \leq c'$ and $c' \leq c$. Hence, $c = c'$. □

Product ($\times_{\mathcal{L}}$) and projection ($\mathit{\Pi}$) of weighted languages have been proved to verify the axioms described in equations 2.1, 2.2, and 2.3. Thus, given a compound system $(\mathcal{L}, \Sigma) = (\mathcal{L}_1, \Sigma_1) \times_{\mathcal{L}} \ldots \times_{\mathcal{L}} (\mathcal{L}_n, \Sigma_n)$, if the communication graphs of this system (defined as in Section 2.2, with languages as systems and alphabets as variables) are trees, then the message passing algorithm converges on any of these graphs. The outcome of the algorithm is a language $\mathcal{L}'_k$ for each $\mathcal{L}_k$, such that $(\mathcal{L}'_k, \Sigma_k) = \mathit{\Pi}_{\Sigma_k}((\mathcal{L}, \Sigma))$. By definition of projection, the two following properties hold for these $\mathcal{L}'_i$:

Property 2.3. *Any $(w_k, c_k) \in \mathcal{L}'_k$ such that c_k is minimal (that is such that there exists no $(w'_k, c'_k) \in \mathcal{L}'_k$ with $c_{k'} < c_k$) is the projection of some minimal cost word in $\mathcal{L}$, in the following sense: $\exists (w, c) \in \mathcal{L}$ such that $w_{|\Sigma_k} = w_k$ and c is minimal. Moreover, $c = c_k$.*

Property 2.4. *Any $(w, c) \in \mathcal{L}$ such that c is minimal can be projected into a minimal cost word in $\mathcal{L}'_k$, in the following sense: $\exists (w_k, c_k) \in \mathcal{L}_k$ such that $w_k = w_{|\Sigma_k}$ and c_k is minimal. Moreover, $c_k = c$.*

These two properties allow one to derive a method to compute a cost-optimal word w in $\mathcal{L}$ using only $\mathcal{L}'_1, \ldots, \mathcal{L}'_n$. This method consists in building $w_{|\Sigma_k}$ in each $\mathcal{L}'_k$ in an order following the communication graph considered.

1. Select a cost-optimal word $(w_i, c) \in \mathcal{L}'_i$ for any i, Property 2.3 ensures that this word is extendable into a cost-optimal word in $\mathcal{L}$.

2. Chose a neighbor $\mathcal{L}_j$ of $\mathcal{L}_i$ in the communication graph and select a cost-optimal word $(w_j, c) \in \mathcal{L}'_j$ which is compatible with w_i. There exists one by Property 2.4 and it is part of a cost-optimal word in $\mathcal{L}$ by Property 2.3. Notice that w_j is any cost-optimal word in $(\mathcal{L}'_j, \Sigma_j) \times_{\mathcal{L}} (\{(w_i, 0)\}, \Sigma_i)$.

3. Chose a neighbor $\mathcal{L}_k$ of $\mathcal{L}_\ell$ in the communication graph, where $\mathcal{L}_\ell$ is any language previously considered (that is, such that w_ℓ has already been selected). Select a cost-optimal word $(w_k, c) \in \mathcal{L}'_k$ which is compatible with w_ℓ. As the communication graph is a tree, w_k is compatible with all the previously selected words.

4. repeat 3 until a word has been selected in each component.

Any interleaving of $w_1, \ldots, w_n$ gives a cost-optimal word in $\mathcal{L}$ and the cost of this word is c.

By Remark 2.1 this immediately gives a method for solving planning problems as described in Definition 2.4. For each component $\mathcal{A}_k$ of a planning problem take its language $\mathcal{L}_k$. Then, from these languages compute the languages $\mathcal{L}'_k$ using the message passing algorithm. Finally, extract the words $w_1, \ldots, w_n$ from these $\mathcal{L}'_k$ as explained above. The tuple $((w_1, c_1), \ldots, (w_n, c_n))$ where c_i is the cost of w_i in $\mathcal{L}_i$ is a solution to the problem presented in Definition 2.4. We conclude this section by giving a sample execution of our planning method on languages.

2.3.4 Sample execution of the MPA on weighted languages

Consider three weighted languages $\mathcal{L}_1$, $\mathcal{L}_2$, and $\mathcal{L}_3$, over alphabets $\Sigma_1 = \{a, \alpha\}$, $\Sigma_2 = \{b, \alpha, \beta\}$, and $\Sigma_3 = \{c, \beta\}$. Their interaction graph (which is also a communication graph) is depicted in Figure 2.5. It is clearly a tree, the message passing algorithm will converge on this graph.

$$\mathcal{L}_1 \overset{\{\alpha\}}{\text{———}} \mathcal{L}_2 \overset{\{\beta\}}{\text{———}} \mathcal{L}_3$$

Figure 2.5: interaction graph of $\mathcal{L}_1$, $\mathcal{L}_2$, and $\mathcal{L}_3$.

Consider that:

$$\mathcal{L}_1 = \{(aa\alpha aa, 1), (\alpha aa\alpha, 0.5), (a\alpha a, 1.5)\},$$

$$\mathcal{L}_2 = \{(b\alpha\beta b, 0.5), (bb\alpha\beta\beta, 2), (\alpha b\beta b\alpha\alpha, 1.5), (\alpha\alpha\beta bb\beta, 1)\}, \text{ and}$$

$$\mathcal{L}_3 = \{(\beta\beta ccc\beta, 0.5), (\beta cc\beta, 1), (\beta\beta c\beta, 1.5), (cc\beta, 1.5)\}.$$

The first message to compute is $\mathcal{M}_{1,2} = \Pi_{\Sigma_2 \cap \Sigma_1}((\mathcal{L}_1, \Sigma_1))$. This message is a weighted language $(\mathcal{L}_{1,2}, \Sigma_{1,2})$, where $\mathcal{L}_{1,2} = \{(\alpha, 1), (\alpha\alpha, 0.5)\}$ and $\Sigma_{1,2} = \{\alpha\}$. The second message is $\mathcal{M}_{2,3} = \Pi_{\Sigma_3 \cap \Sigma_2}(((\mathcal{L}_2, \Sigma_2) \times_{\mathcal{L}} \mathcal{M}_{1,2}, \Sigma_2))$. First one computes $(\mathcal{L}_2, \Sigma_2) \times_{\mathcal{L}} \mathcal{M}_{1,2} = (\mathcal{L}, \Sigma)$ where $\mathcal{L} = \{(b\alpha\beta b, 1.5), (bb\alpha\beta\beta, 3), (\alpha\alpha\beta bb\beta, 1.5)\}$ and $\Sigma = \Sigma_2$. Then this language has to be projected on Σ_3, which leads to a language $(\mathcal{L}_{2,3}, \Sigma_{2,3})$ such that $\mathcal{L}_{2,3} = \{(\beta, 1.5), (\beta\beta, 1.5)\}$ and $\Sigma_{2,3} = \{\beta\}$. Then message $\mathcal{M}_{3,2} = \Pi_{\Sigma_2 \cap \Sigma_3}((\mathcal{L}_3, \Sigma_3)) = (\mathcal{L}_{3,2}, \Sigma_{3,2})$ is computed, and finally $\mathcal{M}_{2,1} = \Pi_{\Sigma_1 \cap \Sigma_2}(((\mathcal{L}_2, \Sigma_2) \times_{\mathcal{L}} \mathcal{M}_{3,2}, \Sigma_2)) = (\mathcal{L}_{2,1}, \Sigma_{2,1})$. One has $\Sigma_{3,2} = \{\beta\}$ and $\mathcal{L}_{3,2} = \{(\beta, 1.5), (\beta\beta, 1), (\beta\beta\beta, 0.5)\}$. And $\mathcal{L}_{2,1} = \{(\alpha, 2), (\alpha\alpha, 2), (\alpha\alpha\alpha, 3)\}$ with $\Sigma_{2,1} = \{\alpha\}$.

From that one can compute $(\mathcal{L}'_1, \Sigma_1) = (\mathcal{L}_1, \Sigma_1) \times_{\mathcal{L}} \mathcal{M}_{2,1}$, $(\mathcal{L}'_2, \Sigma_2) = (\mathcal{L}_2, \Sigma_2) \times_{\mathcal{L}} \mathcal{M}_{1,2} \times_{\mathcal{L}} \mathcal{M}_{3,2}$, and $(\mathcal{L}'_3, \Sigma_3) = \mathcal{L}_3 \times_{\mathcal{L}} \mathcal{M}_{2,3}$. It gives:

$$\mathcal{L}'_1 = \{(aa\alpha aa, 3), (\alpha aa\alpha, 2.5), (a\alpha a, 3.5)\},$$

$$\mathcal{L}'_2 = \{(b\alpha\beta b, 3), (bb\alpha\beta\beta, 4), (\alpha\alpha\beta bb\beta, 2.5)\}, \text{ and}$$

$$\mathcal{L}'_3 = \{(\beta cc\beta, 2.5), (cc\beta, 3)\}.$$

One can check that $\alpha aa\alpha$, $\alpha\alpha\beta bb\beta$, and $\beta cc\beta$ are minimal cost words (resp. in $\mathcal{L}'_1$, $\mathcal{L}'_2$, and $\mathcal{L}'_3$) each with cost 2.5. Moreover these words are clearly compatible, they can, for example give the word $\alpha aa\alpha\beta bcbc\beta$, which is effectively a cost-optimal word from $(\mathcal{L}, \Sigma_1 \cup \Sigma_2 \cup \Sigma_3) = (\mathcal{L}_1, \Sigma_1) \times_{\mathcal{L}} (\mathcal{L}_2, \Sigma_2) \times_{\mathcal{L}} (\mathcal{L}_3, \Sigma_3)$ with cost 2.5.

2.4 Working directly with weighted automata

The method presented in previous section is not applicable in practice because the set of all plans of any sub-problem may contain an infinite number of elements. One thus needs to use a finite encoding of weighted languages. Fortunately, planning problems can be represented by weighted automata. In this section we show that it is possible to directly run message passing algorithm using weighted automata as systems. This is due to the fact that all the operations on weighted languages can be implemented

directly on weighted automata. In particular, this will allow to use the algorithms presented in [76] for implementing message passing for cost-optimal planning in practice. This practical implementation is the topic of Chapter 3.

2.4.1 Plan compatibility: product of weighted automata

We first define a notion of product for weighted automata and show that this product is an implementation of the product of languages. Figure 2.6 represents two weighted automata and their product.

Definition 2.8. *The product* $\mathcal{A}_1 \times_{\mathcal{A}} \mathcal{A}_2 = (S, S^I, S^F, \Sigma, T, c, c^i, c^f)$ *of two weighted automata* $\mathcal{A}_1 = (S_1, S_1^I, S_1^F, \Sigma_1, T_1, c_1, c_1^i, c_1^f)$ *and* $\mathcal{A}_2 = (S_2, S_2^I, S_2^F, \Sigma_2, T_2, c_2, c_2^i, c_2^f)$ *is such that:* $S = S_1 \times S_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $S^I = S_1^I \times S_2^I$, $S^F = S_1^F \times S_2^F$,

$$\begin{aligned} T \quad = \quad & \{((s_1, s_2), \sigma, (s_1', s_2')) \mid (s_1, \sigma, s_1') \in T_1 \\ & \qquad \wedge \ (s_2, \sigma, s_2') \in T_2\} \\ \cup \quad & \{((s_1, s_2), \sigma, (s_1', s_2)) \mid (s_1, \sigma, s_1') \in T_1 \\ & \qquad \wedge \ \sigma \notin \Sigma_2\} \\ \cup \quad & \{((s_1, s_2), \sigma, (s_1, s_2')) \mid (s_2, \sigma, s_2') \in T_2 \\ & \qquad \wedge \ \sigma \notin \Sigma_1\}, \end{aligned}$$

and, for $t = ((s_1, s_2), \sigma, (s_1', s_2')) \in T$, *if* $\sigma \in \Sigma_1 \cap \Sigma_2$, *then* $c(t) = c_1((s_1, \sigma, s_1')) + c_2((s_2, \sigma, s_2'))$, *if* $\sigma \in \Sigma_1 \setminus \Sigma_2$, *then* $c(t) = c_1(s_1, \sigma, s_1')$, *if* $\sigma \in \Sigma_2 \setminus \Sigma_1$, *then* $c(t) = c_2((s_2, \sigma, s_2'))$. *For* $(s_1, s_2) \in S^I$, $c^i((s_1, s_2)) = c_1^i(s_1) + c_2^i(s_2)$, *and for* $(s_1, s_2) \in S^F$, $c^f((s_1, s_2)) = c_1^f(s_1) + c_2^f(s_2)$.

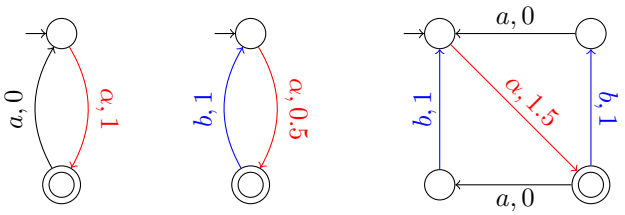


Figure 2.6: two weighted automata (left) and their product (right)

Proposition 2.7. *Product of weighted automata implements product of weighted languages:* $\forall \mathcal{A}_1, \mathcal{A}_2$, $(\mathcal{L}(\mathcal{A}_1), \Sigma_1) \times_{\mathcal{L}} (\mathcal{L}(\mathcal{A}_2), \Sigma_2) = (\mathcal{L}(\mathcal{A}_1 \times_{\mathcal{A}} \mathcal{A}_2), \Sigma_1 \cup \Sigma_2)$.

Proof. Note $(\mathcal{L}(\mathcal{A}_1), \Sigma_1) \times_{\mathcal{L}} (\mathcal{L}(\mathcal{A}_2), \Sigma_2) = (\mathcal{L}, \Sigma)$ and $(\mathcal{L}(\mathcal{A}_1 \times_{\mathcal{A}} \mathcal{A}_2), \Sigma_1 \cup \Sigma 2) = (\mathcal{L}', \Sigma')$. It is clear that $\Sigma = \Sigma' = \Sigma_1 \cup \Sigma_2$. First show that $\mathcal{L}$ and $\mathcal{L}'$ contain the same words, that is $\forall (w, c) \in \mathcal{L}, \exists c', (w, c') \in \mathcal{L}'$ and conversely $\forall (w, c) \in \mathcal{L}', \exists c', (w, c') \in \mathcal{L}$.

Let w be a word from $\mathcal{L}$. Suppose that w does not appear in $\mathcal{L}'$. Thus there is no path π in $\mathcal{A}_1 \times_{\mathcal{A}} \mathcal{A}_2$ such that $\sigma(\pi) = w$. Hence, by definition of product of weighted automata, either there is no path π_1 in $\mathcal{A}_1$ such that $\sigma_1(\pi_1) = w_{|\Sigma_1}$ or there is no path

π_2 in $\mathcal{A}_2$ such that $\sigma_2(\pi_2) = w_{|\Sigma_2}$. So, by definition of product of weighted languages, w is not a word in $\mathcal{L}$. It proves that if w is in $\mathcal{L}$ but not in $\mathcal{L}'$, then $w$is not in $\mathcal{L}$, which is a contradiction. Hence, $\forall (w,c) \in \mathcal{L}, \exists c', (w,c') \in \mathcal{L}'$.

Let w be a word from $\mathcal{L}'$. Suppose that w does not appear in $\mathcal{L}$. Hence, either $w_{|\Sigma_1}$ does not appear in $\mathcal{L}_1$ or $u_{|\Sigma_2}$ does not appear in $\mathcal{L}_2$. Thus, either there is no path π_1 in $\mathcal{A}_1$ such that $\sigma_1(\pi_1) = w_{|\Sigma_1}$ or there is no path π_2 in $\mathcal{A}_2$ such that $\sigma_2(\pi_2) = w_{|\Sigma_2}$. So, by definition of the product of weighted automata, there is either no path π in $\mathcal{A}_1 \times_{\mathcal{A}} \mathcal{A}_2$ such that $\sigma(\pi)_{|\Sigma_1} = w_{|\Sigma_1}$ or no path π in $\mathcal{A}_1 \times_{\mathcal{A}} \mathcal{A}_2$ such that $\sigma(\pi)_{|\Sigma_2} = w_{|\Sigma_2}$. Thus, no path π in $\mathcal{A}_1 \times_{\mathcal{A}} \mathcal{A}_2$ such that $\sigma(\pi) = w$. Hence, w does not appear in $\mathcal{L}'$. It proves that if w is a word from $\mathcal{L}'$ and does not appear in $\mathcal{L}$, then w does not appear in $\mathcal{L}'$, which is a contradiction. Hence, $\forall (w,c) \in \mathcal{L}', \exists c', (w,c') \in \mathcal{L}$.

Finally, one has the same words in $\mathcal{L}$ and $\mathcal{L}'$. The remaining is to prove that, for any $(w,c) \in \mathcal{L}$ the only $(w,c') \in \mathcal{L}'$ is such that $c' = c$.

Let $(w,c) \in \mathcal{L}$ and $(w,c') \in \mathcal{L}'$ and suppose $c < c'$. It means that there is two paths, π_1 in $\mathcal{A}_1$ and π_2 in $\mathcal{A}_2$ such that $\sigma_1(\pi_1) = w_{|\Sigma_1}$, $\sigma_2(\pi_2) = w_{|\Sigma_2}$, and $c_1^i(\pi_1^-) + c_1(\pi_1) + c_1^f(\pi_1^+) + c_2^i(\pi_2^-) + c_2(\pi_2) + c_2^f(\pi_2^+) = c < c'$. In this case there is a path π in $\mathcal{A}_1 \times_{\mathcal{A}} \mathcal{A}_2$ which is constructed from π_1 and π_2, thus $\sigma(\pi) = w$ and $c^i(\pi^-) + c(\pi) + c^f(\pi^+) \leq c_1^i(\pi_1^-) + c_1(\pi_1) + c_1^f(\pi_1^+) + c_2^i(\pi_2^-) + c_2(\pi_2) + c_2^f(\pi_2^+)$. By definition of c' one has $c' \leq c^i(\pi^-) + c(\pi) + c^f(\pi^+)$. Thus $c' \leq c_1^i(\pi_1^-) + c_1(\pi_1) + c_1^f(\pi_1^+) + c_2^i(\pi_2^-) + c_2(\pi_2) + c_2^f(\pi_2^+) < c'$. It is in contradiction with the hypothesis that $c < c'$. Thus finally one has $c \geq c'$.

Now suppose $c' < c$. It means that there is a path π in $\mathcal{A}_1 \times_{\mathcal{A}} \mathcal{A}_2$ such that $\sigma(\pi) = w$ and $c^i(\pi^-) + c(\pi) + c^f(\pi^+) = c' < c$. Hence, there is a path π_1 in $\mathcal{A}_1$ and a path π_2 in $\mathcal{A}_2$, such that $\sigma_1(\pi_1) = w_{\Sigma_1}$, $\sigma_2(\pi_2) = w_{|\Sigma_2}$ and $c_1^i(\pi_1^-) + c_1(\pi_1) + c_1^f(\pi_1^+) + c_2^i(\pi_2^-) + c_2(\pi_2) + c_2^f(\pi_2^+) = c^i(\pi^-) + c(\pi) + c^f(\pi^+)$. Moreover, one has $c_1^i(\pi_1^-) + c_1(\pi_1) + c_1^f(\pi_1^+) + c_2^i(\pi_2^-) + c_2(\pi_2) + c_2^f(\pi_2^+) \geq c$. Thus, $c \leq c_1^i(\pi_1^-) + c_1(\pi_1) + c_1^f(\pi_1^+) + c_2^i(\pi_2^-) + c_2(\pi_2) + c_2^f(\pi_2^+) = c^i(\pi^-) + c(\pi) + c^f(\pi^+) < c$. It is in contradiction with the hypothesis that $c' < c$. Thus finally one has $c' \geq c$.

To conclude: one has $c \geq c'$ and $c' \geq c$. Thus $c' = c$. This concludes the proof that $\times_{\mathcal{A}}$ implements $\times_{\mathcal{L}}$. □

2.4.2 Cost-optimization: projection of weighted automata

Projection of weighted languages can be implemented directly on weighted automata as well. The idea is to remove the transitions corresponding to the actions which no longer appear after the projection. And to add a new transition for each sequence of actions removed by the projection which is followed by an action which is not removed by the projection. This projection is more formally defined in the following and an example is given in Figure 2.7. In a weighted automaton, by (s, Σ, s') we denote a path from s to s' using only transitions with actions from Σ.

Definition 2.9. *The projection* $\Pi_{\Sigma'}(\mathcal{A}) = (S', S^{I'}, S^{F'}, \Sigma', T', c', c^{i'}, c^{f'})$ *of a weighted automaton* $\mathcal{A} = (S, S^I, S^F, \Sigma, T, c, c^i, c^f)$ *is such that:* $S' = S$, $S^{I'} = S^I$, $S^{F'} = S^F \cup \{s \in S : \exists s' \in S, \exists (s, \Sigma \setminus \Sigma', s')$ *in* $\mathcal{A}\}$, $T' = \{(s,\sigma,s') \in T : \sigma \in \Sigma'\} \cup \{(s,\sigma,s') : \sigma \in \Sigma' \wedge \exists s'' \in S, (s'',\sigma,s') \in T, \exists (s, \Sigma \setminus \Sigma', s'')$ *in* $\mathcal{A}\}$, *for* $t = (s,\sigma,s') \in T'$, $c'(t) = min_{t'=(s'',\sigma,s') \in T \wedge \pi = (s, \Sigma \setminus \Sigma', s'')} c(t') + c(\pi)$, *and for* $s \in S^{I'}$, $c^{i'}(s) = c^i(s)$. *finally for* $s \in S^{F'}$, $c^{f'}(s) = min_{s' \in S^F \wedge \pi = (s, \Sigma \setminus \Sigma', s')} c^f(s') + c(\pi)$.

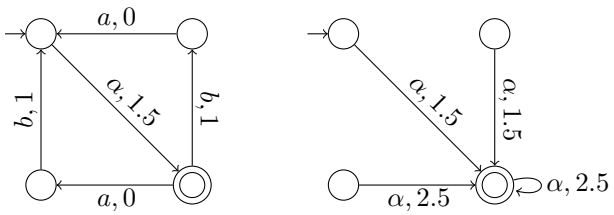


Figure 2.7: a weighted automaton (left) and its projection on $\{\alpha\}$ (right)

Proposition 2.8. *Projection of weighted automata implements projection of weighted languages:* $\forall \mathcal{A}, \forall \Sigma', (\mathcal{L}(\Pi_{\Sigma'}(\mathcal{A})), \Sigma') = \mathit{\Pi}_{\Sigma'}((\mathcal{L}(\mathcal{A}), \Sigma))$.

Proof. Note $\mathit{\Pi}_{\Sigma'}((\mathcal{L}(\mathcal{A}), \Sigma)) = (\mathcal{L}, \Sigma')$ and $\mathcal{L}(\Pi_{\Sigma'}(\mathcal{A})) = \mathcal{L}'$. First show that $\mathcal{L}$ and $\mathcal{L}'$ contain the same words.

Consider a word w in $\mathcal{L}$. Suppose w does not appear in $\mathcal{L}'$. Then, there is no path π in $\mathcal{A}$ such that $\sigma(\pi) = w'$ and $w'_{|\Sigma'} = w$. Thus, there is no word w' in $\mathcal{L}(\mathcal{A})$ such that $w'_{|\Sigma'} = w$. Hence, w is not a word from $\mathcal{L}$. Thus if w appears $\mathcal{L}$ and not in $\mathcal{L}'$, then w does not appear in $\mathcal{L}$. This contradiction implies that any word from $\mathcal{L}$ also appears in $\mathcal{L}'$.

Consider a word w in $\mathcal{L}'$. Suppose w does not appear in $\mathcal{L}$. Then, there is no word w' in $\mathcal{L}(\mathcal{A})$ such that $w'_{|\Sigma'} = w$. Thus, there is no path π in $\mathcal{A}$ such that $\sigma(\pi)_{|\Sigma'} = w$. Hence, w does not appear in $\mathcal{L}'$. It proves that if w appears in $\mathcal{L}'$ but not in $\mathcal{L}$ then w is not a word from $\mathcal{L}'$, which is impossible. So, any word from $\mathcal{L}'$ also appears in $\mathcal{L}$.

It remains to prove that for any $(w, c) \in \mathcal{L}$ and $(w, c') \in \mathcal{L}'$ the costs are such that $c = c'$. There is no $(w', c'') \in \mathcal{L}(\mathcal{A})$ such that $w'_{|\Sigma'} = w$ and $c'' < c$ by definition of projection of weighted languages. Thus, there is no path π in $\mathcal{A}$ such that $\sigma(\pi) = w'$ with $w'_{\Sigma'} = w$ and $c^i(\pi^-) + c(\pi) + c^f(\pi^+) < c$ by definition on the weighted language of a weighted automaton. Hence, there is no path π' in $\Pi_{\Sigma'}(\mathcal{A})$ such that $\sigma(\pi') = w$ and $c^{i'}(\pi'^-) + c'(\pi') + c^{f'}(\pi'^+) < c$ by construction of the projection of weighted automata. So, $c' \geq c$. In the same way one has $c \geq c'$, and then $c = c'$. □

Propositions 2.7 and 2.8 allow to run the message passing algorithm directly on the automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ representing the factors of a factored planning problem. The result obtained will be an automaton $\mathcal{A}'_k$ per factor $\mathcal{A}_k$ with the property that $\mathcal{L}(\mathcal{A}'_k) = \mathcal{L}(\mathcal{A}_k)'$. Thus, from these updated factors $\mathcal{A}'_k$ one easily gets a cost-optimal solution for the factored planning problem considered exactly as it can be done with weighted languages. We conclude this chapter by giving an execution of the message passing with weighted automata.

2.4.3 A sample execution of MPA on weighted automata

Consider the factored planning problem of Figure 2.8, constituted of 3 automata. $\mathcal{A}_1$ and $\mathcal{A}_2$ share a single action α, $\mathcal{A}_2$ and $\mathcal{A}_3$ share a single action β, and $\mathcal{A}_1$ and $\mathcal{A}_3$ share no action. So, the interaction graph (and thus the communication graphs) of this system is a tree: the only edges are between $\mathcal{A}_1$ and $\mathcal{A}_2$ and between $\mathcal{A}_2$ and $\mathcal{A}_3$. Figure 2.9 presents the product of these three automata, which is an automaton $\mathcal{A}$ such that $(\mathcal{L}(\mathcal{A}), \Sigma) = (\mathcal{L}(\mathcal{A}_1), \Sigma_1) \times_{\mathcal{L}} (\mathcal{L}(\mathcal{A}_2), \Sigma_2) \times_{\mathcal{L}} (\mathcal{L}(\mathcal{A}_3), \Sigma_3)$. The only cost-optimal word in $\mathcal{A}$ is $b\beta\alpha$ and its cost is 7.

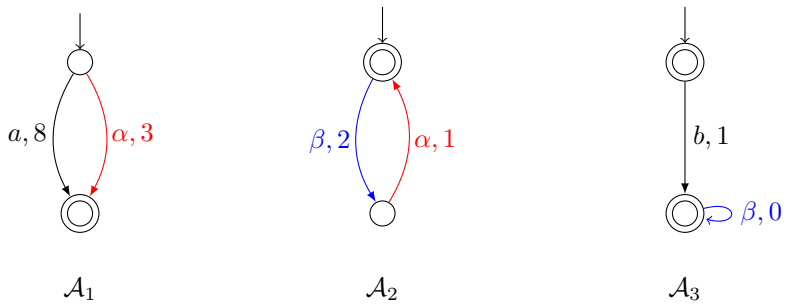


Figure 2.8: A network of three weighted automata.

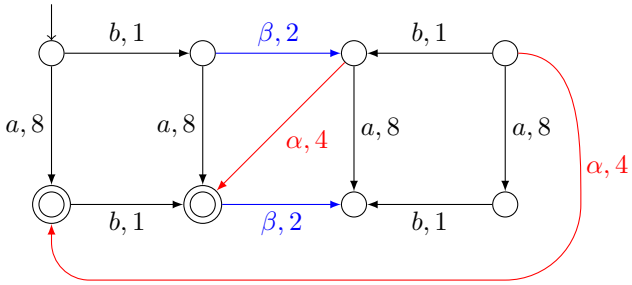


Figure 2.9: The compound system $\mathcal{A} = \mathcal{A}_1 \times_{\mathcal{A}} \mathcal{A}_2 \times_{\mathcal{A}} \mathcal{A}_3$.

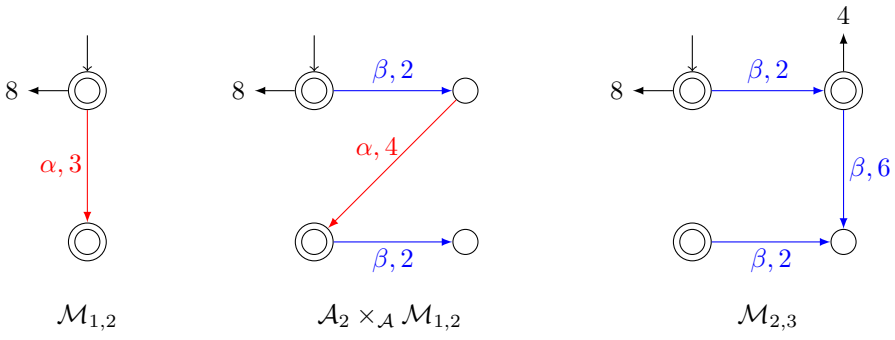


Figure 2.10: Messages from $\mathcal{A}_1$ to $\mathcal{A}_3$.

Applying the message passing to this system allows to find this word and its cost by performing local computations only. In this example 4 messages have to be computed: $\mathcal{M}_{1,2}$, $\mathcal{M}_{2,3}$, $\mathcal{M}_{3,2}$, and $\mathcal{M}_{2,1}$. Figure 2.10 presents the computation of messages from $\mathcal{A}_1$ to $\mathcal{A}_3$, which consists in successively computing $\mathcal{M}_{1,2}$ and $\mathcal{M}_{2,3}$. Figure 2.11 presents the computation of messages in the other way: from $\mathcal{A}_3$ to $\mathcal{A}_1$, which builds $\mathcal{M}_{3,2}$ and then $\mathcal{M}_{2,1}$. Notice that these two computations are independent: $\mathcal{M}_{1,2}$ and $\mathcal{M}_{2,3}$ do not take part into computation of $\mathcal{M}_{3,2}$ and $\mathcal{M}_{2,1}$ (and conversely). Notice also that, with this order of message updates, each message has to be updated only once. Thus, the values of $\mathcal{M}_{1,2}$, $\mathcal{M}_{2,3}$, $\mathcal{M}_{3,2}$, and $\mathcal{M}_{2,1}$ obtained are definitive and allow directly to compute the updated components $\mathcal{A}'_1$, $\mathcal{A}'_2$, and $\mathcal{A}'_3$. The result of these computations is presented in Figure 2.12.

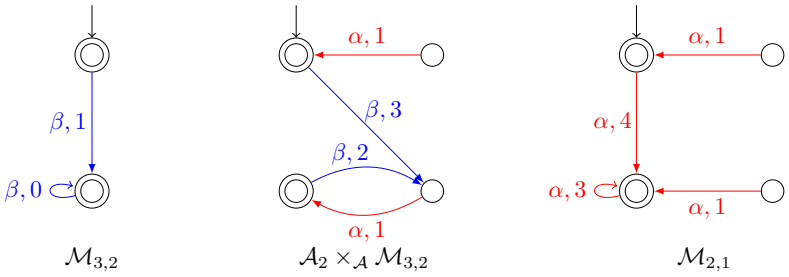


Figure 2.11: Messages from $\mathcal{A}_3$ to $\mathcal{A}_1$.

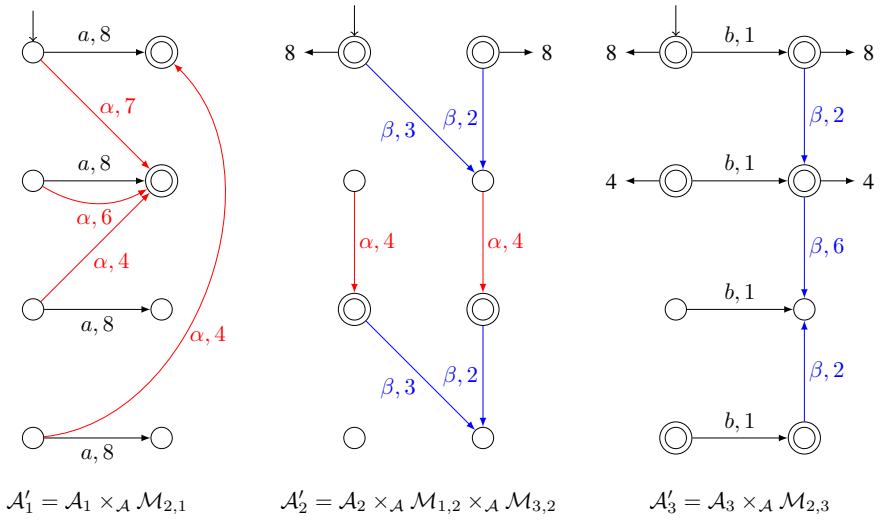


Figure 2.12: Updated components.

In these updated components one can remark that the only cost-optimal plans are α in $\mathcal{A}'_1$, $\beta\alpha$ in $\mathcal{A}'_2$, and $b\beta$ in $\mathcal{A}'_3$. Moreover, these 3 plans have cost 7, which corresponds to the cost of a cost-optimal word in $\mathcal{A}$ as stated above. The only possible interleaving

of α, $\beta\alpha$, and $b\beta$ is $b\beta\alpha$, which is a cost-optimal plan in the factored planning problem considered.

Conclusion

In this chapter we presented a new approach to factored planning, based on message passing algorithms and weighted automata calculus. The main novelty of this approach is that it permits to achieve cost-optimality of plans. Indeed, by contrast with previous approaches our method does not rely on bounds on the length of plans. Such bounds make optimization difficult because cost-optimal plans and shortest plans are not related in general. The core of our approach is in fact a way to refine components of a factored planning problem so that they contain only pieces of global solutions. In each of these updated components one then has to use standard search methods for finding a cost-optimal local plan.

Moreover, due to the use of message passing algorithms, our approach has the interest of being generic: one can use any representation of planning problems as systems as soon as this representation allows to implement the product and the projection of weighted languages. In fact, it is also possible to preserve more than languages: with suitable product and projection operations one could for example imagine preserving simulation relations between systems.

As presented in this chapter, our approach suffers from two weaknesses: even if it is algebraically functional, one still has to prove that it can bring gains in efficiency, and it works only for planning problems living on trees. These weaknesses are addressed in the three following chapters.

In particular the next chapter presents a real implementation of our method based on well known weighted automata algorithms for the implementation of the product and the projection. It focuses especially on methods for reducing the size of the automata involved in the computations (in the example given above one may have noticed that large parts of the automata considered were not useful). In fact, with the formal definitions of product and projection given above the $\mathcal{A}'_k$ would always have the same size as $\mathcal{A} = \mathcal{A}_1 \times_{\mathcal{A}} \ldots \times_{\mathcal{A}} \mathcal{A}_n$. This is not acceptable, as the size of $\mathcal{A}$ is exponential in the size of its components $\mathcal{A}_k$. The possibility to reduce the size of automata is in fact what makes our approach of practical interest, allowing to use only small objects for solving a large planning problem. Notice however that, due to the theoretical complexity of the resolution of planning problems, reducing size of automata will not always be possible.

Chapter 3

Distoplan: a Factored Planner for Cost-Optimal Planning

chapter abstract: *This chapter presents concrete ways to implement the operations presented in the previous chapter, using standard weighted automata algorithms. In particular it focusses on manners of reducing the size of automata involved in computations. It also presents Distoplan, our factored planner, and give some experimental results obtained with it on classical planning benchmarks.*

THE FACTORED PLANNING METHOD presented in the previous chapter has been implemented into a planner called Distoplan. This implementation is based on weighted automata algorithms from [76]. It is however not a direct implementation of product and projection as presented above, which would not be efficient as updated components would all have the size of the compound system. Instead we implement "more efficient" operations of product and projection by trying to reduce as most as possible the sizes of the automata produced by these operations. Remark that, in the worst case, these implementations are not better than the operations of product and projection as defined in the previous chapter as the minimal possible size of an updated component can always be the size of the full compound system (in fact, the use of a determinization procedure for weighted automata can even prevent the projection from terminating). However, in practice, these methods allowed to significantly reduce the size of the automata involved in the message passing, and thus permitted a very efficient resolution of planning problems.

This chapter is organized as follows. We first describe how the product and the projection operations can be implemented in Section 3.1. Then we present techniques for reducing the size of the automata produced by these operations in Section 3.2. Finally, in Section 3.3, we present our planner and give some experimental results obtained with it.

3.1 Algorithms for the product and the projection

This section describes possible implementations of the product and the projection of weighted automata. The projection can be computed thanks to an ε-reduction algorithm

while the product can be obtained using the principles of a breadth first search.

3.1.1 Projection as an ε-reduction

One can remark that the projection operation described in Definition 2.9 corresponds exactly to a standard ε-reduction (to the left) when costs are ignored. In other words, ignoring costs, the projection of an automaton $\mathcal{A}$ over an alphabet Σ' can be implemented as follows. For any states s, s', s'' from $\mathcal{A}$ if there exists a path π from s to s' such that $\sigma(\pi) \in (\Sigma \setminus \Sigma')^*$, for any transition (s', σ, s'') with $\sigma \in \Sigma'$, add (s, σ, s'') in the set of transitions of $\mathcal{A}$. And if s' is final then add s to the set of final states of $\mathcal{A}$. Once this has been done for all states, all transitions (s, σ, s') such that $\sigma \in \Sigma \setminus \Sigma'$ are removed from $\mathcal{A}$. When taking costs into account one has to set the cost of each transition (s, σ, s'') with $\sigma \in \Sigma'$ to the minimum of $c(\pi) + c(t)$ over all paths π from s to some s' such that $\sigma(\pi) \in (\Sigma \setminus \Sigma')^*$ and such that there exists a transition $t = (s', \sigma, s'')$.

More formally, in [76] is presented (in the context of weighted transducers) an algorithm which performs ε-reduction of weighted automata. It uses the notion of ε-*closure* of states. In a weighted automaton $\mathcal{A}$ the ε-closure of a state s is a set of couples (s', c) such that there exists a path π in $\mathcal{A}$ from s to s' with $\sigma(\pi) \in \{\varepsilon\}^*$ and such that c is the minimal cost of such a path in $\mathcal{A}$. Once the ε-closure of each state of an automaton has been constructed it is indeed straightforward to compute the ε-reduction of this automaton. One first removes all transitions labelled by ε from the automaton. Then, for each state s of the automaton and each couple (s', c) in the ε-closure of s one simply has to add a transition $t = (s, \sigma, s'')$ to the set of transitions for each $t' = (s', \sigma, s'')$ already in this set. The cost of t is set to the minimum of its previous cost (which is $+\infty$ if this transition did not exist previously) and of $c + c(t')$. This can be adapted for performing the projection over a set Σ' simply by considering transitions labelled by elements outside of Σ' as ε-transitions. Algorithm 8, at the end of this chapter, concretely describes a possible implementation of the projection.

Computing ε-closure of a state can be done by classical single-source shortest paths algorithms such as the Dijkstra's algorithm [23]. Denote by $|E|$ the number of elements in a set E. For an automaton $\mathcal{A} = (S, S^I, S^F, \Sigma, T, c, c^i, c^f)$, according to [76] and considering that the Dijkstra's algorithm has complexity $O(|S|^2)$, the complexity of this implementation of the projection is $O(|S|^3 + |S||T|)$. Notice that other complexities can be achieved depending on the data-structures used for implementing the Dijkstra's algorithm (see Chapter 4 of [21] for more details).

3.1.2 Product as a breadth first search

A way to compute the product $\mathcal{A}$ of two weighted automata $\mathcal{A}_1$ and $\mathcal{A}_2$ is to perform a breadth first search in $\mathcal{A}$ from a breadth first search in each of its components. Let $\mathcal{A}_1 = (S_1, S_1^I, S_1^F, \Sigma_1, T_1, c_1, c_1^i, c_1^f)$ and $\mathcal{A}_2 = (S_2, S_2^I, S_2^F, \Sigma_2, T_2, c_2, c_2^i, c_2^f)$. The automaton $\mathcal{A}$ is initially empty (i.e. its sets of states and transitions are empty). The states $S_1^I \times S_2^I$ are then added to $\mathcal{A}$ and defined as initials. From that the following steps are repeated until no new state is added to $\mathcal{A}$.

1. Let S be the last set of states added to $\mathcal{A}$.
2. Add to $\mathcal{A}$ all the transitions starting from S private to $\mathcal{A}_1$, that is the following set of transitions:

$$\bigcup_{(s_1,s_2)\in S} \{((s_1, s_2), \sigma, (s_1', s_2)) \ : \ (s_1, \sigma, s_1') \in T_1 \wedge \sigma \in \Sigma_1 \setminus \Sigma_2\}.$$

The cost of one of these transitions is set to the cost of the transition of $\mathcal{A}_1$ from which it has been added.

3. Add to $\mathcal{A}$ all the transitions starting from S private to $\mathcal{A}_2$, that is the following set of transitions:

$$\bigcup_{(s_1,s_2)\in S} \{((s_1,s_2),\sigma,(s_1,s'_2)) \ : \ (s_2,\sigma,s'_2) \in T_2 \wedge \sigma \in \Sigma_2 \setminus \Sigma_1\}.$$

The cost of one of these transitions is set to the cost of the transition of $\mathcal{A}_2$ from which it has been added.

4. Add to $\mathcal{A}$ all the shared transitions starting from S, that is the following set of transitions:

$$\bigcup_{(s_1,s_2)\in S} \{((s_1,s_2),\sigma,(s'_1,s'_2)) \ : \ (s_1,\sigma,s'_1) \in T_1 \wedge (s_2,\sigma,s'_2) \in T_2\}.$$

The cost of one of these transitions is set to the sum of the costs of the transition of $\mathcal{A}_1$ and the transition of $\mathcal{A}_2$ from which it has been added.

5. Add to $\mathcal{A}$ the set of states not previously in $\mathcal{A}$ reached by these new transitions. Any state (s_1, s_2) in this set verifying $s_1 \in S_1^f$ and $s_2 \in S_2^f$ is defined as final and its final cost is set as $c_1^f(s_1) + c_2^f(s_2)$.

Notice that this algorithm does not exactly computes the product of $\mathcal{A}_1$ and $\mathcal{A}_2$ as described in the previous chapter. Instead, $\mathcal{A}$ is the accessible part of this product, that is the automaton obtained by removing all states of $\mathcal{A}_1 \times_{\mathcal{A}} \mathcal{A}_2$ which can not be reached from the initial ones. This automaton has the same language as $\mathcal{A}_1 \times_{\mathcal{A}} \mathcal{A}_2$ but is smaller. Notice also that one could describe a similar algorithm based on a depth first search, or any other search. Algorithm 9, at the end of this chapter, gives a more precise description of this product computation, based on an unspecified search.

The complexity of this product algorithm is linear in the size of its output $\mathcal{A} = (S, S^I, S^F, \Sigma, T, c, c^i, c^f)$ as it corresponds to a breadth first search in $\mathcal{A}$ for which the complexity is $O(|S| + |T|)$ (see Chapter 4 of [21]). For any set T of transitions, denote by $T(\Sigma) \subseteq T$ the set of transitions t such that $\sigma_t \in \Sigma$. With regards to the automata $\mathcal{A}_1 = (S_1, S_1^I, S_1^F, \Sigma_1, T_1, c_1, c_1^i, c_1^f)$ and $\mathcal{A}_2 = (S_2, S_2^I, S_2^F, \Sigma_2, T_2, c_2, c_2^i, c_2^f)$ the complexity of taking the product $\mathcal{A}_1 \times_{\mathcal{A}} \mathcal{A}_2$ is then $O(|S_1||S_2| + |S_2||T_1(\Sigma_1 \setminus \Sigma_2)| + |S_1||T_2(\Sigma_2 \setminus \Sigma_1)| + |T_1(\Sigma_1 \cap \Sigma_2)||T_2(\Sigma_2 \cap \Sigma_1)|)$.

3.2 Reducing the size of the weighted automata

The goal of our work is to deal with very large planning problems. Solving these problems implies to perform a lot of products and projections of automata. Thus, it is fundamental to reduce the cost of performing these operations. For that we try to maintain of reasonable size the automata involved in the computations. The idea is that having a large complexity that depends on the size of a small object may be better for performances than having a small complexity that depends on the size of a large object. We thus increase the complexity of the product and the projection with the result of reducing (hopefully a lot) the size of the automata produced by these operations.

3.2.1 Trimming weighted automata

A first fact allows to reduce the size of the automata produced by the product and the projection described above. These automata can contain states which are not co-accessible (that is states from which no final state can be reached). Moreover the automata produced by the projection can contain states which are not accessible (that is states that can be reached from no initial state). An important fact about states which are not accessible or not co-accessible is that they do not contribute to the languages of the automata. Indeed, no accepted path can use them. These states (and the transitions involving them) can thus be removed from the automata. This can reduce significantly the size of the automata, as for example in Figure 2.12 where instead of 8 states each the automata would have 3, 4, and 3 states. The operation of removing states that are not accessible or not co-accessible in an automaton is called the *trimming*.

The trimming of a (weighted) automaton can be achieved as follows. First perform any search from the initial states for detecting the non-accessible states. Then perform a backward search (that is a search in the automaton in which all transitions have been reversed) from the final states for detecting the states that are not co-accessible. Finally, remove from the automaton all the states that have been found to be non-accessible or non-co-accessible and the transitions involving these states. The complexity of the trimming operation is linear in the size of the automaton on which it is performed [76]. One can thus add to the versions of the product and the projection proposed above a trimming step performed just after these operations.

3.2.2 On the determinization of weighted automata

A second important fact about the automata we consider is that, even if they are initially deterministic, the projection make them lose their determinism. Moreover, the product of two non-deterministic automata is potentially much larger than the product of two deterministic automata. As the product of two non-deterministic automata is also non-deterministic, successive products implying non-deterministic automata may output very large automata from small ones. In particular, consider the product of an automaton $\mathcal{A}$ with itself. If $\mathcal{A}$ is deterministic then $\mathcal{A} \times_{\mathcal{A}} \mathcal{A} = \mathcal{A}$. If $\mathcal{A}$ is not deterministic this is not the case in general, as represented in Figure 3.1. We would thus prefer to use deterministic automata when taking a product. This implies to determinize the result of each projection. The remaining of this section focuses on a manner of determinizing weighted automata. It also presents the limitations of the determinization of weighted automata (not all of them can be determinized).

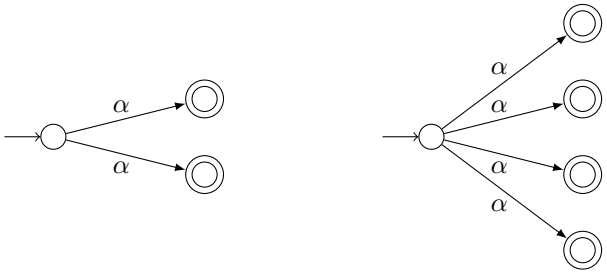


Figure 3.1: An automaton $\mathcal{A}$ (left) and the product $\mathcal{A} \times_{\mathcal{A}} \mathcal{A}$ (right).

Determinization algorithm

There exists an algorithm for determinizing weighted automata [76]. This algorithm is close to the classical subset construction used for the determinization of finite automata [81]. Given a weighted automaton $\mathcal{A} = (S, S^I, S^F, \Sigma, T, c, c^i, c^f)$, let us denote by $Det(\mathcal{A})$ its determinized version provided by the algorithm. The states of $Det(\mathcal{A})$ are of the form $q = (A, \lambda)$, where $A \subseteq S$ and $\lambda : A \to \mathbb{R}_+$. Denote by s^i an initial state of $\mathcal{A}$ with minimal cost. The initial state of $Det(\mathcal{A})$ is (S^I, λ^I), such that $\lambda^I(s) = c^i(s) - c^i(s^i)$ for all $s \in S^I$. The cost of this initial state is $c^i(s^i)$. The other states are obtained by recursion from this initial state: from $q = (A, \lambda)$ the new state $q' = (A', \lambda')$ obtained by firing an action $\sigma \in \Sigma$ is such that $A' = \{s \in S \ : \ \exists s' \in A, (s', \sigma, s) \in T\}$ and, for $s' \in A'$,

$$\lambda'(s') = \overline{\lambda'}(s') - \left(\min_{s'' \in A'} \overline{\lambda'}(s'') \right),$$

where $\overline{\lambda'}(s')$ is defined as:

$$\overline{\lambda'}(s') = \min_{s \in A, t=(s,\sigma,s') \in T} \lambda(s) + c(t).$$

The cost of transition (q, σ, q') in $Det(\mathcal{A})$ is then $min_{s' \in A'} \overline{\lambda'}(s')$. The cost of a final state (A, λ) is $\min_{s \in A \cap S^F} \lambda(s) + c_f(s)$.

Figure 3.2 shows an automaton and its determinized version (in order to clarify the figure, a state $q = (A, \lambda)$ is represented by all the pairs $s, \lambda(s)$ for $s \in A$). The idea behind this construction is the following. Consider the determinized version $Det(\mathcal{A})$ of a weighted automaton $\mathcal{A}$, denote by $q = (A, \lambda)$ a state of $Det(\mathcal{A})$ and by π a path in $Det(\mathcal{A})$ that reaches q. The set A contains exactly the states from S reachable in $\mathcal{A}$ by paths π' such that $\sigma(\pi') = \sigma(\pi)$ (as in the classical subset construction for the determinization of finite automata). Consider a minimum cost path π_m in $\mathcal{A}$ such that $\sigma(\pi_m) = \sigma(\pi)$. The construction of $Det(\mathcal{A})$ ensures that π_m reaches a state s in $\mathcal{A}$ such that $\lambda(s) = 0$. For any other state $s' \in A$, $\lambda(s')$ is the minimal cost that has to be added to the cost of π_m in order to reach s' in $\mathcal{A}$ with a path π' such that $\sigma(\pi') = \sigma(\pi)$. Finally, the cost of π in $Det(\mathcal{A})$ is equal to $c(\pi_m)$.

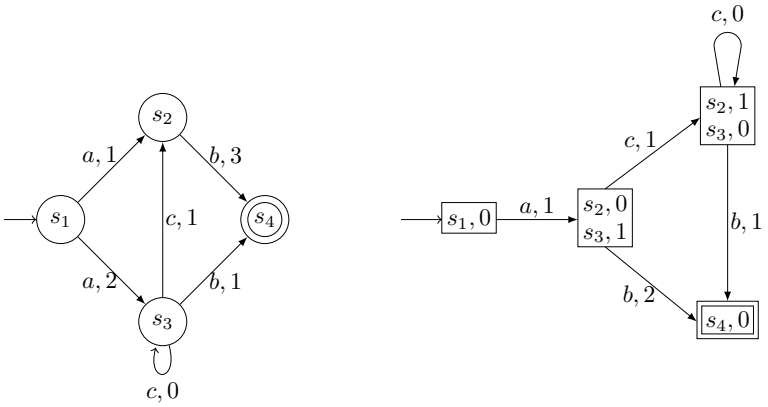


Figure 3.2: A weighted automaton (left) and its determinized version (right).

Limitations of the determinization

Not all weighted automata are determinizable. In other words, there exists weighted automata $\mathcal{A}$ such that no (finite) deterministic weighted automaton $\mathcal{A}'$ exists verifying $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$. Figure 3.3 gives an example of a weighted automaton which can not be determinized. In this automaton, the cost of a word w is $\min(|w|_a, |w|_b)$, where $|w|_a$ is the number of occurrences of action a in w. A deterministic automaton recognizing the same language should "count" the number of a and b in a word, and, thus, could not be finite. Part of an (infinite) determinized version of this automaton is illustrated in Figure 3.4.

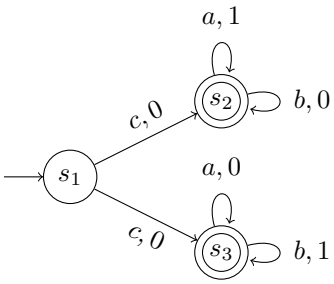


Figure 3.3: A weighted automaton that can not be determinized

There exists a sufficient condition for determinizability of weighted automata [76]. This condition is based on the *twin property*.

Definition 3.1. *In a weighted automaton $\mathcal{A} = (S, S^I, S^F, \Sigma, T, c, c^i, c^f)$, two states s and s' are twins if and only if:*

1. *there exists two paths π and π' such that $\pi^- \in S^I, \pi'^- \in S^I, \pi^+ = s, \pi'^+ = s'$, and $\sigma(\pi) = \sigma(\pi')$, and*
2. *for all $w \in \Sigma^+$ such that there exists a path π and a path π' in $\mathcal{A}$ verifying $\sigma(\pi) = w = \sigma(\pi')$, $\pi^- = \pi^+ = s$, and $\pi'^- = \pi'^+ = s'$ then the costs of these paths are the same: $c(\pi) = c(\pi')$.*

A weighted automaton has the twin property if and only if any two states s, s' of this automaton verifying point 1 of Definition 3.1 are twins. Notice that deciding the twin property in the general case has recently be shown to be PSPACE-complete [59].

Theorem 3.1 (Theorem 5 in [76])**.** *Any weighted automaton verifying the twin property is determinizable.*

The twin property is only a sufficient condition in general. Yet, there exists particular classes of weighted automata for which this condition is also necessary.

Definition 3.2. *An automaton $\mathcal{A}$ is said to be* unambiguous *if and only if for each word w accepted by $\mathcal{A}$ there exists only one accepted path π verifying $\sigma(\pi) = w$.*

The twin property is a necessary condition for determinizability of trim unambiguous weighted automata [14]. All unambiguous weighted automata admit a trim unambiguous weighted automaton recognizing the same language. However, not all weighted automata admit an unambiguous weighted automata accepting the same language [62].

Definition 3.3. *An automaton $\mathcal{A}$ is said to be* finitely ambiguous *if and only if there exists an integer k such that for each word w accepted by $\mathcal{A}$ there exists at most k accepted paths π verifying $\sigma(\pi) = w$.*

In the case of finitely ambiguous weighted automata determinizability of weighted automata is decidable as well [62]. In fact, for trim finitely ambiguous weighted automata the twin property is a necessary condition for determinizability [58].

Definition 3.4. *An automaton $\mathcal{A}$ is said to be* polynomially ambiguous *if and only if there exists a polynomial $P : \mathbb{N} \to \mathbb{N}$ such that for each word w accepted by $\mathcal{A}$ there exists at most $P(|w|)$ accepted paths π verifying $\sigma(\pi) = w$.*

Even for polynomially ambiguous weighted automata determinizability is decidable [58, 60]. However, for the general class of weighted automata there is – to our knowledge – no decidability result for determinizability.

Partial determinization

Even if determinizability was decidable, the problem of non-determinizable weighted automata would remain. We suggest to avoid this problem by performing a partial determinization. That is a determinization procedure that stops determinization at some depth and provides an automaton which is deterministic "at the beginning". Such a procedure allows one to recognize small words with the deterministic part of the automaton, and to use the non-deterministic part for larger words only. One can use the partial determinization when the twin property is not verified, or use it in all cases without considering determinizability of weighted automata (it is not reasonable to consider determinized automata with too many states).

In the determinized version of a weighted automaton it is possible to associate a depth to each state. Initial state has depth 0. The depth of any other state is the minimal depth among its predecessors, plus 1. For example, Figure 3.4 represents the (beginning of the) determinization of Figure 3.3 with depths of states represented by dashed lines.

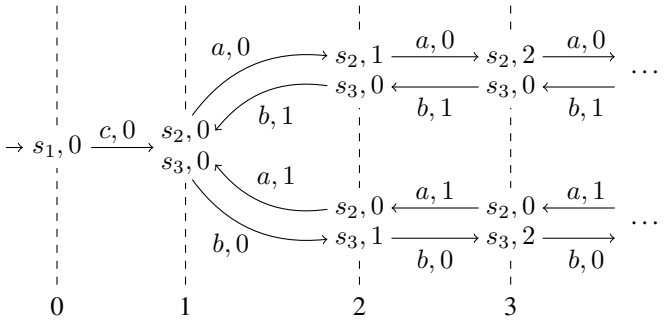


Figure 3.4: First steps of the (infinite) determinization of Figure 3.3. Notice that all states – excepted $(s_1, 0)$ – are final states.

Partial determinization consists in stopping determinization at some depth and then branching the states at this depth to the original (non deterministic) automaton (without initial states). Formally, in partial determinization at depth n of a weighted automaton $\mathcal{A} = (S, S^I, S^F, \Sigma, T, c, c^i, c^f)$, all states of depth smaller or equal to n are obtained

by a standard determinization procedure (as described above). Then, for each state (A, λ) at depth n, a transition (A, σ, s) is added for each s and each σ such that there exists $(s', \sigma, s) \in T$ for some $s' \in A$. The cost of such a transition is such that $c'((A, \sigma, s)) = \min_{s' \in A, (s', \sigma, s) \in T} \lambda(s') + c((s', \sigma, s))$. For example the partial determinization at depth 2 of the automaton of Figure 3.3 is represented in Figure 3.5.

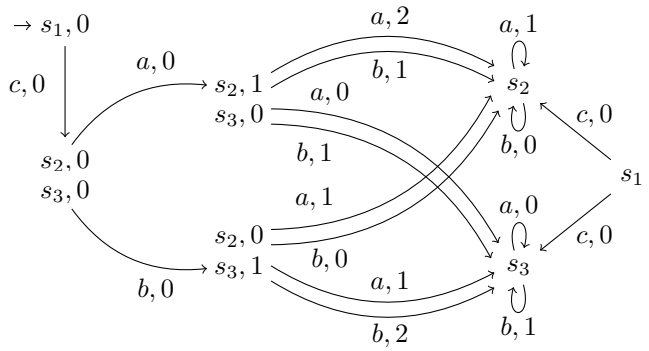


Figure 3.5: Partial determinization at depth 2 of Figure 3.3. Notice that all states – excepted $(s_1, 0)$ and s_1 – are final states. State s_1 may be trimmed.

3.2.3 Minimizing weighted automata

Finally, it is possible to minimize deterministic weighted automata. This means that, for any deterministic weighted automaton $\mathcal{A}$ it is possible to provide another deterministic weighted automaton $\mathcal{A}'$ recognizing the same language and such that any other deterministic weighted automaton recognizing the language of $\mathcal{A}$ would have more states than $\mathcal{A}'$. Hence, if we work with deterministic automata it may be of interest to minimize them.

Minimization of weighted automata can be achieved by a two steps procedure consisting in a weight-pushing step followed by a minimization step [76]. Applying weight-pushing in an automaton aims at "pushing" at most as possible the costs of transitions toward the initial states while preserving the language of the automaton. The minimization step is a standard minimization, as described in Section 3 of Chapter 1 of [81] for example, but considering that in $\mathcal{A} = (S, S^I, S^F, \Sigma, T, c, c^i, c^f)$ the label of a transition $t \in T$ is $(\sigma_t, c(t))$ rather than just σ_t.

More formally, in a weighted automaton $\mathcal{A} = (S, S^I, S^F, \Sigma, T, c, c^i, c^f)$ the weight-pushing can be described by the following equations. They define the new costs c' of transitions, ${c^i}'$ of initial states, and ${c^f}'$ of final states.

$$\begin{aligned} \forall s^i \in S^I, \quad & {c^i}'(s^i) = c^i(s^i) + d(s^i) \\ \forall t \in T, d(t^-) \neq +\infty, \quad & c'(t) = c(t) + d(t^+) - d(t^-) \\ \forall s^f \in S^F, d(s^f) \neq +\infty, \quad & {c^f}'(s^f) = c^f(s^f) - d(s^f) \end{aligned}$$

In these equations, for any state s, $d(s)$ is a shortest distance from s to the set of final states S^F, taking into account final costs. In other words,

$$d(s) = \min_{\pi, \pi^- = s, \pi^+ \in F} c(\pi) + c^f(\pi^+).$$

According to [76] the overall complexity of the minimization of a deterministic weighted automaton $\mathcal{A} = (S, S^I, S^F, \Sigma, T, c, c^i, c^f)$ is $O((|S| + |T|) \log |S|)$.

The remaining of this chapter is dedicated to the presentation of the implementation of the previous algorithms into a factored planner. In particular, we test this planner on some benchmarks from the international planning competitions and we compare its performances with the performances of other up-to-date planners. This experimental analysis will also show the practical interest of reducing the size of automata after each operation. Indeed, we tried various versions of our planner: a simple version doing no size reduction, a version using trimming, and a version using determinization and minimization.

3.3 Distoplan

We implemented the message passing algorithm for weighted automata in a planner called Distoplan. Our implementation is based on the openFst library[1] for most of the operations on weighted automata (ϵ-reduction, trimming, minimization, determinization). Our planner accepts as input planning problems given directly in terms of weighted automata (using the format specified in openFst) or as PDDL (Planning Domain Definition Language) files [34] (which are a standard representation of planning problems in the planning community). The parsing of PDDL files is done using the parser from HSP* planner [39].

Notice that our implementation only deals with factored planning problem. It cannot automatically find a decomposition of a given planning problem. In other words, it is unable to find a decomposition of the set of atoms that ensures that the communication graphs are trees. This is due to the fact that, currently, it is not known what is an efficient decomposition. Methods to decompose problems exists (such as in [22] or similarly in [1]) but they only provide a communication graph which is a tree, not necessarily the best one for factored planning, or even a good one (as it is not know what exactly is a good decomposition). The possibility of automatically finding good decompositions of problems will be discussed in the conclusion of this document as a possible future work.

3.3.1 An extended example

Distoplan has been first tried on a toy example we developed. Consider a problem where a truck has to transport products between different sites: production sites and warehouses. The truck, each warehouse and each production site have maximum storage capacity. The truck has the possibility to move from site i to site j (M_{ij}), where the precise moves and their costs depend on the road network, the truck's load, or any other condition. The truck can also load a unit of product from production site i (L_i) and unload a unit to warehouse i (U_i), while the maximum storage capacities are respected. Production site i can also produce one unit of product (P_i), under the constraints of storage capacity. The production cost depends on the current number of units of product stored at the production site. The staff at a production site can also influence the costs of production and load by being ready for one task or another.

[1] http://www.openfst.org/

Figure 3.6 shows the weighted automata corresponding to such a problem involving a truck (T), two production sites (P_1 and P_2) and a warehouse (W). The shared labels and the corresponding transitions are colored. Each production site has a storage capacity of 2, as the truck, and the warehouse has a storage capacity of 3. Initially the truck is at production site 2, which has one unit of product stored and ready to be loaded. The other sites are empty. The goal is to fill the warehouse. At the end production sites have to be empty and the truck has to be empty and at the warehouse. Notice that when the truck is full it can not go from production site 1 to warehouse directly.

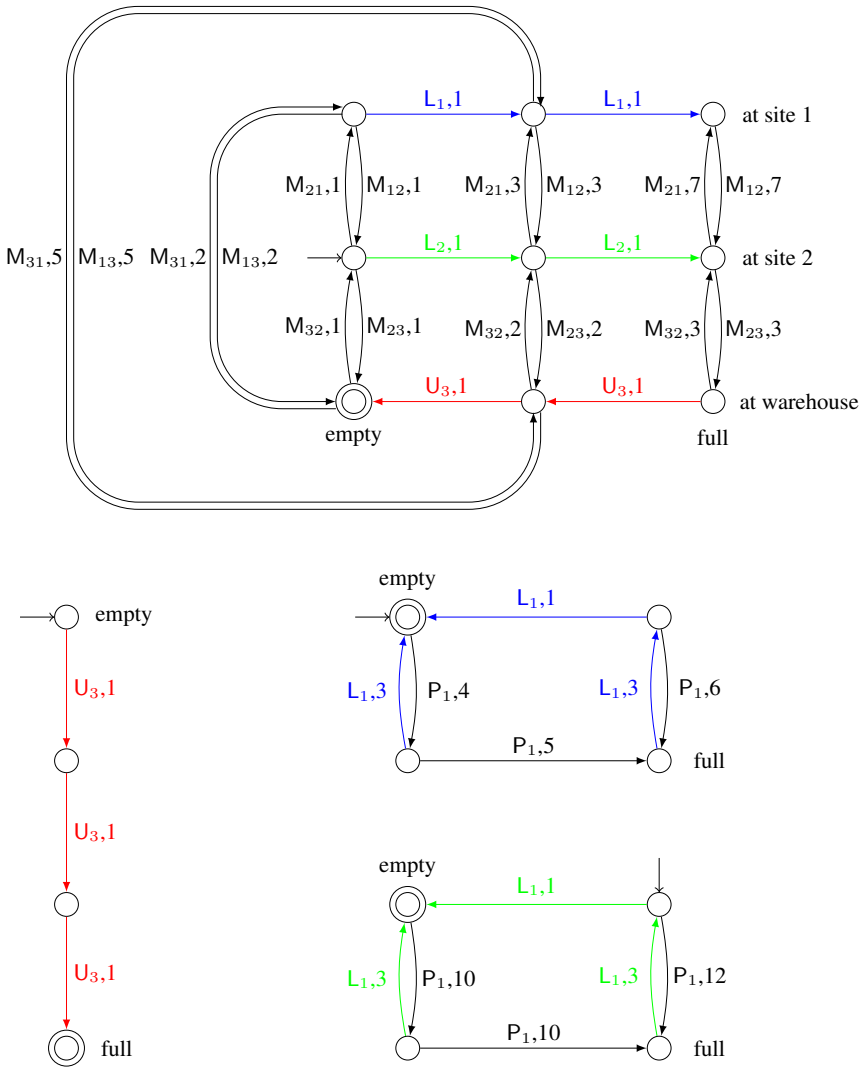


Figure 3.6: Truck (top), warehouse (bottom left), and two production sites (bottom right).

The communication graph of this problem has a star shape, with the truck at the center and the production sites and the warehouse connected to it (Figure 3.7). So the MPA requires six message computations.

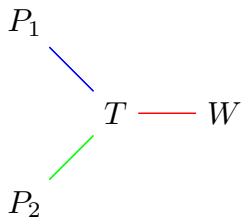


Figure 3.7: Communication graph

First, consider the computation of $\mathcal{M}_{1,T}$, the message from P_1 to T. $\mathcal{M}_{1,T}$ is the projection of P_1 on the load actions, as these are shared with the truck. The steps of this projection appear in Figure 3.8.

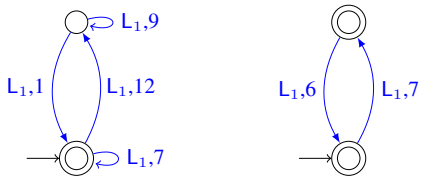


Figure 3.8: P_1 after projection (left) and after determinization and minimization (right).

One can obtain in a similar manner the two other messages sent to the truck component from the second production site ($\mathcal{M}_{2,T}$) and from the warehouse ($\mathcal{M}_{W,T}$) respectively (Figure 3.9). Notice that final states have termination costs (here 1 or 3).

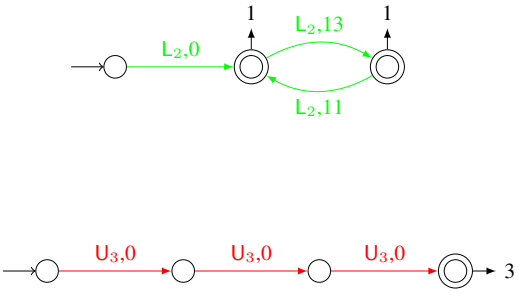


Figure 3.9: messages from P_2 to T (top) and W to T (bottom).

Then the truck updates its own messages $\mathcal{M}_{T,1}$, $\mathcal{M}_{T,2}$, and $\mathcal{M}_{T,W}$ (Figure 3.10). Notice that these updates propagate the constraints imposed by P_1, P_2, and W. For example, only three U_3 are allowed by $\mathcal{M}_{W,T}$, hence the messages $\mathcal{M}_{T,1}$ and $\mathcal{M}_{T,2}$ allow at most three load actions.

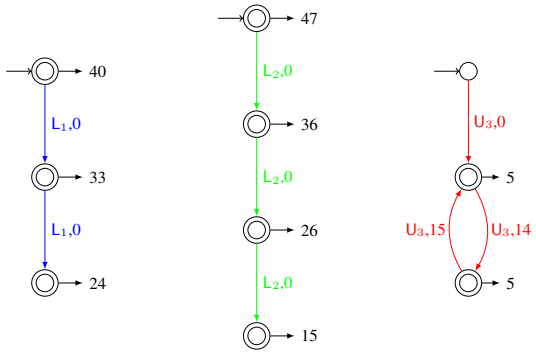


Figure 3.10: messages from T to P_1 (left), P_2 (center), and W (right).

At this point messages are stable and the reduced components P_1', P_2', W' (Figure 3.11), and T' (not represented) can be derived. The following optimal local plans can easily be found: $\mathsf{P}_1\mathsf{P}_1\mathsf{L}_1\mathsf{L}_1$ in P_1', L_2 in P_2', and $\mathsf{U}_3\mathsf{U}_3\mathsf{U}_3$ in W'. Automaton T' is too large to be represented here, one of its optimal local plans is the following: $\mathsf{M}_{21}\mathsf{L}_1\mathsf{M}_{12}\mathsf{L}_2\mathsf{M}_{23}\mathsf{U}_3\mathsf{U}_3\mathsf{M}_{31}\mathsf{L}_1\mathsf{M}_{12}\mathsf{M}_{23}\mathsf{U}_3$. It is easy to see that the four optimal local plans given above can be synchronized into a globally optimal plan of cost 37. Note that these four local plans can be synchronized in different ways: for example the optimal global plan could indifferently start with $\mathsf{P}_1\mathsf{P}_1\mathsf{M}_{21}$, with $\mathsf{P}_1\mathsf{M}_{21}\mathsf{P}_1$, or with $\mathsf{M}_{21}\mathsf{P}_1\mathsf{P}_1$.

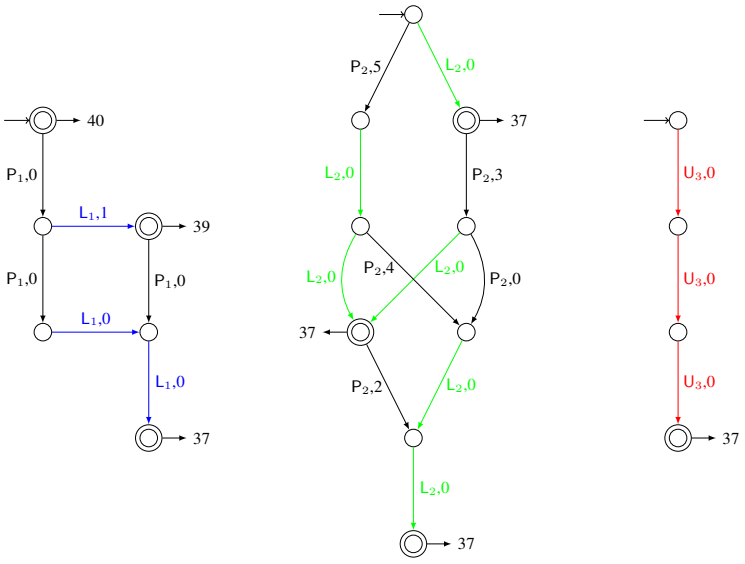


Figure 3.11: MPA output: P_1' (left), P_2' (center), and W' (right).

A possible optimal global plan is then: production site 1 produces two units of product. Then the truck moves to production site 1, loads once, returns at production site 2, loads once, goes to the warehouse, unloads twice, returns to production site 1, loads once and then goes to the warehouse via production site 2, and unloads.

3.3.2 Experimental results

We then tested our planner on a few standard planning problems. When possible we compared our results with up-to-date planners in order to validate our approach. The main difficulty when performing these experiments has been to find problems which were relevant to factored planning.

Rooms and robot

The first test we performed was on a problem presented in [1], called rooms and robot. We wanted to try this problem because [1] presents the only implemented factored planner to our knowledge and uses rooms and robot as a benchmark for testing it. Thus, it was of interest to note if the performances of our planner were comparable to the performances of this one (notice that our planner performs cost-optimal planning, which is not the case of the one from [1]).

The rooms and robot problem is the following: a robot has to move in different rooms in order to close and lock one window per room. More precisely, the rooms are organized into a circle and the robot can only move in one step from the room where it is to an adjacent room on the circle. Inside a room the robot can close the window, and lock it if it is closed. The goal is to lock all windows (in our case with a minimal number of actions).

A possible STRIPS description of the domain of this problem is the following. For each room i there is three atoms: in_i stating that the robot is in the room, $closed_i$ stating that the window is closed, and $locked_i$ stating that the window is locked. From these atoms several actions are defined. An action to close the window in room i: $close_i$. Its precondition is $\{in_i\}$, in other words the robot has to be in the room. It has no negative effect and its positive effect is $\{closed_i\}$. An action to lock a closed window in room i: $lock_i$. Its preconditions is $\{in_i, closed_i\}$: the robot has to be in the room and the window has to be closed. It has no negative effect and its positive effect is $\{locked_i\}$. An action to move the robot from room i to the next room (resp. to the previous room) j on the circle: $movecl_i$ (resp. $moveccl_i$). Its precondition is $\{in_i\}$. Its negative effect is $\{in_i\}$, that removes the robot from room i. Its positive effect is $\{in_j\}$, stating that the robot is now in room j.

A natural decomposition of this problem considers each room as a component. In other words, the component i will be defined by the set of atoms $\{in_i, closed_i, locked_i\}$. However, this decomposition is such that the interaction graph contains a cycle and has no redundant edge, as represented in Figure 3.12 for four rooms (shared actions are represented on edges).

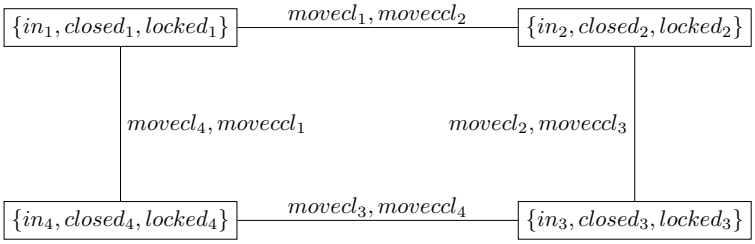


Figure 3.12: Interaction graph for rooms and robot.

In order to obtain a tree shaped communication graph, and thus to permit the use of

the message passing algorithm on this problem, we had to propose another decomposition. For this purpose, we added an atom $isin_i$ for each room i, giving the position of the robot. We accordingly modified the moving actions by adding these new axioms to preconditions and effects. The window closing and window locking actions remained as before. Each set $\{in_i, closed_i, locked_i\}$ still defines a component and the set $\cup_i\{isin_i\}$ defines a new component. The corresponding interaction graph still has cycles. However, some edges are redundant and the only communication graph is a tree. This is represented in Figure 3.13, where dashed edges are redundant.

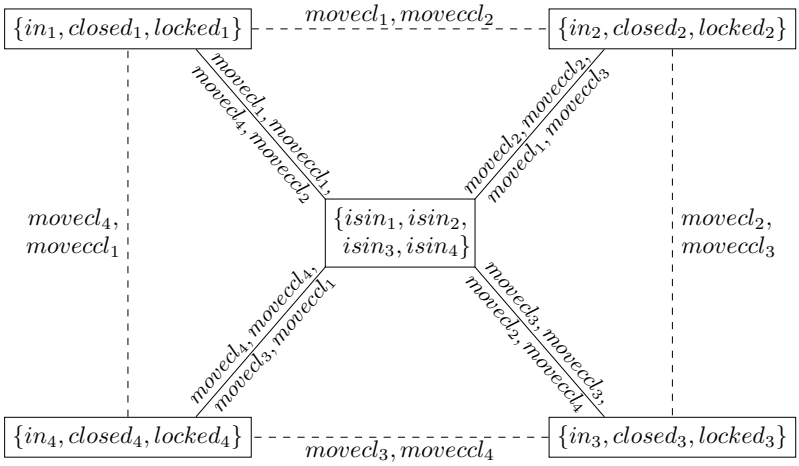


Figure 3.13: Communication graph for rooms and robot.

The results obtained by Distoplan on this problem are given in Figure 3.14. The leftmost plot presents the execution time of three versions of Distoplan on small instances of rooms and robot (5 to 11 rooms). The upper curve is obtained using no size reduction technique. The middle curve is obtained when the automata are trimmed after each product and each projection. The lower curve is obtained when the automata are minimized after each each product and projection (they are thus also determinized after each projection). The rightmost plot presents the execution time of Distoplan on larger instances of rooms and robot (up to 50 rooms). It has been obtained with the version of Distoplan where automata are minimized after each iteration.

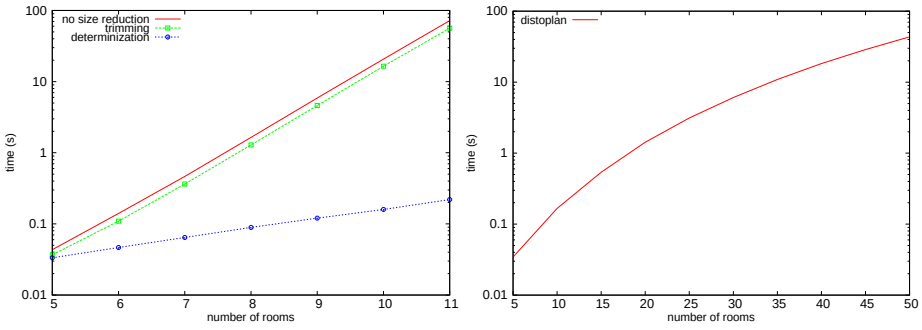


Figure 3.14: results obtained by Distoplan on rooms and robot, the time is in logscale.

The first conclusion we can draw from these results is that, on this particular problem, using minimal deterministic automata strongly improves the time efficiency of Distoplan. This significant efficiency gap shows that the use of deterministic automata - when possible - can really be of interest and may allow a better scalability of our planner.

A second conclusion is that, as expected, the execution time of Distoplan scales well with the size of this problem. We obtain, in fact, a problem solving time subexponential in the number of rooms. This is comparable with what is presented in [1]. It is possible, thought, that a better efficiency can be achieved using another decomposition of the problems. Indeed our decomposition has the drawback that the size of one of the components grows with the number of rooms. Finding a decomposition which avoids this phenomenon may result in better performances. Such a decomposition is proposed in [1], but we were not able to adapt it to our setting (recall that our decompositions are defined by subsets of the atoms while in [1] the decompositions are defined by subsets of the actions).

IPC benchmarks

We then tested Distoplan on problems from international planning competitions (IPC). Among the problems we considered we found two that we were able to decompose well. That is, for which we found a decomposition so that only the number of components is increased (their size remaining the same) with growing instances of the problem. These two problems come from the Promela domains of the fourth international planning competition [48]. These domains regroup problems translated to PDDL from the Promela language. The first problem corresponds to the classical "dinning philosophers" problem, while the second problem is based on a communication protocol for a network of optical telegraphs. In each problem the objective is to find a deadlock. We also considered other versions of these problems where no deadlock exists. In this case one has to detect the absence of deadlock.

For each problem we proposed a decomposition ensuring communication graphs to be trees. Using these decompositions we ran Distoplan on instances of growing size for each problem and compared the run times obtained with the performances of other up-to-date planners. For comparison we used a planner based on Fast-Downward [44] (i.e. an A* based search) with the landmark cut heuristic [45] and the IPC-5 version of SATPLAN [56]. Notice that SATPLAN would not take part in the same track as Distoplan in a planning competition. Indeed it does not ensure cost-optimality of the plans found.

IPC benchmark 1: dinning philosophers

Some philosophers want to eat. They sit all around a table, with one fork between any two philosophers. To eat a philosopher needs two forks: not all philosophers can eat at the same time. He has to take a first fork, then a second one. When he has finished eating a philosopher releases the forks he used. In this setting one has to find deadlocks: situations were no philosopher can eat and no fork is free.

More precisely, philosophers and forks form an alternating cycle. Each philosopher can perform the following actions: 1) take left fork if free, 2) take right fork if free, 3) release right fork if taken, and 4) release left fork if taken. These actions can be performed only in this order. Hence, a simple deadlock occurs when each philosopher has taken his left fork: no fork can be released and no more fork can be taken.

When looking at this problem as a factored problem, an intuitive approach is to consider the set of atoms p_i defining each philosopher as a component and the set of atoms f_i defining each fork as a component. However, in this case the communication graph obtained is not a tree: it is a cycle, as represented in Figure 3.15 (left) for four philosophers. It is possible to come up with a tree shaped communication graph by defining components as the union of the atoms defining each philosopher with the atoms defining the opposite fork on the circle. Figure 3.15 (right) represents the interaction graph obtained for four philosophers. It is a line, and thus a tree.

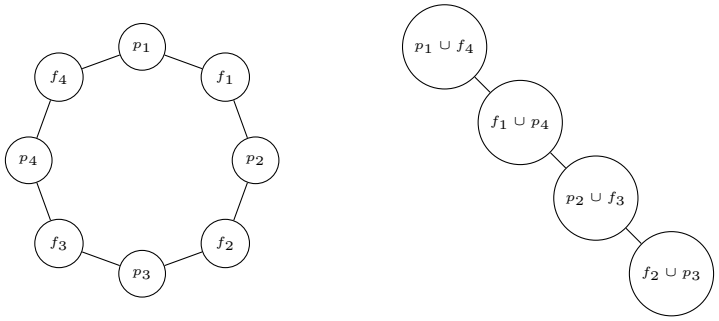


Figure 3.15: Interaction graphs for two possible decompositions of the dinning philosophers problem (with four philosophers).

Results obtained with Distoplan on instances of the dinning philosophers problem of growing size are presented in Figure 3.16. The same figure also gives results obtained with Fast Downward and SATPLAN. The left plot presents results obtained in the exact case presented above, when a deadlock exists. The right plot presents results obtained with slightly modified problems where there is no deadlock (this is achieved by allowing one of the philosophers to take right fork first).

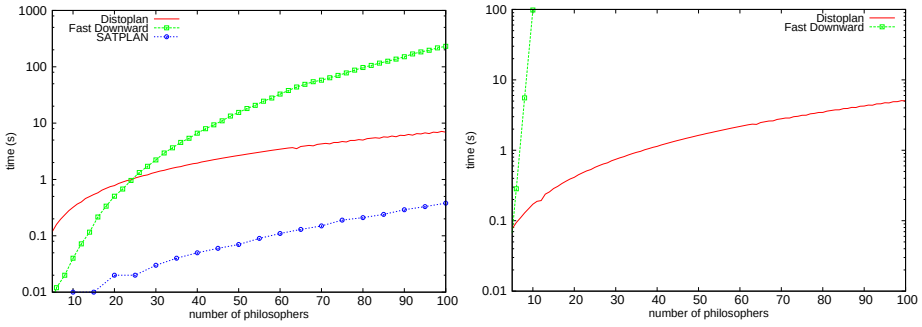


Figure 3.16: Performances of Distoplan, Fast Downward, and SATPLAN on philosophers problems with (letf) and without (right) deadlocks. Time is logscale.

One can notice that Distoplan runs in sub-exponential time in the number of philosophers (that is in the number of components) in both cases. In fact, Distoplan is not affected by the presence or the absence of solution. We used the version of Distoplan which minimizes automata after each operations (as for rooms and robot, the version

with no size reducing technique and the version using trimming were much less efficient). With regards to other planners we remark that SATPLAN is – as expected – very efficient when there is solutions. Remark however that it does not guarantee cost-optimality of solutions. When there is no solution SATPLAN is not able to detect it, that's why it is not plotted in the results. Fast Downward works well in presence of deadlocks, but scales less efficiently than Distoplan. When there is no solutions it has to explore the full state space of the problem to detect it and thus runs in exponential time in the number of philosophers.

IPC benchmark 2: optical telegraph

This second benchmark is the following: some telegraph stations – organized as a circle – have to communicate, following a precise protocol. As for philosophers, the goal is to find potential deadlocks in the communication protocol. In fact, this problem can be seen as philosophers with more private actions. The decomposition we proposed for this problem is based on the same principles as for philosophers. Each natural component of the circle (here telegraph stations and their communication channels) is merged with its opposite on the circle.

The results we obtained are presented in Figure 3.17. The left plot has been obtained for problems with deadlocks. The right plot has been obtained for problems without deadlocks (telegraph stations are organized as a line rather than as a circle).

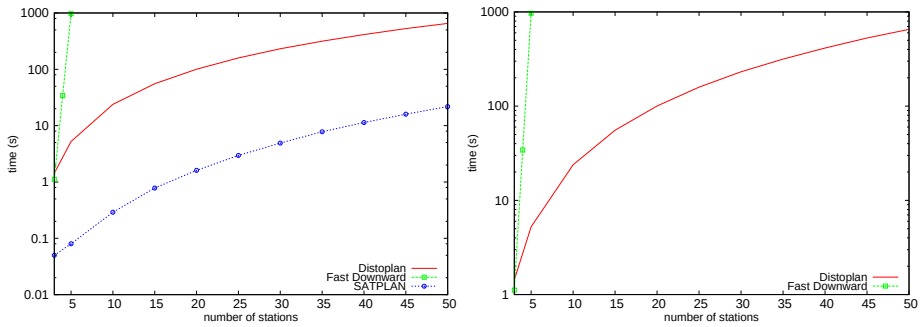


Figure 3.17: Performances of Distoplan, Fast Downward, and SATPLAN on optical telegraph problems with (left) and without (right) deadlocks. Time is logscale.

For Distoplan and SATPLAN these problems are not that different from the dinning philosophers problems. The results obtained are thus similar to the previous ones: a computation time sub-exponential in the size of the problems. The addition of many private actions in each component (compared to the dinning philosophers case) makes the state space of this problem much larger. This explains the lower efficiency of Fast Downward. Notice that, on these problems, most of the computation time of Distoplan (90%) is spent building the initial automata from the PDDL representation of the problems.

Conclusion

In this chapter we explained how the results of Chapter 2 can be implemented concretely. In particular we focused on methods for trying to reduce the size of the au-

tomata involved in the computations. In theory, these methods can result in huge automata, in particular because of the determinization of weighted automata. However, they revealed to be of interest by reducing significantly the computation time needed by our planner for solving all the problems we considered.

The main difficulty when testing has been to find benchmarks which factored well. In other words, we looked for problems such that we were able to propose a decomposition giving tree shaped communication graphs and such that with increasing size of problems the number of components increased but not their size. This explains the little number of problems on which we tried Distoplan.

Our experiments revealed that Distoplan is capable of nice performances on problems suitable for factored planning. In particular, we found optimal plans in two problems from international planning competitions much more efficiently than an up-to-date heuristic search. Moreover we remarked that our approach - as a side effect - is very efficient for detecting absence of solutions in problems. This is due to the possibility of early decision of this absence of solution: as soon as an automaton with empty language is computed during the algorithm there can not exist solutions to the considered problem.

Finally, we remarked, that a large part of the computation time of Distoplan is dedicated to the construction of the automata representation of the problem considered. This was in particular the case for the optical telegraph problems. Once this construction was achieved the message passing algorithm ran very fast. This suggest that for building a competitive version of Distoplan we should focus on this aspect, potentially using representations of automata adapted to planning.

Algorithm 8 Projection

Input: a weighted automaton $\mathcal{A} = (S, S^I, S^F, \Sigma, T, c, c^i, c^f)$ and a set of actions Σ'
Output: a weighted automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}')$ is the projection of $\mathcal{L}(\mathcal{A})$ on Σ'

*/*Initialization*/*
$S' \leftarrow S$
$T' \leftarrow \{t \in T \ : \ \sigma_t \in \Sigma'\}$
$S^{I\prime} \leftarrow S^I$
$S^{F\prime} \leftarrow S^F$
for all $t \in T'$ **do**
  $c'(t) \leftarrow c(t)$
end for
for all $s \in S^{I\prime}$ **do**
  $c^{i\prime}(s) \leftarrow c^i(s)$
end for
for all $s \in S^{F\prime}$ **do**
  $c^{f\prime}(s) \leftarrow c^f(s)$
end for
*/*Closure computation from each state*/*
for all $s \in S$ **do**
  $closure_{\Sigma\setminus\Sigma'}(s) \leftarrow \emptyset$
  $d(s) \leftarrow 0$
  while $S \neq \emptyset$ and $\exists s \in S, d(s) < +\infty$ **do**
    select $s' \in S$ such that $d(s')$ is minimal
    $S \leftarrow S \setminus \{s'\}$
    $closure_{\Sigma\setminus\Sigma'}(s) \leftarrow closure_{\Sigma\setminus\Sigma'}(s) \cup \{(s', d(s'))\}$
    for all $t \in T, t^- = s' \wedge \sigma_t \in \Sigma \setminus \Sigma'$ **do**
      $d(t^+) \leftarrow \min(d(t^+), d(s') + c(t))$
    end for
  end while
end for
*/*Addition of new transitions and new final states*/*
for all $s \in S$ **do**
  for all $(s', c) \in closure_{\Sigma\setminus\Sigma'}(s)$ **do**
    $T'' \leftarrow \{(s, \sigma, s'') \ : \ \sigma \in \Sigma' \wedge \exists (s', \sigma, s'') \in T\}$
    $T' \leftarrow T' \cup T''$
    for all $t \in T''$ **do**
      $c'(t) \leftarrow \min(c'(t), \min_{t'=(s'',\sigma_t,s')\in T} c + c(t'))$ */*$c'(t) = +\infty$ when not defined*/*
    end for
    if $s' \in S^F$ **then**
      $S^{F\prime} \leftarrow S^{F\prime} \cup \{s\}$
      $c^{f\prime}(s) \leftarrow \min(c^{f\prime}(s), c + c_f(s'))$ */*$c^{f\prime}(s) = +\infty$ when not defined*/*
    end if
  end for
end for
return $\mathcal{A}' = (S', S^{I\prime}, S^{F\prime}, \Sigma', T', c', c^{i\prime}, c^{f\prime})$

Algorithm 9 Product

Input: $\mathcal{A}_1 = (S_1, S_1^I, S_1^F, \Sigma_1, T_1, c_1, c_1^i, c_1^f)$, $\mathcal{A}_2 = (S_2, S_2^I, S_2^F, \Sigma_2, T_2, c_2, c_2^i, c_2^f)$
Output: $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A})$ is the product of $\mathcal{L}(\mathcal{A}_1)$ and $\mathcal{L}(\mathcal{A}_2)$

*/*Initialization*/*
$S \leftarrow S_1^I \times S_2^I$; $S^I \leftarrow S$
for all $(s_1, s_2) \in S^I$ **do**
  $c^i((s_1, s_2)) = c_1^i(s_1) + c_2^i(s_2)$
end for
$S^F \leftarrow \emptyset$; $T \leftarrow \emptyset$
$Q \leftarrow S$
*/*Search in $\mathcal{A}$ from $\mathcal{A}_1$ and $\mathcal{A}_2$*/*
while $Q \neq \emptyset$ **do**
  select $(s_1, s_2) \in Q$ and mark it
  $Q \leftarrow Q \setminus \{(s_1, s_2)\}$
  */*New final states*/*
  if $s_1 \in S_1^F$ and $s_2 \in S_2^F$ **then**
    $S^F \leftarrow (s_1, s_2)$
    $c^f((s_1, s_2)) \leftarrow c_1^f(s_1) + c_2^f(s, 2)$
  end if
  */*Private transitions of $\mathcal{A}_1$*/*
  for all $t_1 \in T_1$ such that $t_1^- = s_1$ and $\sigma_{t_1} \notin \Sigma_2$ **do**
    $T \leftarrow T \cup \{((s_1, s_2), \sigma_{t_1}, (t_1^+, s_2))\}$
    $c(((s_1, s_2), \sigma_{t_1}, (t_1^+, s_2))) \leftarrow c_1(t_1)$
    if (t_1^+, s_2) is not marked **then**
      $Q \leftarrow Q \cup \{(t_1^+, s_2)\}$
    end if
  end for
  */*Private transitions of $\mathcal{A}_2$*/*
  for all $t_2 \in T_2$ such that $t_2^- = s_2$ and $\sigma_{t_2} \notin \Sigma_1$ **do**
    $T \leftarrow T \cup \{((s_1, s_2), \sigma_{t_2}, (s_1, t_2^+))\}$
    $c(((s_1, s_2), \sigma_{t_2}, (s_1, t_2^+))) \leftarrow c_2(t_2)$
    if (s_1, t_2^+) is not marked **then**
      $Q \leftarrow Q \cup \{(s_1, t_2^+)\}$
    end if
  end for
  */*Shared transitions*/*
  for all $t_1 \in T_1, t_2 \in T_2$ such that $(t_1^-, t_2^-) = (s_1, s_2)$ and $\sigma_{t_1} = \sigma_{t_2}$ **do**
    $T \leftarrow T \cup \{((s_1, s_2), \sigma_{t_1}, (t_1^+, t_2^+))\}$
    $c(((s_1, s_2), \sigma_{t_1}, (t_1^+, t_2^+))) \leftarrow c_1(t_1) + c_2(t_2)$
    if (t_1^+, t_2^+) is not marked **then**
      $Q \leftarrow Q \cup \{(t_1^+, t_2^+)\}$
    end if
  end for
end while
return $\mathcal{A} = (S, S^I, S^F, \Sigma_1 \cup \Sigma_2, T, c, c^i, c^f)$

Chapter 4

Turbo Algorithms for Factored Planning

chapter abstract: *In this chapter we present an experimental study of the use of the famous turbo methods for planning. Their interest is to permit the use of message passing algorithms on problems not living on trees. We obtain encouraging results by using these methods in Distoplan. This validates the interest of approximate methods (and in particular turbo methods) for factored planning.*

IN THE PREVIOUS CHAPTERS we proposed a new approach to factored planning. The main interest of this approach, based on a message passing algorithm, with regards to previous results in the domain of factored planning is that it allows one to perform factored *cost-optimal* planning. However, as other existing approaches, our approach is based on strong hypothesis on the shape of the communication graphs of the factored planning problems. Namely, these graphs have to be trees. This restriction, even if natural for propagating constraints, is quite limitative. It significantly reduces the range of problems on which our method can be applied.

There exists however a family of algorithms (called turbo algorithms) which gave astonishingly good results when dealing with complex communication graphs (containing cycles) with sparse interaction. These algorithms are mostly known for their use in coding theory [6]. They are closely related (see [70]) to Pearl's belief propagation algorithm, well know in the artificial intelligence community [77] . Nowadays, the turbo algorithms are used with great success in many area of digital communications and signal processing. However, their efficiency is not fully explained theoretically yet.

This chapter presents a study of the turbo algorithms in the context of factored planning using message passing algorithms. Our goal is to show that these algorithms are of interest in the domains of constraint satisfaction (factored planning) and of optimization (factored cost-optimal planning). In fact they will allow one to compute over-approximations of the sets of solutions.

After a brief description of the concept of turbo algorithms and a remark on solution extraction from the updated factors obtained by applying message passing on a factored planning problem (Section 4.1), we consider the use of turbo algorithms for solving factored planning problems (Section 4.2) and their cost-optimal counterpart (Section 4.3). In each case, experimental results are provided that demonstrate

the interest of investigating the use of approximate methods in the domain of factored planning.

4.1 Turbo algorithms

The idea of the turbo algorithms, in our context, is to simply run the message passing algorithms on problems for which the communication graph is not a tree. In general there is no guarantee that, by doing this, convergence will be achieved. However, it is possible to stop the update of the messages by using other notions of convergence than strict stability of their languages. Such methods are discussed in the next sections of this chapter.

4.1.1 About updated components

Consider a factored planning problem defined by the automata $\mathcal{A}_1, \dots, \mathcal{A}_n$. After having stopped updates of messages (using any manner of deciding convergence) components of the considered problem can be updated as if messages stabilized. For each $\mathcal{A}_i$ component of the planning problem an updated component $\mathcal{A}_i'' = \mathcal{A}_i \times_{\mathcal{A}\, j \in \mathcal{N}(i)} \mathcal{M}_{j,i}$ can be computed. If the problem does not live on a tree there is no guarantee that these $\mathcal{A}_i''$ have for language the projection on Σ_i of $\mathcal{L}(\mathcal{A}_1 \times_{\mathcal{A}} \dots \times_{\mathcal{A}} \mathcal{A}_n)$. From now on we denote by $\mathcal{A}_i''$ the automata obtained from message passing algorithm and by $\mathcal{A}_i'$ some automata such that $\mathcal{L}(\mathcal{A}_i')$ is the projection on Σ_i of $\mathcal{L}(\mathcal{A}_1 \times_{\mathcal{A}} \dots \times_{\mathcal{A}} \mathcal{A}_n)$.

What is fundamental to notice about the $\mathcal{L}(\mathcal{A}_i'')$ is that they are over-approximations of the $\mathcal{L}(\mathcal{A}_i')$: $\forall (w,c) \in \mathcal{L}(\mathcal{A}_i'), \exists c', (w,c') \in \mathcal{L}(\mathcal{A}_i'')$. Moreover, these $\mathcal{L}(\mathcal{A}_i'')$ are refinements of the $\mathcal{L}(\mathcal{A}_i)$: $\forall (w,c) \in \mathcal{L}(\mathcal{A}_i''), \exists c', (w,c') \in \mathcal{L}(\mathcal{A}_i)$. This is because only part of the constraints on $\mathcal{A}_i$ are taken into account. Turbo algorithms will thus be of interest if the $\mathcal{A}_i''$ they allow to compute give good approximations of the $\mathcal{A}_i'$. This is this quality of the $\mathcal{A}_i''$ that we investigate in this chapter.

4.1.2 About solution extraction

If Algorithm 6 converges, one is able to extract a solution to the considered factored planning problem from the $\mathcal{A}_i''$ obtained.

When the communication graphs are trees this is straightforward. One can simply apply the method proposed in the previous chapters. Because $\forall i, \mathcal{L}(\mathcal{A}_i'') = \mathcal{L}(\mathcal{A}_i')$ and the communication graph is a tree, one can ensure that the tuple of paths found is indeed a solution of the factored planning considered.

However, when the communication graphs are not trees, one can not ensure that $\mathcal{L}(\mathcal{A}_i'') = \mathcal{L}(\mathcal{A}_i')$. And, even if this is the case, one can not ensure that a solution will be found using the same method as for trees. Finding a solution may require the use of *backtracking*. At some step of the solution extraction, one may be unable to choose a path π_j in $\mathcal{A}_j''$, compatible with the previously chosen paths $\pi_{i_1}, \dots, \pi_{i_k}$ in its neighbors. This would require the modification of at least one of the π_{i_ℓ} (and potentially also the modification of the paths that where used for computing π_{i_ℓ}).

4.2 Turbo algorithms for constraint solving

We first focus on the case of factored planning problems without costs. Simply replacing weighted automata by automata in the definition of factored cost-optimal planning

problems defines such problems. It is straightforward to prove that the message passing algorithms can be used in this case: simply consider weighted automata in which all costs are null. In this case, determinization and minimization of automata are always possible. We thus consider only minimal deterministic automata, which allows us to confound automata and their languages.

This section recalls some known results about turbo algorithms for constraint solving, and show how these results relate to the specific case of planning. Finally our experimental setting is presented and experimental results are given.

4.2.1 Conditions for convergence

In [27] turbo algorithms are studied in detail for the case of "systems defined by local constraints". Factored planning problems belong to this class of systems. Hence, if a partial order $\sqsubseteq$ exists on automata verifying the following axioms, then existence of a unique stabilization point for Algorithm 6 on any factored planning problem is ensured (Lemma 7 of [27]). The first axiom ensures the existence of a least informative system:

$$\exists \mathbb{I}, \forall \mathcal{A}, \mathcal{A} \sqsubseteq \mathbb{I}. \tag{4.1}$$

The second axiom ensures that the synchronous product adds the same amount of information to all the systems:

$$\forall \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_1 \sqsubseteq \mathcal{A}_2 \Rightarrow \mathcal{A}_1 \times \mathcal{A}_3 \sqsubseteq \mathcal{A}_2 \times \mathcal{A}_3. \tag{4.2}$$

Finally, the third axiom ensures that the projection does not add information:

$$\forall \mathcal{A}_1, \mathcal{A}_2, \forall \Sigma, \mathcal{A}_1 \sqsubseteq \mathcal{A}_2 \Rightarrow \mathcal{P}_\Sigma \mathcal{A}_1 \sqsubseteq \mathcal{P}_\Sigma \mathcal{A}_2. \tag{4.3}$$

The idea behind these axioms is that applying Algorithm 6 to a factored planning problem will result into messages having a decreasing evolution with respect to $\sqsubseteq$. There is an obvious partial order on automata verifying these axioms: $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ if and only if $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$.

If, in addition to the existence of a partial order verifying axioms 4.1, 4.2, and 4.3, the number of possible automata is bounded, convergence of Algorithm 6 is ensured within a finite number of steps on any factored planning problem. Indeed, messages decrease for $\sqsubseteq$ and there is a bounded number of possible messages (Theorem 3 of [27]).

It is, however, not the case that the number of languages (and thus of automata) over a given alphabet is bounded. Thus, it is not possible to ensure convergence of Algorithm 6 on any factored planning problem. As an example, consider the problem in Figure 4.1. This problem has no solution: reaching the goal in $\mathcal{A}_1$ implies the firing of an α, which implies the firing of a γ in $\mathcal{A}_3$ and thus the firing of a β in $\mathcal{A}_2$, which enforces the firing of a second α in $\mathcal{A}_1$ and thus the firing of a second γ, and so on. When using message passing on this problem, the message $\mathcal{M}_{1,3}$ from $\mathcal{A}_1$ to $\mathcal{A}_3$ will be progressively updated but will never reach stability. Figure 4.2 represents this message after n updates.

4.2.2 Ensuring convergence in all cases

However, one needs to be able to stop computation in all cases. This can be achieved by using a notion of distance d between languages and deciding convergence as soon as the messages are stable up to some constant with respect to this distance. In other

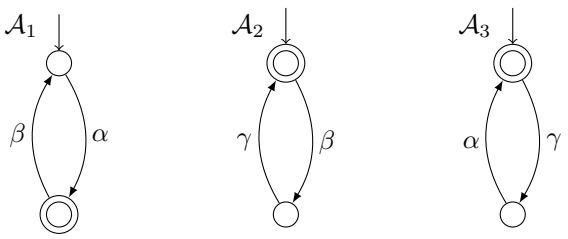


Figure 4.1: A factored planning problem such that turbo algorithms do not converge.

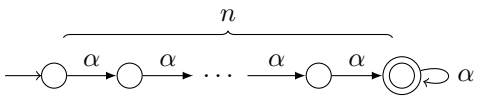


Figure 4.2: $\mathcal{M}_{1,3}$ after n updates of all the messages, assuming $\mathcal{M}_{3,2}$ and then $\mathcal{M}_{2,1}$ are updated between each two updates of $\mathcal{M}_{1,3}$

words, Algorithm 6 will stop as soon as the updating of each message $\mathcal{M}_{i,j}$ results in the new message $\mathcal{M}'_{i,j}$ such that $d(\mathcal{M}_{i,j}, \mathcal{M}'_{i,j}) \leq \epsilon$, for some small ϵ.

A distance which seems reasonable is the following:

$$d(\mathcal{A}_1, \mathcal{A}_2) = \sum_{n=0}^{\infty} \frac{1}{2^n} 1_{\mathcal{L}_n(\mathcal{A}_1) \neq \mathcal{L}_n(\mathcal{A}_2)}$$

where $1_{\mathcal{L}_1 \neq \mathcal{L}_2} = 1$ as soon as $\mathcal{L}_1 \neq \mathcal{L}_2$ and 0 in other cases, and $\mathcal{L}_n(\mathcal{A}) = \{w \in \mathcal{L}(\mathcal{A}) : |w| = n\}$ is the set of words of length n belonging to $\mathcal{L}(\mathcal{A})$. Using this distance almost corresponds at looking only at words under a given length ℓ for checking convergence. As, for any alphabet, the set of words of length smaller than ℓ is finite, and any update of messages in Algorithm 6 can only remove words from the considered messages, convergence is granted using this distance. Moreover, in any planning problem, there exists a bound (difficult to compute in a modular way) such that if in a component there exists no plan with length smaller than this bound, the problem has no solution. This ensures that convergence with respect to this distance (using a correct bound) is sufficient for deciding the absence of a solution.

4.2.3 Experimental results

In this section we present some experimental results obtained by running Algorithm 6 on randomly generated factored planning problems for which communication graphs have cycles. Our goal is to estimate if the over-approximations of the $\mathcal{A}'_i$ computed by the message passing algorithms are of interest for planning. That is, if they allow to find solutions with few backtracking steps.

In all our experiments we randomly generate some factored planning problems. This is done by choosing a shape for the interaction graph of the problem, and then, for each node of this graph, randomly generating an automaton. These automata have up to 20 states and up to three time more transitions. Each automaton shares 2 different labels with each of its neighbors in the interaction graph. Once a problem is generated, we check if it is not trivially solvable (that is no solution can be found without backtracking before updating automata). If it is not trivially solvable we check if it is solvable

by searching for a path into the full product $\mathcal{A}$ of the component automata. Only problems which are not trivially solvable but still have a solution are considered for our experiments. In doing this we select the 5-10% "most complicated" problems among the ones we generate.

Experiment 1: automata on a circle.

For our first experiment we chose a shape of communication graph known to be well suited for turbo methods: a circle. The setting for this experiment is as follows: we consider a circle of n automata (randomly generated as described above) $\mathcal{A}_0, \ldots, \mathcal{A}_{n-1}$ such that $\forall 0 \leq i < n$ one has $\Sigma_i \cap \Sigma_{(i+1)\%n} \neq \emptyset$ (here $\%$ stands for modulo) and $\forall j$, if $|i - j| > 1$ then $\Sigma_i \cap \Sigma_j = \emptyset$. We call an *iteration* an update of all the messages. Iterations are performed in the following way: first the messages of the form $\mathcal{M}_{i,(i+1)\%n}$ are computed by increasing values of i, and then the messages of the form $\mathcal{M}_{i,(i-1)\%n}$ are computed by decreasing values of i. Figure 4.3 presents the communication graph corresponding to this setting for five automata, it also shows in which order the messages are updated in this particular case (for example, $\mathcal{M}_{2,3}$ is updated in third place). Such iterations are repeated until stabilization of the messages, that is when no more messages have been modified during an iteration. As soon as stabilization is reached an attempt to find a solution is initiated. It is done using the method presented in Section 4.1.2 without using backtracking. For this reason, even if all problems have solution, it is not always the case that a solution will be found. Our goal using this method is to get an idea on the quality of the computed $\mathcal{A}_i''$: if in many cases a solution is found, it is likely that in practice few backtracking steps will be needed for finding plans.

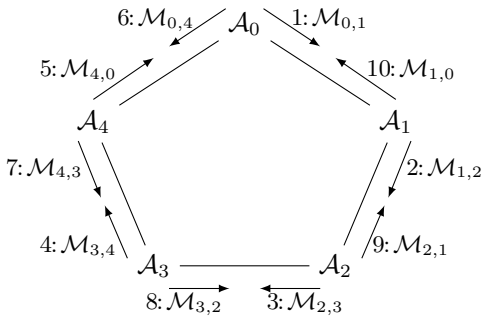


Figure 4.3: Experimental setting and propagation of messages during one iteration for five automata.

Figure 4.4 presents the results obtained in this setting for circles of 3 to 15 automata. In each case, 50 circles were considered. For each circle size the percentage of problems for which convergence occurred in 1, 2, or 3 iterations is represented, as well as the percentage of problems in which a solution (i.e. a plan) has been found (without using backtracking).

We limited our experiments to circles of up to 15 automata because of the method used for selecting problems. Selecting problems is in fact very time consuming. Finding a difficult problem as described above frequently requires to generate more than 20 problems and test them by searching solutions without using turbo algorithms. We however performed some experiments with larger circles, as Algorithm 6 is able to han-

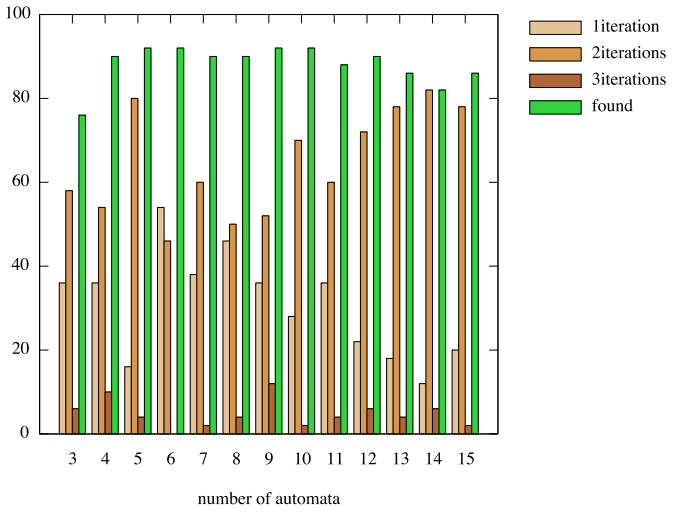


Figure 4.4: Experimental results for circles of 3 to 15 automata: number of iterations before convergence and percentage of solutions found are presented.

dle them. The results are presented in Figure 4.5. For each value of n, 50 cycles were generated but no selection was done among them. For each circle size the percentage of problems for which convergence occurred in 1, 2, or 3 iterations is represented. Due to the limitations presented above, we can not give the percentage of cases where a solution was found (in fact some of the problems considered may have trivial solutions or no solution).

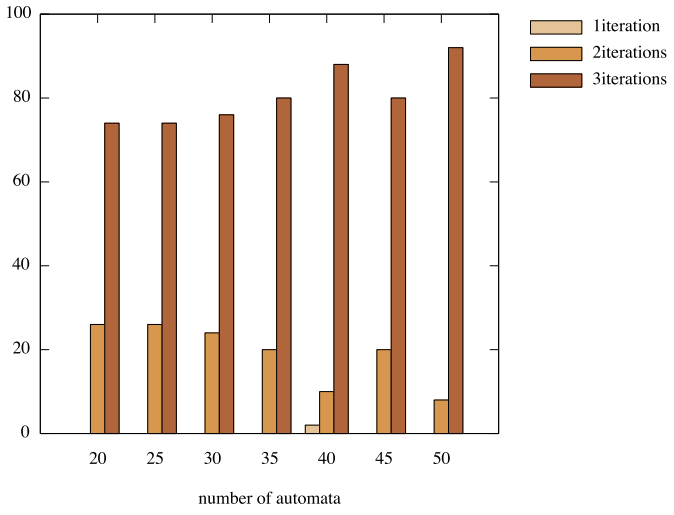


Figure 4.5: Experimental results for circles of more than 15 automata: number of iterations before convergence are presented.

It may seem odd that, in none of the cases considered more than 3 iterations were required for converging. Thus, we performed a much larger experimentation: 10000 cycles of 3 automata were generated (no selection was performed among them). In this case we found: 56.01% of convergences within 1 iteration, 35.04% within 2 iterations, 8.92% within 3 iterations, and 0.03% within 4 iterations.

To conclude, this first experiment gave promising results. In particular a convergence within very few iterations in all cases. And more than 80% of success in searching plans in updated components for circles of more than three automata, without using backtracking at all.

Experiment 2: automata on a tetrahedron.

In this setting, we considered problems with four automata (generated as described above) for which the communication graph is a tetrahedron. The goal here is to try more complex shapes than just circles. An iteration is performed by updating each message exactly once. These updates are done along a path in the communication graph which takes each edge only once in each direction. This is shown in Figure 4.6.

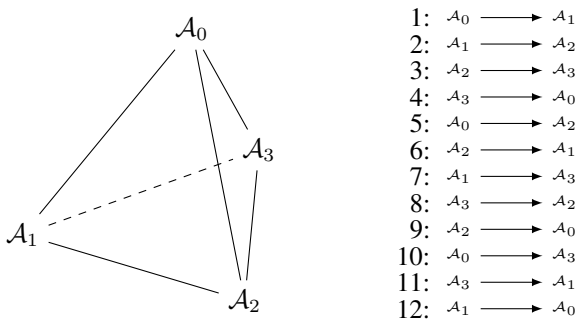


Figure 4.6: Automata on a communication graph with tetrahedra shape (left) and order of message updates during an iteration (right).

We obtained the following results for 50 "difficult" test cases (selected as described above): a quick convergence in all cases (2% within 1 iteration, 52% within 2 iterations, 42% within 3 iterations, and 4% within 4 iterations), and a solution found with no backtracking in 85% of the cases. The slightly slower convergence could come from the way iterations are performed, which may be less efficient than in the circles of automata, or from the stronger interaction in this second setting.

These results lead us to think that using the message passing algorithms directly on problems for which the communication graphs are not trees may give good results as soon as the interactions are sufficiently sparse. In particular, searching for plans in problems updated by Algorithm 6 using backtracking may allow one to find a solution with actually very few backtracks: in our experiments a solution was frequently found with no backtracking at all.

4.3 Turbo algorithms for cost-optimal planning

In this section we extend the results of previous section by suggesting solutions for applying turbo algorithms to systems with quantitative aspects (in our case factored

cost-optimal planning problems). The main issue to deal with is that cycles in the communication graphs usually result in the costs being counted several times along execution of Algorithm 6. This implies that one should use some normalization procedure after taking the products. The first part of this section formally justifies the need for such a normalization. The second part explains concretely how to perform normalization and gives a possible normalization constant. The third part consists of further experimental results.

4.3.1 Necessity of a normalization

Consider the example in Figure 4.7. During the update of the messages, the cost of α will grow unbounded. Consider, for example, an update in the following order: $\mathcal{M}_{1,2}$ followed by $\mathcal{M}_{2,3}$, and finally $\mathcal{M}_{3,1}$. Figure 4.8 presents message $\mathcal{M}_{1,2}$ after n such updates. This example shows that, in any factored planning problem where costs of solutions are not null, it is hopeless to expect to see any stabilization of the messages. This suggests the need for a normalization procedure after taking synchronous product, in order to keep costs of paths within a reasonable scope (one could imagine to track costs associated to each action, but the loss of information during the projection prevents this approach).

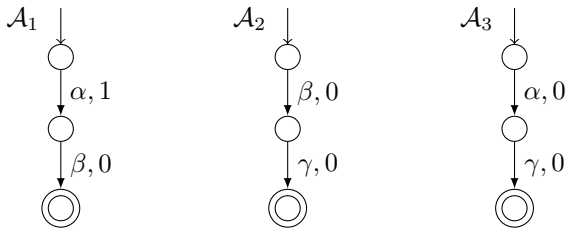


Figure 4.7: An example where stabilization is not possible in presence of costs on transitions.

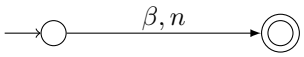


Figure 4.8: Message $\mathcal{M}_{1,2}$ after n updates.

In our setting, one can imagine two ways for normalizing an automaton: either (1) adding a (negative) constant to the cost of each path, or (2) dividing the cost of each path by a constant. The main interest of (2) is that it is the easier to perform in practice: to divide the cost of all the paths of an automaton by a constant c it is sufficient to divide the cost of each transition by c. However, the downside is that the difference of costs between paths is changed. In our case where costs are additive, it may result in choosing the wrong path. For example, see Figure 4.9: originally the best solution consisted of firing an α in each automaton, but after dividing the costs of the paths in $\mathcal{A}_1$ by $c = 3$, the best solution is a firing of β in each automaton.

By contrast, (1) gives small costs to local paths which already have the smallest costs. It concentrates the smallest costs on the paths that are potentially part of a cost-optimal solution. In this way, constraint solving will be helped by costs (paths with very high costs do not have to be considered) and solution extraction should also be

improved (the smallest cost path in each automaton will likely be part of a solution, while the random path considered in the previous case has no particular reason of being part of a solution). For these reasons we will use (1) rather than (2).

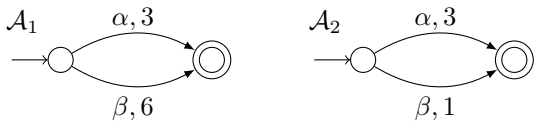


Figure 4.9: Dividing costs is not acceptable.

In the next part we describe concretely how to add a cost to each path of a weighted automaton. We also suggest a possible constant to use for normalization.

4.3.2 Normalization procedure

The idea of our normalization procedure is to add a (negative) constant to the cost of each path of the automaton that is being normalized. A simple way to do this is to add the constant to the initial cost of each initial state of the considered automaton. However, this may lead to automata where there is a huge negative initial cost which compensates for a huge cost for each path. Thus we suggest that, once the constant has been added to the initial costs, it is propagated throughout the automaton, so that costs of transitions do not grow too much. This can be done by pushing this cost toward final states, using algorithms similar to the weight pushing algorithm from [76].

A possible normalization constant is the cost of a shortest path minus one. Using this constant ensures that, after each normalization, the cost of the shortest path(s) is one. This has the interest of guaranteeing that a normalized automaton with no negative costs on transitions always exists. Moreover, this constant is simple to compute due to the small size of the automata involved in the computations along a standard execution of Algorithm 6.

Using this normalization it is very likely that no stabilization will be reached when applying Algorithm 6 to systems with costs. The fact is that normalization (as expected) only stabilizes paths with smallest costs: the difference between the smallest costs and the other costs increases when updating messages. To decide when the execution of Algorithm 6 should stop on a given problem, a distance similar to the one proposed in Section 4.2.2, can be used, thus checking equality of words for a given cost rather than for a given length.

4.3.3 Experimental results

As in the case of factored planning, we generated factored cost-optimal planning problems for our experiments. The automata generated have almost the same characteristics as before, except that they have random costs on transitions. We still consider the "difficult" problems only by eliminating the ones with no solutions and generating automata with many paths with different costs. This way finding a solution may be easy, but finding the best one is more difficult.

In all our experiments, the condition for stopping the algorithm is now stability of the cost-optimal path in each component: if, after an iteration, the cost-optimal plans remain the same (in each component) as they were before the iteration, then stability is considered to be reached. In order to select a solution, one proceeds as follows.

First a (locally) cost-optimal path π_0 is chosen in $\mathcal{A}_0$, and send it to its neighbors as before. Then a (locally) cost-optimal path compatible with π_0 is chosen in each neighbor, and propagated as well. This is done until a path is found in each component, or no compatible path can be found in a given component. We still do not perform backtracking. If a solution is found, one searches for a cost-optimal solution using a centralized approach and compares the costs of the two solutions, in order to estimate the quality of the solution found using the turbo approach.

Experiment 1: automata on a circle.

As was the case in the absence of costs, we first consider automata for which the only communication graph is a circle. We look at circles of 3 to 7 components, trying 20 different circles in each case. We do not test as many problems as for the no-cost case because generating difficult problems requires much more time in practice when there are costs. Minimization of weighted automata may not terminate, and not minimizing significantly increases the computation time in many cases. Convergence rapidity in terms of number of iterations is almost the same as in the no-cost case. Thus, we focus on the percentage of solutions found, and in the cases where a solution is found, on the quality of this solution. Results for this experiment are presented in Figure 4.10. Among the percentage of solutions found, the amount of solutions of different qualities are represented. These qualities are of the form $x - y\%$, meaning that the cost of a solution of this quality is between $x\%$ (exclusive) and $y\%$ greater than the cost of a cost-optimal solution.

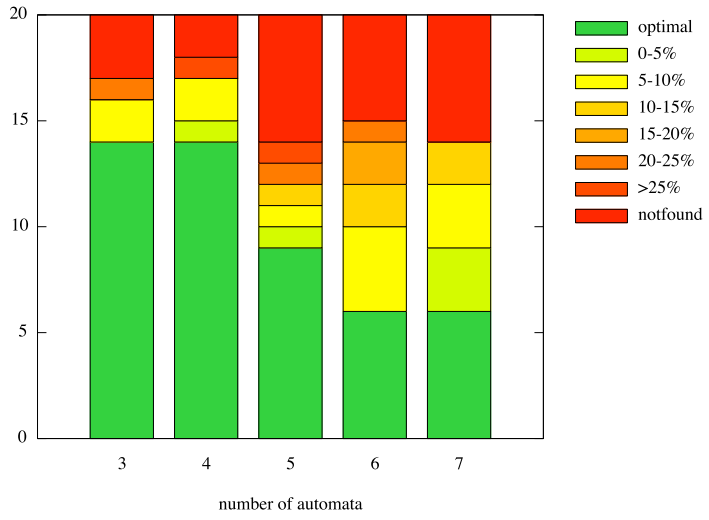


Figure 4.10: Experimental results for circles of 3 to 7 weighted automata.

The fact that less solutions are found than in the no-cost case is probably due to the very restrictive condition used for deciding stability. This fact excepted, the results obtained are encouraging, with a large part of the solutions found being of decent quality (less than 5% greater than a cost-optimal solution).

Experiment 2: automata on a tetrahedron.

We also generated problems where automata are on a tetrahedron, as was the case with no costs. The results obtained on 50 such problems are as follows. There was convergence within few iterations in almost all cases (except in one experiment, stabilization was always reached after 2 or 3 iterations). Moreover, a solution was found without backtracking in 68% of our experiments. Among the 34 solutions found, 17 were optimal. More complete results are presented in Table 4.1.

found	opt	0-5%	5-10%	10-15%	15-20%	>20%
34	17	0	7	3	4	3

Table 4.1: Experimental results for 50 factored cost-optimal planning problems on tetrahedra.

The experimental results obtained here by applying turbo algorithms to factored cost-optimal planning problems for which the communication graphs are not trees show that these algorithms are of interest. In particular, we frequently obtained optimal or close-to-optimal solutions in "difficult" problems. Moreover, these solutions were frequently found quickly (performances were comparable to what we obtained in Chapter 3 for factored cost-optimal planning problems for which the interaction graph was a tree).

Conclusion

In this chapter we presented experimental results on the use of turbo algorithms for factored planning and factored cost-optimal planning. This was done by executing these algorithms on randomly-generated problems. The results obtained thus far are very encouraging.

In the case of constraint solving (factored planning), the algorithms converged in very few iterations on many test cases. Moreover, in almost all cases, solutions have been obtained quickly on networks of up to 50 small automata, which are hardly manageable using a centralized approach. This corresponds to finding a path in an automaton with up to 10^{50} states.

In the case of optimization (factored cost-optimal planning) the algorithms converged in few iterations as well. The solutions obtained were frequently optimal, or close-to-optimal. This can be explained by the fact that the costs of the solutions which are not likely to be optimal diverge along an execution of the algorithm.

One should, however, consider these results carefully as they were obtained using a random problem generator. This generator could generate problems that are particularly well-suited for turbo algorithms. It may be the case that on real problems, turbo algorithms converge slowly, or do not filter sufficiently many wrong plans to ensure a quick isolation of a solution after convergence. However, we believe that the experimental results presented in this chapter show that turbo algorithms may render accessible problems which are otherwise intractable with standard centralized approaches. This should, at least, be a reason for considering these algorithms in the context of planning.

Chapter 5

Networks of Automata with Read Arcs

chapter abstract: *Real planning problems may have actions that can only be performed in one component when another component is in a specific state (an action can read some variables without modifying them). This chapter proposes a mechanism to capture this phenomenon, under the form of automata with read arcs. It is shown that the message passing algorithms can be extended to this new setting.*

THE WORK PRESENTED IN Chapter 2 suffers from a weakness. Planning problems often specify that an action on some variable (or set of atoms) V_n can only be performed if another variable V_m has some specific value. For example, loading a truck requires the presence of the truck, but it does not change the truck location. Moreover, the presence of the truck may also enable another concurrent action, like filling it up. This ability to read a variable is not encoded by standard automata interactions. Consider the example in Figure 5.1, all actions in $\mathcal{A}_1$ and $\mathcal{A}_2$ can only be performed if $\mathcal{A}_3$ is in its initial (and final) state. To model this, one needed to introduce loops in $\mathcal{A}_3$ that synchronize with α, β and γ. As a consequence, for $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ to jointly reach their goal, $\mathcal{A}_3$ must perform one word in $\{\gamma, \alpha\alpha\beta, \alpha\beta\alpha, \beta\alpha\alpha\}$, which forces readings of its state variable to be displayed and ordered. In particular, concurrent readings become impossible.

In this chapter we propose a mechanism that solves this difficulty: it extends the semantics of automata interactions to enable state readings, without forcing a component to fire a transition in order to display its state. In the previous example, this will allow $\mathcal{A}_3$ to stay idle and simply "show" its state to enable actions in $\mathcal{A}_1$ and $\mathcal{A}_2$.

Section 5.1 illustrates the principle of this construction in the simplest setting, while Section 5.2 extends it to the case of networks of automata, with cross and simultaneous readings. Section 5.3 proves that this setting enjoys the right algebraic properties that enable the distributed computation of factored plans. Section 5.4 generalizes the approach to any number of automata. For simplicity of presentation, all these results are described in the context of factored planning without costs.

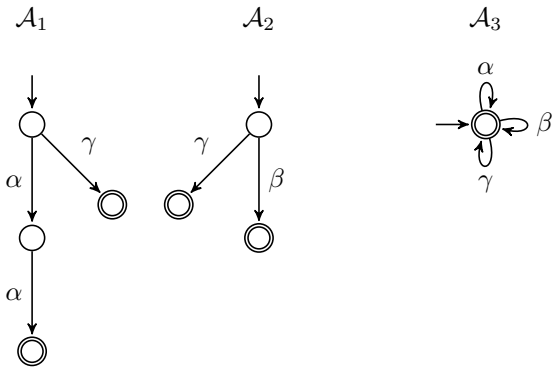


Figure 5.1: Network of three partially synchronized automata.

5.1 Simple reading mechanism

This section illustrates how a reading mechanism can be introduced to define automata interactions. The setting is first limited to the simple case of two automata. The next sections extend it to an arbitrary number of automata, with cross and multiple readings.

5.1.1 Writing and reading

Beyond the simple composition of two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ by the usual synchronous product, one would like to allow a weak form of synchronization, where $\mathcal{A}_2$ is allowed to perform some of its transitions only when $\mathcal{A}_1$ is in specific states. This requires a double mechanism. First, $\mathcal{A}_1$ should display properties of its states. Rather than associating labels to states, it is equivalently assumed that transitions *output* a readable label. So let us define $\mathcal{A}_1 = (S_1, S_1^I, S_1^F, \Sigma_1 \times O_1, T_1)$ where O_1 is a finite set of readable labels, and so $T_1 \subseteq S_1 \times \Sigma_1 \times O_1 \times S_1$. For simplicity, only a single property is displayed by each state (that is produced by each transition). The language $\mathcal{L}(\mathcal{A}_1)$ is defined standardly (as in Section 2.1.1) by considering $\Sigma_1 \times O_1$ as alphabet. Secondly, $\mathcal{A}_2$ must be able to read these values. Let I_2 be the set of "input values" to $\mathcal{A}_2$, that one can define as $\mathcal{A}_2 = (S_2, S_2^I, S_2^F, I_2 \times \Sigma_2, T_2)$, with $\star \in I_2$, and so $T_2 \subseteq S_2 \times I_2 \times \Sigma_2 \times S_2$. The semantics is that a transition $t_2 = (s_2, a, \sigma, s_2') \in T_2$ will be able to fire only if $\mathcal{A}_1$ has displayed the value a at the output of one of its transitions; the special case of $a = \star$ means that t_2 is not reading in $\mathcal{A}_1$. Again, the language $\mathcal{L}(\mathcal{A}_2)$ is obtained by considering $I_2 \times \Sigma_2$ as alphabet.

To define the interactions of $\mathcal{A}_1$ and $\mathcal{A}_2$ based on this reading mechanism, let us first consider the composition of their languages. It consists of words over the alphabet $I_2 \times \Sigma \times O_1$ where $\Sigma = \Sigma_1 \cup \Sigma_2$.

Definition 5.1. *A word* $w = (i_1, \sigma_1, o_1)...(i_K, \sigma_K, o_K)$ *is* coherent *if and only if, for* $1 \leq k \leq K$*, one has*

1. $i_k = o_{k-1}$ *or* $i_k = \star$ *(in particular* $i_1 = \star$*),*
2. $o_k = o_{k-1}$ *if* $\sigma_k \notin \Sigma_1$*, and*
3. $i_k = \star$ *if* $\sigma_k \notin \Sigma_2$*.*

In other words, (1) expresses that if label σ_k is attached to a reading, the previous label must have provided this value as output. Condition (2) expresses that transitions labeled by $\Sigma_2 \setminus \Sigma_1$ (which correspond to private transitions of $\mathcal{A}_2$) can not change (or must propagate) the output produced by $\mathcal{A}_1$. Symmetrically, (3) expresses that private transitions of $\mathcal{A}_1$ can not be associated to a reading.

5.1.2 Operations on languages

Definition 5.2. *The* projection Π_2 *of words over* $I_2 \times \Sigma \times O_1$ *on* $I_2 \times \Sigma_2$ *is defined as the monoid morphism generated by* $\Pi_2(i, \sigma, o) = (i, \sigma)$ *if* $\sigma \in \Sigma_2$*, else* $\Pi_2(i, \sigma, o) = \varepsilon$.

Definition 5.3. *Similarly, the* projection Π_1 *on* $\Sigma_1 \times O_1$ *is the monoid morphism generated generated by* $\Pi_1(i, \sigma, o) = (\sigma, o)$ *if* $\sigma \in \Sigma_1$*, otherwise* $\Pi_1(i, \sigma, o) = \varepsilon$.

Definition 5.4. *The* composition *or* product *of languages* $\mathcal{L}(\mathcal{A}_1) \times_L \mathcal{L}(\mathcal{A}_2)$ *is defined by all coherent words* w *over alphabet* $I_2 \times \Sigma \times O_1$ *such that* $\Pi_1(w) \in \mathcal{L}(\mathcal{A}_1)$ *and* $\Pi_2(w) \in \mathcal{L}(\mathcal{A}_2)$.

Notice that this definition extends the natural synchronous product of languages, where a letter $\sigma \in \Sigma_1 \cap \Sigma_2$ corresponds to synchronous actions of two languages. Here, "letters" (i, σ) and (σ, o) in $\mathcal{L}(\mathcal{A}_2)$ and $\mathcal{L}(\mathcal{A}_1)$ respectively give rise to the synchronous action (i, σ, o) that both needs input i and produces output o.

As an example, let $\mathcal{L}(\mathcal{A}_1) = \{w_1 = (\alpha, 1)(\gamma, 2)(\alpha, 3)(\delta, 4)\}$ and $\mathcal{L}(\mathcal{A}_2) = \{w_2 = (1, \beta)(\star, \gamma)(\star, \beta)(3, \delta), w_2' = (1, \gamma)(1, \beta)\}$, with $\Sigma_1 = \{\alpha, \gamma, \delta\}, \Sigma_2 = \{\beta, \gamma, \delta\}$ and $I = O = \{1, ..., 4\}$. One has $\mathcal{L}(\mathcal{A}_1) \times_L \mathcal{L}(\mathcal{A}_2) = \{ w,\ w' \}$ (Figure 5.2) where

$$\begin{aligned} w &= (\star, \alpha, 1)(1, \beta, 1)(\star, \gamma, 2)(\star, \beta, 2)(\star, \alpha, 3)(3, \delta, 4) \\ w' &= (\star, \alpha, 1)(1, \beta, 1)(\star, \gamma, 2)(\star, \alpha, 3)(\star, \beta, 3)(3, \delta, 4) \end{aligned}$$

Observe that w_2' does not synchronize with w_1, due to the impossibility of reading again 1 after action γ has been performed in $\mathcal{A}_1$. By contrast, words w_1 and w_2 synchronize in two different ways, yielding w and w'. Notice that the readings in $\mathcal{A}_2$ constrain the interleaving of the private events of $\mathcal{A}_1$ and $\mathcal{A}_2$ (see the notion of asymmetric conflict in Petri nets with read arcs in [5, 4]). For example, the first occurrence of β has to be performed after first occurrence of α, while both are private. By contrast, since there is no reading in the second occurrence of β, it can be performed either before or after the second occurrence of α. Notice as well that writings in $\mathcal{A}_1$ are propagated along product words by private events of $\mathcal{A}_2$, which are attached to a "stuttering event" of $\mathcal{A}_1$ (circles in Fig. 5.2). Finally, observe that the usual synchronous product of languages is obtained by positioning all readings to $\star$.

5.1.3 Operations on automata

In standard automata theory, one has that $(\mathcal{L}(\mathcal{A}_1), \Sigma_1) \times_{\mathcal{L}} (\mathcal{L}(\mathcal{A}_2), \Sigma_2)$ is the language $\mathcal{L}(\mathcal{A}_1 \times_{\mathcal{A}} \mathcal{A}_2)$, where $\times_{\mathcal{A}}$ is the synchronous product of automata and $\times_{\mathcal{L}}$ the usual synchronous product of languages (they correspond to the products defined in Chapter 2 when all actions have cost 0) which can be defined as $\Pi_1^{-1}(\mathcal{L}(\mathcal{A}_1)) \cap \Pi_2^{-1}(\mathcal{L}(\mathcal{A}_2))$. As explained in Chapter 2, this property is essential to replace operations on regular languages, which are possibly infinite objects, by operations on their representations as automata, which are finite objects. To extend this property to the writing and reading

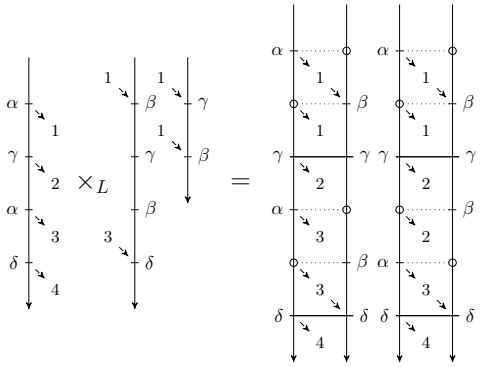


Figure 5.2: Product of languages. Arrows denote output and input values. Product words, on the right, are depicted as synchronized threads where circles denote stuttering events.

automata described above, one needs a notion of automaton with *internal* readings and writings.

Let $\mathcal{A} = (S, S^I, S^F, I \times \Sigma \times O, T)$ be an automaton with (internal) inputs and outputs, assuming $\star \in I$. A path $\pi = t_1 ... t_K$ in $\mathcal{A}$ is said to be *coherent* if and only if its label sequence $(I \times \Sigma \times O)(\pi)$ forms a coherent word over alphabet $I \times \Sigma \times O$. The coherent language of $\mathcal{A}$, denoted $\mathcal{L}_c(\mathcal{A})$, is now defined as the restriction of $\mathcal{L}(\mathcal{A})$ to its coherent words. One has $\mathcal{A} = \mathcal{A}_1 \times_A \mathcal{A}_2$ if and only if $S = S_1 \times S_2, S^I = S_1^I \times S_2^I, S^F = S_1^F \times S_2^F, I = I_2, \Sigma = \Sigma_1 \cup \Sigma_2, O = O_1$, and the transition set $T = T_s \cup T_{1,p} \cup T_{2,p}$ is defined by:

$$\begin{aligned} T_s &= \{\, ((s_1, s_2), i_2, \sigma, o_1, (s_1', s_2')) : \\ &\qquad (s_1, \sigma, o_1, s_1') \in T_1,\ (s_2, i_2, \sigma, s_2') \in T_2 \,\} \qquad (5.1) \\ T_{1,p} &= \{\, ((s_1, s_2), \star, \sigma_1, o_1, (s_1', s_2)) : \sigma_1 \notin \Sigma_2, \\ &\qquad (s_1, \sigma_1, o_1, s_1') \in T_1,\ s_2 \in S_2 \,\} \qquad (5.2) \\ T_{2,p} &= \{\, ((s_1, s_2), i_2, \sigma_2, o_1, (s_1, s_2')) : \sigma_2 \notin \Sigma_1, \\ &\qquad (s_2, i_2, \sigma_2, s_2') \in T_2,\ s_1 \in S_1,\ o_1 \in O_1 \,\} \qquad (5.3) \end{aligned}$$

Synchronized transitions in T_s correspond of course to $\sigma \in \Sigma_1 \cap \Sigma_2$, private transitions of $\mathcal{A}_1$ appear with no reading in $T_{1,p}$, while private transitions of $\mathcal{A}_2$ reproduce all possible outputs of $\mathcal{A}_1$ in $T_{2,p}$.

Proposition 5.1. $\mathcal{L}_c(\mathcal{A}_1 \times_A \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \times_L \mathcal{L}(\mathcal{A}_2)$.

Proof. One can easily proceed by double inclusion. Alternatively, let us first ignore the coherence requirement. Then for $\mathcal{A}_1 \times_A \mathcal{A}_2$ one gets $\mathcal{L}(\mathcal{A}_1 \times_A \mathcal{A}_2) = \Pi_1^{-1}(\mathcal{L}(\mathcal{A}_1)) \cap \Pi_2^{-1}(\mathcal{L}(\mathcal{A}_2))$ as for a standard synchronous product. The proposition follows by restricting both sides to coherent words. □

5.2 Networks of automata with read arcs

To extend read arcs to networks of automata, several refinements are necessary, and in particular one needs a mechanism to specify where input values must be read. This

relies on the notion of *tag*.

5.2.1 Reading and writing tags

Consider a finite set O of output labels partitioned into $O = O_1 \uplus ... \uplus O_N$. In the sequel, each O_n will characterize the possible values displayed by component $\mathcal{A}_n$ in a network of automata with read arcs composed of $\mathcal{A}_1, ..., \mathcal{A}_N$.

Definition 5.5. *A* tag *over O is a function $\alpha : \{1, ..., N\} \to O \cup \{\star\}$ such that $\forall n, \alpha(n) \in O_n \cup \{\star\}$.*

The *support* of α is $Supp(\alpha) = \{n : \alpha(n) \neq \star\}$, and for $J \subseteq \{1, ..., N\}$, $\mathscr{T}_J$ denotes the set of tags whose support is J, $\mathscr{T}_{\subseteq J}$ denotes the set of tags whose support is included in J, and $\mathscr{T}$denotes the set of all tags. Two tags α and β are *compatible*, denoted $\alpha \sim \beta$, if and only if they coincide on $Supp(\alpha) \cap Supp(\beta)$.

Definition 5.6. *The* composition *$\alpha \wedge \beta$ is defined by $(\alpha \wedge \beta)(n) = \alpha(n)$ if $\alpha(n) \neq \star$, otherwise $\beta(n)$.*

Notice that $Supp(\alpha \wedge \beta) = Supp(\alpha) \cup Supp(\beta)$, and $\alpha \wedge \beta = \beta \wedge \alpha$ when $\alpha \sim \beta$. Composition will only be applied in this case. For $J \subseteq \{1, ..., N\}$, the *restriction* of tag α to J, denoted $\alpha_{|J}$, is defined by $\alpha_{|J}(n) = \alpha(n)$ for $n \in J$, otherwise $\alpha_{|J}(n) = \star$.

5.2.2 Automata with read arcs

We assume a partition $O = O_1 \uplus ... \uplus O_N$ given once for all.

Definition 5.7. *An* automaton with read arcs *(ARA) on O is defined as a tuple $\mathcal{A} = (S, S^I, S^F, \mathscr{T} \times \Sigma \times \mathscr{T}_W, T, W)$, where $(S, S^I, S^F, \mathscr{T} \times \Sigma \times \mathscr{T}_W, T)$ is an ordinary automaton, and the* writing set *$\emptyset \neq W \subseteq \{1, ..., N\}$ defines the indices of output sets O_n displayed by $\mathcal{A}$.*

A transition $t = (s, \alpha, \sigma, \beta, s') \in T$ moves from s to s' in $\mathcal{A}$ when action σ is performed, assuming t manages to read the values specified by the *reading tag* $\alpha \in \mathscr{T}$. As a result of the firing, t produces the *writing tag* $\beta \in \mathscr{T}_W$, which means that it changes the output values in each O_n, $n \in W$. Intuitively, transition t changes the state property displayed by each component $\mathcal{A}_n$ for $n \in W$. This is made clear by the semantics of an ARA, and by the composition operations defined below. The *reading set* R of $\mathcal{A}$ is defined as the smallest subset of $\{1, ..., N\}$ such that $T \subseteq S \times \mathscr{T}_{\subseteq R} \times \Sigma \times \mathscr{T}_W \times S$, i.e. such that reading tags of transitions all have their support in R. When needed, R can be appended at the end of the tuple defining $\mathcal{A}$.

Let $\pi = t_1 ... t_K$ be a path in $\mathcal{A}$, and $(\mathscr{T} \times \Sigma \times \mathscr{T}_W)(\pi) = (\alpha_1, \sigma_1, \beta_1)...(\alpha_K, \sigma_K, \beta_K)$ its associated word. This word is *coherent over* W if and only if

$$\forall 2 \leq k \leq K, \qquad (\alpha_k)_{|W} \sim (\beta_{k-1})_{|W} \tag{5.4}$$

In other words, readings and writings along this path must be consistent for every component O_n of the reading and writing tags, for $n \in W$. Given that $Supp(\beta_{k-1}) = W$, (5.4) reproduces condition (1) of Definition 5.1, for every $n \in W$. The readings performed by transitions of $\mathcal{A}$ outside W are not considered. The *coherent language* of $\mathcal{A}$, denoted $\mathcal{L}_c(\mathcal{A})$, is defined as the subset of words in $\mathcal{L}(\mathcal{A})$ that are coherent over its writing set W.

Notice that the above definitions of ARA and their semantics are simply the extension of the automata with inputs and outputs of Section 5.1 to the case of vector readings and writings.

Definition 5.8. *A* network of ARA *is defined as an N-uple $(\mathcal{A}_1, ..., \mathcal{A}_N)$ of ARA such that $\mathcal{A}_n = (S_n, S_n^I, S_n^F, \mathscr{T}_{\subseteq R_n} \times \Sigma_n \times \mathscr{T}_{\{n\}}, T_n, \{n\}, R_n)$, $1 \leq n \leq N$.*

Each *component* $\mathcal{A}_n$ is thus in charge of displaying values in its private set O_n of state properties, by means of the writing tags in $\mathscr{T}_{\{n\}}$ attached to transitions. Notice that $\mathcal{A}_n$ may also read values in its own O_n by the reading tags in $\mathscr{T}_{\subseteq R_n}$, which is somehow redundant with (but not equivalent to) reading its own internal state to enable the firing of a transition. The interest of this phenomenon becomes clear in the definition of the composition of ARA, since it allows cross-readings, and for working with languages, regardless of the actual automata that produced them.

Notice that the networks of ARA are in fact very similar to the asynchronous cellular automata as described in [86]. However, the networks of ARA formalism is much closer to the factored planning formalism presented in the previous chapters of this thesis.

5.2.3 Operations on languages

Definition 5.9. *Consider letter $(\alpha, \sigma, \beta) \in \mathscr{T} \times \Sigma \times \mathscr{T}_W$, and let R', Σ', W' be respectively reading, label and writing sets. The* projection $\Pi_{R',\Sigma',W'}$ *is defined on letters by $\Pi_{R',\Sigma',W'}(\alpha, \sigma, \beta) = (\alpha_{|R'}, \sigma, \beta_{|W'})$ if $\sigma \in \Sigma'$, otherwise $\Pi_{R',\Sigma',W'}(\alpha, \sigma, \beta) = \varepsilon$. It is then extended to words in $(\mathscr{T} \times \Sigma \times \mathscr{T}_W)^*$ as the induced monoid morphism, and to languages by union.*

Notice that if $\mathcal{L}$ is a coherent language over W, its projection $\Pi_{R',\Sigma',W'}(\mathcal{L})$ may lose coherence, due to the missing writing tags attached to letters $\sigma \notin \Sigma'$. Remark: for $w = (\alpha_1, \sigma_1, \beta_1)...(\alpha_K, \sigma_K, \beta_K)$, the definition of $\Pi_{R',\Sigma',W'}$ as a monoid morphism allows one to associate uniquely any letter $(\alpha_k, \sigma_k, \beta_k)$ of w to its image in $\Pi_{R',\Sigma',W'}(w)$, when $\sigma_k \in \Sigma'$.

Definition 5.10 (Weak inclusion)**.** *Let $\mathcal{L}, \mathcal{L}'$ be two languages over $\mathscr{T}_R \times \Sigma \times \mathscr{T}_W$. We write $\mathcal{L} \sqsubseteq \mathcal{L}'$ if and only if $\forall w = (\alpha_1, \sigma_1, \beta_1)...(\alpha_K, \sigma_K, \beta_K) \in \mathcal{L}$ there exists a word $w' = (\alpha'_1, \sigma_1, \beta_1)...(\alpha'_K, \sigma_K, \beta_K) \in \mathcal{L}'$ such that α'_k is a restriction of α_k, $1 \leq k \leq K$.*

In other words, in the above definition, w' is identical to w up to reading tags, that may be less specific. We denote it by $w \sqsubseteq w'$, with a light abuse of notation.

Definition 5.11. *For $i = 1, 2$, let $w_i \in (\mathscr{T}_{\subseteq R_i} \times \Sigma_i \times \mathscr{T}_{W_i})^*$ be a coherent word over W_i. The product $w_1 \times_L w_2$ is defined as the set of words $w \in (\mathscr{T}_{\subseteq R} \times \Sigma \times \mathscr{T}_W)^*$ that are coherent over W, with $R = R_1 \cup R_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$ and $W = W_1 \cup W_2$, and such that*

(I) $\Pi_{R_i,\Sigma_i,W_i}(w) \sqsubseteq w_i$, *for* $i = 1, 2$

(II) for any letter (α, σ, β) along w

- *(a) if $\sigma \in \Sigma_1 \cap \Sigma_2$, let $(\alpha_1, \sigma, \beta_1)$ and $(\alpha_2, \sigma, \beta_2)$ be the projected images of (α, σ, β) in w_1 and w_2, resp., then $\alpha = \alpha_1 \wedge \alpha_2$ and $\beta = \beta_1 \wedge \beta_2$.*

(b) if $\sigma \notin \Sigma_2$, let $(\alpha_1, \sigma, \beta_1)$ be the image of (α, σ, β) in w_1, then $\alpha = \alpha_1 \wedge (\beta_2)_{|W_2 \setminus W_1}$ and $\beta = \beta_1 \wedge (\beta_2)_{|W_2 \setminus W_1}$, where β_2 is such that $(\alpha_2, \sigma', \beta_2)$ is the image of the last letter $(\alpha', \sigma', \beta')$ before (α, σ, β) in w and such that $\sigma' \in \Sigma_2$ (if this letter does not exist, take for β_2 the only element in $\mathscr{T}_\emptyset$).

(c) if $\sigma \notin \Sigma_1$, symmetric conditions of (b).

Condition (I) ensures that w reproduces readings and writings of w_1 and w_2. But the readings in w could erase more $\star$ than necessary, i.e. require more values than those specified in w_1 and w_2. So (II) ensures that the readings of letters in w combine exactly those required by the associated letters in w_1 and w_2, not more. Specifically, (II.a) combines the reading and writing tags of the two image letters in w_1, w_2; the compatibility of these tags is guaranteed by (I). In (II.b), a private event of w_1 has to be reflected in w. Its reading and writing tags are thus augmented to push forward the output values previously positioned by $(\alpha_{2,l}, \sigma_{2,l}, \beta_{2,l})$ on $W_2 \setminus W_1$. Notice that $\alpha_{1,k} \sim \beta_{2,l}$, thanks again to (I), so $\alpha = \alpha_{1,k} \wedge (\beta_{2,l})_{|W_2 \setminus W_1} = \alpha_{1,k} \wedge \beta_{2,l}$.

Notice that conditions (I,II) above can easily be turned into a recursive construction of an element in $w_1 \times_L w_2$, that would interleave letters of these two words, provided the coherence of reading and writing tags is checked before each composition, in order to ensure the coherence of the resulting w over W.

The definition of the product of two words naturally extends to languages by union. It is also clearly associative.

Lemma 5.1. *Let $\mathcal{L}_i$ be a language over $\mathscr{T}_{\subseteq R_i} \times \Sigma_i \times \mathscr{T}_{W_i}$, $i = 1, 2$. Then the projection $\Pi_{R_i, \Sigma_i, W_i}(\mathcal{L}_1 \times_L \mathcal{L}_2)$ is a coherent language over W_i, and satisfies*

$$\Pi_{R_i, \Sigma_i, W_i}(\mathcal{L}_1 \times_L \mathcal{L}_2) \sqsubseteq \mathcal{L}_i.$$

Proof. The result is standard for the usual product of languages (ignoring reading and writing tags). To extend it here, one simply has to check it for any word w in the product of $w_1 \times_L w_2$. The coherence of $\Pi_{R_i, \Sigma_i, W_i}(w)$ over W_i derives from the coherence of w over $W \supseteq W_i$. Then, in the definition/construction of an element in $w_1 \times_L w_2$, reading tags of w_i are modified: for example $\alpha = \alpha_{1,k} \wedge \alpha_{2,l}$, which entails that $\alpha_{1,k}$ and $\alpha_{2,l}$ are restrictions of α. For writing tags, the relation $\beta = \beta_{1,k} \wedge \beta_{2,l}$ entails $\beta_{1,k} = \beta_{|W_1}$ and $\beta_{2,l} = \beta_{|W_2}$, by considering supports. Similar reasonings hold for cases (II.b) and (II.c). As a result, $\Pi_{R_i, \Sigma_i, W_i}(w) \sqsubseteq w_i$. □

Notice that the product $\mathcal{L}_1 \times_L \mathcal{L}_2$ removes more words in each $\mathcal{L}_i$ than the usual synchronous product (that it reinforces). This is due to the necessity of checking more properties in words w_1 and w_2 that are combined, namely the coherence of their cross readings and common writings over $W_1 \cap W_2$. Notice as well that $\Pi_{R_i, \Sigma_i, W_i}(w_1 \times_L w_2)$ is not w_i in general, but a set of versions of w_i where reading tags are reinforced. This is due to readings inherited from the letters of the other word w_j with which they may synchronize.

5.2.4 Product of automata with read arcs

Definition 5.12. *Consider $\mathcal{A}_i = (S_i, S_i^I, S_i^F, \mathscr{T} \times \Sigma_i \times \mathscr{T}_{W_i}, T_i, W_i)$, $i = 1, 2$. The* product *$\mathcal{A}_1 \times_A \mathcal{A}_2$ is defined by $(S, S^I, S^F, \mathscr{T} \times \Sigma \times \mathscr{T}_W, T, W)$ where $S = S_1 \times S_2, S^I = S_1^I \times S_2^I, S^F = S_1^F \times S_2^F, \Sigma = \Sigma_1 \cup \Sigma_2, W = W_1 \cup W_2$, and the transition*

set $T = T_s \cup T_{1,p} \cup T_{2,p}$ *is given by*

$$\begin{aligned} T_s &= \{\, ((s_1,s_2),\alpha_1\wedge\alpha_2,\sigma,\beta_1\wedge\beta_2,(s_1',s_2')) : \\ &\quad (s_i,\alpha_i,\sigma,\beta_i,s_i')\in T_i,\ \alpha_1\sim\alpha_2,\ \beta_1\sim\beta_2 \,\} \qquad (5.5)\\ T_{1,p} &= \{\, ((s_1,s_2),\alpha_1\wedge\beta_2,\sigma_1,\beta_1\wedge\beta_2,(s_1',s_2)) : \\ &\quad \sigma_1\notin\Sigma_2,\ (s_1,\alpha_1,\sigma_1,\beta_1,s_1')\in T_1,\ s_2\in S_2, \\ &\quad \beta_2\in\mathscr{T}_{W_2\setminus W_1},\ \alpha_1\sim\beta_2 \,\} \qquad (5.6)\\ T_{2,p} &= \{\, ((s_1,s_2),\alpha_2\wedge\beta_1,\sigma_2,\beta_1\wedge\beta_2,(s_1,s_2')) : \\ &\quad \sigma_2\notin\Sigma_1,\ (s_2,\alpha_2,\sigma_2,\beta_2,s_2')\in T_2,\ s_1\in S_1, \\ &\quad \beta_1\in\mathscr{T}_{W_1\setminus W_2},\ \alpha_2\sim\beta_1 \,\} \qquad (5.7) \end{aligned}$$

Intuitively, (5.5) merges/synchronizes transitions carrying shared labels, provided they agree on readings and writings. While, (5.6) extends private transitions of $\mathcal{A}_1$ to all possible values of s_2, and to all possible values of reading and writing tags over $W_2 \setminus W_1$. And, (5.7) does the symmetric operation for private transitions of $\mathcal{A}_2$.

Not all transitions defined above are accessible and coaccessible in $\mathcal{A}_1 \times \mathcal{A}_2$. Therefore some trimming should be performed to remove useless transitions, taking into account that (co-) accessibility here must incorporate the coherence over W.

Theorem 5.1. $\mathcal{L}_c(\mathcal{A}_1 \times_A \mathcal{A}_2) = \mathcal{L}_c(\mathcal{A}_1) \times_L \mathcal{L}_c(\mathcal{A}_2)$

Proof. We proceed by double inclusion.

For $\supseteq$, consider $w \in \mathcal{L}_c(\mathcal{A}_i) \times_L \mathcal{L}_c(\mathcal{A}_j)$. Note $w = (\alpha^1,\sigma^1,\beta^1)\dots(\alpha^k,\sigma^k,\beta^k)$. By definition of $\times_L$ there exists $w_i \in \mathcal{L}_c(\mathcal{A}_i)$ and $w_j \in \mathcal{L}_c(\mathcal{A}_j)$ such that $w \in w_i \times_L w_j$. Note $w_i = (\alpha_i^1,\sigma_i^1,\beta_i^1)\dots(\alpha_i^{k_i},\sigma_i^{k_i},\beta_i^{k_i})$ and $w_j = (\alpha_j^1,\sigma_j^1,\beta_j^1)\dots(\alpha_j^{k_j},\sigma_j^{k_j},\beta_j^{k_j})$. These two words are such that there exists $f_i : \{1..k_i\} \to \{1..k\}$ with $f_i(\ell) < f_i(m)$ if and only if $\ell < m$ and f_j similarly defined for w_j, with the properties that $\forall \ell \in \{1..k\}$:

1. if $\sigma^\ell \in \Sigma_i \cap \Sigma_j$, then $\exists \ell_i, \ell_j$, such that $f_i(\ell_i) = f_j(\ell_j) = \ell$, $\alpha^\ell = \alpha_i^{\ell_i} \wedge \alpha_j^{\ell_j}$, $\beta^\ell = \beta_i^{\ell_i} \wedge \beta_j^{\ell_j}$, and $\sigma^\ell = \sigma_i^{\ell_i} = \sigma_j^{\ell_j}$,
2. if $\sigma^\ell \in \Sigma_i \setminus \Sigma_j$, then $\exists \ell_i$, such that $f_i(\ell_i) = \ell$, $\alpha^\ell = \alpha_i^{\ell_i} \wedge \beta_j^{f_j^{-1}(\ell)}$, $\beta^\ell = \beta_i^{\ell_i} \wedge \beta_j^{f_j^{-1}(\ell)}$, and $\sigma^\ell = \sigma_i^{\ell_i}$, with $f_j^{-1}(\ell)$ the greatest ℓ_j such that $f_j(\ell_j) < \ell$,
3. the symmetric with $\sigma^\ell \in \Sigma_j \setminus \Sigma_i$.

Moreover, by definition of the language of an ARA there exists π_i a path in $\mathcal{A}_i$ and π_j a path in $\mathcal{A}_j$ such that $(\mathscr{T} \times \Sigma_i \times \mathscr{T}_{W_i})(\pi_i) = w_i$ and $(\mathscr{T} \times \Sigma_j \times \mathscr{T}_{W_j})(\pi_j) = w_j$, in other words $\pi_i = (s_i^1,\alpha_i^1,\sigma_i^1,\beta_i^1,s_i^{1\prime})\dots(s_i^{k_i},\alpha_i^{k_i},\sigma_i^{k_i},\beta_i^{k_i},s_i^{k_i\prime})$ and $\pi_j = (s_j^1,\alpha_j^1,\sigma_j^1,\beta_j^1,s_j^{1\prime})\dots(s_j^{k_j},\alpha_j^{k_j},\sigma_j^{k_j},\beta_j^{k_j},s_j^{k_j\prime})$. Then, by definition of $\times_A$ any $\pi = (s^1,\hat\alpha^1,\hat\sigma^1,\hat\beta^1,s^{1\prime})\dots(s^k,\hat\alpha^k,\hat\sigma^k,\hat\beta^k,s^{k\prime}) \in \pi_i \times_A \pi_j$ is such that there exists $f_i' : \{1..k_i\} \to \{1..k\}$ with $f_i'(\ell) < f_i'(m)$ if and only if $\ell < m$ and f_j' similarly defined for π_j, with the properties that $\forall \ell \in \{1..k\}$:

1. if $\hat\sigma^\ell \in \Sigma_i \cap \Sigma_j$, then $\exists \ell_i, \ell_j$, such that $f_i'(\ell_i) = f_j'(\ell_j) = \ell$, $s^\ell = (s_i^{\ell_i}, s_j^{\ell_j})$, $s^{\ell\prime} = (s_i^{\ell_i\prime}, s_j^{\ell_j\prime})$, $\hat\sigma^\ell = \sigma_i^{\ell_i} = \sigma_j^{\ell_j}$, $\hat\alpha^\ell = \alpha_i^{\ell_i} \wedge \alpha_j^{\ell_j}$, and $\hat\beta^\ell = \beta_i^{\ell_i} \wedge \beta_j^{\ell_j}$,
2. if $\hat\sigma^\ell \in \Sigma_i \setminus \Sigma_j$, then $\exists \ell_i$, $f_i'(\ell_i) = \ell$, $s^\ell = (s_i^{\ell_i}, s_j^{f_j'^{-1}(\ell)\prime})$, $s^{\ell\prime} = (s_i^{\ell_i\prime}, s_j^{f_j'^{-1}(\ell)\prime})$, $\hat\sigma^\ell = \sigma_i^{\ell_i}$, $\hat\alpha^\ell = \alpha_i^{\ell_i} \wedge \beta_{j,\ell}$, and $\hat\beta^\ell = \beta_i^{\ell_i} \wedge \beta_{j,\ell}$, where $\alpha_i^{\ell_i} \sim \beta_{j,\ell} \in \mathscr{T}_{W_j}$,

3. the symmetric with $\hat{\sigma}^\ell \in \Sigma_j \setminus \Sigma_i$.

Identifying f_i with f'_i, f_j with f'_j, each $\beta_{j,\ell}$ with $\beta_j^{f_j^{-1}(\ell)}$, and each $\beta_{i,\ell}$ with $\beta_i^{f_i^{-1}(\ell)}$ gives a $\pi \in \pi_i \times_A \pi_j$ (here we slightly abuse the notations: this product is in fact the product of two ARA, each one containing only one path) such that $(\mathscr{T}\times \Sigma \times \mathscr{T}_W)(\pi) = w$. Moreover w is coherent by definition of $\times_L$. Thus, $w \in \mathcal{L}_c(\mathcal{A}_i \times_A \mathcal{A}_j)$. We proved the first inclusion: $\mathcal{L}_c(\mathcal{A}_i \times_A \mathcal{A}_j) \supseteq \mathcal{L}_c(\mathcal{A}_i) \times_L \mathcal{L}_c(\mathcal{A}_j)$.

For $\subseteq$, consider $w \in \mathcal{L}_c(\mathcal{A}_i \times_A \mathcal{A}_j)$. By definition of $\mathcal{L}_c$ there exists π a path in $\mathcal{A}_i \times \mathcal{A}_j$ such that $(\mathscr{T}\times \Sigma \times \mathscr{T}_W)(\pi) = w$ and w is coherent. By definition of product there exists π_i a path in $\mathcal{A}_i$ and π_j a path in $\mathcal{A}_j$, such that $\pi \in \pi_i \times_A \pi_j$. Moreover $(\mathscr{T}\times \Sigma_i \times \mathscr{T}_{W_i})(\pi_i) = w_i$ and $(\mathscr{T}\times \Sigma_j \times \mathscr{T}_{W_j})(\pi_j) = w_j$ are coherent (else w would not be coherent). And thus $w_i \in \mathcal{L}_c(\mathcal{A}_i)$ and $w_j \in \mathcal{L}_c(\mathcal{A}_j)$ by construction of π_i and π_j. Moreover $w \in w_i \times_L w_j$ (using the same kind of construction as above), hence $w \in \mathcal{L}_c(\mathcal{A}_i) \times_L \mathcal{L}_c(\mathcal{A}_j)$. This proves the second inclusion: $\mathcal{L}_c(\mathcal{A}_i \times_A \mathcal{A}_j) \subseteq \mathcal{L}_c(\mathcal{A}_i) \times_L \mathcal{L}_c(\mathcal{A}_j)$. □

Theorem 5.1 can be plugged into Lemma 5.1. This gives $\Pi_{R_1,\Sigma_1,W_1}(\mathcal{L}_c(\mathcal{A}_1 \times_A \mathcal{A}_2)) \sqsubseteq \mathcal{L}_c(\mathcal{A}_1)$ (and symmetrically for $\mathcal{A}_2$). So the synchronization of $\mathcal{A}_1$ with $\mathcal{A}_2$ removes words w_1 of $\mathcal{A}_1$ that have no compatible companion w_2 in $\mathcal{A}_2$, both for the firing of common labels in $\Sigma_1 \cap \Sigma_2$ and for the compatibility of readings and writings. For words w_1 of $\mathcal{A}_1$ that are preserved, their writing tags are unchanged, but their reading tags may inherit extra conditions from their companion w_2 in $\mathcal{A}_2$.

5.3 Planning in networks of ARA

As mentioned in Chapter 2, a factored planning problem can be modeled as a network of automata. Similarly it can be defined as a network of automata with read arcs. Given $(\mathcal{A}_1, ..., \mathcal{A}_N)$, a plan is simply a (coherent) word w in $\mathcal{L}_c(\mathcal{A}_1 \times_A \cdots \times_A \mathcal{A}_N)$, i.e. a sequence of events reaching a marked or final state from the/an initial state, ensuring both that synchronized actions are performed correctly, and that readings and writings in each component $\mathcal{A}_n$ are also performed correctly. Factored planning would consist in deriving w under a factored form, i.e. as a tuple $(w_1, ..., w_N)$ of words, with $w_n = \Pi_{R_n,\Sigma_n,W_n}(w) \in \Pi_{R_n,\Sigma_n,W_n}(\mathcal{L}_c(\mathcal{A}_1 \times_A \cdots \times_A \mathcal{A}_N)) \sqsubseteq \mathcal{L}_c(\mathcal{A}_n)$, *without computing* w, since working with $\mathcal{A}_1 \times_A \cdots \times_A \mathcal{A}_N$ might be intractable. Surprisingly, this can be achieved if component interactions are sparse enough, as it was already proved in Chapter 2 for networks of weighted automata. The idea is that the projected languages $\Pi_{R_n,\Sigma_n,W_n}(\mathcal{L}_c(\mathcal{A}_1 \times_A \cdots \times_A \mathcal{A}_N))$ can be derived, without computing first $\mathcal{L}_c(\mathcal{A}_1 \times_A \cdots \times_A \mathcal{A}_N)$, by means of local and coordinated computations. The same procedure can be used as well to select a factored plan $(w_1, ..., w_N)$ in these projections. The central tool to achieve this is a relation between the product and the projection of languages, that we immediately translate into a relation between the product and the projection of ARA, in order to handle finite objects.

5.3.1 ARA representing planning problems

We first give an overview of some useful property of the ARA representing planning problems. This property is central because the ARA having it also verify Theorem 5.2 below, which allows to use ARA instead of their languages when taking a projection.

Notice that, in planning problems, it may be the case that some properties of an initial state have to be displayed. Due to the behavior of ARA this is not directly

possible as the properties of a state are displayed by the transition preceding it in a path (and an initial state is not preceded by a transition in general). This issue could be solved by adding some initial displayed properties before initial states. However it does not fit well with the notion of coherent words: it would require to check coherency inside the word but also with the initial state of the path which gave the word. We thus suggest that the ARA $\mathcal{A} = (S, S^I, S^F, \mathscr{T} \times \Sigma \times \mathscr{T}_W, T, W)$ representing planning problems are always such that: Σ contains the special label $start$, S^I contains only one element s^I, there is no transition $t \in T$ such that $t^+ = s^I$, and each transition $t \in T$ such that $t^- = s^I$ is labeled by $start$. The idea is that s^I will be a dummy initial state. The states corresponding to the initial states of a planning problem will be the ones such that a transition labeled by $start$ reaches them. And these $start$ labeled transitions will be used to display the initial states properties.

Definition 5.13. *An ARA $\mathcal{A} = (S, S^I, S^F, \mathscr{T} \times \Sigma \times \mathscr{T}_W, T, W)$ is* state labeled *if and only if for any pair of transitions $t = (s, \alpha, \sigma, \beta, s'')$ and $t' = (s', \alpha', \sigma', \beta', s''')$ in T, $s'' = s'''$ entails $\beta = \beta'$.*

This means that all writings from transitions reaching given state are the same. In other words, writings characterize a property of the reached state, which matches our interpretation of state reading arcs. Notice that different states can still "carry" the same tag. The reason for defining this property of ARA is that the ARA representing planning problems are clearly state labeled (because in this case the states of the ARA match the states of the planning problem, and the actions will write exactly the state they reach). The property of being state labeled is preserved by several operations on ARA.

Proposition 5.2. *If $\mathcal{A}_i$ and $\mathcal{A}_j$ are state labeled ARA, then $\mathcal{A}_i \times_A \mathcal{A}_j$ is also state labeled.*

Proof. Consider a state (s_i, s_j) in $\mathcal{A}_i \times_A \mathcal{A}_j$ and two transitions $t = (s, \alpha, \sigma, \beta, s'')$ and $t' = (s', \alpha', \sigma', \beta', s''')$ such that $s'' = (s_i, s_j) = s'''$ and $\beta \neq \beta'$. Four cases are possible:

1. t and t' are both shared transitions, in this case $\beta = \beta_i \wedge \beta_j$ and $\beta' = \beta'_i \wedge \beta'_j$, with β_i and β'_i (resp. β_j and β'_j) coming from two transitions t_i and t'_i (resp. t_j and t'_j) leading to the same state s_i in $\mathcal{A}_i$ (resp. s_j in $\mathcal{A}_j$). As $\mathcal{A}_i$ and $\mathcal{A}_j$ are state labeled we have $\beta_i = \beta'_i$ and $\beta_j = \beta'_j$, thus $\beta = \beta'$.

2. t is a shared transition and t' is a private transition. Without loss of generality, assume t' is a transition from $\mathcal{A}_i$. One has that $\beta = \beta_i \wedge \beta_j$ and $\beta' = \beta'_i \wedge \beta'_j$. As $\mathcal{A}_i$ is state labeled one has $\beta_i = \beta'_i$. One has to prove $\beta_j = \beta'_j$. Notice that $s' = (s''_i, s_j)$. Two cases are possible: either $s_j \in S^I_j$ or $s_j \notin S^I_j$. In the first case the fact that $\mathcal{A}_j$ is state labeled ensure directly $\beta_j = \beta'_j$. If $s_j \notin S^I_j$, the fact that $\mathcal{A}_i \times_A \mathcal{A}_j$ is trim ensure the existence of a path π in it such that $\pi = t_1 \dots t_{n-1} t' t_{n+1} \dots t_k$, $\pi^- \in S^I$, and $\pi^+ \in S^F$. We note $t_\ell = ((s^\ell_i, s^\ell_j), \alpha^\ell, \sigma^\ell, \beta^\ell, (s^{\ell\prime}_i, s^{\ell\prime}_j))$. Let m be the largest integer smaller than n such that $s^m_j \neq s_j$. As $\mathcal{A}_j$ is state labeled one has $\beta'_j = \beta^p_j$ for all $p \in \{m..n\}$. For the same reason $\beta_j = \beta^m$. Thus, one can conclude that $\beta_j = \beta'_j$, and finally that $\beta = \beta'$.

3. t and t' are both private transitions from the same automaton. Assume, without loss of generality, that these transitions are from $\mathcal{A}_i$. One has that $\beta = \beta_i \wedge \beta_j$ and

$\beta' = \beta'_i \wedge \beta'_j$. As $\mathcal{A}_i$ is state labeled, and as the transitions in $\mathcal{A}_i$ corresponding to t and t' lead to the same state s_i, one has $\beta_i = \beta'_i$. The same justification as in (2) ensures $\beta_j = \beta'_j$. Thus, $\beta = \beta'$.

4. t is a private transition from an automaton and t' is a private transition from the other automaton. Applying twice the justification from (2) proves that $\beta = \beta'$.

In each possible case one has that $\beta = \beta'$. This proves the proposition. □

Standard determinization procedure (see for example [16]) can be performed on ARA. Indeed determinization preserves the language over alphabet $\mathscr{T}_{\subseteq R'} \times \Sigma' \times \mathscr{T}_{W'}$, it thus preserves the coherent language. Denote by $DET(\mathcal{A})$ the ARA obtained from the ARA $\mathcal{A}$ by applying a standard determinization procedure to it.

Proposition 5.3. *If $\mathcal{A}$ is a state labeled ARA, then $DET(\mathcal{A})$ is state labeled.*

Proof. Let $X \subseteq S$ be a state in $DET(\mathcal{A})$. Considering the standard determinization algorithm, X belongs to $DET(\mathcal{A})$ means that there exists a state $Y \subseteq S$ in $DET(\mathcal{A})$ and a label (α, σ, β) such that $\forall s' \in X, \exists s \in Y, (s, \alpha, \sigma, \beta, s') \in T$. In other words, $\exists \beta$, $\forall s' \in X$, a transition writing β reaches s'. As $\mathcal{A}$ is state labeled, any transition from $\mathcal{A}$ reaching a state in X writes β. Hence, by construction of the determinized, for all $(Y, \alpha, \sigma, \beta, X)$ and $(Z, \alpha', \sigma', \beta', X)$ transitions in $DET(\mathcal{A})$, one has $\beta = \beta'$. One can then conclude that $DET(\mathcal{A})$ is state labeled. □

5.3.2 Projection of an ARA

Definition 5.14. *The* projection *of an automaton with read arcs $\mathcal{A} = (S, S^I, S^F, \mathscr{T} \times \Sigma \times \mathscr{T}_W, T, W)$ on R', Σ', W', respectively reading, label and writing sets, is the ARA $\Pi_{R',\Sigma',W'}(\mathcal{A}) = (S, S^I, {S'}^F, \mathscr{T}_{\subseteq R'} \times \Sigma' \times \mathscr{T}_{W'}, T', W')$. Transitions are defined from a (possibly empty) silent path π, i.e. labeled by $\Sigma \setminus \Sigma'$, followed by a visible transition, i.e. labeled by Σ':*

$$\begin{aligned} T' \quad = \{ \quad & (s, \alpha_{|R'}, \sigma, \beta_{|W'}, s') \ : \ \sigma \in \Sigma', \\ & \exists (s'', \alpha, \sigma, \beta, s') \in T, \\ & \exists \pi \textit{ a path in } \mathcal{A}, \Sigma(\pi) \in (\Sigma \setminus \Sigma')^*, \\ & \pi^- = s, \ \pi^+ = s'' \}. \end{aligned}$$

Finally, the set ${S'}^F$ is also constructed from paths of transitions from $\Sigma \setminus \Sigma'$: ${S'}^F = S^F \cup \{s \in S \ : \ \exists \pi$ a path in $\mathcal{A}, \Sigma(\pi) \in (\Sigma \setminus \Sigma')^*, \ \pi^- = s, \ \pi^+ \in S^F\}$.

The fact that only $\alpha_{|R'}$ is considered in the definition of the transitions requires a comment. In the sequel, R', Σ', W' will be the reading, label and writing sets of some automaton, and thus will select the transitions of this automaton. So any discarded transition t' of path π will be a private transition of another automaton, which can not modify the values necessary to $\alpha_{|W'}$. So $(s, \alpha_{|R'}, \sigma, \beta_{|W'}, s')$ will remain firable in the projection, since coherence is only tested over W'.

As for the product, this definition may produce states and transitions that are not accessible and coaccessible, so a trimming may be necessary. In fact, the construction above corresponds to an ε-reduction to the left: useless transitions are bypassed. It generally results in non-deterministic automata. Thus, one may perform a determinization procedure (and a minimization procedure as well) after taking the projection of an ARA. The determinization may however incur an exponential blowup in the number of states, in the worst case.

In the sequel, we denote by $\Pi_{\mathcal{A}_i}$ the projection Π_{R_i,Σ_i,W_i}, for some ARA $\mathcal{A}_i = (S_i, S_i^I, S_i^F, \mathscr{T}\times\Sigma_i\times\mathscr{T}_{W_i}, T_i, W_i)$, and its writing set R_i. Notice that taking the projection also preserves the property of being state labeled.

Proposition 5.4. *If $\mathcal{A}$ is a state labeled ARA, then $\Pi_{\mathcal{A}'}(\mathcal{A})$ is state labeled for any $\mathcal{A}'$.*

Proof. It comes directly from the definition of projection: the only transitions who may be added are of the form $t = (s, \alpha, \sigma, \beta, s')$ for some existing $t' = (s'', \alpha, \sigma, \beta, s')$ in the original automaton. So every transition going to some vertex are either transitions from the original automaton or copies of these transitions. □

The importance of being state labeled comes from the fact that the coherence of the words produced by $\mathcal{A}_i \times_A \mathcal{A}_j$ is only checked when taking the language. Taking the projection $\Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A \mathcal{A}_j)$ first can thus erase some of the coherence conditions, unless if they are stucturally reflected in the automata.

Theorem 5.2. *If $\mathcal{A}_i$ and $\mathcal{A}_j$ are state labeled then*

$$\mathcal{L}_c(\Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A \mathcal{A}_j)) = \Pi_{\mathcal{A}_i}(\mathcal{L}_c(\mathcal{A}_i \times_A \mathcal{A}_j))$$

Proof. We prove this theorem by double inclusion.

For $\subseteq$, consider w' a word from $\mathcal{L}_c(\Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A \mathcal{A}_j))$. By definition of the language of an automaton, there exists π' a path in $\Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A \mathcal{A}_j)$ such that $(\mathscr{T}\times\Sigma_i\times\mathscr{T}_{W_i})(\pi') = w'$.

One can show that there exists a path π in $\mathcal{A}_i \times_A \mathcal{A}_j$ such that $(\mathscr{T}\times(\Sigma_i\cup\Sigma_j)\times\mathscr{T}_{W_i\cup W_j})(\pi) = w$ and $\Pi_{\mathcal{A}_i}(w) = w'$. The idea is to construct π from $\pi' = t_1 \dots t_k$: by definition of $\Pi_{\mathcal{A}_i}$, for each $t_i = (t_i^-, \alpha(t_i), \sigma(t_i), \beta(t_i), t_i^+)$, either t_i is a transition from $\mathcal{A}_i \times_A \mathcal{A}_j$ or there exists $t_i^1, \dots, t_i^{\ell_i}$, transitions from $\mathcal{A}_i \times_A \mathcal{A}_j$ such that $t_i^{\ell_i} = (s, \alpha, \sigma(t_i), \beta, t_i^+)$ with $\alpha_{|R_i} = \alpha(t_i), \beta_{|W_i} = \beta(t_i)$, $t_i^{1-} = t_i^-$, and $\sigma(t_i^j) \notin \Sigma_i$ for any $i \in \{1..\ell_i - 1\}$. Consider $\tilde{\pi} = t_1^1 \dots t_1^{\ell_1} t_2^1 \dots t_k^1 \dots t_k^{\ell_k}$. As it is a path in $\mathcal{A}_i \times_A \mathcal{A}_j$, one has $(\mathscr{T}\times(\Sigma_i\cup\Sigma_j)\times\mathscr{T}_{W_i\cup W_j})(\tilde{\pi}) = \tilde{w}$, with $\tilde{w} = (\alpha(t_1^1), \sigma(t_1^1), \beta(t_1^1)) \dots (\alpha(t_k^{\ell_k}), \sigma(t_k^{\ell_k}), \beta(t_k^{\ell_k}))$. By construction, one has $\forall j, \forall m < \ell_j, \sigma(t_j^m) \notin \Sigma_i$, and $\forall j, (\alpha(t_j^{\ell_j}), \sigma(t_j^{\ell_j}), \beta(t_j^{\ell_j}))$ is the j^{th} letter in w'. Thus, by definition of projection, $\Pi_{\mathcal{A}_i}(\tilde{w}) = w'$. One can take $\pi = \tilde{\pi}$ and $w = \tilde{w}$.

Moreover, there exists such π which is such that w is coherent. This can be shown by proving that any path $\tilde{\pi}$ in $\mathcal{A}_i \times_A \mathcal{A}_j$ is such that $\tilde{w} = (\mathscr{T}\times(\Sigma_i\cup\Sigma_j)\times\mathscr{T}_{W_i\cup W_j})(\tilde{\pi})$ is coherent. Suppose there exists $\tilde{\pi} = t_1 \dots t_k$ such that $\tilde{w}$ is not coherent. It means that $\exists i$ such that $t_i = (s_i, \alpha_i, \sigma_i, \beta_i, s_i')$, $t_{i+1} = (s_{i+1}, \alpha_{i+1}, \sigma_{i+1}, \beta_{i+1}, s_{i+1}')$, $s_i' = s_i + 1$, and $\alpha_{i+1} \nsim \beta_i$. As $\mathcal{A}_i \times_A \mathcal{A}_j$ is trim it means that there exists $t_i' = ((s_i'', \alpha_i', \sigma_i', \beta_i', s_i'''))$ with $s_i''' = s_{i+1}$ and $\alpha_{i+1} \sim \beta_i'$. We have that $s_i''' = s_i'$ and $\beta_i \neq \beta_i'$. This is not possible as $\mathcal{A}_i$ and $\mathcal{A}_j$, and thus $\mathcal{A}_i \times_A \mathcal{A}_j$ by proposition 5.2, are state labeled. Thus, $\tilde{w}$ has to be coherent.

Finally, as π is such that w is coherent, $w \in \mathcal{L}_c(\mathcal{A}_i \times_A \mathcal{A}_j)$. Moreover, π was constructed such that $\Pi_{\mathcal{A}_i}(w) = w'$, thus $w' \in \Pi_{\mathcal{A}_i}(\mathcal{L}_c(\mathcal{A}_i \times_A \mathcal{A}_j))$. This proves the first inclusion: $\mathcal{L}_c(\Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A \mathcal{A}_j)) \subseteq \Pi_{\mathcal{A}_i}(\mathcal{L}_c(\mathcal{A}_i \times_A \mathcal{A}_j))$.

For $\supseteq$, consider w' a word from $\Pi_{\mathcal{A}_i}(\mathcal{L}_c(\mathcal{A}_i \times_A \mathcal{A}_j))$. By definition of projection there exists $w \in \mathcal{L}_c(\mathcal{A}_i \times_A \mathcal{A}_j)$ such that $\Pi_{\mathcal{A}_i}(w) = w'$. By definition of coherent language of an automaton, there exists a path $\pi = t_1 \dots t_k$ in $\mathcal{A}_i \times_A \mathcal{A}_j$ with $t_i = (s_i, \alpha_i, \sigma_i, \beta_i, s_i')$ such that $(\alpha_1, \sigma_1, \beta_1) \dots (\alpha_k, \sigma_k, \beta_k) = w$. By definition of projection of automata there exists π' a path in $\Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A \mathcal{A}_j)$ such that $(\mathscr{T}\times\Sigma_i\times\mathscr{T}_{W_i})(\pi') = \Pi_{\mathcal{A}_i}(w) = w'$. Thus, $w' \in \mathcal{L}(\Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A \mathcal{A}_j))$. As $w' \in \Pi_{\mathcal{A}_i}(\mathcal{L}_c(\mathcal{A}_i \times_A \mathcal{A}_j))$, by Lemma 5.1 one has that w' is coherent. Finally $w' \in$

$\mathcal{L}(\Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A \mathcal{A}_j))$ and w' coherent means, by definition: $w' \in \mathcal{L}_c(\Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A \mathcal{A}_j))$. This proves the second inclusion: $\mathcal{L}_c(\Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A \mathcal{A}_j)) \supseteq \Pi_{\mathcal{A}_i}(\mathcal{L}_c(\mathcal{A}_i \times_A \mathcal{A}_j))$.

As $\mathcal{L}_c(\Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A \mathcal{A}_j)) \subseteq \Pi_{\mathcal{A}_i}(\mathcal{L}_c(\mathcal{A}_i \times_A \mathcal{A}_j))$ and $\mathcal{L}_c(\Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A \mathcal{A}_j)) \supseteq \Pi_{\mathcal{A}_i}(\mathcal{L}_c(\mathcal{A}_i \times_A \mathcal{A}_j))$, the theorem is proved. □

As explained below, when solving planning problems, it is never requested to take projections that are not of the form $\Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A \mathcal{A}_j)$. In consequence, Theorems 5.1 and 5.2 allow one to work directly on automata instead of languages, which is required in practice as automata are finite objects while languages may be infinite.

5.3.3 Central relation between product and projection

This part presents the key result that motivated the above construction. It enables the computation of a factored plan by local computations as soon as the component interactions are sparse enough. This is done in the same way as in Chapter 2: for each automaton $\mathcal{A}_i$ of a network one computes the automaton $\Pi_{\mathcal{A}_i}(\mathcal{A}_1 \times_A \cdots \times_A \mathcal{A}_N)$ which represents the set of words from $\mathcal{A}_i$ that are compatible with all the other automata, without computing the full system $\mathcal{A}_1 \times_A \cdots \times_A \mathcal{A}_N$.

However, notice that the axiom 2.3 is not verified by the product and the projection of coherent languages of ARA:

$$\exists \mathcal{L}_1, \mathcal{L}_2, \exists R_3 \supseteq R_1 \cap R_2, \Sigma_3 \supseteq \Sigma_1 \cap \Sigma_2, W_3 \supseteq W_1 \cap W_2,$$
$$\Pi_{R_3,\Sigma_3,W_3}(\mathcal{L}_1 \times_L \mathcal{L}_2) \neq \Pi_{R_3,\Sigma_3,W_3}(\mathcal{L}_1) \times_L \Pi_{R_3,\Sigma_3,W_3}(\mathcal{L}_2).$$

As an example, take for $\mathcal{L}_1$ the language of $\mathcal{A}_1$ and for $\mathcal{L}_2$ the language of $\mathcal{A}_2$ in the example below (Figure 5.3). Then consider $\Sigma_3 = \Sigma_2$, $R_3 = R_2$ and $W_3 = W_1$. In $\Pi_{R_3,\Sigma_3,W_3}(\mathcal{L}_1 \times_L \mathcal{L}_2)$ the sequence of actions $\gamma\beta''$ is possible. This is not the case in $\Pi_{R_3,\Sigma_3,W_3}(\mathcal{L}_1) \times_L \Pi_{R_3,\Sigma_3,W_3}(\mathcal{L}_2)$ because after the projections, the readings in $\mathcal{A}_1$ required by the action γ can no longer be achieved.

The fact that axiom 2.3 is not verified forbid the direct use of the results of Chapter 2. We thus have to do a lower level proof that some kind of message passing algorithms can be used in this context. This proof uses the same kind of techniques as in [28]: we first define a notion of separation between the ARA of a network, and then use it to derive a method for computing $\Pi_{\mathcal{A}_i}(\mathcal{A}_1 \times_A \cdots \times_A \mathcal{A}_N)$ with local computations only. In this section we present this method in the restricted case of networks of three ARA only. Section 5.4 then generalizes this method to larger networks.

Definition 5.15. *Let $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ have disjoint writing sets W_i, $\mathcal{A}_2$* separates *$\mathcal{A}_1$ and $\mathcal{A}_3$ if and only if:*

1. $\Sigma_3 \cap \Sigma_1 \subseteq \Sigma_2$ *(separation of actions),*
2. $R_3 \cap W_1 \subseteq R_2$, *and* $\forall\, t_3 = (s, \alpha, \sigma, \beta, s') \in T_3$, $\alpha_{|W_1} \neq \star \;\Rightarrow\; \sigma \in \Sigma_2$ *(propagation of readings),*
3. *and symmetrically, by inverting indexes 1 and 3.*

Condition (2) expresses that any reading performed by $\mathcal{A}_3$ inside $\mathcal{A}_1$ corresponds to an action σ that is shared with $\mathcal{A}_2$. So in the product $\mathcal{A}_3 \times_A \mathcal{A}_2$, component $\mathcal{A}_2$ will inherit the readings in $\mathcal{A}_1$ that are necessary to $\mathcal{A}_3$. This induces the following theorem.

Theorem 5.3. *Let $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ be three state labeled ARA. If $\mathcal{A}_2$ separates $\mathcal{A}_1$ and $\mathcal{A}_3$, then* $\Pi_{\mathcal{A}_1}(\mathcal{L}_c(\mathcal{A}_1 \times_A \mathcal{A}_2 \times_A \mathcal{A}_3)) = \mathcal{L}_c(\Pi_{\mathcal{A}_1}(\mathcal{A}_1 \times_A \Pi_{\mathcal{A}_2}(\mathcal{A}_2 \times_A \mathcal{A}_3)))$.

Proof. By Theorem 5.2 one directly gets:

$$\Pi_{\mathcal{A}_1}(\mathcal{L}_c(\mathcal{A}_1 \times_A \mathcal{A}_2 \times_A \mathcal{A}_3)) = \mathcal{L}_c(\Pi_{\mathcal{A}_1}(\mathcal{A}_1 \times_A \mathcal{A}_2 \times_A \mathcal{A}_3)).$$

Then, the separation property ensures:

$$\mathcal{L}_c(\Pi_{\mathcal{A}_1}(\mathcal{A}_1 \times_A \mathcal{A}_2 \times_A \mathcal{A}_3)) = \mathcal{L}_c(\Pi_{\mathcal{A}_1}(\mathcal{A}_1 \times_A \Pi_{\mathcal{A}_2}(\mathcal{A}_2 \times_A \mathcal{A}_3)))$$

because every interaction between $\mathcal{A}_1$ and $\mathcal{A}_3$ is fully captured by $\mathcal{A}_2$. In particular any shared action between $\mathcal{A}_1$ and $\mathcal{A}_3$ exists also in $\mathcal{A}_2$ (by condition (1) of the definition of separation), the readings $\mathcal{A}_3$ has to perform in $\mathcal{A}_1$ are only from actions shared with $\mathcal{A}_2$ (by condition (2) of the definition of separation) so they are propagated by these shared actions, and finally the coherence of the readings of $\mathcal{A}_1$ into $\mathcal{A}_3$ is ensured by the shared actions of $\mathcal{A}_1$ and $\mathcal{A}_2$ in the same manner (by condition (3) of the definition of separation). □

The practical consequence of this result is that $\Pi_{\mathcal{A}_1}(\mathcal{A})$, which language describes the local plans of $\mathcal{A}_1$, can be derived without taking first the full product $\mathcal{A} = \mathcal{A}_1 \times_A \mathcal{A}_2 \times_A \mathcal{A}_3$, which language describes all global plans. Indeed, it suffices to combine $\mathcal{A}_1$ with the smaller *message* $\Pi_{\mathcal{A}_2}(\mathcal{A}_2 \times_A \mathcal{A}_3)$ (this notion of message is similar to the one presented in the previous chapters and will be made more explicit by the following example) from $\mathcal{A}_2$, and project the result on $\mathcal{A}_1$. Similar ideas allow one to derive as well all the local plan descriptions $\Pi_{\mathcal{A}_i}(\mathcal{A})$. In next section a detailed application example of Theorem 5.3 is presented.

5.3.4 Example

We now illustrate distributed planning on an example with three automata (Figure 5.3). Readings are depicted by "$?x$" and writings by "$!x$", where x is some value. For clarity, readings "$?\star$" are omitted in figures, and initial writings "$!x$" appear on the arrows pointing to initial states. Each of these initial writings "$!x$" is only a graphical representation for an initial state, followed by a transition labeled by $start$ and writing the same "$!x$" (as described in Section 5.3.1).The underlying partition of O is $O_1 = \{1, 2\}$, $O_2 = \{3, 4, 5, 6\}$ and $O_3 = \{7, 8\}$, and one has $\Sigma_1 = \{\alpha, \alpha'\}, \Sigma_2 = \{\beta, \beta', \beta'', \gamma\}, \Sigma_3 = \{\gamma, \delta\}$.

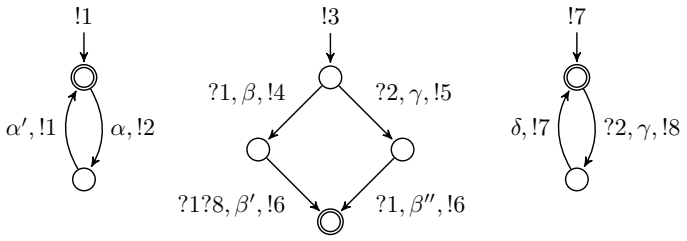


Figure 5.3: Network of ARA: $\mathcal{A}_1$ (left), $\mathcal{A}_2$ (center), and $\mathcal{A}_3$ (right).

Automata $\mathcal{A}_2$ and $\mathcal{A}_3$ both read in $\mathcal{A}_1$; there is no other reading. One can check that $\mathcal{A}_2$ separates $\mathcal{A}_1$ and $\mathcal{A}_3$. Specifically, $\mathcal{A}_1$ and $\mathcal{A}_3$ share no action and $\mathcal{A}_1$ does not read in $\mathcal{A}_3$, and conversely the only readings of $\mathcal{A}_3$ in $\mathcal{A}_1$ are done by action γ, which is shared with $\mathcal{A}_2$. Since $\mathcal{A}_2$ separates $\mathcal{A}_1$ and $\mathcal{A}_3$, the factored planning problem corresponding to the network $(\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3)$ can be solved as suggested above. The

first step is to compute $M_{2,1} = \Pi_{\mathcal{A}_2}(\mathcal{A}_2 \times_A \mathcal{A}_3)$. Notice that the transition labeled by β' in $\mathcal{A}_2$ vanishes in $\mathcal{A}_2 \times_A \mathcal{A}_3$ (which is presented in Figure 5.4 (left) after a trimming step removing several unnecessary transitions), due to requested reading of 8 that is not provided by $\mathcal{A}_3$. This is reflected in the projection (Figure 5.4, right) where only one path to the goal remains possible for component $\mathcal{A}_2$. The grayed transitions in Figure 5.4 (right) indicate parts of the automaton that are not accessible or co-accessible (they can be removed by a trimming) or that are "redundant" (they vanish after a determinization step).

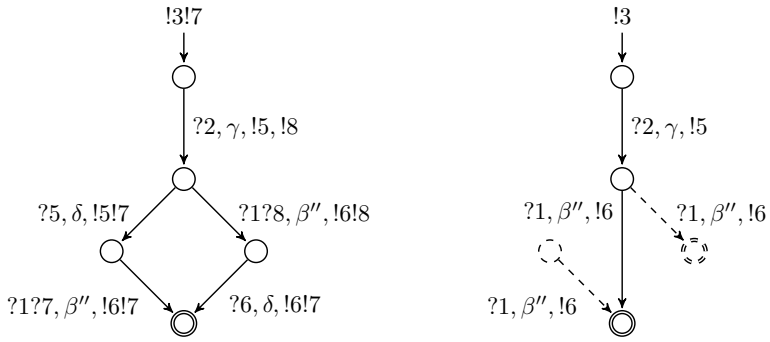


Figure 5.4: Product $\mathcal{A}_2 \times_A \mathcal{A}_3$ after trimming (left) and its projection on $\mathcal{A}_2$ before (right) and after trimming and determinization (right, solid part only).

As soon as $M_{2,1}$ is computed, it is possible to derive $\Pi_{\mathcal{A}_1}(\mathcal{A}_1 \times_A \mathcal{A}_2 \times_A \mathcal{A}_3) = \Pi_{\mathcal{A}_1}(\mathcal{A}_1 \times_A M_{2,1})$. The product $\mathcal{A}_1 \times_A M_{2,1}$ is depicted in Figure 5.5 (left). Notice that, due to reading tags, actions γ and β'' are not possible at some states. This makes necessary at least one execution of the word α, α', which was mandatory in $\mathcal{A}_1$ alone to reach its goal. This feature remains in the projection $\Pi_{\mathcal{A}_1}(\mathcal{A}_1 \times_A M_{2,1})$, depicted in Figure 5.5 (right).

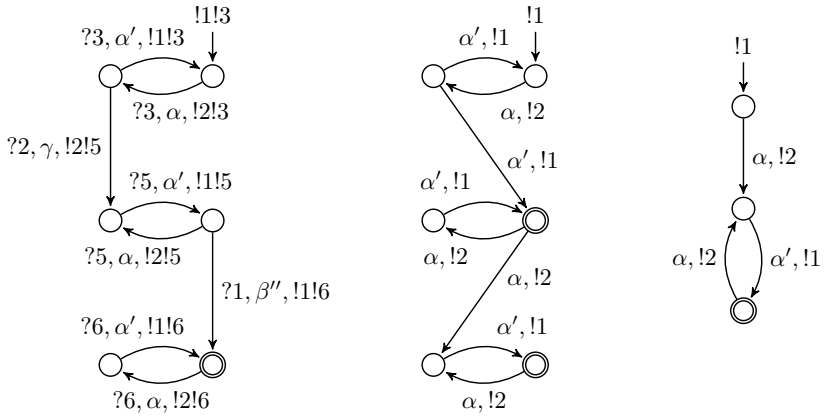


Figure 5.5: Product $\mathcal{A}_1 \times_A M_{2,1}$ after trimming (left), its projection $\Pi_{\mathcal{A}_1}(\mathcal{A}_1 \times_A M_{2,1})$ (center), and the same ARA after determinization and minimization (right).

One already has $\Pi_{\mathcal{A}_1}(\mathcal{A}_1 \times_A \mathcal{A}_2 \times_A \mathcal{A}_3)$: Figure 5.5, right. The same algorithm can be used to compute $\Pi_{\mathcal{A}_3}(\mathcal{A}_1 \times_A \mathcal{A}_2 \times_A \mathcal{A}_3)$ (Figure 5.6). In the particular case

of three ARA, computing $\Pi_{\mathcal{A}_2}(\mathcal{A}_1 \times_A \mathcal{A}_2 \times_A \mathcal{A}_3)$ (Figure 5.6) requires to compute $\mathcal{A}_1 \times_A \mathcal{A}_2 \times_A \mathcal{A}_3$. However, in larger networks our method will be of greater interest.

Figure 5.6: $\Pi_{\mathcal{A}_2}(\mathcal{A}_1 \times_A \mathcal{A}_2 \times_A \mathcal{A}_3)$ (left) and $\Pi_{\mathcal{A}_3}(\mathcal{A}_1 \times_A \mathcal{A}_2 \times_A \mathcal{A}_3)$ (right).

5.4 Generalization to any number of ARA

In this section, we show how the message passing algorithms can be used to solve factored planning problems represented by networks of more than 3 ARA. This generalizes the results of the previous section. The main difficulty is still that, in general, the projection of a product of ARA does not have the same language as the product of the projections of these ARA (axiom 2.3). As above, this difficulty prevents us from using the results of Chapter 2 and we thus derive a message passing algorithm from the separation property.

5.4.1 Communication graph of a network of ARA

It is possible to define a notion of interaction graph on networks of ARA. Then, from the separation property (Definition 5.15) one can define a notion of redundant edges in such graphs, which leads to a notion of communication graph for these networks as it was done in [28] for compound systems.

Definition 5.16. *The* interaction graph *of a network of ARA $(\mathcal{A}_1, \dots, \mathcal{A}_n)$ is the (non-directed) graph $G = (V, E)$ such that $V = \{\mathcal{A}_1, \dots, \mathcal{A}_n\}$ and there is an edge between $\mathcal{A}_i$ and $\mathcal{A}_j$ in E if and only if $\Sigma_i \cap \Sigma_j \neq \emptyset$, or $R_i \cap W_j \neq \emptyset$, or $R_j \cap W_i \neq \emptyset$.*

Definition 5.17. *In a graph $G = (V, E)$ and edge $(\mathcal{A}_i, \mathcal{A}_j) \in E$ is said to be* redundant *if and only if there exists a path $\mathcal{A}_i \mathcal{A}_{k_1} \dots \mathcal{A}_{k_\ell} \mathcal{A}_j$ in G with the following properties:*

1. *$\ell \geq 1$ and $\forall m, k_m \neq i \wedge k_m \neq j$,*
2. *$\forall m, \mathcal{A}_{k_m}$ separates $\mathcal{A}_i$ and $\mathcal{A}_j$.*

From this definition of redundant edges one can define the notion of communication graphs of a network of ARA. These graphs are constructed (slightly differently than in networks of weighted automata) by removing redundant edges from the interaction graph of a network.

Definition 5.18. *Let $G = (V, E)$ be the interaction graph of a network $(\mathcal{A}_1, \dots, \mathcal{A}_n)$ of ARA. Denote by $E' \subseteq E$ a maximal subset of edges such that any $(\mathcal{A}_i, \mathcal{A}_j) \in E'$ verifies:*

1. *$(\mathcal{A}_i, \mathcal{A}_j)$ is redundant in G,*
2. *there exists a path $\mathcal{A}_i \mathcal{A}_{k_1} \dots \mathcal{A}_{k_\ell} \mathcal{A}_j$ in the graph $(V, E \setminus E')$ such that $\forall m, \mathcal{A}_{k_m}$ separates $\mathcal{A}_i$ and $\mathcal{A}_j$.*

Any such E' defines a communication graph $G' = (V, E \setminus E')$ of $(\mathcal{A}_1, \dots, \mathcal{A}_n)$.

5.4.2 Message passing algorithm for ARA

Theorem 5.4 below allows one to derive a message passing algorithm (close to the one presented in Chapter 2) for factored planning in networks of automata with read arcs as soon as they admit a tree shaped communication graph.

In a tree shaped communication graph G each neighbor $\mathcal{A}_{i_\ell}$ of a vertex $\mathcal{A}_i$ induces a subtree. The root of this subtree is $\mathcal{A}_{i_\ell}$ and the rest of its vertices (denoted by $\mathcal{T}_{i_\ell}$) contains exactly the vertices reachable from $\mathcal{A}_{i_\ell}$ in G without using the edge between $\mathcal{A}_{i_\ell}$ and $\mathcal{A}_i$. The edges of this subtree are the ones from G.

Theorem 5.4. *Let G be a communication graph of the network of ARA $(\mathcal{A}_1, \dots, \mathcal{A}_n)$. If G is a tree, any non-leaf vertex $\mathcal{A}_i$ of G separates the subtrees induced by any two of its neighbors $\mathcal{A}_{i_\ell}$ and $\mathcal{A}_{i_m}$.*

Proof. Remark that the only path in G from any vertex $\mathcal{A}_\ell$ in the subtree induced by $\mathcal{A}_{i_\ell}$ to any vertex $\mathcal{A}_m$ in the subtree induced by $\mathcal{A}_{i_m}$ passes through $\mathcal{A}_i$. Then, two cases are possible.

1. Either there were no edge between $\mathcal{A}_\ell$ and $\mathcal{A}_m$ in the interaction graph of the network $(\mathcal{A}_1, \dots, \mathcal{A}_n)$. In this case $\Sigma_\ell \cap \Sigma_m = \emptyset$, $R_\ell \cap W_m = \emptyset$, and $R_m \cap W_\ell = \emptyset$, so $\mathcal{A}_i$ separates $\mathcal{A}_\ell$ and $\mathcal{A}_m$.

2. Or, in the interaction graph of $(\mathcal{A}_1, \dots, \mathcal{A}_n)$ there were an edge e between $\mathcal{A}_\ell$ and $\mathcal{A}_m$. In this case, as the path $\pi_{\ell,m}$ between $\mathcal{A}_\ell$ and $\mathcal{A}_m$ in G is unique and contains at least two edges (because $\mathcal{A}_i$ is on this path by construction), then e was a redundant edge and has been removed. By definition of the communication graph, there exists a path $\pi'_{\ell,m}$ between $\mathcal{A}_\ell$ and $\mathcal{A}_m$ in G such that any ARA on this path separates $\mathcal{A}_\ell$ and $\mathcal{A}_m$. The unicity of $\pi_{\ell,m}$ allows to conclude that $\pi_{\ell,m} = \pi'_{\ell,m}$. As $\mathcal{A}_i$ is on the path $\pi_{\ell,m}$ it separates $\mathcal{A}_\ell$ and $\mathcal{A}_m$.

□

From this theorem one has, for any network of state labeled ARA $\mathcal{A} = \mathcal{A}_1 \times_A \cdots \times_A \mathcal{A}_n$, and any tree shaped communication graph G of this network:

$$\begin{aligned}
\Pi_{\mathcal{A}_i}(\mathcal{L}_c(\mathcal{A})) &= \mathcal{L}_c(\Pi_{\mathcal{A}_i}(\mathcal{A})) \\
&= \mathcal{L}_c(\Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A \mathcal{A}_{i_1} \times_A \mathcal{T}_{i_1} \times_A \cdots \times_A \mathcal{A}_{i_k} \times_A \mathcal{T}_{i_k})) \\
&= \mathcal{L}_c(\Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A \Pi_{A_{i_1}}(\mathcal{A}_{i_1} \times_A \mathcal{T}_{i_1}) \times_A \cdots \times_A \Pi_{A_{i_k}}(\mathcal{A}_{i_k} \times_A \mathcal{T}_{i_k}))).
\end{aligned}$$

This is true due to the same reasons as Theorem 5.3 and using the fact that $\forall \ell, A_{i_\ell}$ separates $\mathcal{T}_{i_\ell}$ from the rest of the graph (direct consequence of Theorem 5.4).

The difference between the message passing algorithms in networks of ARA and the message passing algorithms in networks of finite automata comes from the fact that product and projection of ARA do not verify axiom 2.3. The general message passing algorithm for networks of ARA is formally described in Algorithm 10.

This algorithm takes as input a network of ARA $(\mathcal{A}_1, \dots, \mathcal{A}_n)$ and a tree shaped communication graph $G = (V, E)$ of this network. It outputs an updated version $\mathcal{A}'_i$ of each component $\mathcal{A}_i$ of this network. These updated components are such that $\mathcal{L}_c(\mathcal{A}'_i) = \Pi_{\mathcal{A}_i}(\mathcal{L}_c(\mathcal{A}_1 \times_A \cdots \times_A \mathcal{A}_n))$. Thus, a solution to the factored planning problem corresponding to $(\mathcal{A}_1, \dots, \mathcal{A}_n)$ can be extracted from these updated components, in the same manner as in Chapter 2.

Algorithm 10 MPA for networks of ARA

for all $(\mathcal{A}_i, \mathcal{A}_j) \in E$ **do**
  $\mathcal{M}_{i,j} \leftarrow 1$
end for
repeat
  select $(\mathcal{A}_i, \mathcal{A}_j) \in E$ s.t. $\mathcal{M}_{i,j}$ not updated and $\forall \mathcal{A}_k \in \mathcal{N}(\mathcal{A}_i) \setminus \{\mathcal{A}_j\}, \mathcal{M}_{k,i}$ was updated before
  $\mathcal{M}_{i,j} \leftarrow \Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A (\times_{A\,\mathcal{A}_k \in \mathcal{N}(\mathcal{A}_i)\setminus\{\mathcal{A}_j\}} \mathcal{M}_{k,i}))$
until all messages were updated exactly once
for all $\mathcal{A}_i \in V$ **do**
  $\mathcal{A}'_i = \Pi_{\mathcal{A}_i}(\mathcal{A}_i \times_A (\times_{A\,\mathcal{A}_k \in \mathcal{N}(\mathcal{A}_i)} \mathcal{M}k, i))$
end for

Conclusion

In this chapter we presented a new representation of factored planning problems. Its interest is to avoid the necessity of counting the reading actions in components on which they have no effect. The approach is based on automata displaying information on their states by means of the actions leading to these states. In fact, our networks of ARA are closely related to the asynchronous automata as described in [86].

We shown that it is possible to use an adaptation of the message passing algorithms on these networks of ARA in order to solve the factored planning problem they represent. The main difference with Chapter 2 is that the projections are performed over larger objects to allow propagation of readings between automata.

Notice also that, even if the model presented in this chapter does not take costs into account, it is possible to modify it for allowing factored cost-optimal planning. In fact, the difference between this model and the one of Chapter 2 is on the notion of accepted path only. Thus, the product and the projection operations presented in this section are very similar to the standard ones and costs can be taken into account similarly: using minimization during projection and summing costs of shared actions during product.

This chapter ends the part of this document dedicated to message passing algorithms for factored cost-optimal planning. The next two chapters present a different approach to factored planning, based on a distributed version of the A* algorithm. The main differences between the two approaches are the following. Firstly, the use of message passing algorithms gave rise to a top-down approach of factored planning: in each component the set of all local plans is refined until only the local plans part of global ones are kept. On the contrary, the distributed version of A* presented in the rest of this document is the basis for a bottom-up approach to factored planning: a local plan part of a global one is progressively built in each component along an execution of the algorithm. Secondly, the rest of this document presents a true distributed algorithm. Even if the approach using message passing algorithms can be used as a distributed algorithm, a centralized use of this approach can be preferable as it allows one to perform the minimal possible number of message updates.

Chapter 6

Toward a distributed A*

chapter abstract: *In this chapter we propose two distributed algorithm for solving two simple problems on weighted graphs. These algorithms, based on the standard A*, are the basis for the distributed algorithm for solving factored cost-optimal planning problems presented in Chapter 7.*

IN THE PREVIOUS part of this thesis we focused on the modular search for plans in factored planning problems by computation of the full set of the local plans which are part of the global plans. In other words we presented a top-down approach to planning: solutions were computed by refinement of a set containing them all. In this part we focus on a different approach: a single solution is built progressively. This is a bottom-up approach to planning. This will be done using a version of the A*-algorithm run by an agent in each component and biased by information received from the other agents. In order to fit with the presentation of A* of Section 1.2, in this part we consider factored planning problems given by graphs rather than automata. This is however only a question of vocabulary.

The approach of this part for building a distributed version of A* is to consider problems of increasing generality leading to factored planning problems focusing on the simple case of two agents (thus two graphs). The solution proposed for problems involving two agents is then generalized to problems involving any number of components as soon as their communication graphs are trees. The first problem we consider is called *compatible final states* (CFS). The principle of this problem is the following: two graphs with goal vertices are considered, along with two coloring functions on these goal vertices (one for each graph). The objective is to find a couple of paths, one in each graph, such that these paths reach goal vertices with identical color. Moreover, the sum of the costs of these paths has to be minimal. One can remark that CFS is in some sense a simplified version of factored planning. The differences are: the dynamic allocation of colors (the color of a vertex in a planning problem being the sequence of labels used to reach it, which depends on the path considered) and the infinite number of colors (the number of different possible local plans in a factored planning problem can not easily be bounded).

The second problem we consider is called *compatible colored paths* (CCP). It is, in fact, a version of CFS where colors are allocated dynamically to goal vertices, while still being taken in a finite set. More precisely, in this problem one considers a couple

of graphs with goal vertices and a coloring function over edges in each graph. The objective is to find a couple of paths that reach goal vertices, that use the same colors (the sets of colors on the edges constituting these paths are the same), and that minimize the sum of their costs. One then only has to handle infinite sets of colors to deal with factored planning problems, which is the purpose of Chapter 7.

This chapter is organized as follows. We first formally define CFS and give a method for solving it in the form of a distributed algorithm where an agent runs a slightly modified version of A* on each component (Section 6.1). Then, we formally define CCP and show how it can be re-casted as CFS, this allows to re-use the method proposed for solving CFS in the (closer to planning) context of CCP (Section 6.2).

6.1 Compatible final states

A CFS problem is defined by a couple $(\mathcal{P}_1, \mathcal{P}_2)$ of planning problems on graphs: $\mathcal{P}_k = (V_k, E_k, \Lambda_k, \lambda_k, c_k, i_k, F_k)$ for $k = 1, 2$ (as in Definition 1.2). $\mathcal{P}_1$ and $\mathcal{P}_2$ have no common actions, so $\Lambda_1 \cap \Lambda_2 = \emptyset$. However, their final vertices are "colored" by functions $\gamma_k : F_k \to \Gamma$ where Γ is a finite color set. The CFS problem amounts to finding an optimal distributed plan (p_1, p_2) where, instead of using Definition 2.3, the compatibility condition is defined by a compatibility of the final vertices with respect to coloring: p_1 and p_2 are compatible if and only if $\gamma_1(p_1^+) = \gamma_2(p_2^+)$. For simplicity, we shall assume that there is a unique optimal (common) final color, provided the CFS problem has a solution. Otherwise selecting one optimal final color among several becomes an agreement problem, that can be solved on top of our approach.

The approach we propose for solving CFS consists in associating an agent φ_k to each problem $\mathcal{P}_k$. Each agent performs an A*-like search in its local graph, and takes into account the constraints and costs of the other agent through an appropriate communication mechanism. Communications are asynchronous and can occur at any time. In fact, we consider that each agent can write into a memory that the other agent can only read. An agent does not need to know when the other agent modifies the objects stored into its memory. The only requirements will be on the characteristics of these objects.

6.1.1 Intuition on the approach

Let $\{k, \bar{k}\} = \{1, 2\}$. The agent φ_k attached to problem $\mathcal{P}_k$ relies on four functions. Two relate to the standard shape of a local A* and have to be computed locally by φ_k as usual:

- $g_k : V_k \to \mathbb{R}^+ \cup \{+\infty\}$ yields the (current) best known cost to reach any $v \in V_k$, and
- $h_k : V_k \times \Gamma \to \mathbb{R}^+ \cup \{+\infty\}$ is a set of heuristic functions towards F_k, one per terminal color. Equivalently, one has a heuristic function towards any $F_k \cap \gamma^{-1}(c)$ for $c \in \Gamma$.

Besides, two other functions inform φ_k on the state of the search in the other problem $\mathcal{P}_{\bar{k}}$. These functions have to be provided by $\varphi_{\bar{k}}$. Namely, one has:

- $H_{\bar{k}} : \Gamma \to \mathbb{R}^+ \cup \{+\infty\}$, which is a (generally) time varying heuristic that measures how much color $c \in \Gamma$ is promising at the current point of resolution of problem $\mathcal{P}_{\bar{k}}$, and

- $G_{\bar{k}} : \Gamma \to \mathbb{R}^+ \cup \{+\infty\}$, which eventually indicates the best cost for reaching color c in $\mathcal{P}_{\bar{k}}$.

Both values are stored and updated by agent $\varphi_{\bar{k}}$ in the memory he shares with agent φ_k. We now formalize these features and explain how these four functions are used and updated by each agent, how termination is detected by both of them, and how an optimal distributed plan solving the CFS is extracted.

6.1.2 Proposed algorithm

Let us consider first a non-varying distant heuristic $H_{\bar{k}}$: $H_{\bar{k}}(c) = h_{\bar{k}}(i_{\bar{k}}, c)$ for all $c \in \Gamma$ (it simplifies the presentation of our results and will be relaxed later). To the distant cost function on final colors $G_{\bar{k}}$ one associates an oracle $\Theta_{\bar{k}} : \Gamma \to \{null, optimal, useless\}$, with the following meaning.

- $\Theta_{\bar{k}}(c) = optimal$ means that a best plan towards final vertices of color c is known in $\mathcal{P}_{\bar{k}}$, and in that case $G_{\bar{k}}(c)$ represents the optimal cost to reach color c in $\mathcal{P}_{\bar{k}}$,
- $\Theta_{\bar{k}}(c) = useless$ means that $\varphi_{\bar{k}}$ can guarantee that for sure no optimal distributed plan $(p_k, p_{\bar{k}})$ exists which terminates in color c (for example because no local plan in $\mathcal{P}_{\bar{k}}$ terminates in color c), and
- $null$ is the remaining default (and initial) value of $\Theta_{\bar{k}}$.

This oracle satisfies the following property: for every color $c \in \Gamma$, there exists a finite time at which $\Theta_{\bar{k}}(c)$ jumps from $null$ to either $optimal$ or $useless$, and keeps this value forever. For the moment we assume that $G_{\bar{k}}$ and $\Theta_{\bar{k}}$ are provided to φ_k. After the presentation of our algorithm we will explain how these functions can be computed in practice.

Each agent φ_k executes the variant of A* given in Algorithm 11. Vertices can be marked in three different ways: open, closed, or candidate. A candidate vertex v belongs to F_k, and thus represents a local plan in $\mathcal{P}_k$ that can be proposed to $\varphi_{\bar{k}}$ as a possible local component of a distributed plan. Initially all vertices v in $V_k \setminus \{i_k\}$ are closed and satisfy $g_k(v) = +\infty$. To progressively open them and explore graph $\mathcal{P}_k$, one relies on the ranking function R_k defined as follows. If $v \in V_k$ is not candidate

$$R_k(v) \quad = \quad g_k(v) + \min_{c \in \Gamma} \left(h_k(v, c) + H_{\bar{k}}(c) \right)$$

which integrates the cost of color c for agent $\varphi_{\bar{k}}$, and then optimizes on the possible final color. For a candidate vertex v, one takes

$$\begin{aligned} R_k(v) \quad &= \quad g_k(v) + G_{\bar{k}}(\gamma_k(v)) \text{ if } \Theta_{\bar{k}}(\gamma_k(v)) = optimal \\ &= \quad g_k(v) + H_{\bar{k}}(\gamma_k(v)) \text{ otherwise} \end{aligned}$$

which associates to the possible final vertex v the cost of its color $\gamma_k(v)$ for agent $\varphi_{\bar{k}}$.

The recursive (local) search then proceeds as follows. At each iteration, φ_k selects the most promising non-closed (i.e. open or candidate) vertex v, i.e. the one that minimizes the ranking function R_k. According to the nature of v, agent φ_k either:

1. progresses in the exploration of $\mathcal{P}_k$ using an expansion function (Algorithm 12), this is the case in particular when v is open, or

2. checks whether it can draw some conclusion using the information provided by the other agent $\varphi_{\bar{k}}$. These conclusions can be:
   (a) that v is the goal vertex reached by a local path part of a globally optimal plan (line 9),
   (b) that v will never be the goal vertex reached by a local path part of a globally optimal plan (line 14), or
   (c) nothing for the moment (line 16 and line 11).

The reader familiar with A* may thus immediately identify its shape within Algorithm 11. The main difference lies in the stopping condition, due to the necessity to take into account constraints transmitted by the other agent.

Algorithm 11 executed by φ_k

```
1:  mark i_k open; g_k(i_k) ← 0; calculate R_k(i_k)
2:  while there exist non-closed vertices do
3:    let v be the non-closed vertex with minimal R_k(v)
4:    if v is open then
5:      expand(v)
6:    else
7:      /*v is candidate*/
8:      case: Θ_k̄(γ_k(v)) = optimal
9:        if R_k(v) = g_k(v) + G_k̄(γ_k(v)) then
10:         return v and terminate
11:       else
12:         calculate R_k(v)
13:       end if
14:     case: Θ_k̄(γ_k(v)) = useless
15:       mark v closed
16:     case: Θ_k̄(γ_k(v)) = null
17:       if there exists open vertices then
18:         let v' be the open vertex with minimal R_k(v')
19:         expand(v')
20:       end if
21:   end if
22: end while
```

Notice that the call to the $expand$ function at line 19 of Algorithm 11 is not required for termination nor validity, however it will allow agent $\varphi_{\bar{k}}$ to maintain $G_{\bar{k}}$ and $\Theta_{\bar{k}}$ using its own instance of Algorithm 11. Otherwise $\varphi_{\bar{k}}$ should run a standard A^* algorithm in parallel with Algorithm 11.

Theorem 6.1. *In this context, any execution of Algorithm 11 by φ_k on $\mathcal{P}_k$ terminates. Moreover, if the CFS problem $(\mathcal{P}_1, \mathcal{P}_2)$ has a solution, the output of Algorithm 11 for agent φ_k is a goal vertex $v_k \in F_k$, reached by a local plan p_k. The assembling of p_1 and p_2 provided by agents φ_1 and φ_2 resp. yields an optimal distributed plan (p_1, p_2) solving $(\mathcal{P}_1, \mathcal{P}_2)$.*

Theorem 6.1 is proved in three steps: first termination is proved (Lemma 6.1), then existence of an output when a distributed plan exists is proved (Lemma 6.2), and finally the fact that the output provides an optimal distributed plan is proved (Lemma 6.3).

Algorithm 12 expand function

1: **if** $v \in F_k$ **then**
2: mark v candidate
3: calculate $R_k(v)$
4: **else**
5: mark v closed
6: **end if**
7: **for all** v' such that $(v, v') \in E_k$ **do**
8: $g_k(v') \leftarrow \min(g_k(v'), g_k(v) + c_k((v, v')))$
9: **if** $g_k(v')$ strictly decreased **then**
10: mark v' open
11: $pred(v') \leftarrow v$
12: **end if**
13: calculate $R_k(v')$
14: **end for**

Lemma 6.1. *Algorithm 11 terminates when executed by φ_k on $\mathcal{P}_k$.*

Proof. Suppose φ_k executes Algorithm 11 on $\mathcal{P}_k$ without terminating. It means that there always exists a vertex which is either open or candidate (else the while loop would stop). It also means that the vertex with the smallest R_k value never fulfills the condition of line 9.

Moreover it is not possible to satisfy the condition of line 14 an infinite number of times in a row. This is because (1) V_k is finite, and thus there exists a finite number of possible candidates, and (2) when the condition of line 14 is satisfied a candidate becomes closed an no new vertex becomes candidate.

This implies that the $expand$ function will be called at finite time intervals while there exists open vertices. Hence, after some time all vertices will be either closed or candidate. This is due to the facts that (1) each call to $expand$ makes an open vertex become closed or candidate, (2) V_k is finite, and thus there exists a finite number of possible open vertices, (3) each v marked as open by $expand$ is such that $g_k(v)$ strictly decreased (line 9 of $expand$) and even, from the structure of the considered problems, one has that $g_k(v)$ strictly decreased of at least some constant c which is the minimal non zero difference between the costs of any two transitions from E_k, and (4) for a given v it is not possible that $g_k(v) < 0$ (this is due to the initialization of $g_k(i_k)$, line 1).

As soon as all vertices are either closed or candidate, no new vertex can become open. This is because only $expand$ function can open vertices and the conditions to call $expand$ require an open vertex (lines 17 and 4). Moreover, no new vertices will become candidate, for the same reason.

By definition of $\Theta_{\bar{k}}$, for any color $c \in \Gamma$ there exists a time after which either $\Theta_{\bar{k}}(c) = optimal$ or $\Theta_{\bar{k}}(c) = useless$. In particular, for any candidate (that is for any non-closed) vertex v, after some time, $\Theta_{\bar{k}}(\gamma_k(v)) = optimal$ or $\Theta_{\bar{k}}(\gamma_k(v)) = useless$. Consider the time after which, for any candidate vertex v, either $\Theta_{\bar{k}}(\gamma_k(v)) = optimal$ or $\Theta_{\bar{k}}(\gamma_k(v)) = useless$. Consider the vertex v selected. Two cases are possible: (1) $\Theta_{\bar{k}}(\gamma_k(v)) = useless$, v is closed, the number of candidate vertices strictly decreases, (2) $\Theta_{\bar{k}}(\gamma_k(v)) = optimal$, either algorithm terminate or $R_k(v)$ is calculated and becomes equal to $g_k(v) + G_{\bar{k}}(\gamma_k(v))$ (and thus if v is selected later the algorithm will terminate). This proves that, after some time, either all vertices will

be closed (if $\Theta_{\bar{k}}(\gamma_k(v)) = useless$ for all v candidate) or condition of line 9 will be satisfied. Both these cases are in contradiction with the hypothesis taken that φ_k executes Algorithm 11 on $\mathcal{P}_k$ without terminating. This proves Lemma 6.1. □

Lemma 6.2. *Algorithm 11 outputs some v when executed by φ_k on $\mathcal{P}_k$ if and only if there exists a solution.*

Proof. Suppose Algorithm 11 outputs no v when executed by φ_k on $\mathcal{P}_k$. It means it terminated because all vertices have been marked closed (Lemma 6.1). Thus all reachable goal vertices have been marked as candidate, by definition of the $expand$ function. Then all candidate vertices have been marked as closed, meaning that for any v candidate $\Theta_{\bar{k}}(\gamma_k(v)) = useless$. If there exists a solution it means that there exists some $v \in F_k$, reachable in $\mathcal{P}_k$, such that $\Theta_{\bar{k}}(\gamma_k(v)) = optimal$ at some time. This is not compatible with the facts that all reachable goal vertices have been marked as candidate and that for any v candidate $\Theta_{\bar{k}}(\gamma_k(v)) = useless$. Hence, if Algorithm 11 outputs no v when executed by φ_k on $\mathcal{P}_k$ then there exists no solution. Which proves that if there exists a solution then Algorithm 11 outputs some v when executed by φ_k on $\mathcal{P}_k$.

Suppose Algorithm 11 outputs some v when executed by φ_k on $\mathcal{P}_k$. It means that v has been marked candidate at some point. Which, by construction implies that $v \in F_k$ and there exists a path from i_k to v in $\mathcal{P}_k$. It also means that $\Theta_{\bar{k}}(\gamma_k(v)) = optimal$, which, by definition of $\Theta_{\bar{k}}$ implies that there exists a path in $\mathcal{P}_{\bar{k}}$ from $i_{\bar{k}}$ to some goal vertex with color $\gamma_k(v)$. Thus, there is a path in $\mathcal{P}_k$ reaching a goal vertex of color $\gamma_k(v)$ and there is a path in $\mathcal{G}_{\bar{k}}$ reaching a goal vertex of color $\gamma_k(v)$. This exactly means that there exists a solution. Which proves that if Algorithm 11 outputs some v when executed by φ_k on $\mathcal{P}_k$ then there exists a solution. □

Lemma 6.3. *When Algorithm 11 outputs some v, when executed by φ_k on $\mathcal{P}_k$, it is the goal vertex reached by a path p_k such that there is an optimal distributed plan $(p_k, p_{\bar{k}})$.*

Proof. Notice that any output v of Algorithm 11 is necessarily a goal vertex, by construction. Suppose Algorithm 11 outputs v, a goal vertex reached by a path p_k such that for any $p_{\bar{k}}$, $(p_k, p_{\bar{k}})$ is not an optimal distributed plan. Denote by v' a goal vertex reached by a path p'_k such that there exists an optimal distributed plan $(p'_k, p'_{\bar{k}})$ and $\Theta_{\bar{k}}(\gamma_k(v')) = optimal$ after some time. Such a v' exists because, as Algorithm 11 outputs v, by Lemma 6.2 there exists a solution to the considered planning problem. When Algorithm 11 stops by outputting v, two cases are possible: either (1) $g_k(v')$ is the optimal cost for reaching v' or (2) it is strictly greater than this optimal cost.

Consider case (1). For sure, v' is either open or candidate. The fact that $g_k(v') < \infty$ implies that v' has been marked as open at some time. From that, as $v' \in F_k$, it is not possible that v' has been marked as closed at line 5 of $expand$ function. Moreover, as $\Theta_{\bar{k}}(\gamma_k(v')) = optimal$ after some time, it is not possible that $\Theta_{\bar{k}}(\gamma_k(v'))$ has been equal to $useless$, and so it is not possible that v' has been marked as closed at line 15 of Algorithm 11. As $g_k(v')$ is the optimal cost for reaching v', one has $g_k(v') < g_k(v)$ because v is not part of a globally optimal plan. Moreover $H_{\bar{k}}(\gamma_k(v')) \leq G_{\bar{k}}(\gamma_k(v))$ by definition and if $\Theta_{\bar{k}}(\gamma_k(v')) = optimal$, $G_{\bar{k}}(\gamma_k(v')) \leq G_{\bar{k}}(\gamma_k(v))$. In conclusion one has $R_k(v') < R_k(v)$. This implies that, at line 3 of Algorithm 11, it is never possible to select v before v'. And, for this reason it is not possible to output v.

Consider case (2). Using the same argument than in the proof of the original A^* algorithm (Lemma 1.1), one can show that there exists an open vertex v'' such that: v'' is on a p'_k and $g_k(v'')$ is optimal. For the same reason as for case (1) it is not possible to select v before v'' at line 3 of Algorithm 11. And, so it is not possible to output v.

In both cases a contradiction has been given with the fact that Algorithm 11 can output a goal vertex reached by a path reached by a path p_k such that for any $p_{\bar{k}}$, $(p_k, p_{\bar{k}})$ is not an optimal distributed plan. This proves that when Algorithm 11 outputs some v, it is the goal vertex reached by a path p_k such that there is an optimal distributed plan $(p_k, p_{\bar{k}})$. □

6.1.3 Implementation of $G_{\bar{k}}$ and $\Theta_{\bar{k}}$

The remaining of this section gives a feasible construction of the distant (color) cost function $G_{\bar{k}}$ and of the oracle $\Theta_{\bar{k}}$, showing that Algorithm 11 is usable in practice. These two functions have to be computed by agent $\varphi_{\bar{k}}$ independently of problem $\mathcal{P}_k$, and in particular, independently of G_k and Θ_k. The $expand$ function is considered atomic: no update of $\Theta_{\bar{k}}$ or $G_{\bar{k}}$ will occur during the execution of this function by φ_k.

A possible implementation follows, where $\Theta_{\bar{k}}$ and $G_{\bar{k}}$ are computed within Algorithm 11 by $\varphi_{\bar{k}}$:

initialization: $\forall c \in \Gamma,\ G_{\bar{k}}(c) = +\infty$, and if $F_{\bar{k}} \cap \gamma_{\bar{k}}^{-1}(c) = \emptyset$ then $\Theta_{\bar{k}}(c) = useless$ otherwise $\Theta_{\bar{k}}(c) = null$,

update: as soon as some final vertex $v \in F_{\bar{k}}$ is open or candidate, if no other open vertex $v' \in V_{\bar{k}}$ satisfies $g_{\bar{k}}(v') + h_{\bar{k}}(v', \gamma_{\bar{k}}(v)) < g_{\bar{k}}(v)$, then color $\gamma_{\bar{k}}(v)$ can not be reached with a lower cost in $\mathcal{P}_{\bar{k}}$, so $\Theta_{\bar{k}}(\gamma_{\bar{k}}(v))$ is set to $optimal$ and

$$G_{\bar{k}}(\gamma_{\bar{k}}(v)) = \min_{v' \in F_{\bar{k}}, \gamma_{\bar{k}}(v') = \gamma_{\bar{k}}(v)} g_{\bar{k}}(v')$$

final update: when Algorithm 11 stops, for all $c \in \Gamma$ such that $\Theta_{\bar{k}}(c) = null$, set $\Theta_{\bar{k}}(c) = useless$, and $G_{\bar{k}}(c) = +\infty$.

Proposition 6.1. *For any $c \in \Gamma$ there exists a finite time after which either $\Theta_{\bar{k}}(c) = optimal$ or $\Theta_{\bar{k}}(c) = useless$. Moreover, as soon as the value of $\Theta_{\bar{k}}(c)$ is different from $null$ it no longer changes.*

Proof. Recall that all vertices are accessible in $\mathcal{P}_{\bar{k}}$. One can not rely on the values of Θ_k to prove Proposition 6.1, so, for that purpose, Lemma 6.1 can not be considered as true, the case where Algorithm 11 does not terminates thus has to be considered. Consider $c \in \Gamma$. If no goal vertex with color c exists, then, $\Theta_{\bar{k}}(c) = useless$ from the beginning. Else, two cases are possible: (1) Algorithm 11 terminates, and (2) Algorithm 11 does not terminate.

In case (1) all $c \in \Gamma$ for which $\Theta_{\bar{k}}(c) = null$ are set to $useless$ when Algorithm 11 terminates.

In case (2) it can be shown that the update will set all $\Theta_{\bar{k}}(c)$ which are $null$ to $optimal$ after some time. For the same reasons as in the proof of Lemma 6.1, after some time all vertices are either closed or candidate. Let $v \in F_{\bar{k}}$ be such a vertex, with $\gamma_{\bar{k}}(v) = c$, and $\Theta \neq null$. If v is candidate it means that $\Theta_{\bar{k}}(c) = optimal$ because there is no open vertex, and thus, in particular, no open vertex v' such that

$g_{\bar{k}}(v') + h_{\bar{k}}(v', c) < g_{\bar{k}}(v)$. If v is closed it means that v has been the candidate vertex with the smallest $R_{\bar{k}}$ at sometimes, and thus $\Theta_{\bar{k}}(c)$ as been set to $optimal$.

In all cases $\Theta_j(c) \neq null$, which proves the first part of Proposition 6.1.

The second part of the proposition is straightforward. Only initialization and final update can set $\Theta_{\bar{k}}(c)$ to $useless$. It is not possible that $\Theta_{\bar{k}}(c) = optimal$ before initialization. Moreover, final update can only change those $\Theta_{\bar{k}}(c)$ equal to $null$. Thus it is not possible for $\Theta_{\bar{k}}(c)$ to be set to $useless$ after having being set to $optimal$. Only update can set $\Theta_{\bar{k}}(c)$ to $optimal$. At that time the only c such that $\Theta_{\bar{k}}(c) = useless$ come from initialization. It means that they are such that no goal vertex with that color exists in $\mathcal{P}_{\bar{k}}$. Thus, it is not possible that a vertex with that color becomes either open or candidate. Hence, it is not possible to change $\Theta_{\bar{k}}(c)$ from useless to optimal.

This ends the proof of the second part of Proposition 6.1. □

Proposition 6.2. *For any $c \in \Gamma$ if $\Theta_{\bar{k}}(c) = optimal$ then the value of $G_{\bar{k}}(c)$ is the optimal cost in $\mathcal{P}_{\bar{k}}$ for reaching a goal vertex with color c. Moreover, if $\Theta_{\bar{k}}(c) = useless$, c can not be the color reached by a globally optimal solution.*

Proof. If $\Theta_{\bar{k}}(c) = optimal$ it means that, at some time, there existed $v \in F_{\bar{k}}$ such that v was open or candidate and $\gamma_{\bar{k}}(v) = c$, and there were no open vertex $v' \in V_{\bar{k}}$ such that $g_{\bar{k}}(v') + h_{\bar{k}}(v', c) < g_{\bar{k}}(v)$. At this time $G_{\bar{k}}(c)$ had been set equal to $g_{\bar{k}}(v'')$, where v'' is a goal vertex with color c minimizing the value of $g_{\bar{k}}$. $G_{\bar{k}}(c)$ is thus the cost of a path reaching color c in $\mathcal{P}_{\bar{k}}$. Assume it is possible to find a path with cost c_m strictly smaller than $G_{\bar{k}}(c)$. Let v''' be the goal vertex with color c reached by this path. By a similar argument than in the proof of the original A^* algorithm, either (1) $g_{\bar{k}}(v''') = c_m$, or (2) there exists an open vertex v'''' such that $g_{\bar{k}}(v'''') + h_{\bar{k}}(v'''', c) \leq c_m$. In case (1) v''' could have been selected as a goal vertex with color c minimizing the value of $g_{\bar{k}}$, this is in contradiction with the fact that $c_m < G_{\bar{k}}(c)$. In case (2) the existence of v'''' is in contradiction with the fact that there were no open vertex $v' \in V_{\bar{k}}$ such that $g_{\bar{k}}(v') + h_{\bar{k}}(v', c) < g_{\bar{k}}(v)$. This proves the first part of the Proposition 6.2.

If $\Theta_{\bar{k}}(c) = useless$ two cases are possible. Either no goal vertex exists with color c, in this case c can clearly not be the color reached by a globally optimal solution. Or Algorithm 11 stopped and $\Theta_{\bar{k}}(c)$ has been set to $useless$ during final update. In this case it is possible that no global solution exists, so c can not be the color reached by a globally optimal solution. It is also possible that a global solution exists, reaching color c'. In this case, necessarily, $c' \neq c$. This is due to the fact that, in this case, Algorithm 11, outputs this solution, and thus, just before that, a candidate vertex v of color c' had the minimal value of $R_{\bar{k}}$, thus $\Theta_{\bar{k}}(c')$ has been set to $optimal$. As a globally optimal solution exists reaching $c' \neq c$ it is not possible that a globally optimal solution exists reaching c. Hence, if $\Theta_{\bar{k}}(c) = useless$, c can not be the color reached by a globally optimal solution. This proves the second part of Proposition 6.2. □

6.1.4 Running example

We conclude this section by giving a possible execution of Algorithm 11 on a simple CFS problem. Consider the graphs of Figure 6.1. Heuristics h_1 should have the following properties: $h_1(i_1, r) \leq 1$, $h_1(v_1, r) \leq 0$, and $h_1(v, b) \leq +\infty$ for any v. In the same way the values of H_2 (provided to φ_1 by φ_2) should always be such that: $H_2(r) \leq 2$, and $H_2(b) \leq 2 + 0 = 2$.

Assume φ_1 is running Algorithm 11 on $\mathcal{P}_1$. Initially, i_1 is open, $g_1(i_1) = 0$, and $R_1(i_1) = g_1(i_1) + \min_{c \in \{r,b\}}(h_1(i_1, c) + H_2(c))$. All other vertices are closed and such

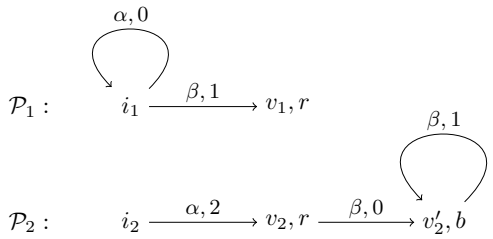


Figure 6.1: A CFS problem. Goal vertices are represented with their color (ex. v_1 is a goal with color r). Costs and labels are written above edges.

that g_1 is infinite. Moreover, $\Theta_1(b) = useless$, as no goal vertex with color b exists in $\mathcal{P}_1$. The first execution of the while loop will directly call the expand function, as i_1 is not candidate. It will be marked as closed (as i_1 is not a goal vertex). As $g_1(i_1) = 0 \leq 0 = g_1(i_1) + 0$, i_1 will not be re-opened. As $g_1(v_1) = +\infty > 1 = g_1(i_1) + 1$, v_1 will be opened, with $g_1(v_1) = 1$, and $R_1(v_1) = g_1(v_1) + \min_{c \in \{r,b\}}(h_1(v_1, c) + H_2(c))$, and $pred(v_1) = i_1$. After that, the expand function terminates. Immediately, $\Theta_1(r) = optimal$ as v_1 is a goal state with color r and no other open vertex exists, $G_1(r) = g_1(v_1) = 1$. As there are open vertices, a second execution of the while loop starts. The open or candidate vertex with minimal value of R_k is v_1. As v_1 is not candidate, a call to expand occurs immediately. As $v_1 \in F_1$, it is now candidate, and $R_1(v_1) = g_1(v_1) + G_2(r)$ if $\Theta_2(r) = optimal$ or $R_1(v_1) = g_1(v_1) + H_2(r)$ else. As v_1 has no neighbors, no new vertices are opened. From that, a new execution of the while loop occurs. As v_1 is candidate it is checked if it allows to conclude. No more calls to expand function occur as there no longer exists open vertices. As soon as $\Theta_2(r) = optimal$, with $G_2(r) = 2$ it is possible to conclude. The only possible local solution is to go from i_1 to v_1 in one step. Its cost is 1 locally, but $1 + 2 = 3$ globally, as the part of the solution in $\mathcal{P}_2$ is to go from i_2 to v_2 in one step.

6.2 Compatible colored paths

In this section we consider a second type of problems: CCP problems. CCP problems are in fact CFS problems with additional dynamicity. The principle is that the color of a goal vertex v is given by the path which allowed to reach it, rather than by v directly (so a vertex can assume several colors). However the number of possible colors remains finite as colors are defined as subsets of a finite set.

Formally, a CCP problem is defined by a couple $(\mathcal{P}_1, \mathcal{P}_2)$ of planning problems on graphs: $\mathcal{P}_k = (V_k, E_k, \Lambda_k, \lambda_k, c_k, i_k, F_k)$ for $k = 1, 2$. $\mathcal{P}_1$ and $\mathcal{P}_2$ have no common actions, so $\Lambda_1 \cap \Lambda_2 = \emptyset$. However, their edges are "colored" by functions $\gamma_k : E_k \to \Gamma$ where Γ is a finite color set. These coloring functions extend to paths in the following way: the color of a path $p_k = e_1 \dots e_n$ in $\mathcal{P}_k$ is $\gamma_k(p_k) = \cup_i \{\gamma_k(e_i)\}$. The CCP problem amounts to finding an optimal distributed plan (p_1, p_2) where the compatibility condition is defined with respect to coloring of paths: $\gamma_1(p_1) = \gamma_2(p_2)$. As above, we shall assume that there is a unique optimal (common) path coloring, if ever the CCP problem has a solution. Otherwise selecting one coloring among several becomes an agreement problem, that can be solved on top of our approach.

In fact, such a CCP problem $(\mathcal{P}_1, \mathcal{P}_2)$ with colors taken in Γ can be re-casted as the

(much larger) CFS problem $(\mathcal{P}'_1, \mathcal{P}'_2)$ with colors taken in $\Gamma' = 2^\Gamma$, such that:

- $V'_k = V_k \times 2^\Gamma$,
- $E'_k = \{((v, C), (v', C')) \ : \ (v, v') \in E_k \wedge C' = C \cup \{\gamma_k((v, v'))\}\}$,
- $\Lambda'_k = \Lambda_k$,
- for $(v, C) \in E'_k, \lambda'_k((v, C)) = \lambda_k(v)$,
- for $(v, C) \in E'_k, c'_k((v, C)) = c_k(v)$,
- $i'_k = (i_k, \emptyset)$,
- $F'_k = F_k \times 2^\Gamma$,
- for $(v, C) \in F'_k, \gamma'_k((v, C)) = C$.

Directly computing $(\mathcal{P}'_1, \mathcal{P}'_2)$ from $(\mathcal{P}_1, \mathcal{P}_2)$ would not be efficient: it implies a significative increase of the size of the problem. Moreover, as soon as one will want to solve real planning problem, the number of colors will in general be infinite (see next chapter). Thus, the corresponding CFS problem will as well be infinite. That is why, in this section, we propose a way to compute $(\mathcal{P}'_1, \mathcal{P}'_2)$ dynamically along the search for an optimal solution of $(\mathcal{P}_1, \mathcal{P}_2)$. The idea is that each agent φ_k will run Algorithm 11 starting from $i'_k = (i_k, \emptyset)$. The only difference will be in the expand function for which Algorithm 13 will be used. This new expand function is responsible for dynamically computing the part of $(\mathcal{P}'_1, \mathcal{P}'_2)$ needed for finding a solution to $(\mathcal{P}_1, \mathcal{P}_2)$.

Algorithm 13 expand function

```
/*expand function has been called with an argument v of the form (v', C)*/
if v' ∈ F_k then
    mark v = (v', C) candidate
    calculate R_k(v)
else
    mark v = (v', C) closed
end if
for all v'' such that (v', v'') ∈ E_k do
    C' ← C ∪ {γ_k((v', v''))}
    g_k((v'', C')) ← min(g_k((v'', C')), g_k((v', C)) + c_k((v', v'')))
    if g_k((v'', C')) strictly decreased then
        mark (v'', C') open
        pred((v'', C')) ← (v', C)
    end if
    calculate R_k((v'', C'))
end for
```

From that it is sufficient to prove that obtaining a solution of $(\mathcal{P}_1, \mathcal{P}_2)$ or $(\mathcal{P}'_1, \mathcal{P}'_2)$ makes effectively no difference, and that Algorithm 11 using the expand function of Algorithm 13 allows one to solve $(\mathcal{P}'_1, \mathcal{P}'_2)$ directly from $(\mathcal{P}_1, \mathcal{P}_2)$. This is what we do in the remaining of this section.

6.2.1 Equivalence of CFS and CCP

We first prove that the solutions of $(\mathcal{P}_1, \mathcal{P}_2)$ and $(\mathcal{P}'_1, \mathcal{P}'_2)$ are actually the same.

Proposition 6.3. *Any distributed plan in $(\mathcal{P}'_1, \mathcal{P}'_2)$ gives a distributed plan in $(\mathcal{P}_1, \mathcal{P}_2)$.*

Proof. Let (p'_1, p'_2) be a couple of paths constituting a distributed plan in $(\mathcal{P}'_1, \mathcal{P}'_2)$. One has $p'_k = (v_k^1, C_k^1) \dots (v_k^{n_k}, C_k^{n_k})$ is a path in (V'_k, E'_k). Notice that $C_k^1 = \emptyset$. Moreover, by definition of a distributed plan, one has $\gamma'_1(p'_1) = \gamma'_2(p'_2)$. By definition of $(\mathcal{P}'_1, \mathcal{P}'_2)$, $\gamma'_k(p'_k) = C_k^{n_k}$. Hence, $C_1^{n_1} = C_2^{n_2}$.

Let $p_k = v_k^1 \dots v_k^{n_k}$. We show that (p_1, p_2) is a couple of paths constituting a distributed plan in $(\mathcal{P}_1, \mathcal{P}_2)$. By definition of $(\mathcal{P}'_1, \mathcal{P}'_2)$, and more precisely of E'_k, there exists an edge between (v_k^i, C_k^i) and (v_k^{i+1}, C_k^{i+1}) in (V'_k, E'_k) only if there is an edge between v_k^i and v_k^{i+1} in (V_k, E_k). This proves that p_k is necessarily a path in (V_k, E_k). To prove that (p_1, p_2) is distributed plan in $(\mathcal{P}_1, \mathcal{P}_2)$ it remains to show that $\gamma_1(p_1) = \gamma_2(p_2)$. Notice that $\gamma_k(p_k) = \cup_i \gamma_k((v_k^i, v_k^{i+1}))$ by definition of $(\mathcal{P}_1, \mathcal{P}_2)$. Moreover, for any i, $C_k^{i+1} = C_k^i \cup \{\gamma_k((v_k^i, v_k^{i+1}))\}$ by construction of $(\mathcal{P}'_1, \mathcal{P}'_2)$. Thus, $\gamma_k(p_k) = C_k^{n_k}$. Which proves that $\gamma_1(p_1) = \gamma_2(p_2)$. □

Proposition 6.4. *Any distributed plan in $(\mathcal{P}_1, \mathcal{P}_2)$ gives a distributed plan in $(\mathcal{P}'_1, \mathcal{P}'_2)$.*

Proof. Let (p_1, p_2) be a couple of paths constituting distributed plan in $(\mathcal{P}_1, \mathcal{P}_2)$. One has $p_k = v_k^1 \dots v_k^{n_k}$ is a path in (V_k, E_k). By definition of a distributed plan one has: $\gamma_1(p_1) = \gamma_2(p_2)$. As $\gamma_k(p_k) = \cup_i \gamma_k((v_k^i, v_k^{i+1}))$ one has $\cup_i \gamma_1((v_1^i, v_1^{i+1})) = \cup_i \gamma_2((v_2^i, v_2^{i+1}))$.

Let $p'_k = (v_k^1, C_k^1) \dots (v_k^{n_k}, C_k^{n_k})$ such that $C_k^{i+1} = C_k^i \cup \gamma_k((v_k^i, v_k^{i+1}))$, and $C_k^1 = \emptyset$. We show that (p'_1, p'_2) is a couple of paths constituting a solution of $(\mathcal{P}'_1, \mathcal{P}'_2)$. By construction of $(\mathcal{P}'_1, \mathcal{P}'_2)$, one has that, if there is an edge $e_k = (v_k, v'_k) \in E_k$ then there is an edge in E'_k between (v_k, C_k) and $(v'_k, C_k \cup \{\gamma_k(e_k)\})$ for any C_k. Thus, for any i, there is an edge in E'_k between (v_k^i, C_k^i) and (v_k^{i+1}, C_k^{i+1}). This proves that p'_k is a path in (V'_k, E'_k). It remains to prove that $\gamma'_1(p'_1) = \gamma'_2(p'_2)$. One has $\gamma'_k(p'_k) = C_k^{n_k}$ by definition. Moreover, $C_k^{n_k} = C_k^{n_k - 1} \cup \gamma_k((v_k^{n_k - 1}, v_k^{n_k})) = \cup_i \gamma_k((v_k^i, v_k^{i+1})) = \gamma_k(p_k)$. So, finally, $\gamma'_1(p'_1) = \gamma_1(p_1) = \gamma_2(p_2) = \gamma'_2(p'_2)$. □

One can remark that given a distributed plan (p_1, p_2) for $(\mathcal{P}_1, \mathcal{P}_2)$, applying the construction of the proof of Proposition 6.4 to (p_1, p_2) leads to a distributed plan (p'_1, p'_2) with an interesting property. Indeed, applying the construction of the proof of Proposition 6.3 to (p'_1, p'_2) leads back to (p_1, p_2). Moreover, p_k and p'_k in the proofs are always such that $c_k(p_k) = c'_k(p'_k)$. This ensures that Proposition 6.3 and Proposition 6.4 hold as well for cost-optimal distributed plans. This remark, Proposition 6.3, and Proposition 6.4 show that providing a solution of $(\mathcal{P}'_1, \mathcal{P}'_2)$ or providing a solution of $(\mathcal{P}_1, \mathcal{P}_2)$ is equivalent. For this reason one could run Algorithm 11 on $(\mathcal{P}_1, \mathcal{P}_2)$ for solving $(\mathcal{P}'_1, \mathcal{P}'_2)$. Has stated before, this is however not reasonable, as $(\mathcal{P}_1, \mathcal{P}_2)$ can be much larger than $(\mathcal{P}'_1, \mathcal{P}'_2)$. This is for this reason that we provided a new version of the expand function (Algorithm 13) which is responsible for computing $(\mathcal{P}_1, \mathcal{P}_2)$ dynamically along the execution of Algorithm 11. This is proven in the next proposition.

Proposition 6.5. *For agent φ_k, executing Algorithm 11 on $(\mathcal{P}_1, \mathcal{P}_2)$ using the expand function from Algorithm 13 is equivalent to executing Algorithm 11 on $(\mathcal{P}'_1, \mathcal{P}'_2)$ using the expand function from Algorithm 12.*

Proof. First notice that the initial state of the system is the same in both cases. Initially, $i'_k = (i_k, \emptyset)$ is open, with $g_k(i'_k) = 0$ and

$$R_k(i'_k) = g_k(i'_k) + \min_C(h_k(i'_k, C) + H_{\bar{k}}(C)).$$

All other couples (v, C) are closed and such that $g_k((v, C)) = +\infty$. The value of $R_k((v, C))$ has not been calculated yet.

Consider a state of the system, that is, for each (v, C), a marking (open, closed or candidate), and a value for $g_k((v, C))$ and $R_k((v, C))$. The goal is to show that, whatever is the next step executed in Algorithm 11 by φ_k it will lead to the same state independently of the fact that $(\mathcal{P}_1, \mathcal{P}_2)$ (with expand function of Algorithm 13) or $(\mathcal{P}'_1, \mathcal{P}'_2)$ (with expand function of Algorithm 12) is considered. Notice that the only possible difference is during a call to the expand function, as the rest of the algorithm is exactly identical. The expand function is always called on an open couple (v, C) minimizing the value of $R_k(v, C)$. So, when considering both $(\mathcal{P}_1, \mathcal{P}_2)$ and $(\mathcal{P}'_1, \mathcal{P}'_2)$, expand is called on the same (v, C). We show that the effects of the call to expand on (v, C) are independent from its implementation (that is Algorithm 12 or Algorithm 13).

After a call to expand a first effect occurs, changing the marking of (v, C) and potentially the value of $R_k((v, C))$. As $v \in F_k$ if and only if $(v, C) \in F'_k$ one directly has that the new marking of (v, C) does not depend of the expand function called. For the same reason, $R_k((v, C))$ is calculated at line 3 of Algorithm 12 if and only if it is calculated at line 4 of Algorithm 13. Moreover both calculations have the same result. Hence, the effects on (v, C) of both expand functions are the same.

After that, the neighbors of v (Algorithm 13) or (v, C) (Algorithm 12) are considered. The important fact to notice is that, (v', C') is a neighbor of (v, C) if and only if v' is a neighbor of v and $C' = C \cup \{\gamma_k((v, v'))\}$. So, in both expand functions the computation of $g_k((v', C'))$ is exactly the same, as well as the computation of $R_k((v', C'))$. Moreover, (v', C') is marked open with the same conditions. So, the effects of both expand functions on the neighbors are the same.

Thus, starting from a given state of the system, the next step executed in Algorithm 11 by φ_k will lead to the same state independently of the fact that $(\mathcal{P}_1, \mathcal{P}_2)$ (with expand function of Algorithm 13) or $(\mathcal{P}'_1, \mathcal{P}'_2)$ (with expand function of Algorithm 12) is considered. This concludes the proof of Proposition 6.5. □

Altogether, Propositions 6.3, 6.4, and 6.5, prove that using Algorithm 11 with the expand function from Algorithm 13 gives a solution to CCP problems.

6.2.2 Running example

As in previous section we conclude the study of CCP problems by giving a possible execution of Algorithm 11 on a simple such problem. Consider the graphs of Figure 6.2. Then, the corresponding CFS problem $(\mathcal{P}'_1, \mathcal{P}'_2)$ is as depicted on Figure 6.3.

An execution of Algorithm 11 by φ_1 on $\mathcal{P}_1$ starts by marking $(i_1, \emptyset)$ as open. Then, expand function is called on $(i_1, \emptyset)$. As i_1 is not a goal vertex it is marked closed. The neighbors of i_1 are i_1 itself (reached by an edge with color r) and v_1 (reached by an edge with color b). Moreover, at that time $g_1((i_1, \{r\})) = g_1((v_1, \{b\})) = +\infty$. Thus, $(i_1, \{r\})$ and $(v_1, \{b\})$ are marked open. Immediately, $\Theta_1(\{b\})$ is set to *optimal* and $G_1(\{b\}) = 1$. From that there exists two open or candidate elements. One of them is selected, depending on the values of heuristics and of the costs of the edges. Suppose $(v_1, \{b\})$ is selected. It is then marked as candidate by expand function. No new elements are opened as v_1 has no successor. After that, either $(v_1, \{b\})$ (candidate) or

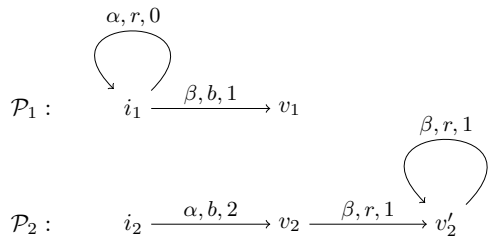


Figure 6.2: A CCP problem. Labels, colors, and costs are written above edges (in this order). All non-initial vertices are goal vertices.

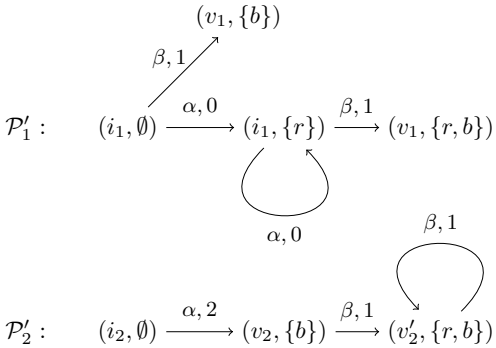


Figure 6.3: CFS problem corresponding to the CCP problem of Figure 6.2. We recall that the color of a goal vertex (v, C) is C. For example, the goal vertex $(v_1, \{b\})$ has color $\{b\}$.

$(i_1, \{r\})$ (open) is selected. In both cases, if $\Theta_2(\{b\}) = null$, expand function is called on $(i_1, \{r\})$. It results in $(i_1, \{r\})$ being closed (and not re-opened) and $(v_1, \{r, b\})$ being opened. Immediately, $\Theta_1(\{r, b\})$ is set to $optimal$ and $G_1(\{r, b\}) = 1$. After that, if still $\Theta_2(\{b\}) = null$, another call to expand makes $(v_1, \{r, b\})$ candidate. Finally, after some time, $\Theta_2(\{b\}) = optimal$ with $G_2(\{b\}) = 2$.Then, necessarily, $R_1((v_1, \{b\})) = g_1((v_1, \{b\})) + G_2(\{b\}) = 3$. If $H_2(\{r, b\})$ is sufficiently tight, that is, $2 < H_2(\{red, blue\}) \leq 3$, φ_1 is immediately able to conclude that its part of the solution should reach v_1 and be with color $\{b\}$. Else, as soon as $\Theta_2(\{r, b\}) \neq null$ it is possible to conclude: if $\Theta_2(\{r, b\}) = useless$, $(v_1, \{r, b\})$ is discarded, and if $\Theta_2(\{r, b\}) = optimal$, then one has $R_1((v_1, \{b\})) < R_1((v_1, \{r, b\}))$.

We let the execution of Algorithm 11 by φ_2 on $\mathcal{P}_2$ to the reader as the reasoning is similar.

Conclusion

In this chapter we have presented two distributed problems: CFS and CCP. We first proposed a method for solving CFS, based on agents running local A* algorithms with a bias due to the other agent results. We shown that our algorithm is usable in practice by providing concrete implementations of all the functions needed for it to run correctly. After that we explained how CCP can be reduced to CFS, allowing the algorithm proposed for solving CFS to be re-used in the case of CCP.

The tools used for the study of these problems lays the foundations for building the distributed algorithm for solving factored planning problems which is presented in the next chapter.

Chapter 7

A#: a Distributed A* for Cost-Optimal Planning

chapter abstract: *In this chapter we extend the results of Chapter 6 to the case of factored planning problems with two components. After that we show how the approach can be generalized to factored planning problems with any number of components as soon as their communication graphs are trees. This leads to a distributed algorithm (that we called A#) for cost-optimal planning.*

IN THIS CHAPTER we focus on solving factored planning problems using the methods proposed in Chapter 6. We consider factored planning problems given as tuples of reachability problems on graphs. In other words, a planning problem $\mathcal{P}$ is of the form $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ where each $\mathcal{P}_i$ is a reachability problem on a graph as in Definition 1.2. The solution of such a problem is a tuple $(p_1, \ldots, p_n)$ of paths (one in each component) such that the corresponding sequences of labels are compatible (as in Chapter 2). We consider reachability problems on graph rather than automata in order to match the earlier presentation of A* (Section 1.2), so that the principles of our algorithm can be made more intuitive (this is however only a matter of notations).

The idea of this chapter is to reduce a general factored planning problem with two components to a CFS problem, in the same way than CCP has been reduced to CFS in the previous chapter. The main difference between these two reductions is that the CFS problem obtained from a factored planning problem usually involves an infinite graph, therefore specific techniques are necessary to accommodate this extra difficulty. A large part of this chapter is thus dedicated to dealing with these particular infinite CFS problems. We will see that a computation of the CFS problem corresponding to a factored planning problem is possible along the search for a distributed plan, allowing to deal with factored planning problems with two components as soon as they have solutions.

We also generalize our results to the factored planning problems with more than two components for which the communication graphs are trees. This is done by remarking that, from the point of view of each agent, everything behaves exactly as if the problem had only two components: the component the agent is working on and the set of all the other components.

This chapter is organized as follows. We first explain in details the relations be-

tween CFS and factored planning, explain how to deal with their differences, and show that our approach is correct (Section 7.1). After that we describe a possible generalization of this approach into a method for solving factored planning problems with more than two components (Section 7.2).

7.1 Distributed planning with two components

This section extends the algorithm proposed to solve CFS problems to the more general framework of factored cost-optimal planning problems in the limited case of two components (or distributed planning, DP). Formally, a DP problem is defined by a couple $(\mathcal{P}_1, \mathcal{P}_2)$ of reachability problems on graphs: $\mathcal{P}_k = (V_k, E_k, \Lambda_k, \lambda_k, c_k, i_k, F_k)$ for $k = 1, 2$. The DP problem amounts to finding an optimal distributed plan (p_1, p_2) where the compatibility condition is defined with respect to labeling of paths over the shared labels $\Lambda_1 \cap \Lambda_2$: $\lambda_1(p_1)_{|\Lambda_1 \cap \Lambda_2} = \lambda_2(p_2)_{\Lambda_1 \cap \Lambda_2} = w$ (when dealing with problems involving more components, this will meet the notion of compatibility of Chapter 2). We shall assume that there is a unique optimal (common) path labeling $w \in (\Lambda_1 \cap \Lambda_2)^*$, if ever the DP problem has a solution. Otherwise selecting one labeling among several becomes an agreement problem, that can be solved on top of our approach.

Compared to CFS, DP problems introduce two difficulties. First, colors are assigned dynamically to vertices: the color of vertex v is not given in advance by some coloring function γ, but is set as a function of the path p leading to this vertex $v = p^+$. We already dealt with this difficulty in the case of CCP problems, and adapt a similar approach in the case of DP problems. Secondly, rather than a finite set Γ of colors, one potentially has an infinite set: the idea is that the "color" of the vertex $v = p^+$ is the sequence of shared actions met along path p leading to v. And there is generally no (efficient) bound on the number of shared actions in a globally optimal distributed plan[1].

As we did for CCP problems, let us recast a DP problem $(\mathcal{P}_1, \mathcal{P}_2)$ as a CFS problem $(\mathcal{P}'_1, \mathcal{P}'_2)$ with color set $\Gamma = (\Lambda_1 \cap \Lambda_2)^*$, the set of sequences of shared actions. One has $\mathcal{P}'_k = (V'_k, E'_k, \Lambda'_k, \lambda'_k, c'_k, i'_k, F'_k)$ with:

- $V'_k = V_k \times \Gamma$,
- $E'_k = \{((v, w), (v', w')) \ : \ (v, v') \in E_k \wedge w' = w\, \pi_{\Lambda_1 \cap \Lambda_2}(\gamma_k((v, v')))\}$,
- $\Lambda'_k = \Lambda_k$
- for $((v, w), (v', w')) \in E'_k, \lambda'_k(((v, w), (v', w'))) = \lambda_k((v, v'))$
- for $((v, w), (v', w')) \in E'_k, c'_k(((v, w), (v', w'))) = c_k((v, v'))$,
- $i'_k = (i_k, \varepsilon)$,
- $F'_k = F_k \times \Gamma$,
- for $(v, w) \in F'_k, \gamma'_k((v, w)) = w$.

[1] Strictly speaking, there exists a bound since the product planning problem $\mathcal{P} = \mathcal{P}_1 \times \mathcal{P}_2$ is finite, and no optimal plan will cross twice the same global state. However, the interest of distributed planning is to deploy local reasonings in $\mathcal{P}_k$ without making assumptions on the size of the 'external' component $\mathcal{P}_{\bar{k}}$.

$(\mathcal{P}'_1, \mathcal{P}'_2)$ has however a major difference with CFS problems considered in Section 6.1 and Section 6.2: V'_1, E'_1, V'_2, and E'_2 may be infinite.

The remaining of this section is dedicated to extending the results of Section 6.2 to the case of the particular infinite graphs considered here. It will allow one to use Algorithm 11 along with the $expand$ function given in Algorithm 14 for solving DP problems. This new $expand$ function is in fact responsible for computing parts of P'_k from P_k, when needed. Three points related to finiteness of graphs where not addressed in Section 6.2 and have to be addressed now:

1. computation of $R_k((v, w))$ sometimes implies to take a minimum over an infinite number of elements,
2. termination of the algorithm relies on finiteness of the graphs,
3. computation of $G_{\bar{k}}$ and $\Theta_{\bar{k}}$ are not directly possible on infinite graphs as non-accessible colors cannot be determined at initialization.

Algorithm 14 expand function

*/*expand function has been called with an argument v of the form (v', w)*/*
if $v' \in F_k$ **then**
  mark $v = (v', w)$ candidate
  calculate $R_k(v)$
else
  mark $v = (v', w)$ closed
end if
for all v'' such that $(v', v'') \in E_k$ **do**
  $w' \leftarrow w\pi_\Gamma(\gamma_k((v', v'')))$
  $g_k((v'', w')) \leftarrow \min(g_k((v'', w')), g_k((v', w)) + c_k((v', v'')))$
  if $g_k((v'', w'))$ strictly decreased **then**
    mark (v'', w') open
    $pred((v'', w')) \leftarrow (v', w)$
  end if
  calculate $R_k((v'', w'))$
end for

7.1.1 Computation of R_k and $H_{\bar{k}}$

For any color w – or at least any color which may correspond to an optimal distributed plan – $H_{\bar{k}}(w)$ should give a lower bound on the cost of reaching w in $\mathcal{P}_{\bar{k}}$. Clearly, taking $H_{\bar{k}}(w) = 0$ for any w gives such a lower bound. However, it is usually better to get a tight bound in order to avoid as much exploration of the graphs as possible. For practical use of our algorithm, using a more accurate $H_{\bar{k}}$ would be recommended. An example of such an $H_{\bar{k}}$ is the following, where $w' < w$ denotes that w' is a prefix of w (recall that $H_{\bar{k}}$ is computed by the agent $\varphi_{\bar{k}}$):

$$H_{\bar{k}}(w) = \min(H^o_{\bar{k}}(w), H^c_{\bar{k}}(w), G_{\bar{k}}(w))$$

with:

$$H^o_{\bar{k}}(w) = \min_{\substack{(v_{\bar{k}}, w') \text{ open} \\ w' < w}} (g_{\bar{k}}((v_{\bar{k}}, w')), h_{\bar{k}}((v_{\bar{k}}, w'))),$$

$$H^c_{\bar{k}}(w) = \min_{\substack{(v_{\bar{k}},w') \text{ candidate} \\ w'<w}} (g_{\bar{k}}((v_{\bar{k}},w'))).$$

Notice that for any w it is possible to compute $H_{\bar{k}}(w)$, as the set of open and candidate (v,w) is always finite. Notice also that all the values of $H_{\bar{k}}$ can be computed by the agent φ_k from a finite number of values of $H_{\bar{k}}$ given by $\varphi_{\bar{k}}$: the $H_{\bar{k}}(w)$ such that $(v_{\bar{k}},w)$ is open or candidate. We denote the part of $H_{\bar{k}}$ corresponding to these values by $\hat{H}_{\bar{k}}$. One then has: $H_{\bar{k}}(w) = \min_{w'<w} \hat{H}_{\bar{k}}(w')$ (and this is a minimum over a finite number of values).

When (v,w) is candidate, the computation of R_k is not an issue, it can be done exactly as in the simpler cases of CFS and CCP. Notice that, when (v,w) is candidate, v is necessarily in F_k. Hence, when $\Theta_{\bar{k}}(w) = optimal$ one has:

$$R_k((v,w)) = g_k((v,w)) + G_{\bar{k}}(w),$$

and in other cases:

$$R_k((v,w)) = g_k((v,w)) + H_{\bar{k}}(w).$$

However, when (v,w) is open it is not possible to directly use the previous definition of $R_k((v,w))$ as it may involve the computation of a minimum over an infinite number of elements. First of all, computing R_k as before would require the computation of $h_k((v,w),w')$ for any color w'. We consider instead $h_k((v,w)) = \min_{w'} h_k((v,w),w')$, which is computable with standard heuristic computation techniques as a lower bound on the cost of a path in $\mathcal{P}_k$ from v to a goal vertex.

From that, when (v,w) is open, we suggest to compute $R_k((v,w))$ as follows:

$$R_k((v,w)) = g_k((v,w)) + h_k((v,w)) + \min_{w'>w} H_{\bar{k}}(w').$$

The second difficulty is that there may be an infinite number of colors w to consider when computing $\min_{w'} H_{\bar{k}}(w') = \min_{w'>w} H_{\bar{k}}(w')$. This suggests to add a constraint on $H_{\bar{k}}$: it should be such that $\min_{w'>w} H_{\bar{k}}(w')$ is computable for any w. Fortunately, using the implementation of $H_{\bar{k}}$ proposed above it is possible. One just has to remark that:

$$\min_{w'>w} H_{\bar{k}}(w') = \min(H_{\bar{k}}(w), \min_{w'>w} \hat{H}_{\bar{k}}(w')),$$

as the number of w where $\hat{H}_{\bar{k}}(w)$ is defined is always finite, this minimum can be computed.

7.1.2 Termination of the algorithm

The main difference here with the cases of CFS and CCP problems is that the termination of the algorithm is not ensured when there is no solution. This is due to the fact that the graph to explore is in general infinite. In fact, it would be possible to ensure termination, as there is a bound on the length of the color corresponding to a possible solution. This bound can be computed by considering the number of vertices in the product of $\mathcal{P}_1$ and $\mathcal{P}_2$: if a solution exists, one is such that it passes at most once in each vertex of this graph. However, as stated before, it is not straightforward to tightly compute this bound in a distributed manner. For this reason, in the following we focus on the case where there exists a solution.

Theorem 7.1. *In this context, if the considered DP problem $(\mathcal{P}_1, \mathcal{P}_2)$ has a solution, then: any execution of Algorithm 11 (using the expand function of Algorithm 14) by the agent φ_k on $\mathcal{P}_k$ terminates. Moreover, the output of Algorithm 11 for agent φ_k is a goal vertex $v_k \in F_k$, reached by a local plan p_k. The assembling of p_1 and p_2 provided by agents φ_1 and φ_2 resp. yields an optimal distributed plan (p_1, p_2) solving $(\mathcal{P}_1, \mathcal{P}_2)$.*

To prove theorem 7.1 one first proves that as soon as a solution exists for $(\mathcal{P}_1, \mathcal{P}_2)$, Algorithm 11 terminates and outputs a vertex (Lemma 7.1). Then, it is sufficient to notice that the proof of Lemma 6.3 (of Chapter 6) never relies on the assumption that V_1, E_1, V_2, or E_2 are finite, so this proposition also applies here.

Lemma 7.1. *When $(\mathcal{P}_1, \mathcal{P}_2)$ has a solution, Algorithm 11 terminates by outputting some (v, w) when executed by φ_k on $\mathcal{P}_k$.*

Proof. Assume $(\mathcal{P}_1, \mathcal{P}_2)$ has a solution but Algorithm 11 does not terminate when executed by φ_k. It means there is always an open or candidate couple (v, w). After some time, such a couple will be candidate. This is due to the fact that there exists a reachable goal vertex (else no solution would exist). After some time, any candidate couple that may be part of a solution will become the couple with the minimal value for R_i. This is due to the fact that

1. all couples (v', w') marked as open by a call to expand function with argument (v, w) are such that $g_k((v', w')) \geq g_k((v, w)) + c$, where c is the minimal cost of an edge in $\mathcal{P}_k$,
2. any candidate couple (v, w) that may be part of a solution is such that $R_k((v, w))$ is smaller than $+\infty$, and
3. as for any w, $\Theta_{\bar{k}}(w)$ has to take a value after some time and Algorithm 11 does not terminate, any candidate couple with minimal value for R_k will be discarded after some time.

After some time a couple (v, w) such that $\Theta_{\bar{k}}(w) = optimal$ will become candidate. Once again this is due to the existence of a solution. From the previous argument, such a couple will become the couple with minimal value for R_k. Thus, Algorithm 11 will terminate thanks to this couple. This is in contradiction with the non termination assumption. Thus this assumption is false: when $(\mathcal{P}_1, \mathcal{P}_2)$ has a solution, Algorithm 11 terminates.

For the second part of the lemma, assume Algorithm 11 terminates without outputting a (v, w). It means that, at some point, all couples (v, w) were closed. However, as $(\mathcal{P}_1, \mathcal{P}_2)$ has a solution, there exists a reachable couple (v', w') such that $v' \in F_k$ and after some time $\Theta_{\bar{k}}(w') = optimal$. As all couples (v, w) were closed it means that all the reachable part of $\mathcal{P}'_k$ has been explored (each (v, w) has been marked open at some time). As v' is a goal vertex, (v', w') has been marked candidate after being marked open. As after some time $\Theta_{\bar{k}}(w') = optimal$ it is not possible that (v', w') has been marked closed before Algorithm 11 terminated. This is in contradiction with our assumption, thus, Algorithm 11 can only terminate by outputting some (v, w). □

7.1.3 Computation of $G_{\bar{k}}$ and $\Theta_{\bar{k}}$

As before, these two functions have to be computed by agent $\varphi_{\bar{k}}$ independently of $\mathcal{P}_k$, and in particular, independently of G_k and Θ_k. A possible implementation, where $\Theta_{\bar{k}}$ and $G_{\bar{k}}$ are computed along execution of Algorithm 11 by $\varphi_{\bar{k}}$, is the following:

initialization: $\forall w \in \Gamma$, $\Theta_{\bar{k}}(w)$ is considered as $null$ and $G_{\bar{k}}(w) = +\infty$ (but only the $\Theta_{\bar{k}}(w) \neq null$ and the corresponding values of $G_{\bar{k}}$ are stored).

update (1): as soon as there exists $v \in F_{\bar{k}}$ such that (v, w) is open or candidate, $\Theta_{\bar{k}}(w) \neq useless$, and there is no open couple (v', w') such that $g_{\bar{k}}((v', w')) + h_{\bar{k}}((v', w')) < g_{\bar{k}}((v, w))$ and $w' < w$, $\Theta_{\bar{k}}(w) = optimal$ and

$$G_{\bar{k}}(w) = \min_{v' \in F_{\bar{k}}} g_{\bar{k}}((v', w)).$$

update (2): as soon as for a given w there exists no $w' < w$ and v such that (v, w') is open with $R_{\bar{k}}((v, w')) < +\infty$ or (v, w) is candidate with $R_{\bar{k}}((v, w)) < +\infty$, if $\Theta_{\bar{k}}(w) = null$, then $\Theta_{\bar{k}}(w)$ is set to $useless$.

final update: when Algorithm 11 stops, for all $w \in \Gamma$ such that $\Theta_{\bar{k}}(w) = null$, set $\Theta_{\bar{k}}(w) = useless$, and $G_{\bar{k}}(w) = +\infty$.

Proposition 6.1 and Proposition 6.2 still hold with this new manner of computing $\Theta_{\bar{k}}$ and $G_{\bar{k}}$. The only difference in the proofs lies in the detection of the non-reachable colors. One just has to remark that, after some time, any color w which can not be part of a distributed solution will be such that no prefix w' of w appears in an open couple (v, w') such that $R_{\bar{k}}((v, w')) < +\infty$ and no (v, w) will be candidate with $R_{\bar{k}}((v, w)) < +\infty$. Checking the value of $R_{\bar{k}}$ is necessary because for some w it is possible that $H_k(w)$ becomes equal to $+\infty$ if no goal vertex can be reached with color w in $\mathcal{P}_k$. This was already the case in previous problems (CFS and CCP) but as we now consider infinite graphs, there is no guarantee that in such cases where $R_{\bar{k}}((v, w)) = +\infty$ the paths starting from (v, w) will be explored. It is possible that an infinite number of couples (v', w') such that $R_{\bar{k}}((v', w')) < +\infty$ have to be considered before being able to treat (v, w). Hence, after some time, $\Theta_{\bar{k}}(w)$ will be equal to $useless$ thanks to update (2). After that it will never become equal to $optimal$ by definition of update (1).

7.1.4 Running example

We conclude this section by a possible run of A# on a small factored planning problem with two components. Consider the graph of Figure 7.1.

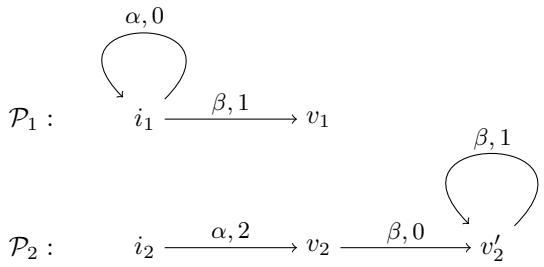


Figure 7.1: A DP problem. All non-initial vertices are goal. Costs and labels are written above edges.

Applying the transformation of DP problems into CFS problems proposed above , the graph $\mathcal{P}'_1$ would be as depicted in Figure 7.2.

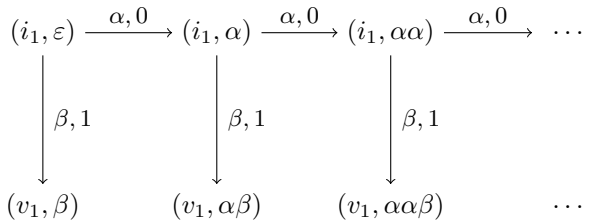


Figure 7.2: The DP problem of Figure 7.1 re-casted as a CFS problem (only $\mathcal{P}'_1$ is depicted). We recall that the color of a goal vertex (v, w) is w. For example, the goal vertex $(v_1, \alpha\beta)$ has color $\alpha\beta$.

An execution of Algorithm 11 by φ_1 on $\mathcal{P}_1$ starts with (i_1, ε) open. Then a call to expand function closes (i_1, ε) and opens (i_1, α) and (v_1, β). After that depending on the values of the different heuristics, a call to expand function will occur on either (i_1, α) or (v_1, β). Assume it is called on (i_1, α). Then (i_1, α) is closed and $(i_1, \alpha\alpha)$ and $(v_1, \alpha\beta)$ are opened. After that expand will be called on either (v_1, β), $(i_1, \alpha\alpha)$ or $(v_1, \alpha\beta)$. Which will either mark (v_1, β) or $(v_1, \alpha\beta)$ candidate, or close $(i_1, \alpha\alpha)$ and open $(i_1, \alpha\alpha\alpha)$ and $(v_1, \alpha\alpha\beta)$. After each time an element $(v_1, w\beta)$ is opened with $w \in \{\alpha\}^*$, $\Theta_1(w\beta) = \mathit{optimal}$ and $G_1(w\beta) = |w|.0 + 1$. As all costs of edges are positive, any open element of the form $(v_1, w\beta)$ with $w \in \{\alpha\}^*$ becomes candidate after a finite time. After some time $\Theta_2(\beta) = \mathit{useless}$ (it is not possible to reach a goal state in G_2 using only one edge with color β), and $\Theta_2(\alpha\beta) = \mathit{optimal}$ with $G_2(\alpha\beta) < \min(H_2(w\beta), G_2(w\beta))$ for all $w \in \{\alpha\}^*$ such that $\Theta_2(w\beta) \neq \mathit{optimal}$ and $G_2(\alpha\beta) < G_2(w\beta)$ for all $w \in \{\alpha\}^*$ such that $\Theta_2(w\beta) = \mathit{optimal}$. It allows φ_1 to conclude that its part of the optimal solution (which has a global cost of 3) reaches v_1 with color $\alpha\beta$. Moreover, the values of pred allow to conclude that the path in $\mathcal{P}_1$ should be to loop on i_1 one time and then go to v_1.

7.2 Generalization to any number of components

In this section we propose a generalization of the A#-algorithm above (Algorithm 11 using the expand function of Algorithm 14) to factored planning problems with any number of components. Our generalization is based on the fact that, from the point of view of an agent φ_k, any factored planning problem with n components can be considered as a factored planning problem with two components only. These two components - for which the communication graph is depicted in Figure 7.3 - are $\mathcal{P}_k$ and $\mathcal{P}_{\bar{k}} = (\mathcal{P}_1, \ldots, \mathcal{P}_{k-1}, \mathcal{P}_{k+1}, \ldots, \mathcal{P}_n)$. Thus, φ_k will simply be able to run Algorithm 11 on its component, as soon as she has access to $H_{\bar{k}}$, $G_{\bar{k}}$, and $\Theta_{\bar{k}}$.

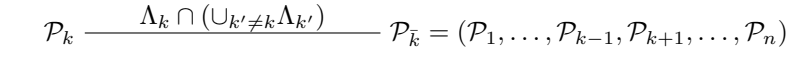


Figure 7.3: Communication graph of the factored planning problem $\mathcal{P} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ from the point of view of φ_k: it has two components.

Obviously, one however does not want to compute $\mathcal{P}_{\bar{k}}$ for each k, as this may be

intractable. Instead, the approach we propose, is for φ_k to use local information sent from its neighbors in a communication graph of $\mathcal{P}$ in order to compute $H_{\bar{k}}$, $G_{\bar{k}}$, and $\Theta_{\bar{k}}$. If this communication graph is not a tree, this computation is in general not possible (even if φ_k uses local information not only from its neighbors but from all the other agents).

The remaining of this section presents a possible way to compute $H_{\bar{k}}$, $G_{\bar{k}}$, and $\Theta_{\bar{k}}$ as soon as the considered factored planning problem lives on a tree. First, we describe the information needed from the neighbors of φ_k and how this information can be used to compute the functions needed by Algorithm 11 (7.2.1). Then, we explain how the neighbors of φ_k can provide the needed information to φ_k (7.2.2).

7.2.1 Building $H_{\bar{k}}$, $G_{\bar{k}}$, and $\Theta_{\bar{k}}$ in a distributed way

From here, we assume that the considered factored planning problem $\mathcal{P} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ lives on a tree. An example of such a tree is represented in Figure 7.4. This figure gives a particular representation of a tree shaped communication graph. The tree is rooted in $\mathcal{P}_k$. The $\mathcal{P}_{k_i}$ are the neighbors of $\mathcal{P}_k$. And for each $\mathcal{P}_{k_i}$, $\mathcal{T}_{k_i}$ is the subtree defined by all the nodes reachable from $\mathcal{P}_{k_i}$ without using the edge $(\mathcal{P}_{k_i}, \mathcal{P}_k)$.

The principle of the computation we propose for $H_{\bar{k}}$, $G_{\bar{k}}$, and $\Theta_{\bar{k}}$ is for the agent φ_k to receive information from each of its neighbors $\varphi_{k'}$, giving him estimations of the costs in the subtree $\mathcal{T}_{k'}$ rooted at $\mathcal{P}_{k'}$. More concretely, the agent $\varphi_{k'}$ will provide to the agent φ_k three functions: $H^k_{k'}$, $G^k_{k'}$, and $\Theta^k_{k'}$. These functions will have the following properties:

- $H^k_{k'} : \Lambda_k \cap \Lambda_{k'} \to \mathbb{R}^+ \cup \{+\infty\}$ is such that for any w, $H^k_{k'}(w)$ gives a lower bound on the cost of a distributed plan in $\mathcal{T}_{k'}$ (including $\mathcal{P}_{k'}$) compatible with w.
- $\Theta^k_{k'} : \Lambda_k \cap \Lambda_{k'} \to \{null, optimal, useless\}$ is such that for any w, $\Theta^k_{k'}(w) = null$ until, after a finite time, it becomes equal to $optimal$ or $useless$. Moreover it can only be equal to $useless$ if no cost-optimal distributed plan exists which is compatible with w.
- $G^k_{k'} : \Lambda_k \cap \Lambda_{k'} \to \mathbb{R}^+ \cup \{+\infty\}$ is such that for any w, as soon as $\Theta^k_{k'}(w) = optimal$, $G^k_{k'}(w)$ is the optimal cost of a distributed plan in $\mathcal{T}_{k'}$ (including $\mathcal{P}_{k'}$) compatible with w.

Using these functions it is quite straightforward to compute $H_{\bar{k}}$, $G_{\bar{k}}$, and $\Theta_{\bar{k}}$. One just has to give them the following values.

$$H_{\bar{k}}(w) = \sum_{k' \in \mathcal{N}(k)} H^k_{k'}(\pi_{\Lambda_{k'}}(w))$$

This effectively ensures that $H_{\bar{k}}(w)$ is a lower bound on the cost of a distributed plan in $\mathcal{P}_{\bar{k}}$ because as $\mathcal{P}$ lives on a tree the $\mathcal{T}_{k'}$ are independent: as soon as a distributed plan p' is found in any $\mathcal{T}_{k'}$ which is compatible with w any distributed plan p'' found in any $\mathcal{T}_{k''}$ (with $k' \neq k''$) which is compatible with w is necessarily compatible with p' as well. Hence, a sum of lower bounds in the $\mathcal{T}_{k'}$ is always a lower bound in $\mathcal{P}_{\bar{k}}$.

$$\Theta_{\bar{k}}(w) = \begin{cases} useless & \text{if } \exists k' \in \mathcal{N}(k), \Theta^k_{k'}(\pi_{\Lambda_{k'}}(w)) = useless \\ optimal & \text{if } \forall k' \in \mathcal{N}(k), \Theta^k_{k'}(\pi_{\Lambda_{k'}}(w)) = optimal \\ null & \text{else} \end{cases} \tag{7.1}$$

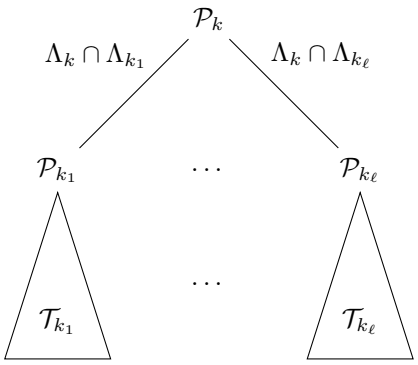


Figure 7.4: A communication graph with tree shape and the subtrees corresponding to the neighbors of $\mathcal{P}_k$. $\mathcal{T}_{k_i}$ represents the tree formed by the descendants of $\mathcal{P}_{k_i}$ in this tree rooted in $\mathcal{P}_k$.

When in one subtree $\mathcal{T}_{k'}$ it is possible to ensure that no distributed plan can possibly be part of a cost-optimal distributed plan of $\mathcal{P}$ involving some w in $\mathcal{P}_k$, one can ensure that w can not be part of a cost-optimal distributed plan. Thus, as soon as for some k', $\Theta^k_{k'}(\pi_{\Lambda_{k'}}(w)) = useless$ one can for sure set $\Theta_{\bar{k}}(w) = useless$. Moreover, when one knows the best cost of a distributed plan in each $\mathcal{T}_{k'}$, for sure it is possible to compute the best cost of a distributed plan in $\mathcal{P}_{\bar{k}}$ (for the same reasons that the computation of $H_{\bar{k}}$ is correct). Thus, as soon as for all k', $\Theta^k_{k'}(\pi_{\Lambda_{k'}}(w)) = optimal$ one can set $\Theta_{\bar{k}}(w) = optimal$. The fact that after some time for any w, $\Theta_{\bar{k}}(w) \neq null$ is ensured by the fact that for any w' and any k', after some time $\Theta^k_{k'}(w') \neq null$.

$$G_{\bar{k}}(w) = \sum_{k' \in \mathcal{N}(k)} G^k_{k'}(\pi_{\Lambda_{k'}}(w)).$$

First, remark that, if $\Theta_{\bar{k}}(w)$ (as defined in Equation 7.1) is optimal for some w, then, necessarily for all k', $\Theta^k_{k'}(\pi_{\Lambda_{k'}}(w)) = optimal$, and thus $G^k_{k'}(\pi_{\Lambda_{k'}}(w))$ is the optimal cost of a distributed plan in $\mathcal{T}_{k'}$ compatible with w. Hence, for the same reasons that the computation of $H_{\bar{k}}$ is correct, the sum of these $G^k_{k'}(\pi_{\Lambda_{k'}}(w))$ is the best possible cost of a path in $\mathcal{P}_{\bar{k}}$ compatible with w.

We shown that, if any agent $\varphi_{k'}$, neighboring φ_k, can provide the functions $H^k_{k'}$, $G^k_{k'}$, and $\Theta^k_{k'}$ to φ_k, then one can let each agent execute Algorithm 11 in order to solve any factored planning problem living on a tree.

7.2.2 Computing local information in practice

To conclude the presentation of A# in its full generality we propose a way to compute in practice the required functions $H^k_{k'}$, $G^k_{k'}$, and $\Theta^k_{k'}$ for any $k' \in \mathcal{N}(k)$. Notice that there may (certainly) exists better solutions for computing these functions. However, our goal here is only to show that A# is implementable.

The function for which it is the simplest to prove implementability is certainly $H^k_{k'}$. One just has to consider that it is always equal to 0. This is correct because all plans have cost greater or equal to 0, so 0 is a lower bound on the optimal cost in any set of plan, and in particular for any w, 0 is a lower bound on the optimal cost of a distributed plan in $\mathcal{T}_{k'}$ compatible with w. Notice that this is in fact this function that should be

improved for efficient use of A#: the most accurate $H^k_{k'}$ is, the most accurate is $H_{\bar{k}}$, and so the most efficient is the search.

Proposing an implementation of $\Theta^k_{k'}$ is more complex. First because it requires checking values of $\Theta^{k'}_{k''}$ for all $k'' \neq k \in \mathcal{N}(k')$. But mainly because $\Theta^k_{k'}$ has $\Lambda_k \cap \Lambda_{k'}$ for domain, while the $\Theta^{k'}_{k''}$ have $\Lambda_{k'} \cap \Lambda_{k''}$ for domain. For this reason several elements from the domain of $\Theta^{k'}_{k''}$ have to be considered in order to be able to decide optimality for a single element of the domain of $\Theta^k_{k'}$. That is why we suggest the following implementation of $\Theta^k_{k'}$ (and $G^k_{k'}$ which is closely related):

initialization: $\forall w \in \Lambda_k \cap \Lambda_{k'}$, $\Theta^k_{k'}(w)$ is considered as $null$ and $G^k_{k'}(w) = +\infty$.

update (1): $\forall w \in \Lambda_k \cap \Lambda_{k'}$, $\Theta^k_{k'}(w) = optimal$ as soon as there exists $v \in F_{\bar{k}}$ and w' such that $\pi_{\Lambda_k}(w') = w$, (v, w') is open or candidate, $\Theta^k_{k'}(w) \neq useless$, $\forall k'' \neq k \in \mathcal{N}(k')$, $\Theta^{k'}_{k''}(\pi_{\Lambda_{k''}}(w')) = optimal$, and there is no open couple (v', w'') such that $\pi_{\Lambda_k}(w'') < w$ with:

$$g_{k'}((v', w'')) + h_{k'}(v') + \min_{w''' > w''} \left(\sum_{\substack{k'' \in \mathcal{N}(k') \\ k'' \neq k}} H^{k'}_{k''}(\pi_{\Lambda_{k''}}(w''')) \right) \tag{7.2}$$

$$< g_{k'}((v, w')) + \sum_{\substack{k'' \in \mathcal{N}(k') \\ k'' \neq k}} G^{k'}_{k''}(\pi_{\Lambda_{k''}}(w')),$$

and in this case:

$$G^k_{k'}(w) = \min_{\substack{v' \in F_{k'} \\ \pi_{\Lambda_k}(w') = w}} \left(g_{k'}((v', w')) + \sum_{\substack{k'' \in \mathcal{N}(k') \\ k'' \neq k}} G^{k'}_{k''}(\pi_{\Lambda_{k''}}(w')) \right) \tag{7.3}$$

update (2): $\forall w \in \Lambda_k \cap \Lambda_{k'}$, as soon as there exists no (v, w') verifying the two following properties:

1. $\pi_{\Lambda_k}(w') < w$ and (v, w') is open, or $\pi_{\Lambda_k}(w') = w$ and (v, w') is candidate, and
2. $R_{k'}((v, w')) < +\infty$ and for all $k'' \neq k \in \mathcal{N}(k'), \Theta^{k'}_{k''}(\pi_{\Lambda_{k''}}(w')) \neq useless$,

if $\Theta_{\bar{k}}(w) = null$, then $\Theta^k_{k'}(w)$ is set to $useless$.

final update: when Algorithm 11 stops, for all $w \in \Lambda_k \cap \Lambda_{k'}$ such that $\Theta^k_{k'}(w) = null$, set $\Theta^k_{k'}(w) = useless$.

One may notice that, in the case of a factored planning problem with two components $(\mathcal{P}_k, \mathcal{P}_{\bar{k}})$, the values of $\Theta^k_{\bar{k}}$ (resp. $G^k_{\bar{k}}$) as defined here correspond exactly to the values of $\Theta_{\bar{k}}$ (resp. $G_{\bar{k}}$) as defined in Section 7.1.3. The principles of this computation are the same as in Section 7.1.3 but also take into account the costs coming from $\mathcal{T}_{k'}$. This imposes, for checking optimality for a given w, to consider all possible distributed plans in $\mathcal{T}_{k'}$ which may be compatible with any plan in $\mathcal{P}_{k'}$ giving w. The fact that, for any w, $\Theta^k_{k'}(w)$ will ultimately be different from $null$ is due to the tree shape of the communication graph considered. At the leaves of this tree the decision of optimality can be done

independently from other components as leaves have only one neighbor. From that, the neighbors of the leaves can decide optimality, and then their own neighbors can decide optimality, and so on. Finally, the computation of the minimums requested above can effectively be achieved, even if they are minimums over theoretically infinite sets. For the minimum in the computation of $G^k_{k'}$ (Equation 7.3) this is due to the fact that, at any time, there exists only a finite number of (v', w') such that $G^{k'}_{k''}(\pi_{\Lambda_{k''}}(w')) < +\infty$ for all $k'' \neq k \in \mathcal{N}(k')$. For the other minimum (Equation 7.2) this is due to the fact that it is not possible to store an infinite number of values for $H^{k'}_{k''}$, so in practice only a finite number of value will be considered, allowing computation of this minimum.

Conclusion

In this chapter we have extended the results of Chapter 6 in order to be able to solve factored cost-optimal planning problems, first with two components only and then with any number of components but living on trees. In each case we provided implementations for all the functions needed for estimating costs. This shows that our algorithm can be used in practice. Moreover, as for A*, it is possible to use different heuristics that the user can define as soon as they verify some properties. The private heuristic in each component (h_k) can be any admissible heuristic, in the same sense than in A*. The heuristics shared with other components ($H^{k'}_k$) should be such that for any $w, H^{k'}_k(w)$ gives a lower bound on the cost of a distributed plan compatible with w in the subtree $\mathcal{T}_k$ of the communication graph for which $\mathcal{P}_k$ is the root and which does not contain $\mathcal{P}_{k'}$. Moreover it should be possible to have access to this heuristic by keeping only a finite number of values in memory.

In order to improve this algorithm one could imagine driving the search in each local component so that candidate elements are treated as fast as possible. For example an agent could decide to temporary ignore the informations from other agents in order to quickly check if a given candidate is really of interest. An other modification could be to update values of R_k dynamically: when the value of some function changes, potentially modifying the value of R_k, one could recompute it and thus change the order in which elements will be considered.

Conclusion and Perspectives

THIS DOCUMENT has presented several results related cost-optimal factored planning, organized into two families of algorithms.

The first family relies on a message passing strategy, and aplies to planning problems encoded as networks of weighted automata. For each component of such a problem, they compute the set of all local plans that are part of a global plan. Equivalently, they discard from each component all local plans that can not be part of a global (or factored) plan, so these methods proceed by filtering. Moreover, they also output local plans that are part of a *cost-optimal* global plan. Extracting a cost-optimal global/factored plan is then simple: one has to isolate a tuple of optimal local plans, one per component, in such a way that they are compatible. This problem again can be solved with a message passing algorithm. One may argue that this last step is again equivalent to a factored planning problem, but the set of local plans to explore to build a compatible tuple can be much smaller than in the original planning problem. Two extensions of this approach have been proposed (read arcs and turbo planning), and this method has been implemented and experimented on classical benchmarks, with promising results. In particular, the experiments with turbo planning suggest that this approch can be extremely valuable for approximate factored cost-optimal planning on huge intractable models. Besides being a mixture of ideas borrowed to different research domains.

The second family of algorithms may look a little more familiar to the planning community since it extends the classical A* approach to a distributed setting. We called A# this multi-agent version of A*. It does not simply consist in running in parrallel several interacting searches for a plan on the same model/graph, as one would imagine. The idea here is that agents are each in charge of a component, in a network of interacting components. The main difference with the message passing approach is that the agents do not proceed by filtering out local plans that are not possible, they proceed in the reverse direction, by trying to *build* a local plan that would both match the proposals of the other agents (compatibility), and would make the so obtained factored plan globally cost-optimal. The driving idea is that each agents performs a best first search on his planning problem, like in a classical A*, but his search is biased by cost functions provided by its neighbors. These cost functions ensure that a consensus is reached on actions that must be performed in common, and guarantee that the obtained factored plan is optimal.

We detail below what we believe are the contributions of each chapter, before drawing some perspectives for this work.

Contributions

In Chapter 2 a first contribution is the idea of handling all plans of a component using weighted automata. A second one is to show that weighted automata along with well defined projection and product can be used as systems for message passing algorithms. The conjunction of these two ideas gives the main contribution of this chapter: the first – to our knowledge – factored planning algorithm allowing to find cost-optimal plans.

In Chapter 3 a first contribution is to show that the algorithm of the previous chapter can be implemented in practice using standard weighted automata algorithms. A second one is the idea of partial determinization of weigthed automata for dealing with non-determinizability. The main contribution is the first implementation of a factored cost-optimal planner (which is also one of the few implementations of factored planners) and the test of this planer on standard planning benchmarks.

In Chapter 4 a first contribution is the idea of using turbo algorithms for factored planning for providing over-approximations of the sets of local plans. A second contribution is an experimental study of this approach showing that approximate methods are of interest for factored planning.

In Chapter 5 the main contribution is the proposition of a new formalism for describing planning problems: automata with read arcs, along with the presentation of a product and a projection well suited for this formalism and the description of a new version of the message passing adapted to it.

In Chapter 6 and Chapter 7 the principal contribution is the description of a distributed version of A* and the proof of validity of this algorithm for solving planning problems. The presentation of two simple problems and algorithms for solving them (as well as proofs of validity for these algorithms) in Chapter 6 are side contributions of these chapters.

Perspectives

Decomposition of planning problems. A weakness of our approaches to factored planning is that they rely on the existence of a representation of the planning problem as a network of interacting components/sub-problems, this network taking ideally the shape of a tree. It is always possible to reshape a planning problem under that form, and methods exist to do so. Typically, one can start from components that represent each a single variable of the planning problem, which induces a dense interaction graph. Then one groups these elementary sub-problems into larger components until the resulting interaction graph is a tree, while at the same time minimizing the size of each component. Minimal size components are not aways the best optimization criterion, however, since this size is only a crude approximation to the actual complexity of the planning problem. It is probably better, for example, to gather components with strong interactions (in order to minimize internal concurrency), and conversely to keep components with little interaction separated, to take advantage of concurrency. One needs to identify the relevant criteria to perform such decompositions, and to develop the adequate combinatorial optimization techniques. Further, one could imagine that the actual decomposition used to solve a problem could be dynamic, i.e. could change along the execution of the planning algorithm. Notice that, even for turbo methods one

will certainly benefit from the use of a well chosen decomposition. In particular one would like to be able to decide when turbo methods should be preferred to standard methods requiring a tree-shaped communication graph.

Benchmarks adapted to factored planning. During our experiments we noticed a lack of factored planning benchmarks, and the factorization of standard planning problems is not straightforward. In order to be able to test and compare factored planning methods independently of problem decomposition issues, it would be of interest to propose realistic factored planning oriented benchmarks. We did a first step in this direction by designeing a generator of random factored problem, which was used in our experiments on turbo methods. However, the random generation of problems may be biased, it is not clear what the good parameters of these generators can be, and results obtained on random problems may not remain true on more realistic factored planning problems.

Metrics to compare weighted automata. In the standard setting of stochastic systems, the convergence analysis of turbo methods makes use of correlation metrics. Typically, if the correlation between components vanishes along cycles of the interaction graph, then it is likely that ignoring these cycles will not be damageable, and so turbo inference methods will behave well. Similarly, such metrics can be used to detect the converge of these iterative methods, when the messages exchanged bring vanishing information. In our implementation, we have used a simple distance between weighted automata, which gives importance to short words. This seems reasonable because we know that optimal words can not be extremely long. However we have to decide arbitrarily up to which length words should be considered: evaluating precisely the maximal possible length of a cost-optimal word is complex and may impose to look at the global problem. It would be of interest to consider other metrics on weighted automata, allowing to measure the proximity of two automata, and thus decide convergence of turbo algorithms. Better, one could hope being able to decide beforehand whether interactions are sparse enough to make turbo algorithms reliable. Naturally, such metrics would help as well building the appropriate problem decompositions mentioned above.

Taking advantage of internal concurrency. This work has proposed to model components either as weighted oriented graphs or as weighted automata. So the only notion of concurrency that was exploited lies between components, through the fact that private events of distinct components may not be causally related. The representation of global plans as tuples of compatible local plans allows one to model this concurrency. But local plans are still sequences of events, so the concurrency that is internal to components is completely unexploited. Clearly, this ability would be necessary, since for example decomposition methods proceed by grouping state variables in order to obtain a tree shaped interaction graph of components. Petri nets could therefore be an interesting alternative as component models, provided one is able to define the operations that are necessary to derive message passing algorithms. The product is standard and raises no difficulty, but the projection seems more problematic. Alternatively, one could imagine using more structured sub-classes of Petri nets, as causal nets for example, or other ways to encode sets of trajectories, as unfoldings and their variants.

More elaborate planing objectives. For a given factored planning problem $\mathcal{A} = \mathcal{A}_1 \times_{\mathcal{A}} \cdots \times_{\mathcal{A}} \mathcal{A}_n$, our message passing algorithm computes an updated version $\mathcal{A}'_i$

of each component $\mathcal{A}_i$ such that $\mathcal{L}(\mathcal{A}'_i) = \Pi_{\Sigma_i}(\mathcal{L}(\mathcal{A}))$. One may be interested in obtaining stronger properties on these new components $\mathcal{A}'_i$. For example to ensure that $\mathcal{A}'_i$ simulates $\Pi_{\Sigma_i}(\mathcal{A})$ or even that these objects are bisimilar. This would be of interest for finding plans with more complex requirements than just reaching a goal state in each sub-problem. Thinking about applications, one quickly realizes that distributed optimal planning should not only provide good or optimal paths to a goal, but also guarantee extra properties along this path. For example, one may wish to ensure that intermediate goal states are met, or conversely that specific intermediate states are avoided. Such issues appear in particular in reconfiguration problems for large systems, where one wishes to drive the system to a better or target state/configuration (e.g. for maintenance) and at the same time guarantee a minimal service disruption at any step. Similarly, security concerns may appear, if one wishes to drive the system to a better configuration while ensuring that on the path no known vulnerability is created. One may also consider how a distributed/factored plan is effectively implemented: each component knows what it must do, and when it must synchronize with its neighbors. But such rendez-vous require communications that may be difficult to implement. This is another source of constraints about the desirable plans one may wish to propose. Finally, the field of partially controllable components has not been addressed by this thesis.

Bibliography

[1] Eyal Amir and Barbara Engelhardt. Factored Planning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 929–935, 2003.

[2] Fahiem Bacchus and Qiang Yang. Downward Refinement and the Efficiency of Hierarchical Problem Solving. *Artificial Intelligence*, 71(1):43–100, 1994.

[3] Christer Bäckström. Equivalence and Tractability Results for SAS+ Planning. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning*, pages 126–137, 1992.

[4] Paolo Baldan, Nadia Busi, Andrea Corradini, and Giovanni Michele Pinna. Domain and Event Structure Semantics for Petri Nets with Read and Inhibitor Arcs. *Teoretical Computer Science*, 323(1-3):129–189, 2004.

[5] Paolo Baldan, Andrea Corradini, and Ugo Montanari. Contextual Petri Nets, Asymmetric Event Structures and Processes. *Information and Computation*, 171(1):1–49, 2001.

[6] Claude Berrou, Alain Glavieux, and Punya Thitimajshima. Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes. In *Proceedings of the IEEE International Conference on Communications*, pages 1064–1070, 1993.

[7] Jean Berstel. *Transductions and Context-Free Languages*. Electronic Edition, 2009.

[8] Avrim Blum and Merrick Furst. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, 90(1-2):281–300, 1995.

[9] Hans Bodlaender. A Linear Time Algorithm for Finding Tree-Decompositions of Small Treewidth. In *Proceedings of the 25th annual ACM Symposium on Theory of Computing*, pages 226–234, 1993.

[10] Blai Bonet and Héctor Geffner. Planning as Heuristic Search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[11] Blai Bonet, Patrik Haslum, Sarah Hickmott, and Sylvie Thiébaux. Directed Unfolding of Petri Nets. *Transactions on Petri Nets and other Models of Concurrency*, 1(1):172–198, 2008.

[12] Ronen Brafman and Carmel Domshlak. Factored Planning: How, When, and When Not. In *Proceedings of the 21st AAAI Conference on Artificial Intelligence*, pages 809–814, 2006.

[13] Ronen Brafman and Carmel Domshlak. From One to Many: Planning for Loosely Coupled Multi-Agent Systems. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling*, pages 28–35, 2008.

[14] Adam Buchsbaum, Raffaele Giancarlo, and Jeffery Westbrook. On the Determinization of Weighted Finite Automata. *SIAM Journal on Computing*, 30(5):1502–1531, 2000.

[15] Tom Bylander. Complexity Results for Planning. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 274–279, 1991.

[16] Christos Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic, 1999.

[17] Yixin Chen, Benjamin Wah, and Chih-Wei Hsu. Temporal Planning Using Subgoal Partitioning and Resolution in SGPlan. *Journal of Artificial Intelligence Research*, 26(1):323–369, 2006.

[18] Christian Choffrut. Une caractérisation des fonctions séquentielles et des fonctions sous-séquentielles en tant que relations rationnelles. *Theoretical Computer Science*, 5(3):325–337, 1977.

[19] Jeasik Choi and Eyal Amir. Factored Planning for Controlling a Robotic Arm: Theory. In *Proceedings of the 5th International Workshop on Cognitive Robotics*, pages 47–54, 2006.

[20] Joseph Culberson and Jonathan Schaeffer. Pattern Databases. *Artificial Intelligence*, 14(3):318–334, 1998.

[21] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGraw-Hill, 2008.

[22] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[23] Edsger Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[24] Stefan Edelkamp. Planning with Pattern Databases. In *Proceedings of the 12th International Conference on Automated Planning and Scheduling*, pages 13–24, 2001.

[25] Kutuhan Erol, Dana Nau, and V. S. Subrahmanian. On the Complexity of Domain-Independent Planning. In *Proceedings of the 10th AAAI Conference on Artificial Intelligence*, pages 381–386, 1992.

[26] Javier Esparza, Stefan Romer, and Walter Vogler. An Improvement of McMillan's Unfolding Algorithm. *Formal Methods in System Design*, 20(3):285–310, 1996.

[27] Eric Fabre. Convergence of the Turbo Algorithm for Systems Defined by Local Constraints. Technical Report RR-4860, INRIA, 2003.

[28] Eric Fabre. *Bayesian Networks of Dynamic Systems*. Habilitation à diriger des recherches, Université de Rennes1, 2007.

[29] Eric Fabre and Loïg Jezequel. Distributed Optimal Planning: an Approach by Weighted Automata Calculus. In *Proceedings of the 48th IEEE Conference on Decision and Control*, pages 211–216, 2009.

[30] Eric Fabre and Loïg Jezequel. On the Construction of Probabilistic Diagnosers. In *Proceedings of the 10th International Workshop on Discrete Event Systems*, pages 229–234, 2010.

[31] Eric Fabre, Loïg Jezequel, Patrik Haslum, and Sylvie Thiébaux. Cost-Optimal Factored Planning: Promises and Pitfalls. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling*, pages 65–72, 2010.

[32] Richard Fikes and Nils Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3):189–208, 1971.

[33] Cenk Gazen and Craig Knoblock. Combining the Expressivity of UCPOP with the efficiency of GRAPHPLAN. In *proceedings of the 4th European Conference on Planning*, pages 221–233, 1997.

[34] Malik Ghallab, Craig Isi, Scott Penberthy, David Smith, Ying Sun, and Daniel Weld. PDDL - The Planning Domain Definition Language. Technical report, Yale Center for Computational Vision and Control, 1998.

[35] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[36] Cordell Green. Application of Theorem Proving to Problem Solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 219–239, 1969.

[37] Peter Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[38] Peter Hart, Nils Nilsson, and Bertram Raphael. Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *ACM SIGART Bulletin*, 37:28–29, 1972.

[39] Patrik Haslum. Tp4'04 and HSP*-a. In *4th International Planning Competition Booklet*, pages 38–40, 2004.

[40] Patrik Haslum. *Admissible Heuristics for Automated Planning*. Phd thesis, Linköpings Universitet, 2006.

[41] Patrik Haslum. Reducing Accidental Complexity in Planning Problems. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 1898–1903, 2007.

[42] Patrik Haslum and Héctor Geffner. Admissible Heuristics for Optimal Planning. In *Proceedings of the 5th International Conference on Automated Planning and Scheduling*, pages 140–149, 2000.

[43] Patrik Haslum, Malte Helmert, and Jörg Hoffmann. Explicit-State Abstraction: A New Method for Generating Heuristic Functions. In *proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pages 1547–1550, 2008.

[44] Malte Helmert. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26(1):191–246, 2006.

[45] Malte Helmert and Carmel Domshlak. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling*, pages 162–169, 2009.

[46] Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible Abstraction Heuristics for Optimal Sequential Planning. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling*, pages 176–183, 2007.

[47] Sarah Hickmott, Jussi Rintanen, Sylvie Thiébaux, and Lang White. Planning via Petri Net Unfolding. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 1904–1911, 2007.

[48] Jörg Hoffmann, Stefan Edelkamp, Sylvie Thiébaux, Roman Englert, Frederico dos Santos Liorace, and Sebastian Trüg. Engineering Benchmarks for Planning: the Domains Used in the Deterministic Part of IPC-4. *Journal of Artificial Intelligence Research*, 26(1):453–541, 2006.

[49] Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered Landmarks in Planning. *Journal of Artificial Intelligence Research*, 22(1):215–278, 2004.

[50] Loïg Jezequel and Eric Fabre. Networks of Automata with Read Arcs: A Tool for Distributed Planning. In *Proceedings of the 18th IFAC World Congress*, pages 7012–7017, 2011.

[51] Loïg Jezequel and Eric Fabre. A#: A Distributed Version of A* for Factored Planning. In *Proceedings of the 51th IEEE Conference on Decision and Control*, page to appear, 2012.

[52] Loïg Jezequel and Eric Fabre. A-sharp: A Distributed A-star for Factored Planning. Technical Report RR-7927, INRIA, 2012.

[53] Loïg Jezequel and Eric Fabre. Turbo Planning. In *Proceedings of the 11th International Workshop on Discrete Event Systems*, page to appear, 2012.

[54] Andreas Junghanns and Jonathan Schaeffer. Domain-Dependent Single-Agent Search Enhancements. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 570–577, 1999.

[55] Erez Karpas and Carmel Domshlak. Cost-Optimal Planning With Landmarks. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 1728–1733, 2009.

[56] Henry Kautz, Bart Selman, and Jörg Hoffmann. SATPLAN: Planning as Satisfiability. In *5th International Planning Competition Booklet*, pages 45–46, 2006.

[57] Elena Kelareva, Olivier Buffet, Jinbo Huang, and Sylvie Thiébaux. Factored Planning Using Decomposition Trees. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 1942–1947, 2007.

[58] Daniel Kirsten. A Burnside Approach to the Termination of Mohri's Algorithm for Polynomially Ambiguous Min-Plus-Automata. *RAIRO - Theoretical Informatics and Applications*, 42(3):553–581, 2008.

[59] Daniel Kirsten. Decidability, Undecidability, and PSPACE-Completeness of the Twins Property in the Tropical Semiring. *Theoretical Computer Science*, 420:56–63, 2012.

[60] Daniel Kirsten and Sylvain Lombardy. Deciding Unambiguity and Sequentiality of Polynomially Ambiguous Min-Plus Automata. In *Proceedings of the 26th International Symposium on Theoretical Aspects of Computer Science*, pages 589–600, 2009.

[61] Daniel Kirsten and Ina Maürer. On the Determinization of Weighted Automata. *Journal of Automata, Languages and combinatorics*, 10(2):287–312, 2005.

[62] Ines Klimann, Sylvain Lombardy, Jean Mairesse, and Christophe Prieur. Deciding Unambiguity and Sequentiality From a Finitely Ambiguous Max-Plus Automaton. *Theoretical Computer Science*, 327(3):349–373, 2004.

[63] Craig Knoblock. Learning Abstraction Hierarchies for Problem Solving. In *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 923–928, 1990.

[64] Craig Knoblock. Automatically Generating Abstractions for Planning. *Artificial Intelligence*, 68(2):243–302, 1994.

[65] Craig Knoblock. Generating Parallel Execution Plans With a Partial-Order Planner. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems*, pages 98–103, 1994.

[66] Richard Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27(1):97–109, 1985.

[67] Richard Korf. Real-Time Heuristic Search. *Artificial Intelligence*, 42(2-3):189–211, 1990.

[68] Richard Korf. Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. In *Proceedings of the 14th National Conference on Artificial Intelligence*, pages 700–705, 1997.

[69] Sylvain Lombardy and Jean Mairesse. Series Which are Both Max-Plus and Min-Plus Rational are Unambiguous. *RAIRO - Theoretical Informatics and Applications*, 40(1):1–14, 2006.

[70] Robert McEliece, David MacKay, and Jung-Fu Cheng. Turbo Decoding as an Instance of Pearl's Belief Propagation Algorithm. *IEEE Journal on Selected Areas in Communications*, 16(2):140–152, 1998.

[71] Kenneth McMillan. Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits. In *Proceedings of the 4th International Workshop on Computer Aided Verification*, pages 164–177, 1993.

[72] Mehryar Mohri. Minimization of Sequential Transducers. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 151–163, 1994.

[73] Mehryar Mohri. On Some Applications of Finite-State Automata Theory to Natural Language Processing. *Journal of Natural Language Engineering*, 2(1):61–80, 1996.

[74] Mehryar Mohri. Finite-State Transducers in Language and Speech Processing. *Computational Linguistics*, 23(2):269–311, 1997.

[75] Mehryar Mohri. Weighted Finite-State Transducer Algorithms: An Overview. *Formal Languages and Applications*, 148(1):551–564, 2004.

[76] Mehryar Mohri. *Handbook of Weighted Automata*, chapter 6. Springer, 2009.

[77] Judea Pearl. Reverend Bayes on Inference Engines: A Distributed Hierarchical Approach. In *Proceedings of the 2nd National Conference on Artificial Intelligence*, pages 133–136, 1982.

[78] Judea Pearl. Fusion, Propagation, and Structuring in Belief Networks. *Artificial Intelligence*, 29(3):241–288, 1986.

[79] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1986.

[80] Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks Revisited. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pages 975–982, 2008.

[81] Jacques Sakarovitch. *Éléments de théorie des automates*. Vuibert, 2003.

[82] Rong Su. *Distributed Diagnosis for Discrete-Event Systems*. Phd thesis, University of Toronto, 2004.

[83] Rong Su and W. M. Wonham. Global and Local Consistencies in Distributed Fault Diagnosis for Discrete-Event Systems. *IEEE Transactions on Automatic Control*, 50(12):1923–1935, 2005.

[84] Rong Su, W. M. Wonham, James Kurien, and Xenofon Koutsoukos. Distributed Diagnosis for Qualitative Systems. In *Proceedings of the 6th International Workshop on Discrete Event Systems*, pages 169–174, 2002.

[85] David Thorsley and Demosthenis Teneketzis. Diagnosability of Stochastic Discrete-Event systems. *IEEE Transactions on Automatic Control*, 50(4):476–492, 2005.

[86] Wieslaw Zielonka. *The Book of Traces*, chapter 7. World Scientific, 1995.

Résumé

La planification est un domaine de l'intelligence artificielle qui a pour but de proposer des méthodes permettant d'automatiser la recherche et l'ordonnancement d'ensembles d'actions afin d'atteindre un objectif donné. Un ensemble ordonné d'actions solution d'un problème de planification est appelé un plan. Parfois, les actions disponibles peuvent avoir un coût ; on souhaite alors trouver des plans minimisant la somme des coûts des actions les constituant. Ceci correspond en fait à la recherche d'un chemin de coût minimal dans un graphe, et est donc traditionnellement résolu en utilisant des algorithmes tels que A*.

Dans ce document, nous nous intéressons à une approche particulière de la planification, dite factorisée ou modulaire. Il s'agit de décomposer un problème en plusieurs sous-problèmes (généralement appelés composants) le plus indépendants possibles, et d'assembler des plans pour ces sous-problèmes en un plan pour le problème d'origine. L'intérêt de cette approche est que, pour certaines classes de problèmes de planification, les composants peuvent être bien plus simples à résoudre que le problème initial.

La première partie de ce document présente une méthode de planification factorisée basée sur l'utilisation d'algorithmes dits à passage de messages. Une représentation des composants sous forme d'automates à poids nous permet de capturer l'ensemble des plans d'un sous-problème, et donc de trouver des plans de coût minimal, ce que ne permettaient pas les approches précédentes de la planification factorisée. Cette première méthode est ensuite étendue~: en utilisant des algorithmes dits « turbos », permettant une résolution approchée des problèmes considérés, puis en proposant une représentation différente des sous-problèmes, afin de prendre en compte le fait que certaines actions ne font que lire dans un composant.

La seconde partie de ce document présente une autre approche da la planification factorisée, basée sur une version distribuée de l'algorithme A*. Dans chaque composant, un agent réalise la recherche d'un plan local en utilisant sa connaissance du sous-problème qu'il traite, ainsi que des informations transmises par les autres agents. La principale différence entre cette méthode et la précédente est qu'il s'agit d'une approche distribuée de la planification modulaire.

Abstract

Automated planning is a field of artificial intelligence that aims at proposing methods to chose and order sets of actions with the objective of reaching a given goal. A sequence of actions solving a planning problem is usually called a plan. In many cases, one does not only have to find a plan but an optimal one. This notion of optimality can be defined by assigning costs to actions. An optimal plan is then a plan minimizing the sum of the costs of its actions. Planning problems are standardly solved using algorithms such as A* that search for minimum cost paths in graphs.

In this document we focus on a particular approach to planning called factored planning or modular planning. The idea is to consider a decomposition of a planning problem into almost independent sub-problems (or components). One then searches for plans into each component and try to assemble these local plans into a global plan for the original planning problem. The main interest of this approach is that, for some classes of planning problems, the components considered can be planning problems much simpler to solve than the original one.

The first part of this document proposes a study of the use of some message passing algorithms for factored planning. In this case the components of a problem are represented by weighted automata. This allows to handle all plans of a sub-problems, and permits to perform factored cost-optimal planning. Achieving cost-optimality of plans was not possible with previous factored planning methods. This approach is then extended by using approximate resolution techniques («turbo» algorithms) and by proposing another representation of components for handling actions which read-only in some components.

The second part of this document describes another approach to factored planning: a distributed version of the famous A* algorithm. Each component is managed by an agent which is responsible for finding a local plan in it. For that, she uses information about her own component, but also information about the rest of the problem, transmitted by the other agents. The main difference between this approach and the previous one is that it is not only modular but also distributed.

N° d'ordre : ENSC-2012 n°398

École normale supérieure de Cachan - Antenne de Bretagne

Campus de Ker Lann - Avenue Robert Schuman - 35170 BRUZ

Tél : +33(0)2 99 05 93 00 - Fax : +33(0)2 99 05 93 29