Calcul de plus courts chemins multicritères et problèmes géométriques connexes

Antonin Lentz

▶ **To cite this version:**

Antonin Lentz. Calcul de plus courts chemins multicritères et problèmes géométriques connexes. Algorithme et structure de données [cs.DS]. Université de Bordeaux, 2021. Français. NNT : 2021BORD0179 . tel-03346036

HAL Id: tel-03346036
https://theses.hal.science/tel-03346036

Submitted on 16 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE

par **Antonin LENTZ**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Multicriteria shortest paths and related geometric problems

Soutenue le 5 juillet 2021

Membres du jury :

Nicolas BONICHON	Maître de conférence, Université de Bordeaux	Examinateur
Prosenjit BOSE	Full Professor, Carleton University	Rapporteur
Johanne COHEN	Directrice de Recherche CNRS, Université Paris Sud	Examinatrice
David COUDERT	Directeur de Recherche INRIA, Université Côté d'Azur	Examinateur
Nicolas HANUSSE	Directeur de Recherche CNRS, Université de Bordeaux	Directeur
David ILCINKAS	Chargé de Recherche CNRS, Université de Bordeaux	Directeur
Colette JOHNEN	Professeure, Université de Bordeaux	Présidente
Laurent VIENNOT	Directeur de Recherche INRIA, IRIF	Rapporteur

Contents

Remerciements

Je tiens tout d'abord à remercier mes directeurs de thèse, pour votre investissement et votre persévérance. En particulier, merci Nicolas pour ta grande disponibilité et David pour ta rigueur à toute épreuve, nécessaire à toute activité sérieuse de recherche. Merci à Jit et Laurent d'avoir relu mon manuscrit, étape fondamentale de cette thèse. Merci aussi aux autres membres du jury, Niko, Colette, David et Johanne d'avoir accepté de m'évaluer.

Au-delà des chercheurs, je remercie le pôle administratif grâce à qui j'ai pu mener mes activités dans des conditions plus que décentes. N'oublions pas qu'elles le sont aussi grâce au personnel de ménage que je remercie.

Mais mes activités de recherches n'auraient pas été aussi plaisantes si elles n'étaient accompagnées d'enseignements. Je tiens donc à remercier les différents enseignants avec qui j'ai pu interagir et apprendre durant ces années. En particulier, merci à Guillaume pour ton investissement et ton efficacité. Merci à Carole de s'intéresser au bien-être des étudiants et d'apporter par là une vision plus humaine de l'enseignement. Merci Akka de m'avoir accompagné auprès d'une filière que beaucoup snobent mais qui est finalement la plus enrichissante pour les enseignants. Merci à Hervé pour tous les conseils et toutes les discussions informelles. On ne peut être que de bonne humeur en te parlant. Enfin, je tiens à remercier mes élèves, pour qui ce fut un réel plaisir d'enseigner, et grâce à qui j'ai découvert ma vocation.

Que ce soit dans un contexte studieux ou non, le bureau est le centre de la thèse. Je remercie donc les différentes personnes avec qui j'ai partagé ce lieu. Merci Dimitri pour tes cours de programmation et nos différentes discussions autour de l'enseignement. Merci Tobias pour les coups de main en mandarin. Promis, je m'y remets "bientôt". Merci Alexandre pour avoir su animer le bureau de discussions enflammées, autour du cinéma ou des USA notamment. Avengers Endgame n'existe pas. Voilà, c'est écrit. Merci à Théo pour l'accueil et l'aide à dépoussiérer un bureau croulant sous les fossiles à mon arrivée. D'ailleurs, nous avons encore une machine de Monroe si un archéologue passant par là est intéressé.

Au delà du bureau, les autres jeunes du labo y garantissent une bonne ambiance. Merci Rohan pour toutes les soirées B&T. Merci à Paul et Jason pour les divers coups de main. Merci Marthe pour les repas à Capperi et les gâteaux. Merci Lamine pour tes encouragements récurrents. Merci aussi à tous les autres du labo, Momo, Luis, Henri, Jonathan, Victor,...

En s'écartant un peu du LaBRI, on peut découvrir un laboratoire peuplé de gens tout aussi sympathiques, l'IMB. Merci Vassilis pour m'avoir plus appris le français que l'inverse (je ne plaisante pas). Merci Marc'Anto pour savoir animer les jeux de

sociétés comme jamais. Par contre, je ne me risquerais pas à faire un Jungle Speed contre toi. Merci Sergio pour ton sourire ineffaçable et ta classe irréprochable. Merci Yeeye pour les explications quant au mandarin. Merci Paul de toujours t'intéresser aux autres. Attention à ne pas bourdonner aux oreilles de Marc'Anto.

Qu'ils soient à l'IMB ou pas, je remercie ensuite tous les amis bordelais avec qui j'ai pu partager de nombreux moments autour d'un verre. Tout d'abord, je remercie les "anciens". Merci Nicolas pour ton humour fulgurant et pour m'avoir permis de survivre à ma première porte rouge. J'espère que le bureau du proviseur de Montaigne sera plus confortable que les précédents. Merci Jonathan de ne jamais accepter qu'une soirée se finisse. Tu manies si bien l'algorithme de Dijkstra que tu nous as trouvé un chemin pour visiter le massif de la Chartreuse sans quitter Bordeaux. Merci Nikola pour ta forte personnalité et tes plans improbables. Je sens que celui de cet été sera encore plein de surprises. Merci Roxan(n)e pour tes cocktails et ta gestion de la musique toujours impeccables. Merci Guillaume de partager ta passion du vin, des jeux de société, des films,... De tout en fait, merci de rendre les conversations intéressantes quelque soit le sujet. Merci Manon pour les discussions sur l'enseignement, les conseils syndicaux et pour m'avoir fait découvrir les meilleures pizzas de Bordeaux. Merci Elsa d'avoir été le véritable moteur social de l'IMB, et au-delà. J'espère qu'on pourra se revoir à Toulouse l'an prochain. Merci Sami qui parvient brillamment à allier simplicité et qualité à toutes ses blagues. Merci Karine pour ~~ton accent chantant~~ ta bonne humeur de tous les instants. Enfin, merci à Osman pour les soirées intemporelles.

Pour passer au moins anciens (comprenez ce que vous voulez, je voulais juste faire deux paragraphes), je remercie tout particulièrement Tomtom pour m'avoir intégré dès mon arrivée. Sans toi cette thèse n'aurait pas été possible. Merci tant pour les discussions recherche/enseignement/politique/random que pour ta deuxième (première ?) personnalité, Cristiano Traquenardo. Merci Aurore pour les véritables festins rue Ambroise. Merci aussi d'avoir pour vocation de faire sortir les chercheurs de leur trou, je te souhaite bien du courage. Merci à Corentin de savoir toujours mettre une ambiance folle, en faisant pouet-pouet, en programmant la musique et en faisant du Ricard au couteau. Merci Chloé pour les cours de respiration et les astuces pour se faire très vite beaucoup d'argent avec Twitch (j'ai essayé mais ça n'a pas marché...). Merci Alexandre pour les soirées jeux, pour toujours garder le rythme au BMF et pour m'avoir fait découvrir une façon tout à fait originale de porter son masque. Merci Baptiste pour toutes les raclettes, pour enflammer la piste de danse et pour ta bienveillance à toute épreuve. A quand la fin des travaux ? Merci Léa grâce à qui j'ai toujours pu être à jour sur les activités de Jean-Mi, le *meilleur* ami des profs. J'espère que Nobel est maintenant Papatte 40. Merci Lara d'avoir enfin apporté un peu de lumière sur tous ces groupies de la purée de pois chiche et de nous avoir fait découvrir le vrai Houmous. Merci Nicoletta de partager ton appréciation du Jack feu. Reviens nous vite d'Italie.

Une petite pause pour deux hors catégories, devinez celui qui n'est pas sincère. Merci à François Ruffin de ne pas limiter ses engagements à des discussions de soirées comme nous. Et merci aux différents rectorats m'ayant accueilli pour être tout aussi compétent qu'humain.

Même si je suis parti de Lyon depuis quelques années, je tiens à remercier ceux

qui m'y ont marqué et qui continuent à le faire. Ma première rencontre à Lyon, j'ai nommé Valou. Merci à notre petit diable des soirées (im)mémorables de mettre toujours une animation des plus imprévisibles. Heureusement que tu as arrêté de casser des steaks avec ton front pour qu'on fasse des burgers avec ! Merci Alex pour toutes les soirées liqueur de gingembre devant des films chinois particulièrement mauvais. Merci pour les discussions sérieuses et moins sérieuses. Merci Tomtom pour les toboggans, le club cocktail et les histoires de rubans. Je te pardonne de m'avoir honteusement doublé au tour de France, c'est de toute façon toi le meilleur. J'espère qu'ils n'attendent plus de nouvelles des respos animation. Merci Adri pour une coloc des plus inoubliables, les bons conseils à tout moment et les aprem Borderlands. Je ne parlerai pas de brownie ici. Merci Mathilde d'avoir supporté les animaux de coloc d'Adri lorsque tu lui rendais visite. Bon courage pour la fin de thèse. Merci Nadia pour les skypéros depuis la cabane au fond du jardin (ce n'est pas celle que vous croyez).

Merci Willy pour l'aide dans les cours de crypto imbitables et les sessions geek. J'espère que maintenant tu as compris que les assiettes n'étaient pas faites pour être lancées mais pour manger. Merci Edwin pour m'avoir fait découvrir l'Ardèche. Merci Quentin grâce à qui je connais les corrélations surprenantes avec le fait d'aimer la mousse de la bière. Merci Antoine pour les soirées films route de Vienne. Vivement la prochaine dégustation de Bourbon. Merci Robin pour les conseils culinaires et les recettes toujours plus folles. Merci Grégoire pour les vacances dans les Alpes et pour avoir partagé ta passion du handball. Merci à Benjamin pour ~~ton animalité~~ ta générosité et ta gentillesse hors paires. Merci Ronan pour m'avoir fait découvrir le salon des vignerons indépendants, fournisseur officiel de mon pot de thèse. J'essaie de te garder du Poulet & fils pour les Pyrénées. Merci Anissa pour tous les apéros déjantés au foyer. Merci Eva pour avoir organisé une semaine de vacances complètement folle. J'ai entendu parler d'une tonnelle pour le pot si il pleut, attention ! Merci Clémence d'avoir permis la création du pire instagram du monde. Essaie de ne pas lancer de combat de catch avec tes élèves. Merci Ritas pour les bons plans juridiques et pour m'avoir fait découvrir qu'on peut avoir 17 voitures mais plus de permis. Merci Clément Picard de nous faire vivre ta passion du football (et de la mauvaise foi). Merci Sam d'inventer des jeux toujours aussi bêtes et intelligents à la fois. Merci Halima pour les dégustations de vin à Bordeaux. Merci Philomène pour ton organisation toujours au top. Merci Fabio pour dégager plus de chill que quiconque.

Enfin, je terminerai en remerciant ma famille. En particulier, merci à ma mère pour m'avoir fourni une éducation et une instruction dont j'espère pouvoir être fier. Merci Camille pour les projections privées et les conseils cinéma toujours surprenants. Merci Jean-François pour m'avoir fait découvrir les livres qui m'ont marqué. Merci à Léo pour ses billes et ses cartes Pokemon. Par contre, non merci pour les cochons d'indes ! Merci à mon père pour m'avoir fait découvrir la guitare. Promis, je rebosse ça après la thèse. Merci Myriam pour ton accueil toujours chaleureux. Merci Victor pour les soirées films. Attention, il y a beaucoup de toilettes au LaBRI, ne loupe pas la soutenance ! Merci Éric pour ton soutien contre le quinoa et les légumes vapeurs. Vive la charcuterie et le fromage !

Résumé

Cette thèse s'intéresse au calcul de plus court chemin multicritère. Afin de mieux comprendre le travail effectué, nous commençons par présenter le contexte dans lequel on se place.

Plus courts chemins unicritères

Le calcul de plus court chemin dans un graphe pondéré est un problème fondamental en algorithmique des graphes. Lors d'un déplacement dans un réseau de transport, on peut vouloir minimiser le temps, la distance parcourue, ou encore le coût financier. Mais cette problématique ne se limite pas à ce type de réseau. Par exemple, on peut s'intéresser à la latence ou au coût dans des réseaux de communications.

Ce problème a été longuement étudié dans le cadre unicritère. Étant donné un graphe dont les arcs sont pondérés par des réels, le poids d'un chemin est la somme des poids des arcs qui le constituent. Parmi les chemins de poids minimaux, nous n'en conservons qu'un seul, peu importe lequel. Les deux solutions les plus connues sont les algorithmes de Dijkstra et de Bellman-Ford. Ces algorithmes ne sont hélas pas assez performants pour de très gros graphes.

Afin d'accélérer le calcul de plus court chemin, nombre de méthodes de pré-calcul ont été développées. Ces méthodes résument au préalable des informations sur un graphe donné. Puis, en utilisant ces informations, un algorithme permet de trouver rapidement un plus court chemin dans ce graphe entre n'importe quel couple de sommets. Pour se faire une idée des performances, certaines méthodes permettent actuellement de trouver en quelques micro-secondes un plus court chemin entre un couple de sommets donné dans un graphe ayant des millions de sommets. Le pré-calcul effectué en amont demande moins d'une heure, ce qui est raisonnable si l'on souhaite ensuite pouvoir répondre à de nombreuses requêtes de plus court chemin.

Plusieurs généralisations sont pertinentes. D'une part, il est intéressant de prendre en compte le dynamisme du graphe. En pratique, on remarque que l'utilisation des transports publics dans un trajet impose des restrictions temporelles. Des modèles permettent de prendre en compte cette composante temporelle et de nombreux algorithmes offrent des performances raisonnables. Cependant, l'association de moyens de transports ayant des contraintes (train, bus) et n'en ayant pas (à pied par exemple) n'est pas triviale et est encore actuellement sujette à beaucoup d'améliorations.

Une autre généralisation concerne la multiplicité des critères. Dans les réseaux de transport, le critère que l'on cherche habituellement à minimiser est le temps

de parcours. Comme on l'a dit précédemment, ce n'est pas le seul qui peut nous intéresser. De plus, le cadre multimodal introduit beaucoup de critères supplémentaires. On peut penser à l'effort physique dans le cadre d'un déplacement à vélo ou à pied, au rejet de CO_2 en voiture, ou encore au nombre de correspondances dans les transports en commun. Bien que cette thèse se restreigne aux graphes statiques, les graphes multimodaux sont d'une part une motivation pour la multiplicité des critères, mais aussi un cadre plus général dans lequel intégrer nos algorithmes.

Plus courts chemins multicritères

Il est naturel de chercher à prendre en compte ces différents critères simultanément. Étant donné d critères, les poids sont maintenant des vecteurs d-dimensionnels, chaque dimension correspondant à un critère. De même qu'en unicritère, si plusieurs chemins ayant les mêmes extrémités ont le même vecteur de poids, un seul nous intéresse. Un chemin est optimal si aucun autre chemin ayant les mêmes extrémités n'est meilleur sur tous les critères à la fois.

Une différence majeure avec le cas unicritère est que l'unicité de la longueur du chemin optimal n'est plus garantie. En effet, l'ordre sur les chemins n'est pas total puisqu'ils peuvent être incomparables. Prendre un hélicoptère pour se déplacer est beaucoup plus rapide que de marcher, mais c'est aussi beaucoup plus cher. D'autres moyens de transport, tels que le bus ou le taxi, constituent des chemins de coûts intermédiaires. Ils sont tous incomparables pour les deux critères temps et prix. Un ensemble de chemins représentant tous les chemins optimaux est appelé un ensemble de Pareto.

Le problème du plus court chemin devient donc, dans le cadre *multicritère*, la recherche d'ensembles de Pareto d'une source à une destination données. Le calcul d'un ensemble de Pareto dans ce contexte a été initié par Hansen [Han80] avec deux critères et généralisé à un nombre arbitraire de critères par Martins [Mar84] avec une variante de l'algorithme de Dijkstra.

Tout comme le cas unicritère, bien d'autres méthodes ont par la suite été développées afin d'accélérer ce calcul. Cependant, la taille des ensembles de Pareto peut rendre le calcul de ces ensembles infructueux. En effet, de simples petits graphes artificiels permettent des construire des ensembles de Pareto de taille exponentielle en le nombre de sommets. Bien que dans certains cas, notamment en 2D quand les critères sont corrélés, ces ensembles restent de taille raisonnable, l'augmentation du nombre de critères démultiplie le nombre de chemins optimaux.

Approximation

Afin de pallier à ce problème, l'approche générale est de tenter de résumer ces ensembles de Pareto. Pour cela, on calcule ce que l'on appelle des ensembles de Pareto approchés. Ce type d'ensemble est tel que pour tout chemin optimal, il existe un chemin dans l'ensemble de Pareto approché, qui, sur chaque critère, n'est pas pire que le poids du chemin optimal à une certaine constante multiplicative près. Par exemple, si cette constante est 2, un ensemble de Pareto approché est tel que

tout chemin optimal est au pire deux fois meilleur sur chacun des critères qu'un des chemins de l'ensemble approché. On dit que l'ensemble approché couvre l'ensemble de Pareto.

L'intérêt majeur de cette notion est que tout ensemble de Pareto est couvert par un sous ensemble de chemins de taille polynomiale. Ce dernier est facilement calculable si l'ensemble de Pareto est initialement connu. On peut donc extraire d'un ensemble de Pareto un ensemble de chemins représentatif et de taille raisonnable. Cependant, le problème principal est d'éviter de calculer l'ensemble de Pareto en entier. Il est donc nécessaire que n'importe quel algorithme de calcul d'ensembles de Pareto approchés élague des chemins optimaux durant son exécution. La difficulté majeure est de savoir lesquels.

Il existe deux grandes familles de solutions à ce problème. L'une, théorique, garantit que la sortie de l'algorithme couvre l'ensemble de Pareto et propose des complexités intéressantes. En terme de complexité, la meilleure connue est celle de Tsaggouris et Zaroliagis [TZ09]. Cependant, les différentes méthodes de cette école, et en particulier celle de Tsaggouris et Zaroliagis, sont inutilisables en pratique puisqu'elles élaguent trop peu de chemins optimaux.

A l'inverse, d'autres méthodes proposent un élagage brutal afin d'obtenir des performances en pratique bien plus raisonnables [BBS13; Hrn+17]. Le problème est que ce gain s'accompagne d'une perte importante : la sortie n'est pas nécessairement une couverture de l'ensemble de Pareto. De plus, les complexités sont rarement étudiées dans ce cadre.

Enfin, des compromis récents proposent de reprendre les méthodes théoriques en les modifiant légèrement pour obtenir des performances raisonnables [BDH17]. Néanmoins, les complexités peuvent même devenir moins intéressantes que celles correspondant à un calcul exact. Et afin d'être utilisable sur des graphes de tailles importantes, il faut tout de même supprimer la garantie de couverture.

Une dernière difficulté est que les méthodes existantes ne garantissent pas que la sortie n'est constituée que de chemins optimaux. Il est donc difficile de comparer pertinemment les complexités des différents algorithmes en fonction de la taille de leur sortie.

Contributions. *Nous proposons des compromis afin d'obtenir à la fois une couverture de l'ensemble de Pareto, mais aussi une complexité et des performances raisonnables. Plus précisément, nous proposons un méta-algorithme* Meta Rank *qui peut être dérivé de différentes façons pour un calcul d'ensembles de Pareto exacts ou approchés. Ce méta algorithme est une variante de l'algorithme de Martins, et donc de celui de Dijkstra. Une première version,* Bucket*, calcule des ensembles de Pareto exacts en remplaçant la méthode naïve d'élagage utilisée par Martins par celles de [KLP75], afin d'améliorer la complexité. Ensuite, nous proposons plusieurs versions approchées* Sector*,* SSector*,* QSSector*, valables en toute dimension.*

Pour le cas $d = 2$, nous proposons un algorithme d'approximation qui ne conserve que des chemins optimaux. Sa complexité est, dans le pire cas, du même ordre de grandeur que celle de l'algorithme exact de Hansen. Des expériences permettent d'observer un gain non négligeable, ce dernier devenant vraiment intéressant lorsque les ensembles de Pareto sont très gros, cas naturellement atteint lorsque le nombre d de dimensions augmente.

Theta-graphes

Dans le cas tricritère approché, notre algorithme Meta Rank peut naturellement s'appuyer sur une structure que l'on appelle un Theta-graphe afin de déterminer efficacement si des chemins peuvent être élagués.

Étant donné un ensemble de points dans le plan, un Theta-graphe consiste en un graphe dont les sommets sont ces points et qui résume bien les distances entre ces points. Pour un point donné, on découpe l'espace autour de lui en cônes d'angles tous égaux. Ce point est relié au point le plus proche dans chaque cône. Le cas qui nous intéresse correspond à ne prendre que six cônes. De plus, seul un cône sur deux (en alternant) nous intéresse : on appelle ce graphe un demi-Theta-6.

Beaucoup de structures similaires existent. Pour certaines d'entre elles, notamment les triangulations de Delaunay ou encore les diagrammes de Voronoï, il existe des algorithmes de maintenance dynamique. Il est possible de rajouter ou de supprimer un point en mettant à jour la structure efficacement. Il n'existe cependant pas de telle méthode pour les Theta-graphes, ni même pour les demi-Theta-6.

Contributions. *Nous proposons un algorithme efficace permettant l'insertion et la suppression d'un point dans un demi-Theta-*6*, et donc dans un Theta-*6 *aussi, tout en préservant sa structure. Ces deux algorithmes nous permettent de proposer* TSector*, une variante de* Meta Rank *tricritère. Ensuite, nous proposons un algorithme, qui à partir d'un demi-Theta-*6*, calcule les points les plus proches dans un cône donnée. Enfin, nous fournissons une étude détaillée d'un algorithme classique de calcul de dominant approché. Assemblé avec l'algorithme précédent, il nous permet de proposer* DSector*, une autre variante de* Meta Rank *tricritère.*

Abstract

This thesis focuses on the computation of approximate multicriteria shortest paths. In a multicriteria context, computing the Pareto sets, i.e. all the optimal solutions, is often prohibitive. Many approaches consist in computing only a subset. Some offer reasonable computation times but no guarantee about the representability, i.e. the distribution of the subset output among the whole Pareto set. Others methods guarantee a certain representability and interesting complexities but these are generally unpractical. Another issue is that, strictly speaking, both these approaches usually do not guarantee that the output is really a subset of the Pareto set, i.e. they might output non optimal paths.

First, we propose two optimizations of classical exact methods: MC Dijkstra 2D for the bicriteria case and Bucket for higher dimension cases. Then, we propose approximation algorithms with interesting theoretical guarantees. Several of those, Sector, SSector and QSSector, work in any dimension and their output sensitive complexity is interesting. The latter being incomparable to the Pareto set size, we propose a 2D optimization, Frame, which guarantees that the output is only constituted by optimal paths. We deduce that Frame's complexity is lower than or equal to the best known exact computation algorithm. In order to evaluate the pruning capability of Frame, we conduct an experimental study. This study shows that our algorithm is interesting when the Pareto set sizes are large.

In order to accelerate our approximation algorithms in 3D, we study Theta-graphs. We propose efficient algorithms for the dynamic maintenance of these graphs. Then, we study an oriented proximity query, using a Theta-graph to find nearest neighbors in a given direction. Finally, we detail how to apply our Theta-graph algorithms for tricriteria shortest path computation and provide two variants of Meta Rank called TSector and DSector.

Chapter 1

Introduction

Moving is a fundamental action. We naturally think of a person's journey, to get food, to see friends, or to go to work. The same dynamic can be seen for objects, to carry mail for example. Not only material, a displacement can operate on virtual data, it is the case of communications via Internet.

In a generic way, we have a network, i.e. a set of vertices, connected to each other by arcs. When walking in a city, the vertices would be the street intersections, and an arc would be the part of a street connecting two intersections. For the Internet, the vertices would correspond to the routers.

Organizing journeys in a network raises many problems, the most fundamental being to find a path through the network. More specifically, one can look for the existence of a path from one vertex to another (*reachability*). If we know that there is at least one, we may want the best one in a certain sense (*Shortest Path Problem*).

Other problems are strongly related to these. For example, one can consider that the network is dynamic [Cas18], or even that the variations are unknown. One must then take into account the uncertainty of the known data on the network [Dib+13]. A related problem is that of *congestion*: in a transportation network, the uncertainties about travel time are partly related to the fact that it depends on the number of users on the road. Thus, one may want to avoid that everyone uses the same road. One can then use a flow model [ZT87]. Finally, if we take the example of mail, its distribution leads to search for the fastest mailman's round. This problem, known as the *Traveling Salesman Problem* (TSP), consists in searching for a best Hamiltonian cycle, i.e. a best path that passes through all vertices exactly once, starting and ending at the same given vertex. This problem is particularly difficult since it is NP-hard [Boo75].

In this thesis, we focus on the Shortest Path Problem. The best path in a network can be the one that traverses the least number of arcs, the one that takes the least time, the one that costs the least, etc... How to find it?

1.1 Overview

Unicriterion setting. This problem has been studied extensively in the unicriterion setting. We give ourselves a graph whose arcs are weighted by real numbers. The weight of a path is the sum of the weights of its arcs. Among the paths having

minimal weights, we keep only one, no matter which one. The two best known solutions to compute shortest paths are the Dijkstra and Bellman-Ford algorithms. Unfortunately, these algorithms are not efficient enough for very large graphs.

In order to speed-up the computation of shortest paths, many precomputation methods have been developed. These methods first summarize information about a given graph. Then, using this information, an algorithm can quickly find a shortest path in this graph between any pair of vertices. To get an idea of the efficiency, some methods can find, in a few micro seconds, a shortest path between a couple of vertices in a graph with millions of vertices. The precomputation performed beforehand requires less than an hour, which is reasonable if one wishes to answer many shortest path queries afterwards.

Several generalizations are relevant. First, it is interesting to take into account the dynamism of the graph. In practice, we notice that the use of public transportation in a journey is restricted by the time. Models take into account this temporal dimension and many algorithms offer reasonable performances. However, the association of means of transport with constraints (train, bus) and means without constraint (i.e. walking) is not trivial and is still subject to many improvements.

Another generalization concerns the criteria multiplicity. In transportation networks, the criterion that one usually wishes to minimize is travel time. However, this is not the only criterion of interest, especially in the multimodal setting, which introduces many more criteria. One can think of the physical effort for cycling or walking, the CO_2 rejection for driving, or the number of connections in public transportation. Although this thesis is restricted to static graphs, multimodal graphs are on the one hand a motivation for the criteria multiplicity, but also a more general setting in which our algorithms can be integrated.

Multicriteria setting. We are interested in taking into account simultaneously these different criteria. Given d criteria, the weights are now d-dimensional vectors, each dimension corresponding to a criterion. Similarly to the unicriterion setting, if several paths share the same endpoints and the same weight vector, only one is kept. A path is optimal if no other path with the same endpoints is better on all criteria at the same time.

A fundamental difference with the unicriteria case is that optimal paths are no longer unique. Indeed, the order on the paths is not total since they can be incomparable. Taking a helicopter is much faster than walking, but it is also much more expensive. Other means of transportation, such as bus or taxi, provide intermediate tradeoffs. They are all incomparable for both time and financial cost. A set of paths representing all optimal paths is called a Pareto set. In Figure 1.1, there are two optimal paths. One has a smaller time travel while the other covers a shorter distance.

In the multicriteria setting, the Shortest Path Problem consists in finding the Pareto set from a given source to a destination. The computation of Pareto sets in this context was initiated by Hansen [Han80] with two criteria and generalized to any number of criteria by Martins [Mar84], both based on Dijkstra algorithm.

Similarly to the unicriterion case, many other methods were developed to speed up Pareto sets computation. However, the size of the Pareto sets can make this

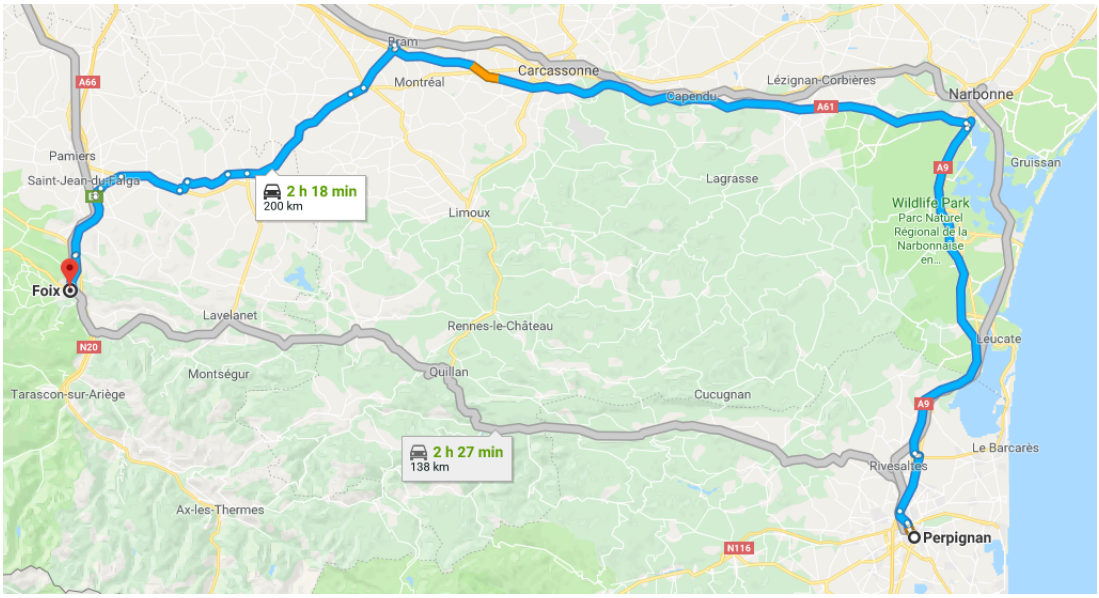


Figure 1.1 – Two bicriteria shortest paths (Google Maps).

computation unpractical. Indeed, simple small artificial graphs generate exponential size Pareto sets. In some cases, especially in 2D when the criteria are correlated, these sets remain of reasonable size. However, the increase of the number of criteria multiplies the number of optimal paths.

Approximation of Pareto sets. To overcome this issue, the general approach is to try to summarize these Pareto sets. To do this, a so-called approximated Pareto set is computed. This set is defined such that, for any optimal path, there exists a path in the approximated Pareto set which is, on each criterion, not worse than the weight of the optimal path multiplied by some constant. For example, if this constant is 2, an approximated Pareto set is such that any optimal path is at worst twice as good on each criteria as one of the paths in our approximated set. We say that the approximated set covers the Pareto set.

A significant advantage of this notion is that any Pareto set can be covered by a polynomial subset of paths. The latter is easily computable if the Pareto set is initially given. We can therefore extract from a Pareto set a representative set of paths of reasonable size. However, the main problem is to avoid computing the whole Pareto set. It is therefore necessary that any approximated Pareto set computation algorithm prunes optimal paths during its execution. The difficulty is to know which ones.

There are two main groups of solutions to this problem. One, more theoretical, guarantees that the output of the algorithm covers Pareto sets and has interesting complexities. About the complexity, the best known is that of Tsaggouris and Zaroliagis [TZ09]. However, the different methods of this group, and in particular the one of Tsaggouris and Zaroliagis, are unpractical since they prune too few optimal paths.

On the other hand, some methods propose a drastic pruning in order to obtain

much more reasonable performances in practice. The problem is that this gain is followed by an important loss: the output is not necessarily a coverage of Pareto sets. Moreover, complexities are rarely studied in these approaches.

The problem we address is to find a good tradeoff in order to obtain a coverage of the Pareto set, a reasonable complexity and good performance.

Theta-graphs. In the approximated tricriteria case, it is possible to use structures called Theta-graphs to efficiently determine whether paths can be pruned.

Given a set of points in the plane, a Theta-graph consists in connecting these points in a certain way. For a given point, the space around is partitioned into cones of equal angles. This point is connected to the nearest point in each cone. The case we are interested in corresponds to taking only six cones. Moreover, only one cone out of two (alternating) interests us: this graph is called a half-Theta-6.

Many similar structures exist. For some of them, such as Delaunay triangulations or Voronoi diagrams, there are dynamic maintenance algorithms. It is possible to add or remove a point by updating the structure efficiently. However, there is no such method for Theta-graphs, nor even for half-Theta-6. Given a solution, it would be possible to efficiently prune tricriteria shortest paths.

In the following, we present an extension of this overview.

1.2 Unicriterion shortest paths

1.2.1 Model

A graph consists of a set of entities, called vertices, and a set of relations between them, called arcs. For example, in a virtual social network, an account is a vertex and when two accounts are connected, then there is an arc between them. This relationship may be symmetrical: we are (a priori) friends of our friends. This is the case for networks such as Facebook. We talk about undirected graphs. But this is not always the case. Some relationships may correspond to information links, allowing to see the content shared by a followed account. We can think of Twitter or Instagram. We then say that the network is directed, that it is a digraph. The undirected case can be seen as a special case of the directed one where, for each arc, the reverse arc exists. This distinction is important, especially in the study of graph properties [Tro+21]. Of course, this distinction is not limited to social networks. In a transportation network, a road may be one-way or a bus trip may not take the same route in one direction as in the other. We will therefore consider, in this thesis, the general case of digraphs, that we will simply call graphs.

First, one may consider that the distance between two vertices is the minimum number of arcs used to go from one to the other. We will refer to it as the *u-distance*. However, this u-distance is often irrelevant. For example, let us consider that an arc corresponds to a mean of transportation. In Lyon, to go from the University of Lyon 1, in the north, to the Gerland neighborhood, in the south, you can take tramway line 1 without any connection, thus with a single arc. But it takes half the time to use this streetcar line only at the beginning, then the subway line B as

soon as possible, which consists in two arcs. The problem is that the arcs are not equivalent.

So we associate a weight to each arc. These weights can correspond to the time it takes to traverse the arc, to the physical distance, or to the price. Nevertheless, we consider for the moment only one of these criteria. The classical criteria take only positive values. However, negative weights can be found in some cases, such as the difference in altitude, or the battery charge of an electric car [Bau+20].

Formal definitions. More formally, a *unicriterion weighted directed graph* is a triplet $\mathcal{G} = (V, A, w)$, with V a set of n vertices and $A \subseteq V^2$ a set of m arcs[1]. The *weight function* $w : A \to \mathbb{R}_+$ associates to each arc a a non-negative weight $w(a)$ in $\mathbb{R}_+$. For an arc $a = (u, v)$, its *starting vertex* is u and its *endvertex* is v. A *path* is a sequence $(a_1, a_2, \ldots, a_k)$ of arcs such that the endvertex of each arc is the starting vertex of the following arc, i.e. for each i in $[\![1, k-1]\!]$, if $a_i = (v_i, v_{i+1})$ and $a_{i+1} = (v'_{i+1}, v'_{i+2})$, then $v_{i+1} = v'_{i+1}$. Its *starting vertex* is the one of a_1 and its *endvertex* is the one of a_k. A path composed of k arcs is a *k-hop path*. A *cycle* is a path such that its starting vertex is also its endvertex. The cost $c(P)$ of a k-hop path $P = (a_1, \ldots, a_k)$ is the sum $\sum_{1 \leq i \leq k} w(a_i)$.

Let s and t be two vertices. A *shortest path* from the source s to the destination t is a path such that no other path from s to t has a strictly smaller cost. The *distance* from s to t is the cost of a shortest path from s to t. If no such path exists, we say that the distance is infinite. Notice that the distance is unique but there might be several shortest paths, all of them having the same cost. In general, we are interested in finding a minimum distance path, no matter which one since we cannot differentiate their costs. It is sometimes interesting to distinguish paths by the arcs that compose them. However, this will not be done here.

If we allow negative weights, we must make sure that the graph does not contain any cycle of negative cost. Otherwise we might obtain paths whose cost is as small as we want, by taking a negative cycle enough times. If the graph does not contain any negative cycle, we notice that a shortest path is a simple path, i.e. it passes at most once through each vertex. Thus, between any pair of vertices, there exists a k-hop optimal path, with $k \leq n - 1$.

Problem 1. *For s, t two vertices, the* Unicriterion Shortest Path Problems *are:*

one-to-one: *to find a path P such that:*

$$c(P) = \min\{c(Q) | Q \text{ path from } s \text{ to } t\},$$

one-to-all: *to find, for every v in V a path P_v such that:*

$$c(P_v) = \min\{c(Q_v) | Q_v \text{ path from } s \text{ to } v\},$$

all-to-all: *to find, for every u, v in V, a path $P_{u,v}$ such that:*

$$c(P_{u,v}) = \min\{c(Q_{u,v}) | Q_{u,v} \text{ path from } u \text{ to } v\}.$$

[1] We restrict ourselves to simple graphs for legibility reasons. The generalization for multiarcs is straightforward.

1.2.2 Algorithms

Computing a unicriterion shortest path has been widely studied. We present the classical algorithms which solve the Problem 1.

Those presentations are often restricted to the distance computation, i.e. the costs of the shortest paths. It is usually easy to complete an algorithm to compute a corresponding shortest path. The general idea is that each distance is associated to the penultimate vertex of a shortest path, this vertex being used to find the antepenultimate one, etc...

All-to-all. Here, we want to compute a shortest path between any pair of vertices, implying a quadratic space use. Graphs are therefore generally represented by weight matrices.

The most classical solution is the Floyd-Warshall algorithm [Roy59]. It uses dynamic programming. If we have a list of n vertices $u_0, u_1, \ldots, u_{n-1}$ and a weight matrix $(w(u_i, u_j))_{\substack{0 \leq i < n \\ 0 \leq j < n}}$ containing the weights of the arcs between each pair of vertices, $+\infty$ if such an arc does not exist. Then from this matrix, we compute iteratively, for $k \in [\![0, n-1]\!]$, the distances between each pair of vertices passing only through intermediate vertices of indexes less than or equal to k. Its complexity is in $O(n^3)$.

Another method consists in using exponentiation by repeatedly squaring the weight matrix with min-sum matrix products (we replace the sums by minima and the products by sums). This allows to compute the distances between each pair of vertices at an at most 2^k u-distance after k iterations. Its complexity is $O(n^3 \log n)$.

One-to-all. If we restrict ourselves to a single source s, classical algorithms start from s and explore the graph with BFS variants. A BFS (Breadth-First-Search) starting from a vertex s consists in iteratively exploring the vertices at a u-distance k, for k from 1 to n.

The most popular algorithm is that of Dijkstra [Dij59]. The latter explores the vertices in the order of their distance from s, and considers them as processed once it knows their distance from s. At any time, there is a current distance d such that all vertices at distance less than d have been processed and all unprocessed vertices are further away. The next processed vertex is a neighbor of a processed vertex whose distance is minimal and which is not processed yet. Depending on the data structure used to choose the next vertex to process, we can obtain a complexity in $O(nm)$ with a naive search, in $O(m \log n)$ using a priority queue (Section A.3) implemented with a binary heap, or in $O(m+n \log n)$ if the priority queue is a Fibonacci heap. In order to guarantee the correctness of the algorithm, it is necessary to have non-negative weights.

Another classical solution is the Bellman-Ford algorithm [Bel58]. The latter consists in exploring the vertices in the same order as a BFS. However, once a vertex has been processed, the known distance can be improved: the algorithm can subsequently discover a path of lower cost but with a higher u-distance. Thus, to go from exploring u-distance k paths to $k+1$, Bellman-Ford algorithm must (re)consider all vertices with u-distance less than or equal to $k+1$. In the classical

description of the algorithm, it even considers all, setting the other vertices to an infinite distance. The rule for updating distances is based on dynamic programming. It computes shortest paths using at most $k+1$ arcs from those using at most k, extending those already known with any arc at the end. This algorithm is in $O(mn)$. A first advantage of this method is that it allows negative weights, without negative cycle. Another advantage is its simplicity, which can sometimes prove to be efficient in practice.

One-to-one. The shortest path computation is often used to go from one point to another. We want only one path, starting from a source vertex, and going to a destination vertex. Using Dijkstra's algorithm, we can limit the exploration of the graph to vertices with a distance less than or equal to the destination's one since the vertices are processed by increasing distance. However, without prior knowledge on the graph, we cannot avoid exploring all these vertices, being all potentially on a wanted shortest path.

An improvement consists in providing a heuristic function on the distance from any vertex to the destination. The A* algorithm [HNR68] is based on it. This algorithm can be seen as a generalization of Dijkstra. In this version, the next vertex to be processed is the one whose distance from the source, added to the heuristic of its distance to the destination, is minimal. If the heuristic is accurate, the exploration of the vertices is directed towards the destination.

The choice of the heuristic function is important. The latter must satisfy certain properties in order to ensure the algorithm's termination (positive) and correctness (smaller than the distance in the graph). The closer it is to the distance in the graph, the more targeted the exploration will be. This function depends on the context. For example, in the case of a geometric graph, i.e. when the vertices have coordinates, we can use the Euclidean distance. The latter is relevant in the case of a road network but can be problematic in the case of a labyrinth.

k-shortest path computation. A more general problem consists in computing a set of k paths between a same pair of vertices, such that no other path is shorter than one of them. A desirable complication of this problem is to prohibit cycles in the given paths. Yen proposes the algorithm with the best currently known time complexity, in $O(kn(m+n\log n))$ [Yen71]. Many heuristic improvements have been proposed afterwards. For instance, Al Zoobi et al. propose several interesting tradeoffs between the computation time and the memory used [AZCN21].

1.2.3 Speed-up

In order to accelerate the computation of a shortest path in the one-to-one case, a widely used method is to summarize information on the graph beforehand, to factorize exploration processing to obtain faster queries afterwards. The algorithm is therefore split into two steps.

1. A precomputation: we build a data structure containing information about the graph.

2. One-to-one queries, between different pairs of vertices, using the precomputation data structure to get hints.

Once the precomputation is done, it is possible to perform as many queries as wanted, using the same precomputation data structure. However, beware of the dynamism of the graph. If it changes, we may have to repeat the precomputation step. There are three main measures of performance:

- the precomputation time,
- the memory used by the data structure,
- the query time.

A very simple example consists in computing a shortest path between all pairs of vertices, then to store their costs in a matrix of size $n \times n$. Then, each query consists in reading the corresponding coefficient in the matrix, in constant time. Even using the Floyd-Warshall algorithm, both the precomputation time and the memory used by the matrix are very high, but the query time is optimal. This method can be interesting for small static graphs. On the other hand, as soon as the number of vertices is large, the precomputation becomes prohibitive.

On the opposite side, Dijkstra's algorithm has a null cost precomputation step (there is none) but its query time is prohibitive for large graphs, about a few seconds for a graph with tens of millions of vertices and arcs. Many methods have been developed to offer different tradeoffs. We present here briefly the different ideas on which they are based, referring mainly to the survey [Bas+16]. A more detailed presentation can be found in the latter.

Goal-directed methods. Some of them, similarly to A*, are based on an exploration from the source and directed towards the destination. A* itself can be seen as a precomputation method if the heuristic requires one.

For example, the ALT algorithm [GH05] uses landmark vertices. For each of these vertices, we precompute its distance to all the other vertices of the graph. We can deduce a heuristic function by triangular inequality on the distance. One needs to balance the number of landmark vertices and to choose them well to guarantee both a reasonable precomputation step and an accurate heuristic function to obtain efficient queries.

Another method, called Arc Flags [DGJ09], consists in partitioning the graph. Each arc is associated to a boolean which represents whether it is on a shortest path from its starting vertex to a vertex in each part. During the exploration of a query, we avoid extending paths with the arcs that are not on a shortest path going to any vertex in the part containing the destination.

Separators. Other methods are based on the use of separators. To start with a simple example, consider that we have two islands connected by a single bridge. We want to find a route from a source on the first island to a destination on the second one. It is sufficient to compute a path from the source to the bridge, then from the bridge to the destination.

More generally, we consider a set of vertices (vertex separators [SP02]) or arcs (arc separators [VV78]) partitioning the graph. We add shortcuts in the graph, by connecting each pair of separator vertices (or extremities of separator arcs) by an arc, with the distance in the initial graph as a weight. The precomputation difficulty consists in finding a relevant separator set, not too large but partitioning the graph in equal parts. To answer a query, we can explore the graph from the source to the separators bounding its part, perform the same with the destination, and then use the shortcuts to conclude.

Hierarchical methods. The use of shortcuts is also used in a hierarchical context (Contraction Hierarchies for road networks [Gei+12]). The basic idea is that during a journey in a road network, one tends to join roads that are more and more important while moving away from the source. And then one goes back on smaller and smaller roads while approaching the destination.

More formally, we sort the vertices in a chosen order, their position in this order defining a hierarchy on the vertices. The precomputation processes the vertices in increasing order, performing a contraction operation: for the current vertex v and each u, w neighbors of v, if v is on a unique shortest path from u to w, then the shortcut (u, w) is added to the graph and v is removed from the graph. The queries are bidirectional explorations (from the source and the destination), in increasing order positions. It only stops when the explorations are complete. Then, the algorithm concatenates paths from both explorations and outputs the shortest one. It is proven that such a path is a shortest path.

An improvement (Customizable Contraction Hierarchies [DSW14]) consists in performing two precomputations: one without weights, long and not to be repeated, and a second one with weights, faster. Thus, no need to start again the entire precomputation when the weights change.

Two-hop computation. A more extreme case consists in adding shortcuts so that only two of them are used in a given distance computation. This is the case for the PLL algorithm (Pruned Landmark Labeling [Aki+14]). A so-called *Hub Labeling* method consists in precomputing, for each vertex u, a set $\mathcal{L}(u)$ containing some vertices along with there distances to u. Those sets a computed so that, for each pair (s, t) of vertices, there exists $u \in \mathcal{L}(s) \cap \mathcal{L}(t)$ with u a vertex in a shortest path from s to t. A query consists in computing such a u minimizing $d(s, u)+d(u, t)$. Then this method is applied recursively to join the pairs (s, u) and (u, t).

Practice and theory. We presented review the general ideas used to speed-up one-to-one queries. Precomputation methods allow to speed up queries and offer interesting tradeoffs. For example, Akiba et al. tested their PLL algorithm on different unweighted graphs (computing the u-distance, which is the same as putting a weight of 1 on each arc). In this case, Dijkstra's algorithm reduces to performing a BFS. On an Internet subgraph of 1.7 million vertices and 11 million arcs, the algorithm requires six minutes of precomputation time and 2.7 GB of precomputation memory. The queries run in 2.3 μs while a BFS without precomputation takes 190ms.

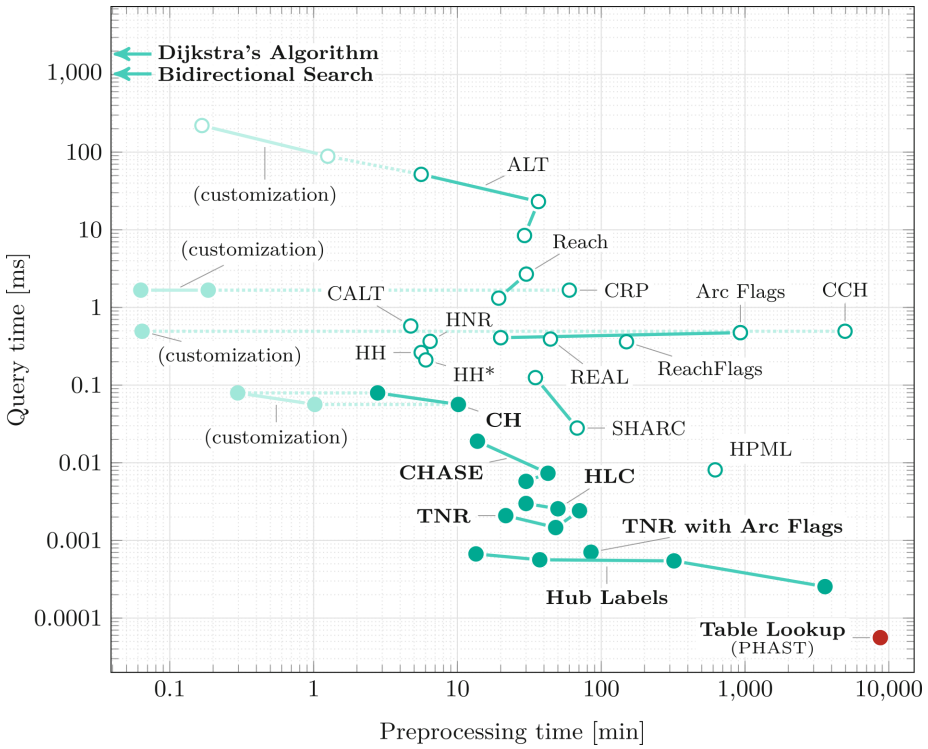


Figure 1.2 – Tradeoffs between the preprocessing time and the query time for different speed-up techniques [Bas+16].

A detailed experimental comparison has been conducted in [Bas+16]. The input graph represents the Western Europe road network, with 18.10^6 vertices and 42.10^6 arcs. The results are depicted in Figure 1.2. Some algorithms, such as the hub labeling ones, are not outperformed on both criteria by other algorithms. We say that they are Pareto optimal (defined latter in Section 1.3).

The performance of precomputation methods is often context dependent and these are heuristics. However, there are some theoretical results. Methods whose efficiency is guaranteed are proposed if the graphs verify certain properties. For instance, a metric on graphs called the *skeleton dimension*, can be defined as follow: *"the skeleton dimension is the maximum, taken over all nodes u of the graph and all radii $r > 0$, of the number of distinct nodes at distance r from u in the set of all shortest paths originating at u and having length at least $3r/2$."* [KV17]. In this thesis, Kosowski et al. present an almost-linear time algorithm to compute each label for a hub labeling algorithm. If k is the skeleton dimension and D the diameter of the graph, then the size of each label is in $O(k \log \log k \log D)$ time with high probability.

Another classical example concerns the hierarchical contraction method. If the tree-width of the graph with n vertices is k, then a query explores only $O(k.\log n)$ vertices [Bau+16].

1.2.4 Multimodal networks

Previously, the presentation was limited to static graphs. However, for some means of transportation, the duration of a journey depends on the departure time. One can think of the traffic flow of a road network, but it is critical while using public transports such as bus, trains or planes. The waiting time at a bus stop depends on the arrival time at that stop.

Temporal graphs. Several models exists. Two main ones consist either to consider that the travel time of an arc is a time-dependent function (*time-dependent model*), or that each departure time of a means of transport is a specific arc (*time-expanded model*). In the latter case, we have a finite number of arcs (u, v, t, λ), with u, v the two endpoints of the arc, t the departure time and λ its duration. The set of arcs is usually called the timetable.

In this context, the notion of minimum path is no longer unique. We list four classic kinds of temporal minimum paths:

- The *earliest-arrival path*, which arrives the earliest with a given departure time. This kind of path is relevant when you end a workday at a given time and you want to go home as soon as possible.
- The *latest-departure path*, which leaves the latest with a given arrival time. This case is interesting if you have an appointment at a given time.
- The *fastest path*, being the path having a minimum time duration within a given time interval. Here, imagine you have the whole day to travel, and you just want the trip to be as quick as possible.
- The *shortest path*, being the path having a minimum time duration in transportation (connection waiting time does not count). If during connection time, you might find things to do, and that you only want to spend as little time as possible in transports, then this one is for you.

Solutions. Some methods developed for static graphs can be adapted. For example, a shortest path is computable with Dijkstra algorithm [Wu+16]. In this article, Wu et al. presents that in order to compute earliest-arrival paths, one can use the particular structure of temporal graphs to simplify the Dijkstra algorithm by getting rid of the priority queue. In Dijkstra algorithm, the arcs exploration order corresponds to the chronological order of their departure. It is therefore interesting to have sorted the timetable beforehand. We can find a multicriteria adaptation (whose general case will be presented later) for the computation of the fastest path, the criteria being both the departure time and the arrival time. The last two algorithms have been developed at the same time by Dibbelt et al. [Dib+13] with a quite different formulation, under the name of CSA and pCSA.

A quite different algorithm, named RAPTOR, is based on dynamic programming [DPW14]. It computes the travel times from a source to any vertex of the graph using at most k means of transport, from the travel times using at most $k-1$ means of transport. It stops when no improvement is done. The solution for the

highest found k is the earliest-arrival time. This is in practice one of the most efficient algorithm without precomputation. An adaptation, called rRaptor, enable fastest paths computation instead of earliest-arrival ones.

Multimodal. So far, algorithms have been presented either for static graphs or for temporal graphs. A multimodal network combine both, time-constrained means of transport and unconstrained ones. A first difficulty is to model the union of these two types of networks in order to adapt the existing algorithms to this case [Paj09]. This kind of network is sometimes referred to as intermodal, the term multimodal simply referring to the combination of different means of transport, even if they are of the same type.

Algorithms for static graphs can be used as intermediate steps. During the exploration of a temporal graph, connections between vertices can be found in an underlying static graph. For instance, a 2-hop precomputation technique has been developed to be used with CSA and RAPTOR in a multimodal context [PV19].

Restrictions. When computing routes, and this is even more frequent in the multimodal setting, it is common to accelerate queries by introducing some restrictions. For instance, in a multimodal trip, walking parts are often limited to very short times, in order to go from a train station to a bus stop across the street. But walking for an hour may seem unreasonable.

However, Wagner et al. [WZ17] observe that, depending on whether or not long walks are allowed, shortest paths vary significantly. This variation is striking during off-peak hours, when public transports are scarce, but it is still significant during rush hour.

1.3 Multicriteria shortest paths

Although travel time is the most common criterion, it is not the only one that is interesting to minimize. In a transportation network for instance, one is often interested in finding a path minimizing several other criteria like the distance, the financial cost, or the physical effort.

The multiplicity of criteria is particularly important to take into account with the development of public transportation systems, in a multimodal context. For instance, the number of connections matters especially when time tables are uncertain. Even considering only time, the notion of minimum paths is not unique as presented in Section 1.2.4.

1.3.1 Optimal Paths

Although the multimodal context increases the number of criteria, we will restrict ourselves to static graphs, which already offer enough challenges. As mentioned in Section 1.2.4, our algorithms for static graph can be used as an intermediate step in a multimodal setting.

Formal definitions. The input of our problem is a *multicriteria weighted directed graph* $\mathcal{G} = (V, A, w)$ of n vertices and m arcs defined on d criteria, and a source vertex s. The graph may contain multiple arcs and loops. The *weight* $w(a)$ of an arc a is a d-dimensional vector in $\mathbb{R}_+^d$. The *cost* $c(P) = (P_1, \ldots, P_d)$ of a k-hop path $P = (a_1, \ldots, a_k)$ is the vector sum $\sum_{1 \leq i \leq k} w(a_i)$. The question is now to define what is a shortest path.

The main approaches to tackle multicriteria shortest path computation are the following.

Linear combination. For a given path of cost $(c_1, c_2, \ldots, c_d)$, the first natural approach consists in computing a linear combination of the costs, that is $\sum_{1 \leq i \leq d} \alpha_i c_i$, for some coefficients α_i. The advantage of this method is that, afterwards, any algorithm dedicated to unicriterion shortest path computation can be used. This approach has several drawbacks: *how to set up the α_i's? Does such a formula have a semantic meaning?*

Domination. We therefore prefer to keep all relevant paths. By relevant, we mean any path such that no other path is better on all criteria at the same time. More formally, we say that a path P *dominates* a path P' if $P_i \leq P'_i$ for every $i \in [\![1, d]\!]$. This dominance relation defines a partial order. If a path is dominated by another, it is useless to consider it since we try to minimize all the criteria at once. Thus, we define a *shortest path* as a path that is not dominated by any other path with a different cost and having the same extremities.

In this manuscript, we will only consider this dominance approach.

Constrained shortest paths. A related problem consists in optimizing a single criterion, while bounding the others. It is called the *Constrained Shortest Path Problem* (CSP). A generalization allows to optimize several criteria at the same time, keeping all the optimal paths according to the dominance relation on these criteria, while bounding the other criteria [Shi+17].

Optimal paths. The dominance enable to define what is an optimal path: it is a path which is not dominated by any other path having the same destination but a different weight. Our goal is to compute these optimal paths.

1.3.2 Pareto set

We define the notion of Pareto set, representing a set of optimal paths in a multicriteria setting.

A *Pareto set* of a set $\mathcal{T}$ of paths is a set of incomparable[2] paths from $\mathcal{T}$, that are not dominated by any other path from $\mathcal{T}$ with a different cost, and which is

[2] w.r.t. dominance

maximal by inclusion. In particular, if several paths of $\mathcal{T}$ have the same cost, then at most one is kept in a Pareto set of $\mathcal{T}$. Notice that if $\mathcal{S}$ is a Pareto set of some set $\mathcal{T}$, then the Pareto set of $\mathcal{S}$ is $\mathcal{S}$ itself.

Problem 2. *The* Exact Multicriteria Shortest Path Problem *consists in finding, for each vertex $v \in V$, a Pareto set $\mathcal{S}_v$ of the set of all paths from s to v.*

We use the notations $S_v = |\mathcal{S}_v|$ and $S = \sum_{v \in V} S_v$. The values S_v and S do not depend on the actual choices of the sets $\mathcal{S}_v$, since these values derive from the size of the unique Pareto set of the path costs.

1.4 Exact multicriteria shortest path computation

In order to solve Problem 2, the algorithms developed for dimension 1 are generalized to the multicriteria case. The main difficulty here is that several paths having the same destination may be of interest. We present the various existing approaches.

1.4.1 Classic algorithms

Most algorithms for multicriteria shortest path computations are based on Dijkstra and Bellman-Ford algorithms. An attempt to unify algorithms based on those two is addressed by Bökler et al. in [BC19]. This approach can also be found in [PS13]. The general idea is to maintain two sets of seen paths: those to be extended in $\mathcal{T}$ and those already extended in $\mathcal{S}$. Whenever the framework algorithm discover a path, it might be inserted into $\mathcal{T}$. In this case, it later removes the path from $\mathcal{T}$ and tries to insert it into $\mathcal{S}$. If successful, the algorithm tries to insert the path extensions into $\mathcal{T}$. Two classes of algorithms derive from this.

Multicriteria Label Setting (MLS). Here, *setting* refers to the fact that once a path is inserted into $\mathcal{S}$, it cannot be removed from it. It is guaranteed that such a path is optimal. Therefore, $\mathcal{S}$ is incremental. One can for instance use a lexicographically sorted priority queue to implement $\mathcal{T}$, guaranteeing that outgoing paths are necessarily better or incomparable to the following ones.

A version of MLS, called in this thesis MC Dijkstra, computes exact Pareto sets for two [Han80] or more dimensions [Mar84]. When there are no multiple arcs, it is proved in [BDH17] that MC Dijkstra in dimension d has a time complexity in $O(nS^2)$ and uses $O(nS)$ space, with S the size of the output. Although S can reach $\Theta(n(nC)^{d-1})$ when the weights are integers bounded by a constant C, it is very unlikely in practice to get such a size. In this context, the bicriteria algorithm proposed by Hansen operates in $O(mnC \log(nC))$ time. This is due to the fact that in this framework, it is much easier, with only two criteria, to check if a path is dominated by previously seen path.

Sometimes, MLS simply refers to Hansen's algorithm and the letters M and S may also stand for "Multiobjective" and "Scheme" respectively.

Multicriteria Label Correcting (MLC). The term *correcting* indicates that this time, a path inserted in $\mathcal{S}$ can potentially be improved later. We may therefore have to replace it. A classical implementation of $\mathcal{T}$ consists in using a queue, which causes to lose the guarantee provided by the priority queue example given for MLS. We may therefore have to process more paths. On the other hand, the simplicity of the queue induces a speed-up for the processing of $\mathcal{T}$. Different variants are proposed in [GM01].

The multicriteria generalization of the Bellman Ford unicriterion algorithm can be seen as a particular case of this category. We iterate n times the same step. After the k-th step, $\mathcal{S}$ contains a Pareto set of the k-hop paths. Some paths in $\mathcal{S}$ might be dominated by paths containing more arcs than k. The $(k+1)$-th step consists in extending all the paths from $\mathcal{S}$, inserting them in it and then keeping only the non-dominated paths. This gives us a Pareto set of the $(k+1)$-hop paths. Here, no need for a $\mathcal{T}$ structure to store some unprocessed extensions of $\mathcal{S}$ paths: all paths of $\mathcal{S}$ are extended.

Comparisons. To compare these two categories, Paixão et al. [PS13] propose an extensive experimental study with 27 MLS and MLC variants. They applied the algorithms on synthetic graphs of different topologies, ranging from less than a hundred vertices to 20000 and having 2 to 20 criteria. The authors observe that the MLC variant with a queue is often slightly better than the other methods.

Another experimental comparison restricted to the two-dimensional case is presented in [RE09] on grids and road networks up to 300K nodes. Here again, the MLC variant is usually more efficient and the gain can even be relatively important. But the latter is very variable and the MLS variant is sometimes more efficient.

1.4.2 Speed-up techniques

To speed up the one-to-one queries, it is not obvious that all the unicriterion speed-up techniques are adaptable to the multicriteria setting, and even then, they might not be efficient for multicriteria queries.

If we allow a light preprocessing, NAMOA* [MM12; MDLC10] is a generalization of the well-known A* search algorithm to multicriteria queries. For a given vertex, the heuristic function does not necessarily give a unique cost since real costs are not unique either in a multicriteria setting.

A classic preprocessing of the heuristic function consists in computing a reversed shortest path from the destination for each criterion to build $h(v)$ as a vector of d dimensions. This preprocessing requires $\Theta(md)$ steps and $\Theta(nd)$ memory space. Whenever the criteria are correlated, $h(v)$ can be quite close to an actual minimum cost.

NAMOA* also includes another interesting aspect of path computation, the multiobjective concept. We may want to find the nearest bakery around our home for example. In the multicriteria setting, this question becomes all the more interesting as several bakeries may interest us for different reasons. One may be more quickly accessible than the others. Another one, further away, may require less effort to walk to because there is no difference in altitude on the way. Given a set of targets,

NAMOA* computes a Pareto set on the union of the paths going to these targets.

The word *multiobjective* is also sometimes used to refer to the multicriteria setting, which is confusing. This term refers to the fact that a path cost is defined by applying objective functions on the weight of its arcs. In our case, the cost of a path is the sum of the weights of the arcs and the objective function is said to be linear.

Other unicriterion solutions have been generalized to the multicriteria case, such as a generalization of the SHARC algorithm, based on the arc flags technique [DW09]. Finally, for temporal graphs, RAPTOR algorithm is intrinsically multicriteria, with both the travel time and the number of connections as criteria. This algorithm has also been generalized (MCR) to enable the addition of any other criteria.

1.4.3 Limitations

In theory, a Pareto set can be prohibitively large. In [Han80], Hansen introduces a bicriteria example with constant degree for which a Pareto set is exponential in the number of vertices. As a consequence, the computation may take a lot of time and require a significant amount of space.

Even in practice, these algorithms are often not scalable: without any preprocessing, it takes a few seconds to answer a unicriterion query in a network of 18 millions vertices modeling western Europe [Bas+16]. Informally, even for a city like Prague with 65K nodes, for a given pair of source and destination, an exact Pareto set often contains thousands of paths for three criteria [Hrn+17] and its computation may take around 10 minutes. Since a query can require to store all the incomparable paths for one source, the amount of memory can be a thousand times larger than the storage of the graph itself.

It is interesting to observe that exact Pareto sets are not always large in practice, especially if the criteria are correlated. In [MHW06], Pareto sets sizes are often smaller than 100 for real graphs and synthetic graphs with a random weight assignment. However, when the number of criteria grows and some are negatively correlated, Pareto set sizes can be unpractical. Some examples can be found in [BFS19].

Conclusion. The existing solutions to Problem 2 are not practical. In order to speed-up these computations, there are two approaches: to use precomputation methods or to summarize Pareto sets. The latter is often used when the former is. We will focus on the second one.

1.5 Approximation

In order to obtain reasonable computation times when Pareto sets are large, only a sample of them may be kept. Another motivation is that in practice, a user does not want to be given a very large number of proposals. Some would even say that giving too many choices could decrease the relevance of the choice made. But the given Pareto set sample must be representative of the whole set. If our criteria are time and financial cost, we don't want to obtain only the few fastest (and therefore most

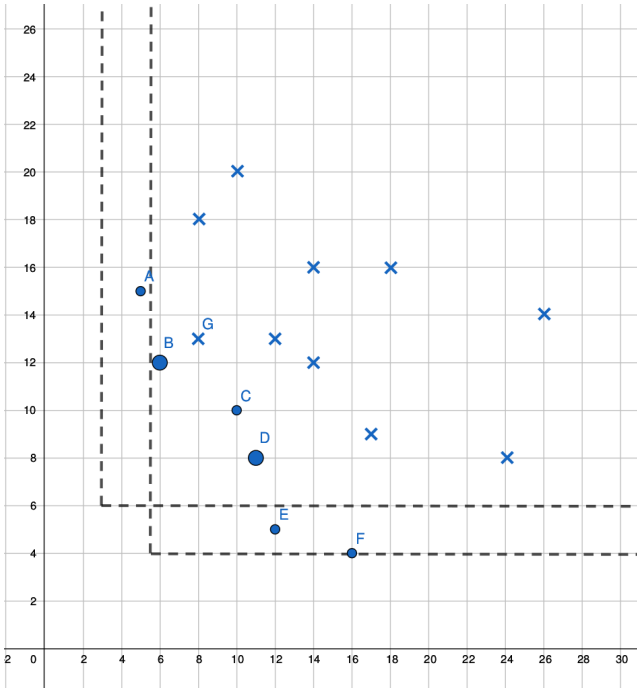


Figure 1.3 – $\{B, D\}$ is a 2-Pareto set

expensive) paths. Instead, we would like to have an idea of the different possible tradeoffs.

In this Section, we present different approaches to summarize Pareto sets.

1.5.1 Approximated Pareto set

To guarantee a certain representability, the notion of $(1+\varepsilon)$-*Pareto set* has been proposed. The idea is that for any path in a Pareto set, there exists a path in a $(1+\varepsilon)$-Pareto set which is, at worst, larger by a factor $(1+\varepsilon)$ on each criterion. For example, if $\varepsilon = 1$ and the criteria are time and financial cost, we want that for any optimal path, the algorithm returns a path at worst twice as slow and twice as expensive.

More formally, a path P $(1+\varepsilon)$-*covers* a path P' if $P_i \leq (1+\varepsilon)P'_i$ for every $i \in [\![1, d]\!]$. A $(1+\varepsilon)$-Pareto set of a set $\mathcal{T}$ is a set $\mathcal{S}_\varepsilon$ of incomparable paths from $\mathcal{T}$, such that any path of $\mathcal{T}$ is $(1+\varepsilon)$-covered by a path in $\mathcal{S}_\varepsilon$. In particular, a 1-Pareto set is a Pareto set and vice versa.

In Fig. 1.3, $\mathcal{S} = \{A, B, C, D, E, F\}$ is a Pareto set of all the paths, whereas $\{B, D\}$ is a 2-Pareto set. The two quadrants bounded by the dashed lines represent the areas 2-covered by B and D. Note that there may be various $(1+\varepsilon)$-Pareto sets. For example the set $\{G, D\}$ is also a 2-Pareto set even though $G \notin \mathcal{S}$.

In this manuscript, we will summarize Pareto sets with this approximation approach.

Problem 3. *The* $(1+\varepsilon)$-approximated Multicriteria Shortest Path Problem *consists in finding, for each vertex* $v \in V$*, a* $(1+\varepsilon)$*-Pareto set* $\mathcal{S}_{v,\varepsilon}$ *of the set of all paths from* s *to* v*.*

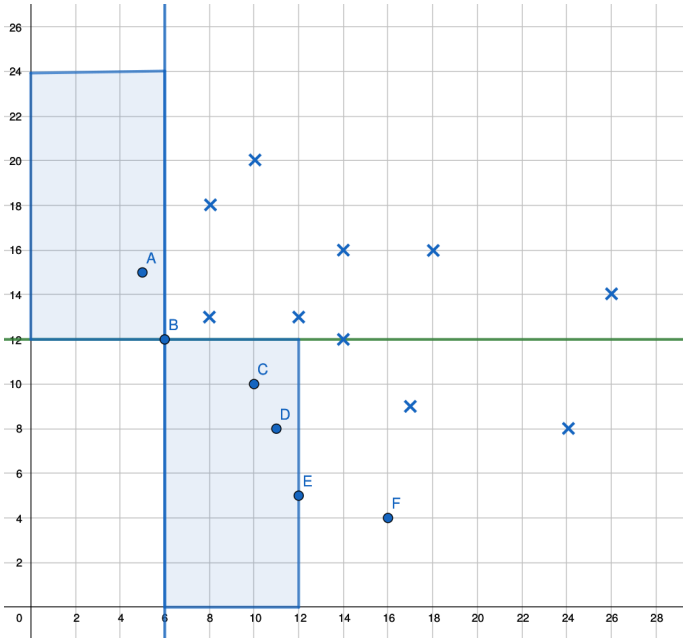


Figure 1.4 – Regions containing the incomparable paths 2-covering B

It was proved to always exist even with the constraint of having a polynomial size in n [PY00]. More precisely, in their paper, Papadimitriou and Yannakakis show that for any multiobjective optimization problem, there exists a $(1+\varepsilon)$-Pareto set $(\mathcal{S}_{v,\varepsilon})_{v\in V}$ of polynomial size in n, even if the aspect ratio is exponential in n. In our context, they show that if the paths costs are between 1 and a constant C on each dimension, $S_{v,\varepsilon}$ can be in $O\left(\left(\frac{\log(nC)}{\varepsilon}\right)^{d-1}\right)$. It means that the output can be quite small. But the existence of such a set does not guarantee its computability in a reasonable time.

1.5.2 Approximation algorithms

A simple subsequent algorithm can be used to get a constant approximation of S^*_ε, the minimum cardinality of a $(1+\varepsilon)$-Pareto set. But this does not prevent the prohibitive cost of computing exact Pareto sets. Therefore, we need to prune optimal paths during the computation. We present the classical algorithms that solve Problem 3.

Dijkstra-like algorithms (d=2). For $d=2$, Hansen [Han80] proposes a solution applying m times MC Dijkstra on the initial graph. At each iteration, an arc weight w is chosen and the algorithm rounds the weights of the arcs according to w. The time complexity is in $O\left(\frac{m^2n^2}{\varepsilon}\log\left(\frac{n^2}{\varepsilon}\right)\right)$.

Wang et al. develop in [Wan+16] a new algorithm called α-Dijkstra to compute approximated constrained shortest path for $d=2$. If weights are integers bounded by C, then α-Dijkstra time complexity is in $O(mnC\log(nC))$. This algorithm can

be adapted in order to compute $(1+\varepsilon)$-approximated Pareto sets. It prunes paths with a variable severity, depending on the number of best paths kept at a certain stage of the algorithm. It is designed to be used in COLA algorithm, which itself computes approximated constrained shortest paths.

Bellman-Ford-like algorithms. In the following, weights are between 1 and a constant C on each criterion. Warburton [War87] proposes an algorithm for any d, calling a Bellman-Ford-like algorithm several times. Each call prunes less severely than the previous one. Its complexity for $d = 2$ is in $O\left(\frac{n^3 \log n \log(nC)}{\varepsilon}\right)$ and for $d \geq 3$, it is in $O\left(n^3 \left(\frac{n^2 \log(nC)}{\varepsilon^2}\right)^{d-1}\right)$. This algorithm could require less MC Dijkstra iterations than Hansen's, but this number is still claimed in [BC19] to be too huge in order to be competitive in practice.

The best known complexity is obtained by Tsaggouris and Zaroliagis [TZ09], with another Bellman-Ford-based algorithm TZ operating in $O\left(nm\left(\frac{n \log(nC)}{\varepsilon}\right)^{d-1}\right)$ time. The cost space is partitioned so that two costs in the same part $(1+\varepsilon)^{1/n}$-covers each other. Only one path per part is kept. This choice of coverage is done so that n successive prunings keep a $(1+\varepsilon)$-coverage at the end.

Although interesting in theory, this solution has two major flaws in practice. The first one is that it does not prune all dominated paths, which would increase the complexity. The second one is that the very restrictive used coverage results in keeping all paths, the notion of part becoming superfluous.

Hybrid solutions. Inspired from TZ, Breugem *et al.* [BDH17] proposed a Dijkstra-based algorithm with TZ pruning technique. This algorithm is called Hydrid. It runs in $O\left(n^3 \left(\frac{n \log(nC)}{\varepsilon}\right)^{2d-2}\right)$ time. The authors have made an experimental comparison with Hydrid and TZ modified in order to prune dominated paths. For large graphs, they use a $(1+\varepsilon)^r$ coverage with a custom r in order to be practical. However, with this modification, the output is no longer guaranteed to be a $(1+\varepsilon)$-approximation. A comparison is made between these two approximated Pareto sets computations and the standard MC Dijkstra. The new Hydrid algorithm is efficient and sometimes outperforms MC Dijkstra whenever Pareto sets are very large. It is also interesting to notice that TZ does not prune a lot of explored paths. It means that it can be much worse than MC Dijkstra for small Pareto sets.

Bökler et al. conduct an experimental study, with TZ, Hydrid and some variants, in order to compare these different solutions [BC19].

1.5.3 Alternative Pareto set Summaries

For large and real graphs, the computation time of guaranteed approximated Pareto set can be too long.

Some heuristics have been proposed and speed up drastically the computation time, but without any guarantee [Hrn+17]. The main idea is to combine several filtering and rounding techniques in order to prune explored paths. Unfortunately, there is no guarantee that the outputs of these algorithms are $(1+\varepsilon)$-Pareto set for a given ε. On the city of Prague with three criteria, the time of the fastest heuristics can divide the MC DIJKSTRA time by a factor of a few thousands. NAMOA* is in-between but provides a guarantee since it computes exact Pareto sets. Using two measures of quality on the output paths, the authors show in their experiments that solution sets of the heuristics are quite close to Pareto sets.

Bast et al. propose the TNT filtering method, combining several heuristic methods in order to prune some optimal paths in a multimodal setting [BBS13]. These include both rounding and thresholding of weights. Moreover, combinations of means of transport that seem absurd are pruned: the use of a private car between two means of public transport is unlikely.

Other attempts have been done to summarize Pareto sets [BFS19; SJS15]. A *linear path skyline*, defined as a subset of conventional Pareto sets, is a set of paths optimal under a linear combination of their cost values. Multicriteria being especially relevant in a multimodal setting, a different approximation definition has been proposed in [DDP19]. In this paper, a Pareto set is summarized by the paths such that their projections on two specific criteria (arrival time and number of trips) are additively not far from an optimal one. On the Munich Open StreetMap with 221K nodes, the average number of paths is divided by 5 with respect to exact Pareto sets.

Conclusion. To summarize, some existing solutions have nice complexities. But in order to make them practical, the known modifications no longer guarantee to outputs $(1+\varepsilon)$-Pareto sets. Other approaches directly developed with the intention of being practical have the same drawback. Our work aims at providing a tradeoff between the complexity and the practicability, while guaranteeing to output $(1+\varepsilon)$-Pareto sets.

1.6 Geometric graphs

In the tricriteria case, our solutions consider sets of paths of the same rank[3], thus in a same plane. In order to summarize these sets of paths efficiently, the use of geometric planar graphs, and more specifically those called Theta-graphs, is natural.

In this Section, we present these graphs and the related notions. An important parameter is the notion of distance considered. Depending on the distance definition, results, algorithms and proofs can differ significantly.

1.6.1 Definitions

Let us consider a set of points V in the plane. Many structures can be defined on V. We present the most classical ones.

[3]The rank of a path is the sum of its cost criteria

Voronoi diagrams and Delaunay triangulations. One of the most interesting structure is the Voronoi diagram of V. The plane is partitioned into cells, each one containing a single point $u \in V$ and the set of points in the plane to which u is the nearest point in V. Its dual structure is called the Delaunay triangulation of V. The pairs of points from V, whose Voronoi cells have one side in common, are connected by an arc. Except for the degenerate cases, it is a triangulation of V as its name implies. Formal definitions and study of fundamental properties of these structures can be found in [TOG17].

Depending on the distance considered in the definition of a Voronoi diagram, we obtain different properties on the associated Delaunay triangulation. In the case of the Euclidean distance, the Delaunay triangulation (noted L_2 Delaunay triangulation) is the triangulation of the plane such that the interior of the circumscribed circle of any triangle contains no points of V. The notion of Voronoi diagram is generalized to any convex distance. We can find a study of some properties in [AP14].

Theta graphs. Another structure consists in building a graph over V. Each point of V is connected to its nearest neighbors around it, one by cone of angle θ. More precisely, for $u \in V$, we partition the plane in k cones of same angle $\theta = 2\pi/k$ around u, for $k \geq 2$. Then, for each cone of u, we add the arc (u, v) with v a nearest point from u in the cone of u containing v. The obtained graph depends on the distance considered. If we use the Euclidean distance, the graph is called a Yao-graph.

Another interesting distance is the triangular distance[4]. For any point $v \in V$, the *triangular distance* from u to v is the Euclidean distance from u to the orthogonal projection of v on the bisector of the cone of u containing v. The associated graph is called a Theta-graph[5] [Cla87] and is denoted $\Theta_k(V)$ or Theta-k-graph of V. If V contains n points, $\Theta_k(V)$ is computable in time $O(n \log n)$ [TOG17].

On Figure 1.5, we have represented a partition in 6 cones around u. We can see the orthogonal projection of the points around u on their associated bisector. The point u is connected to the points a, b, c, d, e and f, each one being the point such that the orthogonal projection is the nearest from u in its cone. We have thus represented exactly the arcs coming out of u in the Θ_6 of the represented set of points. Notice that b is not the nearest from u in Euclidean distance in its cone. Therefore, the Yao-graph is different from the Theta-graph for this set of points.

Half-Theta-graphs. If the number of cones is even, we say that one cone out of two is positive, the other is negative, the disposition around the considered point being alternated. A half-Theta-graph, noted Θ_6^+, is a Theta-graph in which only the outgoing arcs in the positive cones are kept. On the Figure 1.5, we consider that the positive cones are those with an arrow at the end of their bisector. The corresponding arcs are depicted with a continuous green line. A surprising result is that the half-Theta-6 corresponds exactly to the Delaunay triangulation for the triangular distance (noted TD Delaunay triangulation) [Bon+10].

[4]Strictly speaking, this is not a distance since it is not symmetrical.

[5]Sometimes, it is referred to as a triangular distance Yao-graph.

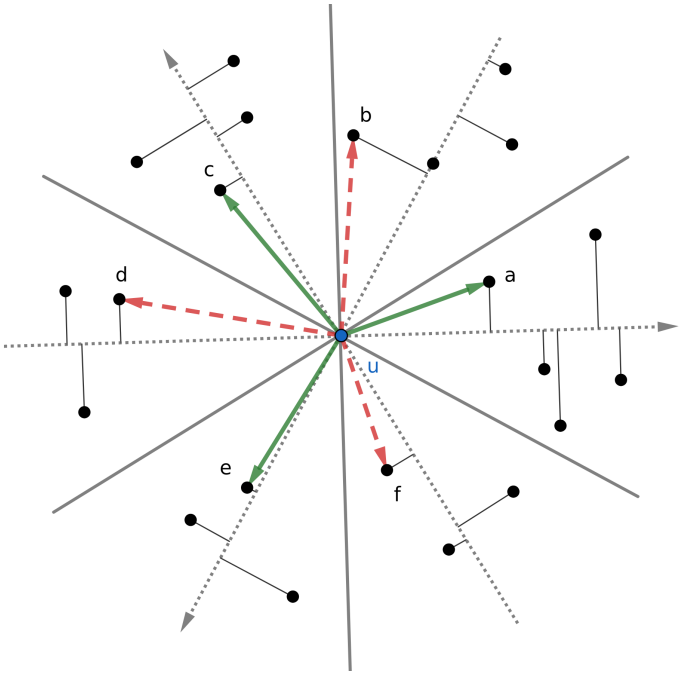


Figure 1.5 – Representation of the cones of u and the arc in each of them for $k = 6$.

Higher dimensions. These notions can be generalized to any number of dimension with a certain definition of the cones. If we give ourselves a set V of n points in $\mathbb{R}^d$, we can compute the $\Theta_k(V)$ in time $O(n \log(n)^{d-1})$. For dimension 2, and then for higher dimensions, one can find a detailed presentation of the definitions and the construction algorithms in [NS07].

1.6.2 Spanners

In order to illustrate the relevance of these structures but also the variety of properties obtained depending on the structure considered, we present a quick overview about spanners.

Let $V \subset \mathbb{R}^2$ be a set of n points. The *geometric complete graph* of V is the graph $\mathcal{G} = (V, A)$ such that A consists of the edges connecting all pairs of vertices. These edges are weighted by the distance between its ends. Depending on the chosen distance, different results are obtained. Each edge is represented by a segment between its extremities.

Spanners. A *t-spanner* of $\mathcal{G}$ is a subgraph $\mathcal{G}'$ of $\mathcal{G}$ such that any pair of vertices from V is connected by a path in $\mathcal{G}'$ whose cost is at most t times the distance between the two vertices in $\mathcal{G}$. The factor t is usually called either the *stretch factor* or the *spanning ratio*. This definition can be extended to any graph but is mainly studied for geometric complete graphs.

When designing a spanner, minimizing t is not the only objective, one can also aim at minimizing the number of edges or the fault tolerance. Some of the results mentioned below are presented in a more complete way in [BS13].

Planar spanners. Many works focus on the case of planar spanners, i.e. when no edge crosses another. The results differ depending on the distance considered:

- L_1 and L_∞ Delaunay triangulations are $\sqrt{4+2\sqrt{2}}$-spanners [Bon+12]. This is the best possible stretch factor in that case.
- L_2 Delaunay triangulations are 1.998-spanners [Xia11].
- TD Delaunay triangulations are 2-spanners [PC89]. Since these triangulations corresponds exactly to half-Theta-6 graphs, the latter are also 2-spanners. Here again, the stretch factor is the best possible.

Theta-graphs and spanners. Although Theta-graphs are not planar in general, they enable to improve the stretch factor and to bound the outgoing degree: it is the number of cones considered [NS07].

If $k \geq 9$, then $\Theta_k(V)$ is a t-spanner of V with $t = 1/(\cos\theta - \sin\theta)$ and $\theta = 2\pi/k$. From another perspective, if $t > 1$ is fixed, there exists a Theta-graph with a linear number of arc and a logarithmic diameter, both in the given number of points.

In some cases, it is easy to find a path that approximate by a factor at most t the Euclidean distance. For Theta-graphs with at least 9 cones, a simple greedy algorithm enable to find such paths. For all points $s, t \in V$, if we start from s and recursively go to the neighbor of the current vertex in the cone containing t, then, not only do we arrive eventually at t, but taking a path whose cost is at most t times the Euclidean distance between s and t.

Other spanners. The distance notion can be generalized to convex and compact shapes containing the origin in its interior. Delaunay triangulation based on these "distances" are spanners whose stretch factors depend only on the convex shape. Precise stretch factors are given in [Bos+08].

1.6.3 Dynamic Euclidean Delaunay Triangulation

We have seen that the results can change depending on the distance considered. A particularly interesting example for multicriteria shortest path computation is the dynamic maintenance of Delaunay triangulation. Given a Delaunay triangulation, how to manage the insertion or the removal of a point without recomputing everything? We present the existing works on L_2 Delaunay triangulations.

Insertion of a point. In the case of L_2 Delaunay triangulations, an algorithm for inserting a point is introduced by Guibas et al. [GS83]. A more detailed version can be found in [GKS92]. Using this algorithm, the computation of the Delaunay triangulation of a set of n points can be done incrementally, i.e. by inserting the points one by one. Guibas et al. proved that such an incremental algorithm is in $O(n \log n)$ expected time if, for a given set of points, the latter are taken in a random order. This algorithm is also presented in [KMS91] with a different formalism.

Deletion of a point. Kao et al. propose also a deletion algorithm in a L_2 Delaunay triangulation. When a point is deleted, its neighbors form an empty convex polygon. If this polygon is made of k points, we can triangulate it in linear time [Agg+89]. But this method being too expensive in practice, the authors propose an algorithm whose average time complexity is in $O(k \log k)$. The average degree being constant, they show that their algorithm is on average in $O(\log n)$. The steps of their algorithm are essentially those of the insertion in reverse order.

They also propose a solution whose mean complexity amortized over a long series of insertions and deletions is in $O(\log n)$. Meanwhile, Devilliers et al. propose a structure enabling a deletion in $O(\log \log n)$ on average, while an insertion is in $O(\log n)$ [DMT92].

Kinetic. Another form of dynamism consists in the motion of the points, according to any trajectory. For example, an algorithm is proposed to maintain, in the plane, the Voronoi diagram structure [Alb+98]. At each change, the complexity is in $O(\log n)$, which is optimal. Rahmati et al [Rah+15; Rah+19] defined structures for both Yao and half-Yao graphs to allow efficient updates. Finally, there are models combining both types of dynamism, insertion/deletion and motion for the 3D case [SMH04]. The proposed solutions are efficient in practice, according to the experimental studies conducted by their authors.

Triangular distance. To the best of our knowledge, no efficient algorithm has been proposed yet for the triangular distance. The adaptation of algorithms for Euclidean distance is not trivial. One of our contribution in Chapter 6 is to propose algorithms for TD Delaunay triangulations.

1.6.4 Proximity problems

The previously presented structures are strongly related to proximity queries. We give ourselves a set V of n points in $\mathbb{R}^d$. For a given point $u \in V$, we try to find the nearest point $v \in V \backslash \{u\}$ in a certain sense.

Usually, a precomputation on V is done, outputting a structure $\mathcal{S}$. Then, proximity queries on V can be efficiently answered using $\mathcal{S}$. For instance, we can be interested in finding the nearest point in a cone. In triangular distance, a precomputation giving the Theta-graph of V allows queries in constant time. The same is true for the Euclidean distance with Yao-graphs.

The definition of nearest can be relaxed by a factor $(1 + \varepsilon)$: one might want to find a point at a distance at most $(1 + \varepsilon)$ further than the nearest point, both in a specific cone. For the Euclidean distance, Funke et al. propose to precompute an approximate Voronoi diagram [Fun+15]. With $\alpha = 1/\varepsilon$, the precomputation step is in $O\left(n\alpha^d \log(n\alpha) \log \alpha\right)$ time. The output diagram is in $O\left(n\alpha^d \log \alpha\right)$ space, and a request is in $O\left(\log(n\alpha)\right)$ time.

Nearest Neighbors. The same question arises without cones. The *Nearest Neighbor Problem* consists in finding, for a point in V, its nearest neighbor in $V \backslash \{u\}$, according to a certain distance.

Again, with a Theta-graph, one can consider all cones at once. Delaunay triangulations and Voronoi diagrams may also be used. For instance, if this time the query is made on a point which is not in V, the Voronoi cell which contains it can be found in $O(\log n)$ [Kir83], giving the nearest neighbor. Other classical algorithms are presented in [Cla06].

One can also relax the definition of a nearest neighbor to a factor $(1+\varepsilon)$: the problem is to find a point at a distance at most $(1+\varepsilon)$ further than the nearest point. In the Euclidean distance case, one method consists in covering the space of points into balls and guarantees, for n large enough, a time complexity in $O(dn^{\sigma})$ with $\sigma = 1/(1+\varepsilon)^2$ [AI06]. An overview on generalizations to other distances can be found in [Ind04].

If the set V is dynamic, i.e. if points can be inserted into V or deleted from it, then dynamic Voronoi diagrams can be used to perform the queries [Kap+20]. The update of the structure, as well as the query time, is a polylog, while the space usage is in $O(n(\log n)^3)$.

k-Nearest Neighbors. Not being limited to a single nearest neighbor, one might want to find the k nearest neighbors. The structures presented by Rahmati et al. also deal with this query.

Some algorithms answering this query have an efficiency depending on the distribution of the points in the space. For example, Clarkson proposes an algorithm in $O(n \log \mathcal{A}_R)$ with $\mathcal{A}_R$ the aspect ratio on the distance between pairs of points in V [Cla83].

For any metric on $\mathbb{R}^d$, another algorithm using Delaunay triangulations runs in $O(k(n+\log n))$ time [DE96]. Related problems addressed in this paper are those of finding the k closest pairs of points or all pairs at a distance less than a given threshold.

Conclusion. The geometric graphs defined in this Section are therefore objects that have been extensively studied. However, we have seen on the one hand that the results can vary according to the considered distance. On the other hand, some problems on TD Delaunay triangulation, such as the dynamic maintenance, remain open. In Part II, we propose algorithms on this subject and use some of them for multicriteria shortest path computation. We detail the precise contributions in the following Section.

1.7 Contributions

1.7.1 Multicriteria shortest path computation

Chapters 3 and 4 are dedicated to multicriteria shortest path computation. We first provide a framework: Meta Rank algorithm (Section 3.2). It uses a `Sample` function. Depending on this `Sample` function, Meta Rank can compute exact or approximated Pareto sets.

Exact computation. We propose several `Sample` functions to apply in Meta Rank in order to compute exact Pareto sets $\mathcal{S} = \bigcup_{v \in V} \mathcal{S}_v$. The provided algorithms are:

- MC Dijkstra `2D` (Section 4.1), an optimized version of MC Dijkstra. We detail the data structures used in order to refine the complexity.
- Bucket (Section 3.3), an optimized version of MC Dijkstra in general dimension. This algorithm is based on the use of efficient methods to remove dominated paths from a set (see Section 2.2). As the name suggests, paths are processed in buckets.
- Dijkstra Post (Section 3.3), a variant of MC Dijkstra introduced for experimental analysis. It corresponds to the adaptation of MC Dijkstra in Meta Rank's framework.

Approximated Pareto set. Then we propose several `Sample` functions for Meta Rank in order to compute guaranteed $(1+\varepsilon)$-Pareto sets $\mathcal{S}_\varepsilon = \bigcup_{v \in V} \mathcal{S}_{v,\varepsilon}$. The provided algorithms are:

- Sector (Section 3.5.2), a first $(1+\varepsilon)$-approximation algorithm, based on the framing of paths for pruning,
- SSector (Section 3.5.3), a faster version of Sector, using range queries (see Section 2.2.4),
- QSSector (Section 3.5.4), a heuristic improvement of SSector for cases where many paths have common values on some criteria[6],
- TSector (Section 6.4), a variant of Sector based on Theta-graphs,
- DSector (Section 7.3), a variant of Sector based on Theta-graphs and dominating sets,
- Frame (Section 4.2.3), a variant of Sector, optimized in dimension 2, with a stronger framing of the pruned paths.

Complexities. In Table 1.1, we present the computation times for one-to-all queries. Those are expressed in the output sensitive complexity, i.e. depending on the output size. Δ is the maximum degree and Λ is the number of non-empty ranks, defined in Section 2.1. Whenever the weights are integers bounded by a constant C, $\Lambda \leq dnC$.

For an exact computation, S is the output size, and thus the Pareto set size. For approximation algorithms, S_ε denotes the size of the output, which is a $(1+\varepsilon)$-Pareto set. It might be much larger than S^*_ε, the minimum cardinality of a $(1+\varepsilon)$-Pareto set. However, starting from $\mathcal{S}_\varepsilon$, a linear time algorithm (Algorithm 3) can output $\mathcal{S}'_\varepsilon \subseteq \mathcal{S}_\varepsilon$ such that $\mathcal{S}'_\varepsilon = O(S^*_\varepsilon)$.

[6]For example, consider trips by car and on foot with three criteria: time, physical effort and price. All car journeys will have zero physical effort, while walking will be free of charge.

	Output sensitive complexity $O(\cdot)$	Pareto compatible	Ref.
MC DIJKSTRA	ΔS^2	✓	[Han80; BDH17]
BUCKET ($d > 2$)	$S \cdot \min\left\{(\Delta+\Lambda)\cdot\log^{d-2}(S), \Delta S\right\}$ $+\Lambda\log(\Lambda)$	✓	Theorem 19
SSECTOR	$\Delta S_\varepsilon \log^{d-1}(\Delta S_\varepsilon)$		Theorem 32
TSECTOR	$(\Delta S_\varepsilon)^2 \log(\Delta S_\varepsilon)\log\log(\Delta S_\varepsilon)$		Theorem 80
DSECTOR	$(\Delta S_\varepsilon)^4$		Theorem 90
TZ	$n\Delta S_\varepsilon$		[TZ09]
HYDRID	?		[BDH17]
MC DIJKSTRA ($d = 2, 3$)	$\Delta S \log(\Delta S)$	✓	Proposition 13
FRAME ($d = 2$)	$\Delta S_\varepsilon \log(\Delta S_\varepsilon)$	✓	Theorem 45
HYDRID ($d = 2$)	$nS_\varepsilon^2 \leq n^2 S^3$		[BDH17]

Table 1.1 – Output-sensitive complexities for shortest path computation.

Pareto compatibility. We introduce the notion of Pareto compatibility in Section 4.2.2. An algorithm is said to be *Pareto compatible* if and only if its solution $(\mathcal{S}_{v,\varepsilon})_{v\in V}$ to the $(1+\varepsilon)$-approximated Multicriteria Shortest Path Problem is always a subset of a Pareto set $\mathcal{S}_v$, for every vertex v. This property is useful since it guarantees that the size S_ε of the output of an approximation algorithm is always at most S, the size of a Pareto set.

We prove that FRAME is Pareto compatible. Thus, we have $S_\varepsilon \leq S$ for that algorithm and we can hope that its computation time is in practice significantly smaller than the one of MC DIJKSTRA algorithm in 2D. In Section 4.3, we conduct an extensive experimental study on different kind of graphs that shows an interesting gain whenever S is large.

Comparisons. Hybrid [BDH17] and TZ [TZ09] are not Pareto compatible. However, for $d = 2$, $S_\varepsilon(\text{HYDRID}) \leq nS$. More generally, for $d \geq 3$, it is a priori impossible to claim which one is the smallest output among these different algorithm output sizes: $\mathcal{S}_\varepsilon$(SSECTOR), $\mathcal{S}_\varepsilon$(HYDRID), $\mathcal{S}_\varepsilon$(TZ) or $\mathcal{S}$(MC DIJKSTRA).

If the arc weights are integers, then we prove that the output size S_ε of SSECTOR is in $O\left((nC)^{d-1}\log_{1+\varepsilon}(nC)\right)$. We can observe for certain parameters (C moderate, Δ constant and ε small), SSECTOR provides smaller upper bounds on the time complexity than TZ.

1.7.2 Half-Theta-6-graphs

Our contributions on geometric graphs concern half-Theta-6 graphs and are presented in Part II. We start by defining a projection of the 2D points in 3D, in order to greatly simplify the study of these objects (Section 5.1.2).

Dynamic Θ_6^+. In Chapter 6, we propose the first efficient algorithms enabling the insertion and the removal of a point in a Θ_6^+, in order to provide dynamism to this object. The insertion algorithm surprisingly does not use any triangular distance test, as the Guibas algorithm does with the Euclidean distance. It is sufficient to ensure that the graph remains planar and that each vertex has exactly one outgoing vertex in each cone.

Nearest Neighbors. We also propose an algorithm computing nearest neighbors in a cone (Section 7.1). More precisely, we give ourselves a set V of points and $\Theta_6^+(V)$. For a point $u \in V$ and a cone $\mathcal{C}$ of u in $\Theta_6^+(V)$, we compute the at most k nearest neighbors at a distance at most $(1+\varepsilon)$ from u in $\mathcal{C}$. This algorithm explores the points of $\mathcal{C}$ in increasing triangular distance. The particular structure of $\Theta_6^+(V)$ allows to go from points to points by limiting to the points in $\mathcal{C}$.

In Section 7.2, we present a classical algorithm computing approximated dominating set for undirected graphs. We provide detailed data structures in order to study the complexity.

Applications to Sample Sector. We adapt the Θ_6^+ dynamic maintenance algorithms to the approximated shortest path computation in Section 6.4. One step of Sample Sector, the `Sample` function of Sector, consists in determining if a path is $(1+\varepsilon)$-covered in a given cone. The use of a Θ_6^+ allows to answer this question in constant time, which divides the overall complexity of Sample Sector by a n factor (without taking into account the construction of the half-Theta-6). Meta Rank using this method as a `Sample` function is called TSector.

Finally, we propose a `Sample` function combining our algorithm computing the nearest neighbors in a cone and the one computing approximated dominating set (Section 7.3). The obtained subset of points is potentially much smaller than the one obtained with the method from Section 6.4. We can thus hope to prune much more in Meta Rank.

Complexities. We summarize the complexities of our algorithms in Table 1.2. Input graphs contains n vertices. For the insertion or the deletion of a point u, Δ_u is the degree of u in the Θ_6^+ of the set of points containing u. The complexity given for the insertion does not take into account the point location request in $O(\log n(\log\log n)^2)$, nor that of the update of the associated structure in $O(\Delta_u \log n \log\log n)$.

For the approximated dominating set computation, Δ_+ (resp. Δ_-) is the maximum outgoing (resp. ingoing) degree.

	Complexity $O(\cdot)$	Ref.
Insertion	Δ_u	Theorem 74
Deletion	$\Delta_u \log \Delta_u$	Theorem 77
k-Nearest Neighbors	$k \log n$	Theorem 82
Approximated dominating set	$n\Delta_+\Delta_-$	Theorem 86

Table 1.2 – Complexities for geometric graph algorithms.

Part I

Multicriteria shortest paths

Chapter 2

Preliminaries for shortest path computation

In this chapter, we formally present the notions necessary for a good understanding of both Chapters 3 and 4. First, we introduce the general notations for graph manipulation in Section 2.1. A summary of the notations useful in the Chapters 2, 3 and 4 is given (Table 2.1). Then, we detail the notion of vector Pareto sets and how to compute them (Section 2.2). These notions are adapted to the Pareto sets of paths in Section 2.3. Finally, a description of Dijkstra's algorithm is given in Section 2.4 to better understand its multicriteria generalization.

2.1 Notations

Let $\mathcal{G} = (V, A, w)$ be a *weighted directed graph* defined on $d \in \mathbb{N}$ criteria :

- V is the set of n vertices,
- an *arc* a is a pair of vertices $(u, v) \in V^2$, u being the source vertex of a and v its destination vertex,
- A is the set of m arcs,
- $w : A \to (\{0\} \cup [1, C])^d$ a weight function, with $C > 1$ a constant.

Remark. *It is possible that a graph contains multi-arcs and loops. In a multicriteria setting, it is relevant to allow the existence of several arcs having the same source and the same destination, but whose weights are not* better *than each other. The meaning of* better *will be defined in the Section 2.2. However, for legibility reason, we consider in this manuscript that the graph is simple. Generalizing the following algorithms to non-simple graphs is straightforward.*

A *path* is a sequence $(a_i)_{1 \leq i \leq k}$ for $k \in \mathbb{N}$, such that for all $i \in [\![0, k-1]\!]$, the destination vertex of a_i is the source vertex of a_{i+1}. The cost $c(P) = (P_1, \ldots, P_d)$ of a k-hop path $P = a_1, \ldots, a_k$ is the vector sum $\sum_{1 \leq i \leq k} w(a_i)$. The *source* of a path $P = a_1, \ldots, a_k$ is the source of a_1 and its *destination* is that of a_k.

If $P = a_1, \ldots, a_k$ is a path and a_{k+1} is an arc whose source is the destination of a_k, notation $P \cdot a_{k+1}$ stands for the path $a_1, \ldots, a_k, a_{k+1}$, defined by the extension of P by a_{k+1}.

For a path P of cost $c(P) = (P_1, P_2, \ldots, P_d)$, its *rank* is defined as $\texttt{rank}(P) = \sum_{1 \leq i \leq d} P_i$. We define Λ as the number of rank values of the explored paths. It depends on the algorithm used. For legibility reasons, **each arc rank is strictly positive** in our algorithms description.

Most of the paths considered in the following will have a same source s. It will be specified otherwise. The algorithms presented will aim at computing shortest paths from s to all other vertices. This kind of query is called *one-to-all*.

The Table 2.1 gathers the notations used in this manuscript for multicriteria shortest path computation.

2.2 Pareto set of vectors

We first define how to compare path costs and what is an optimal cost. Then, we review existing algorithms which compute the so-called optimal costs among any set of costs.

2.2.1 Definitions

In dimension 1, we can compare two paths with the natural order on $\mathbb{R}$. In order to compare vectors in dimension $d > 1$, we consider the following partial order: a vector is smaller than another if it is smaller on all dimensions.

Definition 4 (Domination)**.** *Let $d \in \mathbb{N}\backslash\{0\}$ and $\mathcal{D} = (E_i, \leq_i)_{1 \leq i \leq d}$ be a Cartesian product of fully ordered sets $(E_i, \leq_i)$, called a set of dimensions. It is provided with the following order of domination:*

$$\forall x = (x_i)_i, y = (y_i)_i \in \mathcal{D}, x \leq_{Dom} y \Leftrightarrow \forall i \in [\![1; d]\!], x_i \leq_i y_i$$

For $x, y \in \mathcal{D}$, if $x \leq_{Dom} y$, we say that x dominates y. If x does not dominate y, nor y dominates x, then we say that x and y are incomparable.

For example, in Figure 2.1, the cost W dominates every green costs, such as A, while it is dominated by all red costs, like B. The blue costs are incomparable with W since they are larger on one coordinate and smaller on the other. Thus, C and D are incomparable with W.

Depending on the context, maximum or minimum vectors, Pareto sets (mathematics) or Skylines (data-mining) are different names of the same notion: the subset of vectors that are not dominated by the others.

Definition 5 (Pareto set)**.** *Let $\mathcal{T}$ be a set of n vectors of $\mathcal{D}$. The Pareto set of $\mathcal{T}$ is:*

$$\texttt{PS}(\mathcal{T}) = \{x \in \mathcal{T}, \forall y \in \mathcal{T}, not(y \leq_{Dom} x)\}$$

n	number of vertices
m	number of arcs
d	number of dimensions
Δ	maximum outgoing degree
s	source vertex
ε	$(1+\varepsilon)$ is the approximation factor
$x \leq_{Dom} y$	x dominates y $(\forall i, x_i \leq y_i)$
$\texttt{PS}(X)$	Pareto set of the set X
$\texttt{rank}(P)$	$\sum_{1 \leq i \leq d} P_i$ ranking function
Λ	number of ranks seen by the algorithm
$P \cdot a$	the extension of the path P by the arc a
$\mathcal{P}_{s \rightsquigarrow u}$	the set of paths from s to u
$\mathcal{P}^{v}_{s \rightsquigarrow u}$	the set of paths from s to u passing through v
$\mathcal{S}_u$	solution set for an exact algorithm, containing paths from s to u
S_u	$\lvert\mathcal{S}_u\rvert$
$\mathcal{S}$	solution set for an exact algorithm (thus, a Pareto set)
S	$\lvert\mathcal{S}\rvert$
$\mathcal{S}_\varepsilon$	solution set for an approximation algorithm, not necessarily a $(1+\varepsilon)$-Pareto set
S_ε	$\lvert\mathcal{S}_\varepsilon\rvert$
$\mathcal{T}$	set of temporary paths
$\mathcal{T}_u$	set of temporary paths of destination u
$\mathcal{R}$	set of paths of same rank,
R	$\lvert\mathcal{R}\rvert$

Table 2.1 – Notations for shortest path computation.

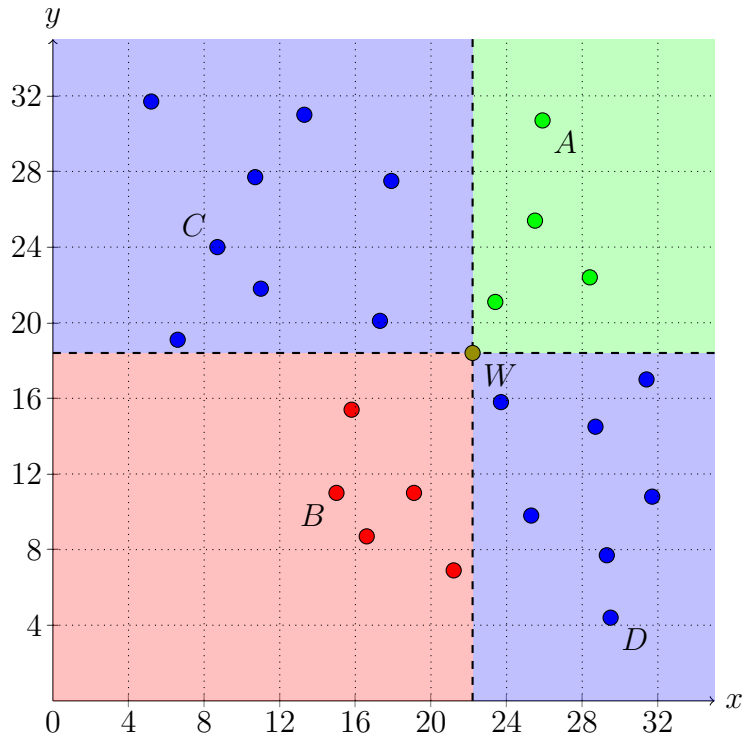


Figure 2.1 – Domination with two dimensions

In order to illustrate this notion, let us take the case of dimension 2, with $\mathcal{D} = \mathbb{R}^2$, through the example depicted in Figure 2.2. We represent a set of green points, noted $\mathcal{P}$. There is also a set of gray points, noted $\mathcal{R}$. The whole set is $\mathcal{T} = \mathcal{P} \cup \mathcal{R}$. We can observe:

- in green (above right), the areas dominated by points in $\mathcal{P}$. Any point of $\mathcal{R}$ is in a green zone so they are dominated by those in $\mathcal{P}$, which implies that they are not in the Pareto set. So $\texttt{PS}(\mathcal{T}) \subseteq \mathcal{P}$.
- in red (below left), the zones dominating the points of $\mathcal{P}$. We notice that no point is in the red zone of another one, which ensures that $\mathcal{P} \subseteq \texttt{PS}(\mathcal{T})$.

Thus, $\mathcal{P}$ is exactly the Pareto set of $\mathcal{T}$.

2.2.2 Exact algorithms

The input is a finite set of vectors $\mathcal{T} \subset \mathbb{R}^d$, of size $T = |\mathcal{T}|$. Let $S = |\texttt{PS}(\mathcal{T})|$ be the size of the output. We describe how to compute its Pareto set.

Naive algorithm

To compute $\texttt{PS}(\mathcal{T})$, a naive algorithm consists in comparing each pair of vectors. A decremental version is described in Algorithm 1, in which a vector is removed as soon as we notice that it is dominated. Its complexity is $\Theta(T^2)$, since each pair is compared if the input is already a Pareto set.

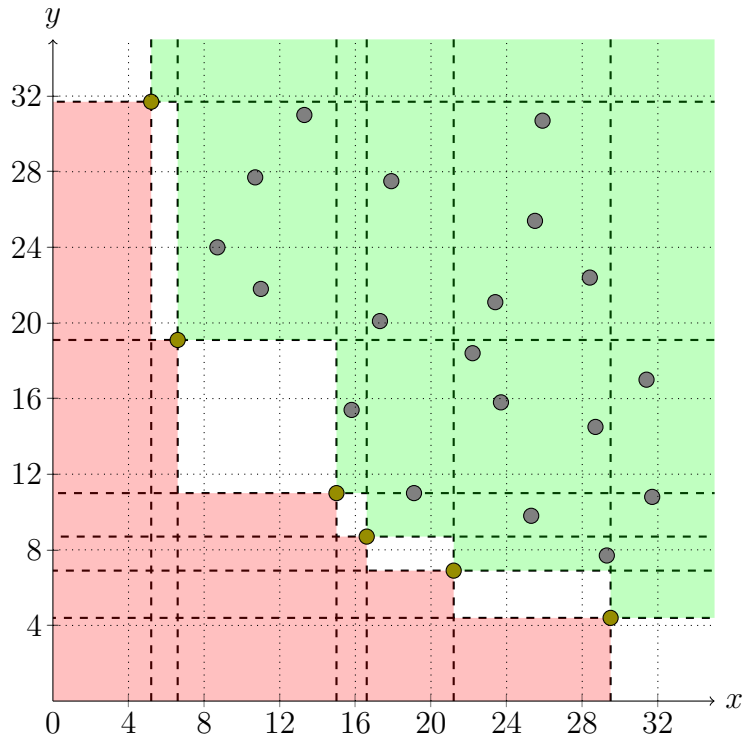


Figure 2.2 – Pareto set with two dimensions

Input: $\mathcal{T} \subset \mathbb{R}^d$ set of vectors
Output: $\mathcal{S} \subseteq \mathcal{T}$

```
1 S ← T
2 foreach w ∈ T do
3     foreach w' ∈ S\{w} do
4         if w' ≤_Dom w then
5             S ← S\{w}
6             break ; // no need to continue processing w
```

Algorithm 1: RemoveDominated naive algorithm

Algorithm by sorting

A well-known improvement consists in sorting the vectors beforehand. The procedure is incremental contrary to the presentation of the previous algorithm. $\mathcal{S}$ represents the Pareto set of the paths already considered, so it is empty at the beginning. Then, we go through the vectors in order. Each vector is inserted into $\mathcal{S}$ if and only if it is not dominated by any other vector of $\mathcal{S}$. In order to avoid the risk of having to delete vectors from $\mathcal{S}$, there are two usual orders:

- the lexicographic order,
- the rank order.

With each of these orders, a vector cannot be dominated by one of its successors. So if a vector is added in $\mathcal{S}$, it stays there definitively. It is thus an incremental algorithm.

Input: $\mathcal{T} \subset \mathbb{R}^d$ set of vectors
Output: $S \subseteq T$
1 $\mathcal{T}' = (w^i)_{1 \leq i \leq n} \leftarrow Sort(\mathcal{T})$
2 $\mathcal{S} \leftarrow \emptyset$
3 **foreach** $i \in [\![1, n]\!]$ **do**
4 $isDominated \leftarrow false$
5 **foreach** $w \in \mathcal{S}$ **do**
6 **if** $w \leq_{Dom} w_i$ **then**
7 $isDominated \leftarrow true$
8 **break** ; `// no need to continue processing` w_i
9 **if** $isDominated$ *is* $false$ **then**
10 $\mathcal{S} \leftarrow \mathcal{S} \cup \{w_i\}$

Algorithm 2: Algorithm for computing the Pareto set by sorting

It is interesting to express the output-sensitive complexity since the output size can be much smaller than the input one. Recall that $S = |\texttt{PS}(\mathcal{T})|$ denotes the output size. Beware that S is the size of $\mathcal{S}$ *at the end* of the algorithm, but not during the execution. Since any insertion into $\mathcal{S}$ is permanent, the complexity can be expressed as a function of S: $O(T \cdot (S + \log T))$. Indeed, the sorting is in $O(T \log T)$. Then each of the T vectors of $\mathcal{T}$ is compared to the elements of the current $\mathcal{S}$. At any time, $\mathcal{S}$ cannot contain more than S elements since no element is deleted from this set.

Offline algorithms

In the *offline* setting, the whole set of the T points on which we want to compute a Pareto set is given at the beginning.

Fast algorithms were proposed by Kung et al. [KLP75]. Those are preceded by a preliminary lexicographic sorting, followed by an algorithm in $O(T \log^{d-2}(T))$. There are three algorithms:

- For dimension 2, a simple linear algorithm is used, very similar to Algorithm 2, but more subtle. When the algorithm processes a vector $w = (w_1, w_2)$, it only compares it with the last kept vector. To convince ourselves that this is sufficient, we note $\mathcal{P}$ the set of kept vectors and w' the last one that was kept. When processing w, the paths in $\mathcal{P}$ are necessarily inferior or equal on the first dimension, and w' is the greater of those. Since $\mathcal{P}$ is a Pareto set of itself, $\forall w'' \in \mathcal{P} \backslash \{w'\}, w''_1 \leq w'_1$ and then $w''_2 \geq w'_2$. Thus, if w is not dominated by w', it is not either by any path in $\mathcal{P}$.

- For dimension 3, a balanced binary tree (Section A.2) is used to store a subset of the vectors already seen, or more precisely the projections of the vectors on the two last dimensions. For each processed vector, the domination test is logarithmic and the update of the tree has a logarithmic amortized complexity.

- The paradigm *divide and conquer* is used for any dimension $d > 3$, decreasing by 1 the dimension d and splitting the set of vectors in two equal parts at each recursive call. Then at dimension 3, it uses an algorithm very similar to the second algorithm.

	Sort	Remove Dominated
Complexity	$O(T \cdot \log(T))$	$O\left(T \cdot \log^{d-2}(T)\right)$

Furthermore, in the same paper, Kung et al. show that it is impossible to do better than $S \log(S)$. The proof is based on the same idea as the lower bound for sorting by comparison algorithms.

For $d > 3$, it has been improved in [GBT84] to $O(T \log^{d-3} T \log \log T)$.

In order to obtain output-sensitive complexities, Kirkpatrick *et al.* [KS85] proposed two algorithms running in $O(T \log S)$ time for $d = 2$ and $O(T \log^{d-2} S)$ time for $d > 2$.

Online algorithms

The *online* setting corresponds to the situation when the vectors are processed one by one, without knowing those which will come after. Although designed for *offline* purposes, the algorithms in [KLP75] for $d = 2$ and $d = 3$ can be simplified for the *online* setting when the vectors are processed in the lexicographic order. The sorting step becomes irrelevant and the computation time drops to $O(1)$ per insertion for $d = 2$. Thus, in an online setting, Kung et al. algorithms have the following complexities:

- $O(T)$ for $d = 2$,

- $O(T \log T)$ for $d = 3$.

Online algorithms are very useful for multicriteria shortest path computation as it will be detailed in Section 3.1.2. The processed vectors are not known in advance but are discovered little by little in the lexicographic order.

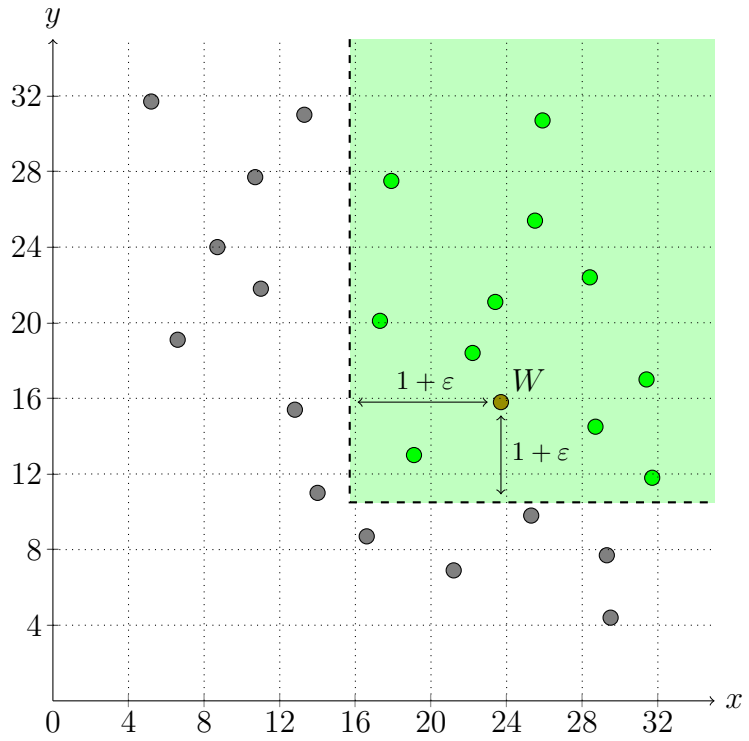


Figure 2.3 – $(1+\varepsilon)$-coverage with $\varepsilon = 0.5$

2.2.3 Approximation

In some cases, Pareto sets can be unreasonably large. An example will be given in Section 2.3.3. To summarize them, the method chosen in this manuscript is the coverage-based approximation. We restrict ourselves to strictly positive vectors.

Definition 6 ($(1+\varepsilon)$-coverage)**.** *Let $w, w' \in \mathbb{R}_+^d$. We say that w $(1+\varepsilon)$-covers w' if $\forall i \in [\![1, d]\!], w_i \leq (1+\varepsilon)w'_i$. A set of vectors $\mathcal{W} \subset \mathbb{R}_+^d$ is $(1+\varepsilon)$-covered by a set of vector $\mathcal{S}$ if $\forall w \in \mathcal{W}, \exists w' \in \mathcal{S}, w'$ $(1+\varepsilon)$-covers w. Whenever $\mathcal{W}$ is a Pareto set of itself, we also say that $\mathcal{W}$ is $(1+\varepsilon)$-approximated by $\mathcal{S}$.*

Let $C \geq 1$ be a constant and $\mathcal{P} \subset [1, C]^d$ be a Pareto set of itself. Notice that the vectors are positive and that the aspect ratio, the quotient of the maximum coordinate and the minimum coordinate, is bounded by C. In [PY00], Papadimitriou and Yannakakis showed that, whatever the size of $\mathcal{P}$, there always exists a $(1+\varepsilon)$-approximation of $\mathcal{P}$ whose size is polynomial in $1/\varepsilon$ and C. More precisely stated in [TZ09], we have the existence of a $(1+\varepsilon)$-approximation of $\mathcal{P}$ of size at most $\log_{1+\varepsilon}^{d-1}(C)$.

This result is based on the fact that a space covering $\mathcal{T}$ can be partitioned into $\log_{1+\varepsilon}^{d}(C)$ parts, as we will see in section 3.4.1. Each part is composed of the set of vectors $w = (w_i)_{1 \leq i \leq d}$ sharing the same vector $\left(\lfloor \log_{1+\varepsilon}(w_i) \rfloor\right)_i$. It is then sufficient to keep only one per part in order to cover every path from $\mathcal{P}$. Better, for each value $x \in [\![0, \lfloor \log_{1+\varepsilon}(C) \rfloor]\!]$, we consider only the non-empty part having x as first coordinate. This allows to decrease the exponent by one.

In order to compute this $(1+\varepsilon)$-approximation, we can easily deduce Algorithm 3, which is in $O\left(\log^d_{1+\varepsilon}(C)\right)$.

Input: $\mathcal{P} \subset (\mathbb{R}_+^*)^d$ a Pareto set of itself
Output: $\mathcal{S} \subseteq \mathcal{P}$
$\mathcal{S} \leftarrow$ empty d-dimensional array
foreach $P \in \mathcal{P}$ **do**
  $pos \leftarrow \left(\lfloor \log_{1+\varepsilon}(P_i) \rfloor\right)_{2 \le i \le d}$
  if $\mathcal{S}[pos]$ *is empty or* $P_1 < \mathcal{S}[pos]_1$ **then**
    $\mathcal{S}[pos] \leftarrow P$

Algorithm 3: Algorithm for computing $(1+\varepsilon)$-approximation of a Pareto set

2.2.4 Range queries.

In this section, the problem is to find out if a point P is $(1+\varepsilon)$-covered by a point set $\mathcal{S}$, not necessarily being a Pareto set. For example, this can be used to compute a $(1+\varepsilon)$-approximation of $\mathcal{S}$, by removing iteratively $(1+\varepsilon)$-covered points. This problem can be solved using *range queries* in dimension d.

Given a Cartesian product of intervals $\mathcal{I} = [x_1, x_1'] \times [x_2, x_2'] \times \ldots \times [x_d, x_d']$ and a point set $\mathcal{S}$, $\texttt{RangeQuery}(\mathcal{I}, \mathcal{S})$ reports every point Q in $\mathcal{S} \cap \mathcal{I}$.

We use such queries to test $(1+\varepsilon)$-coverage or finer properties. A point P is $(1+\varepsilon)$-covered by a point set $\mathcal{S}$ if and only if $\texttt{RangeQuery}\left(\prod_i [0, (1+\varepsilon)P_i], \mathcal{S}\right)$ is not empty.

In our case, we will not require to report every point in the subspace specified by the intervals but just to learn if there is at least one point. Thus, we have:

Lemma 7 ([Mor06]). *Given a point $P \in \mathbb{R}^d$ and a set $\mathcal{S} \subset \mathbb{R}^d$ of n points, a data structure $\mathcal{D}(\mathcal{S})$ using $O(n \log^{d-1} n)$ memory space can be preprocessed, such that any orthogonal range query and thus any $(1+\varepsilon)$-coverage checking can be done in $O\left(\left(\frac{\log n}{\log\log n}\right)^{d-1}\right)$ time. Moreover, adding or deleting a point in the data structure $\mathcal{D}(\mathcal{S})$ takes $O(\log^{d-1} n)$ time.*

2.3 Pareto set of paths

Here, the concepts of optimality and Pareto set are extended to paths and a study about the links between the optimal paths and the optimal prefixes is conducted. The paths considered in this section are in the same weighted directed graph $\mathcal{G} = (V, A, w)$. Let $s \in V$ be a source vertex and $t \in V$ be a target vertex.

2.3.1 Definitions

Let us note:

- $\mathcal{P}_{s \rightsquigarrow t}$, the set of paths from s to t,
- $\mathcal{C}_{s \rightsquigarrow t} = \{c(P) | P \in \mathcal{P}_{s \rightsquigarrow t}\}$ the set of their costs.

Definition 8 (Shortest path)**.** *Let s be a source vertex and $t \in$ a target vertex. A path P is a* shortest path *from s to t if $c(P) \in \mathtt{PS}(\mathcal{C}_{s \rightsquigarrow t})$. In other words, it is a path whose cost is not dominated by the cost of another path having the same source and destination.*

In general, it is possible to have several paths sharing source, destination and cost but being different because not containing the same edges. In this manuscript, we only want to differentiate the paths by their cost and not the arcs that compose them. This is why we quotient the set of paths by the following relation: $P \sim Q$ if and only if P and Q have the same source, destination and cost. Informally, we consider only one path among those which share these same characteristics, no matter which one.

Remark. *It can be interesting to distinguish these paths in other contexts, for example when one wants to find several paths with the least common vertices. This can be useful in case of unexpected deletion of an arc, because of a car accident or a router failure.*

Definition 9 (Pareto set of paths)**.** *Let $\mathcal{P}$ be a set of paths having the same source and the same destination. Let $\mathcal{C} = \{c(P) | P \in \mathcal{P}\}$ be the set of costs of these paths. The* Pareto set *of $\mathcal{P}$ is $\mathtt{PS}(\mathcal{P}) = \{P \in \mathcal{P} | c(P) \in \mathtt{PS}(\mathcal{C})\}$. Thus, the set of shortest paths from s to t is exactly $\mathtt{PS}(\mathcal{P}_{s \rightsquigarrow t})$.*

For instance, consider the graph of the Figure 2.4. Many paths allow to go from s to t. The set $\mathcal{P}_{s \rightsquigarrow t}$ is composed of the following paths:

- $P^{(1)} : s \to u_1 \to u_4 \to t$. Its cost is $(23, 6)$,
- $P^{(2)} : s \to u_1 \to u_4 \to u_5 \to t$. Its cost is $(24, 11)$,
- $P^{(3)} : s \to u_2 \to u_4 \to t$. Its cost is $(21, 12)$,
- $P^{(4)} : s \to u_2 \to u_4 \to u_5 \to t$. Its cost is $(22, 17)$,
- $P^{(5)} : s \to u_2 \to u_5 \to t$. Its cost is $(16, 15)$,
- $P^{(6)} : s \to u_2 \to u_6 \to t$. Its cost is $(14, 12)$,
- $P^{(7)} : s \to u_3 \to u_2 \to u_4 \to t$. Its cost is $(15, 24)$,
- $P^{(8)} : s \to u_3 \to u_2 \to u_4 \to u_5 \to t$. Its cost is $(16, 29)$,
- $P^{(9)} : s \to u_3 \to u_2 \to u_5 \to t$. Its cost is $(10, 27)$,
- $P^{(10)} : s \to u_3 \to u_2 \to u_6 \to t$. Its cost is $(8, 24)$,
- $P^{(11)} : s \to u_3 \to u_6 \to t$. Its cost is $(10, 17)$.

We have $\mathtt{PS}(\mathcal{P}_{s \rightsquigarrow t}) = \{P^{(1)}, P^{(6)}, P^{(10)}, P^{(11)}\}$. On Figure 2.5, the Pareto set is composed of the green points, while the gray points are dominated.

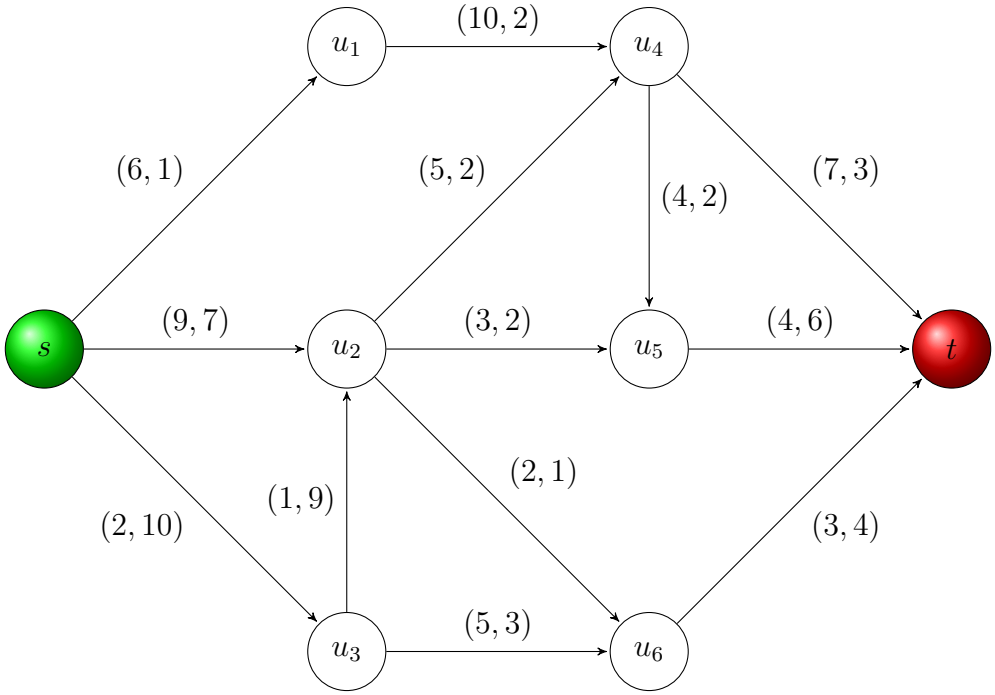


Figure 2.4 – Example of Pareto set of paths

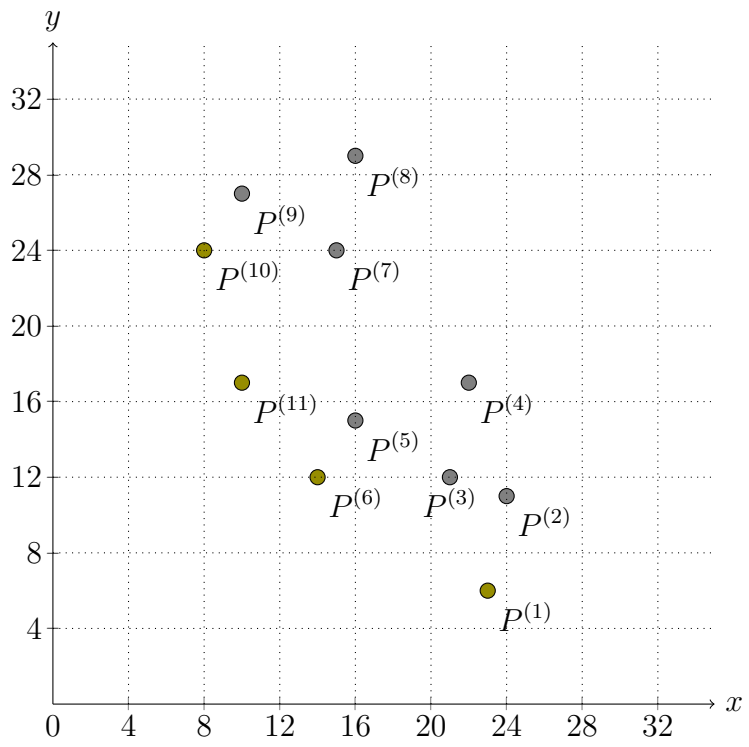


Figure 2.5 – Pareto set with two dimensions

2.3.2 Impact of the deletion of optimal paths

Let $u \in V$ be a vertex. We introduce two notations:

- $\mathcal{P}^u_{s \rightsquigarrow t}$ the set of paths from s to t passing through u,
- $\mathcal{A} \cdot \mathcal{B} = \{P = Q \cdot R | Q \in \mathcal{A}, R \in \mathcal{B}\}$ the Cartesian product of two path sets $\mathcal{A}$ and $\mathcal{B}$, i.e. the set of all concatenations between a path from $\mathcal{A}$ and one from $\mathcal{B}$.

In dimension 1, $\texttt{PS}(\mathcal{P}^u_{s \rightsquigarrow t}) = \texttt{PS}(\mathcal{P}_{s \rightsquigarrow u}) \cdot \texttt{PS}(\mathcal{P}_{u \rightsquigarrow t}) = \texttt{PS}(\texttt{PS}(\mathcal{P}_{s \rightsquigarrow u}) \cdot \texttt{PS}(\mathcal{P}_{u \rightsquigarrow t}))$. Some algorithms for shortest path computation are based on this fact, such as the PLL algorithm (see Section 1.2.3). This is no longer true in dimension superior to 1. The more general issue that interests us in this section is the following: if we have only some paths from s to u and some from u to t, which optimal paths from s to t passing through u can be found. More formally:

Question 10. *Let $\mathcal{A} \subseteq \mathcal{P}_{s \rightsquigarrow u}$ and $\mathcal{B} \subseteq \mathcal{P}_{u \rightsquigarrow t}$. Depending on $\mathcal{A}$ and $\mathcal{B}$, what is the relationship between $\texttt{PS}(\mathcal{P}^u_{s \rightsquigarrow t})$ and $\texttt{PS}(\mathcal{A} \cdot \mathcal{B})$?*

First of all, if we give ourselves $\mathcal{P}_{s \rightsquigarrow u}$ and $\mathcal{P}_{u \rightsquigarrow t}$, it is easy to deduce $\texttt{PS}(\mathcal{P}^u_{s \rightsquigarrow t})$, since $\mathcal{P}^u_{s \rightsquigarrow t} = \mathcal{P}_{s \rightsquigarrow u} \cdot \mathcal{P}_{u \rightsquigarrow t}$. Thus, $\texttt{PS}(\mathcal{P}^u_{s \rightsquigarrow t}) = \texttt{PS}(\mathcal{P}_{s \rightsquigarrow u} \cdot \mathcal{P}_{u \rightsquigarrow t})$.

Non optimal prefixes and suffixes are useless. We then notice that we can restrict ourselves to optimal paths in $\mathcal{A}$ and $\mathcal{B}$.

Lemma 11. *Assume that:*

- $\texttt{PS}(\mathcal{P}_{s \rightsquigarrow u}) \subseteq \mathcal{A} \subseteq \mathcal{P}_{s \rightsquigarrow u}$,
- $\texttt{PS}(\mathcal{P}_{u \rightsquigarrow t}) \subseteq \mathcal{B} \subseteq \mathcal{P}_{u \rightsquigarrow t}$.

Then $\texttt{PS}(\mathcal{A} \cdot \mathcal{B}) = \texttt{PS}(\mathcal{P}^u_{s \rightsquigarrow t})$.

Proof. For the direct inclusion, let $P \in \texttt{PS}(\mathcal{A} \cdot \mathcal{B})$. Let $Q \in \mathcal{A}, R \in \mathcal{B}$ such that $P = Q \cdot R$. Reasoning by the absurd, if $P \notin \texttt{PS}(\mathcal{P}^u_{s \rightsquigarrow t})$, then $\exists P' \in \texttt{PS}(\mathcal{P}^u_{s \rightsquigarrow t})$ such that P' dominates P. We can decompose $P' = Q' \cdot R'$ with $Q' \in \mathcal{P}_{s \rightsquigarrow u}$ and $R' \in \mathcal{P}_{u \rightsquigarrow t}$. Then $Q' \in \texttt{PS}(\mathcal{P}_{s \rightsquigarrow u})$, otherwise it would be dominated by a path Q'' and $Q'' \cdot R'$ would dominate P'. For the same reason, $R' \in \texttt{PS}(\mathcal{P}_{u \rightsquigarrow t})$. This gives us that $P' \in \mathcal{A} \cdot \mathcal{B}$, which is absurd by definition of P.

For the reverse inclusion, let $P \in \texttt{PS}(\mathcal{P}^u_{s \rightsquigarrow t})$. Similarly to the direct inclusion, we can decompose $P = Q \cdot R$. By the same arguments, Q and R are optimal. Therefore, using the hypothesis of the lemma, $Q \in \mathcal{A}$ and $R \in \mathcal{B}$. It means that $P \in \mathcal{A} \cdot \mathcal{B}$. If P is dominated by a path $P' \in \texttt{PS}(\mathcal{A} \cdot \mathcal{B})$, then by the direct inclusion, $P' \in \texttt{PS}(\mathcal{P}^u_{s \rightsquigarrow t})$. This is absurd since this set is a Pareto set and P' dominates P, while being both in it. □

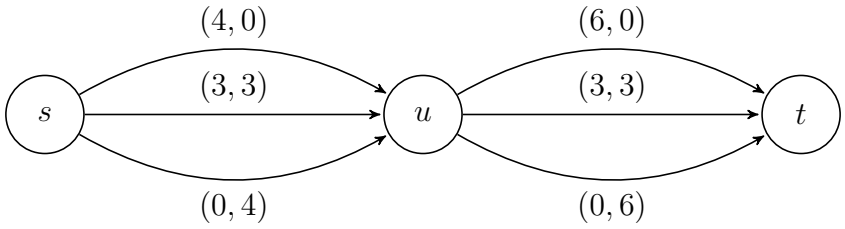


Figure 2.6 – Example when removing an optimal prefix has no impact

An optimal solution may be lost if an optimal prefix or suffix is not given. We suppose now that there exists $P \in \texttt{PS}(\mathcal{P}_{s \rightsquigarrow u}) \backslash \mathcal{A}$. Then, it is possible to no longer be able to compute all the optimal paths from s to t passing through u from $\mathcal{A}$ and $\mathcal{B}$, or more formally that there exists $Q \in \texttt{PS}(\mathcal{P}^u_{s \rightsquigarrow t}) \backslash \texttt{PS}(\mathcal{A} \cdot \mathcal{B})$.

For instance, consider a graph with only three vertices: s, u and t. From s to u, there are two incomparable arcs a and a' and there is a single arc b from u to t. Then, $a \cdot b$ and $a' \cdot b$ are incomparable. Thus the deletion of a makes lose the path $a \cdot b$ which is however part of $\texttt{PS}(\mathcal{P}^u_{s \rightsquigarrow t})$.

However, this is not always the case. An example is given such that $\texttt{PS}(\mathcal{P}^u_{s \rightsquigarrow t}) = \texttt{PS}(\mathcal{A} \cdot \mathcal{B})$ in Figure 2.6. The arc from s to u of weight $(3, 3)$ is in $\texttt{PS}(\mathcal{P}_{s \rightsquigarrow u})$ but is not a prefix of a path in $\texttt{PS}(\mathcal{P}^u_{s \rightsquigarrow t})$. Thus, not having it in $\mathcal{A}$ is not a problem.

Can we lose them all? Since we do not systematically lose an optimal solution by removing an optimal prefix, then we risk losing them all. Indeed, this would mean that some prefixes are useless and if only useless prefixes are kept, no optimal solution can be found. For instance, in the example of Figure 2.6, if $\mathcal{A}$ contains only the arc of weight $(3, 3)$, then $\texttt{PS}(\mathcal{A} \cdot \mathcal{B}) \cap \texttt{PS}(\mathcal{P}^u_{s \rightsquigarrow t}) = \emptyset$. So we have to be careful which paths are removed.

Keeping the minima on each criteria guarantees to output some optimal paths but a user might be more interested by intermediate paths, since those represent compromises.

In general, to compute $\texttt{PS}(\mathcal{P}^u_{s \rightsquigarrow t})$, it is necessary to have all the prefixes from $\texttt{PS}(\mathcal{P}_{s \rightsquigarrow u})$ and all the suffixes from $\texttt{PS}(\mathcal{P}_{u \rightsquigarrow t})$.

2.3.3 Approximation

The definition of coverage is the same for paths as for weights.

Definition 12 ($(1+\varepsilon)$-coverage)**.** *Let P, P' be two paths. We say that P $(1+\varepsilon)$-covers P' if $c(P)$ $(1+\varepsilon)$-covers $c(P')$. A set of paths $\mathcal{P}$ is $(1+\varepsilon)$-covered by a set of paths $\mathcal{S}$ if $\forall P \in \mathcal{P}, \exists P' \in \mathcal{S}$, P' $(1+\varepsilon)$-covers P. Whenever $\mathcal{P}$ is a Pareto set of itself, we also say that $\mathcal{P}$ is $(1+\varepsilon)$-approximated by $\mathcal{S}$.*

Remark. *Let $P = a_0, \ldots, a_k$ and $P' = a'_0, \ldots a'_{k'}$ be two paths sharing the same source and destination: $a_0 = a'_0 = s$ and $a_k = a_{k'}$. If $\texttt{rank}(P) > \texttt{rank}(P')$ then P cannot dominate P'. Depending on $\varepsilon > 0$, P could however $(1+\varepsilon)$-cover P'. In Figure 1.3, $\texttt{rank}(G) = 21$ and $\texttt{rank}(B) = 18$ but G 2-covers B.*

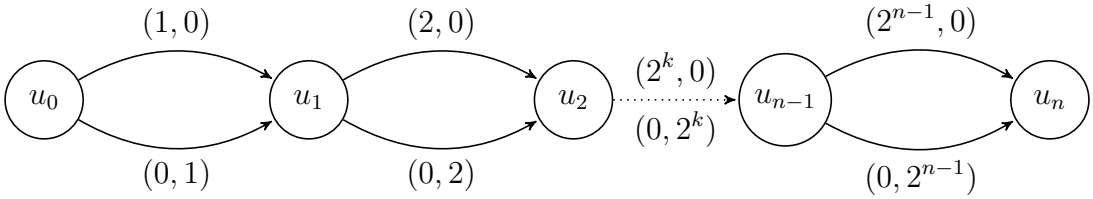


Figure 2.7 – Pathological example

Huge Pareto set of paths. In Section 2.2, we studied the $(1+\varepsilon)$-approximation of a Pareto set, the motivation being that a Pareto set can be too large. One naturally wonders how large a Pareto set of paths can be. In [Han80], a simple example of a Pareto set having exponential size can be found. This example is represented on Figure 2.7. There is a sequence of $n+1$ vertices $u_0, \ldots, u_n$, such that, $\forall i \in [\![0, n-1]\!]$, there are two arcs from u_i to u_{i+1} whose weights are respectively $(2^i, 0)$ and $(0, 2^i)$.

We can show that any path from u_0 to u_n is non-dominated by the others. So there are 2^n optimal paths. The size of the Pareto set is therefore exponential in the number of vertices. Notice however that the weights are themselves exponential. Hansen specifies in [Han80] that the size of the Pareto set is polynomial in the weights.

This example is not a simple graph since it contains multiple arcs, i.e. several arcs having the same source and destination. But it is easy to adapt it by adding intermediate vertices in the middle of each arc. Breugem et al. propose in [BDH17] an adaptation with fewer arcs. This time, the graph is a tournament, i.e. a complete directed graph. For any $0 \leq i < j \leq n$, (u_i, u_j) has weight $(2^{j-1} - 2^i, 2^{j-1})$. All the paths in this graph from u_0 to u_n are optimal and there are 2^{n-1} since choosing a path means taking a subset of intermediate vertices through which the path passes. In the following, we will use this graph to conduct experiments on large Pareto sets.

Normalization of weights. To approximate Pareto set of weights, we had restricted ourselves to weights in $[1, C]^d$ with $C \geq 1$ a constant. In a more general case, if the weights are in $(\mathbb{R}_+^*)^d$, then we can normalize them by multiplying them by the aspect ratio C. It may be tempting to use a function $f : (\mathbb{R}_+^*)^d \to (\mathbb{R}_+^*)^d$ whose application to the weights of the arcs would make the computation of shorter paths easier, by reducing the aspect ratio for example. However, if we want to preserve the order on the paths, only very specific f are suitable.

Let us look at the dimension 1. Let $x, y \in \mathbb{R}_+^*$. We notice that f must be increasing: if two arcs have weights x and y, then $x < y \Rightarrow f(x) < f(y)$. Moreover, let us take a graph with three vertices: s, u and t, with an arc from s to u of weight x, one from u to t of weight y and one from s to t of weight $x + y$. In order for both paths to keep the same weight, we need $f(x) + f(y) = f(x + y)$.

Then, it is a classical exercise to show that $f = f(1) \cdot Id_{\mathbb{R}_+^*}$. For this, we can show by induction that $\forall x \in \mathbb{R}_+^*, \forall k \in \mathbb{N}, f(k \cdot x) = k \cdot f(x)$. We deduce that, for all $p, q \in \mathbb{N}$ and $q \neq 0$, $f(1) = qf\left(\frac{1}{q}\right)$ and thus that $f\left(\frac{p}{q}\right) = \frac{p}{q} f(1)$. Thus, $\forall r \in \mathbb{Q}, f(r) = r \cdot f(1)$. Since f is increasing, by density of $\mathbb{Q}$ in $\mathbb{R}$, we have that f

is linear of coefficient $f(1)$.

2.4 Computing shortest paths in dimension 1: Dijkstra

In this section, we restrict ourselves to the case $d = 1$. We thus have that the weight function has its image in $\mathbb{R}_+$.

2.4.1 Algorithm

The single-criterion Dijkstra's algorithm can be summarized as follows. Given a weighted directed graph $\mathcal{G}$ and a source vertex s, the algorithm maintains a set $\mathcal{T}$ of vertices to process, initially containing only s and, for each vertex u, a candidate distance $d(u)$ from s to u, initially set to 0 for s and to infinity for the other vertices. At each step of the algorithm, the vertex $u \in \mathcal{T}$ with minimum distance value $d(u)$ is considered, and removed from the set $\mathcal{T}$. For each arc (u, v), a corresponding length $l_v = d(u) + c(u, v)$ is computed and compared with $d(v)$. If $l_v < d(v)$, then $d(v)$ is set to l_v, otherwise nothing happens. The algorithm terminates when $\mathcal{T}$ becomes empty. At the end of the algorithm, the distance value $d(u)$ maintained by the algorithm is the exact distance from s to u.

In order to efficiently find the vertex v of minimum distance, $\mathcal{T}$ is a priority queue (Section A.3). It contains pairs (u, l_u), with u a vertex and $l_u = d(u)$ its tentative distance. The order of the priority queue is induced by the second coordinate, i.e. the distance, which allows to determine the vertex of minimum distance. We specify here that we use a priority queue allowing to decrease a key, that is to say that it allows to replace a pair (u, l_u) into (u, l'_u), with $l'_u < l_u$. We note this operation `decreaseKey`, which takes three parameters: a priority queue $\mathcal{T}$, a vertex u such that $\mathcal{T}$ contains a pair (u, l_u), and a new distance $l'_u < l_u$. This way, when $d(v)$ is modified for a given vertex v, the pair (v, l_v) in $\mathcal{T}$ can also be modified. The distances are thus kept up to date in the priority queue. The corresponding pseudo code is the Algorithm 4.

Correctness. The correctness of this algorithm can be verified by induction showing that at any time, for any vertex u, $d(u)$ is the minimum distance between s and u passing only through vertices already processed, i.e. having already exited the priority queue $\mathcal{T}$.

Complexities. The time complexity of Dijkstra algorithm depends on the data structure chosen for $\mathcal{T}$:

- $O(m \log n)$ with a binary heap. This is based on the fact that each edge is processed at most once, that each update of $\mathcal{T}$ due to the exploration of a new edge is logarithmic, and finally that $\mathcal{T}$ contains at most one pair for each vertex. An advantage of the binary heap is its simplicity in practice.

Input: Graph $\mathcal{G} = (V, A, w)$ with V the vertices, A the arcs, w the weight function, $s \in V$ the source vertex
Output: Distance function $d : V \to \mathbb{R}_+$

```
1  begin Initialization
2      foreach u ∈ V do
3          d(u) ← +∞
4      d(s) ← 0
5      T ← {(s, 0)}
6  while T ≠ ∅ do
7      let (u, l_u) min of T
8      T ← T \ {(u, l_u)}
9      foreach (u, v) ∈ A do
10         if l_u + w(u, v) < d(v) then // v is not processed yet
11             d(v) ← l_v
12             if d(v) = +∞ then // v has not been seen yet
13                 T ← T ∪ {(v, l_u + w(u, v))}
14             else // v is a key in T
15                 T ← decreaseKey(T, v, l_u + w(u, v))
```

Algorithm 4: Dijkstra algorithm

- $O(m + n \log n)$ with a strict Fibonacci heap. Insertions and `decreaseKey` are in constant time but deletions remain in logarithmic time. However, there are only n deletions since each vertex passes only once through $\mathcal{T}$. However, this data structure is complex and less efficient in practice [Fre+86].

Distances to paths. This algorithm does not compute shortest paths but only distances, i.e. their costs. It can easily be adapted to retrieve a path corresponding to each distance. For any vertex v, $d(v) = (l_v, u)$ is now a pair made of a distance and a vertex. The associated vertex u is the penultimate vertex of the path taken that has given the distance l_v. More precisely, the line 11 is replaced by $d(v) \leftarrow (l_v, u)$. To unfold the path, just determine the predecessor recursively starting from v, until arriving at s. The correction of this method can also be shown by induction, based on the fact that when we modify $d(v) = (l_v, u)$, then $d(u)$ cannot be modified afterwards and thus allows to find a shorter path from s to u.

Remark. *For each vertex u, Dijkstra's algorithm computes* a *shortest path from s to u. There might exists several paths having the same weight, but only one is kept.*

In a context of road or foot transportation, the graphs are:

- almost planar ("almost" because of bridges/tunnels),
- connected: there is no point in exploring connected components other than the starting vertex one,

- simple: if there are two edges between the same vertices, you might as well keep only the best one. This remark becomes false in the multicriteria case.

Thanks to Euler's formula, we know that, for a connected simple planar graph, the number of arcs is linear in the number of vertices. More precisely, we have the following inequality: $m \leq 3n - 6$.

Remark. *However, transport networks are not always planar, especially airline or maritime networks.*

Negative weights, Bellman-Ford algorithm. An important limitation of the Dijkstra algorithm is the positivity of the weights. If there are negative weights, the guarantee that the outgoing distances of the priority queue are increasing is lost. It then becomes possible to insert a pair (u, l_u) into $\mathcal{T}$ while a pair (u, l'_u), with $l_u < l'_u$, has already been extracted from $\mathcal{T}$. With the pseudo-code of Algorithm 4, the announced complexities are no longer true. One might also argue that the correction is no longer true either. This is due to the fact that we can consider that each vertex is extracted from $\mathcal{T}$ only once. We have not imposed this restriction here to anticipate with the multicriteria generalization.

Bellman-Ford algorithm allows to deal with negative weights. Its time complexity is in $O(mn)$. Its is based on dynamic programming paradigm, computing iteratively the distance from the source to each vertex using at most 1 arc, then 2, etc... More formally, for any vertex u and any positive integer k, we compute $d(u, k)$ the distance from s to u through at most k arcs using:

- $d(u, 0) = +\infty$ for all $u \neq s$ and $d(s, 0) = 0$.
- $d(u, k) = \min \left\{ d(u, k-1), \min\limits_{(v,u) \in A} \{d(v, k-1) + w(v, u)\} \right\}$

2.4.2 Example

In order to illustrate the execution of the Algorithm 4, consider the graph depicted in Figure 2.8. The table 2.2 represents the content of the priority queue $\mathcal{T}$, as well as the tentative distances $d(u)$ for each vertex $u \neq s$ of the graph. These values are initialized to $+\infty$. The boxes of $d(u)$ are grayed out from the step where u leaves the priority queue: thereafter, $d(u)$ becomes constant.

In the case of one-to-one request, i.e when we want to go from one vertex to another, a halt criterion can be added. If we only want to know the distance from s to t, the last step of the algorithm is not necessary. Indeed, t becomes grayed out at step 8, which means that its distance will not be improved afterwards. The algorithm can therefore stop.

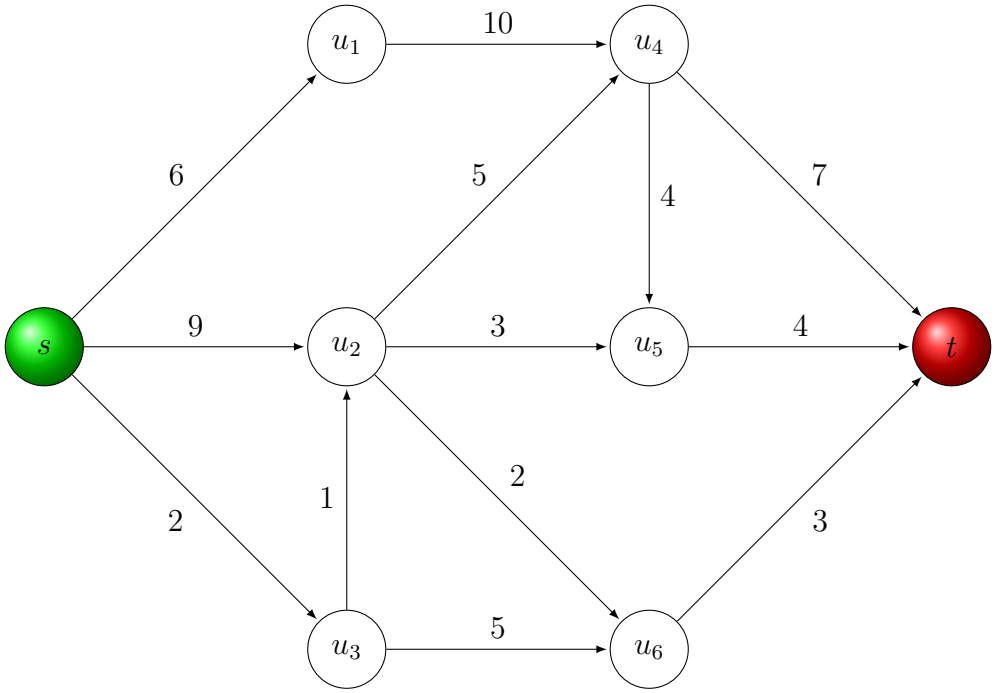


Figure 2.8 – Graph on which the Dijkstra algorithm is executed

Step	$\mathcal{T}$	$d(u_1)$	$d(u_2)$	$d(u_3)$	$d(u_4)$	$d(u_5)$	$d(u_6)$	$d(t)$
1	$(s, 0)$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
2	$(u_3, 2) \mid (u_1, 6) \mid (u_2, 9)$	6	9	2	$+\infty$	$+\infty$	$+\infty$	$+\infty$
3	$(u_2, 3) \mid (u_1, 6) \mid (u_6, 7)$	6	3	2	$+\infty$	$+\infty$	7	$+\infty$
4	$(u_6, 5) \mid (u_1, 6) \mid (u_5, 6) \mid (u_4, 8)$	6	3	2	8	6	5	$+\infty$
5	$(u_1, 6) \mid (u_5, 6) \mid (t, 8) \mid (u_4, 8)$	6	3	2	8	6	5	8
6	$(u_5, 6) \mid (t, 8) \mid (u_4, 8)$	6	3	2	8	6	5	8
7	$(t, 8) \mid (u_4, 8)$	6	3	2	8	6	5	8
8	$(u_4, 8)$	6	3	2	8	6	5	8
9	$\emptyset$	6	3	2	8	6	5	8

Table 2.2 – Content of data structures during the execution of Dijkstra algorithm.

Chapter 3

Multicriteria shortest path computation

In this chapter, the number of dimension d is no longer equal to 1. We first describe MC Dijkstra (Section 3.1) in order to better understand our algorithms presented afterwards. Then, we propose Meta Rank, a variant of MC Dijkstra in the form of a framework (Section 3.2).

We propose in Section 3.3 two instantiations of this framework, Bucket and Dijkstra Post for the exact Pareto set computation. Then we study the approximated computation. In the Section 3.4, we first study the first natural ideas and their limits. Finally, in the Section 3.5, we propose three instantiations of Meta Rank: Sector, SSector and QSSector. These algorithms allow the computation of $(1+\varepsilon)$-approximated Pareto sets with reasonable output sensitive complexities.

3.1 MC Dijkstra

The MC Dijkstra algorithm follows the same general idea as Dijkstra (see Section 2.4), adapted to the case of multiple criteria. In this case, the goal is to obtain the Pareto set from s to v for each vertex v: this is a *one-to-all query* in a multicriteria context. For this reason, the algorithm maintains a set $\mathcal{T}$ of paths rather than vertices. This set is initialized with the empty path from s to s. Also, for each vertex v, the algorithm maintains a candidate Pareto set $\mathcal{S}_v$, initialized to the empty set.

Similarly as in the single-criterion case, MC Dijkstra selects at each step the minimum of $\mathcal{T}$. More precisely, MC Dijkstra selects the path P in $\mathcal{T}$ which has the lexicographically minimum cost. If v is the destination of P, then P is added to the set $\mathcal{S}_v$. Again similarly, each path P' which consists of P plus one arc from the destination of P is considered. Let w be the destination of P'. If P' is dominated by a path in $\mathcal{S}_w$ or by a path in $\mathcal{T}$ with the same destination, P' is discarded. Otherwise, P' is added to $\mathcal{T}$, and any path $P'' \in \mathcal{T}$ with the same destination as P' which is dominated by it is removed from $\mathcal{T}$.

The algorithm terminates when $\mathcal{T}$ is empty at the end of a step. At that time, the sets $\mathcal{S}_v$ contain Pareto sets from s to every vertex v.

3.1.1 Pseudo code

A more formal description of MC DIJKSTRA is given in Algorithm 5. In this algorithm, we use the following two functions:

- `IsNotDominated`$(P, \mathcal{S})$ takes a path P and a Pareto set $\mathcal{S}$ as input. It returns `True` if the path P is not dominated by any path in $\mathcal{S}$, and `False` otherwise. This operation corresponds to the test Line 10 in Algorithm 4 in the multicriteria setting.

- `InsertAndClean`$(P, \mathcal{S})$ takes a path P and a Pareto set $\mathcal{S}$ as input and returns a Pareto set of $\mathcal{S} \cup \{P\}$. This operation generalizes Lines from 11 to 15 in Algorithm 4.

We also denote lexmin of a set $\mathcal{P}$ of paths a path of minimum cost in lexicographic order, i.e. a path in $\arg\min\{c(P) | P \in \mathcal{P}\}$.

Input: Graph $\mathcal{G} = (V, A, w)$ with V the vertices, A the arcs, w the weight function, $s \in V$ the source vertex
Output: Sets $\mathcal{S}_u$ for every vertex u

1 **begin** Initialization
2 **foreach** $u \in V$ **do**
3 $\mathcal{S}_u \leftarrow \emptyset$
4 $\mathcal{T}_u \leftarrow \emptyset$
5 $\mathcal{T}_s \leftarrow \{$empty path from s to $s\}$
6 **while** $\bigcup_{u \in V} \mathcal{T}_u \neq \emptyset$ **do**
7 let P of destination v be the lexmin of $\bigcup_{u \in V} \mathcal{T}_u$
8 $\mathcal{T}_v \leftarrow \mathcal{T}_v \setminus \{P\}$
9 $\mathcal{S}_v \leftarrow \mathcal{S}_v \cup \{P\}$
10 **foreach** $(v, w) \in A$ **do**
11 **if** `IsNotDominated`$(P \cdot (v, w), \mathcal{S}_w)$ **then**
12 $\mathcal{T}_w \leftarrow$ `InsertAndClean`$(P \cdot (v, w), \mathcal{T}_w)$

Algorithm 5: MC DIJKSTRA overview

3.1.2 Existing results

MC DIJKSTRA algorithm computes the exact Pareto sets from a source vertex to any other vertex (see [Mar84] and [Ehr05]). Its complexity heavily depends on the parts removing dominated paths, i.e. on the functions `IsNotDominated` and `InsertAndClean`. Nevertheless, existing papers simply use a naive algorithm for these functions, except for dimension 2, for which [Han80] claims a logarithmic complexity. In order to lower the complexity of MC DIJKSTRA, we may use the algorithms described in Section 2.2 to remove paths that are dominated. For $d = 2$

and $d = 3$, we can use online algorithms since MC DIJKSTRA processes elements in lexicographic order. The following proposition is more or less an agglomeration of existing results, with small adjustments in order to obtain a consistent statement.

Proposition 13 (partially from [Han80] and [BDH17])**.** *Let μ be the maximum number of arcs between a pair of vertices, and S be the size of the Pareto et. The output-sensitive time complexity of* MC DIJKSTRA *is:*

- $O(\Delta S \log(\Delta S))$ *for* $d \leq 3$
- $O(\mu \Delta S^2)$ *for* $d > 3$

Proof. In all cases, the size of a set $\mathcal{T}_u$ (the subset of paths from $\mathcal{T}$ having the same destination u) is upper-bounded by μS, since any path is an extension of an optimal one (a path in some $\mathcal{S}_v$), and there exist at most μ extensions of a path having the same destination. The same reasoning leads to the fact that the union of all the sets $\mathcal{T}_u$ has cardinality at most ΔS.

Besides, the repeated application of Line 7 requires to efficiently store the sets $\mathcal{T}_u$. The used data structure keeps the elements in $\bigcup_{u \in V} \mathcal{T}_u$ sorted (see Section 4.1). This hidden sorting in Lines 8 and 12 leads to a complexity in $O(\log(\Delta S))$ when inserting or removing a vertex.

Therefore, in each of the at most ΔS iterations of the while loop, the time complexity is upper-bounded by $O(\log(\Delta S))$ (the sorting time) plus the time needed to execute the functions `IsNotDominated` and `InsertAndClean`.

For $\mathbf{d = 2}$, the proof is essentially the same as in [Han80]. Since in MC DIJKSTRA the path P is lexicographically larger than any element in $\mathcal{S}$, the function `IsNotDominated`$(P, \mathcal{S})$ can be computed in constant time with the algorithm in [KLP75], instead of time $O(\log(\mu S))$ by using a tree as proposed in [Han80]. However, the function `InsertAndClean`$(P, \mathcal{T})$ has an amortized complexity of $O(\log |\mathcal{T}|)$ to keep the structure sorted, amortized since it may remove a lot of paths during one call but a path can be removed only once. Anyway, the complexity in this case is dominated by the sorting time, leading to the overall complexity $O(\Delta S \log(\Delta S))$.

For $\mathbf{d = 3}$, using the algorithm proposed in [KLP75] and the same reasoning as in the $d = 2$ case, the functions `IsNotDominated` and `InsertAndClean` can be computed in logarithmic time, leading to the same overall complexity as in the case $d = 2$.

For $\mathbf{d > 3}$, we extend the proof for $\mu = 1$ (simple graph) given in [BDH17]: the dominance relation of the current path is iteratively tested with each element of the sets $\mathcal{S}_u$ and $\mathcal{T}_u$ for some u. The latter being upper-bounded by μS, we obtain the overall time complexity in $O(\mu \Delta S^2)$.

□

3.1.3 Example

Consider the graph of Figure 3.1 and execute MC DIJKSTRA on this graph with the vertex A as source. The content of the data structures $\mathcal{T}$ and $\mathcal{S}$ is described in Table 3.1. For $\mathcal{T}$, the costs are prepend by the name of the destination of the path.

Thus, $B(1,3)$ represents a path from A to B and of cost $(1,3)$. The first steps are as follows:

1. The initialization inserts the path of cost $(0,0)$ in the priority queue $\mathcal{T}$.

2. This path is extracted from $\mathcal{T}$ since it is the only one inside. Then, it is extended by the arcs going from A to B and C. These two extensions are placed in $\mathcal{T}$.

3. Then the path represented as $B(1,3)$ is extracted from $\mathcal{T}$ since it is the minimum in the lexicographic order. It is then extended by the only arc leaving B, which gives the path represented as $D(2,5)$. This path is placed in $\mathcal{T}$.

The same process is iterated for the following steps. Notice that at step 4, a path $E(6,5)$ is added to $\mathcal{T}$ but it is removed at step 8 when a path that dominates it is inserted into $\mathcal{T}$.

3.1.4 Termination criterion

MC Dijkstra is designed to answer one-to-all requests. However, for a one-to-one query (i.e. for computing the Pareto set between a pair of vertices), this algorithm can waste time on useless computations, especially whenever the destination vertex is not far from the source.

To tackle this issue, a test to remove paths going too far from the destination can be added as in [MDLC10]: when checking if a path ending at any vertex is dominated, it can also be compared with paths going to the destination. If its cost is dominated, then it is useless to extend it. Indeed, any extension of this path ending at the destination is dominated since an extension cannot decrease any coordinate of its cost. Taking C as the unique destination of interest in the previous example, the computation would stop in 4 steps.

3.2 Meta Rank

In MC Dijkstra, a critical operation is to determine whether a path is dominated by other known paths. We present a review of known methods in Section 2.2 that can be useful in this context. Given a set of costs, these methods remove dominated ones from it. Thus, if the whole set of paths to process is known at the beginning, these algorithms can be used. Unfortunately, MC Dijkstra process path one by one. Therefore, the function `IsNotDominated()` consider only one path at the time.

If $d \leq 3$, it is possible to adapt the methods from Section 2.2, which gives a $O(\Delta S \log(\Delta S))$ time complexity for MC Dijkstra (Proposition 13). However, these efficient methods are not suitable in dimensions larger than 3. Indeed, if $d > 3$, they are offline, i.e they require to have all the paths in advance. Then, a naive algorithm is used, which implies a quadratic time complexity for MC Dijkstra.

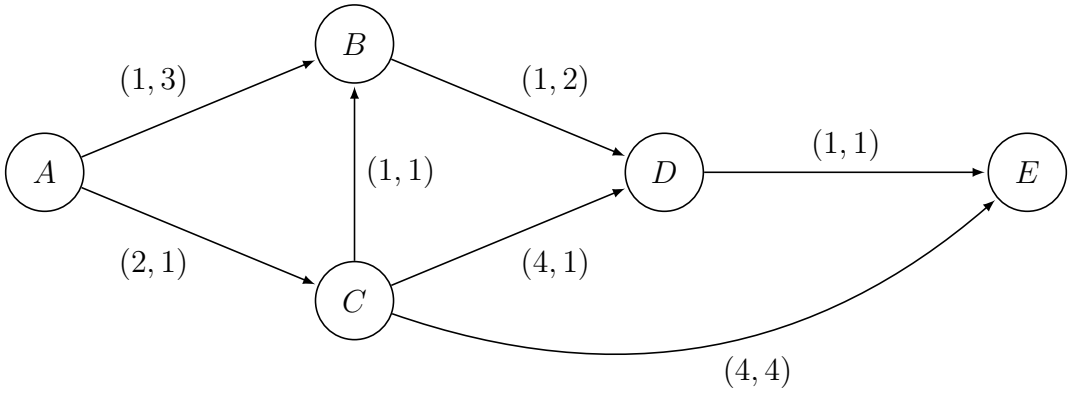


Figure 3.1 – Example graph for MC DIJKSTRA execution.

Step	$\mathcal{T}$	$\mathcal{S}_B$	$\mathcal{S}_C$	$\mathcal{S}_D$	$\mathcal{S}_E$
1	$A(0,0)$				
2	$B(1,3) \mid C(2,1)$				
3	$C(2,1) \mid D(2,5)$	$(1,3)$			
4	$D(2,5) \mid B(3,2) \mid D(6,2) \mid E(6,5)$	$(1,3)$	$(2,1)$		
5	$B(3,2) \mid E(3,6) \mid D(6,2) \mid E(6,5)$	$(1,3)$	$(2,1)$	$(2,5)$	
6	$E(3,6) \mid D(4,4) \mid D(6,2) \mid E(6,5)$	$(1,3) \mid (3,2)$	$(2,1)$	$(2,5)$	
7	$D(4,4) \mid D(6,2) \mid E(6,5)$	$(1,3) \mid (3,2)$	$(2,1)$	$(2,5)$	$(3,6)$
8	$E(5,5) \mid D(6,2)$	$(1,3) \mid (3,2)$	$(2,1)$	$(2,5) \mid (4,4)$	$(3,6)$
9	$D(6,2)$	$(1,3) \mid (3,2)$	$(2,1)$	$(2,5) \mid (4,4)$	$(3,6) \mid (5,5)$
10	$E(7,3)$	$(1,3) \mid (3,2)$	$(2,1)$	$(2,5) \mid (4,4) \mid (6,2)$	$(3,6) \mid (5,5)$
11		$(1,3) \mid (3,2)$	$(2,1)$	$(2,5) \mid (4,4) \mid (6,2)$	$(3,6) \mid (5,5) \mid (7,3)$

Table 3.1 – Content of data structures during the execution of MC DIJKSTRA.

Rank order. Yet, if the paths are processed in subsets, an offline method can be applied to each subset. For this purpose, we choose to process the paths leaving $\mathcal{T}$ by increasing rank order, instead of lexicographic order. We consider all paths from $\mathcal{T}$ having the same rank all together. This allows to process several paths with the same destination at the same time. The increasing rank order enable to keep the nice property that the "smallest" elements of $\mathcal{T}$ incorporated in $\mathcal{S}$ cannot be dominated by paths that are discovered later. This idea to process paths in increasing rank order is already used in [TTLC92] to compute exact Pareto sets.

The paths are leaving $\mathcal{T}$ in increasing rank order, while this is not the case for their entering. Thus, we test dominance when paths are leaving $\mathcal{T}$ rather than when they enter it, contrary to MC DIJKSTRA. Furthermore, we may take advantage of this dominance pruning step by group to also remove some optimal paths in order to output a smaller approximated Pareto set.

3.2.1 Algorithm

In order to implement this versatility, we propose a meta-algorithm META RANK (see Algorithm 6) which uses a blackbox function called `Sample`.

Definition 14 (Sample function)**.** *On input* $(\mathcal{R}, \mathcal{S}_v, \varepsilon)$, `Sample` *must output a subset of* $\mathcal{R}$*. If this function simply removes paths dominated by permanent solutions from* $\mathcal{S}_v$*,* META RANK *solves the Exact Multicriteria Shortest Path Problem. In the following, additional properties on* `Sample` *are defined in order to ensure that* META RANK *solves the* $(1+\varepsilon)$*-approximated Multicriteria Shortest Path Problem. Later on, instantiations of* `Sample` *are provided.*

We note $C_{\texttt{Sample}}(n, S_\varepsilon, \Delta, \Lambda)$ *the complexity of the repeated usage of* `Sample` *during* META RANK*.*

At each step, for each vertex v, META RANK selects from $\mathcal{T}$ the set of paths $\mathcal{R}$ of minimum rank. Some paths from $\mathcal{R}$ are extracted using `Sample` function, and inserted in $\mathcal{S}_v$. Those paths are also extended by any arc starting from v and the extensions are inserted in $\mathcal{T}$.

3.2.2 Data structures

We provide details about the chosen data structures. For a better legibility, we introduce the notations $\mathcal{T}^{(r)}$ (resp. $\mathcal{T}_u^{(r)}$) as the subset of $\mathcal{T}$ (resp. $\mathcal{T}_u$) of paths having a rank r.

- The set $\mathcal{T}$ is a priority queue (Section A.3) and its elements are the sets $\mathcal{T}^{(r)}$. The priority is given by r (the smaller r, the higher priority). We use a strict Fibonacci heap, guaranteeing:
  - a constant time complexity for an insertion and to find the minimum,
  - a $O(\log \Lambda)$ complexity to remove the highest priority element.
- For a given rank r, $\mathcal{T}^{(r)}$ is an array. A unique identifier in $[\![0, n-1]\!]$ is given to each vertex. If the identifier of u is i_u, $\mathcal{T}^{(r)}[i_u] = \mathcal{T}_u^{(r)}$, guaranteeing a constant *worst-case* time complexity for accessing or removing a $\mathcal{T}_u^{(r)}$ set.

Input: Graph $\mathcal{G} = (V, A, w)$ with V the vertices, A the arcs, w the weight function, $s \in V$ the source vertex

Output: Sets $\mathcal{S}_v$ for every vertex v

1 **begin** Initialization
2 **foreach** $u \in V$ **do**
3 $\mathcal{S}_u \leftarrow \emptyset$
4 $\mathcal{T}_u \leftarrow \emptyset$
5 $\mathcal{T}_s \leftarrow \{\text{empty path from } s \text{ to } s\}$
6 **while** $\bigcup_{u \in V} \mathcal{T}_u \neq \emptyset$ **do**
7 let r be the minimum rank in $\bigcup_{u \in V} \mathcal{T}_u$
8 **foreach** $v \in V$ **do**
9 let $\mathcal{R}$ be the paths of rank r in $\mathcal{T}_v$
10 $\mathcal{R}' \leftarrow \texttt{Sample}(\mathcal{R}, \mathcal{S}_v, \varepsilon)$
11 $\mathcal{S}_v \leftarrow \mathcal{S}_v \cup \mathcal{R}'$
12 $\mathcal{T}_v \leftarrow \mathcal{T}_v \setminus \mathcal{R}$
13 **foreach** $P \in \mathcal{R}'$ **do**
14 **foreach** $(v, w) \in A$ **do**
15 $\mathcal{T}_w \leftarrow \mathcal{T}_w \cup \{P \cdot (v, w)\}$

Algorithm 6: META RANK overview

- The sets $\mathcal{T}_u^{(r)}$ are represented as linked lists in order to obtain a constant time insertion.
- $\mathcal{S}$ is also an array and the sets $\mathcal{S}_v$ are linked lists.

Remark. *Although the implementation of the sets $\mathcal{T}^{(r)}$ is interesting from a theoretical point of view, a hash table would be more relevant in practice for memory purposes, since $\mathcal{T}$ may contain only a small part of V at the same time. This choice would only guarantee a constant* mean *time complexity. A key would be a vertex and the associated value to a key u would be $\mathcal{T}_u^{(r)}$.*

Later, variants for the representation of the sets $\mathcal{S}_u$ will be used:

Arrays: the set of costs that the optimal paths can have is partitioned in a finite number of parts. Each $\mathcal{S}_u$ is an array, and each cell corresponds to a part. Any path whose cost is in a given part, is stored in the corresponding cell. A cell can contain a bounded number of paths, meaning that some paths may not be stored. This variant will be presented more precisely in Section 3.4.

Trees : it may be useful to allow an efficient search in $\mathcal{S}$ (Chapter 4). The linked lists representing the $\mathcal{S}_u$ will be replaced by balanced binary trees (Section A.2) in order to guarantee insertions and searches in logarithmic time. This worsens the complexity of the line 11. The second complexity announced by the Theorem 15 is then no longer correct. The first one is still true, so META RANK time complexity is $C_{\texttt{Sample}}(n, S_\varepsilon, \Delta, \Lambda) + O(\Delta S_\varepsilon \log(\Delta S_\varepsilon))$ in this case.

3.2.3 Complexities

Given the previously described data structures, the following theorem gives the complexity of Meta Rank, depending on `Sample`'s one.

Theorem 15. *Let S_ε be the size of* Meta Rank*'s output. Then* Meta Rank *time complexity is:*

- $C_{\texttt{Sample}}(n, S_\varepsilon, \Delta, \Lambda)+O(\Delta S_\varepsilon \log(\Delta S_\varepsilon))$*, or more precisely:* $C_{\texttt{Sample}}(n, S_\varepsilon, \Delta, \Lambda)+ O(\Delta S_\varepsilon + \Lambda \log(\Lambda))$.
- $C_{\texttt{Sample}}(n, S_\varepsilon, \Delta, \Lambda) + O(\Delta n(nC)^{d-1} \log(\Delta nC))$ *if the weights are in* $[\![1, C]\!]$.

Proof. With the given data structures,

- $\bigcup_{u \in V} \mathcal{S}_v$ is only concerned by insertions and its size at the end is S_ε. Thus, there is exactly S_ε insertions and the repetition of line 11 has an overall $O(S_\varepsilon)$ complexity.
- As stated, the data structure used for $\mathcal{T}$ allows to factorize the deletion of line 12 for each vertex, i.e. for each loop of the line 8. Since Λ is the number of rank values of the explored paths, this deletion is in $O(\log \Lambda)$. Thus, the repetition of Line 12 is in $O(\Lambda \log \Lambda)$. For legibility reason, the following inequality is used : $\Lambda \leq \Delta S_\varepsilon$. Then, we obtain that the repetition of this line is in $O(\Delta S_\varepsilon \log(\Delta S_\varepsilon))$.
- Line 9 consists in finding the path of minimum rank and is therefore in $O(1)$. Thus its repetition is in $O(\Lambda)$.
- The repetition of the loop from line 13 to line 15 has an overall complexity of $O(\Delta S_\varepsilon)$ since the number of added path in some T_w is upper-bounded by ΔS_ε.

□

We list in the Table 3.2 the main $C_{\texttt{Sample}}(n, S_\varepsilon, \Delta, \Lambda)$ complexities presented later.

3.3 Sample functions for exact computations

3.3.1 Bucket

First of all, we instantiate Meta Rank (Alg. 6) to compute exact Pareto sets, that is to solve the Exact Multicriteria Shortest Path Problem. For this purpose, we define the so-called *Exact property* that the `Sample` function should satisfy. In this section, the complexities will be given for $d > 2$. For $d = 2$, the complexities are the same as those given for $d = 3$.

Definition 16 (Exact property)**.** *A function* `Sample` *outputting $\mathcal{R}' \subseteq \mathcal{R}$ on input $(\mathcal{R}, \mathcal{S}, \varepsilon)$ satisfies the* Exact property *if $\mathcal{S} \cup \mathcal{R}'$ is a Pareto set of $\mathcal{S} \cup \mathcal{R}$.*

Remark. *The fact that* Meta Rank *with any* `Sample` *function satisfying the* Exact property *does solve the Exact Multicriteria Shortest Path Problem will be proved in Section 3.5.1 (this is an immediate consequence of Proposition 23 and Theorem 24).*

We propose an algorithm for `Sample` verifying the Exact property.

`Sample` function used	$C_{\texttt{Sample}}(n, S_\varepsilon, \Delta, \Lambda)$ in $O(\cdot)$	Ref.
Sample Bucket ($d > 2$)	$S \cdot \min\left\{(\Delta + \Lambda)\log^{d-2}(S_\varepsilon), \Delta S_\varepsilon\right\}$	Section 3.3
Naive Sample Part	ΔS_ε	Section 3.4.1
Naive Sample Set	ΔS_ε^2	Section 3.4.2
Sample Sector	$(\Delta S_\varepsilon)^2$	Section 3.5.2
Small Sample Sector	$\Delta\mathcal{S}_\varepsilon \log^{d-1} \Delta\mathcal{S}_\varepsilon$	Section 3.5.3
Sample Frame	$\Delta S_\varepsilon \log(\Delta S_\varepsilon)$	Section 4.2.3
Theta Sample Sector	$(\Delta S_\varepsilon)^2 \log(\Delta S_\varepsilon) \log\log(\Delta S_\varepsilon)$	Section 6.4
Dominating Sample Sector	$(\Delta S_\varepsilon)^4$	Section 7.3

Table 3.2 – Overall sample complexities.

Sample Bucket. For a given destination u, the first argument $\mathcal{R}$ of `Sample` will be called a *bucket*, being the set of paths of minimum rank and same destination u in $\mathcal{T} = \bigcup_{v \in V} \mathcal{T}_v$. `Sample` has to remove the paths from $\mathcal{R}$ that are dominated by the final set of paths $\mathcal{S}_u$ (which is the second argument of `Sample`). It outputs $\mathcal{R}'$, so that Meta Rank adds the set of remaining paths $\mathcal{R}'$ in $\mathcal{S}_u$. We run in parallel two alternatives to remove dominated paths and stop as soon as one of the two algorithm ends:

1. the naive algorithm, comparing each path from $\mathcal{R}$ with each one of $\mathcal{S}_u$;
2. a Pareto set computation parametrized by d using divide-and-conquer techniques [KS85].

Proposition 17. Sample Bucket *satisfies the Exact property when $\mathcal{R}$ and $\mathcal{S}$ are both Pareto sets such that any path of $\mathcal{R}$ has a larger rank than any path of $\mathcal{S}$.*

Proof. Let P and Q be two paths. If $\texttt{rank}(P) < \texttt{rank}(Q)$, then it is impossible that Q dominates P. Therefore, no path from $\mathcal{R}$ could dominate a path from $\mathcal{S}$. Therefore, Sample Bucket does compute $\mathcal{R}'$ such that $\mathcal{S} \cup \mathcal{R}'$ is a Pareto set of $\mathcal{S} \cup \mathcal{R}$. □

Definition 18 (Bucket algorithm)**.** *The* Bucket *algorithm is the* Meta Rank *algorithm with* Sample Bucket *as the* `Sample` *function.*

Theorem 19. Bucket *has a $O\left(S \cdot \min\left\{(\Delta + \Lambda)\log^{d-2}(S), \Delta S\right\} + \Lambda \log \Lambda\right)$ time complexity.*

Proof. We want to choose the best algorithm to run `Sample` for each bucket, but $|\mathcal{R}'|$ is a priori unknown. A way to circumvent this problem is to start both algorithms in parallel. When one of them stops, we stop also the other and ignore it. The obtained complexity is the double of the best algorithm's complexity. The second algorithm used is the one from [KS85] with complexity $O\left((|\mathcal{R}| + |\mathcal{S}_u|)\log^{d-2}(|\mathcal{R}'| + |\mathcal{S}_u|)\right)$.

Recall that a bucket contains only paths having the same destination and the same rank. For each pair of rank and destination, at most one bucket is considered by `Sample` during Bucket execution since the ranks are considered in increasing order and that an extension of a path has a higher rank than it.

The number of paths being in $\mathcal{T}$ at some point is at most ΔS since any path in $\mathcal{T}$ is necessarily an extension of a path inserted in $\mathcal{S}$. In Bucket, if `Sample` is processing the bucket of rank r and destination u, we note:

- T_u^r the size of the bucket, which is the size of the first parameter of `Sample`,
- S_u^r the size of the output of `Sample`,
- $S_u^{<r}$ the number of optimal solutions known at this stage of the algorithm, which are the Pareto paths of ranks *inferior* to r and of destination u. It is the size of the second parameter of `Sample`.

First, we have $S_u^r \leq S$. Then we compute $C_{\texttt{Sample}}(n, S, \Delta, \Lambda)$, the cumulative complexity of Sample Bucket functions during Meta Rank:

1. (Naive algorithm): for a rank r and a destination u, the naive algorithm costs $T_u^r \cdot S_u^{<r}$. Notice that it is not necessary to compare paths from $\mathcal{R}$ with each other since they have the same rank, and thus are incomparable. Thus, if we sum over each destination vertex u and each rank r, we obtain an overall complexity which is a big O of:
$$\sum_{r,u} T_u^r \cdot S_u^{<r} \leq \sum_{r,u} T_u^r \cdot S \leq \Delta \cdot S^2$$
2. ([KS85] methods) for a rank r and a destination u, the complexity is a big O of $(T_u^r + S_u^{<r}) \cdot \log^{d-2}(S_u^r + S_u^{<r})$. Then, Bucket complexity is a big O of:
$$\begin{aligned}\sum_{r,u} (T_u^r + S_u^{<r}) \log^{d-2}(S_u^r + S_u^{<r}) &\leq \sum_{r,u} (T_u^r + S_u) \log^{d-2}(S) \\ &\leq (\Delta S + \Lambda \cdot S) \log^{d-2}(S) \\ &\leq (\Delta + \Lambda) S \log^{d-2}(S)\end{aligned}$$

Then, $C_{\texttt{Sample}}(n, S, \Delta, \Lambda) = O\left(S \cdot \min\left\{(\Delta + \Lambda)\log^{d-2}(S), \Delta S\right\}\right)$. Using Theorem 15 gives the claimed complexity. □

3.3.2 Dijkstra Post

In order to get a better comparison between MC Dijkstra algorithm (Alg. 5) and Frame algorithm (Def. 42), we introduce an in-between algorithm that we call Dijkstra Post algorithm. It consists in using the Meta Rank algorithm (Alg. 6) with a `Sample` function satisfying the Exact property with the following modification: instead of choosing in the sets $\mathcal{T}_u$ the paths of minimum rank, the lexicographically minimum path is chosen, like in MC Dijkstra.

An advantage of this Dijkstra Post algorithm is its simplicity, as it will be explained in Section 4.1, while a major flaw is that more paths may be added to the sets $\mathcal{T}_u$, since the pruning is only done when the paths leave these sets.

3.4 Naive approximated solutions

We start by studying simple natural ideas: when we discover "too many" "close" paths, we keep only a subset of them. We will clarify what "too many" and "close" mean. Two similar methods are presented in Sections 3.4.1 and 3.4.2. These are very efficient but, as it will be explained in Section 3.4.3, they do not guarantee that the output is a $(1+\varepsilon)$-approximated Pareto set.

Recall that `Sample` considers only paths having same source and destination. The following presentation is therefore written for paths having the same source and destination. As explained in Section 2.3, if two paths have the same cost, then only one of them is kept. Thus, we will make no difference between a path and its cost.

3.4.1 Naive Sample Part

The first idea is to partition $\mathbb{R}_+^d$ so that two paths are in the same part are $(1+\varepsilon)$-covering each other. In MC Dijkstra, for a given vertex, the solution set contains only a limited number of paths within a part. Jacob et al. [Hrn+17] propose to only keep one, the first one. This kind of partition can also be found in [TZ09], used this time with a Bellman-Ford type algorithm.

For the sake of clarity, we restrict to costs superior or equal to 1 on each dimension. More formally, we introduce the following notations:

- for a path $P = (P_1, \ldots, P_d)$, its position is $\texttt{Position}(P) = (\lfloor \log_{1+\varepsilon}(P_i) \rfloor)_{1 \leq i \leq d}$,
- for each dimension i, C_i is the highest weight an arc can have on the i-th dimension,
- $C = \max_i C_i$.

A part is the set of path sharing the same position.

This partition is depicted in Figure 3.2, with $\varepsilon = 1$ and the cost superior or equal to 1. The set of blue dots represents a Pareto set $\mathcal{P}$. The crosses are a $(1+\varepsilon)$-Pareto set of $\mathcal{P}$, since there is one per zone. Notice that the green one is optional. The parts positions are indicated in blue.

As announced at the end of Section 3.2, each $\mathcal{S}_u$ is here a d-dimensional array, of size $\lfloor \log_{1+\varepsilon}(nC_1) \rfloor \times \ldots \times \lfloor \log_{1+\varepsilon}(nC_d) \rfloor$. In the cell of index $(x_i)_{1 \leq i \leq d}$, the array contains the paths P such that $\texttt{Position}(P) = (x_i)_i$.

Then, at each call of `Sample`, *for each cell*, the paths of the current rank are either inserted in this array, or pruned. Several criteria are possible:

- The k first paths seen in a given cell are kept, with k fixed. When $k > 1$, for each new path, it is possible to test if it is dominated by those already in place in the same cell. On the other hand, the opposite test is not mandatory: the paths being processed by increasing ranks, the k first seen cannot be dominated by the following ones.
- Another method is to keep only one path per coordinate, the one that minimizes it in that cell.

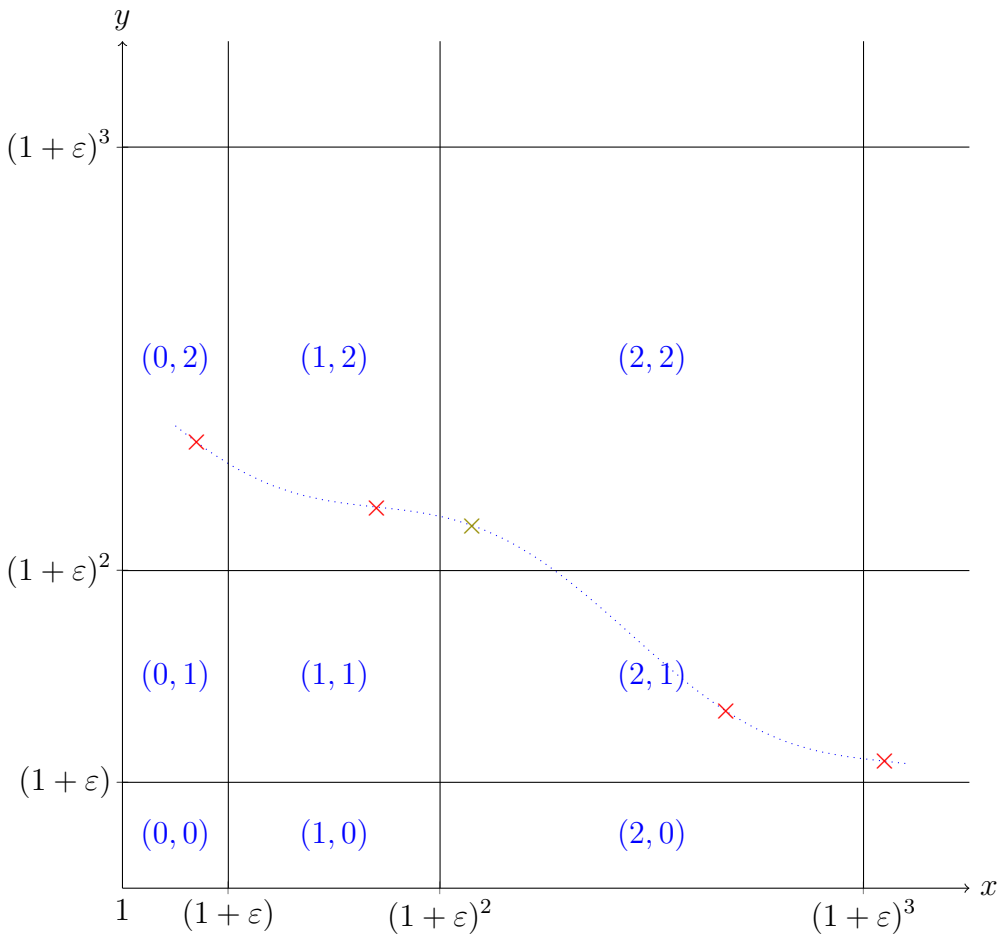


Figure 3.2 – Partition of the plan with $\varepsilon = 1$.

Algorithm 7 is the first method with $k = 1$. Line 2 insures that the considered path is a simple path. In order to adapt to other methods, you just have to adapt the test line 4 and the following insertion line.

```
   Input: R, S set of paths, ε > 0
1  for P ∈ R do
2      if P ≤_Dom (nC_i)_{1≤i≤d} then // the i-th coordinate is bounded by
         nC_i
3          pos ← Position(P)
4          if S[pos] = ∅ then
5              S[pos] ← P
```

Algorithm 7: Naive Sample Part

Theorem 20. *The complexity of* Meta Rank *using* Naive Sample Part *as* `Sample` *is in* $O\left(\Delta n \log^d_{1+\varepsilon}(nC) \log(\Delta n \log_{1+\varepsilon}(nC))\right)$. *The output-sensitive complexity is in* $O(\Delta S_\varepsilon \log(\Delta S_\varepsilon))$.

Proof. Algorithm 7 is linear in the size of $\mathcal{R}$. Thus, by using this algorithm as `Sample` in Meta Rank, $C_{\texttt{Sample}}(n, S_\varepsilon, \Delta, \Lambda) = O(\Delta S_\varepsilon)$. The announced output-sensitive complexity is found according to Theorem 15. Then, for each vertex u, S_u is an array of size $\log^d_{1+\varepsilon}(nC)$ and there is at most one path per cell. Thus, $S_\varepsilon \leq n \log^d_{1+\varepsilon}(nC)$. □

Remark. *We can also be interested in an additive split. Indeed, a user may find the absolute difference between two paths much more relevant. However, we end up with* $\left(\frac{nC^{max}}{1+\varepsilon}\right)^d$ *parts, a quantity potentially prohibitive.*

Domination on the first dimension. In TZ, the Algorithm of Tsaggouris et al. [TZ09], the exponent is $d-1$ instead of d since the arrays have one dimension less. Indeed, there are $\log_{1+\varepsilon}(nC)$ cells which have the same $d-1$ last coordinates: if we fix these coordinates, the first one can still take $\log_{1+\varepsilon}(nC)$ different values. However, we are interested in only one of these cells, the one whose first coordinate is the smallest and which is not empty. Indeed, the path of this cell covers all the other cells sharing the same $d-1$ last dimensions.

In TZ, the algorithm is of MLC type, i.e. a path already in S_u can be improved, i.e. replaced by a better path discovered later. However, our complexity analysis requires that the algorithm be of the MLS type, i.e. once a path is placed in S_u, it cannot be removed. Adding this method to Meta Rank is therefore not possible since we discover the paths by ranks: for two paths P and Q, we can have:

$$\begin{cases} P_1 > Q_1 \\ \forall i \in [\![1, d]\!], \lfloor \log_{1+\varepsilon}(P_i) \rfloor = \lfloor \log_{1+\varepsilon}(P_i) \rfloor \\ \texttt{rank}(P) < \texttt{rank}(Q) \end{cases}$$

Using the method of TZ, Meta Rank first processes P but replaces it by Q. A solution would be to replace the order by rank of Meta Rank by the one of

MC Dijkstra, i.e. the lexicographic order. Then, the time complexity becomes in $O\left(\Delta n \log_{1+\varepsilon}^{d-1}(nC) \log(\Delta n \log_{1+\varepsilon}(nC))\right)$.

Domination in general. These changes allow to avoid filling two cells that share the same $d-1$ last coordinates and that dominate each other. But one must beware of the fact that our algorithm may well fill cells that are dominated by others already filled. We have suggested a small test of domination within a cell if we want to keep k paths per cell, with $k > 1$. But no domination test is performed between cells. And these tests are costly since they square the complexity if we use a naive method of domination testing, comparing any new path with those already known. More precisely, the complexity of dealing with domination test is in $O\left(\Delta n \log_{1+\varepsilon}^{2d}(nC)\right)$, multiplying the number of seen paths with the number of kept paths for a given vertex. This complexity is also the overall complexity, dominating the complexity stated in Theorem 20.

In [BC19], the authors propose to use the offline methods of Kung et al. [KLP75] but it seems that they are used for each added path, which worsens the complexity by adding a logarithmic factor to the complexity obtained with the naive test method. The authors then point out that in practice, these offline methods are only profitable for a very large number of paths (at least 10^5) and therefore decide to fall back on the naive method for their experiments.

3.4.2 Naive Sample Set

With the previous algorithm, two paths very close but in different parts will be kept. We remove a path when another one $(1+\varepsilon)$-covers it and is in the same part. We wonder what happens if we decide to remove the constraint of being in the same part. Now, as soon as we discover a $(1+\varepsilon)$-covered path, we remove it.

The solution sets $\mathcal{S}_u$ are represented as linked lists, in order to efficiently iterate over it. The condition to remove a path is more complex to check here. We are no longer restricted to comparing it with a bounded number of paths, or even one, that are in the same part. We may have to compare it with all the kept paths, as the pseudo-code of the Algorithm 8 suggests.

Theorem 21. Naive Sample Set *(Algorithm 8) has a $O(|\mathcal{R}| \cdot |\mathcal{S}|)$ time complexity. Then, the time complexity of* Meta Rank *using this algorithm as* `Sample` *is in $O\left(\Delta \log_{1+\varepsilon}^{2d}(nC)\right)$.*

Proof. Naive Sample Set potentially compares each path of $\mathcal{R}$ with each path of $\mathcal{S}$, with a constant time test. As for its use in Meta Rank, the number of paths kept is bounded by $\log_{1+\varepsilon}^{d}(nC)$ since we keep at most one path per cell. The paths transiting in $\mathcal{R}$ are extensions of kept paths, so it is at most the number of kept paths, multiplied by Δ. □

In order to improve the complexity, one can place the solutions in an array as in Naive Sample Part, and then compare any new path with those in its part and in the neighboring parts, which bounds the number of cells in the array to be tested by $2^d - 1$.

Input: $\mathcal{R}, \mathcal{S}$ sets of paths, $\varepsilon > 0$
Output: $\mathcal{R}'$ set of paths

$\mathcal{R}' \leftarrow \emptyset$
for $P \in R$ **do**
  if $P \leq_{Dom} (nC_i)_{1 \leq i \leq d}$ **then** // the i-th coordinate is bounded by nC_i
    $Covered(P) \leftarrow False$
    for $Q \in S$ **do**
      if $Q \leq_{Dom} (1+\varepsilon)P$ **then**
        $Covered(P) \leftarrow True$
        break ; // P is $(1+\varepsilon)$-covered
    if $Covered(P) = False$ **then** // P is $(1+\varepsilon)$-covered
      $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{P\}$
    return $\mathcal{R}'$

Algorithm 8: Naive Sample Set

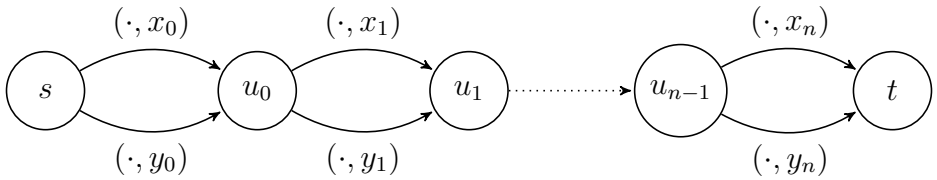


Figure 3.3 – Study of the gap between two paths.

3.4.3 Fast heuristics.

These algorithms have nice complexities. However, they do not guarantee to output a $(1+\varepsilon)$-approximation. To convince ourselves of this fact, we start by studying what gap can appear between two paths, when the prefix of the first one is responsible for the pruning of the second one. More precisely, we consider a sequence of $n+1$ pairs of arcs represented by the graph of Figure 3.3. The weights are here in dimension 2 and we represent only the second dimension. We use here the set version, taking the vertex s as source.

We suppose the following processing: when discovering the paths that arrive at u_i, we keep only those that go above, of weight $(\cdot, x_i)$. We notice that such an unfolding implies that from s to any vertex u_i, the algorithm discovers only two paths. Both arrive at u_{i-1} by using only the above arcs of weight $(\cdot, x_j)$, then branch to go from u_{i-1} to u_i either by going above or below. Only one of the two being then kept (the above one), this reasoning is recursive.

The condition of keeping the above path gives that after k steps:

$$\sum_{i=0}^{k-1} x_i + x_k \leq (1+\varepsilon) \left(\sum_{i=0}^{k-1} x_i + y_k \right)$$

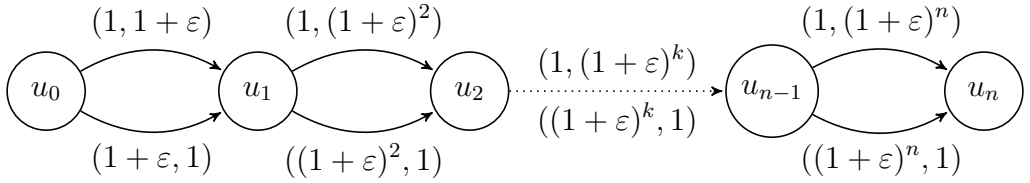


Figure 3.4 – Counter-example of the $(1+\varepsilon)$-approximation guarantee.

i.e.:

$$x_k \leq \epsilon \sum_{i=0}^{k-1} x_i + (1+\varepsilon) y_k$$

By induction:

$$x_k \leq \sum_{i=0}^{k-1} \epsilon (1+\varepsilon)^{k-i} y_i + (1+\varepsilon) y_k$$

and:

$$\sum_{k=0}^{n} x_k \leq \sum_{k=0}^{n} (1+\varepsilon)^{n-k+1} y_k$$

We notice that it is the beginning of the path that is most likely to make lose the $(1+\varepsilon)$-coverage. In other words, the loss in precision can cascade: if a path P_1 is pruned in favor of a path P_2, and later the extension of P_2 is pruned in favor of a path P_3, the extension of P_1 could no longer be $(1+\varepsilon)$-covered by P_3.

Counter-example to $(1+\varepsilon)$-approximation guarantee. We thus consider the counter-example described by the Figure 3.4.

We assume that the first coordinate is privileged, meaning that when considering two paths of same rank, the processing order is the lexicographic order. Without this assumption, it is sufficient to shift the weights very slightly to force the order but respect the inequalities we are going to use. This last remark is useful to adapt our counter-example to Algorithm 7. We therefore end up keeping only the path passing through the above arc and ignoring the bottom one. Indeed, let assume that from s to u_k, only the path going through above arcs has been kept. Therefore, this path has a cost of: $\left(k, \sum_{1 \leq i \leq k} (1+\varepsilon)^i\right)$.

Its extensions to u_{k+1} are the following (seen in lexicographic order since they have same rank):

- $P^{(k)}$ of cost $\left(k+1, \sum_{1 \leq i \leq k+1} (1+\varepsilon)^i\right)$,
- $Q^{(k)}$ of cost $\left(k + (1+\varepsilon)^{k+1}, \sum_{0 \leq i \leq k} (1+\varepsilon)^i\right)$.

On the first coordinate, $Q^{(k)}$ is larger than $P^{(k)}$. On the second one, since $(1+\varepsilon)\cdot\sum_{0\leq i\leq k}(1+\varepsilon)^i = \sum_{1\leq i\leq k+1}(1+\varepsilon)^i$, then $Q^{(k)}$ is at a factor $1+\varepsilon$ from $P^{(k)}$. Hence, $Q^{(k)}$ is $(1+\varepsilon)$ covered by $P^{(k)}$. Thus, $Q^{(k)}$ is pruned. Recursively, we obtain that the only path going from u_0 to u_n that is be kept is the one using only the above arcs, noted $P^{(n)}$. We note $R^{(n)}$ the path using only the arcs from below. We have:

$$\begin{aligned} R_2^{(n)} &= n \\ P_2^{(n)} &= \sum_{1\leq i\leq n}(1+\varepsilon)^i = \frac{1+\varepsilon}{\epsilon}\left[(1+\varepsilon)^n-1\right] \\ &= \frac{1+\varepsilon}{n\epsilon}\left[(1+\varepsilon)^n-1\right]\cdot R_2^{(n)} \end{aligned}$$

However:

$$\frac{(1+\varepsilon)^n-1}{n} \xrightarrow[n\to\infty]{} +\infty$$

Thus, for n large enough, $R^{(n)}$ is not $(1+\varepsilon)$-covered by $P^{(n)}$.

A simple modification. The same kind of study was conducted in [BC19] which leads the authors to the same solution as in TZ. They use much smaller parts, with a factor: $(1+\varepsilon)^{\frac{1}{n}}$. This way, the error can be set to the power n without exceeding $(1+\varepsilon)$. However, the number of parts and their hop numbers are prohibitive. In practice, all paths will be in separate parts. Combined with the absence of domination test, TZ is completely unpractical despite its nice complexity. In [BDH17], the authors tackle this issue by replacing the exponent by a reasonable value, chosen by hand. But this value is arbitrary and does not guarantee that we obtain a $(1+\varepsilon)$-approximation.

Conclusion. We have therefore presented here two relatively natural implementations of Meta Rank. These algorithms have very interesting complexities. The first one, using Naive Sample Part as a `Sample` function is however unpractical: it does not remove all dominated paths. Thus, it is necessary to add this operation which increases the complexity of the algorithm. Another problem is that this algorithm, as well as its variant using Naive Sample Set as a `Sample` function, do not guarantee to output a $(1+\varepsilon)$-approximated Pareto set.

3.5 Solutions with sectors

3.5.1 Elimination criterion

It turns out that the framework provided by Algorithm Meta Rank (Alg. 6) can compute $(1+\varepsilon)$-Pareto paths, by defining an appropriate `Sample` function. To guarantee Algorithm Meta Rank to output a $(1+\varepsilon)$-approximated Pareto set, we require the following ε-weak framing property.

Definition 22 (ε-weak framing property)**.** *A function* `Sample` *outputting* $\mathcal{R}'\subseteq\mathcal{R}$ *on input* $(\mathcal{R},\mathcal{S},\varepsilon)$ *satisfies the* ε-weak framing property *if, for every path* $P\in\mathcal{R}\backslash\mathcal{R}'$*,*

there exist d representative paths $Q^{(1)}, \ldots, Q^{(d)}$ in $\mathcal{S} \cup \mathcal{R}'$ such that, for every i,

$$\begin{cases} Q_i^{(i)} \leq (1+\varepsilon) P_i \\ \forall j \neq i, Q_j^{(i)} \leq P_j \end{cases}$$

Furthermore, $\mathcal{S} \cup \mathcal{R}'$ is a set of incomparable paths.

Proposition 23. *The 0-weak framing property is equivalent to the Exact property.*

Proof. When $\varepsilon = 0$, the first part of the definition of the ε-weak framing property is equivalent to the property that every path in $\mathcal{R} \setminus \mathcal{R}'$ is dominated by a path in $\mathcal{S} \cup \mathcal{R}'$. Combined with the second part of the definition, we obtain the equivalence. □

Notice that if $P \in \mathcal{R}$ is dominated by $Q \in \mathcal{S}$, it is sufficient to set $Q^{(i)} = Q$ for all i. Overall, this ε-weak property guarantees that the output of Meta Rank is a $(1+\varepsilon)$-Pareto set.

Theorem 24. *With a function* `Sample` *satisfying the ε-weak framing property,* Meta Rank *algorithm (Alg. 6) solves the $(1+\varepsilon)$-approximate Multicriteria Shortest Path Problem.*

Proof. Let $\mathcal{S}$ be a Pareto set and $\mathcal{S}_a$ be the output of the algorithm. It is sufficient to show that for any path $P \in \mathcal{S}$, there exists a path $Q \in \mathcal{S}_a$ such that P is $(1+\varepsilon)$-covered by Q and $\texttt{rank}(Q) \leq \texttt{rank}(P)$. By contradiction, let $P' \in \mathcal{S}$ be a minimum rank path not $(1+\varepsilon)$-covered by any $Q \in \mathcal{S}_a$ such that $\texttt{rank}(Q) \leq \texttt{rank}(P')$. P' cannot be an empty path since the only one the algorithm can process is the one from the source to itself, and being the first one to leave $\mathcal{T}$, it is inserted in S. Thus, we can write $P' = P \cdot e$, with P a path and e the last arc of P'. P having an inferior rank than $P \cdot e$, there exists a path $Q \in S_a$ $(1+\varepsilon)$-covering P. If P is kept in $\mathcal{S}_a$, then $P \cdot e$ is inserted in $\mathcal{T}$ and is either kept in $\mathcal{S}_a$ or removed because of some representatives. In either cases, it is $(1+\varepsilon)$-covered, which is absurd. Otherwise, P is not kept in $\mathcal{S}_a$ and in particular, $P \neq Q$. Since $\texttt{rank}(Q) \leq \texttt{rank}(P)$, there exists a dimension i such that $Q_i \leq P_i$. Furthermore, $Q \in \mathcal{S}_a$ implies that it is extended and that $Q \cdot e$ is inserted in $\mathcal{T}$. However, Q $(1+\varepsilon)$-covers P, thus $Q \cdot e$ $(1+\varepsilon)$-covers $P \cdot e$ and:

$$\begin{cases} Q_i + e_i \leq P_i + e_i \\ \forall j \neq i, Q_j + e_j \leq (1+\varepsilon)(P_j + e_j) \end{cases}$$

That is why $Q \cdot e$ cannot be in $\mathcal{S}_a$. This means that $Q \cdot e$ is removed because of some representative paths, among which a path $R \in \mathcal{S}_a$, with $\texttt{rank}(R) \leq \texttt{rank}(Q \cdot e)$, that satisfies:

$$\begin{cases} R_i \leq (1+\varepsilon)(Q_i + e_i) \\ \forall j \neq i, R_j \leq Q_j + e_j \end{cases}$$

Then:

$$\begin{cases} R_i \leq (1+\varepsilon)(Q_i + e_i) \leq (1+\varepsilon)(P_i + e_i) \\ \forall j, R_j \leq Q_j + e_j \leq (1+\varepsilon)(P_j + e_j) \end{cases}$$

Which means that $P \cdot e$ is $(1+\varepsilon)$-covered by R. Since R is in $\mathcal{S}_a$, we obtain a contradiction. □

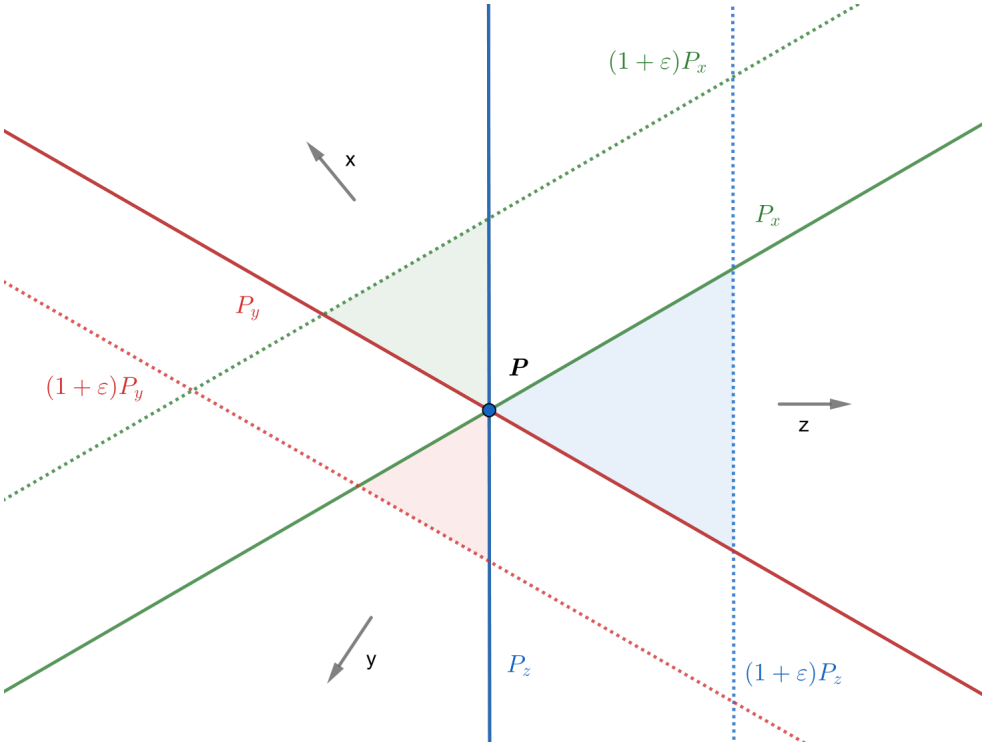


Figure 3.5 – Representation of the three sectors of a path P in 3D.

Two characteristics of `Sample` are of particular interest:

- the time complexity,
- the number of paths the function removes.

Naive greedy algorithms are not efficient for either of these metrics. Thus, we propose a sample algorithm guaranteeing the ε-weak framing property, achieving a good tradeoff for the two characteristics. Given a d-dimensional space, we define d *sectors* for every path P.

Definition 25. *The* i-th sector *of* P *contains every point* Q *with* $Q_j \leq P_j$ *for* $j \neq i$. *Given* ε, *the boolean function* $coverSector(P, Q, i, \varepsilon)$ *is* `True` *if* Q *belongs to the* i*-th sector and* $(1+\varepsilon)$*-covers* P, *that is*

$$coverSector(P, Q, i, \varepsilon) = (Q_i \leq (1+\varepsilon)P_i) \wedge \bigwedge_{j \neq i} (Q_j \leq P_j)$$

In Figure 1.4, the two rectangles represent the incomparable part of the two sectors 2-covering the point B, i.e., the points Q not dominating B such that $coverSector(B, Q, 1, 1)$ is true (for instance C, D and E), and $coverSector(B, Q, 2, 1)$ is true respectively (such as A). For three criteria, Figure 3.5 depicts the three sectors covering a point P, restricted to the plane of paths of rank equal to $\texttt{rank}(P)$. The green zone is the sector for the x dimension, the red one for y and the blue one for z. Notice that their sizes is not the same, since it depends on the coordinate of P in that dimension.

3.5.2 A first version : Sample Sector

We propose a greedy algorithm Sample Sector (Alg. 9), considering each path only once to determine if it is removed. Informally, the criterion is the following: whenever a path P is processed, if it is represented by d paths seen so far, it is removed. A weakness of this method is that we do not have any guarantee on the number of paths kept over the minimum number we could have kept.

Input: $\mathcal{R}, \mathcal{S}$ sets of paths, $\varepsilon > 0$
Output: $\mathcal{R}'$ set of paths
$\mathcal{R}' = \emptyset$
foreach $P \in \mathcal{R}$ **do**
  foreach $i \in [\![1; d]\!]$ **do**
    $covered \leftarrow$ **False**
    foreach $Q \in \mathcal{S} \cup \mathcal{R}'$ **do**
      if $coverSector(P, Q, i, \varepsilon)$ **then** // a representative Q is found
        $covered \leftarrow$ **True**
        break ; // no need to search in this sector anymore
    if $covered =$ **False then** // P is not covered in at least one sector
      $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{P\}$; // then it is kept
      break ; // no need to search for other sectors anymore

Algorithm 9: Sample Sector

Definition 26. *The algorithm* Sector *is the* Meta Rank *algorithm (Alg. 6) using* Sample Sector *(Alg. 9).*

Theorem 27. Sample Sector *satisfies the ε-weak property when $\mathcal{R}$ and $\mathcal{S}$ are both Pareto sets such that any path of $\mathcal{R}$ has a larger rank than any path of $\mathcal{S}$.*

Proof. If a path $P \in \mathcal{R}$ is not added in $\mathcal{R}'$, then during the processing of P, there exists d paths in $\mathcal{R}' \cup \mathcal{S}$ $(1+\varepsilon)$-covering P respectively in each sector. Since no path is removed from R', those representatives are still in $\mathcal{R}' \cup \mathcal{S}$ at the end of Sample Sector. Moreover, paths from R cannot dominate those from S because their ranks are greater, and dominated paths from R are not in R' since those are covered in each sector by a dominating path. □

Theorem 28. *The complexity of* Sector *algorithm (Def. 26) is $O\left((\Delta S_\varepsilon)^2\right)$.*

Proof. The algorithm considers ΔS_ε paths at most. Since every path must be compared to every other one, $C_{\text{Sample Sector}}(n, \mathcal{S}_\varepsilon, \Delta, \Lambda) = O((\Delta S_\varepsilon)^2)$. Thus, using Theorem 15, the overall time complexity is in $O\left((\Delta S_\varepsilon)^2 + \Delta S_\varepsilon \log(\Delta S_\varepsilon)\right)$, which is in $O\left((\Delta S_\varepsilon)^2\right)$. □

Limitations. Let r be a rank, H the hyperplane of rank r and $\mathcal{S}$ the set of paths in H that the algorithm outputs from the priority queue. We notice that for any path $P \in \mathcal{S}$, if we split H in two half-hyperplanes H_1, H_2 such that P is on the frontier of both, then our elimination criterion requires P to have a representative in both H_1 and H_2.

In Sample Sector (Alg. 9), if $\mathcal{R}$ is ordered in lexicographic order, only dominated paths are removed: indeed, at any step, already seen paths in $\mathcal{R}$ are contained in a semi-hyperplane. Thus, if $\mathcal{S}$ is empty, the algorithm would keep every non-dominated path. A simple improvement would be to use a uniformly random order.

Another consequence is that paths that are in the convex hull of S cannot be deleted. Then we can precompute the convex hull $Conv(S)$ of S, consider $Conv(S)$ as kept and apply our algorithm to the remaining paths of $S \backslash Conv(S)$.

If one wants to use this method, we recall the following complexities:

Lemma 29 ([Cha93; Cha96])**.** *Let $d \geq 1$ be a dimension, S a set of points in $\mathbb{R}^d$ and $h = |Conv(S)|$ the output size. The convex hull $Conv(S)$ of S can be computed in:*

- $O(n \log(h))$ *in 2D [Cha96]*
- $O(n \log(n) + n^{\lfloor d/2 \rfloor})$ *in the general case [Cha93]*

For more details on this subject, a review can be found in [Ind04].

3.5.3 With RangeQueries : Small Sample Sector

We propose another algorithm implementing the `Sample` function, Small Sample Sector, in order to have a better output sensitive complexity for Meta Rank. It considers each dimension i independently to compute a set of paths $\mathcal{R}'_i \subseteq \mathcal{R}$ and the output of the algorithm is $\mathcal{R}' = \bigcup_{i=1}^{d} \mathcal{R}'_i$.

Let r be the rank of the paths in $\mathcal{R}$, and let i be a dimension. We partition $\mathcal{R}$ into *strips* $\mathcal{R}_i^{(l)}$, for $l \in [\![0, \lceil \log_{1+\varepsilon} r \rceil + 1]\!]$. $\mathcal{R}_i^{(0)}$ (resp. $\mathcal{R}_i^{(1)}$) contains the paths such that $P_i = 0$ (resp. $P_i = 1$). For $l \geq 2$, $P \in \mathcal{R}$ belongs to $\mathcal{R}_i^{(l)}$ if its i-th coordinate P_i is in $\left((1+\varepsilon)^{l-2}, (1+\varepsilon)^{l-1}\right]$.

Our algorithm Small Sample Sector proceeds as follows:

1. $\mathcal{R} \cup \mathcal{S}$ is first preprocessed to answer quickly range queries.
2. Then, for every path $P \in \mathcal{R}_i^{(l)}$, P is inserted into $\mathcal{R}'_i$ if P is not $(1+\varepsilon)$-covered in its i-th sector by a path of $\mathcal{R} \cup \mathcal{S}$ in the same strip $\mathcal{R}_i^{(l)}$.

The second step can be done using the following range query (Section 2.2.4): `RangeQuery`$([0, P_1] \times \cdots \times [0, P_{i-1}] \times [P_i, (1+\varepsilon)^{l-1}] \times [0, P_{i+1}] \times \cdots \times [0, P_d], \mathcal{R} \cup \mathcal{S})$. For legibility reason, this request will be noted `RangeQuerySector`$(P, i, \mathcal{R} \cup \mathcal{S})$.

In Figure 3.6, the gray z-strip contains only 6 points, the other one in the sector cannot be used to represent P since it is outside the gray zone.

Input: $\mathcal{R}, \mathcal{S}$ sets of paths, $\varepsilon > 0$
Output: $\mathcal{R}'$ set of paths

$\mathcal{R}' = \emptyset$
$r \leftarrow \texttt{rank}(\mathcal{R})$
foreach $i \in [\![1; d]\!]$ **do**
    Set up $\mathcal{R}^{(0)}, \ldots, \mathcal{R}^{(\lceil \log_{1+\varepsilon} r \rceil + 1)}$ to empty sets
    foreach $P \in \mathcal{R}$ **do**
        if $P_i = 0$ **then**
            Add P to $\mathcal{R}^{(0)}$
        else
            Add P to $\mathcal{R}^{(\lceil \log_{1+\varepsilon} P_i \rceil + 1)}$
    foreach $l \in [\![0; \lceil \log_{1+\varepsilon} r \rceil + 1]\!]$ **do**
        $\mathcal{R}'^{(l)} \leftarrow \emptyset$
        foreach $P \in \mathcal{R}^{(l)}$ **do**
            if $\texttt{RangeQuerySector}(P, i, \mathcal{R} \cup \mathcal{S}) = \emptyset$ **then**
                $\mathcal{R}'^{(l)} \leftarrow \mathcal{R}'^{(l)} \cup \{P\}$
        $\mathcal{R}'_i \leftarrow \mathcal{R}'_i \cup \mathcal{R}'^{(l)}$
    $\mathcal{R}' \leftarrow \mathcal{R}' \cup \mathcal{R}'_i$

Algorithm 10: Small Sample Sector

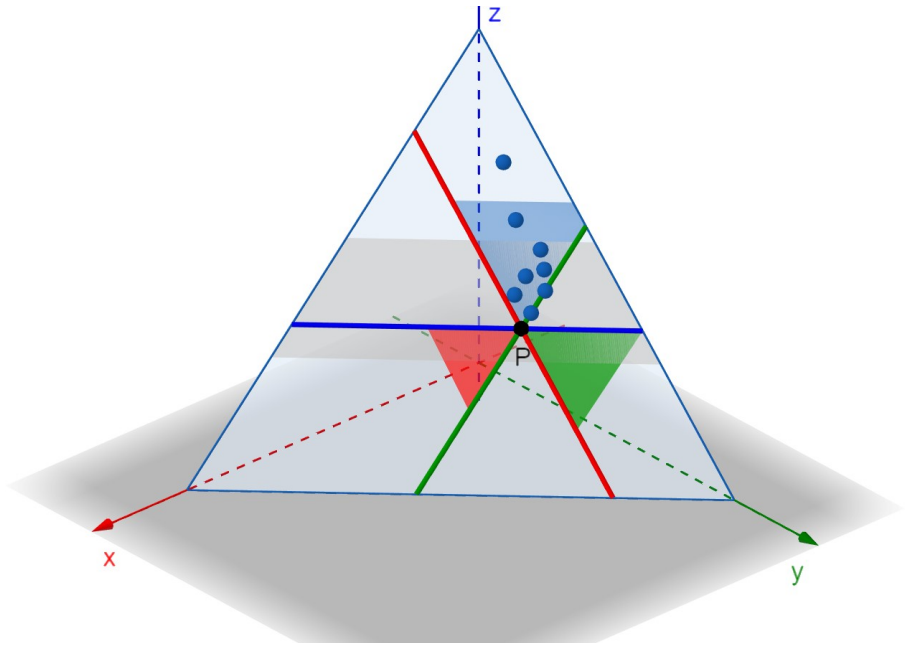


Figure 3.6 – In 3D, the three sectors covering P at distance at most $(1 + \varepsilon)$ are depicted in green, red and blue. Only 6 points are within the gray z-strip of P.

Definition 30. *Algorithm* SSector *is the* Meta Rank *algorithm (Alg. 6) using* Small Sample Sector.

As mentioned in the introduction, SSector solves the $(1+\varepsilon)$-Multicriteria Shortest Path Problem. Combined with Theorem 24, the following theorem confirms that.

Theorem 31. Small Sample Sector *satisfies the ε-weak property when $\mathcal{R}$ and $\mathcal{S}$ are both Pareto sets such that any path of $\mathcal{R}$ has a larger rank than any path of $\mathcal{S}$.*

Proof. In both `Sample` functions, we have to prove that if a path P of rank r has been removed, $\mathcal{S} \cup \mathcal{R}'$ contains d paths guaranteeing the ε-weak framing property. Let us focus on one dimension i.

If the range query returns a non empty set $\mathcal{Q}$ for the P's i-th sector of its strip, we have two cases: (1) the corresponding subspace contains at least a permanent path in $\mathcal{S}$ or (2) only contains paths of same rank. In the first case, we are sure that path P will have a representative path in its i-th sector whereas in the second case, these paths might be not kept in $\mathcal{R}'_i$. This case is not possible since the path in $\mathcal{Q}$ with the highest value for its i-th coordinate is added in $\mathcal{R}'_i$. In both cases, if a path does not belong to $\mathcal{R}'_i$, then there is at least one path in $\mathcal{R}'_i \cup \mathcal{S}$ that $(1+\varepsilon)$-covers P in its i-th sector.

By construction, any path P kept in Small Sample Sector has no representative path in at least one of its sector in the same strip. □

The following theorem states the output-sensitive time complexity of SSector given in Table 1.1, along with the space complexity and the time complexity in the special case where weights are integers. In order to conclude, it is sufficient to compute the sum of the complexities of the Small Sample Sector calls in SSector, and then to use Theorem 15.

Theorem 32. *The time complexity of* SSector *is $O(\Delta S_\varepsilon \log^{d-1}(\Delta S_\varepsilon))$ and the space complexity is $\Theta(\Delta S_\varepsilon \log^{d-1}(\Delta S_\varepsilon))$. If the arc weights are integers, the output $\mathcal{S}_\varepsilon$ of* SSector *is of size $S_\varepsilon = O\left((nC)^{d-1}\log_{1+\varepsilon}(nC)\right)$.*

Proof. Assume first that the weights are integers. Given a current rank r and a strip $\mathcal{R}_i^{(l)}$, Small Sample Sector stores at most one path for every $x \in \mathbb{Z}^{d-2}$. Thus for every i, $|\mathcal{R}_i'^{(l)}| = O(r^{d-2})$. Since we have at most $\lceil 2 + \log_{1+\varepsilon} r \rceil$ strips and d dimensions, $|\mathcal{R}'|$ is smaller than or equal to $d(r+1)^{d-2}(\lceil 2+\log_{1+\varepsilon} r\rceil)$. Since we have dnC ranks, $S_\varepsilon = O(d(dnC)^{d-1}\log_{1+\varepsilon}(dnC))$. Since d is constant, S_ε is in $O((nC)^{d-1}\log_{1+\varepsilon}(nC))$.

To get bounds on $C_{\text{Small Sample Sector}}$, we have to build data structures dedicated to range queries. The number of insertions to do before the queries is bounded by $O(\Delta\mathcal{S}_\varepsilon)$. Each of these insertions is in $O(\log^{d-1}\Delta\mathcal{S}_\varepsilon)$ and a range query is in $O\left(\left(\frac{\log \mathcal{S}_\varepsilon}{\log\log \mathcal{S}_\varepsilon}\right)^{d-1}\right)$ [Mor06]. Then the number of range queries is at most $d\Delta\mathcal{S}_\varepsilon$. Thus $C_{\text{Small Sample Sector}} = O(\Delta\mathcal{S}_\varepsilon \log^{d-1}\Delta\mathcal{S}_\varepsilon)$.

From Theorem 15, we have to add $O(\Delta S_\varepsilon \log(\Delta S_\varepsilon))$ time steps to get the complexity of both algorithms assuming d is constant. Whenever the arc weights are integers we also have $\Delta S_\varepsilon \leq dnC$. □

Comparison with TZ. We can observe that whenever C is moderate, SSector provides smaller upper bounds on the time complexity than TZ. Both complexities are in O of:

- $\Delta(nC)^{d-1}\log_{1+\varepsilon}(nC)\log^{d-1}\left(\Delta(nC)^{d-1}\log_{1+\varepsilon}(nC)\right)$ for SSector,
- $\Delta n^{d+1}\left(\log_{1+\varepsilon}(nC)\right)^{d-1}$ for TZ.

To make a simple comparison, let us consider $\Delta = \Theta(1)$ and $\varepsilon << 1$. Removing negligible terms and multiplicative constants, while using the assumption with ε, we obtain :

- $(nC)^{d-1}\frac{\log(nC)}{\varepsilon}\log^{d-1}\left(\frac{(nC)^{d-1}}{\varepsilon}\right)$ for SSector,
- $n^{d+1}\left(\frac{\log(nC)}{\varepsilon}\right)^{d-1}$ for TZ.

We use the following inequality for $x \geq 2$ and $y \geq 2$:

$$\log(xy) = \log(x) + \log(y) \leq x + \log(y) \leq x\log(y)$$

It is more than reasonable to consider that $\log((nC)^{d-1}) \geq 2$, and since $\varepsilon << 1$, we obtain that:

$$\log^{d-1}\left(\frac{(nC)^{d-1}}{\varepsilon}\right) \leq \frac{1}{\varepsilon^{d-1}}\log^{d-1}\left((nC)^{d-1}\right).$$

And then:

- $(nC)^{d-1}\left(\frac{\log(nC)}{\varepsilon}\right)^{d}$ for SSector,
- $n^{d+1}\left(\frac{\log(nC)}{\varepsilon}\right)^{d-1}$ for TZ.

and if $C^{d-1}\log(nC) \leq \varepsilon n^2$ then SSector has a smaller worst-case time complexity than TZ. For instance, if $d = 3$, it is the case if $C \leq \sqrt{\varepsilon}n^{1-\alpha}$ for any $\alpha > 0$.

3.5.4 Heuristic speed-up : Quick Sample Sector

In order to speed-up SSector, we propose Quick Sample Sector, a preprocessing algorithm for Small Sample Sector. It does not use $\mathcal{S}$ and just performs a quick (but light) pruning of the paths in $\mathcal{R}$. Similarly as Small Sample Sector, this algorithm considers each dimension independently to compute sets of paths $\mathcal{R}'_i \subseteq \mathcal{R}$ and the output of both algorithms is $\mathcal{R}' = \bigcup_{i=1}^{d}\mathcal{R}'_i$.

We use the same strip partitioning as for Small Sample Sector. Our sample algorithm Quick Sample Sector (see Alg. 11) proceeds as follows. For each path P of a given strip, P is kept in $\mathcal{R}'_i$ if and only if there is no other point Q belonging simultaneously to the same strip and the i-th sector of P, sharing the $d-2$

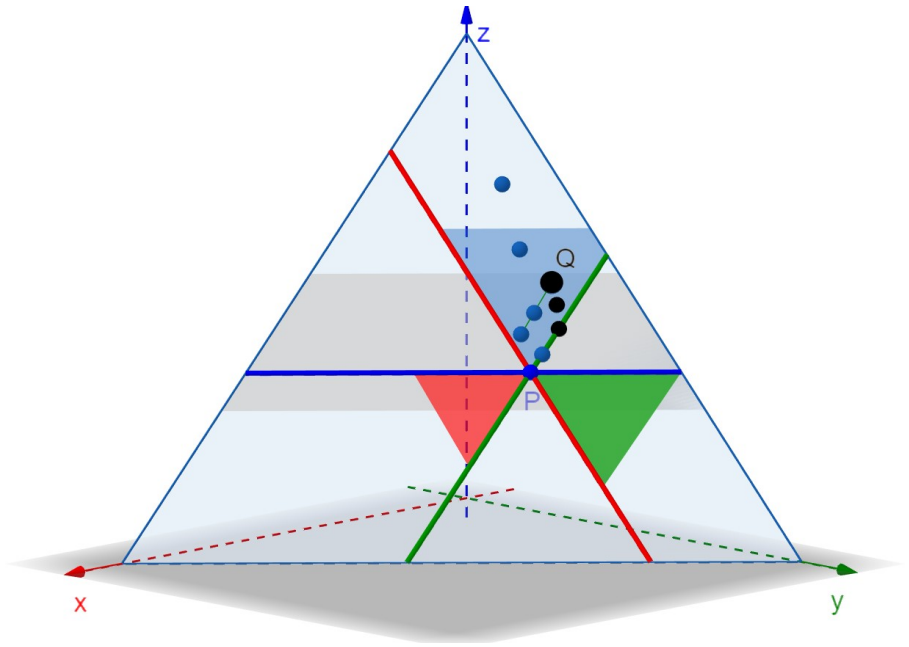


Figure 3.7 – In 3D, the 3 covering sectors of P are depicted in green, red and blue. Only 7 points are within the gray z-strip of P. Blacks points are kept by QUICK SAMPLE SECTOR and only Q is kept by SMALL SAMPLE SECTOR.

coordinates $i+1 \mod d, i+2 \mod d, \cdots, i+d-2 \mod d$ and such that $Q_i > P_i$. To do that, we store in a data structure the largest i-th coordinate for each encountered tuple formed by the previously described $d-2$ coordinates.

More precisely, let $\mathcal{I} = \{i+1 \mod d, i+2 \mod d, \ldots, i+d-2 \mod d\}$. Let x be in $\mathbb{Z}^{d-2}$. $Q_{\mathcal{I}}(x) = \{P \in \mathcal{R}_i^{(l)} | P_{i+j \mod d} = x_j \text{ for } j \in [1, d-2]\}$ is a set of paths sharing $d-2$ same coordinates in the same strip. Add $Q = \arg\max\{P_i | P \in Q_{\mathcal{I}}(x)\}$ to $\mathcal{R}'_i$.

We will show that the selection process can be done efficiently starting with the following property:

Lemma 33. *Let P and Q be two paths of the same strip. If P and Q differs on exactly two coordinates with $Q_i > P_i$, then Q belongs to the i-th sector of P and $(1+\varepsilon)$-covers P.*

Proof. For the sake of simplicity, let us take $i = 1$ and assume that $P_j = Q_j$ for $j \in [2, d-1]$. If $Q_1 > P_1$, since P and Q have same rank, we have $Q_d < P_d$. Thus, Q belongs to the i-th sector of P. Moreover, P and Q are in the same strip implying that $Q_1 \leq (1+\varepsilon)P_1$. □

Instead of using an array at line 11, one can use a balanced binary tree (Section A.2) to guarantee a time complexity in $O(\mathcal{R} \log \mathcal{R})$, or a hash table to get linear time complexity on average.

This algorithm cannot be used on itself as a `Sample` function. Indeed, it does not take $\mathcal{S}$ into account and therefore does not remove any dominated path. META RANK would then have an infinite loop since the paths would be extended indefi-

Input: $\mathcal{R}, \mathcal{S}$ sets of paths, $\varepsilon > 0$
Output: $\mathcal{R}'$ set of paths

$\mathcal{R}' = \emptyset$
$r \leftarrow \texttt{rank}(\mathcal{R})$
foreach $i \in [\![1; d]\!]$ **do**
    Set up $\mathcal{R}^{(0)}, \ldots, \mathcal{R}^{(\lceil \log_{1+\varepsilon} r \rceil + 1)}$ to empty sets
    foreach $P \in \mathcal{R}$ **do**
        if $P_i = 0$ **then**
            Add P to $\mathcal{R}^{(0)}$
        else
            Add P to $\mathcal{R}^{(\lceil \log_{1+\varepsilon} P_i \rceil + 1)}$
    foreach $l \in [\![0; \lceil \log_{1+\varepsilon} r \rceil + 1]\!]$ **do**
        Set up $\mathcal{R}'^{(l)}$ as an array indexed by $[\![0, r]\!]^{d-2}$ with empty entries
        $\mathcal{I} \leftarrow (i + 1 \mod d, i + 2 \mod d, \ldots, i + d - 2 \mod d)$
        foreach $P \in \mathcal{R}^{(l)}$ **do**
            $Q \leftarrow \mathcal{R}'^{(l)}[P_{\mathcal{I}_1}, P_{\mathcal{I}_2}, \ldots, P_{\mathcal{I}_{d-2}}]$
            if $Q = \emptyset$ *or* $P_i > Q_i$ **then**
                Substitute Q by P
        $\mathcal{R}'_i \leftarrow \mathcal{R}'_i \cup \mathcal{R}'^{(l)}$
    $\mathcal{R}' \leftarrow \mathcal{R}' \cup \mathcal{R}'_i$

Algorithm 11: Quick Sample Sector

nitely. We therefore limit ourselves to using it in a precomputation step for Small Sample Sector.

Definition 34. *Algorithm* QSSector *is the* Meta Rank *algorithm (Alg. 6) using* Quick Sample Sector *then* Small Sample Sector.

This way, `Sample` starts pruning as many paths as possible with an average constant time complexity per path, depending only on the size of $\mathcal{R}$. Then, it samples with Small Sample Sector algorithm whose complexity per path is logarithmic, and depends on the size of $\mathcal{S}$. If $\mathcal{R}'$ is the output of Quick Sample Sector$(\mathcal{R}, \mathcal{S})$, then adding Quick Sample Sector as a preprocessing step gives the following time complexity $O(|\mathcal{R}| + |\mathcal{R}'| \log(|\mathcal{R}' \cup \mathcal{S}|))$ instead of $O(|\mathcal{R}| \log(|\mathcal{R} \cup \mathcal{S}|))$. The overall complexity is also in $O(\Delta S_\varepsilon \log(\Delta S_\varepsilon))$.

Note that, given the same $\mathcal{R}$ and $\mathcal{S}$, the output $\mathcal{S}_u^r(\text{QSSector}) \subseteq S_u^r(\text{SSector})$. However, is not clear whether $S_\varepsilon(\text{QSSector})$ is smaller than $S_\varepsilon(\text{SSector})$. Intuitively, in practice, QSSector should be quicker than SSector.

Chapter 4

Shortest paths in 2D

The dimension 2 case allows an important speed-up for MC Dijkstra. We investigate a practical version of it, and provide a detailed description in Section 4.1. The associated data structure guarantees the complexity announced in the section 3.1.2. Then we propose a simple variant, Dijkstra Post, in order to reduce the computation time in practice in some cases. These algorithms are for an exact Pareto set computation.

After that, we propose Frame algorithm in order to optimize Sector in 2D (Section 4.2). For that, we introduce a new elimination criterion: the ε-strong framing property, stronger than the one introduced in Section 3.5.1 as the name suggests. Frame is based on this criterion, which guarantees that any output path is optimal. Finally, an extensive experimental study is presented in Section 4.3. We compare the performance of MC Dijkstra 2D, Dijkstra Post and Frame in different contexts.

4.1 MC Dijkstra in 2D

The algorithm described by [Han80] is relatively high level as for the choice of data structures to be used. However, it seems necessary to use a non trivial well known method in order to obtain the announced complexity $O(\Delta S \log(\Delta S))$. We detail it to give the reader a better understanding of the experimental results following this part.

4.1.1 Data structures

The interesting part happens during the extension of a path P. In order to insert it into the set $\mathcal{T}$ of temporary paths, we need to determine if it is dominated or not, whether by permanent or temporary solutions.

Permanent solutions. For each vertex u, the set $\mathcal{S}_u$ of permanent paths of destination u can be stored within a list using a lexicographic order. Since paths are processed in lexicographic order, inserting paths in $\mathcal{S}_u$ at the end of the list preserves the order. Whether the list is implemented with a linked list or a dynamic array, an insertion takes $O(1)$ amortized time (for the line 9 in Algorithm 5). The

dynamic array data structure refers to an array which size is doubled when required, guaranteeing a $O(1)$ amortized time per insertion. Furthermore, these arrays allow a logarithmic time search.

The function `IsNotDominated()` (line 11) can be done in constant time, comparing the extended path $P \cdot (v, w)$ with the first path in $\mathcal{S}_w$, which is sufficient since $P \cdot (v, w)$ is necessarily greater in lexicographic order that any path in $\mathcal{S}_w$.

Temporary solutions. For each vertex u, the set $\mathcal{T}_u$ is a self-balancing binary search tree (Section A.2) using the lexicographic order, allowing logarithmic search, insertion and deletion. Another tree $\mathcal{T}_{global}$, of the same type, is added. It contains the smallest path, in lexicographic order, of $\mathcal{T}_u$, for each vertex $u \in V$. In $\mathcal{T}_{global}$, paths with the same cost are kept and can be ordered by some vertex identifiers in order to guarantee a logarithmic time search. The two following operations are necessary:

- *Removal of a global minimum path in $\mathcal{T}$ in order to extend it* (lines 7 and 8). Let P be such a path. It is a minimum in $\mathcal{T}_{global}$. Let u be the destination of P. P is removed from $\mathcal{T}_u$ and replaced in $\mathcal{T}_{global}$ by the new minimum path of $\mathcal{T}_u$, in order to keep the minimum path of each non empty $\mathcal{T}_u$ in $\mathcal{T}_{global}$. This last operation can be done in $O(\min\{\log S, \log n\})$.

- `InsertAndClean` operation (line 12). Let P be the path inserted, and u its destination. Informally, if $\mathcal{T}_u$ is a sequence of path organized in lexicographic order, the paths that are not in a Pareto set of $\mathcal{T}_u \cup \{P\}$ form a subsequence of consecutive paths ending at P (the latter being included or not depending whether it is dominated by $\mathcal{T}_u$ or not).

  1. First, P is inserted in $\mathcal{T}_u$.
  2. Then, let $Q \in \mathcal{T}_u$ be the path following P in lexicographic order.
  3. If Q dominates the path P' preceding Q in $\mathcal{T}_u$, then P' is removed.
  4. This step is repeated until P' is not dominated by Q.

  Notice that the first instantiation of P' is P guaranteeing that P is not kept if it is dominated by $\mathcal{T}_u$. The amortized time complexity per path is in $O(\log |\mathcal{T}_u|)$.

This data structure has several drawbacks. First, it is much more technical than the classical implementations, based on a heap for example. The complexity is of the same order of magnitude, but one can notice that the multiplicative constants hidden in the O are larger. Secondly, in order to implement MC Dijkstra, this data structure is, to our knowledge, not implemented in standard libraries, contrary to heaps which are generally optimized there. The previous points imply a potentially time-consuming implementation in order to obtain reasonable performances. For these reasons, we propose to use Dijkstra Post as an alternative.

4.1.2 Dijkstra Post

If dominated paths are pruned while leaving $\mathcal{T}$ only by comparing them to paths in $\mathcal{S}$, the latter set could simply be a priority list ordered in lexicographic order, a

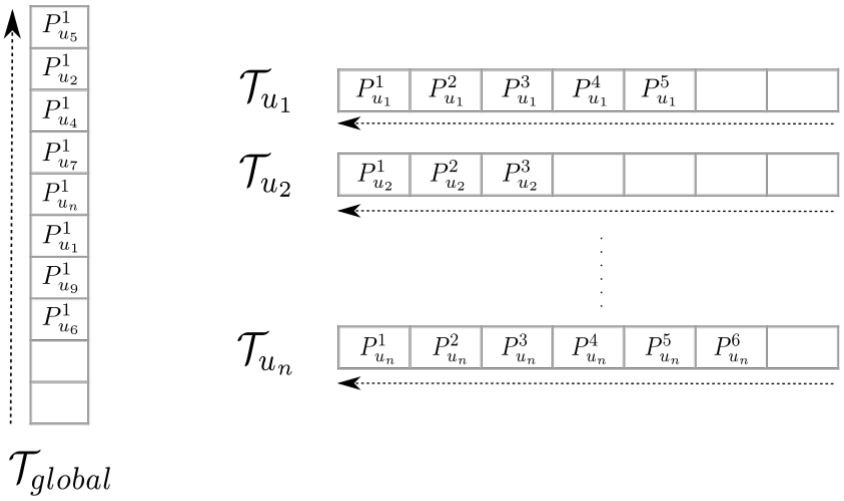


Figure 4.1 – MC DIJKSTRA 2D temporary solutions data structures.

data structure efficiently implemented in most common programming languages. On the other hand, we recall that testing dominance when paths are leaving $\mathcal{T}$ rather than entering it has the drawback to consider more paths. Indeed, contrary to MC DIJKSTRA, the priority list $\mathcal{T}$ is not a Pareto set. It is only when leaving this set that a path can be detected as useless. This implies a potentially bigger set $\mathcal{T}$, i.e., a higher number of seen paths.

4.2 Frame

Let us focus now on approximated Pareto set computation in dimension 2.

4.2.1 Elimination criterion

SECTOR could potentially return non optimal solutions. As it will be proven in Section 4.2.2, we prevent this from happening by introducing a stronger property, based on the idea that the representatives of a path have to cover themselves too. However, we will restrict the definition for $d = 2$ because it is not giving satisfying theoretical results in higher dimensions.

We start by giving the formal definition of what we call being framed between two paths. This definition is commented and illustrated afterwards.

Definition 35 (Frame)**.** *For any paths A, P, B such that $\texttt{rank}(A) \leq \texttt{rank}(P)$ and $\texttt{rank}(B) \leq \texttt{rank}(P)$, we say that A and B $\texttt{frame}$ P, or that P is* framed *between A and B if:*

$$\begin{array}{llll} \text{(i)} & A_1 \leq P_1 & \text{(iii)} & A_2 \leq (\texttt{rank}(P) - B_1)(1+\varepsilon) \\ \text{(ii)} & B_2 \leq P_2 & \text{(iv)} & B_1 \leq (\texttt{rank}(P) - A_2)(1+\varepsilon) \end{array}$$

We will note this property $\texttt{frame}(A, P, B, \varepsilon)$.

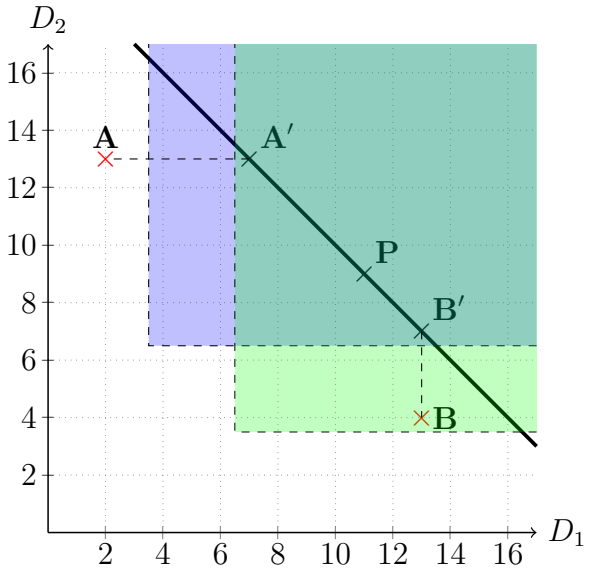


Figure 4.2 – A and B frame P for $\varepsilon = 1$, but not for $\varepsilon = 0.5$.

In the particular case where the two paths A and B have the same rank as the path P, if $\texttt{frame}(A, P, B, \varepsilon)$, then A and B match the $Q^{(1)}$ and $Q^{(2)}$ representatives of P in the ε-weak framing property, with the additional constraint that A and B $(1+\varepsilon)$-cover each other. This definition is extended for A and B having lower ranks than $\texttt{rank}(P)$, projecting those into the line of paths having the same rank as P. A is projected on the second dimension, B on the first one.

These projections of A and B are depicted in Figure 4.2 as A' and B'. The blue (resp. green) zone corresponds to the paths 2-covered by A' (resp. B'). Notice that the frame property requires the projections A' and B' to cover each other, but not necessarily A and B. Thus, in this example, $\texttt{frame}(A, P, B, 1)$ since $\texttt{frame}(A', P, B', 1)$.

We define the ε-strong framing property as a particular case of the ε-weak framing property for which the representatives of a path are framing it according to Definition 35.

Definition 36 (ε-strong framing property)**.** *A function* `Sample` *outputting* $\mathcal{R}'$ *on input* $(\mathcal{R}, \mathcal{S}, \varepsilon)$ *satisfies the* ε-strong framing property *if:*

- $\forall P \in \mathcal{R} \setminus \mathcal{R}',\ \exists A, B \in \mathcal{S} \cup \mathcal{R}',\ \texttt{frame}(A, P, B, \varepsilon)$,
- $\mathcal{R}'$ *is minimal by inclusion,*
- $\mathcal{S} \cup \mathcal{R}'$ *is a Pareto set.*

As the name suggests, the strong property is stronger than the weak one since it requires some conditions between the representatives, as well as the minimality of the output.

Proposition 37. *The ε-strong framing property implies the ε-weak framing property.*

Proof. Let `Sample` verifying the ε-strong framing property on inputs $(\mathcal{R}, \mathcal{S}, \varepsilon)$. Let $P \in \mathcal{R} \backslash \mathcal{R}'$. There exists $A, B \in \mathcal{S} \cup \mathcal{R}'$ such that $\texttt{frame}(A, P, B, \varepsilon)$. Since $\mathcal{S} \cup \mathcal{R}'$ is a Pareto set, we only need to show that there exists two representatives $Q^{(1)}, Q^{(2)} \in \mathcal{S} \cup \mathcal{R}'$, such that:

$$\begin{cases} Q_1^{(1)} \leq (1+\varepsilon)P_1 \\ Q_2^{(1)} \leq P_2 \end{cases} \quad \begin{cases} Q_2^{(2)} \leq (1+\varepsilon)P_2 \\ Q_1^{(2)} \leq P_1 \end{cases}$$

Unfortunately, setting $Q^{(1)} = B$ and $Q^{(2)} = A$ is not always sufficient. We consider three cases:

- if $A_2 \leq P_2$, then $Q^{(1)} = Q^{(2)} = A$ is correct,
- if $B_1 \leq P_1$, then $Q^{(1)} = Q^{(2)} = B$ is correct,
- otherwise, we take $Q^{(1)} = B$ and $Q^{(2)} = A$. Indeed:
  - $Q_1^{(2)} = A_1 \leq P_1$ and $Q_2^{(1)} = B_2 \leq P_2$ by (i) and (ii) (cf Def. 35),
  - $Q_1^{(1)} = B_1 \leq (1+\varepsilon) \cdot (\texttt{rank}(P) - A_2) = (1+\varepsilon) \cdot (P_1 + P_2 - A_2) \leq (1+\varepsilon)P_1$ by (iv)
  - $Q_2^{(2)} \leq (1+\varepsilon)P_2$ using symmetric arguments.

□

The following theorem is a direct corollary of Theorem 24 and the previous proposition.

Theorem 38. *With a function* `Sample` *satisfying the ε-strong framing property,* Meta Rank *algorithm (Alg. 6) solves the $(1+\varepsilon)$-approximate Multicriteria Shortest Path Problem.*

Proof. Corollary of Theorem 24 and Proposition 37. □

4.2.2 Pareto compatible property

In this section, we check that, by using a `Sample` function which satisfy the ε-strong property, Meta Rank returns only optimal solutions. We first recall the definition given in Section 1.7.

Definition 39 (Pareto compatible)**.** *An algorithm is said to be* Pareto compatible *if and only if its solution $(\mathcal{S}_{v,\varepsilon})_{v \in V}$ to the $(1+\varepsilon)$-approximated Multicriteria Shortest Path Problem is always a subset of a Pareto set $\mathcal{S}_v$, for every vertex v.*

During a Meta Rank execution, a path P could be framed, then removed. Furthermore, the extensions of the representatives could be themselves framed and removed, and so on. We show that the extensions of P are nevertheless still framed by kept paths in the ε-strong setting.

Lemma 40. *Let $\mathcal{S}_\varepsilon$ be the output of* Meta Rank *(Alg. 6) using a* `Sample` *function satisfying the ε-strong property. Any path P is framed by some paths $A, B \in \mathcal{S}_\varepsilon$.*

The idea is, by contradiction, to consider, among the paths not framed, one with minimum rank. This path cannot be empty, thus it can be written $P \cdot e$, with P a path and e an arc. By definition of $P \cdot e$, P is framed. Using paths A and B framing P, we can show that their extensions $A \cdot e$ and $B \cdot e$ are framing $P \cdot e$. These extensions are either kept in $\mathcal{S}_\varepsilon$ or in turn framed by some paths of $\mathcal{S}_\varepsilon$ framing $P \cdot e$ too.

Proof. For paths A, B and P, if P is framed by A and B, we note: $\alpha(P) = A, \beta(P) = B$ (beware that α and β are not functions, A and B not being necessarily unique). By contradiction, let us assume that there exist paths in the Pareto set that are not framed by the output. Let P' be such a path of minimum rank. If P' is an empty path, then it is the first path seen by the algorithm, and it is kept, giving directly a contradiction. Otherwise, we can write $P' = P \cdot e$, with e being the last arc of P'. We have $\texttt{rank}(P) < \texttt{rank}(P \cdot e)$, thus P is framed by two paths $\alpha(P), \beta(P) \in \mathcal{S}_\varepsilon$ framing P. Notice that if P is kept, we can say that P is framed by (P, P). We will note: $A = \alpha(P) \cdot e$ and $B = \beta(P) \cdot e$. Since $\alpha(P)$ and $\beta(P)$ are kept, A and B will be considered by the algorithm but not necessarily kept.

We consider three cases:

1. If the algorithm keeps both A and B, then they frame $P \cdot e$, since they have inferior ranks and:
   (i) $A_1 = \alpha(P)_1 + e_1 \leq P_1 + e_1$
   (ii) $B_2 = \beta(P)_2 + e_2 \leq P_2 + e_2$
   (iii) $\begin{aligned} A_2 &= \alpha(P)_2 + e_2 \\ &\leq (1+\varepsilon)(\texttt{rank}(P) - \beta(P)_1) + e_2 \\ &\leq (1+\varepsilon)(\texttt{rank}(P) - \beta(P)_1) + (1+\varepsilon)e_2 \\ &\leq (1+\varepsilon)(\texttt{rank}(P) - \beta(P)_1) + (1+\varepsilon)(\texttt{rank}(e) - e_1) \\ &\leq (1+\varepsilon)(\texttt{rank}(P) + \texttt{rank}(e) - \beta(P)_1 - e_1) \\ &\leq (1+\varepsilon)(\texttt{rank}(P \cdot e) - B_1) \end{aligned}$
   (iv) $B_1 \leq (\texttt{rank}(P \cdot e) - A_2)(1+\varepsilon)$ by a reasoning similar to (iii)

2. The algorithm keeps only one. W.l.o.g., we can consider that A is kept. B being removed, it is framed by $\alpha(B)$ and $\beta(B)$.

   - Either $\alpha(B)_1 \leq P_1 + e_1$, in which case, P' is framed by $\alpha(B)$ and $\beta(B)$ too. Indeed, we have $\beta(B)_2 \leq B_2 \leq P_2 + e_2 = P'_2$ giving (ii). And (iii), (iv) are given by the fact that $\texttt{rank}(B) \leq \texttt{rank}(P')$.
   - Otherwise A and $\alpha(B)$ frame P'. Indeed,
     (i) $A_1 = \alpha(P)_1 + e_1 \leq P_1 + e_1 = P'_1$
     (ii) $\alpha(B)_2 = \texttt{rank}(\alpha(B)) - \alpha(B)_1 \leq \texttt{rank}(P \cdot e) - (P_1 + e_1) \leq P_2 + e_2 = P'_2$
     (iii) $A_2 \leq (1+\varepsilon)(\texttt{rank}(P) - \beta(P)_1) + e_2 \leq (1+\varepsilon)(\texttt{rank}(P \cdot e) - B_1) \leq (1+\varepsilon)(\texttt{rank}(P \cdot e) - \alpha(B)_1)$
     (iv) $\alpha(B)_1 \leq B_1 \leq (1+\varepsilon)(\texttt{rank}(P) - \alpha(P)_2) + e_1 \leq (1+\varepsilon)(\texttt{rank}(P \cdot e) - A_2)$

3. The last case corresponds to removing both A and B. As in the previous case, if $\alpha(B)_1 \leq P_1 + e_1$, P is framed by $\alpha(B)$ and $\beta(B)$. Otherwise, A and $\alpha(B)$ frame P' and we can use the same reasoning than before, replacing B by $\alpha(B)$.

We have proved that P' is framed, leading to a contradiction.

□

It can be deduced from this lemma that the ε-strong property implies the Pareto compatibility.

Theorem 41. META RANK *(Alg. 6) using a* `Sample` *function satisfying the ε-strong property is Pareto compatible.*

Proof. By contradiction, we assume that a path $P \in \mathcal{S}_\varepsilon$ is dominated by some path Q. If $Q \in \mathcal{S}_\varepsilon$, then P cannot be kept since it is processed after Q and is dominated. Therefore, $Q \notin \mathcal{S}_\varepsilon$. According to Lemma 40, there exists $A, B \in \mathcal{S}_\varepsilon$ framing Q. Thus, A, B frame P, which would mean that P is not kept since $\mathcal{S}_\varepsilon$ is minimal. □

Remark. *This results guarantees that* META RANK *using a* `Sample` *function satisfying the ε-strong property output's size is smaller or equal than the Pareto set size. However, it does not enable any comparison with the minimum size of a $(1+\varepsilon)$-approximated Pareto set.*

4.2.3 Frame algorithm

We provide an efficient algorithm for `Sample`: SAMPLE FRAME and we prove that it verifies the ε-strong property (Definition 36). It guarantees that by using it in META RANK, the latter is Pareto compatible (Theorem 41).

Restriction to same rank paths. The algorithm is first presented in a simplified version, which is generalized afterwards. Let $\mathcal{R} = \{P^{(1)}, \cdots, P^{(k)}\}$ be a set of paths of rank r. We assume the paths $P^{(i)}$ to be sorted in lexicographic order.

The simplified algorithm consists in finding the maximum index j such that $P^{(1)}$ and $P^{(j)}$ cover each other. Then, $\forall\ 1 < i < j$, `frame`$(P^{(1)}, P^{(i)}, P^{(j)}, \varepsilon)$ holds, and those paths in-between are removed. Next, the algorithm is repeated recursively on $\mathcal{R}' = \{P^{(j)}, \cdots, P^{(k)}\}$ until $\mathcal{R}'$ contains at most two paths. The output of the simplified algorithm consists of the set of paths from $\mathcal{R}$ that were not removed. See Algorithm 12 for a more formal description of the simplified algorithm.

Input: k paths $(P^{(1)}, \cdots, P^{(k)})$ of same rank sorted in lexicographic order, $\varepsilon > 0$

```
1 i_min ← 1
2 for i = 2 to k − 1 do
3 |   if frame(P^(i_min), P^(i), P^(i+1), ε) then
4 |   |   Remove P^(i)
5 |   else
6 |   |   i_min ← i
```

Algorithm 12: SAMPLE FRAME SAME RANK

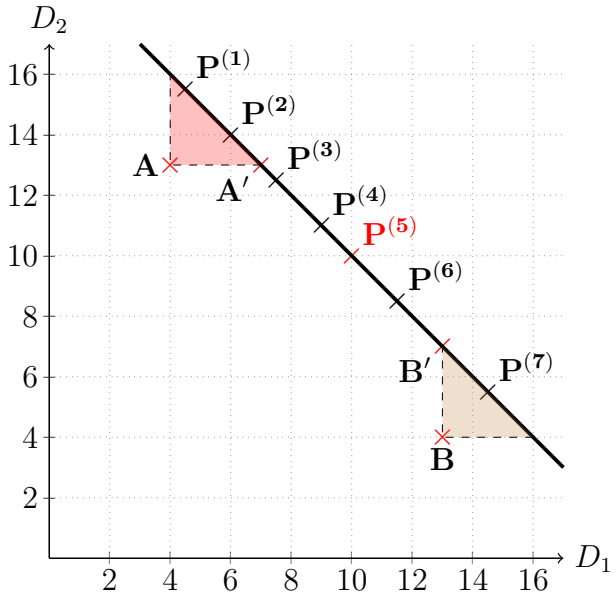


Figure 4.3 – Sample Frame. The paths $P^{(3)}, P^{(4)}, P^{(5)}$ and $P^{(6)}$ are framed by A and B for $\varepsilon = 1$, but not for $\varepsilon = 0.5$. In this latter case, the algorithm keeps the middle point $P^{(5)}$.

Framing with lower rank paths. In order to improve the pruning capability, paths from lower ranks are actually used to frame current rank paths. Assume that A and B are two paths of rank lower than r such that $\forall P \in \mathcal{R}, A_1 \leq P_1 \leq B_1$ and $B_2 \leq P_2 \leq A_2$. Then Sample Frame performs the following three steps:

1. Paths from $\mathcal{R}$ dominated by A are removed.
2. Let $A' = (r - A_2, A_2)$ and $B' = (B_1, r - B_1)$ be projections of A and B on the current rank r. If $P^{(i)}, \cdots, P^{(j)}$ are the paths from $\mathcal{R}$ non dominated by A or B, and sorted in lexicographic order, then the simplified algorithm is applied on $\{A', P^{(i)}, \cdots, P^{(j)}, B'\}$.
3. Paths from $\mathcal{R}$ dominated by B are removed.

An example of this case is depicted in Figure 4.3 for $\epsilon = 0.5$. The first step removes $P^{(1)}$ and $P^{(2)}$ since they are dominated by A. Then the second step computes the fact that A' and $P^{(5)}$ cover each other but not A' and $P^{(6)}$. Thus, $P^{(3)}$ and $P^{(4)}$ are removed too. Since $P^{(5)}$ and B' cover each other, $P^{(6)}$ is removed. Finally, during the third step, $P^{(7)}$ is removed because B dominates it. Sample Frame's output is $\{P^{(5)}\}$.

Remark. *If the input of* Sample Frame Same Rank *is a set of paths of rank r, then the output size is smaller than or equal to $2\left\lceil \log_{1+\varepsilon} r \right\rceil$. Indeed, on the first coordinate there is at most two path in any range of size $1 + \varepsilon$.*

Sample Frame Algorithm. In a general setting, the inputs of Sample Frame are an unordered set $\mathcal{R} = \{P^{(1)}, \cdots, P^{(k)}\}$ of paths of rank r, and a Pareto set $\mathcal{S}$ of paths of rank lower than r. Algorithm Sample Frame proceeds as follows:

1. First, $\mathcal{R}$ is sorted in lexicographic order.

2. Then, let $A = \arg\max_{Q \in \mathcal{S}} \{Q_1 | Q_1 \leq P_1^{(1)}\}$ and $B = \arg\min_{Q \in \mathcal{S}} \{Q_1 | Q_1 > P_1^{(1)}\}$. Note that B is the path following A in $\mathcal{S}$ in lexicographic order.

3. If A is not defined, then set $A' = P^{(1)}$ and apply the algorithm to the subset of paths $\mathcal{R} = \{P^{(2)}, \cdots, P^{(k)}\}$. Symmetrically, if B is not defined, then set $B' = P^{(k)}$ and apply the algorithm to $\mathcal{R} = \{P^{(1)}, \cdots, P^{(k-1)}\}$.

4. Let j be the maximum index such that $P_1^{(j)} < B_1$. Intuitively, the paths $P^{(1)}, \cdots, P^{(j)}$ are the paths between A and B as in the previously described situation.

5. Sample Frame applies the corresponding three steps to these paths.

6. Then, this algorithm is recursively applied on $\{P^{(j+1)}, \cdots, P^{(k)}\}$.

To search A and B among $\mathcal{S}$ efficiently, $\mathcal{S}$ is a balanced search tree allowing a logarithmic time search. Similarly to Sector using Sample Sector, we can now define our algorithm Frame using Sample Frame.

Definition 42 (Frame Algorithm)**.** *Algorithm* Frame *is the* Meta Rank *algorithm (Alg. 6) using* Sample Frame *as the* `Sample` *function.*

In order to prove that Frame is Pareto compatible, it is sufficient to verify that Sample Frame satisfies the ε-strong property thanks to Theorem 41. Intuitively, one can see on the example depicted in Figure 4.2 that any removed path is either between two consecutive (in lexicographic order) kept paths, or dominated, thus framed by the dominating path.

Theorem 43. Sample Frame *algorithm satisfies the ε-strong framing property.*

Proof. Deleted paths are always framed by kept paths. Furthermore, the output is minimal since the algorithm is framing the largest interval possible. Finally, for A and B fixed, steps 1 and 3 remove dominated paths, guaranteeing to have a Pareto set as output. □

4.2.4 Complexities

Sample Frame$(\mathcal{S}, \mathcal{R}, \varepsilon)$ is efficient since it processes sequentially the paths from $\mathcal{R}$, and potentially for each one of those, performs a logarithmic search through $\mathcal{S}$.

Proposition 44. *Let R (resp. S) be the number of paths of rank r (resp. inferior to r). The complexity of the* Sample Frame *algorithm is $O(R(\log R + \log S)\})$.*

Proof. Paths of rank r are sorted in $O(R \log R)$ time. Then these paths are considered only once and each one may require to search for A and B in $O(\log S)$ time. □

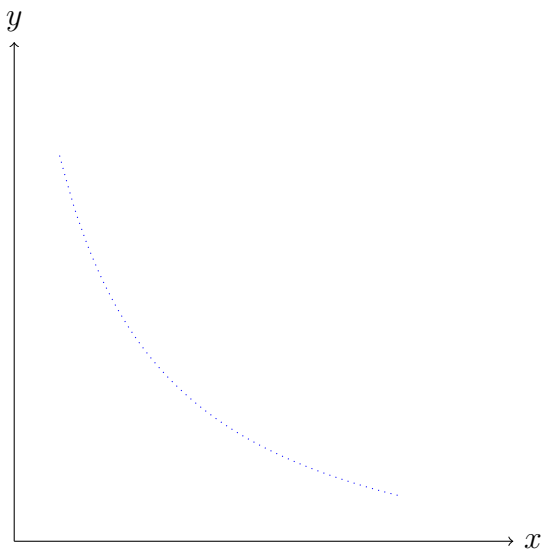


Figure 4.4 – Worst-case for FRAME: the whole Pareto set is kept.

Remark. *If $S/\log(S)$ is small compared to R, it would be more efficient to browse in parallel through the paths of inferior ranks in order to use those as representatives. This browsing consists in $O(R+S)$ operations. With the initial sorting of $\mathcal{R}$, the overall complexity would be $O(R\log(R)+S)$. Since we know the values of R and S before applying* SAMPLE FRAME*, we may choose the more appropriate method.*

With the previous proposition and Theorem 15, the time complexity of FRAME, claimed in Table 1.1, is computable by summing the complexities of each call to SAMPLE FRAME.

Theorem 45. *Let S_ε be the size of the output of* FRAME*. The output-sensitive time complexity of* FRAME *is in $O\left(\Delta S_\varepsilon \log(\Delta S_\varepsilon)\right)$.*

Proof. For each vertex u and rank r, let T_u^r be the size of the first parameter of `Sample`, and $S_u^{<r}$ be the size of the second parameter of `Sample`. Then the complexity of `Sample` using SAMPLE FRAME is $O(T_u^r(\log S_u^{<r} + \log T_u^r))$ which is in $O(T_u^r(\log(\Delta S_\varepsilon)))$ since $T_u^r \leq \Delta S_\varepsilon$. Repeating this operation over each vertex and rank gives $C_{\texttt{Sample}}(n, S_\varepsilon, \Delta, \Lambda) = O(\Delta S_\varepsilon \log(\Delta S_\varepsilon))$. Furthermore, recall that adding an optimal path to the set of permanent paths costs $O(\log S_\varepsilon)$, therefore the overall complexity for the line 13 of META RANK (Alg. 6) is $O(S_\varepsilon \log S_\varepsilon)$. Applying Theorem 15 allows us to conclude. □

Remark. *Although* FRAME *is Pareto compatible, it is possible that it computes the whole Pareto set. For instance, if the Pareto set has the shape depicted in Figure 4.4, no path is pruned. Indeed, the first path processed will be the one in the center, then as the rank increases, the paths on both sides are processed. And none of them can be framed on both sides: on one side, all the other paths have a higher rank.*

4.3 Experiments : is the Pareto compatible property practically relevant?

Although FRAME is Pareto compatible, it is interesting to check whether S_ε given by FRAME is really smaller than S in practice. The potential for path pruning using *Pareto-compatible approximation algorithms* allows us to hope for better performance in practice.

4.3.1 Protocol

Algorithms

In the following, we mainly focus on the only Pareto compatible algorithm known so far: FRAME. Remind that TZ and HYDRID are not Pareto-compatible. Furthermore, these two algorithms are unpractical, and the modified versions do not guarantee that the output is a $(1+\varepsilon)$-approximated Pareto set (see Sections 1.5.2 and 3.4.3 for more details). The main question is the following : *in which cases is* FRAME *better than* MC DIJKSTRA*?*

We will then analyze the reasons that lead to the observed gains. We also compare the performance of MC DIJKSTRA and DIJKSTRA POST: MC DIJKSTRA stores less paths since it prunes the paths before DIJKSTRA POST but DIJKSTRA POST data structures are simpler and quicker.

We focus only on the *one-to-one query*, by incorporating into our algorithms the termination criterion described in Section 3.1.4. Our hypothesis is that the efficiency of FRAME is highly linked to the Pareto set size S. Informally, we can say that S is small if it is smaller than 10, intermediate for a few dozens to hundreds and large for thousands.

Parameters

First, we identify the main parameters that are likely to impact the efficiency of the FRAME.

- The main one is the exact Pareto set size S. Whenever it is small, there is little to gain, whereas significant pruning can be made with large Pareto sets. This remark leads us to wonder how to increase it and how these different techniques impact on the gain.
- A second parameter, more obvious fact, is *the choice of* ε: if this one is too small, FRAME cannot prune, similarly as TZ. By default, in our experiments, we take $\varepsilon = 1$.

Evaluation criteria

Then, we must choose the criteria that are relevant to evaluate.

- *The time* is the most natural to evaluate during an algorithm execution. However, it is biased by the implementation choices. We also consider other criteria

to get a deeper understanding of the differences in time efficiency of the algorithms.

- The *approximated Pareto set size* S_ε, that is the size of the output, gives a direct idea of the pruning rate.
- We also study the number of paths built within the algorithms as a measure of the *memory space consumption*.
- A measure of ε a posteriori, noted ε_{post}, evaluates *the similarity* between $\mathcal{S}_\varepsilon$ and $\mathcal{S}$. This parameter is defined as the minimum ε such that the output $\mathcal{S}_\varepsilon$ of FRAME is a $(1+\varepsilon)$-approximation of the Pareto set $\mathcal{S}$. More formally:

$$\varepsilon_{post} = \max_{P=(P_1,P_2)\in\mathcal{S}} \left(\min_{Q=(Q_1,Q_2)\in\mathcal{S}_\varepsilon} \max \left\{ \frac{Q_1}{P_1}, \frac{Q_2}{P_2} \right\} \right)$$

In the d-dimensional case, we compute the a posteriori epsilon ϵ_{post} with the Algorithm 13.

Input: S exact solution, S_ε approximated solution
Output: ε_{post}

```
1 ε_post ← 0
2 foreach P ∈ S\S_ε do
3     δ ← 1
4     foreach Q ∈ S_ε do
5         δ_Q ← 1
6         for i ∈ ⟦1, d⟧ do
7             δ_Q ← max{Q_i/P_i, δ_Q}
8         δ ← min{δ_Q, δ}
9     ε_post ← max{δ, ε_post}
```

Algorithm 13: Computing epsilon a posteriori

Graphs

Our experimental study is done on synthetic graphs and a dataset of real graphs. All arc weights are pairs of positive integers.

Complete oriented graphs $\overrightarrow{K_n}$. We borrow the graph construction proposed by Breugem [BDH17] since it allows to generate and tune graphs from large to small Pareto sets only by modifying arc costs and the density of the graph. As mentioned in Section 2.3.3, Breugem et al. shows how to build a large Pareto set of size 2^{n-2} based on the complete oriented graph $\overrightarrow{K_n}$ with n vertices. Vertex identifiers are $\{u_0, u_1, \ldots, u_{n-1}\}$ and arcs are such that (u_i, u_j) exists if and only if $i < j$, with weight $w(u_i, u_j) = (2^{j-1} - 2^i, 2^{j-1})$.

We focus on paths from u_0 to u_{n-1}. Each path $P = (P_1, P_2)$ is optimal with $P_1 \in [\![0, 2^{n-2} - 1]\!]$ and $P_2 = 2^{n-1} - 1 - P_1 \in [\![2^{n-2}, 2^{n-1} - 1]\!]$. In order to avoid having a null weight, we slightly modify the first dimension: $w(u_i, u_j) = (2^{j-1} - 2^i + j - i, 2^{j-1})$, which adds $n - 1$ to the first coordinate of all paths.

Thus, we can bound the optimal $(1 + \varepsilon)$-Pareto set size S^*_ε for n large enough:

$$\lfloor (n-2) \log_{1+\varepsilon}(2) \rfloor - 1 \leq S^*_\varepsilon \leq 2 \left(\lceil (n-2) \log_{1+\varepsilon}(2) \rceil + 1 \right)$$

The lower bound is obtained by bounding the aspect ratio of the first dimension and the upper bound by counting the number of parts covering the Pareto set, with the partition defined in Section 3.4.1.

Here we take $\overrightarrow{K_n}$ with $n \in [\![9, 19]\!]$. Then we follow the same protocol as in [BDH17], using two variation models:

- First of all, in the *weight variation model*, $w_{i,j}$ is a pair of random variables following a Normal law of mean the two initial weights $2^{j-1} - 2^i$ and 2^i and of variable standard deviation σ. We take σ an even number in $[\![0, 50]\!]$.

- The second model is the *arc variation model*: those are intermediate graphs between $\overrightarrow{K_n}$ and the standard Erdös-Renyi random graphs. Parameter p defines the closeness to these two extreme graphs: every arc of $\overrightarrow{K_n}$ is changed (removed or redirected) with probability p. Whenever $p = 0$, we get $\overrightarrow{K_n}$, and for $p = 1$, we have a pure random graph. More formally, for vertices i and j, with $i < j$, with probability p, the arc (u_i, u_j) is removed. In that case, with probability 0.8, we replace it with a new arc of weights chosen uniformly at random in $[\![1, 2^{n-1}]\!]$.

For these two models, increasing either σ or p leads to graphs with smaller Pareto sets.

Grid-based graphs: R_n and $\overrightarrow{U_n}$. Grids are good abstractions for road networks and can be good candidates to get Pareto sets of different sizes. Again, two models are proposed:

- The first graph model deals with the *bidirected rectangular grids* R_n [BDH17; RE09] showing the impact of the topology. We take a rectangular grid of size $n = n_1 \times n_2$, with n_1 the *width* and n_2 the *height*. The source vertex is connected to the n_2 vertices on the left-hand side using unidirected arcs and the target has the n_2 vertices of the right-hand side as in-neighbors. In our experiments, we take $n = 8192 = 2^{13}$ and the width n_1 is a power of 2 to get a variation of the aspect ratio from $1/n$ to n. As in [RE09], weights are uniformly drawn at random within the set $\{1, 2, \ldots, 10\}$.

- The *unidirected square grids* $\overrightarrow{U_n}$, with n the number of vertices, aims at evaluating the impact of the size of the graph and the correlations of the weights. We consider $\overrightarrow{U_n}$ for $n \in \{100, 1024, 10000\}$ vertices. We choose uniformly at random 100 pairs of reachable sources and targets. The arcs can only go to the right or upwards. The weight of the first coordinates are uniformly drawn

at random in $[\![1, C]\!]$, with $C \in \{5, 10, 15, 20, 50, 100\}$. To set the second coordinate, we use three methods: *uniform*, *correlated* (as time and distance) and *anticorrelated* (as time and financial cost).

- For the correlated weight generation, the second coordinate follows a Normal law of expectation the value of the first coordinate and of standard deviation $\sigma = 0.25 \cdot (C - 1)$.
- For the anticorrelated weight generation, two randoms variables are drawn uniformly at random: an integer rank R within $[\![2, 2C]\!]$ and the first coordinate $X \in [\![1, \min(C, R - 1)]\!]$. The correspondent arc weight is $(X, R - X)$.

In both grid models, the experiments are averaged on 20 random weights assignments.

DIMACS. The 9^{th} DIMACS challenge [DGJ] contains a set of real bicriteria (distance and edge traversal time) datasets representing road networks for every state of the US. We have selected 49 graphs of different sizes up to 2 millions of arcs. Two states, Texas and Illinois, are not presented: the experiments required more than 16Go of RAM. A hundred pairs of reachable source and destination are chosen randomly.

Weight correlation. The criteria correlation can be measured in different ways, the most standard being the quotient of the covariance by the product of the variances. This measure has a value in $[-1, 1]$. A value of 1 means that one criterion is a linear function of the other, with a positive slope, while a value of -1 indicates a negative slope. On the Dimacs graphs, we observe a correlation ranging from 0.4 to 0.7. The anticorrelation grids have negative correlations around -0.2.

Implementation

Algorithms have been implemented in C++, using data structures which guarantee the desired complexities for dimension 2. Temporary and permanent solution sets ($\mathcal{T}_u$ and $\mathcal{S}_u$) are implemented using *std::set* class template. For MC Dijkstra, a global temporary solution is used to store the minimum path of each $\mathcal{T}_u$. It is also a *set*, and the priority list of Meta Rank is implemented using *std::map* class template. Domination tests for MC Dijkstra are in constant time, according to the method presented in Section 4.1.1. The program is compiled with g++-8 and the option -o2. It is executed on a computer running Ubuntu 18.04.3, having 16GB RAM and an Intel Core i7-6700 processor.

4.3.2 Pareto set size

Our experiments show that the efficiency of Frame is highly correlated to the Pareto set size S. Whenever it is small, Frame has an overhead with respect to MC Dijkstra and Dijkstra Post since it runs a sample function but keeps

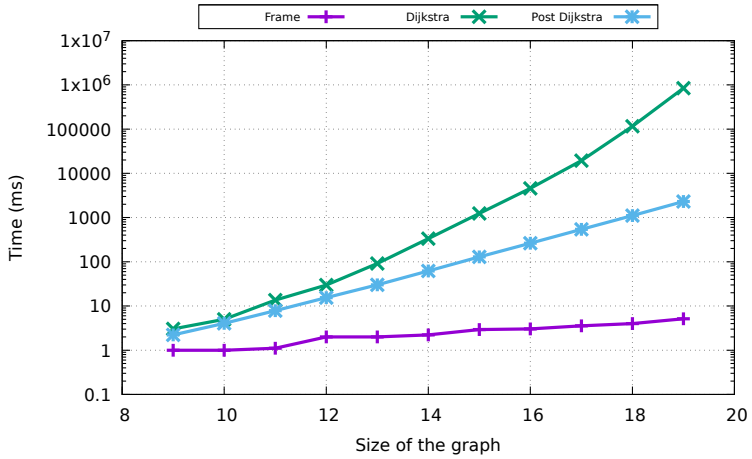


Figure 4.5 – Execution time for $\overrightarrow{K_9}$ up to $\overrightarrow{K_{19}}$.

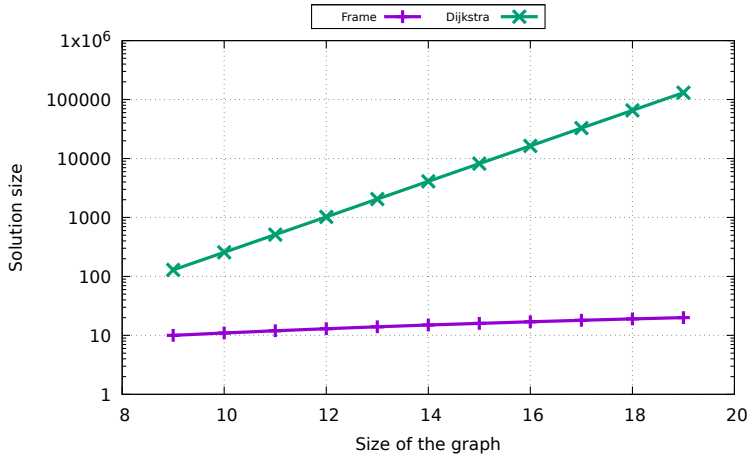


Figure 4.6 – Solution size for $\overrightarrow{K_9}$ up to $\overrightarrow{K_{19}}$.

almost all paths. The goal of this section is to measure the impact of the different parameters and graph models on the gain of Frame.

Large Pareto sets

$\overrightarrow{K_n}$ Pareto set size is 2^{n-2}. In Figures 4.5 and 4.6, we observe an exponential time gain. For $n = 19$, Frame is more than 10^5 faster than MC Dijkstra for $d = 2$. For this extreme case, S_ε is much smaller than S. The number S_ε of solutions output by Frame is of the same order of magnitude as the minimum $(1+\varepsilon)$-Pareto set size S^*_ε.

Surprisingly, MC Dijkstra is much less efficient than Dijkstra Post. This difference could be explained by the MC Dijkstra data structures being more complex than these of Dijkstra Post. It is important to notice than all the paths are incomparable and thus are kept by both algorithms. A deeper analysis could be required to completely explain the bad behavior of MC Dijkstra.

Variations

These graphs being quite particular, we now take the variations of $\overrightarrow{K_n}$ with 19 vertices:

- *Weight variation model.* For every even value σ in $[\![0, 50]\!]$, we generate 100 random graphs and we represent the average measures. As observed in [BDH17], the weight variation leads to smaller Pareto sets (Figure 4.7). Thus MC Dijkstra and Dijkstra Post take less time whereas the running time of Frame is more or less unchanged. It is quite natural since the output does not change drastically for Frame. For a wide range of value of $\sigma \in [10, 50]$, Frame is at least 10 times faster than MC Dijkstra.

- *Arc variation model.* This model gives a random graph being an intermediate of the standard Erdös-Rényi graph and $\overrightarrow{K_{19}}$. Whenever $p = 1$, it corresponds to a random graph with an average degree $0.8n$ with a random assignment of the arc weight. We observe a time gain of several order of magnitude whenever $p < 0.5$ (Figure 4.8). However, MC Dijkstra performance improves whenever p increases and that of Frame remains stable. This is explained by S being small for p close to 1 (Figure 4.9).

  Besides the correlation between the time gain and the size of the Pareto set, Frame performs better whenever Λ, the number of rank values processed by Frame, is small. For $p = 0$, it is typically $\Lambda = n = 19$ and for $i > 0$, every paths from s to u_i has the same rank : $2^i - 1$.

We measure the loss of efficiency of Frame by computing ε_{post}. We observe in Figure 4.10 that even if $\varepsilon = 1$, ε_{post} is much smaller as p converges to 1. In this situation, Λ becomes larger.

Worst-case

In order to challenge Frame, we multiply artificially the number of processed ranks Λ while keeping the same number of paths. In $\overrightarrow{K_{19}}$, the first coordinate of each arc is multiplied by 2. This transformation is depicted in Figure 4.11. S is unchanged but we get $\Lambda = 2^{n-2}$, meaning that each path has a distinct rank. For the output of Frame, the situation is a worst-case: for every path P from s to t, there is a sector with no path of smaller rank. Thus P cannot be framed and there is no interest in using Frame in this situation with respect to Dijkstra Post. The performance of Frame drops and aligns with that of Dijkstra Post, as can be seen in Figure 4.12.

Typical topologies

For the different variations of $\overrightarrow{K_n}$, the graphs are small, quite dense and with a huge maximum weight $C = 2^{n-1}$. These cases are quite extreme. Let us focus on more standard topologies to see if there is still a relationship between the Pareto set size and the efficiency of Frame.

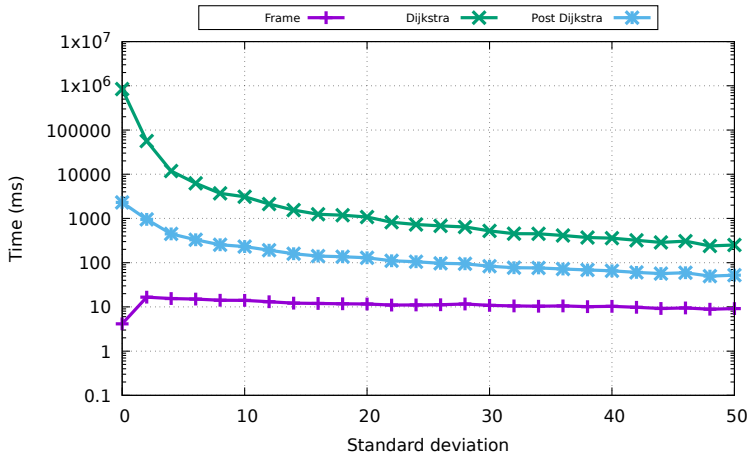


Figure 4.7 – Execution time for $\overrightarrow{K_{19}}$ using the weight variation.

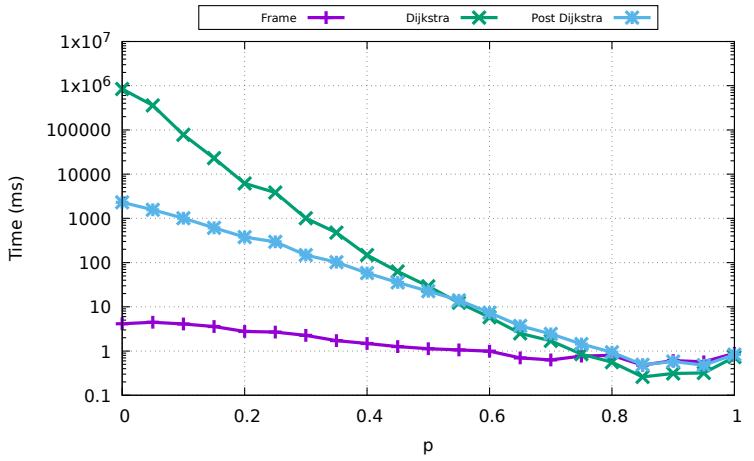


Figure 4.8 – Execution time for $\overrightarrow{K_{19}}$ using the arc variation.

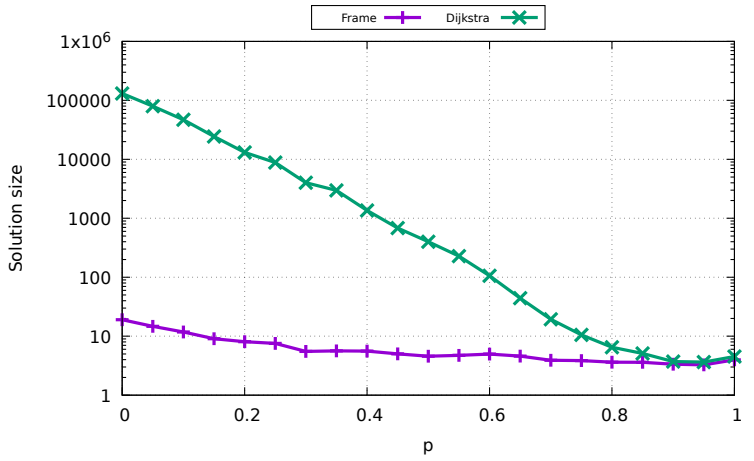


Figure 4.9 – Solution size for $\overrightarrow{K_{19}}$ using the arc variation.

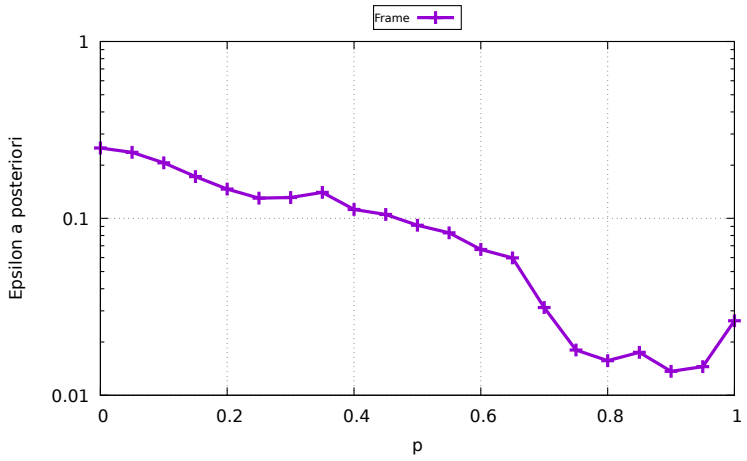
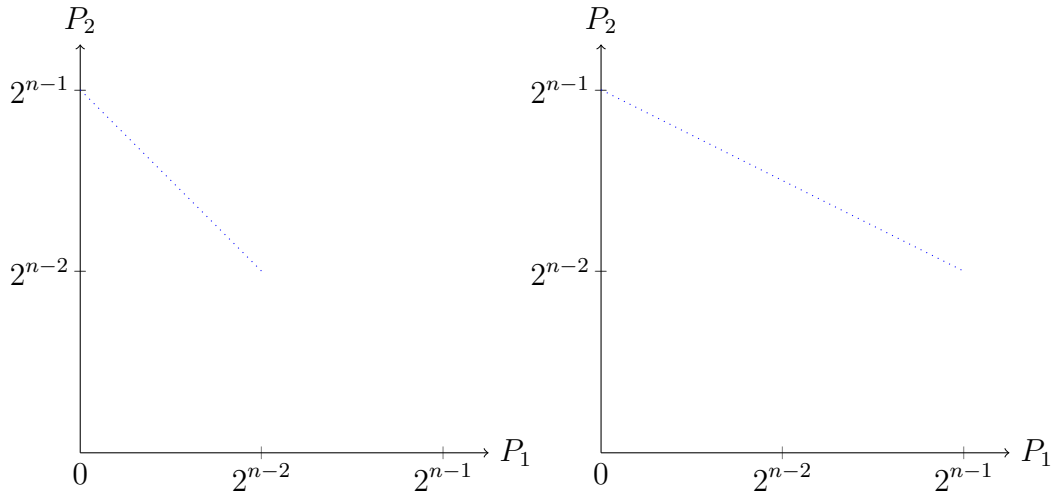


Figure 4.10 – Loss of efficiency for $\overrightarrow{K_{19}}$ using arc variation.



(a) Pareto set of $\overrightarrow{K_n}$.

(b) Bent Pareto set of $\overrightarrow{K_n}$.

Figure 4.11 – Transformation of the Pareto set of $\overrightarrow{K_n}$ to challenge FRAME.

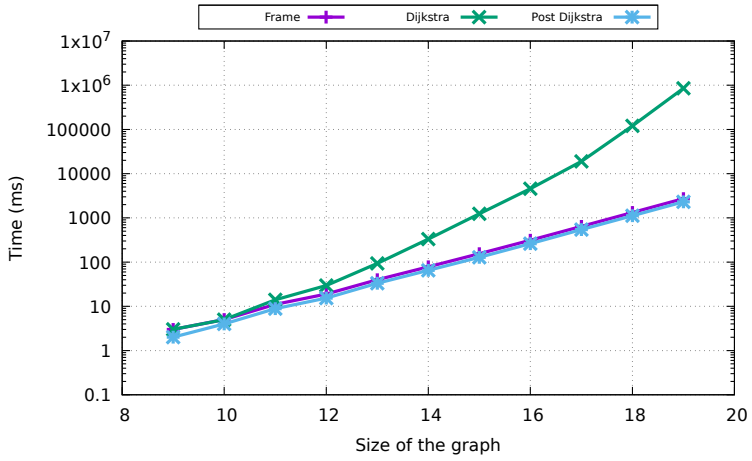


Figure 4.12 – Worst-case execution time comparison.

Vertices	MC Dijkstra time	Frame time	S_u	$S_{u,1}$(Frame)
100	0.684	0.631	3.2215	2.7140
1024	42.803	31.316	18.7455	11.5770
10000	2827.055	1714.527	116.8230	64.2525

Table 4.1 – Unidirected grids $\overrightarrow{U_n}$: impact of the size variation.

Impact of the size of the graph. Increasing S can be done by increasing the size of the graph. For grids $\overrightarrow{U_n}$ up to $100 \times 100 = 10000$ vertices (Table 4.1) with $C = 100$, S increases with n but remains intermediate. We observe that Frame has an output almost twice smaller than MC Dijkstra's one. As for the time, the gain is 40% for $n = 10000$.

This trend can be also be seen for DIMACS real graphs in Figures 4.13 and 4.14 for intermediate S (VA or NC for instance). When S is small, Frame and MC Dijkstra performs similarly. Although S seems to be correlated to n, we have some exceptions: for instance, the California (CA) anf Florida (FL) graphs have quite small Pareto sets in comparison to their number of nodes. It can explained by their thin geometric shape. The values corresponding to each state can be found in the appendix, in the Table B.1.

Impact of the correlation of arc weights and the maximum weight. In this experiment, we first show the impact of the maximum weight C for each weight distribution. Informally, increasing the range of arc weights can increase the number of incomparable paths and thus S. We investigate the impact of the maximum weight in the three scenarios: *uniform*, *correlated* and *anticorrelated*. For every scenario, when $C = 100$, Frame is from 1.4 up to 2 times faster than MC Dijkstra and Dijkstra Post (see Figures 4.15, 4.16 and 4.17). This gain can be explained by S_ε being almost twice smaller than S (see Figures 4.18, 4.19 and 4.20). This observation has already been done [MHW06] for other graph families.

By comparing the different kinds of weight distributions, we can confirm the intuition that, in the correlated case, the Pareto sets are smaller than those in the uniform case, which are themselves smaller than those in the anticorrelated case.

Impact of the topology shape.

The experimental results on California state suggest an influence of the grid shapes on the size of the Pareto set. This has already been established in [BDH17; RE09] for bidirected grids R_n. With $n = 8192$, whenever we increase the width of the grids, and therefore decrease the height, S increases (see Figure 4.21). For each algorithm, the total number of paths seen is depicted in Figure 4.22. For this graph family, we notice that MC Dijkstra prunes more paths than Dijkstra Post and Frame. Surprisingly, although Frame builds 50% more paths than MC Dijkstra, it has a similar running time (see Figure 4.23), which exhibits the advantage of Frame data structure. However, Dijkstra Post is slower due to the larger number of seen paths.

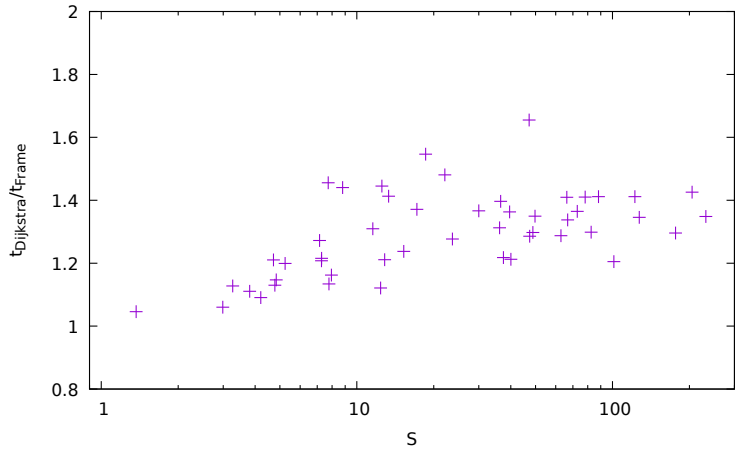


Figure 4.13 – Execution time gain of Frame over MC Dijkstra on Dimacs graphs.

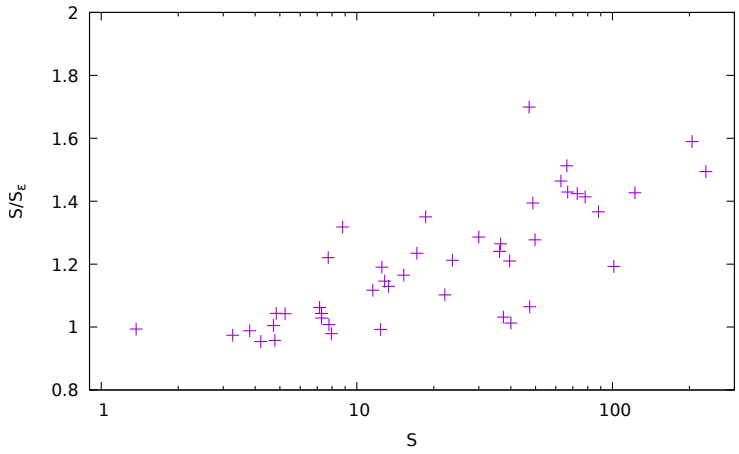


Figure 4.14 – Solution size gain of Frame over MC Dijkstra on Dimacs graphs.

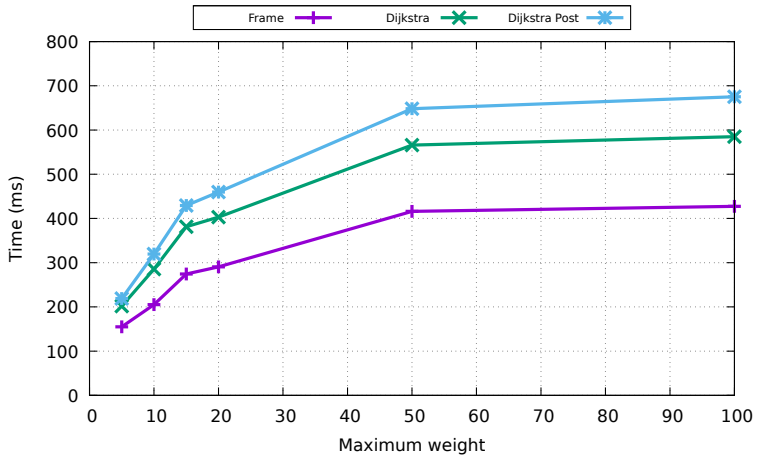


Figure 4.15 – Execution time for correlated distribution in $\overrightarrow{U_n}$.

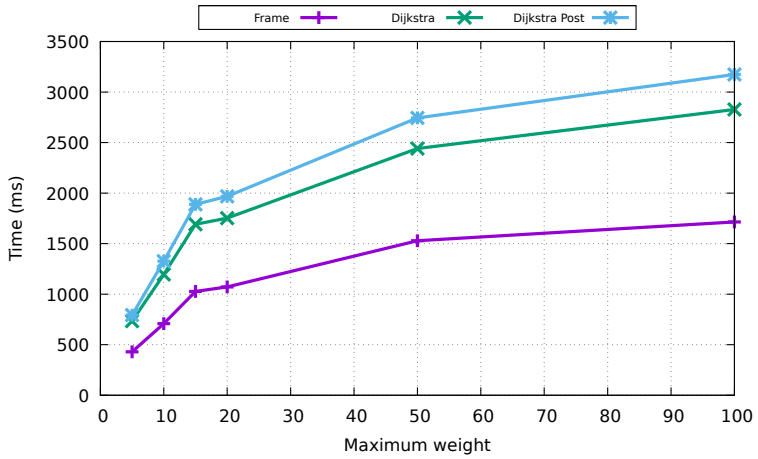


Figure 4.16 – Execution time for uniform distribution in $\overrightarrow{U_n}$.

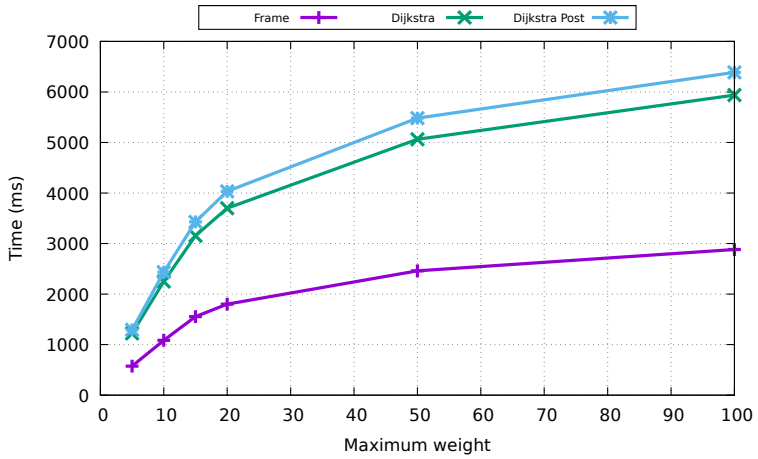


Figure 4.17 – Execution time for anticorrelated distribution in $\overrightarrow{U_n}$.

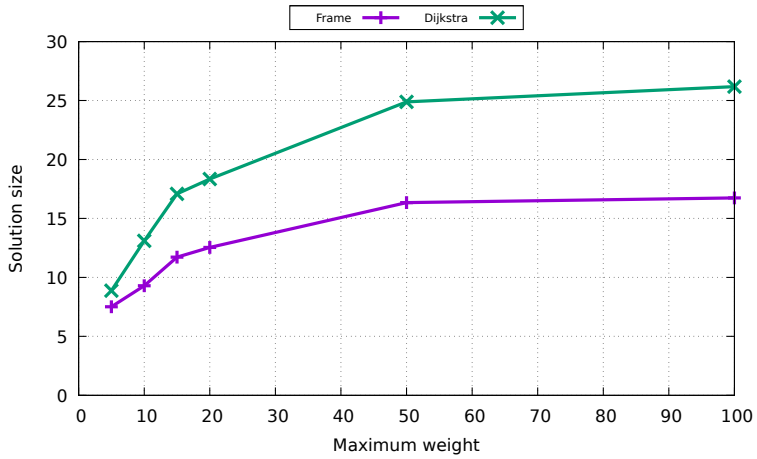


Figure 4.18 – S_u for correlated distribution in $\overrightarrow{U_n}$.

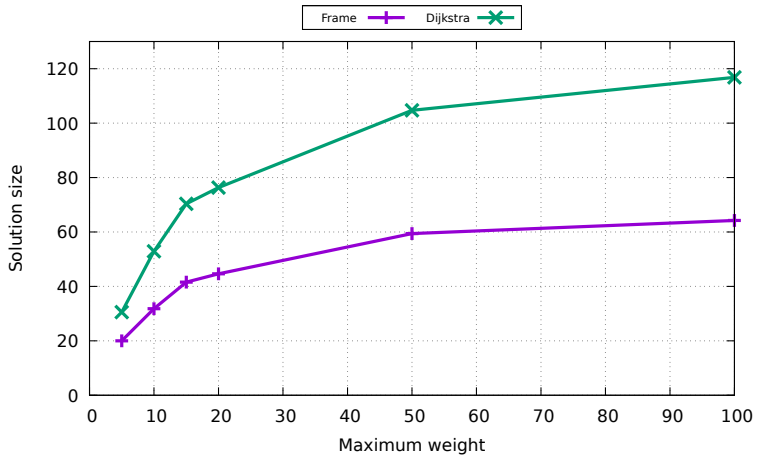


Figure 4.19 – S_u for uniform distribution in $\overrightarrow{U_n}$.

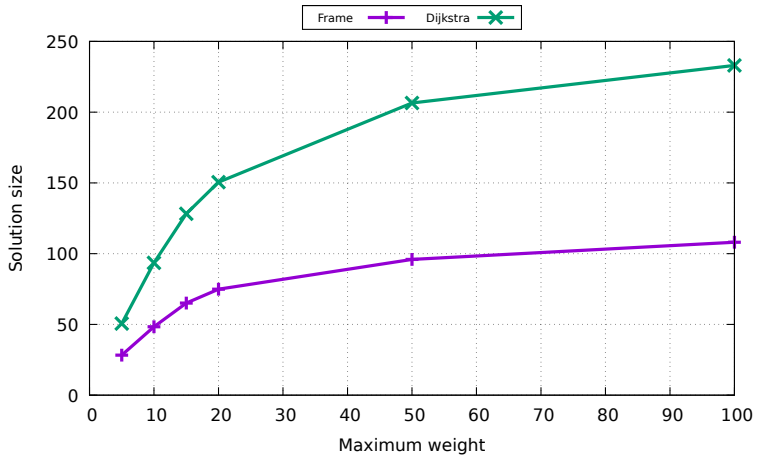


Figure 4.20 – S_u for anticorrelated distribution in $\overrightarrow{U_n}$.

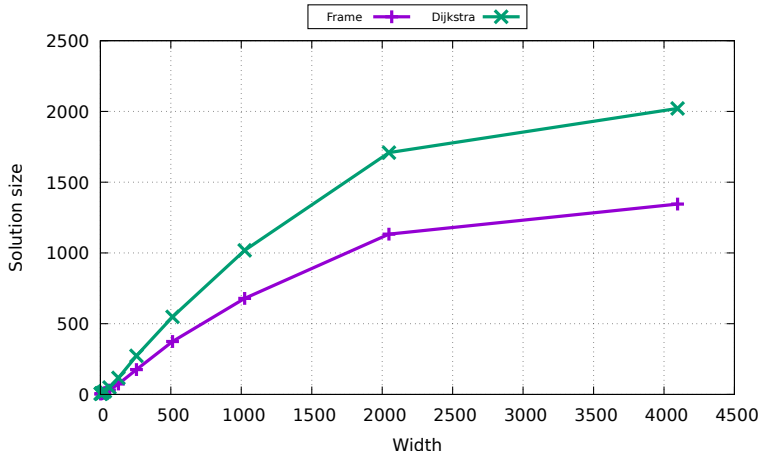


Figure 4.21 – Pareto set size in function of the width of R_n.

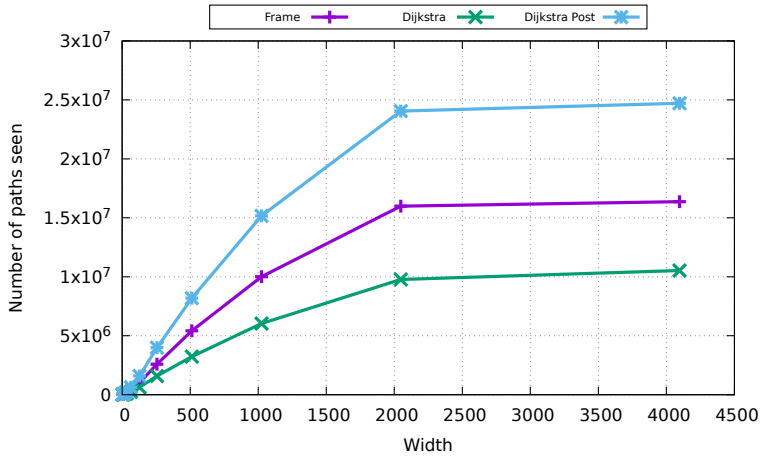


Figure 4.22 – Pruning early is efficient with R_n.

Since S_ε becomes quite large as the width increases (more than 1000 for a width greater than 2000), ε_{post} decreases drastically as shown in Figure 4.24.

4.3.3 Impact of ε

Up to now, we set up ε to 1. We now range $\varepsilon = 10^k$ with $k \in [\![-3, 1]\!]$ for grids of 10000 vertices. The arcs weights are drawn uniformly at random between 1 and 100. Sources and destinations are also randomly chosen.

Figure 4.25 shows that whenever ε goes to 0, Frame's output converges to $\mathcal{S}$. This implies a performance loss for Frame, which is well reflected in the Figure 4.26. In particular, if ε is very close to 0, Frame is slower than MC Dijkstra and is comparable to Dijkstra Post because S^*_ε reaches S. Finally, we notice logically that ε_{post} increases at the same time as ε, while remaining an order of magnitude below. We can see this correspondence in Figure 4.27.

Whenever ε is above 1, there is no drastic change. On one hand, any pair (P, Q) of Pareto paths in this experiment is a 2-coverage of S and on the other hand, the output of Frame is at least 60. It means that for large ε, due to the nu-

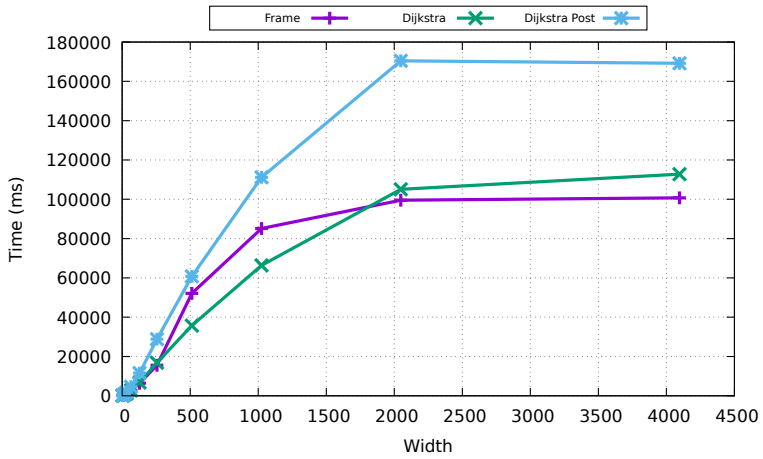


Figure 4.23 – Impact of pruning on the execution time with R_n.

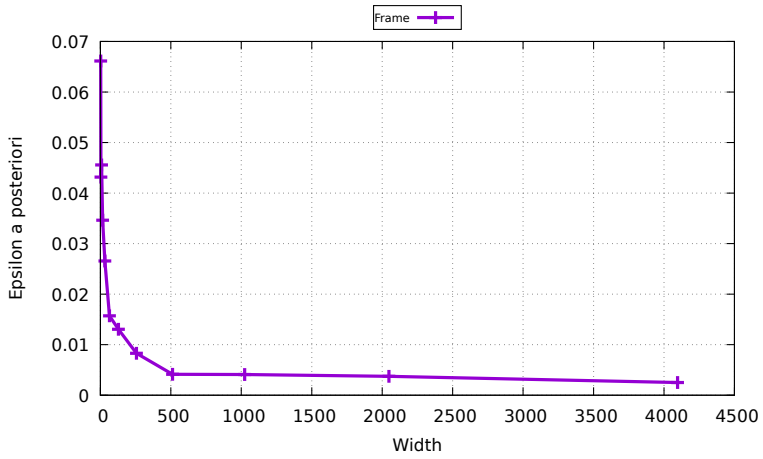


Figure 4.24 – A posteriori epsilon with R_n.

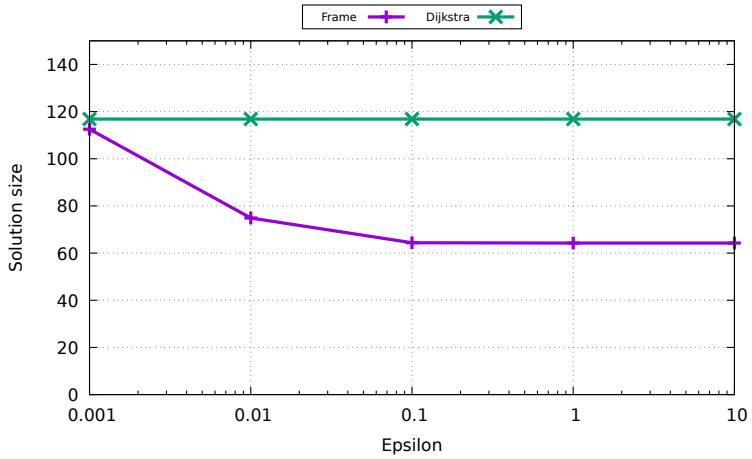


Figure 4.25 – Solution size depending on ε in $\overrightarrow{U_n}$

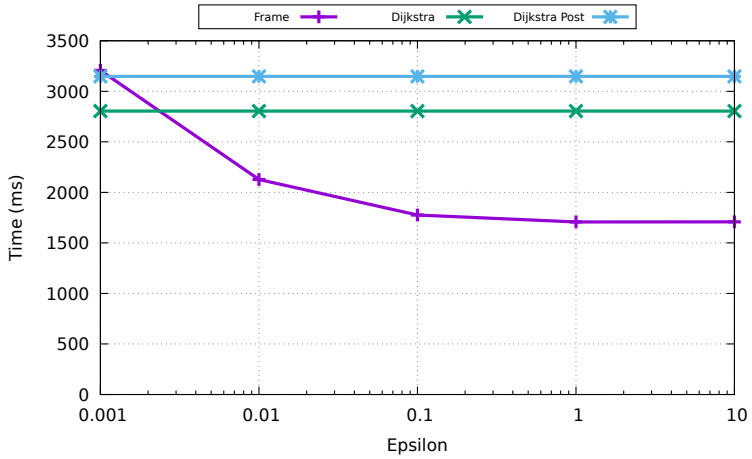


Figure 4.26 – Execution time depending on ε in $\overrightarrow{U_n}$

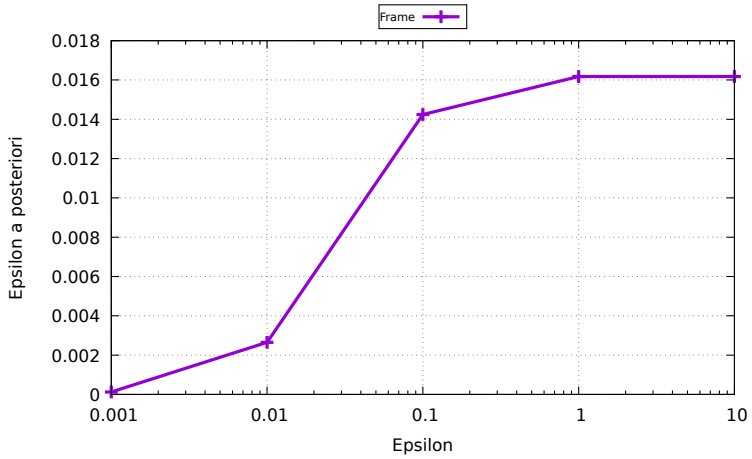


Figure 4.27 – The influence of ε on ε_{post}

merous number of processed ranks, FRAME is unable to summarize the Pareto set although $S^*_\varepsilon \leq 2$. This loss of efficiency is confirmed by ε_{post} being much smaller than ε for $\varepsilon > 1$. It shows the limitation of the Pareto compatibility property of FRAME.

4.3.4 Conclusion

We observe in practice that the gain of FRAME can be exponential for a particular class of graphs. Two concordant reasons: all paths have the same rank and both the Pareto sets are huge. When this number of ranks decreases, the gain also decreases, as does the size of the Pareto sets. Then, we increase the size of the Pareto sets by other methods (size of the graphs, variations of the weights, topology of the graphs). We still observe an interesting gain but between 40% and 100%. This time, however, the Pareto sets are of moderate size. The comparisons on real datasets yield the same results. Finally, we observe that the gain of FRAME is interesting for $\varepsilon \geq 0.01$.

Part II

Geometric graphs

Chapter 5

Preliminaries for geometric graphs

The ε-weak framing property defined Section 3.5.1 leads to the study of oriented proximity queries. In order to know if a path can be pruned, SECTOR computes if there exist other paths covering it in some sectors around it. Theta-graphs are particularly suitable graphs to answer this query. Given such a graph whose vertices are the weights of the paths, to determine if a path is covered in a given sector is in constant time. More precisely, we are interested in the specific tricriteria case, where the same rank path costs are in a plane. The graphs we are interested in are half-Theta-6, noted Θ_6^+.

First, we define these graphs in Section 5.1. Then, in Section 5.2, we prove some properties on the positioning of vertices neighbors. In order to take advantage of Θ_6^+ for the tricriteria shortest path computation, we propose two algorithms enabling the dynamic maintenance of a Θ_6^+, one for the insertion of a point, the other for the deletion (Chapter 6). Then, we present how to use these algorithms (Section 6.4).

In order to minimize the number of kept points, we propose an algorithm to enhance the Θ_6^+ structure (Section 7.1). Finally, in Section 7.2, we detail an existing algorithm for computing guaranteed small dominating sets. Combined, these last two algorithms allow to keep only a number of paths close to the minimum with respect to the ε-weak property (Section 7.3).

5.1 Definitions

In this part, we consider geometric graphs in the plane. We give ourselves a finite set of points in $\mathbb{R}^2$, which are connected according to certain rules to form graphs. We study the structural properties of these graphs and their manipulation.

5.1.1 Theta-graphs

Let $u \in \mathbb{R}^2$ and $k \geq 2$. We partition the plane into k cones around u, all with angles equal to $2\pi/k$.

Definition 46 (Cones)**.** *Let $u \in \mathbb{R}^2$. Let D be the ray (half-line) with initial point u, of direction the vector $(1, 0)$ and deprived of u. For any $i \in [\![1, k]\!]$, the i-th cone of u*

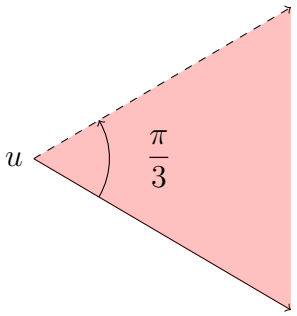


Figure 5.1 – Frontier convention.

is the set of points $\mathcal{C}_i(u) = \bigcup_{\frac{2\pi}{k}(i-1)\leq\alpha<\frac{2\pi}{k}i} r_{\alpha,u}(D)$, *with* $\forall\alpha \in [0, 2\pi[, r_{\alpha,u}$ *being the rotation of angle* α *and of center* u.

If k *is even, odd-numbered cones are said to be* positive *and the others are* negative.

The "Theta" of Theta-graph refers to the angle $\Theta = 2\pi/k$ of a cone. The cones of a Theta-graph are open on one of the two sides: if a cone goes from a ray D to the ray $r_{\Theta,u}(D)$, the latter is not in the cone (dotted line on Figure 5.1 for $\Theta = \pi/3$). The cones around u and $\{u\}$ thus form a partition of the plane.

We use the Euclidean distance, noted d_2 as well as the triangular distance d_Δ which is defined as follows.

Definition 47 (Triangular distance d_C in a cone)**.** *Let* $u \in \mathbb{R}^2$ *and* $\mathcal{C}$ *be a cone of* u. *Let* $v \in \mathcal{C}$ *and* $\mathcal{B}$ *be the bisector of* $\mathcal{C}$. *The* triangular distance in a cone $d_C(u, v)$ *from* u *to* v *is the Euclidean distance between* u *and the orthogonal projection of* v *onto* $\mathcal{B}$.

If $v, w \in \mathcal{C}$ *such that* $d_C(u, v) = d_C(u, w)$, *we say that* v *is nearer from* u *than* w *if the angle* $\widehat{w, u, v}$ *is in* $[0, \pi[$. *Informally, among equidistant points, the nearest is the first in clockwise order. This gives us the uniqueness of the nearest in a cone.*

Notice that $d_C(u, v)$ is well defined for any pair of points $u \neq v$ since v is necessarily in a cone of u.

Remark. *The triangular distance* d_C *is, strictly speaking, not a distance but can be qualified as a quasidistance. Indeed, this function is not symmetric, i.e.* $d_C(u, v) \neq d_C(v, u)$ *does not hold in general. On the other hand,* $d_C(u, v)+d_C(v, u)$ *is a distance.*

In order to visualize the previous definitions, consider the Figure 5.2. For $k = 8$, we observe the location of the eight cones around the point u. The positive cones are blue and the negative cones are red. In $\mathcal{C}_3(u)$, we have v' the orthogonal project of v on the bisector of the cone. So $d_C(u, v) = d_2(u, v')$. Similarly, in $\mathcal{C}_1(u)$, we have $d_C(u, w) = d_2(u, w')$. Notice that the point x is nearer from u than w in Euclidean distance but it is the opposite in triangular distance d_C.

Definition 48 (Theta-graph)**.** *Let* V *be a set of points in the plane. Let* $k \geq 2$ *and* $\theta = 2\pi/k$. *The Theta-*k*-graph of* V *is the graph* $\Theta_k(V) = (V, A)$ *such that for*

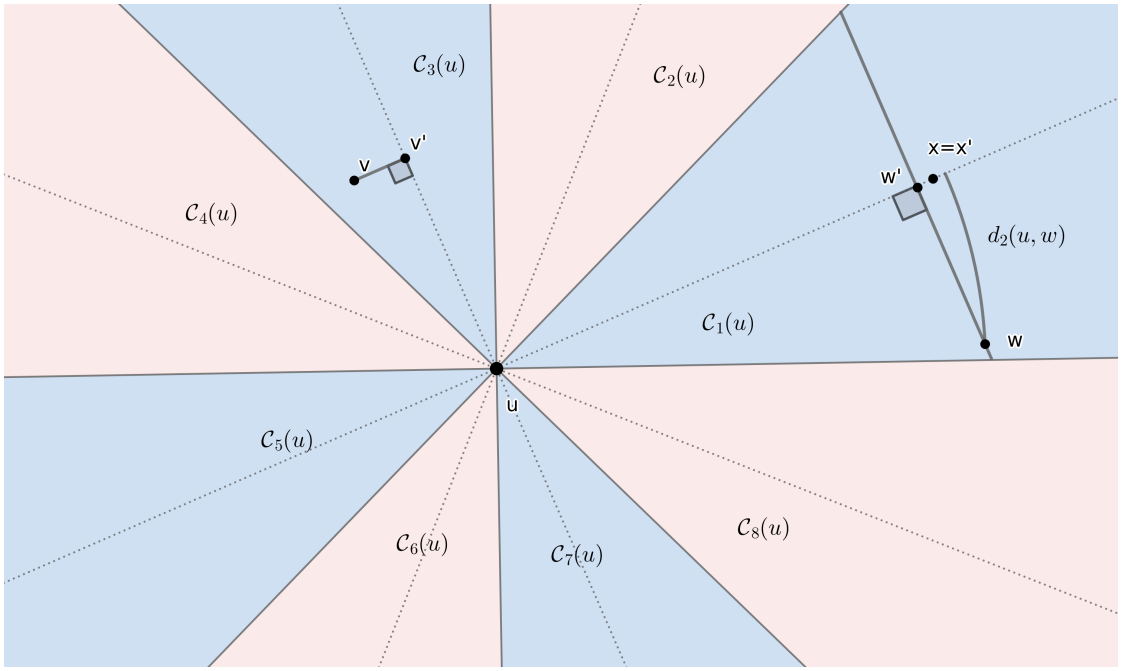


Figure 5.2 – Triangular distance d_C with $k = 8$ cones around u.

each u, v in V, $(u, v) \in A \Leftrightarrow v$ is the nearest point from u in a cone of u using the triangular distance d_C.

If k is even, the half-Theta-k-graph, denoted $\Theta_k^+(V)$, is the graph containing only the arcs outgoing in the positive cones.

Informally, u is connected to its nearest point in each cone. Recall that if two points are equidistant in a same cone, the nearest is the first in clockwise order.

Construction of a Theta-graph. Given a set of n points in the plane, we can compute a Theta-graph in time $O(n \log n)$ [TOG17]. The idea is to treat each cone direction separately. For a given cone direction, we sort the vertices according to a certain direction (orthogonal to an edge of the cone). Then we use a sweep line algorithm, i.e. a simple linear process of the points in this order. We provide ourselves with a balanced binary tree (Section A.2) containing some points already seen. When we process a given point, we are able to find the neighbor to be connected to in the considered cone by performing a search in the tree in logarithmic time. The update of the tree, in order to take into account the current point, is also in logarithmic time.

5.1.2 Half-Theta-6

For the tricriteria shortest path computation, we are interested in the particular case of half-Theta-6, noted Θ_6^+. Figure 5.3 illustrates a Θ_6^+ with 40 points.

The path costs are in 3D but we restrict ourselves to same rank paths. Thus, the graph associated to these costs is in a plane. We explain the link between the 2D

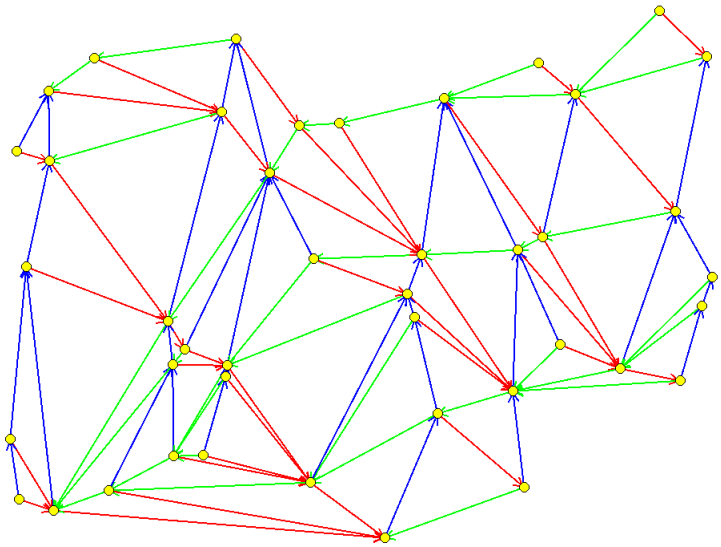


Figure 5.3 – Example of a Θ_6^+. Arcs have different colors depending on which cone they belong. Representation from an application[1] developed by Nicolas Bonichon.

coordinates of the graph vertices and the path costs 3D coordinates. In the following sections, the latter will often enable to express the proofs easier.

We embed the $\mathbb{R}^2$ plane in $\mathbb{R}^3$. For a point u, we note (u_X, u_Y), resp. (u_x, u_y, u_z), its coordinates in 2D, resp. in 3D. The embedding is not a canonical embedding, but from the following transformations:

$$\left.\begin{aligned} u_x &= u_X \\ u_y &= -\frac{u_X}{2} + \frac{\sqrt{3}}{2}u_Y \\ u_z &= -\frac{u_X}{2} - \frac{\sqrt{3}}{2}u_Y \end{aligned}\right| \begin{aligned} u_X &= u_x \\ u_Y &= \frac{u_y - u_z}{\sqrt{3}} \end{aligned}$$

The corresponding plane in $\mathbb{R}^3$ is the one of null rank, i.e. of equation $u_x + u_y + u_z = 0$, depicted in Figure 5.4. We note this plane $\mathcal{P}$. This can be generalized to any fixed rank r.

Remark. *For legibility reason, we made a rotation of angle $\pi/6$ with respect to Definition 48.*

Notation. *The six cones of a point u can be easily characterized in 3D as follows:*

$$\begin{aligned}
\mathcal{C}_x^+(u) = \mathcal{C}_1(u) &= \{v \in \mathcal{P}(V) | v_x > u_x, v_y < u_y, v_z \leq u_z\} \\
\mathcal{C}_y^+(u) = \mathcal{C}_3(u) &= \{v \in \mathcal{P}(V) | v_y > u_y, v_z < u_z, v_x \leq u_x\} \\
\mathcal{C}_z^+(u) = \mathcal{C}_5(u) &= \{v \in \mathcal{P}(V) | v_z > u_z, v_x < u_x, v_y \leq u_y\} \\
\mathcal{C}_x^-(u) = \mathcal{C}_2(u) &= \{v \in \mathcal{P}(V) | v_x < u_x, v_y > u_y, v_z \geq u_z\} \\
\mathcal{C}_y^-(u) = \mathcal{C}_4(u) &= \{v \in \mathcal{P}(V) | v_y < u_y, v_z > u_z, v_x \geq u_x\} \\
\mathcal{C}_z^-(u) = \mathcal{C}_6(u) &= \{v \in \mathcal{P}(V) | v_z < u_z, v_x > u_x, v_y \geq u_y\}
\end{aligned}$$

[1] `http://www.labri.fr/perso/bonichon/bounded/bounded.jar`

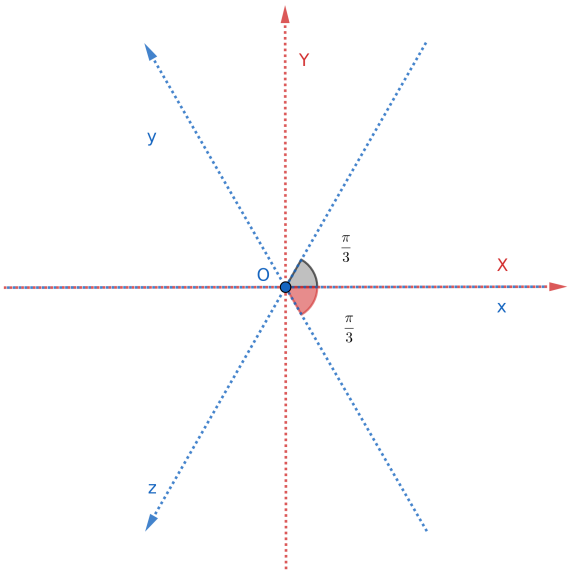


Figure 5.4 – Embedding in $\mathbb{R}^3$ (dashed axes) vs $\mathbb{R}^2$ (plain axes)

This will allow us to characterize more simply the belonging of a point to a cone in the following.

Remark. *When implementing Θ_6^+, one must be aware that coordinate conversions may induce computational approximations that can change the structure of Θ_6^+, especially when points have a common 3D coordinate which places them on cone edges.*

We have defined the triangular distance d_C in a cone (Definition 47). However, for half-Theta-graphs, only half the cones are considered. We thus generalize the notion of triangular distance in a cone for negative cones and provide a simple expression in 3D.

Definition 49 (Triangular distance d_Δ for Θ_6^+)**.** *Let $u, v \in \mathbb{R}^2$, $v \neq u$.*

- *If v is in a positive cone $\mathcal{C}$ of u, then the triangular distance is $d_\Delta(u, v) = d_C(u, v)$.*
- *Otherwise, v is in a negative cone $\mathcal{C}_i$ of u. Considering indices modulo 6, $\mathcal{C}_{i-1}$ and $\mathcal{C}_{i+1}$ are the two cones around $\mathcal{C}_i$. Let v', resp. v'', the projection over the bisector of $\mathcal{C}_{i-1}$, resp. $\mathcal{C}_{i+1}$. The triangular distance $d_\Delta(u, v)$ from u to v is the maximum between $d_2(u, v')$ and $d_2(u, v'')$.*

For v in any cone (positive or negative), this distance can be expressed as follows:

$$d_\Delta(u, v) = \max\{v_x - u_x, v_y - u_y, v_z - u_z\} \tag{5.1}$$

In the following, whenever we use a triangular distance without specifying d_C or d_Δ, we refer to the latter.

Remark. *In the* max*, one or two of the terms are non-negative, the others being non-positive since* $\texttt{rank}(u) = \texttt{rank}(v)$.

In a Θ_6^+, a point has at most three outgoing neighbors but can have many incoming neighbors. In order to refer to them, we introduce the following notations:

Notation. *Let* $u \in \mathbb{R}^2$*. We note* $E_x(u)$*, resp.* $E_y(u)$*, resp.* $E_z(u)$*, the ingoing neighbors of* u *in* $\Theta_6^+(V)$ *in* $C_x^-(u)$*, resp.* $C_y^-(u)$*, resp.* $C_z^-(u)$*.*

For a set of points V, $\Theta_6^+(V)$ was proven to be a triangulation of V in [Bon+10]. For even the external face to be triangular and to simplify the analysis of $\Theta_6^+(V)$, we add three virtual points $\infty_x, \infty_y, \infty_z$ placed far enough away in each direction from the points in V to ensure that each point has at least one point in each positive cone.

In practice, one can take any point O as a reference, consider the three bisecting half-lines of the positive cones of O and place one virtual point per half-line at the infinity. With these three points in V, $\Theta_6^+(V)$ is the graph such that each point of V not to infinity has exactly 3 outgoing neighbors, the nearest in each positive cone.

In the following, V **will be a set of points in the plane, with** $\infty_x, \infty_y, \infty_z \in V$**.**

The notations used in this chapter are summarized in Table 5.1.

5.2 First properties

We start by presenting some properties on Theta graphs, mainly on Θ_6^+, in order to better understand their behavior.

5.2.1 Order in half-Theta-6 cones

We study, for a given point in V, the relationship between different orders in the sets of ingoing neighbors in $\Theta_6^+(V)$.

Definition 50. *Let* $u \in V$*. We provide the sets of ingoing neighbors of* u *with the following orders:*

- $(E_x(u), \prec_x)$ *such that* $\forall v, w \in E_x(u), v_z < w_z \Rightarrow v \prec_x w$*. The set is ordered according to the* z *coordinate.*
- $(E_y(u), \prec_y)$ *such that* $\forall v, w \in E_y(u), v_x < w_x \Rightarrow v \prec_y w$*. The set is ordered according to the* x *coordinate.*
- $(E_z(u), \prec_z)$ *such that* $\forall v, w \in E_z(u), v_y < w_y \Rightarrow v \prec_z w$*. The set is ordered according to the* y *coordinate.*

Lemma 51. *Let* $u \in V$*. The sets* $(E_x(u), \prec_x)$*,* $(E_y(u), \prec_y)$ *and* $(E_z(u), \prec_z)$ *are totally ordered.*

V	set of points in the plane, containing ∞_x, ∞_y and ∞_z
d_2	Euclidean distance
d_C	triangular distance projection on the bisector of the concerned cone
d_Δ	triangular distance d_C in positive cones projection in neighboring positive cones otherwise
$\Theta_6^+(V)$	half-Theta-6 of V
$\mathcal{P}$	plane in $\mathbb{R}^3$ of equation $x+y+z=0$
$\mathcal{C}_x^+(u)$	$\{v \in \mathcal{P} \mid v_x > u_x, v_y < u_y, v_z \leq u_z\}$
$\mathcal{C}_y^+(u)$	$\{v \in \mathcal{P} \mid v_y > u_y, v_z < u_z, v_x \leq u_x\}$
$\mathcal{C}_z^+(u)$	$\{v \in \mathcal{P} \mid v_z > u_z, v_x < u_x, v_y \leq u_y\}$
$\mathcal{C}_x^-(u)$	$\{v \in \mathcal{P} \mid v_x < u_x, v_y > u_y, v_z \geq u_z\}$
$\mathcal{C}_y^-(u)$	$\{v \in \mathcal{P} \mid v_y < u_y, v_z > u_z, v_x \geq u_x\}$
$\mathcal{C}_z^-(u)$	$\{v \in \mathcal{P} \mid v_z < u_z, v_x > u_x, v_y \geq u_y\}$
$(E_i(u), \prec_i)$	set of ingoing neighbors of u in $\mathcal{C}_i^-(u)$ ordered by counterclockwise order and for $i \in \{x,y,z\}$
$\texttt{previous}(u,v)$	neighbor of u preceding v in counterclockwise order
$\texttt{next}(u,v)$	neighbor of u following v in counterclockwise order
$\texttt{first}_i(u)$	first neighbor of u in $E_i(u)$ in counterclockwise order for $i \in \{x,y,z\}$
$\texttt{last}_i(u)$	last neighbor of u in $E_i(u)$ in counterclockwise order for $i \in \{x,y,z\}$
Δ_u	degree of u
$S_{i,\varepsilon}^k(u)$	the at most k nearest points from u at triangular distance at most $(1+\varepsilon)\cdot u_i$ from u in $\mathcal{C}_i^+(u)$ and for $i \in \{x,y,z\}$

Table 5.1 – Notations for Theta-graphs.

Proof. We prove the Lemma for $(E_x(u), \prec_x)$. The proof is the same for $(E_y(u), \prec_y)$ and $(E_z(u), \prec_z)$.

Let $v, w \in E_x(u), v \neq w$. By contradiction, assume that $v_z = w_z$. Since $\texttt{rank}(v) = \texttt{rank}(w)$, then $v_y \neq w_y$. Without loss of generality, we assume that $v_y < w_y$. Thus $v_x > w_x$ and $v \in \mathcal{C}_x^+(w)$. Since $u_x > v_x > w_x$, $(w, u) \notin \Theta_6^+(V)$ and $w \notin E_x(u)$, which is absurd. Thus $v_z \neq w_z$. □

These total orders implies partial orders according to another coordinate.

Lemma 52. *Let $u \in V$.*

- *$(E_x(u), \prec_x)$ is sorted by non-increasing order according to the y coordinates, i.e. $\forall v, w \in E_x(u), v \prec_x w \Rightarrow v_y \geq w_y$.*

- *$(E_y(u), \prec_y)$ is sorted by non-increasing order according to the z coordinates, i.e. $\forall v, w \in E_y(u), v \prec_y w \Rightarrow v_z \geq w_z$.*

- *$(E_z(u), \prec_z)$ is sorted by non-increasing order according to the x coordinates, i.e. $\forall v, w \in E_z(u), v \prec_z w \Rightarrow v_x \geq w_x$.*

Proof. We prove the Lemma for $(E_x(u), \prec_x)$. The proof is the same for $(E_y(u), \prec_y)$ and $(E_z(u), \prec_z)$.

Let $v, w \in E_x(u)$. Without loss of generality, $v_z \leq w_z$. By Lemma 51, $v_z < w_z$. For the purpose of contradiction, assume that $v_y < w_y$. Since $\texttt{rank}(v) = \texttt{rank}(w)$, then $v_x > w_x$. Thus $v \in \mathcal{C}_x^+(w)$. Again, since $u_x > v_x > w_x$, $(w, u) \notin \Theta_6^+(V)$ and $w \notin E_x(u)$, which is absurd. Thus, $v_y \geq w_y$. □

Then, we notice that these orders correspond to the counterclockwise order.

Lemma 53. *Let $u \in V$. The sets $(E_x(u), \prec_x)$, $(E_y(u), \prec_y)$ and $(E_z(u), \prec_z)$ are sorted in strict counterclockwise order with respect to u. In particular, two neighbors of u cannot be aligned with u.*

Proof. We prove the Lemma for $(E_x(u), \prec_x)$. Let $v, w \in E_x(u)$ such that $v \prec_x w$. Thus $v_z < w_z$ and $v_y \geq w_y$ (Lemma 52). By definition of the determinant and the argument, we have:

$$det(v - u, w - u) > 0 \iff arg(v - u, w - u) \in]0; \pi[$$
$$det(v - u, w - u) < 0 \iff arg(v - u, w - u) \in]\pi; 2\pi[$$

The vertices v and w being in the same cone of u, the sign of the determinant gives us the counterclockwise order with respect to u. We use the conversions formula

from 2D to 3D, and the fact that each point has a null rank:

$$\begin{aligned}
det(v-u, w-u) &= (v_X - u_X)(w_Y - u_Y) - (v_Y - u_Y)(w_X - u_X) \\
&= \frac{1}{\sqrt{3}}[(v_x - u_x)(w_y - w_z - u_y + u_z) \\
&\quad -(v_y - v_z - u_y + u_z)(w_x - u_x)] \\
&= \frac{1}{\sqrt{3}}[-(v_y + v_z - u_y - u_z)(w_y - w_z - u_y + u_z) \\
&\quad +(v_y - v_z - u_y + u_z)(w_y + w_z - u_y - u_z)] \\
&= \frac{1}{\sqrt{3}}[(v_y - u_y + v_z - u_z)(w_z - u_z - w_y + u_y) \\
&\quad -(v_z - u_z - v_y + u_y)(w_y - u_y + w_z - u_z)] \\
&= \frac{2}{\sqrt{3}}\left[(v_y - u_y)(w_z - u_z) - (w_y - u_y)(v_z - u_z)\right]
\end{aligned}$$

However:

$$\begin{aligned}
u_y < w_y \leq v_y \\
u_z \leq v_z < w_z
\end{aligned}$$

Thus, by replacing the first v_y by w_y and the second v_z by w_z, we obtain that:

$$det(v-u, w-u) > \frac{2}{\sqrt{3}}\left[(w_y - u_y)(w_z - u_z) - (w_y - u_y)(w_z - u_z)\right] = 0$$

The inequality is strict since $v_z < w_z$ and $w_y - u_y > 0$. Therefore, we have shown that no pair of points had the same argument.

The proof is the same for $(E_y(u), \prec_y)$ and $(E_z(u), \prec_z)$, by applying the permutation (x, y, z) for E_y and (x, z, y) for E_z. □

Now that we know better how the incoming neighbors of a point are distributed, we defined how to process them.

Definition 54. *Let $u \in V \backslash \{\infty_x, \infty_y, \infty_z\}$ and $v_1, \ldots, v_k$ its neighbors (ingoing and outgoing) in counterclockwise order in $\Theta_6^+(V)$. The indices of the vertices are expressed modulo k. We define:*

- $\texttt{previous}(u, v_i) = v_{i-1}$*, the neighbor of u preceding v_i in counterclockwise order.*
- $\texttt{next}(u, v_i) = v_{i+1}$*, the neighbor of u following v_i in counterclockwise order.*
- $\texttt{first}_x(u) = \underset{w \in E_x(v)}{\arg\min}\, w_z$ *the first vertex of $E_x(u)$ in counterclockwise order. If $E_x(u)$ is empty, then it is the outgoing vertex of u in $C_y^+(u)$, i.e. the neighbor preceding the cone $C_x^-(u)$. Same for $\texttt{first}_y(u)$ and $\texttt{first}_z(u)$.*

- $\texttt{last}_x(u) = \underset{w \in E_x(v)}{\arg\max}\, w_z$ *the last vertex of* $E_x(u)$ *in counterclockwise order. If* $E_x(u)$ *is empty, then it is the outgoing vertex of* u *in* $C_z^+(u)$*, i.e. the neighbor following the cone* $C_x^-(u)$*. Same for* $\texttt{last}_y(u)$ *and* $\texttt{last}_z(u)$*.*

By Lemma 53, these functions are well defined. Notice that the definitions of `previous` and `next` are not restricted to a particular cone: $\texttt{next}(u, v)$ can be in a different cone of u than v.

Lemma 55. *With a slightly improved version of a combinatorial map (Section A.4) to represent* $\Theta_6^+(V)$*,* `previous`*,* `next`*,* `first` *and* `last` *are in constant time.*

Proof. Let (D, σ, α) a combinatorial map representing $\Theta_6^+(V)$. For both `previous` and `next`, the information is local, it is sufficient to apply σ to rotate around a vertex. For `first` and `last`, we add, for each incoming cone of each vertex of V, two pointers, one to each corresponding neighbor. If an arc is deleted or added, their update is done in constant time. □

Lemma 56. *Let* $u \in V \backslash \{\infty_x, \infty_y, \infty_z\}$*. Let* $v \in V$ *be a neighbor of* u *and* $w = \texttt{next}(u, v)$*. The angle* $\widehat{vuw}$ *is smaller than* π *and* w *is connected to* v*.*

Proof. Two given outgoing cones of u are strictly included in a half plane. Let w' be the outgoing vertex of u following v in counterclockwise order, and v' being either the precedent if v is an ingoing vertex of u, or v itself. The angle $\widehat{vuw}$ is smaller than or equal to $\widehat{v'uw'}$ which is smaller than π.

Furthermore, v and w are connected since $\Theta_6^+(V)$ is a triangulation of the plane. □

5.2.2 Nearest point

In this section, we study the connectivity of a point with its nearest point in the Theta-graph, depending on the distance considered.

With Euclidean distance

In a Theta-k with $k \geq 6$, any point is connected with the nearest point(s) in Euclidean distance.

Lemma 57. *Let* $u \in V$*. Let* $v \in V \backslash \{u\}$ *be a nearest point in Euclidean distance. Then for* $k \geq 6$*,* $(v, u) \in \Theta_k(V)$*.*

Proof. The proof is illustrated in Figure 5.5, for $k = 7$. Consider the open disk D according to the Euclidean distance, of center u and radius $d_2(u, v)$. By definition of v, the only point of V in D is u. Let $\mathcal{C}$ be the cone of v containing u and let T be the triangle defined as the part of $\mathcal{C}$ of triangular distance less than or equal to $d_\triangle(v, u)$. In the figure, T is the union of the blue zone and the red zone. We show that T is included in D. It is sufficient to conclude since $D \cap V = \{u\}$. Be careful that T is open on one side, i.e. it does not contain this side (the dashed one in the figure), by definition of $\mathcal{C}$.

Let $v' \in T$. We define the three following points in $\mathcal{C}$ at triangular distance $d_\triangle(v, u)$:

- B the point on the bisector of $\mathcal{C}$,
- w the point on the edge of $\mathcal{C}$ that belongs to $\mathcal{C}$,
- w' the point of the ray of origin v containing v'.

We prove that the triangle $(v, w, w') \setminus \{v\} \subset D$. This triangle is depicted in blue. Let α be the measure of the angle $\widehat{wvB}$ and β the measure of $\widehat{uvB}$ (absolute values). By definition of the points, since $k \geq 6$, we have:

$$\begin{cases} \alpha = \dfrac{\pi}{k} \quad \Longrightarrow \quad 0 < \alpha \leq \dfrac{\pi}{6} \\ 0 \leq \beta \leq \alpha \quad \Longrightarrow \quad 0 \leq \beta \leq \dfrac{\pi}{6} \end{cases} \tag{5.2}$$

Furthermore:

$$\tan(\alpha) = \frac{d_2(B,w)}{d_2(B,v)}\,, \qquad \cos(\beta) = \frac{d_2(B,v)}{d_2(u,v)} \qquad \text{and} \qquad \sin(\beta) = \frac{d_2(B,u)}{d_2(u,v)}$$

We can deduce that:

$$d_2(B,u) = \sin(\beta) d_2(u,v) \qquad \text{and} \qquad d_2(B,w) = \tan(\alpha)\cos(\beta) d_2(u,v)$$

Let $f_\alpha : \beta \mapsto \sin(\beta) + \tan(\alpha)\cos(\beta)$. The derivative f'_α of f_α is given by the formula:

$$f'_\alpha(\beta) = \cos(\beta) - \tan(\alpha)\sin(\beta)$$

With the Inequalities 5.2, we have that, for any $0 \leq \beta \leq \alpha$:

$$f'_\alpha(\beta) \geq \frac{\sqrt{3}}{2} - \frac{1}{2} \cdot \frac{1}{\sqrt{3}} > 0$$

proving that the function f_α is increasing in the interval $[0, \alpha]$. By definition of f_α, we also notice that: $f_\alpha(\alpha) = 2\sin(\alpha) \leq 1$.

The edge opposite to the one containing w is not in $\mathcal{C}$. There are two cases :

- If u lies on the other side of B than w (B included), we have $\beta < \alpha$. Thus:

$$\begin{cases} d_2(u,w') \leq d_2(u,w) = d_2(B,u) + d_2(B,w) \\ d_2(B,u) + d_2(B,w) = f_\alpha(\beta) \cdot d_2(u,v) < f_\alpha(\alpha) \cdot d_2(u,v) \leq d_2(u,v) \end{cases}$$

  Therefore: $d_2(u,w') \leq d_2(u,w) < d_2(u,v)$.

- Otherwise:

$$\begin{cases} d_2(u,w) < d_2(B,u) + d_2(B,w) \\ d_2(u,w') < d_2(B,u) + d_2(B,w) \\ d_2(B,u) + d_2(B,w) = f_\alpha(\beta) \cdot d_2(u,v) \leq f_\alpha(\alpha) \cdot d_2(u,v) \leq d_2(u,v) \end{cases}$$

  Therefore: $d_2(u,w) < d_2(u,v)$ and $d_2(u,w') < d_2(u,v)$.

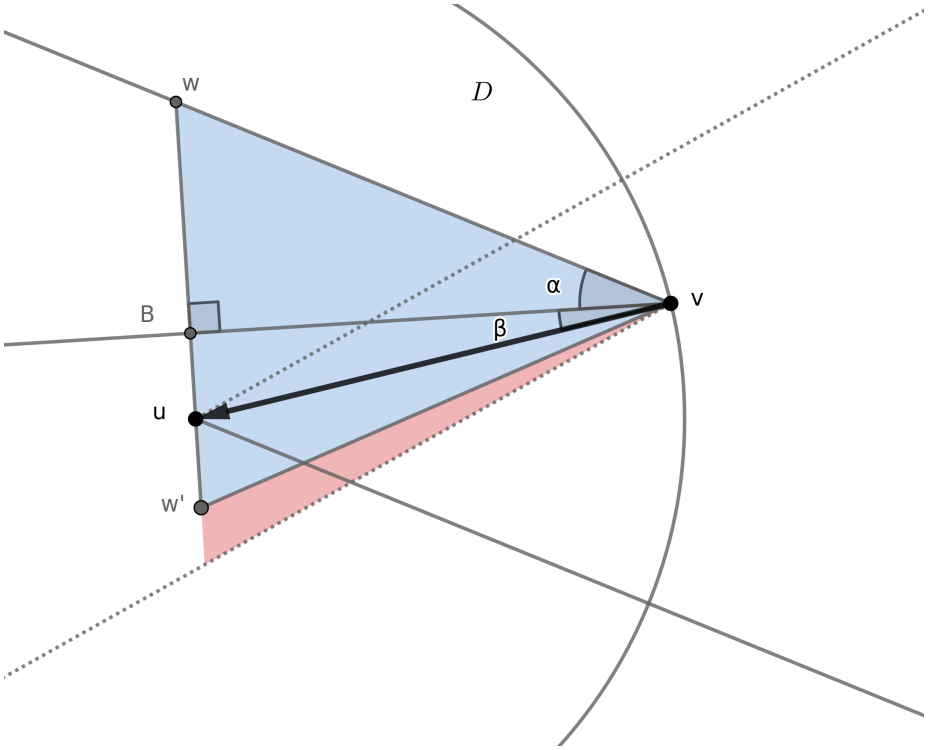


Figure 5.5 – A nearest point in Euclidean distance is connected.

In both cases, w and w' belong to D. Since D is convex, the triangle $(v, w, w')\backslash\{v\}$ is included in D. We have shown that, for any $v' \in T$, $v' \in D$. Thus $T \subseteq D$. □

Remark. *In $\Theta_6^+(V)$, if the nearest point v in Euclidean distance is in an outgoing cone of u, the two points are not necessarily connected. They might even be at an arbitrary hop distance.*

With triangular distance

In a Θ_6^+, any point u is connected to the nearest point in each positive cone. We notice that it may be also true in negative cones. Since we do not restrict to one cone, there is no uniqueness of a nearest point. Relying on Lemma 58, we show that any point in connected to a nearest point according to d_C (Lemma 59). Afterwards, we show the same property according to d_Δ (Lemma 60).

Lemma 58. *Let $u \in V$. There exists $v \in V\backslash\{u\}$ such that (u, v) and (v, u) are in $\Theta_6(V)$.*

Proof. Let $v = \arg\min\{d_C(u, w) | w \in V\backslash\{u\}\}$ be a nearest point from u in $V\backslash\{u\}$. Let $d = d_C(u, v)$.

- If v is in a negative cone of u, we consider without loss of generality that v is in $\mathcal{C}_z^-(u)$. By definition of v, $(u, v) \in \Theta_6(V)$. We show that (v, u) is in $\Theta_6^+(V)$, and thus in $\Theta_6(V)$. Let $w \in \mathcal{C}_z^+(v)$ such that $(v, w) \in \Theta_6^+(V)$. By definition of

v and w, we have:

$$\begin{cases} v_x > u_x \\ v_y \geq u_y \\ v_z < u_z \end{cases} \quad \text{and} \quad \begin{cases} w_x < v_x \\ w_y \leq v_y \\ w_z > v_z \end{cases} \tag{5.3}$$

Thus $u \in \mathcal{C}_z^+(v)$. We show that $w = u$. By contradiction, since $w_z \leq u_z$, then: $w \in \mathcal{C}_x^+(u) \cap \mathcal{C}_y^+(u) \cap \mathcal{C}_z^-(u)$.

- If $w \in \mathcal{C}_x^+(w)$, then $d_C(u,w) = w_x - u_x$. From Equation 5.3, we can deduce that:

$$d_C(u,w) = w_x - u_x < v_x - u_x = u_y + u_z - v_y - v_z \leq u_z - v_z = d_C(u,v).$$

- If $w \in \mathcal{C}_y^+(w)$, then $d_C(u,w) = w_y - u_y$. From Equation 5.3, we can deduce that:

$$d_C(u,w) = w_y - u_y \leq v_y - u_y = u_x + u_z - v_x - v_z < u_z - v_z = d_C(u,v).$$

- If $w \in \mathcal{C}_z^-(w)$, then $d_C(u,w) = u_z - w_z$. From Equation 5.3, we can deduce that:

$$d_C(u,w) = u_z - w_z \leq u_z - v_z = d_C(u,v).$$

In all cases, $d_C(u,w) < d_C(u,v)$. Thus, $(v,u) \in \Theta_6^+(V)$.

- If v is in a positive cone of u, then $(u,v) \in \Theta_6^+(V)$ by definition of v. Similarly to the previous case, we have that $(v,u) \in \Theta_6^-(V)$, the subgraph of $\Theta_6(V)$ containing only the arcs outgoing in negative cones. Thus, $(v,u) \in \Theta_6(V)$,

□

Lemma 59. *Let $u \in V$. There exists $v \in V \backslash \{u\}$ a nearest point from u according to d_C, such that u and v are connected in $\Theta_6^+(V)$.*

Proof. From Lemma 58, (u,v) and (v,u) are in $\Theta_6(V)$. Thus, by definition of half-Theta-graphs, either (u,v) or (v,u) is in $\Theta_6^+(V)$. □

This results is still true according to d_Δ. It is obvious in positive cones. However, notice that d_C and d_Δ are not the same in negative cones.

Lemma 60. *Let $u \in V$. There exists $v \in V \backslash \{u\}$ a nearest point from u according to d_Δ, such that u and v are connected in $\Theta_6^+(V)$.*

Proof. Let $d = \min\{d_\Delta(u,v) | v \in V \backslash \{u\}\}$ be the triangular distance from u to its nearest point in V. Let $\mathcal{M} = \{v \in V | d_\Delta(u,v) = d\}$ be the set of points of minimum triangular distance from u. We need to prove that there exists $v \in \mathcal{M}$, such that u is connected to v in $\Theta_6^+(V)$.

If there exists $v \in \mathcal{M}$ and in an outgoing sector of u, then by definition of $\Theta_6^+(V)$, the point u is connected to a point in $\mathcal{M}$. Otherwise, without loss of generality,

assume that $\mathcal{M} \cap \mathcal{C}_z^-(u) \neq \emptyset$. Let $v = \arg\min\{w_x | w \in \mathcal{M} \cap \mathcal{C}_z^-(u)\}$, i.e. the point with minimal x coordinate. We show that $(v, u) \in \Theta_6^+(V)$. By definition of v, we have:

$$\begin{cases} v_z < u_z \\ v_x > u_x \\ v_y \geq u_y \end{cases}$$

Thus $u \in \mathcal{C}_z^+(v)$. Let us assume that there exists $w \in \mathcal{C}_z^+(v)$ such that $w \neq u$ and $(v, w) \in \Theta_6^+(V)$. Then $d_\Delta(v, w) \leq d_\Delta(v, u)$. We show that $d_\Delta(u, w) \leq d_\Delta(u, v)$.

According to Equation 5.1:

$$\begin{cases} d_\Delta(u, v) &= \max\{v_x - u_x, v_y - u_y\} \\ d_\Delta(v, w) &= w_z - v_z \\ d_\Delta(v, u) &= u_z - v_z \end{cases}$$

Thus $w_z \leq u_z$, giving $w_z - u_z \leq 0$. We can conclude that $d_\Delta(u, w) = \max\{w_x - u_x, w_y - u_y\}$. Since $w \in \mathcal{C}_z^+(v)$,

$$\begin{cases} w_x < v_x \\ w_y \leq v_y \end{cases}$$

Thus :

$$\begin{cases} w_x - u_x < v_x - u_x \\ w_y - u_y \leq v_y - u_y \end{cases} \tag{5.4}$$

Therefore, $d_\Delta(u, w) \leq d_\Delta(u, v)$. By definition of v, $d_\Delta(u, w) = d_\Delta(u, v)$, i.e.

$$\max\{w_x - u_x, w_y - u_y\} = \max\{v_x - u_x, v_y - u_y\}$$

With the Equation 5.4, $d_\Delta(u, w) = w_y - u_y$ and thus $d_\Delta(u, v) = v_y - u_y$. We can deduce that $w_y = v_y$. The point w cannot lie:

- in $\mathcal{C}_z^-(u)$ by minimality if v_x,
- in $\mathcal{C}_y^+(u)$ since it is an outgoing cone of u.

Therefore, w lie on the edge of the cone $\mathcal{C}_x^-(u)$ and we have $w_z = u_z$. Since w is the outgoing neighbor of v in $\mathcal{C}_z^+(v)$, we have $w_y \leq u_y$, and thus $w_y = u_y$. Combined with $w_z = u_z$ and the equality of ranks, we obtain $w = u$, a contradiction. □

Chapter 6

Dynamic Theta graphs

We recall that the construction of $\Theta_k(V)$ is in $O(kn \log n)$ [NS07]. Thus, the construction of $\Theta_6^+(V)$ is in $O(n \log n)$. But this construction requires to know V from the beginning. We would like to be able to perform insertions and deletions in V on the fly, while preserving the structure of Θ_6^+.

We first present related work for insertions in a L_2 Delaunay triangulation in Section 6.1. Secondly, we propose a similar insertion algorithm for Theta-graphs in Section 6.2 and a deletion one in Section 6.3.

We can easily adapt these algorithms to handle Θ_6^- (Theta-graphs with only the arcs outgoing in negative cones). Therefore, these algorithms also enable the dynamic maintenance of Θ_6 since the latter are the union of a Θ_6^+ and a Θ_6^-. We recall a fundamental property of Θ_6^+ used in the following: they are planar [Bon+10].

6.1 Related work

In order to better understand how our insertion algorithm works for Theta-graphs, we give the main ideas of the Guibas algorithm [GS83] for Delaunay Triangulation in Euclidean distance.

Guibas algorithm. This algorithm adds a new point u in a Delaunay triangulation with the Euclidean distance. It starts by finding the enclosing triangle of u: this is the request *point location*. Note that this triangle is not necessarily well defined in general.

- The authors consider that u is within the convex hull of the triangulation.
- u can be at the intersection of two or even three triangles. If it is at the intersection of three of them, then u is already in the triangulation, no need to add it. If it is at the intersection of two triangles, the following reasoning is applied to the quadrilateral formed by the two triangles instead of one triangle.

Then, the points of the triangle (or quadrilateral) are connected to u. These edge creations potentially challenge the correctness of existing edges of the triangle. This questioning is tested with a constant time procedure called *InCircle* (computation of a determinant of dimension 4), potentially resulting in the deletion of the edge. If

in a quadrilateral $uvww'$, the edge $\{v, w'\}$ is deleted, then the edge $\{u, w\}$ replaces it. This operation is called a swap: it challenges the correctness of other edges of the quadrilateral on which the same principle is applied recursively. The order does not matter. To store the edges to be tested, a stack may be used to simulate a recursive algorithm. The algorithm is linear in the number of connections added.

Point Location. Independently of the insertion problem, the *Point Location Problem* is also well studied. In the general case, given a polygonal partition P of the plane and any point u, the *point location* query returns the polygon(s) containing the point u. In the book [TOG17] are presented various methods for the dimensions 1 and 2, with dynamic or static point sets. We provide an overview of some relevant methods.

For a single point location without precomputation in a Delaunay triangulation, in dimension $d = 2$ or $d = 3$, Mücke et al. present a randomized algorithm whose complexity is said to be close to $O(n^{1/(d+1)})$ [MSZ99]. For example, with $d = 3$ and some fixed parameters, the mean time complexity is in $O\left(n^{1/4}\left(\frac{\log n}{\log\log n}\right)^{3/4}\right)$. In order to perform several point locations in a row with a set of static points, [Kir83] presents a method including precomputation with the following complexities: the precomputation is in $O(n \log n)$ time and in a linear space, while a query is in $O(\log n)$. However, and this is the context in which this query is needed, the set of points can be dynamic.

For a polygonal partition $\mathcal{P}$ of the plane, we therefore want a structure $S(\mathcal{P})$ on which we can perform two operations:

- the *point location* query, giving the polygon containing a given point,
- the update of the structure, computing $S(\mathcal{P}')$ with $\mathcal{P}'$ being $\mathcal{P}$ to which a segment has been inserted or removed.

To the best of our knowledge, there is no method that performs these two operations in $O(\log n)$. Different tradeoffs are possible and the one that seems the most relevant guarantees queries in $O(\log n(\log\log n)^2)$ time and structure updates in $O(\log n \log\log n)$ [CN18] time. In the same paper are also presented other methods that guarantee incomparable tradeoffs. For instance, one of them allows to obtain a complexity in $O(\log(n)^{1+\varepsilon})$ (resp. $O(\log n)$) for queries (resp. updates) this for any $\varepsilon > 0$. These two complexities are interchangeable. The latter method has a linear space complexity.

Point Location for insertion. Back to the case we are interested in, i.e. the insertion of a point in a L_2 Delaunay triangulation. In the Guibas algorithm, a point insertion implies the modification of $O(\Delta_u)$ arcs and as many updates of the structure associated to the point location.

Guibas et al. propose a new structure for the point location query. For the insertion of n points in a uniform random order, the mean time complexity is in $O(n \log n)$, the factor Δ_u having disappeared because the average degree is constant in a triangulation. As for the space complexity, it is also in $O(n)$ on average but

the worst case is in $O(n^2)$. This method has the advantage of preserving the history of previous insertions. Kao [KMS91] presents another method which guarantees the same complexities except the worst case in space which becomes linear. Nevertheless, this latter method has the disadvantage of not allowing to easily recover the modification history.

These algorithms have been adapted to a distributed context, guaranteeing a linear complexity in the number of changes to operate [SSB05].

6.2 Insertion algorithm

We propose an efficient algorithm to insert a vertex in Θ_6^+. We recall that a Θ_6^+ is also a Delaunay triangulation, but with the triangular distance instead of the Euclidean distance, and whose edges are directed. We propose an algorithm that performs a sequence of arc swaps similar to the Guibas algorithm. However, no test is based on distance, the properties of triangulation and keeping a unique outgoing neighbor in each cone being sufficient.

Outline. Initially, we give ourselves the Θ_6^+ of the set of points V and a point u to insert. We start by performing a `PointLocation` to find the enclosing triangle(s) of u, of which we connect each point to u. The orientation of the arcs depends on the relative position of the pairs of points concerned. The general idea is then to successively redirect the arcs which are not in $\Theta_6^+(V \cup \{u\})$. Each of them is replaced by an arc for which one end is u.

We use a structure $\mathcal{G} = (V, E, F)$ with $(V, E \cup F)$ being a graph that is a triangulation of the plane, and F containing the arcs being processed. The latter are new arcs being in $\Theta_6^+(V \cup \{u\})$ but not in $\Theta_6^+(V)$.

- At the beginning, the arcs connecting u to its enclosing triangle are placed in F. Any point of V has exactly one arc in each outgoing cone and these arcs are in E.

- Then, an arc from F is moved to E. The insertion of this arc may involve the deletion of another arc in E. Indeed, if the source of the added arc already has an outgoing arc in the same cone, the latter arc is deleted. If an arc is deleted, then $(V, E \cup F)$ is no longer a triangulation. If in the quadrilateral $uvww'$, the arc (v, w') has been deleted, then (u, w) or (w, u) is added in F, depending on the relative position of u and w.

- The algorithm ends when F is empty.

For the `PointLocation` query, we only consider the complexity of adding a single point and not the average over the incremental insertion of all points. The method from [CN18] allows a query in $O(\log n(\log\log n)^2)$ and an update in $O(\log n \log\log n)$ while guaranteeing a linear memory complexity. We will thus use this method to minimize the worst-case complexity. The associated data structure for the point location query is noted P_L in the algorithm description.

Almost TD graphs. We consider bicolored simple graphs $\mathcal{G} = (V, E, F)$ defined by the set V of points in the plane and two *disjoint* arc sets E and F. We note $E' = E \cup F$. The following definition corresponds to the state in which a bicolor graph is during the algorithm.

Definition 61 (almost TD)**.** *We say that a bicolored graph $\mathcal{G} = (V, E, F)$ is* almost TD *with respect to some point $u \in V$, noted $\texttt{almostTD}_u$, when:*

1. *the graph (V, E') is a triangulation,*
2. *each point $v \neq u$ not being at infinity has exactly one neighbor in E in each outgoing cone,*
3. *any arc of F has u as one of its extremities.*

Recall that we add points at infinity in the directions of the outgoing cones in order to guarantee that every point has an outgoing neighbor in each cone and that a Θ_6^+ is a triangulation of the plane.

Initialization of an almost TD graph. We note $\tilde{\Theta}_6^+(V)$ the bicolored graph corresponding to the half-Theta-6 graph on V, with all arcs in E (and thus none in F).

Lemma 62. *The bicolored graph $\tilde{\Theta}_6^+(V)$ is $\texttt{almostTD}_u$ for any node $u \in V$.*

Proof. This follows from the fact that F is empty and that a half-Theta-6 graph with three points at infinity framing the other points is a triangulation. □

Operations on almost TD graphs. In order to present the pseudo code of our algorithm and to analyze it, we introduce the following operations:

Definition 63. *We define the following five operations on a bicolored graph $\mathcal{G} = (V, E, F)$:*

- $\mathcal{G}.\texttt{insert}_E(v, w)$*, resp.* $\mathcal{G}.\texttt{insert}_F(v, w)$*, inserts the arc (v, w) to the set E, resp. F. It requires that (v, w) is not in $E' = E \cup F$.*
- $\mathcal{G}.\texttt{remove}(v, w)$ *removes the arc (v, w) from its set. It requires that $(v, w) \in E'$.*
- $\mathcal{G}.\texttt{move}(v, w)$*, where $(v, w) \in F$, moves the arc (v, w) from F to E. It is used only when $v = u$.*
- $\mathcal{G}.\texttt{replace}(v, u')$ *requires that $u' = u$ with u in the outgoing cone $\mathcal{C}$ of v and that $(v, u) \in F$. Furthermore, the operation is valid only if $\mathcal{G}$ is $\texttt{almostTD}_u$. Let w be the outgoing neighbor in E of v in the cone $\mathcal{C}$. $\mathcal{G}$ being a triangulation, the arc (v, w) belongs to two triangles, whose third vertices are v' and u (because u and w are the only neighbors of v in the cone $\mathcal{C}$ in E'). By triangulation of $\mathcal{G}$, v' and u are not neighbors in E' (proven in Lemma 65). The operation does:*
  - *delete (v, w) from E,*

- *move* $e = (v, u)$ *from* F *to* E,
- *insert* (u, v'), *resp.* (v', u), *to* F *if* v' *is in an outgoing cone of* u, *resp. if* u *is in an outgoing cone of* v'.

Given these operations, we present our algorithm for inserting a point into a half-Theta-6 in Algorithm 14.

Input: $\Theta_6^+(V)$ for a given set of points V and $u \notin V$
Output: $\tilde{\Theta}_6^+(V \cup \{u\})$

$\mathcal{G} \leftarrow \tilde{\Theta}_6^+(V)$ with u added to the set of nodes
$S \leftarrow \texttt{PointLocation}(P_L, u)$
if $|S| = 4$ **then**
  Let (v, v') be the arc on which u lies
  $\mathcal{G}.\texttt{insert}_E(v, u)$
  $\mathcal{G}.\texttt{insert}_E(u, v')$
  $\mathcal{G}.\texttt{remove}(v, v')$
  $S \leftarrow S \backslash \{v, v'\}$
foreach $v \in S$ **do**
  $\mathcal{G}.\texttt{insert}_F(u, v)$ `// or` $\mathcal{G}.\texttt{insert}_F(v, u)$ `depending on the relative positions of` u `and` v
while $F \neq \emptyset$ **do**
  Let $a = (v, w)$ be an element of F `//` $v = u$ `or` $w = u$ `when` $\mathcal{G}$ `is` $\texttt{almostTD}_u$
  if $w = u$ **then**
    $\mathcal{G}.\texttt{replace}(v, u)$
  else
    $\mathcal{G}.\texttt{move}(u, w)$
return $\mathcal{G}$

Algorithm 14: Insertion of a vertex in a Θ_6^+.

In Lemmas 64 and 65, we analyze how the almost TD property behaves with respect to the operations that are used in the iteration. The relation between the almost TD property and the initialization of the algorithm will be studied in Lemma 67.

Lemma 64. *Assume* $\mathcal{G} = (V, E, F)$ *is a bicolored graph* $\texttt{almostTD}_u$ *with* $u \in V$. *Let* $v \in V$ *be in an outgoing cone of* u *such that* $(u, v) \in F$. *Then, after* $\mathcal{G}.\texttt{move}(u, v)$, $\mathcal{G}$ *is still* $\texttt{almostTD}_u$.

Proof. (V, E') is still a triangulation as it was not changed by the operation. Moreover, the neighbors in E in outgoing cones did not change except for u, which is not concerned by the second property of $\texttt{almostTD}_u$. Finally, the third property remains true since no arc has been inserted in F. □

Lemma 65. *Assume* $\mathcal{G} = (V, E, F)$ *is a bicolored graph* $\texttt{almostTD}_u$ *with* $u \in V$. *Let* $v \in V$ *be in an ingoing cone of* u *such that* $(v, u) \in F$. *Then the operation* $\mathcal{G}.\texttt{replace}(v, u)$ *is well defined and the resulting bicolored graph is still* $\texttt{almostTD}_u$.

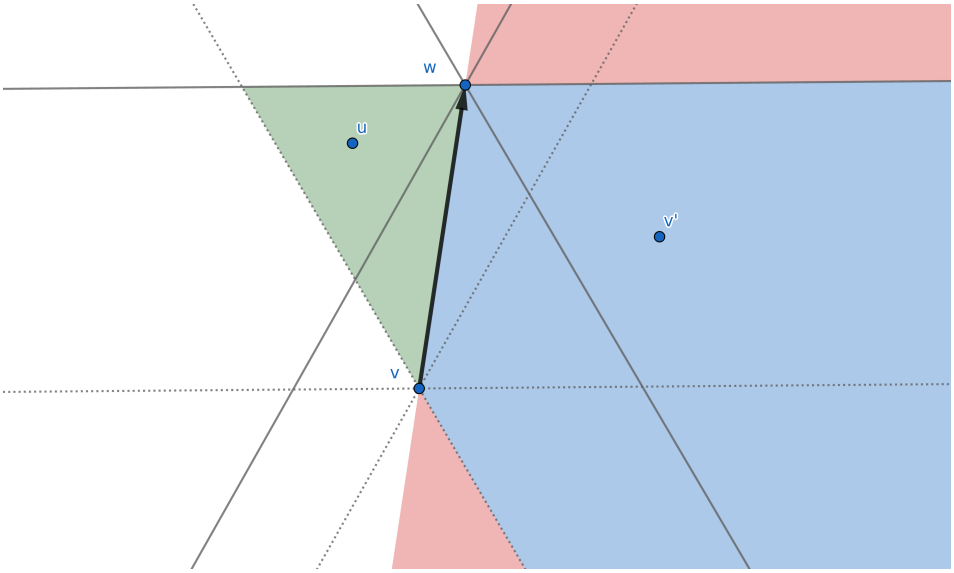


Figure 6.1 – The arc between u and v' must cross (v, w).

Proof. Since u is in an outgoing cone $\mathcal{C}$ of v, we have that v cannot be a point at infinity and that there exists initially a unique neighbor w of v in E and in $\mathcal{C}$. Since the only neighbor of v in F is u, the arcs (v, u) and (v, w) delimit the triangular face Δ_{uvw}. Let $\Delta_{v'vw}$ be the other triangular face incident to (v, w) in the triangulation (V, E').

We will now prove that v' and u are not connected in E' otherwise (v, w) would cross (u, v'). Figure 6.1 helps to visualize this. Without loss of generality, u is on the left side of the arc (v, w) and, because v is not a point at infinity, v' is on the right side (red and blue areas). We notice that:

- v has an outgoing neighbor in the bottom right cone. Since v' is the previous neighbor of v before w, the point v' cannot be in the below red zone.
- If w is not a point at infinity, it has outgoing neighbors in the bottom left and bottom right cones. Since u and v' are respectively the previous and next neighbors of w with respect to v, the points u and v' cannot be above the horizontal line passing through w. It means that u, resp. v', is the green zone, resp. the blue zone.
- If w is a point at infinity, it must be the point at infinity in the direction of cone $\mathcal{C}$ because v (which is not a point at infinity) has w in its outgoing cone $\mathcal{C}$. Therefore, w is linked to the other two points at infinity, one on the left of v, one of the right of v, and both below the horizontal line passing through w. Since u and v' are respectively the previous and next neighbors of w with respect to v, the points u and v' cannot be above that horizontal line either.

Then, we verify the `almostTD` properties:

1. The only modification of the operation $\mathcal{G}$.`replace`(a) in E' consists of the replacement of the arc (v, w) by the arc (u, v') or (v', u) in the quadrilateral $uvv'w$. Thus (V, E') is still a triangulation.

2. The only modification of the operation $\mathcal{G}$.`replace`(a) in E consists of the replacement of the arc (v, w) by the arc (v, u). Since u and w are in the same outgoing cone $\mathcal{C}$ of v , the second property of $\texttt{almostTD}_u$ still holds.

3. Finally, the only new arc in F is (u, v') or (v', u), which both have u as one of their extremities.

□

A property to certify that arcs removals are correct: `propDelete`**.**

Definition 66. *Given two bicolored graphs $\mathcal{G}$ and $\mathcal{G}'$ on the same set V of vertices, and $u \in V$, we say that the property* `propDelete`$(\mathcal{G}, \mathcal{G}', u)$ *holds if, for every v and v' in V with v' in some outgoing cone $\mathcal{C}$ of v, the statement:*

- *the arc (v, v') belongs to $\mathcal{G}$ but not to $\mathcal{G}'$*

is equivalent to the conjunction of the following four statements:

1. *v and v' are both different from u,*
2. *u is in the (outgoing) cone $\mathcal{C}$ of v*
3. *the arc (v, v') belongs to $\mathcal{G}$,*
4. *the arc (v, u) belongs to the first arc set E of $\mathcal{G}'$ but does not belong to $\mathcal{G}$.*

Intuitively, the arc (v, v') is deleted from $\mathcal{G}$ to $\mathcal{G}'$ if and only if the arc (v, u) is added from $\mathcal{G}$ to $\mathcal{G}'$.

We now analyze the algorithm for the set V of points and the point $u \notin V$. Let $\mathcal{G}_{init}$ be the bicolored graph $\mathcal{G}$ after its initialization in Line 1. Let $\mathcal{G}_i$ be the value of $\mathcal{G}$ when evaluating the while condition at Line 11 after $i \geq 0$ executions of the while loop body.

Lemma 67. *The algorithm is well defined and, for every $i \geq 0$ such that the algorithm executes at least i times the while loop, we have that:*

- *$\mathcal{G}_i$ is* $\texttt{almostTD}_u$*,*
- *any arc in $\mathcal{G}_i$ and not in $\mathcal{G}_{init}$ has u as one of its extremities,*
- `propDelete`$(\mathcal{G}_{init}, \mathcal{G}_i, u)$ *is satisfied.*

Moreover the algorithm always terminates, after at most $|V|$ executions of the while loop.

Proof. The only definition problem of the algorithm is Line 12 where the arc from F is claimed to have u as one of its extremities. Therefore, to prove that the algorithm is well defined, it is sufficient to prove that the claimed properties of $\mathcal{G}_i$ (in particular the first one) are satisfied.

We prove them by induction on i. For the case $i = 0$, we distinguish whether $|S|$ is equal to 4 or not.

- If $|S| \neq 4$, then $|S| = 3$ and u lies inside a triangle which is not the external one defined by the points at the infinity. The only difference between $\mathcal{G}_{init}$ and $\mathcal{G}_0$ is that arcs between u and the vertices of the enclosing triangle are added to the second arc set F of $\mathcal{G}$. By Lemma 62, $\mathcal{G}_0$ satisfies all the required properties in this case.

- If $|S| = 4$, the arc (v, v') on which u lies is not an arc of the external triangle defined by the points at the infinity. Let w and w' be the two other points of S. The initialization consists of replacing (v, v') by (v, u) and (u, v') in E, and adding arcs between u and w, w' into F. Again using Lemma 62, $\mathcal{G}_0$ thus satisfies all the required properties in this case as well.

Assume now that the algorithm executes at least $i + 1$ times the while loop, for $i \geq 0$, and that $\mathcal{G}_i$ satisfies the required properties by induction hypothesis. In particular, $\mathcal{G}_i$ is $\texttt{almostTD}_u$, and by Lemmas 64 and 65, so is $\mathcal{G}_{i+1}$. Similarly, the second item is inductively satisfied by $\mathcal{G}_i$, and since the only arc added to $\mathcal{G}_i$ to obtain $\mathcal{G}_{i+1}$ has u as one of its extremities, $\mathcal{G}_{i+1}$ also satisfies the property of the second item. The induction hypothesis, the test Line 13, and the definitions of $\texttt{move}(\cdot)$ and $\texttt{replace}(\cdot)$ imply that $\mathcal{G}_{i+1}$ satisfies the property of the third item as well.

Finally, it remains to prove that the algorithm terminates after at most $|V|$ iterations of the while loop. We first note that each iteration can be characterized by an arc between u and some $v \in V$. This arc is added to E. By the property $\texttt{propDelete}(\mathcal{G}_{init}, \mathcal{G}_i, u)$ for $i \geq 0$, those arcs are never removed. Therefore each iteration of the while loop is characterized by an arc between u and a point $v \in V$ specific to this iteration, which proves the desired property and concludes the proof of the lemma. □

Correctness. Thanks to Lemma 67, we can now define $\mathcal{G}_{end}$ as the bicolored graph output of the algorithm. Let us summarize the properties satisfied by $\mathcal{G}_{end}$ (the proof will follow).

Definition 68 ($\mathcal{P}$ property)**.** *A bicolored graph $\mathcal{G} = (V \cup \{u\}, E, F)$ satisfies the property $\mathcal{P}(\mathcal{G}_{init}, \mathcal{G}, u)$ when:*

1. *$\mathcal{G}$ is $\texttt{almostTD}_u$,*

2. *any arc in $\mathcal{G}$ but not in $\mathcal{G}_{init}$ has u as one of its extremities,*

3. *$\texttt{propDelete}(\mathcal{G}_{init}, \mathcal{G}, u)$ is satisfied,*

4. *any two neighbors of u in $\mathcal{G}$ that are consecutive in the trigonometric cyclic order form an angle smaller than π,*

5. *the second arc set F of $\mathcal{G}$ is empty.*

Lemma 69. *The bicolored graph $\mathcal{G}_{end}$ satisfies $\mathcal{P}(\mathcal{G}_{init}, \mathcal{G}, u)$.*

Proof. The first three properties follow directly from Lemma 67 because $\mathcal{G}_{end}$ corresponds to the last $\mathcal{G}_i$. For the fourth property, it is sufficient to prove it for $\mathcal{G}_0$ because no arc incident to u is ever removed from $\mathcal{G}$ (property $\texttt{propDelete}(\mathcal{G}_{init}, \mathcal{G}_i, u)$ for all $i \geq 0$). When $|S| = 3$, u is linked in $\mathcal{G}_0$ to the vertices of the internal triangular face it initially lies in. When $|S| = 4$, u is linked in $\mathcal{G}_0$ to the four vertices of the quadrilateral formed by the two internal triangles it initially lies in. In both cases, the property holds. Finally, the fact that F is empty in $\mathcal{G}_{end}$ directly follows from the condition of the while loop line 11 and the fact that the algorithm terminates. □

The fact that $\mathcal{G}_{end}$ is equal to $\tilde{\Theta}_6^+(V \cup \{u\})$ as desired will follow from the facts that $\tilde{\Theta}_6^+(V \cup \{u\})$ also satisfies $\mathcal{P}(\mathcal{G}_{init}, \cdot, u)$ and that at most one bicolored graph can satisfy it.

Lemma 70. *$\mathcal{P}\left(\mathcal{G}_{init}, \tilde{\Theta}_6^+(V \cup \{u\}), u\right)$ holds.*

Proof. The first property follows from Lemma 62.

Now consider two points v and v', with v' in an outgoing cone $\mathcal{C}$ of v. The arc (v, v') belongs to $\tilde{\Theta}_6^+(V \cup \{u\})$ if and only if v' is the nearest point to v in $\mathcal{C}$ in a certain way (see Definition 47). If both v and v' are different from u, then that property also holds without u. Thus the arc is also in $\tilde{\Theta}_6^+(V)$ and in $\mathcal{G}_{init}$ in this case. This proves the second property.

Still considering these points v and v', the arc (v, v') is in $\mathcal{G}_{init}$ (and thus in $\tilde{\Theta}_6^+(V)$) but not in $\tilde{\Theta}_6^+(V \cup \{u\})$ if and only if v', resp. u, is the nearest point to v in $\mathcal{C}$ in $\tilde{\Theta}_6^+(V)$, resp. $\tilde{\Theta}_6^+(V \cup \{u\})$. This proves the third property.

Point u has points in each of its outgoing cones (at least one of the points at infinity) and thus has a neighbor in each of its outgoing cones. By the definition of these cones, the fourth property follows.

Finally, the fifth and last property follows from the definition of $\tilde{\Theta}_6^+(V \cup \{u\})$. □

The fact that only one bicolored graph $\mathcal{G}$ satisfies $\mathcal{P}(\mathcal{G}_{init}, \mathcal{G}, u)$ is proved in two steps, the first one focusing on u's neighbors.

Lemma 71. *Given a point $u \notin V$, and a bicolored graph $\mathcal{G}$ on $V \cup \{u\}$, two bicolored graphs $\mathcal{G}_A$ and $\mathcal{G}_B$ also defined on $V \cup \{u\}$ satisfying respectively $\mathcal{P}(\mathcal{G}, \mathcal{G}_A, u)$ and $\mathcal{P}(\mathcal{G}, \mathcal{G}_B, u)$ have the same arcs incident to u.*

Proof. Without loss of generality, let v be a neighbor of u in $\mathcal{G}_A$ but not in $\mathcal{G}_B$ (note that we don't need to specify which kind of neighbor because the second arc sets of $\mathcal{G}_A$ and $\mathcal{G}_B$ are empty). We first prove that u does not have a neighbor in $\mathcal{G}_B$ on the same ray from u than v is. Indeed, assume that v' would be such a neighbor. If v' is nearer to u than v, then v' would be on the arc between u and v in $\mathcal{G}_A$, contradicting the fact that $\mathcal{G}_A$ is a triangulation. Similarly, if v is nearer to u than v', then v would be on the arc between u and v' in $\mathcal{G}_B$, contradicting the fact that $\mathcal{G}_B$ is a triangulation.

Since such a v' does not exist, and since u has at least three neighbors in $\mathcal{G}_B$ because of the fourth property of $\mathcal{P}(\mathcal{G}, \mathcal{G}_B, u)$, there exist two consecutive neighbors w and w' of u in $\mathcal{G}_B$ such that the ray from u through v lies strictly inside the $\widehat{wuw'}$ angle (on the non-reflex side). Without loss of generality, assume that w' lies in an outgoing cone $\mathcal{C}$ of w.

In the triangulation $\mathcal{G}_B$, the points u, w, and w' form an empty triangle, thus with v on the outside. In $\mathcal{G}_A$, the arc (w, w') does not exist, otherwise it would cross the arc (u, v). However, the arc (w, w') is in $\mathcal{G}$, since it is not incident to u and because of the second property of $\mathcal{P}(\mathcal{G}, \mathcal{G}_B, u)$. Since the arc (w, w') is in $\mathcal{G}$ but not in $\mathcal{G}_A$, and by $\texttt{propDelete}(\mathcal{G}, \mathcal{G}_A, u)$, we have that u also lies in the outgoing cone $\mathcal{C}$ of w.

To summarize, u and w' are both in the outgoing cone $\mathcal{C}$ of w , and are both neighbors of w in $\mathcal{G}_B$. This contradicts the fact that B is $\texttt{almostTD}_u$, and concludes the proof by contradiction of the lemma. □

Lemma 72. *Given a point $u \notin V$, and a bicolored graph $\mathcal{G}$ on $V \cup \{u\}$, two bicolored graphs $\mathcal{G}_A$ and $\mathcal{G}_B$ also defined on $V \cup \{u\}$ satisfying respectively $\mathcal{P}(\mathcal{G}, \mathcal{G}_A, u)$ and $\mathcal{P}(\mathcal{G}, \mathcal{G}_B, u)$ are necessarily equal.*

Proof. Thanks to Lemma 71, we already know that u has the same neighbors in $\mathcal{G}_A$ and in $\mathcal{G}_B$. From the second property of $\mathcal{P}(\mathcal{G}, \mathcal{G}_A, u)$ and $\mathcal{P}(\mathcal{G}, \mathcal{G}_B, u)$, there are no arcs in $\mathcal{G}_A$ or in $\mathcal{G}_B$ which are not incident to u and not in $\mathcal{G}$. So the only differences that may exist between $\mathcal{G}_A$ and $\mathcal{G}_B$ concern the arcs that are in $\mathcal{G}$ but not in $\mathcal{G}_A$ or $\mathcal{G}_B$. However, by the properties $\texttt{propDelete}(\mathcal{G}, \mathcal{G}_A, u)$ and $\texttt{propDelete}(\mathcal{G}, \mathcal{G}_B, u)$, those arcs that were removed from $\mathcal{G}$ are exactly determined by the arcs added to $\mathcal{G}$. The latter being the same in $\mathcal{G}_A$ and $\mathcal{G}_B$, the former are also the same in $\mathcal{G}_A$ and $\mathcal{G}_B$. Therefore $\mathcal{G}_A$ and $\mathcal{G}_B$ are equal. □

The correctness of our algorithm now directly follows from Lemmas 69, 70 and 72.

Theorem 73. *Given a point $u \notin V$, Algorithm 14 outputs $\tilde{\Theta}_6^+(V \cup \{u\})$.*

6.2.1 Complexity.

Theorem 74. *The time complexity of Algorithm 14 is in $O(\Delta_u)$, where Δ_u is the final degree of the added vertex u, plus the time complexity to query (and update) the P_L data structure.*

Proof. With a combinatorial map (D, σ, α) (Section A.4), the initialization and each iteration of the while loop can be executed in constant time (not taking into account the `PointLocation` request). One would need to attach a bit consistent with α to the darts to specify whether the edge belongs to the first or second edge set. All operations performed by the algorithm are local and only concern a constant number of faces. Moreover, the points coordinates can be used to retrieve the necessary information about cones.

Furthermore, an arc incident to u is added in the first arc set E at each iteration of the while loop and such arcs are never removed. Therefore, there are at most Δ_u iterations of the while loop before the algorithm terminates, and each iteration takes constant time (as far as P_L is not concerned). □

6.3 Deletion algorithm

We are now interested in the dynamic maintenance of a half-Theta-6 when deleting a vertex, i.e. keeping the structure of half-Theta-6. If we delete a vertex $u \in V$, we

need to redirect the incoming arcs (v, u) for all v. Since v loses its outgoing neighbor in the corresponding cone, we have to find the new outgoing neighbor. That is, we replace (v, u) by an arc (v, w), with $w \in V$ in the same outgoing cone of v that u.

6.3.1 Algorithm

We propose an algorithm allowing an efficient deletion, whose idea is the following: without loss of generality, we first study the sectors with increasing z. We process $E_z(u)$, the set of incoming neighbors of u in the direction of z, in counter-clockwise order. These are the points for which we need to find a new outgoing neighbor.

For each of these points v, the candidates are neighbors of u, in one of the three following sectors: $\mathcal{C}_z^+(u)$, $\mathcal{C}_y^-(u)$ and $\mathcal{C}_x^-(u)$ as we will verify it with the Lemma 75. For the first one, it is easy: there is only one (outgoing) neighbor of u in $\mathcal{C}_z^+(u)$. For the other two, we go through them in the order of increasing z. In each of these three cones, we find the nearest point from v also in $\mathcal{C}_z^+(v)$. Among these three points, we take again the nearest one: it is the new outgoing neighbor of v according to the direction z. In particular, if among these three points, several have the same z coordinate, we take the one whose coordinate y is minimal.

Then we move to the next point of $E_z(u)$. But instead of restarting the process of $E_x(u)$ and $E_y(u)$ from the beginning, we resume the process where it was at the end of the previous step. The pseudo-code is detailed with the algorithm 15

6.3.2 Correctness

Lemma 75. *Let $u, v \in V$ such that $(v, u) \in \Theta_6^+(V)$ with u in the ingoing cone $\mathcal{C}$ of v. Let $w \in V$ such that $(v, w) \in \Theta_6^+(V \backslash \{u\})$ with w in the same cone $\mathcal{C}$. Then w is either in $\mathcal{C}_z^+(u)$, $\mathcal{C}_y^-(u)$ or $\mathcal{C}_x^-(u)$. Furthermore, w is a neighbor of u in $\Theta_6^+(V)$.*

Proof. Without loss of generality, we consider that v is in the incoming cone $\mathcal{C}_z^-(u)$. Therefore, $v \in E_z(u)$. If $w \notin \mathcal{C}_z^+(u) \cup \mathcal{C}_y^-(u) \cup \mathcal{C}_x^-(u)$, then $w \in \mathcal{C}_z^-(u) \cup \mathcal{C}_y^+(u) \cup \mathcal{C}_x^+(u)$ since these six cones partition the plane.

- By definition of the cones $\mathcal{C}_z^-(u)$ and $\mathcal{C}_y^+(u)$, if w is in one of those, then we have $w_z < u_z$.
- If $w \in \mathcal{C}_x^+(u)$, then $w_z \leq u_z$ and $w_x > u_x$.

But v has only one outgoing neighbor in $\mathcal{C}$ so $(v, w) \notin \Theta_6^+(V)$. Thus, w should be its outgoing neighbor instead of u. So w must be $\mathcal{C}_z^+(u) \cup \mathcal{C}_y^-(u) \cup \mathcal{C}_x^-(u)$.

Then, we check for each of the three cones, that if w is in one of them, then it is connected to u.

- If $w \in \mathcal{C}_z^+(u)$, then the nearest point to u in $\mathcal{C}_z^+(u)$ is the nearest from v in the same cone.
- If $w \in \mathcal{C}_x^-(u)$, then we assume by the absurd that $(w, u) \notin \Theta_6^+(V)$. Visually, this can be seen with the Figure 6.2. We show that the neighbor of w in $\mathcal{C}_x^+(w)$ can be neither in the red, nor in the blue zone.

Input: $\Theta_6^+(V)$ for a given set of points V and $u \in V$
Output: $\Theta_6^+(V \backslash \{u\})$
// We handle the z direction, same for the others
$v \leftarrow \texttt{first}_z(u)$ // process in the counterclockwise order
$w^x \leftarrow \texttt{last}_x(u)$ // process in the clockwise order
$w^y \leftarrow \texttt{last}_y(u)$ // process in the clockwise order
while $v \in \mathcal{C}_z^-(u)$ **do** // going through $E_z(u)$ vertices to redirect them
  // While there are still neighbors in $E_x(u)$ to consider and they are in the cone of the current vertex
  while $\texttt{previous}(u, w^x) \in \mathcal{C}_x^-(u)$ *and* $\texttt{previous}(u, w^x)_y \leq v_y$ **do**
    $w^x \leftarrow \texttt{previous}(u, w^x)$
  // While there are still neighbors in $E_y(u)$ to consider and they are in the cone of the current vertex
  while $\texttt{previous}(u, w^y) \in \mathcal{C}_y^-(u)$ *and* $(w^y)_x \geq v_x$ **do**
    $w^y \leftarrow \texttt{previous}(u, w^y)$
  Let w' be the outgoing neighbor of u in its z direction
  if $w^x \in \mathcal{C}_x^-(u) \cap \mathcal{C}_z^+(v)$ *and* $w_z^x < w_z'$ **then**
    $w' \leftarrow w^x$
  if $w^y \in \mathcal{C}_y^-(u) \cap \mathcal{C}_z^+(v)$ *and* $w_z^y \leq w_z'$ **then**
    $w' \leftarrow w^y$
  $\mathcal{G}.\texttt{insert}(v, w')$
  $v \leftarrow \texttt{next}(u, v)$
$\mathcal{G}.\texttt{remove}(u)$

Algorithm 15: Deletion of a vertex in a Θ_6^+.

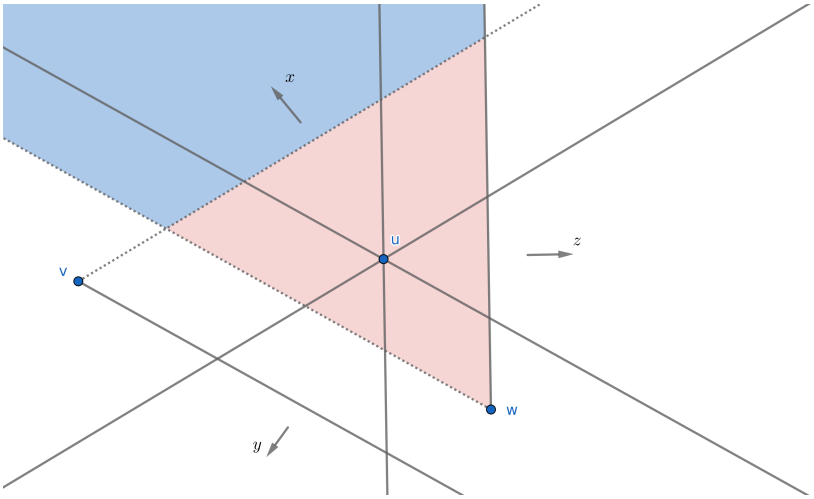


Figure 6.2 – If w is the neighbor replacing u when it is removed, then w is connected to u.

Formally, w has an outgoing neighbor other than u in its cone $\mathcal{C}_x^+(w)$, we note it w'. We show that $w' \in \mathcal{C}_z^+(v)$:

1. $w'_x \leq u_x < v_x$ since w is not connected to u and $v \in \mathcal{C}_z^-(u)$.
2. $w'_y < w_y \leq v_y$ since $w' \in \mathcal{C}_x^+(w)$ and $w \in \mathcal{C}_z^+(v)$.
3. $w'_z > v_z$ since v and w' have same rank.

Thus, w' cannot be in the blue zone. Furthermore, by definition of w', we have: $w'_z \leq w_z$ and $w'_y < w_y$. Therefore, w' is nearest than w in $\mathcal{C}_z^+(v)$, which contradicts the definition of w.

- If $w \in \mathcal{C}_y^-(u)$, then we assume by the absurd that $(w, u) \notin \Theta_6^+(V)$. Thus, w has an outgoing neighbor other than u in its cone $\mathcal{C}_y^+(w)$, we note it w'. We show that $w' \in \mathcal{C}_z^+(v)$:

  1. $w'_x \leq w_x < v_x$ since $w' \in \mathcal{C}_y^+(w)$ and $w \in \mathcal{C}_z^+(v)$.
  2. $w'_y \leq u_y \leq v_y$ since w is not connected to u and $v \in \mathcal{C}_z^-(u)$.
  3. $w'_z > v_z$ since v and w' have same rank.

  Furthermore, by definition of w', we have: $w'_z < w_z$ and $w'_y < w_y$. Therefore, w' is nearest than w in $\mathcal{C}_z^+(v)$, which contradicts the definition of w.

□

Theorem 76. *Algorithm 15 taking as input $\Theta_6^+(V)$ and $u \in V$, outputs $\Theta_6^+(V \backslash \{u\})$.*

Proof. Note that the sets $E_x(u)$ and $E_y(u)$ may be empty. Thus, `last` returns a point in another cone and the tests Lines 10 and 12 guarantee that such irrelevant points will not be considered. Furthermore, if $E_x(u)$ is not empty, then the test Line 5 insures that w^x will never leave $\mathcal{C}_x^-(u)$. Same for E_y.

We show the correction by induction on the ingoing neighbors of u. For v the first of them, our algorithm correctly computes the nearest points from v in $\mathcal{C}_z^+(u)$, $\mathcal{C}_y^-(u)$ and $\mathcal{C}_x^-(u)$ restricted to $\mathcal{C}_z^+(v)$. Using the Lemma 75, we have therefore redirected v correctly by taking the nearest point among those three points.

For the iteration, we assume that we have just processed v. We have to prove that it is not necessary to reset w^x and w^y to process the point $v' = \texttt{next}(u, v)$. We are processing the points v in the order of strictly increasing y since we use `next`, and $E_x(u)$ in the order of increasing y since we use `previous`. So at the beginning of the iteration of the while loop Line 4 for v', only w^x and the points of $E_x(u)$ processed afterwards using `previous` are potentially to be connected to v'. Same argument for E_y. □

6.3.3 Complexity

As for the insertion, we use combinatorial maps (Section A.4). Keep in mind that a local modification of an arc is in constant time.

Theorem 77. *Let $u \in V$. Let Δ_u be the degree of u in $\Theta_6^+(V)$. The Algorithm 15 given $\Theta_6^+(V)$ as input is in $O(\Delta_u \log \Delta_u)$.*

Proof. We analyze the cost of each iteration of the main while loop (Line 4), that is to say for each point $v \in E_z(u)$.

- The operations `previous` and `next` are in constant time.
- If we add the arc (v, w), then we have to add two half arcs, each corresponding to one of the two points, so that it stores the position of the other among its neighbors in clockwise order. On the side of v, since it is connected to only one point in $\mathcal{C}_z^+(v)$, the insertion of w consists in replacing u and thus is in $O(1)$.
- On the other hand, from the point of view of w, we do not know where to place v. Even if the points v are considered in counterclockwise order with respect to u, this is not necessarily true with respect to w.

  To be efficient, we put all these insertions on hold during the while loop. Once we have finished processing $E_z(u)$, for each neighbor of u, we sort the pending insertions and perform them. Since at most Δ_u insertions can be put on hold, the sorting step implies a $O(\Delta_u \log \Delta_u)$ time complexity.
- The cumulation of the internal while loops only covers $E_x(u)$ and $E_y(u)$ once at worst. Thus, the cumulative complexity is linear.

The loop performs at most Δ_u iterations so the constant time operations in the loop give a linear accumulation and are therefore negligible.

Moreover, the operations `first` and `last` are also constant. Finally, the deletion of u is in $O(\Delta_u)$ since the deletions of each half-arch are local. Thus, we obtain the announced complexity. □

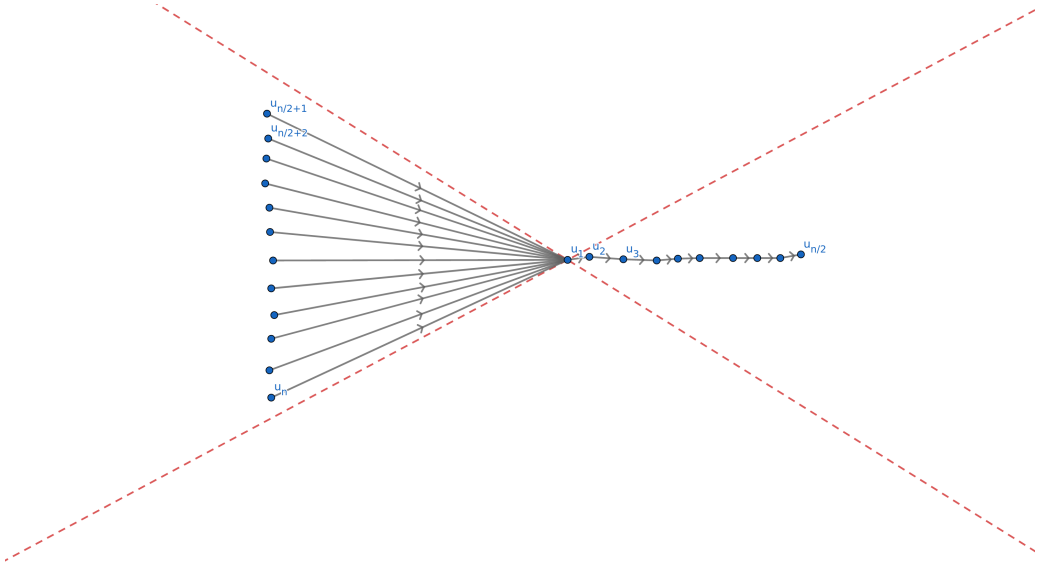


Figure 6.3 – Worst Case for add and deletion

Worst-case. One might think that successively deleting each point from V might give an "interesting" amortized complexity. We show that it is at least linear in the worst-case.

Lemma 78. *Let $n = |V|$. There exists a sequence of n point deletions from V such that $\Theta(n^2)$ redirections are required.*

Proof. Such a sequence is represented in Figure 6.3, with n even. In this example, we assume that the points u_i are deleted in the increasing indices order. The points $u_{n/2+1}, \ldots, u_n$ all have u_1 as their outgoing neighbor. When u_1 is deleted, we have to redirect them all to u_2, etc... Thus, each deletion requires $\Theta(n)$ redirection, and there is a $\Theta(n)$ points deleted.

□

Remark. *This lemma also stands for a sequence of insertions.*

6.4 Application to tricriteria shortest path computation

In Section 3.5, an improvement of Algorithm 9 consists in using Theta graphs. We restrict ourselves to the case $d = 3$. The sectors we are interested in correspond to the outgoing cones of a Θ_6^+, bounded by a factor $(1+\varepsilon)$ on the increasing coordinate. In order to prune a path P, we want to know if its cost $c(P)$ is $(1+\varepsilon)$-covered in its three sectors. Given a Θ_6^+ on path costs, we can find out whether $c(P)$ is $(1+\varepsilon)$-covered in the cone $\mathcal{C}_z^+(c(P))$ by looking at its outgoing neighbor v in that cone. v $(1+\varepsilon)$-covers $c(P)$ if and only if $v_z \leq (1+\varepsilon)c(P)_z$. The same is true for the other two cones.

We therefore propose Theta Sample Sector (Algorithm 16). This algorithm consists in processing a set of paths and keeping some of them. If the current

path P is not $(1+\varepsilon)$-covered in each sector by the paths previously seen and kept, P is kept. Otherwise, it is not kept. In order to know if a path is $(1+\varepsilon)$-covered in each sector, its cost is inserted in an initially empty Θ-graph. If the path is not kept, its cost is deleted from the Θ-graph so that it is not used to cover the paths seen a posteriori. We use for that the Algorithms 14 and 15, along with a Point Location data structure and its maintenance. A last step at Line 11 removes paths dominated by inferior rank paths in $\mathcal{S}$ from remaining paths.

Input: $\mathcal{R}, \mathcal{S}$ set of paths, $\varepsilon > 0$
Output: $\mathcal{R}'$ set of paths
1 $\mathcal{G}_\Theta \leftarrow \Theta_6^+(\{\infty_x, \infty_y, \infty_z\})$
2 $\mathcal{R}' \leftarrow \emptyset$
3 **foreach** $P \in \mathcal{R}$ **do**
4 `insert`$(\mathcal{G}_\Theta, c(P))$
5 **foreach** $i \in \{x, y, z\}$ **do**
6 Let Q s.t. $c(Q) \in C_i^+(c(P))$ and $(c(P), c(Q)) \in \mathcal{G}_\Theta$
`// Q is null if` $c(Q) = \infty_i$
7 **if** $Q = null$ *or* $Q_i \geq (1+\varepsilon)P_i$ **then**
8 $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{P\}$; `// then it is kept`
9 **break** ; `// no need to search for other sectors anymore`
10 `remove`$(\mathcal{G}_\Theta, c(P))$ `// it not kept since it is covered`
11 `RemoveDominated`$(\mathcal{R}', \mathcal{S})$ `// removes from` $\mathcal{R}'$ `path dominated by` $\mathcal{S}$

Algorithm 16: Theta Sample Sector: Sample Sector using Theta graphs.

Definition 79 (TSector Algorithm)**.** *Algorithm* TSector *is the* Meta Rank *algorithm (Alg. 6) using* Theta Sample Sector *as the* `Sample` *function.*

Theorem 80. *Let S_ε be the size of the output of* TSector*. The output-sensitive time complexity of* TSector *is in* $O\left((\Delta S_\varepsilon)^2 \log(\Delta S_\varepsilon) \log\log(\Delta S_\varepsilon)\right)$.

Proof. Given a set $\mathcal{R}$ of paths of size $R = |\mathcal{R}|$, Theta Sample Sector processes each path $P \in \mathcal{R}$ with the following operations:

- It inserts P in the current Theta-graph. The point location query is in time $O(\log R(\log\log R)^2)$ and the modifications are in $O(R \log R \log\log R)$.
- It may delete P from the current Theta-graph. The complexity is in $O(R \log R)$.

Thus, the main loop is in $O(R^2 \log R \log\log R)$.

The removal of dominated paths in TSector is in $O((\Delta S_\varepsilon)^2)$ since each pair of processed paths is compared at most once. Therefore, repeating Theta Sample Sector as a `Sample` function over each vertex and rank gives $C_{\texttt{Sample}}(n, S_\varepsilon, \Delta, \Lambda) = O((\Delta S_\varepsilon)^2 \log(\Delta S_\varepsilon) \log\log(\Delta S_\varepsilon))$. Applying Theorem 15 allows us to conclude. □

The removal of dominated paths at Line 11 could be replaced by adding to the Θ_6^+ the projections of the paths $\mathcal{S}$ in each cones (each projection consists in increasing

only one coordinate). The insertions, followed by a potential deletions, worsen the complexity of Theta Sample Sector if the values of $\Delta_{c(P)}$, the degree of $c(P)$ in the current Θ_6^+, are in $\Omega(n)$. However, we know that $\Delta_{c(P)}$ is on average constant since the Θ_6^+ are planar graphs. So we can expect a gain in practice.

Chapter 7

Proximity queries and dominating set.

7.1 k-Nearest Neighbors in a sector

The method proposed in Section 6.4 does not give any interesting bound over the number of kept path. In order to do so, we enhance the Θ_6^+ structure in this section in order to use it the following ones to bound the number of kept path by a log factor multiplied by the minimum possible with respect to the ε-weak framing property.

For each $u \in V$, $\Theta_6^+(V)$ gives us the nearest point to u in each of its outgoing cones in triangular distance. In this part, we want to generalize to several points. More precisely, we want to compute the at most k nearest points from u, at triangular distance at most $(1+\varepsilon)$ from u. We note these sets of point $S^k_{x,\varepsilon}(u) \subseteq \mathcal{C}^+_x(u)$, $S^k_{y,\varepsilon}(u) \subseteq \mathcal{C}^+_y(u)$ and $S^k_{z,\varepsilon}(u) \subseteq \mathcal{C}^+_z(u)$. The related set V is implicit for legibility reason.

In this section, we propose an algorithm for the computation of $S^k_{z,\varepsilon}(u)$, the others being similar. We recall that for a point $u \in V$, if $v, w \in V$, with $v \neq w$, v and w being in the same cone of u and $d_\Delta(u,v) = d_\Delta(u,w)$ (here $v_z = w_z$), then, among v and w, the nearest point from u is the first in clockwise order (here v is nearer if and only if $v_y < w_y$).

7.1.1 Algorithm

We describe our algorithm computing $S^k_{z,\varepsilon}(u)$ for a vertex u. The algorithm is the same for $S^k_{x,\varepsilon}(u)$ and $S^k_{y,\varepsilon}(u)$. It processes iteratively vertices from $\mathcal{C}^+_z(u)$. Each vertex is given one of the following three colors:

white : the vertex has not been seen yet,

gray : it has been seen but not yet processed,

black : it has been seen and processed.

Each vertex is first white, then can become gray and finally black. At each time, any black vertex v is associated to at most three gray vertices: $\texttt{Rep}_\texttt{x}(v)$, $\texttt{Rep}_\texttt{y}(v)$

and $\texttt{Rep}_\texttt{z}(v)$, each contained in one of the following sets: $E_x(v)$, $E_y(v)$ and $\{s_z(v)\}$. That is to say that these are neighbors in each cone with coordinate z greater than v_z.

The algorithm starts by coloring u in gray. Then, it iterates the following steps:

1. Let v be the gray vertex nearest to u (with minimal z coordinate and in case of non-unicity, with minimal y coordinate among those with minimal z coordinate). It colors v in black.

2. For any black vertex w such that $\texttt{Rep}_\texttt{x}(w) = v$, $\texttt{Rep}_\texttt{x}(w)$ takes as value the neighbor of w following v in counterclockwise order (i.e. $\texttt{next}(w, v)$, corresponding to the order of increasing z in $E_x(w)$). If v is the last vertex of $E_x(w)$, i.e. $\texttt{last}_x(w) = v$, then $\texttt{Rep}_\texttt{x}(w)$ is set to *null*. Otherwise, $\texttt{Rep}_\texttt{x}(w)$ is colored in gray.

3. The same for $\texttt{Rep}_\texttt{y}$ in clockwise order, with $\texttt{previous}(w, v)$. $\texttt{Rep}_\texttt{y}(w)$ is set to *null* if $\texttt{first}_y(w) = v$.

4. For each black vertex w such that $\texttt{Rep}_\texttt{z}(w) = v$, $\texttt{Rep}_\texttt{z}(w)$ is set to *null*.

5. Then we define the represented points of v by taking:

   - $\texttt{Rep}_\texttt{x}(v)$ the nearest vertex from u in $E_x(v) \cap \mathcal{C}_z^+(u)$,
   - $\texttt{Rep}_\texttt{y}(v)$ the nearest vertex from u in $E_y(v) \cap \mathcal{C}_z^+(u)$,
   - $\texttt{Rep}_\texttt{z}(v)$ the outgoing neighbor of v in $\mathcal{C}_z^+(v)$.

The algorithm stops when there are k black vertices or when the vertex v being processed is at a triangular distance superior to $(1 + \varepsilon)$ from u, i.e. such that $v_z > (1 + \varepsilon)u_z$ since here, considered vertices are in $\mathcal{C}_z^+(u)$. The corresponding pseudo code is the Algorithm 17.

The definition of:

- v at Line 6 can be written as $\arg\min\{v'_y | v' \in \arg\min\{v_z | v \text{ gray vertex }\}\}$. Indeed, the nearest is the minimum along z coordinate. If there are several of them, the nearest is, among those, the one whose y coordinate is minimal. This case is handled by a second $\arg\min$ with y coordinate.

- $\texttt{Rep}_\texttt{x}(v)$ at Line 23 can be written as $\arg\min\{w_z | w \in E_x(v) \text{ and } w_y \leq u_y\}$ since the nearest is the minimal along z coordinate. Such a vertex is unique since the order $\prec_x$ on $E_x(v)$ is total. The test for belonging to $\mathcal{C}_z^+(u)$ is reduced to a comparison over y coordinates.

- $\texttt{Rep}_\texttt{y}(v)$ at Line 24 can be written as $\arg\min\{w'_y | w' \in \arg\min\{w_z | w \in E_y(v) \text{ and } w_x \leq u_x\}\}$, for the same reasons as for the definition of v.

Remark. *One can simply wish to obtain the k nearest points, resp. all points at a multiplicative distance $(1 + \varepsilon)$. For this, set $\varepsilon = +\infty$, resp. $k = n$.*

On Figure 7.1, we compute the k nearest points from u in the blue triangle. If v is the nearest point from u, then we observe that the other points of interest are necessarily in its right half-plane (otherwise v would not be the nearest). We

Input: $\Theta_6^+(V)$ for a given set of points V, $u \in V$, $k \in \mathbb{N}$, $\varepsilon > O$
Output: $S_{z,\varepsilon}^k(u)$

$\mathcal{S} \leftarrow \emptyset$
Let $v \in V$ be the outgoing neighbor of u in $\mathcal{C}_z^+(u)$
if $v_z \leq (1+\varepsilon)u_z$ **then**
    $\texttt{Color}(v) \leftarrow gray$
while *at least one vertex is gray and* $|\mathcal{S}| < k$ **do**
    $v \leftarrow$ nearest gray point from u
    $\texttt{Color}(v) \leftarrow black$
    foreach $w^x \in \texttt{Rep}_\texttt{x}^{-1}(v)$ **do** `// i.e.` w^x `such that` $\texttt{Rep}_\texttt{x}(w^x) = v$
        if $\texttt{next}(w^x, v) \in \mathcal{C}_x^-(w^x)$ **then**
            $\texttt{Rep}_\texttt{x}(w^x) \leftarrow \texttt{next}(w^x, v)$
            $\texttt{Color}(\texttt{Rep}_\texttt{x}(w^x)) \leftarrow gray$
        else
            $\texttt{Rep}_\texttt{x}(w^x) \leftarrow null$
    foreach $w^y \in \texttt{Rep}_\texttt{y}^{-1}(v)$ **do** `// i.e.` w^y `such that` $\texttt{Rep}_\texttt{y}(w^y) = v$
        if $\texttt{previous}(w^y, v) \in \mathcal{C}_y^-(w^y)$ **then**
            $\texttt{Rep}_\texttt{y}(w^y) \leftarrow \texttt{previous}(w^y, v)$
            $\texttt{Color}(\texttt{Rep}_\texttt{x}(w^y)) \leftarrow gray$
        else
            $\texttt{Rep}_\texttt{y}(w^y) \leftarrow null$
    foreach $w^z \in \texttt{Rep}_\texttt{z}^{-1}(v)$ **do** `// optional`
        $\texttt{Rep}_\texttt{z}(w^z) \leftarrow null$
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$
    $\texttt{Rep}_\texttt{x}(v) \leftarrow$ nearest point from u in $E_x(v) \cap C_z^+(u)$
    $\texttt{Rep}_\texttt{y}(v) \leftarrow$ nearest point from u in $E_y(v) \cap C_z^+(u)$
    $\texttt{Rep}_\texttt{z}(v) \leftarrow$ the outgoing neighbor of v in $\mathcal{C}_z^+(v)$
    $\texttt{Color}(\texttt{Rep}_\texttt{x}(v)) \leftarrow gray$
    $\texttt{Color}(\texttt{Rep}_\texttt{y}(v)) \leftarrow gray$
    $\texttt{Color}(\texttt{Rep}_\texttt{z}(v)) \leftarrow gray$
return $\mathcal{S}$

Algorithm 17: k-Nearest Neighbors in a sector

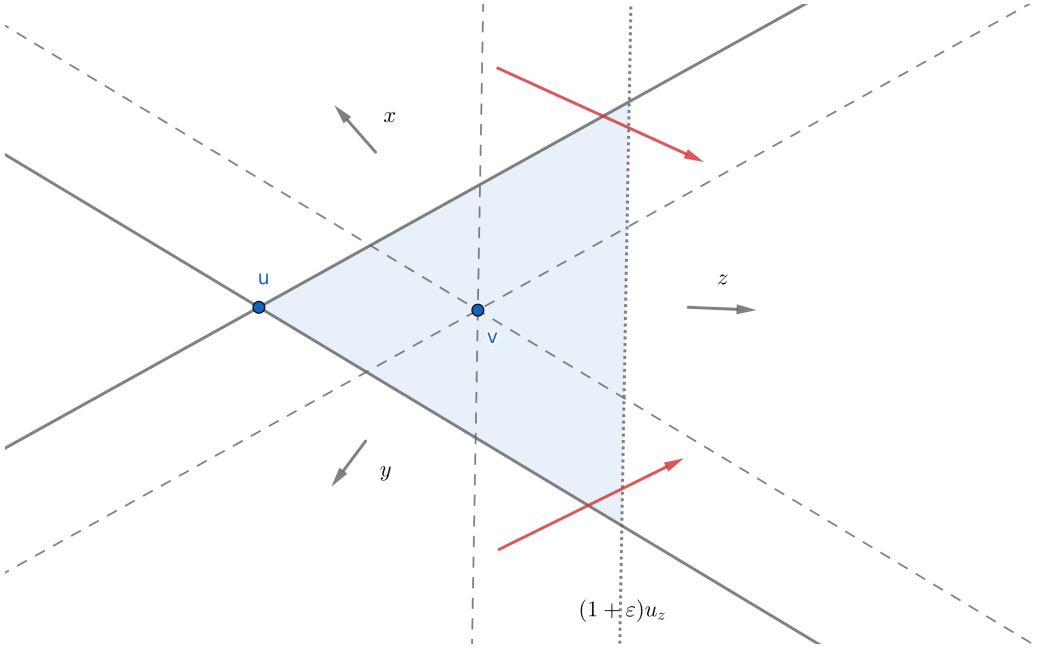


Figure 7.1 – Construction of $S^k_{z,\varepsilon}(u)$.

must therefore look at the corresponding three cones: $\mathcal{C}^-_x(v)$, $\mathcal{C}^-_y(v)$ and $\mathcal{C}^+_z(v)$. This remains true for the points that will follow v since the gray vertices are processed by increasing z coordinates.

Among the three cones, the outgoing cone is easy to process (only one neighbor). For the others, the algorithm processes them according to the order of the red arrows. Note that the red arrows are not included in the blue triangle: one must therefore restrict the process of these cones to $\mathcal{C}^+_z(u)$ and to the points at distance $(1+\varepsilon)$ from u according to z. After v is processed, it becomes black and the first vertex of each of its cones becomes gray. When one of those first vertices is processed, it becomes also black and the algorithm colors the second in gray, etc...

7.1.2 Correctness

Theorem 81. *Given $u \in V$, $\Theta^+_6(V)$, $k \in \mathbb{N}$ and $\varepsilon > 0$, Algorithm 17 outputs $S^k_{z,\varepsilon}(u)$.*

Proof. Let $(v^{(1)}, v^{(2)}, \ldots, v^{(k)})$ be the elements from $S^k_{z,\varepsilon}(u)$ sorted by increasing triangular distance (and by increasing y coordinate among those with the same distance). We show by induction on $i \geq 1$ that after i iterations of the while loop, the vertices $v^{(1)}, \ldots, v^{(i)}$ are black.

By definition of $\Theta^+_6(V)$, the vertex v defined at line 2 is $v^{(1)}$. It is the only gray vertex at the beginning of the first iteration of the while loop, so it is the only black vertex at the end of this iteration.

Then, we suppose that for $i \in [\![1, k-1]\!]$, after i iterations of the while loop, the vertices $v^{(1)}, \ldots, v^{(i)}$ are black. We show that $v^{(i+1)}$ is the vertex which becomes black during the $(i+1)$-th iteration, that is to say that it is gray and that among the gray vertices, it is the nearest from u. By definition of the order on the $(v_j)_j$ points, there are only three cases:

- $v^{(i)} \in C_x^+(v^{(i+1)})$. Let w be the outgoing vertex of $v^{(i+1)}$ in $\Theta_6^+(V)$ in the same cone. We show that w is a black vertex.

  - $w_x \leq v_x^{(i)} < u_x$, the first inequality since $(v^{(i+1)}, w) \in \Theta_6^+(V)$, the second since $v^{(i)} \in C_z^+(u)$.
  - $w_y < v_y^{(i+1)} \leq u_y$, the first inequality since $w \in C_x^+(v^{(i+1)})$, the second since $v^{(i+1)} \in C_z^+(u)$.
  - u and w have same rank, thus : $w_z = u_z + (u_x - w_x) + (u_y - w_y) > u_z$ using the previous inequalities,
  - $w_z \leq v_z^{(i+1)} \leq (1+\varepsilon)u_z$ since $w \in C_x^+(v^{(i+1)})$ and $v^{(i+1)} \in S_{z,\varepsilon}^k(u)$.

  Therefore, $w \in C_z^+(u)$ is black. $\texttt{Rep}_\texttt{x}$ processes vertices from $E_x(w) \cap C_z^+(u)$ in increasing triangular distance from u and each vertex $w' \in E_x(w) \cap C_z^+(u)$ nearer from u than $v^{(i+1)}$ is black by induction hypothesis. Thus $\texttt{Rep}_\texttt{x}(w) = v^{(i+1)}$ at the $(i+1)$-th iteration and $v^{(i+1)}$ is gray.

- $v^{(i)} \in C_y^+(v^{(i+1)})$. Let w be the outgoing vertex of $v^{(i+1)}$ in $\Theta_6^+(V)$ in the same cone. We show that w is a black vertex.

  - $w_x \leq v_x^{(i+1)} < u_x$, the first inequality since $w \in C_y^+(v^{(i+1)})$, the second since $v^{(i+1)} \in C_z^+(u)$.
  - $w_y \leq v_y^{(i)} \leq u_y$, the first inequality since $(v^{(i+1)}, w) \in \Theta_6^+(V)$, the second since $v^{(i)} \in C_z^+(u)$.
  - u and w have same rank, thus : $w_z = u_z + (u_x - w_x) + (u_y - w_y) > u_z$ using the previous inequalities,
  - $w_z < v_z^{(i+1)} \leq (1+\varepsilon)u_z$ since $w \in C_y^+(v^{(i+1)})$ and $v^{(i+1)} \in S_{z,\varepsilon}^k(u)$.

  Therefore, $w \in C_z^+(u)$ is black. $\texttt{Rep}_\texttt{y}$ processes vertices from $E_y(w) \cap C_z^+(u)$ in increasing triangular distance from u and each vertex $w' \in E_y(w) \cap C_z^+(u)$ nearer from u than $v^{(i+1)}$ is black by induction hypothesis. Thus $\texttt{Rep}_\texttt{y}(w) = v^{(i+1)}$ at the $(i+1)$-th iteration and $v^{(i+1)}$ is gray.

- $v^{(i)} \in C_z^-(v^{(i+1)})$. By definition of the order on the $(v_j)_j$ points, $v^{(i+1)}$ is the outgoing vertex of $v^{(i)}$ in its cone $C_z^+(v^{(i)})$ in $\Theta_6^+(V)$. Thus, $v^{(i+1)}$ is gray.

In all cases, $v^{(i+1)}$ gray, and it is the nearest one from u by induction hypothesis and by definition of the order on the $(v_j)_j$ points. □

7.1.3 Complexity

The following data structures are used:

- The black vertices are stored in a linked list, each cell containing a vertex v and pointers to the three vertices it represents $\texttt{Rep}_\texttt{x}(v)$, $\texttt{Rep}_\texttt{y}(v)$ et $\texttt{Rep}_\texttt{z}(v)$.

- The gray vertices are in a balanced binary tree (Section A.2) $\mathcal{T}$ ordered over the z coordinate. The removal of the minimum in $\mathcal{T}$ is in $O(\log|\mathcal{T}|)$. We consider that $\mathcal{T}$ is a set to avoid duplicate gray vertices. Thus, an insertion requires a prior find request to know if the element to insert is already in $\mathcal{T}$. Both insertion and find requests are in $O(\log|\mathcal{T}|)$.

- Each gray vertex v has, for each direction, a linked list of pointers to the black vertices of which they are the representatives, i.e. three lists containing pointers to the w such that $\texttt{Rep}_\texttt{x}(w) = v$, $\texttt{Rep}_\texttt{y}(w) = v$ or $\texttt{Rep}_\texttt{z}(w) = v$. A pointer from v to each of these pointers is stored with v in order to modify each list in constant time.

- The sets E_x and E_y are balanced binary trees in order to have Lines 23 and 24 in $O(\log n)$.

Theorem 82. *Given $k \in \mathbb{N}$, the time complexity of Algorithm 17 is in $O(k \log n)$ with n the number of points in V.*

Proof. The algorithm does not color more than k vertices in black. Thus there are no more than k iterations of the while loop. Moreover, each black vertex represents at most three gray vertices. So there are no more than $3k$ gray vertices and $|\mathcal{T}| \leq 3k$. We analyze the operations in the while loop to express their cumulative complexity, i.e. their sum over all the iterations of the while loop:

- The extraction of the minimum from $\mathcal{T}$ is in $O(\log k)$, giving an overall complexity in $O(k \log k)$.

- Each insertion in $\mathcal{T}$ is in $O(\log k)$ and there are at most $3k$ of those. Thus the overall insertion complexity is in $O(k \log k)$.

- The inner while loops go through each arc of $C_z^+(u)$ outgoing from a black vertex only once. There is a $O(k)$ arcs concerned since they lie in a zone containing at most $3k$ vertices and $\Theta_6^+(V)$ is planar. Therefore, there are $O(k)$ iterations of these loops during the whole algorithm. Their internal operations are in logarithmic time if we use balanced binary trees for `next` and `previous` but in constant time if we also have a combinatorial map to represent the graph (Lemma 55). The total complexity is therefore $O(k \log n)$ or $O(k)$ depending on the structure used.

- If the current vertex is v, finding its first represented vertex $\texttt{Rep}_\texttt{x}(v)$ is performed with a logarithmic query on the tree containing $E_x(v)$ which size is in $O(n)$. So the complexity is in $O(\log n)$. The same is true for $\texttt{Rep}_\texttt{y}(u)$. Repeating these operations for each iteration of the while loop gives an overall complexity in $O(k \log n)$.

The initialization of $\texttt{Rep}_\texttt{x}$ and $\texttt{Rep}_\texttt{y}$ dominates the others steps and the time complexity of the algorithm is therefore in $O(k \log n)$. □

7.2 Dominating set

In this section, our goal is to cover in one direction the set V by a subset, as small as possible. The natural object corresponding to the wanted subset is a minimal dominating set. First, we introduce formally the definitions in Section 7.2.1. Then we describe a classical algorithm to compute a reasonable size dominating set in Section 7.2.2. The correctness is proven in Section 7.2.3 and we provide a detailed data structure in order to establish the complexity (Section 7.2.4). Finally, we recall the classical result maximizing the size of the dominating set computed in Section 7.2.5.

7.2.1 Definitions and related work

We first introduce the coverage definition for a given direction.

Definition 83. *Let $V' \subseteq V$ et $\varepsilon > 0$. We say that V' $(1+\varepsilon)$-covers V with respect to z if:*

$$\forall v \in V, \exists v' \in V' \cap \mathcal{C}_z^+(v), \text{ such that } v' \ (1+\varepsilon)\text{-covers } v.$$

The definition is similar for the other two directions.

We note $\mathcal{N}_{z,\varepsilon}(V) = (V, A)$ the graph such that $A = \big\{(u,v) | v \in V, u \in S^n_{x,\varepsilon}(v)\big\}$, with $S^n_{x,\varepsilon}(u)$ defined in Section 7.1. Computing a $(1+\varepsilon)$-cover of V w.r.t z is equivalent to computing a dominating set in $\mathcal{N}_{z,\varepsilon}(V)$. We recall the definition of a dominating set in a directed graph.

Definition 84 (Directed out-dominating set)**.** *Let $\mathcal{G} = (V, E)$ be a simple directed graph and $\mathcal{S} \subseteq V$. We say that $\mathcal{S}$ is a directed out-dominating set of $\mathcal{G}$ if:*

$$\forall u \in V \backslash \mathcal{S}, \exists s \in \mathcal{S}, (s,u) \in E$$

The definition of a *directed in-dominating set* is symmetrical. Note that the directed out-dominating set of a given directed graph corresponds to the directed in-dominating set of the same graph but with in the reverse orientation. In the following, *directed dominating set* stands for the *directed out-dominating set*.

The orientation can make a real difference. In 1963, Erdös [Erd63] showed that it exists an orientation of the complete graph requiring a directed dominating set of size at least $\log n - 2\log\log n$ but less than $\log(n+1)$ whereas any single vertex is a dominating set of the complete graph. More generally, there is a huge litterature for the classic dominating set but the directed dominating set has received less attention. In [CH12], the authors exhibits the differences and relationships between dominating sets and directed dominating sets for different graph families and parameters.

As for the undirected case, the directed dominating set computation can be done with a $\log n$-approximation (see Theorem 11 of [CC08]) using the classic greedy approximation algorithm:

1. sort the vertices with respect to the out-degree,
2. add the vertex of highest out-degree to the directed dominating set and remove it from the graph,

3. repeat step 2 until every vertex is dominated.

Even for bounded degree directed graphs, such an algorithm can take $\Theta(n^2)$ computation steps. Can we propose an efficient algorithm with a better computational time complexity ?

In the following, we present an algorithm running in $O(\Delta_-\Delta_+ n)$ time. The main contribution is to provide a detailed data structure achieving the best worst-case time complexity to our knowledge.

7.2.2 Algorithm

We first provide a high level presentation of the existing greedy algorithm computing directed dominating sets.

Outline. Each vertex is associated with a color: white (non-dominated), black (dominating) and gray (dominated non-dominating). For any vertex u, let $p_{degree}(u)$ be its pseudo outgoing degree, i.e. its current outgoing degree, plus one if the vertex is white. This pseudo degree depends on the moment of the algorithm's execution at which it is considered.

The initialization colors all vertices in white. Then the following method is iterated:

1. a vertex of maximum p_{degree} is selected,
2. it is colored in black (it becomes a dominant),
3. its outgoing neighbors are colored in gray (which are necessarily white because of the following step),
4. all the incoming edges of the newly colored vertices (in gray or black) are deleted.

The algorithm stops when there is no more white vertex. The corresponding pseudo-code is presented in Algorithm 18. The operation `removeInArc` deletes the ingoing arcs of the given vertex.

Remark. *Before selecting the vertices of maximum p_{degree}, we can apply the steps* 2 *to* 4 *of the algorithm outline to the source vertices, i.e. those which do not have incoming arcs. Indeed, these last ones are necessarily in any dominating set.*

7.2.3 Correctness

Theorem 85. *Given a graph $\mathcal{G} = (V, A)$, Algorithm 18 terminates and outputs a dominating set of $\mathcal{G}$.*

Proof. The algorithm terminates since the number of white vertices decreases at each iteration of the while loop. Furthermore, it is correct because at the end:

- the black vertices are in $\mathcal{S}$,

Input: $\mathcal{G} = (V, A)$ a graph
Output: $\mathcal{S} \subseteq V$
1 $\mathcal{S} \leftarrow \emptyset$
2 **foreach** $u \in V$ **do**
3 $\quad Color(u) \leftarrow white$
`// Delete nodes with highest` p_{degree}
4 **while** *white nodes exist* **do**
5 $\quad$ Let u be the highest pseudo-degree vertex
6 $\quad \mathcal{S} \leftarrow \mathcal{S} \cup \{u\}$
7 $\quad Color(u) \leftarrow black$
8 $\quad$ **foreach** v *s.t.* $(u, v) \in A$ **do**
9 $\quad\quad Color(v) \leftarrow gray$
10 $\quad\quad \mathcal{G}$.`removeInArc`(v)
11 $\quad \mathcal{G}$.`removeInArc`(u)
12 **return** $\mathcal{S}$

Algorithm 18: High-level algorithm to compute a directed dominating set

- there is no more white vertices, which implies that any non black vertex is gray,
- if a vertex if gray, then, when it was colored, one of its ingoing neighbors was being colored in black. This black vertex dominates it.

Thus, any vertex is dominated, by itself or by another one. □

7.2.4 Complexity

In order to precisely analyze the complexity, we provide ourselves with simple data structures. Then a detailed version of Algorithm 18 is presented in Algorithm 22.

Data structures

We put the set of vertices $\{u_1 \ldots, u_n\}$ in a data structure $\mathcal{D}$. This structure is a linked list where each cell contains the set of vertices having the same pseudo-degree. Each cell stores also its corresponding pseudo-degree. The cells are sorted in pseudo-degree decreasing order. The set of vertices of the same pseudo-degree is itself a linked list.

To each vertex u is associated a quadruplet $(adj^+(u), adj^-(u), ptr_{\mathcal{D}}(u), ptr_{\mathcal{D}_{pos}}(u))$ with:

- $adj^+(u)$ the list of u's outgoing vertices, represented as a linked list. To each vertex $v \in adj^+(u)$ is associated $ptr_{u \to v}$ pointing to the position of u in $adj^-(v)$.
- $adj^-(u)$ the list of u's ingoing vertices, represented as a linked list. To each vertex $w \in adj^-(u)$ is associated $ptr_{u \leftarrow w}$ pointing to the position of u in $adj^+(w)$.
- $ptr_{\mathcal{D}}(u)$ pointing to the cell of $\mathcal{D}$ corresponding to the pseudo-degree of u. The values of $ptr_{\mathcal{D}}$ are stored in an array.

- $ptr_{\mathcal{D}_{pos}}(u)$ pointing to the cell containing u in the linked list in $\mathcal{D}$ pointed by $ptr_{\mathcal{D}}(u)$. The values of $ptr_{\mathcal{D}_{pos}}$ are stored in an array.

The color are implicit: $\mathcal{D}$ contains exactly the white and gray vertices, and the difference between those two is inferred from the p_{degree} value of the $\mathcal{D}$'s cell containing the vertex. This difference can as well be represented by a booleen. Notice also that combinatorial maps might also be used as an alternative to represent graphs instead of adj^+ and adj^- (Section A.4). Furthermore, each vertex is associated with its initial outgoing degree, that is the size of adj^+ (not necessary but simpler for Algorithm 19, computing it at start does not change the overall complexity).

Subroutines

We describe the subroutines used in the Algorithm 22:

- `create`$\mathcal{D}$ (Algorithm 19) creates the initial structure $\mathcal{D}$,
- `popMaxElement` (Algorithm 20) removes from $\mathcal{D}$ a vertex of maximum p_{degree} and returns it,
- `updateElement` (Algorithm 21) updates the position of a vertex in $\mathcal{D}$ when its pseudo-degree has changed.

Those subroutines use classic linked lists primitives detailed in Section A.1. It is easy to provide a linked list implementation with those primitives working in constant time.

Subroutine `create`$\mathcal{D}$**.** The initialization of $\mathcal{D}$ consists in sorting all vertices by pseudo-degree and then to insert them in $\mathcal{D}$. At the same time, we initialize the pointers $ptr_{\mathcal{D}}$ and $ptr_{\mathcal{D}_{pos}}$, as described in Algorithm 19. If we want Algorithm 22 to process the source vertices before the others, we place them at the beginning of $\mathcal{D}$ (therefore at the end in Algorithm 19) in an additional cell.

Subroutine `popMaxElement`**.** In order to get the maximum pseudo degree vertex, it is sufficient to take the first vertex of the first list of $\mathcal{D}$. But we also add the update of $\mathcal{D}$ in order to remove u and to keep the first cell not empty.

Subroutine `updateElement`**.** The subroutine *updateElement* updates the position in $\mathcal{D}$ of a vertex that has changed its degree. The test Line 10 guarantees that whenever the input vertex has a null pseudo-degree, it is removed from $\mathcal{D}$.

Algorithm revisited

From Algorithm 18, the operations `removeInArc` and `removeOutArc` translate into deletions from the lists adj^- and adj^+ in Algorithm 22, using the pointers $p_{u\to v}$ and $p_{u\leftarrow v}$. Processing the lists adj^+ and adj^- can be written with primitives too but we avoid to unnecessarily complexify the pseudo-code, the goal being to make explicit the handling of $\mathcal{D}$.

Input: $\mathcal{G} = (V, A)$ a graph
Output: $\mathcal{D}$

```
1  (u_0, ··· , u_{n-1}) ← IncreasingPseudoDegreeOrderSort(V)
2  D = createList()
3  i ← 0
4  while i < n do
5      D_pos ← createList()
6      p ← p_degree(u_i) // size of adj^+(u_i)
7      while i < n and p_degree(u_i) = p do
8          D_pos ← insertBegin(D_pos, u_i)
9          ptr_{D_pos}(u_i) ← D_pos
10         i ← i + 1
11     D ← insertBegin(D, D_pos)
12     while not isEmpty(D_pos) do
13         ptr_D(value(D_pos)) ← D
14         D_pos ← next(D_pos)
15     i ← i + 1
16 return D
```

Algorithm 19: create$\mathcal{D}$

Input: $\mathcal{D}$ a linked list of linked lists

```
1  u ← value(value(D))
2  D ← replaceBegin(D, deleteBegin(value(D))
3  if isEmpty(value(D)) then
4      D ← deleteBegin(D)
5  return u
```

Algorithm 20: popMaxElement

Input: $\mathcal{D}$ a linked list of linked lists, u vertex

$\mathcal{D}_{curr} \leftarrow ptr_{\mathcal{D}}(u)$ `// cell of` $\mathcal{D}$ `containing the list containing` u
$\mathcal{D}_{next} \leftarrow \texttt{next}(\mathcal{D}_{curr})$
$p \leftarrow p_{degree}(\mathcal{D}_{curr})$
$\mathcal{D}_{pos} = \texttt{value}(\mathcal{D}_{curr})$ `// list in` $\mathcal{D}$ `containing` u
$\mathcal{D}_{pos} \leftarrow \texttt{deleteAfter}(\mathcal{D}_{pos}, ptr_{\mathcal{D}_{pos}}(u))$ `// delete` u `from it`
if $\texttt{isEmpty}(\mathcal{D}_{pos})$ **then**
    $\mathcal{D} \leftarrow \texttt{deleteAfter}(\mathcal{D}, \mathcal{D}_{curr})$
else
    $\mathcal{D} \leftarrow \texttt{replaceAfter}(\mathcal{D}, \mathcal{D}_{curr}, \mathcal{D}_{pos})$
if $p > 1$ **then**
    if $\texttt{isEmpty}(\mathcal{D}_{next})$ *or* $p_{degree}(\mathcal{D}_{next}) \neq p - 1$ **then**
        $\mathcal{D}_{pos} \leftarrow \texttt{createList}()$
        $\mathcal{D} \leftarrow \texttt{insertAfter}(\mathcal{D}, \mathcal{D}_{curr}, \mathcal{D}_{pos})$
        $\mathcal{D}_{next} \leftarrow \texttt{next}(\mathcal{D}_{curr})$
        $\mathcal{D}_{curr} \leftarrow \mathcal{D}_{next}$
    $\mathcal{D}_{pos} \leftarrow \texttt{insertBegin}(\texttt{value}(\mathcal{D}_{next}), u)$
    $\mathcal{D} \leftarrow \texttt{replaceAfter}(\mathcal{D}, \mathcal{D}_{curr}, \mathcal{D}_{pos})$
    $ptr_{\mathcal{D}_{pos}} \leftarrow \texttt{value}(\mathcal{D}_{next})$

Algorithm 21: updateElement

Input: $\mathcal{G} = (V, A)$ a graph
Output: $\mathcal{S} \subseteq V$

$\mathcal{S} \leftarrow \emptyset$
$\mathcal{D} \leftarrow \texttt{create}\mathcal{D}(\mathcal{G})$
`// Delete nodes with highest` p_{degree}
while *not* $\texttt{isEmpty}(\mathcal{D})$ **do**
    $u \leftarrow \texttt{popMaxElement}(\mathcal{D})$
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{u\}$
    foreach $x \in adj^-(u)$ **do**
        $Remove(u, adj^+(x))$
        $\texttt{updateElement}(\mathcal{D}, x)$
    $adj^-(u) \leftarrow \emptyset$
    foreach $v \in adj^+(u)$ **do**
        $Remove(u, adj^-(v))$
        foreach $x \in adj^-(v)$ **do**
            $Remove(v, adj^+(x))$
            $\texttt{updateElement}(\mathcal{D}, x)$
        $adj^-(v) \leftarrow \emptyset$
    $adj^+(u) \leftarrow \emptyset$
return $\mathcal{S}$

Algorithm 22: Low-level algorithm to compute a directed dominating set

Theorem 86. *Given a graph with n vertices and m arcs, Algorithm 22 has a time complexity in $O(n\Delta_+\Delta_-)$ and uses a $O(n+m)$ memory space.*

Proof. First, we give the complexity of each subroutine:

- `create`$\mathcal{D}$ is in $O(n)$, by using a counting sort. Each vertex requires a constant number of primitive calls.
- `popMaxElement` and `updateElement` are in $O(1)$ since it is a constant number of primitive calls.

For each vertex u, the algorithm processes the list of outgoing neighbors of u. For each one of them, it processes their list of ingoing neighbors. For any arc (u, v), the operation `Remove`(u, v) is in constant time : the positions of v in $adj^+(u)$ and of u in $adj^-(v)$ are deduced from each other in constant time thanks to the associated pointers $ptr_{u\to v}$ and $ptr_{v\leftarrow u}$. It is then sufficient to use the primitive `deleteAfter`.

So each iteration of the while loop is in $O(\Delta_+\Delta_-)$. The accumulation of the iterations is then in $O(n\Delta_+\Delta_-)$ since $\mathcal{D}$ contains initially n elements and at least one is removed at each iteration. The initialization of the algorithm being linear, we have the announced time complexity.

Each array is of size n and $\mathcal{D}$ contains at most one subcell per vertex, thus is of size n too. The linked lists adj^+ and adj^- contains one cell per arc and thus are of size m altogether. □

7.2.5 Size of the output

The resulting dominating set cannot be much larger than a minimum set as the following Theorem states.

Theorem 87. *Let $\mathcal{S}$ be the output of the algorithm and $\mathcal{S}^*$ an optimal dominating set. Let Δ_{out} be the maximum outgoing degree over the set of vertices at the beginning. We have:*

$$|\mathcal{S}| \leq \ln(\Delta_{out})|\mathcal{S}^*|$$

Remark. *At any moment of the algorithm, each vertex has only white vertices as outgoing neighbors since any change of color implies the suppression of the incoming edges.*

Proof. This proof strongly relies on lecture notes written by Fabian Kuhn, University of Freiburg, for undirected graphs.

Stars. We partition the graph into stars. We consider a subgraph $\mathcal{G}' = (V, E')$ such that each vertex of $V\backslash\mathcal{S}^*$ has in-degree 1 and all arcs go from vertices in $\mathcal{S}^*$ to vertices in $V\backslash\mathcal{S}^*$. We thus obtain a subgraph of $\mathcal{G}$.

Each connected component of it is a star centered on a vertex of $s \in \mathcal{S}^*$ and the arcs are outgoing from s to vertices of $V\backslash\mathcal{S}^*$. The set of stars vertices is a partition of V into different connected components: it is a cover because each vertex is dominated, and the stars are disjoint because we take only one incoming edge per vertex in $V\backslash\mathcal{S}^*$.

Weight. During the algorithm, when a vertex is selected in order to insert it in $\mathcal{S}$, a weight of 1 is given to it, and redistributed. If s is inside $\mathcal{S}$, let $\mathcal{N}(s)$ be the set of outgoing neighbors of s (white vertices) at the time of its insertion in $\mathcal{S}$, in addition to s if the latter is white when the algorithm selects it. The weight of s is redistributed equally to all the vertices of $\mathcal{N}(s)$. Thus, each one obtains a weight of $\dfrac{1}{|\mathcal{N}(s)|}$.

Let the weight of an arc be the sum of the weights of its vertices, and similarly for the total weight of the graph, which is thus equal to $|\mathcal{S}|$ since each vertex of $\mathcal{S}$ is given a weight of 1 before it is redistributed.

With this process, a vertex increases its weight only when it loses its white color. This gain happens only once during the whole execution. For any vertex u, let w_u be this final weight.

Bound on the weight of a star. It is now sufficient to show that each star has at most a weight of $\ln(\Delta_{out})$. Indeed, there are at most $|\mathcal{S}^*|$ stars so the total weight of the graph would be at most $|\mathcal{S}^*| \ln(\Delta_{out})$.

Let $s \in \mathcal{S}^*$ and w_e be the weight of the star associated to s. Note first that the number of vertices in a star is at most $\Delta_{out} + 1$, since the center has at most Δ_{out} neighbors.

Let $(u_i)_{1 \leq i \leq k}$ be the set of vertices of the star of s sorted by order of loss of their white color in the process of the algorithm (for those which change at the same time, the order does not matter). Let $(p_i)_{1 \leq i \leq k}$ be the values of $p_{degree}(s)$ just before u_i changes from white to another color. At this moment, the vertices $(u_j)_{i<j \leq k}$ are still white by definition of the order on u_j. So $p_i \geq k - i + 1$. Thus:

$$\begin{aligned}\sum_{i=1}^{k} \frac{1}{p_i} &\leq \sum_{i=1}^{k} \frac{1}{k-i+1} = \sum_{i=1}^{k} \frac{1}{i} \\ &\leq \ln(k+1) + 1 \leq \ln(\Delta_{out} + 1) + 1 \\ &\leq \ln(\Delta_{out}) + 2\end{aligned}$$

Furthermore, $w_{u_i} \leq 1/p_i$. Indeed, once s has been processed, no u_i can be white. Thus, when u_i gets its weight of w_{u_i}, the vertex that redistributes it has a pseudo degree greater than or equal to p_i. Therefore:

$$w_e = \sum_{i=1}^{k} w_{u_i} \leq \ln(\Delta_{out}) + 2$$

□

7.3 Minimizing representatives in tricriteria shortest path computation

In order to minimize the number of paths kept by `Sample` for tricriteria shortest path computation, we propose an alternative solution to the one given in Section 6.4. It

consists in combining Algorithm 17 and Algorithm 18.

Definition 88 (Dominating Sample Sector Algorithm)**.** *Let $\mathcal{S}$ and $\mathcal{R}$ the inputs of* `Sample` *function. We note $V_{\mathcal{R}}$ all the costs of $\mathcal{R}$. We compute a dominating set $\mathcal{D}_x$, resp. $\mathcal{D}_y$, resp. $\mathcal{D}_z$, of $\mathcal{N}_{x,\varepsilon}(V_{\mathcal{R}})$, resp. $\mathcal{N}_{y,\varepsilon}(V_{\mathcal{R}})$, resp. $\mathcal{N}_{z,\varepsilon}(V_{\mathcal{R}})$. Then we output the paths whose costs are in $\mathcal{D}_x \cup \mathcal{D}_y \cup \mathcal{D}_z$, minus those dominated by $\mathcal{S}$.*

We call this algorithm Dominating Sample Sector*.*

Definition 89 (DSector Algorithm)**.** *Algorithm* DSector *is the* Meta Rank *algorithm (Alg. 6) using* Dominating Sample Sector *as the* `Sample` *function.*

Theorem 90. *Let S_ε be the size of the output of* DSector*. The output-sensitive time complexity of* DSector *is in $O\left((\Delta S_\varepsilon)^4\right)$.*

Proof. If the input of Dominating Sample Sector is of size R, then the applications of Algorithm 17 are in $O(R \log R)$ and the ones of Algorithm 18 are in $O(R^3)$. The removal of dominated paths is in $O(R^2)$.

Therefore, repeating Dominating Sample Sector as a `Sample` function over each vertex and rank gives $C_{\texttt{Sample}}(n, S_\varepsilon, \Delta, \Lambda) = O((\Delta S_\varepsilon)^4)$. Applying Theorem 15 allows us to conclude. □

Even though the time complexity is huge, this algorithm guarantees that at each rank, if there is R current rank paths, then it keeps at most $3\log(R)R^*$ paths, with R^* the optimal number of paths to kept with respect to the ε-weak framing property.

The complexity can be improved by computing only the k nearest points in Algorithm 17, i.e. the $S^k_{i,\varepsilon}$, with $k << n$. However, the guarantee on the number of kept points is no longer true.

Remark. *One could simply compute a dominating set on the subgraph of $\Theta_6^+(V)$ by keeping only the arcs outgoing in the cones $\mathcal{C}_z^+$. But a point may be connected to many others in the same cone in $\mathcal{N}_{z,\varepsilon}(V)$, so we hope to get a smaller dominating set using this augmented graph.*

This method was not conclusive in preliminary experiments. We think that this is due to the too small number of paths having the same rank, which makes the overhead of this method the main part of the execution time. Complementary experiences on tricriteria graphs, with more path sharing the same rank, must be done.

Conclusion

In this thesis, we first propose solutions for multicriteria shortest path computation in Part I. Then, we focus on Theta-graphs in Part II, for which we develop dynamic maintenance and related algorithms.

Shortest path computation

A framework. Based on the algorithm MC Dijkstra, we propose a variant Meta Rank, in the form of a framework (Section 3.2). It has the advantage to process the paths by buckets since it processes paths in increasing rank order. The variable component of this framework is contained in a function `Sample`. This function processes the buckets of paths one by one. The data structures used are detailed and the complexity of Meta Rank is expressed as a function of `Sample`'s one. Depending on the choice of `Sample`, Meta Rank computes exact or approximate Pareto sets.

Exact computation. First, we introduce Dijkstra Post, the algorithm corresponding to MC Dijkstra in Meta Rank's framework (Section 3.3.2). Then, we notice that processing paths by buckets allows us to efficiently prune dominated paths using known methods. These methods are offline in general and are therefore not suitable for MC Dijkstra, which treats the paths one by one. We name Bucket the algorithm Meta Rank using a function `Sample` based on these methods (Section 3.3.1). The efficiency of Bucket depends on the size of the buckets.

Therefore, it would be interesting to implement Bucket *and to compare it experimentally with both* Dijkstra Post *and* MC Dijkstra. *However, the methods used in* Bucket *are known to be efficient only for large Pareto sets. It is therefore necessary to generate graphs with such Pareto set sizes, in addition to the tournament presented in Section 4.3.1.*

For the MC Dijkstra algorithm in both 2D and 3D, dominated paths can be pruned efficiently with online methods. For the 2D case, we detail the data structures used in what we call MC Dijkstra `2D` (Section 4.1.1). This algorithm has the advantage of processing less paths than Dijkstra Post but it uses more complex data structures. Therefore, we compare them experimentally (Section 4.3), and depending on the input, each can outperform the other. *However, the performance difference can sometimes be surprising and it deserves a more detailed investigation.*

In MC Dijkstra `2D`, removing dominated paths as detailed in Section 4.1.1 rather than with a naive algorithm allows a significant theoretical gain, but also a

practical one. Such a theoretical gain is also feasible in 3D since it is also possible to prune the dominated paths online. *It would therefore be interesting to propose detailed data structures for the 3D setting, as well as an experimental study.*

Finally, it is particularly frustrating to know so little about online methods for pruning dominated paths. To the best of our knowledge, no method allows a theoretical gain in dimension greater than 3. Furthermore, the 2D and 3D efficient methods are both online under the assumption that the paths are considered in lexicographic order. However, META RANK considers paths in increasing rank order. We can easily find an alternative version for the 2D, based on a binary search algorithm. *But to find an efficient online algorithm for the 3D case in increasing rank order (and higher dimensions whatever the order) remains open.*

Approximated computation. In order to compare the output sensitive complexities of approximation algorithms with exact algorithm ones, we introduce the Pareto compatible concept in Section 4.2.2. An algorithm is Pareto compatible if it outputs a subset of the Pareto set.

Then, we propose an approximation algorithm for the bicriteria case: FRAME (Section 4.2.3). This algorithm is Pareto compatible and its complexity cannot be worse than those of MC DIJKSTRA and DIJKSTRA POST in order of magnitude. Despite the overhead that FRAME induces, compared to these two exact algorithms, it can prune optimal paths. We check that, in practice, this pruning is substantial. We conduct a detailed experimental study which shows that if the Pareto set sizes are large, the pruning and the gain are significant.

First, it would be interesting to deepen the experimental study with even larger Pareto sets, especially on real graphs with anticorrelated criteria. We do not have such graphs for now. Thus, a graph generation work is yet to be done. A more detailed study about the impact of the value of ϵ also remains to be performed, to identify the relationship that this parameter has with the others.

In order to generalize to higher dimensions, we propose the approximation algorithms SECTOR, SSECTOR and QSSECTOR (Section 3.5). Those are designed for any number of dimensions. The first one has the advantage of being simple to implement and has a quadratic complexity as well as MC DIJKSTRA in the general case. However, we lack sufficient data in order to perform a relevant experimental study, especially in the tricriteria when many paths share the same rank.

Our two other algorithms improve the complexity of SECTOR by relying on range queries. But for our three approximation algorithms, the stated complexities depend on the size of the output. They are not Pareto compatible, which makes them incomparable with exact algorithms. *A highly interesting further work is then to implement them in order to make an experimental comparison with* DIJKSTRA POST *and* MC DIJKSTRA.

Another main challenge is to adapt these algorithms to be Pareto compatible. This property is obtained for FRAME *since it uses a strong framing criterion to prune paths. However, the results related to this criterion do not generalize to higher dimensions.*

Half-Theta-graphs

FRAME's generalization to higher dimensions starts with the tricriteria case. In this context, the notion of framing is naturally linked to specific geometric graphs: Θ_6^+ (half-Theta-6-graphs). Some properties are already known on these graphs but the development of several algorithms is necessary in order to use those for shortest path computation.

Dynamic Θ_6^+. First, we propose an efficient algorithm allowing the maintenance of a Θ_6^+ when a point is inserted (Section 6.2). This algorithm requires handling a point location structure. This operation is the bottleneck of the algorithm in terms of complexity. *Thus, we should try to improve the point location complexity. In particular, the known point location data structures are designed in a more general context than ours. The latter may be easier to handle.*

Another way to improve the complexity is to avoid point location requests. An idea consists in finding the nearest point from the inserted one, and then iterating an algorithm from this point. However, we have not yet succeeded in proposing an efficient algorithm starting from this point.

The opposite problem consists in deleting a point while preserving the Θ_6^+ structure. We propose an algorithm solving this issue in Section 6.3. *A first improvement consists in decreasing the complexity to a linear one. This possibility seems intuitive if we reverse the steps of the insertion algorithm. But the formalization is not obvious and is yet to be done.*

We propose to use these dynamic Θ_6^+ in order to speed up our algorithm SECTOR (Section 6.4). *It would be interesting to implement our dynamic maintenance algorithms to evaluate the performance of the modified* SECTOR. *Here again, datasets that involve many paths having the same rank need to be generated.*

Proximity queries and dominating set. In order to maximize the pruning of optimal paths in an approximated multicriteria shortest path computation, we propose to augment the Θ_6^+ structure. In the latter, each point is connected to its nearest point in each positive cone. The augmentation consists in also connecting it to all other points in the same cone at a bounded distance. In order to obtain this structure, we propose an algorithm which, given a Θ_6^+, a point and one of its positive cone, computes the set of points at a bounded distance (Section 7.1).

It may seem feasible to reduce our complexity to a linear one but such an improvement remains open and requires a better understanding of the structural properties.

Afterwards, we detail a well known algorithm computing a dominating set of reasonable size on this graph (Section 7.2). The data structures description given allows a straightforward implementation and a detailed proof of the complexity. Applied on an augmented Θ_6^+, this algorithm allows to prune optimal paths in META RANK, by keeping only the dominating set computed (Section 7.3).

The complexity is potentially cubic in the number of vertices. Thus, this method might be slower than the one based only on dynamic Θ_6^+. Nevertheless, it allows to prune more drastically the optimal paths and we can hope for a practical gain since

Meta Rank applies this method several times and its performance depends on the number of paths considered at each step.

It is therefore particularly interesting to conduct an experimental study to compare Sector *and both these variants based on Θ_6^+. To do this, we need to generate tricriteria graphs with many paths having the same rank.*

Broader perspectives. *We only studied Θ_6^+ and Θ_6. A first generalization would be to change the number of cones. The Θ_6^+ structure is particularly rigid since it is a triangulation. The correctness of the insertion algorithm relies heavily on it. Therefore, our approach does not seem relevant for a general Θ_k. We would have to develop other techniques.*

The distance notion can also be generalized. It sounds reasonable to adapt out k-nearest neighbor algorithm to any convex distance, with a complexity inversely proportional to the aspect ratio of a ball. Technical difficulties remain in the proof.

As for the shortest path computation, another broader issue concerns the dimension. In Meta Rank*, the function* `Sample` *processes the same rank paths all together. In the tricriteria setting, those paths are in the same plane. Thus, if we consider multicriteria paths in dimension $d+1$, can we naturally adapt this study to geometric graphs in dimension d?*

Our algorithms do not limit to be used by themselves. They can be integrated in precomputation methods. For example, in a hub labeling method, the shortest paths are concatenations of two precomputed shortest paths, the intermediate vertex being called a hub. If we set correctly the approximation factor for the precomputation, we can obtain approximated Pareto sets, concatenating precomputed paths. A preliminary work proposing a multicriteria hub labeling algorithm has been developed by Yoan Picchi during his internship at LaBRI. In this work, the hubs are separators. In order to speed up the precomputation step, a multi-level separator generalization is proposed but it requires more caution with the approximation factors.

Another promising idea is the integration of our algorithms into more general settings. Precomputation methods for static graphs can be used as a step in multimodal algorithms. Similarly, our algorithms can be used to speed up the computation by summarizing intermediate Pareto sets for approximate multimodal computation.

Finally, other definitions of approximation can be investigated. It may be relevant to consider an additive coverage rather than a multiplicative one. For example, we can consider that a trip covers another if it only takes at most ten minutes longer, for any travel time.

Bibliography

[Agg+89] A. Aggarwal, L. J. Guibas, J. Saxe, and P. W. Shor. "A linear-time algorithm for computing the voronoi diagram of a convex polygon". en. In: *Discrete & Computational Geometry* 4.6 (Dec. 1989), pp. 591–604.

[AI06] A. Andoni and P. Indyk. "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions". In: *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*. ISSN: 0272-5428. Oct. 2006, pp. 459–468.

[Aki+14] T. Akiba, Y. Iwata, K.-i. Kawarabayashi, and Y. Kawata. "Fast Shortest-path Distance Queries on Road Networks by Pruned Highway Labeling". en. In: *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. Ed. by C. C. McGeoch and U. Meyer. Philadelphia, PA: Society for Industrial and Applied Mathematics, May 2014, pp. 147–154.

[Alb+98] G. Albers, L. J. Guibas, J. S. B. Mitchell, and T. Roos. "Voronoi Diagrams of Moving Points". en. In: *International Journal of Computational Geometry & Applications* 08.03 (June 1998), pp. 365–379.

[AP14] F. Aurenhammer and G. Paulini. "On shape Delaunay tessellations". en. In: *Information Processing Letters* 114.10 (Oct. 2014), pp. 535–541.

[AZCN21] A. Al Zoobi, D. Coudert, and N. Nisse. *Finding the k Shortest Simple Paths: Time and Space trade-offs*. Research Report. Inria ; I3S, Université Côte d'Azur, Apr. 2021.

[Bas+16] H. Bast et al. "Route Planning in Transportation Networks". en. In: *Algorithm Engineering*. Lecture Notes in Computer Science. Springer, Cham, 2016, pp. 19–80.

[Bau+16] R. Bauer, T. Columbus, I. Rutter, and D. Wagner. "Search-space size in contraction hierarchies". en. In: *Theoretical Computer Science* 645 (Sept. 2016), pp. 112–127.

[Bau+20] M. Baum, J. Dibbelt, D. Wagner, and T. Zündorf. "Modeling and Engineering Constrained Shortest Path Algorithms for Battery Electric Vehicles". In: *Transportation Science* 54.6 (Oct. 2020). Publisher: INFORMS, pp. 1571–1600.

[BBS13] H. Bast, M. Brodesser, and S. Storandt. “Result Diversity for Multi-Modal Route Planning”. In: *13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*. Ed. by D. Frigioni and S. Stiller. Vol. 33. OpenAccess Series in Informatics (OASIcs). ISSN: 2190-6807. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 123–136.

[BC19] F. Bökler and M. Chimani. “Approximating Multiobjective Shortest Path in Practice”. In: *2020 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. Proceedings. Society for Industrial and Applied Mathematics, Dec. 2019, pp. 120–133.

[BDH17] T. Breugem, T. Dollevoet, and W. van den Heuvel. “Analysis of FPTASes for the multi-objective shortest path problem”. In: *Computers & Operations Research* 78.Supplement C (Feb. 2017), pp. 44–58.

[Bel58] R. Bellman. “On a routing problem”. en. In: *Quarterly of Applied Mathematics* 16.1 (1958), pp. 87–90.

[BFS19] F. Barth, S. Funke, and S. Storandt. “Alternative Multicriteria Routes”. In: *2019 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*. Proceedings. Society for Industrial and Applied Mathematics, Jan. 2019, pp. 66–80.

[Bon+10] N. Bonichon, C. Gavoille, N. Hanusse, and D. Ilcinkas. “Connections between Theta-Graphs, Delaunay Triangulations, and Orthogonal Surfaces”. en. In: *Graph Theoretic Concepts in Computer Science*. Ed. by D. M. Thilikos. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 266–278.

[Bon+12] N. Bonichon, C. Gavoille, N. Hanusse, and L. Perković. “The Stretch Factor of L1- and L{∞}-Delaunay Triangulations”. en. In: *Algorithms – ESA 2012*. Ed. by L. Epstein and P. Ferragina. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 205–216.

[Boo75] R. V. Book. “Richard M. Karp. Reducibility among combinatorial problems. Complexity of computer computations, Proceedings of a Symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Center, Yorktown Heights, New York, edited by Raymond E. Miller and James W. Thatcher, Plenum Press, New York and London 1972, pp. 85–103.” en. In: *The Journal of Symbolic Logic* 40.4 (Dec. 1975). Publisher: Cambridge University Press, pp. 618–619.

[Bos+08] P. Bose, P. Carmi, S. Collette, and M. Smid. “On the Stretch Factor of Convex Delaunay Graphs”. en. In: *Algorithms and Computation*. Ed. by S.-H. Hong, H. Nagamochi, and T. Fukunaga. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 656–667.

[BS13] P. Bose and M. Smid. “On plane geometric spanners: A survey and open problems”. en. In: *Computational Geometry*. EuroCG 2009 46.7 (Oct. 2013), pp. 818–830.

[Cas18] A. Casteigts. "Finding Structure in Dynamic Networks". en. In: *arXiv:1807.07801 [cs]* (July 2018). arXiv: 1807.07801.

[CC08] M. Chlebík and J. Chlebíková. "Approximation hardness of dominating set problems in bounded degree graphs". en. In: *Information and Computation* 206.11 (Nov. 2008), pp. 1264–1275.

[CH12] Y. Caro and M. A. Henning. "Directed domination in oriented graphs". en. In: *Discrete Applied Mathematics* 160.7 (May 2012), pp. 1053–1063.

[Cha93] B. Chazelle. "An optimal convex hull algorithm in any fixed dimension". en. In: *Discrete & Computational Geometry* 10.4 (Dec. 1993), pp. 377–409.

[Cha96] T. M. Chan. "Optimal output-sensitive convex hull algorithms in two and three dimensions". en. In: *Discrete & Computational Geometry* 16.4 (Apr. 1996), pp. 361–368.

[Cla06] K. L. Clarkson. "Nearest-Neighbor Searching and Metric Space Dimensions". en. In: *Nearest-neighbor methods for learning and vision: theory and practice* (2006), pp. 15–59.

[Cla83] K. L. Clarkson. "Fast algorithms for the all nearest neighbors problem". In: *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. ISSN: 0272-5428. Nov. 1983, pp. 226–232.

[Cla87] K. Clarkson. "Approximation algorithms for shortest path motion planning". en. In: *Proceedings of the nineteenth annual ACM conference on Theory of computing - STOC '87*. New York, New York, United States: ACM Press, 1987, pp. 56–65.

[CN18] T. M. Chan and Y. Nekrich. "Towards an Optimal Method for Dynamic Planar Point Location". In: *SIAM Journal on Computing* 47.6 (Jan. 2018). Publisher: Society for Industrial and Applied Mathematics, pp. 2337–2361.

[DDP19] D. Delling, J. Dibbelt, and T. Pajor. "Fast and Exact Public Transit Routing with Restricted Pareto Sets". In: *2019 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*. Proceedings. Society for Industrial and Applied Mathematics, Jan. 2019, pp. 54–65.

[DE96] M. T. Dickerson and D. Eppstein. "Algorithms for proximity problems in higher dimensions". en. In: *Computational Geometry* 5.5 (Jan. 1996), pp. 277–291.

[DGJ] C. Demetrescu, A. Goldberg, and D. Johnson. *9th DIMACS Implementation Challenge: Shortest Paths*.

[DGJ09] C. Demetrescu, A. Goldberg, and D. Johnson, eds. *The Shortest Path Problem*. en. Vol. 74. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. Providence, Rhode Island: American Mathematical Society, July 2009.

[Dib+13] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner. “Intriguingly Simple and Fast Transit Routing”. en. In: *Experimental Algorithms*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, June 2013, pp. 43–54.

[Dij59] E. W. Dijkstra. “A note on two problems in connexion with graphs”. en. In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271.

[DMT92] O. Devillers, S. Meiser, and M. Teillaud. “Fully dynamic delaunay triangulation in logarithmic expected time per operation”. en. In: *Computational Geometry* 2.2 (Oct. 1992), pp. 55–80.

[DPW14] D. Delling, T. Pajor, and R. F. Werneck. “Round-Based Public Transit Routing”. In: *Transportation Science* 49.3 (Oct. 2014), pp. 591–604.

[DSW14] J. Dibbelt, B. Strasser, and D. Wagner. “Customizable Contraction Hierarchies”. en. In: *Experimental Algorithms*. Lecture Notes in Computer Science. Springer, Cham, June 2014, pp. 271–282.

[DW09] D. Delling and D. Wagner. “Pareto Paths with SHARC”. en. In: *Experimental Algorithms*. Ed. by J. Vahrenhold. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 125–136.

[Ehr05] M. Ehrgott. *Multicriteria optimization*. en. 2nd ed. Berlin ; New York: Springer, 2005.

[Erd63] P Erdös. “On Schütte problem”. In: *Math. Gaz* 47 (1963), pp. 220–222.

[Fre+86] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. “The pairing heap: A new form of self-adjusting heap”. en. In: *Algorithmica* 1.1 (Nov. 1986), pp. 111–129.

[Fun+15] S. Funke, T. Malamatos, D. Matijevic, and N. Wolpert. “Conic nearest neighbor queries and approximate Voronoi diagrams”. en. In: *Computational Geometry* 48.2 (Feb. 2015), pp. 76–86.

[GBT84] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. “Scaling and related techniques for geometry problems”. In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. STOC ’84. New York, NY, USA: Association for Computing Machinery, Dec. 1984, pp. 135–143.

[Gei+12] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. “Exact Routing in Large Road Networks Using Contraction Hierarchies”. In: *Transportation Science* 46.3 (Apr. 2012). Publisher: INFORMS, pp. 388–404.

[GH05] A. V. Goldberg and C. Harrelson. “Computing the shortest path: A search meets graph theory.” In: *SODA*. Vol. 5. Citeseer, 2005, pp. 156–165.

[GKS92] L. J. Guibas, D. E. Knuth, and M. Sharir. “Randomized incremental construction of Delaunay and Voronoi diagrams”. en. In: *Algorithmica* 7.1 (June 1992), pp. 381–413.

[GM01] F. Guerriero and R. Musmanno. “Label Correcting Methods to Solve Multicriteria Shortest Path Problems”. en. In: *Journal of Optimization Theory and Applications* 111.3 (Dec. 2001), pp. 589–613.

[GS83] L. J. Guibas and J. Stolfi. “Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams”. en. In: *Proceedings of the fifteenth annual ACM symposium on Theory of computing - STOC ’83*. Not Known: ACM Press, 1983, pp. 221–234.

[Han80] P. Hansen. “Bicriterion Path Problems”. en. In: *Multiple Criteria Decision Making Theory and Application*. Ed. by G. Fandel and T. Gal. Lecture Notes in Economics and Mathematical Systems. Springer Berlin Heidelberg, 1980, pp. 109–127.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (July 1968). Conference Name: IEEE Transactions on Systems Science and Cybernetics, pp. 100–107.

[Hrn+17] J. Hrnčíř, P. Žilecký, Q. Song, and M. Jakob. “Practical Multicriteria Urban Bicycle Routing”. In: *IEEE Transactions on Intelligent Transportation Systems* 18.3 (Mar. 2017), pp. 493–504.

[Ind04] P. Indyk. “Nearest Neighbors in High-Dimensional Spaces”. en. In: *Handbook of Discrete and Computational Geometry, Second Edition*. Ed. by J. Goodman and J. O’Rourke. Vol. 20042571. Series Title: Discrete Mathematics and Its Applications. Chapman and Hall/CRC, Apr. 2004.

[Kap+20] H. Kaplan et al. “Dynamic Planar Voronoi Diagrams for General Distance Functions and Their Algorithmic Applications”. en. In: *Discrete & Computational Geometry* 64.3 (Oct. 2020), pp. 838–904.

[Kir83] D. Kirkpatrick. “Optimal Search in Planar Subdivisions”. In: *SIAM Journal on Computing* 12.1 (Feb. 1983). Publisher: Society for Industrial and Applied Mathematics, pp. 28–35.

[KLP75] H. T. Kung, F. Luccio, and F. P. Preparata. “On Finding the Maxima of a Set of Vectors”. en. In: *Journal of the ACM* 22.4 (Oct. 1975), pp. 469–476.

[KMS91] T. Kao, D. M. Mount, and A. Saalfeld. “Dynamic Maintenance of Delaunay Triangulations”. en. In: *Proceedings Auto-Carto* 9.32 (1991), p. 15.

[KS85] D. G. Kirkpatrick and R. Seidel. “Output-size sensitive algorithms for finding maximal vectors”. en. In: *Proceedings of the first annual symposium on Computational geometry - SCG ’85*. Baltimore, Maryland, United States: ACM Press, 1985, pp. 89–96.

[KV17] A. Kosowski and L. Viennot. “Beyond Highway Dimension: Small Distance Labels Using Tree Skeletons”. In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. Proceedings. Society for Industrial and Applied Mathematics, Jan. 2017, pp. 1462–1478.

[Mar84] E. Q. V. Martins. “On a multicriteria shortest path problem”. In: *European Journal of Operational Research* 16.2 (May 1984), pp. 236–245.

[MDLC10] L. Mandow and J. L. P. De La Cruz. “Multiobjective A* search with consistent heuristics”. en. In: *Journal of the ACM* 57.5 (June 2010), pp. 1–25.

[MHW06] M. Müller-Hannemann and K. Weihe. “On the cardinality of the Pareto set in bicriteria shortest path problems”. en. In: *Annals of Operations Research* 147.1 (Oct. 2006), pp. 269–286.

[MM12] E. Machuca and L. Mandow. “Multiobjective heuristic search in road maps”. en. In: *Expert Systems with Applications* 39.7 (June 2012), pp. 6435–6445.

[Mor06] C. W. Mortensen. “Fully Dynamic Orthogonal Range Reporting on RAM”. In: *SIAM Journal on Computing* 35.6 (Jan. 2006). Publisher: Society for Industrial and Applied Mathematics, pp. 1494–1525.

[MSZ99] E. P. Mücke, I. Saias, and B. Zhu. “Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations”. en. In: *Computational Geometry* (1999), p. 21.

[NS07] G. Narasimhan and M. Smid. *Geometric Spanner Networks*. en. Google-Books-ID: SY4cNZWc4GAC. Cambridge University Press, Jan. 2007.

[Paj09] T. Pajor. “Multi-Modal Route Planning”. MA thesis. Universität Karlsruhe (TH), Mar. 2009.

[PC89] L. Paul Chew. “There are planar graphs almost as good as the complete graph”. en. In: *Journal of Computer and System Sciences* 39.2 (Oct. 1989), pp. 205–219.

[PS13] J. M. Paixão and J. L. Santos. “Labeling Methods for the General Case of the Multi-objective Shortest Path Problem – A Computational Study”. en. In: *Computational Intelligence and Decision Making*. Ed. by A. Madureira, C. Reis, and V. Marques. Intelligent Systems, Control and Automation: Science and Engineering. Springer Netherlands, 2013, pp. 489–502.

[PV19] D.-M. Phan and L. Viennot. “Fast Public Transit Routing with Unrestricted Walking Through Hub Labeling”. en. In: *Analysis of Experimental Algorithms*. Ed. by I. Kotsireas et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 237–247.

[PY00] C. H. Papadimitriou and M. Yannakakis. “On the approximability of trade-offs and optimal access of Web sources”. In: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. 2000, pp. 86–92.

[Rah+15] Z. Rahmati et al. “A simple, faster method for kinetic proximity problems”. en. In: *Computational Geometry* 48.4 (May 2015), pp. 342–359.

[Rah+19] Z. Rahmati, M. A. Abam, V. King, and S. Whitesides. “Kinetic k-Semi-Yao graph and its applications”. en. In: *Computational Geometry*. Canadian Conference on Computational Geometry 77 (Mar. 2019), pp. 10–26.

[RE09] A. Raith and M. Ehrgott. “A comparison of solution strategies for biobjective shortest path problems”. en. In: *Computers & Operations Research* 36.4 (Apr. 2009), pp. 1299–1331.

[Roy59] B. Roy. “Transitivité et connexité”. In: *Comptes Rendus Hebdomadaires Des Seances De L Academie Des Sciences* 249.2 (1959). Publisher: GAUTHIER-VILLARS/EDITIONS ELSEVIER 23 RUE LINOIS, 75015 PARIS, FRANCE, pp. 216–218.

[Shi+17] N. Shi et al. “The multi-criteria constrained shortest path problem”. In: *Transportation Research Part E: Logistics and Transportation Review* 101 (May 2017), pp. 13–29.

[SJS15] M. Shekelyan, G. Josse, and M. Schubert. “Linear path skylines in multicriteria networks”. en. In: *2015 IEEE 31st International Conference on Data Engineering*. Seoul, South Korea: IEEE, Apr. 2015, pp. 459–470.

[SMH04] G. Schaller and M. Meyer-Hermann. “Kinetic and dynamic Delaunay tetrahedralizations in three dimensions”. en. In: *Computer Physics Communications* 162.1 (Sept. 2004), pp. 9–23.

[SP02] Sungwon Jung and S. Pramanik. “An efficient path computation model for hierarchically structured topographical road maps”. In: *IEEE Transactions on Knowledge and Data Engineering* 14.5 (Sept. 2002). Conference Name: IEEE Transactions on Knowledge and Data Engineering, pp. 1029–1046.

[SSB05] G. Simon, M. Steiner, and E. Biersack. “Distributed Dynamic Delaunay Triangulation in d-Dimensional Spaces”. en. In: *Institut Eurocom Tech. Rep.* (2005), p. 12.

[TOG17] C. D. Toth, J. O’Rourke, and J. E. Goodman. *Handbook of Discrete and Computational Geometry*. en. Google-Books-ID: 9mlQDwAAQBAJ. CRC Press, Nov. 2017.

[Tro+21] T. Trolliet et al. “Interest Clustering Coefficient: A New Metric for Directed Networks Like Twitter”. en. In: *Complex Networks & Their Applications IX*. Ed. by R. M. Benito et al. Studies in Computational Intelligence. Cham: Springer International Publishing, 2021, pp. 597–609.

[TTLC92] C. Tung Tung and K. Lin Chew. “A multicriteria Pareto-optimal path algorithm”. In: *European Journal of Operational Research* 62.2 (Oct. 1992), pp. 203–209.

[TZ09] G. Tsaggouris and C. Zaroliagis. “Multiobjective Optimization: Improved FPTAS for Shortest Paths and Non-Linear Objectives with Applications”. en. In: *Theory of Computing Systems* 45.1 (June 2009), pp. 162–186.

[VV78] D. Van Vliet. “Improved shortest path algorithms for transport networks”. en. In: *Transportation Research* 12.1 (Feb. 1978), pp. 7–20.

[Wan+16] S. Wang, X. Xiao, Y. Yang, and W. Lin. “Effective indexing for approximate constrained shortest path queries on large road networks”. In: *Proceedings of the VLDB Endowment* 10.2 (Oct. 2016), pp. 61–72.

[War87] A. Warburton. “Approximation of Pareto Optima in Multiple-Objective, Shortest-Path Problems”. en. In: *Operations Research* 35.1 (Feb. 1987), pp. 70–79.

[Wik21] Wikipedia contributors. *Combinatorial map — Wikipedia, The Free Encyclopedia*. 2021.

[Wu+16] H. Wu et al. “Efficient Algorithms for Temporal Path Computation”. In: *IEEE Transactions on Knowledge and Data Engineering* 28.11 (Nov. 2016), pp. 2927–2942.

[WZ17] D. Wagner and T. Zündorf. “Public Transit Routing with Unrestricted Walking”. In: *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*. Ed. by G. D’Angelo and T. Dollevoet. Vol. 59. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 7:1–7:14.

[Xia11] G. Xia. “Improved upper bound on the stretch factor of delaunay triangulations”. In: *Proceedings of the twenty-seventh annual symposium on Computational geometry*. SoCG ’11. New York, NY, USA: Association for Computing Machinery, June 2011, pp. 264–273.

[Yen71] J. Y. Yen. “Finding the K Shortest Loopless Paths in a Network”. en. In: *Management Science* 17.11, (1971), pp. 712–716.

[ZT87] D. J. Zawack and G. L. Thompson. “A Dynamic Space-Time Network Flow Model for City Traffic Congestion”. In: *Transportation Science* 21.3 (Aug. 1987). Publisher: INFORMS, pp. 153–162.

Appendix A

Data structures

In this section, we give some insight on classic data structures used in the document.

A.1 Linked lists

Let L be a list. In fact, we handle a cursor on the first cell of L, which represents the list. We will not make the difference between a list and the associated cursor. We enumerate the primitives used in Section 7.2:

- `value`(L): returns the content of the first cell of L,
- `next`(L): returns the cursor on the cell following the first one of L, *null* if there is none,
- `isEmpty`(L): returns if L is empty or not,
- `createList`(): return an empty list,
- `insertBegin`(L, x): returns L with x inserted at its beginning,
- `deleteBegin`(L): returns L with x deleted at its beginning,
- `replaceBegin`(L, x): returns L after replacing the element at the beginning by x,
- `insertAfter`(L, L', x): returns L with x inserted after the cursor L',
- `deleteAfter`(L, L'): returns L after having deleted the element after the cursor L',
- `replaceAfter`(L, L', x): returns L with x replacing the element after the cursor L'.

A.2 Balanced binary trees

A *binary search tree* is a tree such that each node has at most two children. To each node is associated a key in a totally ordered set and all the keys of its left (resp. right) subtree are smaller (resp. greater). These trees support the three following operations: insertion, deletion and search. We use these trees by performing sequences of these operations.

If a binary tree contains n nodes, then its height is in $\Omega(\log n)$. Whenever it is close to $\log n$, we say that the tree is *balanced*, but it is not always the case. Given a balanced binary tree, inserting or deleting elements can unbalance it: the height can reach $n-1$. Some structures allow for efficient rebalancing when this happens: *self-balanced binary trees*. We recall two structures that allow this:

Self-balanced binary trees. We recall the definitions of two classic self-balanced binary trees.

- A *red-black tree* is a binary search tree such that a color a given to each node: red or black. The leaves are black, and any child of a red node is black. Furthermore, all paths from the root to the leaves have the same number of black nodes.

- An *AVL tree* is a binary search tree such that, for any node, the heights of its two subtrees differ at most by 1.

For both these trees, a rotation system guarantees that insertions, deletions and searches are in $O(\log n)$ time.

A.3 Priority queue

A *priority queue* is a data structure containing pairs (k, e). k is called a key and e an element. The elements are in a totally ordered set. Let P be a priority queue. We list the primitives used in this manuscript:

- `min`(P): returns the pair from P having a minimum element,

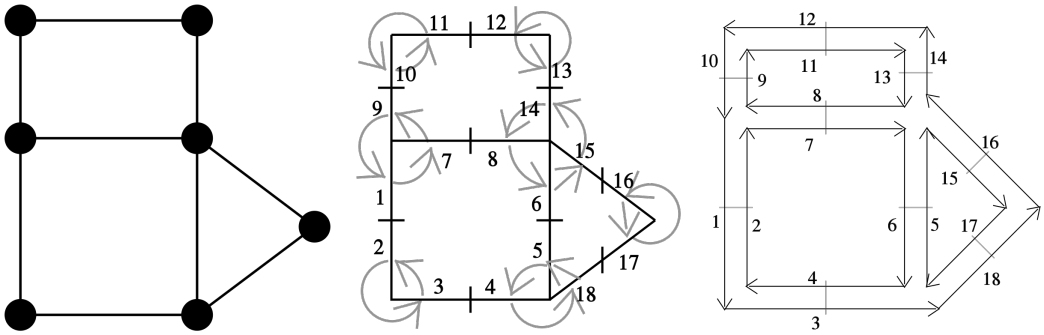
- `pop`(P): returns the priority queue P without the pair having a minimum element,

- `insert`(P, k, v): inserts in P the pair (k, v),

- `decreaseKey`(P, k, v): if P contains a pair (k, v') with $v < v'$, then it replaces it by the pair (k, v).

Notice that the definition of `decreaseKey` is implicitly based on the unicity of each key in a priority queue. This primitive is used only in this context.

There are two types of priority queues in which we are interested in.



(a) Graph represented. (b) Permutation σ. (c) Involution α.

Figure A.1 – Combinatorial map (D, σ, α) [Wik21] CC BY SA.

Binary Heap. A *binary heap* is a binary tree such that a node key is greater than any of those in its subtree. Simple implementations can use references, but also with arrays. For the latter, if u has an index p_u, and l_u, r_u its left and right children, then $p_{l_u} = 2 \cdot p_u$ and $p_{r_u} = 2 \cdot p_u + 1$. Deletion can unbalance the heap, but on insertion, one can choose on which side to insert, rebalancing it.

Strict Fibonacci Heap. This data structure is similar to a binary heap. The elements are partitioned in several heaps instead of one. Mergers are done in a lazy manner, which gives constant time for insertions.

Differences. For a priority queue containing n elements, the primitives have the following complexities:

Heap type	`min`	`pop`	`insert`	`decrease_key`
Binary	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Strict Fibonacci	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$

Thus, a strict Fibonacci heap is better in theory than a binary heap, thanks to its constant time `insert` and `decrease_key` primitives. However, in practice, strict Fibonacci heaps are heavy to implement and less efficient than binary heaps.

A.4 Combinatorial map

The planar graphs from Part II, and especially the bicolored graph in Section 6.2, can be implemented using combinatorial map in order to easily access geometric information without having to rely (too much) on the points' coordinates.

A *combinatorial map* is a triplet (D, σ, α) where D is a finite set of darts (half-edges), σ is a permutation on D providing the cyclic order around the points, and α is an involution on D with no fixed point merging the half-edges into edges. This structure is illustrated in Figure A.1.

The permutations σ and α can be implemented with linked list. Given an arc, the neighbouring arcs can be found in constant time. Furthermore, the insertion of

an arc is in constant time if an arc of the face containing the arc to insert is given. Deletions are also in $O(1)$.

Appendix B

Data

Graph	Vertices	Arcs	MC Dijkstra Time	Frame Time	S_u	$S_{u,1}$(Frame)
DC	9559	14909	79.34	76.02	4.84	4.22
DE	49109	60512	305.83	293.21	7.27	6.02
RI	53658	69213	154.78	148.49	5.24	4.37
HI	64892	76809	32.49	35.93	2.99	2.82
AK	69082	78100	14.17	14.26	1.37	1.31
VT	97975	107558	1599.46	1321.96	39.61	29.06
NH	116920	133415	201.93	180.75	11.55	8.82
CT	153011	187318	471.70	458.65	7.28	5.99
ME	194505	214921	760.20	615.78	17.18	12.53
ND	210801	260902	675.34	500.23	18.59	12.02
SD	212313	259622	2611.50	2579.06	40.07	33.05
UT	248730	295763	93.54	98.04	4.21	3.86
WY	253077	304014	309.18	253.28	7.73	5.31
NV	261155	311043	289.63	295.83	7.95	6.84
MD	265912	317624	340.41	320.45	7.15	5.62
ID	271450	318761	390.80	346.06	13.31	9.42
WV	300146	328858	201.60	200.04	7.78	6.86
NE	308157	392008	3378.26	2723.28	36.19	27.57
MA	308401	385164	706.77	536.26	8.80	6.11
MT	317905	360936	84.38	88.15	4.78	4.23
NJ	330386	436036	282.78	281.44	4.72	3.90
IA	390002	502269	34731.65	25906.71	127.25	94.56
MS	413250	483306	100483.23	67887.91	176.43	136.16
LA	413574	499254	3864.74	2274.64	47.19	28.51
CO	448253	539295	489.56	493.39	12.39	11.05
SC	463652	553599	20692.42	14529.97	72.83	53.36
NM	467529	567084	1333.50	1209.93	22.09	14.92
KY	467967	525995	478.43	417.53	12.85	10.61
KS	474015	474015	6554.61	6156.66	47.44	36.90
AR	483175	563036	5916.52	4243.27	48.80	37.61
IN	497458	629750	113260.17	71255.94	204.85	143.67
WI	519157	635436	11995.69	8778.60	88.20	62.48
OR	536236	628167	549.63	461.76	12.53	8.67
OK	540981	664215	1412.45	1368.92	37.43	30.73
AZ	545111	665827	300.96	308.97	3.27	2.90
MN	547028	670443	21121.55	13962.90	66.25	47.00
AL	566843	661487	9187.04	7703.13	101.26	84.04
WA	575860	675049	158.23	160.11	3.81	3.43
TN	583484	676080	894.40	767.86	15.26	12.33
VA	630639	714809	10943.98	7475.28	62.87	48.84
MI	673534	845087	13129.40	10230.36	82.53	63.56
MO	675407	807892	2927.27	2276.36	29.98	21.94
OH	676058	842872	27250.71	19097.33	122.39	86.71
NY	716215	897451	5074.74	3973.49	49.79	36.90
GA	738879	869890	111154.91	74385.28	231.85	171.93
PA	874843	1088296	9745.94	6892.3	78.16	55.42
NC	887630	1009846	25206.34	17637.98	66.78	49.93
FL	1048506	1330551	5011.51	3962.47	36.52	26.15
CA	1613325	1989149	3630.50	2995.51	23.65	18.52

Table B.1 – MC Dijkstra vs Frame on USA states from DIMACS