

ECOLE POLYTECHNIQUE DE L'UNIVERSITÉ FRANÇOIS RABELAIS DE TOURS

Département Informatique

64 avenue Jean Portalis

37200 Tours, France

Tél. +33 (0)2 47 36 14 14

polytech.univ-tours.fr

Projet Recherche & Développement

2019-2020

Parallélisation de la recherche de plus court chemin multi-objectif

**POLYTECH[®]**
TOURS

Tuteur académique
Emmanuel NERON

Étudiant
Simon MOULARD (DI5)

12 avril 2020



Liste des intervenants

Nom	Email	Qualité
Simon MOULARD	simon.moulard@laposte.net	Étudiant DI5
Emmanuel NERON	emmanuel.neron@univ-tours.fr	Tuteur académique, Département Informatique



Avertissement

Ce document a été rédigé par Simon MOULARD susnommé l'auteur.

L'Ecole Polytechnique de l'Université François Rabelais de Tours est représentée par Emmanuel NERON susnommé le tuteur académique.

Par l'utilisation de ce modèle de document, l'ensemble des intervenants du projet acceptent les conditions définies ci-après.

L'auteur reconnaît assumer l'entière responsabilité du contenu du document ainsi que toutes suites judiciaires qui pourraient en découler du fait du non respect des lois ou des droits d'auteur.

L'auteur atteste que les propos du document sont sincères et assument l'entière responsabilité de la véracité des propos.

L'auteur atteste ne pas s'approprier le travail d'autrui et que le document ne contient aucun plagiat.

L'auteur atteste que le document ne contient aucun propos diffamatoire ou condamnable devant la loi.

L'auteur reconnaît qu'il ne peut diffuser ce document en partie ou en intégralité sous quelque forme que ce soit sans l'accord préalable du tuteur académique et de l'entreprise.

L'auteur autorise l'école polytechnique de l'université François Rabelais de Tours à diffuser tout ou partie de ce document, sous quelque forme que ce soit, y compris après transformation en citant la source. Cette diffusion devra se faire gracieusement et être accompagnée du présent avertissement.



Pour citer ce document

Simon MOULARD, *Parallélisation de la recherche de plus court chemin multi-objectif*, Projet Recherche & Développement, Ecole Polytechnique de l'Université François Rabelais de Tours, Tours, France, 2019-2020.

```
@mastersthesis{
  author={MOULARD, Simon},
  title={Parallélisation de la recherche de plus court chemin multi-objectif},
  type={Projet Recherche \& Développement},
  school={Ecole Polytechnique de l'Université François Rabelais de Tours},
  address={Tours, France},
  year={2019-2020}
}
```

Table des matières

Liste des intervenants	a
Avertissement	b
Pour citer ce document	c
Table des matières	i
Table des figures	vi
1 Introduction	1
1 Objectifs	1
2 Base méthodologique	1
2 Description générale	3
1 Fonctionnalités du système	3
2 Utilisation	3
3 Entrées	3
4 Fichiers d'entrée existants	4
5 Sorties	4
6 Méthode utilisée	4
6.1 Première phase : prétraitement	5
6.2 Seconde phase : phase principale	5
6.3 Exploitation des résultats de la première phase lors de la seconde	5
7 Structure générale du système	6
3 Etat de l'art	7
1 Recherche parallèle de plus court chemin mono-objectif	7

1.1	L'algorithme de Δ -Stepping	7
1.2	La parallélisation de l'algorithme de Dijkstra	7
2	Définitions pour le problème de recherche de plus court chemin multi-objectif	8
2.1	Domination et front de Pareto.....	8
3	Recherche parallèle de plus court chemin multi-objectif.....	9
4	Analyse et conception	10
1	Adaptation de l'algorithme de <i>A parallelization of Dijkstra's shortest path algorithm</i> [2]	10
1.1	Adaptation des critères.....	10
1.2	Multitude d'étiquettes par nœud	11
1.3	Boucle principale des stratégies parallèles et condition d'arrêt	11
2	Patron de conception Stratégie.....	12
3	Choix de l'architecture pour la parallélisation	12
5	Mise en œuvre	14
1	Outils utilisés.....	14
2	Librairies utilisées	14
2.1	Boost.....	14
2.2	Osmium	15
2.3	OpenMP	15
2.4	Rigport	15
3	Implémentation	15
3.1	Implémentation du patron de conception Stratégie	15
3.2	Stratégies.....	16
3.2.1	SndPhaseSequentiel.....	16
3.2.2	SndPhaseSequentielT	16
3.2.3	SPPT3md	16
3.2.4	SPPT4asyncmd	17
3.2.5	Limites des méthodes parallèles implémentées	18
3.3	Structures pour les files d'exploration et les listes d'étiquettes par nœud ...	18
3.3.1	LabelArray	18
3.3.2	ExplorationArray	18
3.4	Limites	19
4	Qualité et performances	19
4.1	Tests unitaires	19
4.2	Lancement de campagnes de résolution d'instances	19
4.2.1	Vérification de la correction des stratégies	19
4.2.2	Mesure des temps de résolution des instances	20

6	Bilan et conclusion	21
1	Fait et Reste à faire	21
2	Aléas.....	22
3	Planning du second semestre	22
4	Qualité	22
5	Gestion de projet	23
	Annexes	24
A	Spécifications fonctionnelles de l'existant	25
1	Constantes globales	25
2	Les sorties	25
2.1	Sortie standard.....	25
2.2	Sortie standard d'erreur.....	26
2.3	Sous-dossier des résultats.....	26
2.4	Fichier results.csv	26
2.5	Fichiers results_[identifiant de l'instance].csv.....	26
3	Fichiers d'entrée.....	26
4	La classe Parser	27
4.1	void Parser::parseVerticesFile(std::string const &).....	27
4.2	void Parser::parseEdgesFile(std::string const &).....	27
4.3	void Parser::parseIntancesFile(std::string const &).....	27
5	La classe Graph.....	27
5.1	void Graph::setVertices(std::vector<Vertex>).....	28
5.2	void Graph::setEdges(std::vector<Edge>)	28
5.3	void Graph::printData() const	28
6	La classe LSDPF	28
6.1	void LSDPF::runFirstPhase()	28
6.2	void LSDPF::runSecondPhase().....	29
6.3	vertex_label_list_t const & LSDPF::getParetoFront() const.....	29
6.4	void LSDPF::printStatistics() const	29
6.5	void LSDPF::printStatistics(int64_t) const.....	29
7	La classe Writer.....	29
7.1	Writer::Writer(std::string const &)	29
7.2	void Writer::writeResults(Instance const &, LSDPF const *, double, double)	30
7.3	void Writer::writeParetoFrontFile(Instance const&, LSDPF const*)	30

B	Spécifications fonctionnelles des modifications apportées à l'existant	31
1	Modification de la définition de la classe Label	32
2	Modification de la définition de la classe LSDPF	32
3	Définition de la méthode LSDPF::runSecondPhase<T>	33
4	Définition de l'interface SndPhase.....	33
C	Spécifications fonctionnelles concernant les nouvelles stratégies	35
1	SndPhaseSequentiel.....	35
2	SndPhaseSequentielT	35
3	Structures utiles pour les stratégies parallèles	35
3.1	Définition de Buffer98T	35
3.2	LabelArray	36
3.3	LabelPosition.....	36
3.4	ExplorationArray	36
3.5	Description des classes de préfixe Worker.....	37
4	Stratégie parallèle : SPPT3md	38
4.1	Spécificités du worker associé à SPPT3md	39
5	Stratégie parallèle : SPPT4asyncmd.....	39
5.1	Spécificités du worker associé à SPPT4asyncmd	39
5.2	StaticCriteriaValuesContainer	40
D	Spécifications non fonctionnelles	42
1	Contraintes de développement et conception	42
2	Contraintes de fonctionnement et d'exploitation	42
2.1	Modes de fonctionnement	42
2.2	Contrôlabilité.....	42
E	Guide d'installation	43
1	Environnement	43
2	Librairies.....	43
2.1	Boost.....	43
2.2	Osmium	43
2.3	Rigport::MPMCQueue	44
3	Doxygen	44
4	Python.....	44
F	Guide d'utilisation	45
1	Utilisation courante	45
1.1	Configuration.....	45
1.1.1	Stratégie séquentielle originelle	46

1.1.2	Stratégie séquentielle avec template	46
1.1.3	Stratégie parallèle avec thread dédié suivant le cycle	46
1.1.4	Stratégie parallèle avec thread dédié en dehors du cycle	46
1.2	Lancement du programme	47
1.3	Sortie du programme	47
2	Benchmark	47
3	Tests unitaires	48
3.1	Exécution des tests	48
3.2	Ajouter un test unitaire	48
G	Dossiers de tests	49
1	Tests unitaires	49
2	Vérification de la correction des stratégies	49
3	Mesure des temps de résolution	50
3.1	Stratégies séquentielles	50
3.2	Stratégies parallèles	50
4	Résultats	51
4.1	Comparaison des versions séquentielles	51
4.2	Comparaison des versions parallèles à la version séquentielle de référence	53
4.2.1	Limites	55
	Bibliographie	56



Table des figures

B Spécifications fonctionnelles des modifications apportées à l'existant

1	Classes en relation avec LSDPF, dans l'existant.....	31
2	Classes en relation avec LSDPF, dans la proposition.....	32

1

Introduction

La recherche du plus court chemin est un problème d'optimisation classique, dont de nombreuses variantes sont étudiées, comme le problème du plus court chemin multi-objectif. Ce problème a de nombreuses applications dans le calcul d'itinéraires.

Il devient de plus en plus difficile de trouver des améliorations aux méthodes de résolution de ce problème, par conséquent on se tourne vers la parallélisation pour réduire les temps de calcul.

Ce projet de conception et d'implémentation d'une méthode parallèle exacte pour ce problème est réalisé par Simon Moulard et encadré par Emmanuel Néron.

Une méthode exacte séquentielle écrite par Antoine Giret est décrite par Yannick Kergosien dans [4], et sera désignée en tant qu'existant.

1 Objectifs

L'objectif de ce projet est d'adapter une méthode séquentielle existante qui détermine exactement les plus courts chemins multi-objectifs entre deux nœuds d'un graphe orienté, en la rendant parallèle. La méthode séquentielle est divisée en deux phases, et les efforts seront concentrés sur la parallélisation de la deuxième phase, car la première est un prétraitement qui est déjà parallélisée.

A cette fin, on s'inspirera de méthodes parallèles pour la résolution du problème de plus court chemin mono-objectif disponibles dans la littérature, et on tentera de les adapter au problème multi-objectif.

On envisagera l'exploitation des positions géographiques des nœuds dans la conception d'une méthode parallèle pour les graphes représentant des réseaux routiers réels.

2 Base méthodologique

Pour l'évolution du programme existant, on utilisera le langage C++, puisque c'est le langage dans lequel l'existant est écrit. On utilisera la bibliothèque Boost, puisqu'elle est déjà partiellement utilisée par l'existant. On s'autorisera à utiliser OpenMP, MPI ou cuda pour réaliser la

partie parallèle. Il ne s'agit pas là d'une contrainte. Le choix de cuda demandera néanmoins de se procurer une machine avec un GPU compatible. On utilisera Git en tant que logiciel de gestion de versions.

On utilise GanttProject, afin de réaliser des diagrammes de Gantt. On utilise Umbrello, Paint.net ou AlgoUML pour la représentation des diagrammes UML.

2

Description générale

1 Fonctionnalités du système

Le programme existant permet de résoudre le problème de plus court chemin multi-objectif entre une source et un puits, c'est-à-dire de calculer l'ensemble des chemins optimaux de la source vers le puits, qu'on appelle front de Pareto.

Le programme résout le problème pour chaque paire de source et de puits qui lui est fournie.

2 Utilisation

Le nombre d'objectifs est un paramètre défini au moment de la compilation, donc une version donnée du programme ne sera capable de traiter que les problèmes ayant ce nombre d'objectifs. Le programme ne compare pas le nombre d'objectifs pour lequel il a été compilé et le nombre d'objectifs du problème donné en entrée.

Le programme est appelé depuis une console, et n'a pas d'IHM. Toutes les informations nécessaires pour la réalisation du traitement sont données par l'utilisateur dans la commande de lancement du programme. Les paramètres sont les suivants :

- Nom du fichier définissant les nœuds;
- Nom du fichier définissant les arcs;
- Nom du fichier définissant les instances;
- Nom du dossier dans lequel doivent être écrits les résultats;
- (Entier) paramètre `ITERATION_LABEL_MAJ_DYN_PF` : permet de faire varier la fréquence de mise à jour dynamique du front de Pareto.

3 Entrées

Les données du problème sont les suivantes :

- k , le nombre d'objectifs;
- V , un ensemble de nœuds;
- E inclus dans $V \times V$, un ensemble d'arcs orientés;

- une fonction de coût $c : E \rightarrow \mathbb{R}_+^k$ qui donne le poids multi-dimensionnel de chaque arc ;
- des paires de nœuds (source, puits) avec source et puits dans V . Ces paires sont appelées instances. Il y a autant de problèmes à résoudre que d'instances, et autant d'appels de la méthode de résolution.

k est connu au moment de la compilation du programme.

Le reste de ces données est contenu dans des fichiers dont le chemin d'accès est défini par l'utilisateur dans la commande de lancement du programme. Ces fichiers sont au format csv, avec pour caractère de séparation la tabulation.

Le fichier dédié à V contient un nœud par ligne, et sur chaque ligne se trouve un identifiant, une latitude et une longitude.

Le fichier dédié à E et c contient un arc par ligne, et sur chaque ligne se trouve l'identifiant du nœud de départ, l'identifiant du nœud d'arrivée, puis chaque composante du poids de l'arc. On ne vérifie pas que le nombre de composantes est bien égal à k , l'utilisateur en est responsable.

Le fichier dédié aux instances contient une instance par ligne, et sur chaque ligne se trouve un identifiant de l'instance, suivi de l'identifiant de la source, puis de celui du puits.

4 Fichiers d'entrée existants

Des fichiers d'instances, de nœuds et d'arcs ont été fournis avec l'existant. Ils serviront pour la vérification et la mesure du temps pris par les différentes versions de la seconde phase. La liste des graphes fournis est disponible en annexe.

De nombreux graphes fournis représentent des réseaux routiers réels, et ils contiennent les coordonnées géographiques des nœuds.

5 Sorties

Le programme écrit dans plusieurs flux : dans la sortie standard, les nombres de nœuds et d'arcs du graphe traités sont affichés, puis quelques résultats de la résolution de chaque instance ; dans la sortie standard d'erreur, si une exception est levée, le message associé à l'erreur est affiché.

Le programme écrit aussi dans des fichiers :

- un sous-dossier est créé pour contenir tous les fichiers de résultats ;
- le fichier `results.csv` est créé, qui contient les paramètres de l'algorithme qui ne font pas partie du problème : la stratégie d'exploration, `ITERATION_LABEL_MAJ_DYN_PF`, les barycentres utilisés par les deux phases ; `results.csv` contient également une synthèse des résultats pour chaque instance ;
- `results_[identifiant de l'instance, avec un 0 devant s'il est plus petit que 10].csv`, pour chaque instance, qui contient le front de Pareto.

Plus de détails sont disponibles dans la partie "Sortie des spécifications fonctionnelles de l'existant".

6 Méthode utilisée

L'existant implémente une méthode à deux phases afin de résoudre le problème de plus court chemin multi-objectif d'une source vers un puits : une phase de prétraitement, où des recherches de chemins mono-objectifs du puits vers la source ont lieu, puis une phase principale, où un plus grand nombre de chemins multi-objectifs sont découverts, et où le front de Pareto est mis à jour dynamiquement.

6.1 Première phase : prétraitement

La liste des barycentres à coefficients positifs utilisée lors du prétraitement est connue à la compilation. Ces barycentres servent à définir la combinaison linéaire à utiliser sur les objectifs pour passer d'un problème multi-objectif à un problème mono-objectif. Pour chaque combinaison linéaire, une recherche de plus court chemin mono-objectif est réalisée, du puits vers la source, en remontant le graphe. Cela a pour effet de créer une étiquette par nœud et par combinaison linéaire. L'ensemble de ces étiquettes est représenté sous la forme d'un `std::vector<std::vector<std::vector<double>>>` : la première dimension est la combinaison linéaire, la seconde dimension est le nœud, la troisième dimension est l'objectif.

6.2 Seconde phase : phase principale

La seconde phase est la partie principale de la recherche de plus court chemin multi-objectif. Des étiquettes sont placées dans une file d'exploration. A chaque itération, une étiquette est retirée de la file en suivant une stratégie d'exploration (généralement, on retire la plus petite étiquette, il reste à définir la relation d'ordre). On propage cette étiquette aux nœuds successeurs du nœud associé à l'étiquette. Cela engendre de nouvelles étiquettes associées à ces nœuds. Pour toutes ces étiquettes, on vérifie qu'elles ne sont ni dominées localement, ni dominées au puits (en ajoutant la borne inférieure). Si elles ne sont pas dominées dans ces deux cas, on ajoute ces étiquettes dans la file d'exploration. On purge les nœuds des étiquettes locales qui seraient dominées par les nouvelles étiquettes récemment ajoutées.

6.3 Exploitation des résultats de la première phase lors de la seconde

Chaque étiquette créée pendant le prétraitement représente un chemin du nœud associé jusqu'au puits :

- *Initialisation du front de Pareto* : Parmi ces nœuds, il y a la source. On a donc des chemins de la source au puits. On initialise le front de Pareto avec ces solutions.

A partir des chemins d'un nœud jusqu'au puits, on calcule un chemin idéal. Tous les chemins du nœud jusqu'au puits sont dominés par ce chemin idéal. Ce chemin idéal n'existe généralement pas. On peut s'en servir pour calculer la projection d'une étiquette en sommant le chemin idéal du nœud avec l'étiquette du nœud, c'est-à-dire un chemin réel de la source jusqu'au nœud.

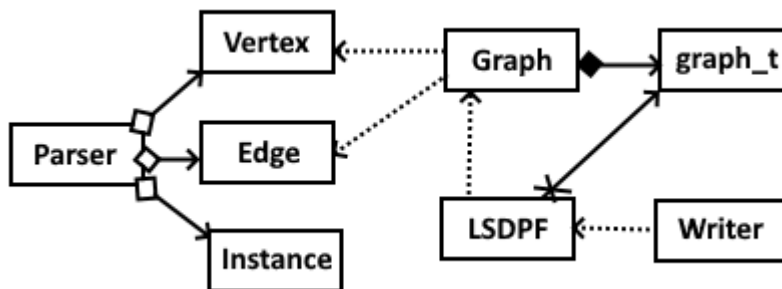
- *Élimination des étiquettes* : Si la projection est dominée par une solution du front de Pareto, alors tous les chemins suivant le sous-chemin de la source au nœud décrit par l'étiquette sont dominés. On peut immédiatement supprimer l'étiquette.
- *Heuristique* : On peut utiliser la projection dans la stratégie d'exploration. Au lieu de retirer la plus petite étiquette de la file d'exploration, on retire l'étiquette de plus petite projection. L'utilisation d'une telle heuristique rend l'algorithme label-correcting au lieu de label-setting.
- *Mise à jour dynamique du front de Pareto* : En sommant une étiquette représentant un chemin de la source à un nœud, avec une étiquette représentant un chemin du nœud au puits calculé durant la première phase, on obtient l'étiquette d'un chemin de la source au puits. On peut alors vérifier si un tel chemin est dominé par une solution du front de Pareto provisoire. Si le chemin n'est pas dominé, on peut l'ajouter au front de Pareto provisoire. Les solutions ajoutées dynamiquement risquent de ne pas être présentes dans le front de Pareto final, car on n'a aucune garantie qu'elles soient non-dominées.

7 Structure générale du système

Le programme est principalement composé :

- d'une classe Parser, qui lit les fichiers de nœuds et d'arcs pour construire un graphe, et qui lit le fichier des instances ; il est composé de listes d'arcs, de nœuds, et d'instances ;
- d'une classe Graph, qui contient une liste d'adjacence de Boost de type renommé graph_t ; la classe Graph alimente la liste d'adjacente avec une liste de nœuds et d'arcs.
- d'une classe LSDPF, qui lit un graphe et des instances et résout le problème de plus court chemin multi-objectif ;
- d'une classe Writer, qui interroge une instance d'LSDPF pour obtenir ses résultats, et qui les écrit dans les différents fichiers de sortie.

Les instances des classes Vertex, Edge et Instance sont immuables.



3

Etat de l'art

1 Recherche parallèle de plus court chemin mono-objectif

1.1 L'algorithme de Δ -Stepping

L'algorithme de Δ -Stepping est décrit par [5]. C'est une variante de l'algorithme de Dijkstra.

L'algorithme de Δ -Stepping ne consiste pas à manipuler une queue de priorité, il consiste à manipuler un tableau B de buckets tel que $B[i]$ contienne les nœuds v tel que la distance de la source à v est non-fixée, et est comprise entre $i * \Delta$ et $(i + 1) * \Delta$.

A chaque itération de la boucle principale, le bucket courant est vidé, tous les arcs dits légers (de poids inférieur à Δ) sont utilisés pour propager les distances provisoires. Ces propagations entraînent l'apparition éventuelle de nouveaux nœuds dans le bucket courant, qu'il faut à nouveau vider. L'utilisation des arcs dits lourds (de poids supérieur à Δ) pour la propagation des distances provisoires ne peut pas insérer de nœuds dans le bucket courant, donc on peut les utiliser chacun une seule fois après la propagation avec les arcs légers.

L'algorithme s'arrête quand tous les buckets sont vides, ou que le puits voit sa distance fixée.

Δ est un paramètre de cette méthode, et a une grande influence sur la vitesse d'exécution et le nombre de buckets à traiter.

Si Δ est infini, le Δ -Stepping est équivalent à l'algorithme de Bellman-Ford.

1.2 La parallélisation de l'algorithme de Dijkstra

Pour paralléliser l'algorithme de Dijkstra, on peut retirer de la file d'exploration plusieurs nœuds à la fois. Néanmoins, faire cela pourrait faire perdre à l'algorithme le statut de label-setting.

On se demande alors comment déterminer l'ensemble des nœuds que l'on peut retirer de la file d'exploration sans perdre le statut de label-setting, c'est-à-dire quels nœuds peuvent avoir leur distance fixée avec l'assurance qu'une plus petite distance ne pourra pas leur être trouvée.

De tels critères sont proposés par [2]. Soit V l'ensemble des nœuds du graphe, F l'ensemble des nœuds de distance non fixé. On calcule en prétraitement $\delta_{out}(u) = \min_{v \in V}(c(u, v))$, $\forall u \in V$; et $\delta_{in}(v) = \min_{u \in V}(c(u, v))$, $\forall v \in V$.

- OUT-critère : on calcule $L = \min_{u \in F}(tent(u) + \delta_{out}(u))$. Tous les nœuds $u \in F$ dont $tent(u) \leq L$ peuvent être retirés de la file d'exploration.
- IN-critère : on calcule $M = \min_{u \in F}(tent(u))$. Tous les nœuds $u \in F$ dont $tent(u) - \delta_{in}(u) \leq M$ peuvent être retirés de la file d'exploration.

L'OUT-critère est redécouvert dans le cadre d'une amélioration de l'algorithme de Δ -Stepping par [1], le radius-Stepping, qui consiste à utiliser l'OUT-critère pour dimensionner le bucket suivant, c'est à dire l'ensemble des nœuds dont la distance doit être fixée, au lieu d'utiliser un pas fixe.

Des critères plus raffinés sont décrits par [3]. Néanmoins, ils demandent plus de ressources pour être calculées.

Dans l'algorithme proposé par [2], les nœuds sont partitionnés entre chaque thread. Chaque thread a une file d'exploration, dans laquelle ne peut se trouver que les distances des nœuds dont il a la responsabilité. Chaque thread a un buffer de tâches, et un buffer de requêtes. L'algorithme est la répétition des cinq phases suivantes jusqu'à ce que le plus court chemin soit trouvé :

- 1 le critère global est calculé ;
- 2 les nœuds satisfaisant le critère sont retirés de la file d'exploration de chaque thread. Leur distance à la source est alors optimale. La tâche associée à un nœud est de propager sa distance à la source. Le coût de cette tâche est proportionnel au nombre d'arcs sortant du nœud. Les tâches sont distribuées aux threads en réalisant une prefix-sum, afin de distribuer la charge, et sont placés dans les buffers de tâches ;
- 3 chaque thread lit son buffer de tâches. Pour chaque successeur d'un nœud dont la distance vient d'être fixée, si la distance provisoire du successeur est améliorée par la propagation de la distance fixée, on crée une requête d'amélioration ;
- 4 on ajoute les requêtes d'amélioration dans les buffers des threads qui ont la responsabilité des nœuds sujets de l'amélioration ;
- 5 chaque thread lit son buffer de requêtes. Les améliorations pertinentes sont prises en compte, ce qui met à jour la file d'exploration.

Le critère est calculé en maintenant une file de priorité pour les nœuds visités de distance non fixée. La priorité d'un nœud u est $tent(u) + \delta_{out}(u)$.

[3] décrit une implémentation de cette algorithme, et propose des mesures expérimentales. [3] propose également des critères plus raffinés, qui permettent d'augmenter le nombre de nœuds dont la distance est propagée à chaque phase, et donc de diminuer le nombre de phases, mais en augmentant le coût du calcul du critère.

2 Définitions pour le problème de recherche de plus court chemin multi-objectif

2.1 Domination et front de Pareto

Lorsqu'on choisit de considérer plusieurs objectifs au lieu d'un, on perd une propriété intéressante : avec un seul objectif, il ne peut y avoir qu'une seule distance optimale (il peut néanmoins

y avoir plusieurs chemins qui réalisent cette distance); avec plusieurs objectifs, le nombre de distances optimales dépend de l'instance.

Soient u et v deux vecteurs représentant des distances sur $|K|$ objectifs.

u domine strictement $v \iff \exists i, u_i < v_i \ \& \ \forall i, u_i \leq v_i$

On définit l'ensemble des solutions optimales du problème de plus court chemin multi-objectif comme étant l'ensemble des distances non-dominées entre la source et le puits, aussi appelé front de Pareto.

3 Recherche parallèle de plus court chemin multi-objectif

Un algorithme de label-setting de plus court chemin multi-objectif parallèle est proposé par [6]. Il permet de paralléliser l'algorithme NAMOA*, qui est un algorithme de recherche de plus court chemin multi-objectif séquentiel dirigé. [6] propose l'utilisation d'une file de Pareto pour un nombre d'objectifs égal à 2 comme file d'exploration, qui permet de retirer les étiquettes non-dominées de la file d'exploration. Il est difficile de généraliser une telle file pour un nombre d'objectifs supérieur ou égal à 3, néanmoins on peut se contenter de prendre en compte seulement deux objectifs, et de préférence des objectifs non corrélés ou négativement corrélés entre eux.

4

Analyse et conception

1

Adaptation de l'algorithme de A *parallelization of Dijkstra's shortest path algorithm* [2]

1.1 Adaptation des critères

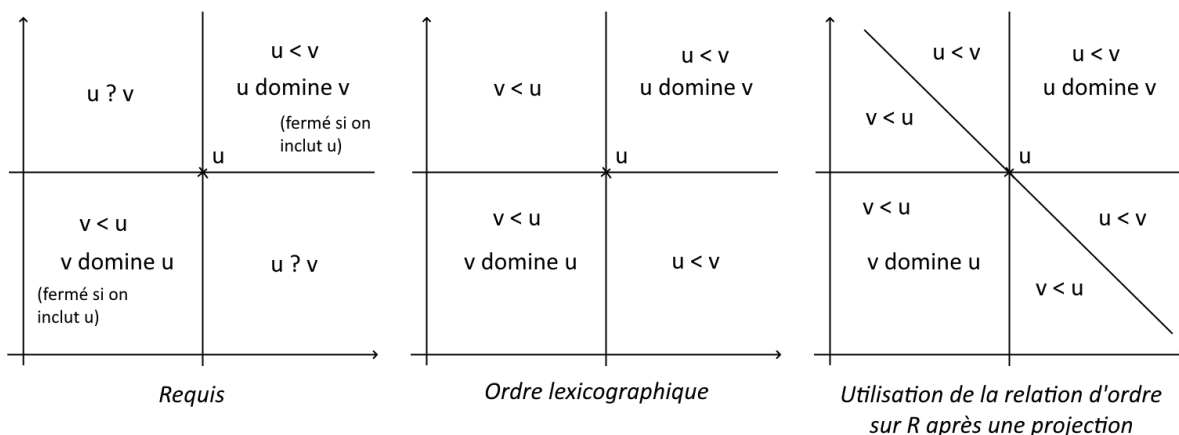
Les critères proposés par [2] peuvent être adaptés au cas multi-objectif, en prenant une relation d'ordre vérifiant :

u domine strictement $v \implies u < v$

Voici deux candidats :

- 1 L'ordre lexicographique : on compare les deux premières composantes u_1 et v_1 des étiquettes u et v . Si $u_1 > v_1$, alors $u > v$. Si $u_1 < v_1$, alors $u < v$. Si $u_1 = v_1$, alors on compare u_2 et v_2 . S'il n'y a plus de composantes à comparer, alors $u = v$.
- 2 On utilise une combinaison linéaire à coefficients strictement positifs pour passer de $|K|$ dimensions à une seule, et on utilise ensuite la relation d'ordre classique sur les réels.

Ces deux ordres sont déjà implémentés dans les stratégies d'exploration de la seconde phase séquentielle (la combinaison linéaire choisie revient au calcul de l'isobarycentre des composantes de l'étiquette).



Les stratégies parallèles ont été implémentées seulement avec l'ordre lexicographique.

1.2 Multitude d'étiquettes par nœud

Lorsqu'il y a plusieurs objectifs, on peut avoir plusieurs étiquettes non-dominées sur un même nœud. On ne connaît pas le nombre maximal d'étiquettes qui sera sur un nœud avant la résolution. Dans la version mono-objectif, il ne peut y avoir qu'une seule étiquette, donc ce problème ne se pose pas. On doit dimensionner des buffers pour chaque nœud.

Lorsque la capacité en étiquette pour un nœud est dépassée, alors on a deux cas :

- ou bien une forme d'allocation dynamique est autorisée, et on alloue un buffer plus grand.
- ou bien on ne peut pas allouer un buffer plus grand, et alors l'exactitude de la résolution ne peut être garantie, donc la deuxième phase doit être interrompue et reprise avec des buffers plus larges. Cela implique de gérer un nouveau cas de figure, l'interruption.

1.3 Boucle principale des stratégies parallèles et condition d'arrêt

Dans l'algorithme décrit dans [2], la boucle principale est décomposée en plusieurs étapes. Chaque étape est réalisée par tous les threads, et tous les threads doivent avoir terminé une étape pour commencer la suivante.

Chaque thread a une file d'exploration en accès privé, un buffer de tâches et un buffer de requêtes qui sont accessibles à tous les threads, et un ensemble de nœuds dont il est responsable.

On rappelle les différentes étapes de la boucle principale :

- 1 Calcul des IN- et OUT-Criteria ;
- 2 Extraction des tâches respectant les critères depuis les files d'exploration et leur insertion dans les buffers de tâches, en tentant de répartir la charge ;
- 3 Lecture par chaque thread de son buffer de tâches, et création éventuelle des requêtes d'ajout de nouvelles étiquettes ;
- 4 Insertion de chaque requête dans le buffer de requêtes du thread concerné (qui est responsable du nœud de l'étiquette) ;
- 5 Lecture par chaque thread de son buffer de requêtes, et mise à jour de sa file d'exploration et des liste d'étiquettes des nœuds sous sa responsabilité.

Les étapes 3 et 4 ne forment qu'une seule étape. Les requêtes peuvent être insérées dans les buffers appropriés au fur et à mesure qu'elles sont créées.

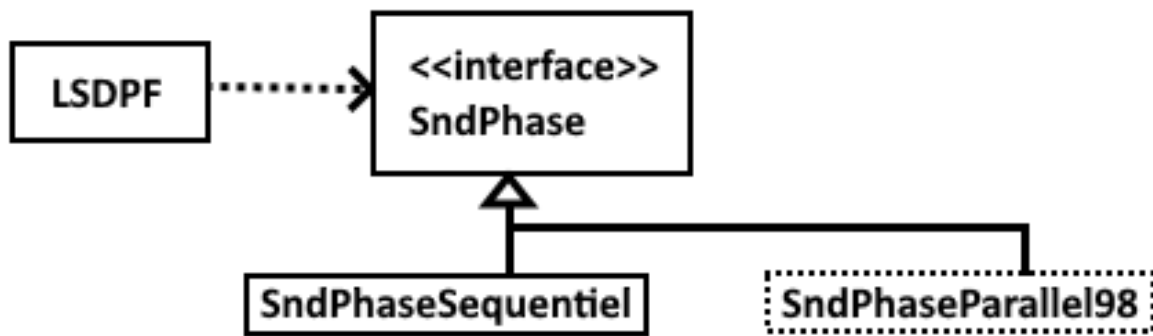
Si aucune tâche dans les files d'exploration ne satisfait les critères, c'est que toutes les files d'exploration sont vides. On peut donc utiliser une des trois conditions d'arrêt suivantes :

- après le calcul des IN- et OUT-Criteria (1) : si les critères ont leur valeur par défaut, alors la recherche est terminée ;
- après le calcul des IN- et OUT-Criteria (1) : si tous les threads déclarent que leur file d'exploration est vide, alors la recherche est terminée ;
- après l'insertion des tâches dans les buffers de tâches (2) : si aucun thread n'a inséré de tâche, alors la recherche est terminée ;

Ces trois conditions d'arrêt sont équivalentes. La condition d'arrêt utilisée pour l'implémentation des stratégies parallèles présentées dans ce rapport est la dernière.

2 Patron de conception Stratégie

On veut concevoir une nouvelle implémentation pour accomplir la deuxième phase de la méthode de résolution du problème de plus court chemin multi-objectif, on aura donc plusieurs méthodes qui ont les mêmes arguments et les mêmes type de retours. Le patron de conception Stratégie convient à ce contexte pour sélectionner différentes implémentations pendant la même exécution.



On pourra utiliser plusieurs fois les calculs réalisés dans la première phase pour différentes implémentations de la seconde phase.

3 Choix de l'architecture pour la parallélisation

On note les architectures de parallélisation sur plusieurs critères de 1 à 5, où 5 est la meilleure note, afin de montrer leurs forces et leurs faiblesses.

Les architectures de parallélisation revues sont :

- MT signifie multi-thread, tous les nœuds de calcul ont accès à la même mémoire en parallèle ;
- MPI signifie Message Passing Interface, chaque nœud de calcul a sa propre mémoire, la mémoire est distribuée ;
- GPU signifie carte graphique ;

Voici les critères propres à chaque architecture, indépendamment du cas d'utilisation :

	Capacité de calcul à chaque nœuds	Coût des communications entre nœuds	Nombre de nœuds
MT	4	5	2
MPI	5	2	4
GPU	2	4	5

La parallélisation avec un CPU permet de distribuer les calculs sur plusieurs cœurs, qui ont accès à une mémoire partagée. Le nombre d'exécution en parallèle est limité, mais les coûts de communication entre les threads sont faibles par rapport à MPI, et la mémoire disponible est plus abondante qu'avec un GPU.

La parallélisation avec MPI permet de distribuer les calculs sur plusieurs machines ou nœuds, qui ont chacune leur propre mémoire. La mémoire est distribuée, elle n'est pas partagée. Les communications entre les nœuds sont plus coûteuses.

La parallélisation avec un GPU offre une grande quantité de nœuds qui individuellement sont plus lents qu'un CPU, mais leur nombre compense cet aspect. Les nœuds ont accès à une mémoire partagée.

Voici les critères liés au problème qu'on tente de résoudre :

	File d'exploration	Allocation dynamique	Barrières
MT	5	5	5
MPI	5	5	2
GPU	3	1	2

Les files d'exploration sont incontournables dans la recherche d'un plus court chemin. Avec un CPU, donc pour MT et MPI, il existe de nombreuses implémentations de files de priorité. Avec un GPU, on ne trouve ces mêmes structures. Une implémentation de file de priorité utilisée pour l'algorithme de Dijkstra est le tas de Fibonacci, qui implique la réalisation de nombreuses allocations dynamiques, ce qui motive [3] à choisir un allocateur particulier afin de diminuer le coût des allocations. Cuda n'offrant pas d'allocations dynamiques, une telle structure ne peut pas être implémentée facilement. Par contre, on peut simplement utiliser un tableau et réaliser des recherches linéaires à chaque fois qu'on veut trouver le minimum. Les résultats d'expérimentaux de [3] montrent que l'utilisation d'un tableau au lieu d'un tas de Fibonacci peut donner de meilleures performances, dans le cas mono-objectif.

En MPI, les barrières demandent de la communication entre les nœuds, donc cet aspect hérite du haut coût des communications pour MPI. En cuda, il n'y a pas de barrière inter-bloc, à part l'appel d'un nouveau kernel. Cela n'a pas été vu par le développeur, et demandera plus de recherches.

Voici les critères liés aux moyens disponibles :

	Accessibilité	Expérience du développeur
MT	5	5
MPI	1	1
GPU	4	3

Les connaissances du développeur en MPI étant nulles, et la demande d'accès à une machine adaptée serait non triviale, donc cette option est abandonnée. Le développeur n'a de plus aucune expérience dans l'utilisation de la bibliothèque MPI. L'utilisation de cette bibliothèque est donc écartée.

Étant donné le besoin de plus amples recherches dans la synchronisation inter-bloc pour le GPU nécessaire à l'implémentation de l'algorithme de 98 [2], l'utilisation du GPU ne sera envisagée qu'après la réalisation d'une parallélisation multi-threadée au niveau du CPU (MT), d'autant plus que le développeur a moins d'expérience avec les GPU qu'avec les CPU.

Le temps disponible pour ce projet a été consacré à réalisation des différentes versions de parallélisation multi-threadée, ce qui n'en laisse pas pour la conception d'une version sur GPU.

5

Mise en œuvre

1 Outils utilisés

Voici la liste des outils utilisés :

- g++ 7.4.0, Ubuntu 7.4.0-1ubuntu1 18.04.1 ;
- doxygen 1.8.13;
- nano 2.9.3;
- ArgoUML 0.34;

2 Librairies utilisées

Les librairies utilisées sont Boost, Osmium et OpenMP, ainsi que la librairie standard du C++. Une structure de file MPMC sous licence MIT a également été utilisée dans une des stratégies parallèles de la seconde phase.

2.1 Boost

On appellera composant les parties de Boost qui nécessitent d'être compilées.

Boost est utilisé pour les fins suivantes :

- la représentation du graphe sous la forme d'une liste d'adjacence (`boost::adjacency_list`);
- l'inversion du graphe lors du prétraitement (`boost::reverse_graph`);
- la représentation des files d'exploration et listes d'étiquettes par nœud (`boost::multi_index_container`);
- le lancement des recherches mono-objectifs dans le graphe inversé pendant le prétraitement (composant `thread`);
- la création du dossier dans lequel les résultats des recherches de plus court chemin seront écrits (composant `filesystem`);
- les tests unitaires (composant `unit_test_framework`);

Toutes ces utilisations de Boost, à l'exception des tests unitaires, étaient présentes dans l'existant. Trois dépendances à des composants de Boost ont été supprimées : `chrono`, `system` et `timer`. Ces composants permettaient la mesure du temps pris pour résoudre chaque instance, mais cette

tâche est désormais assurée avec la librairie standard du C++. Ce changement dans la façon de mesurer le temps a été motivé par le besoin de mesurer le temps réel au lieu du temps CPU total, car l'existant mesurait le temps CPU total et ce temps s'écoule à une vitesse proportionnelle au nombre de threads actifs du processus.

On envisagera de supprimer la dépendance au composant filesystem en utilisant `std::filesystem` qui a été introduit depuis C++17.

2.2 Osmium

L'unique élément issu de Osmium est la classe `osmium::Location`, qui stocke la position géographique d'un nœud. Les données géographiques ne sont néanmoins pas exploitées. C'est un vestige de l'existant, qui ne s'en servait pas non plus.

2.3 OpenMP

OpenMP est utilisé dans chaque version parallèle de la seconde phase.

2.4 Rigport

`Rigport::MPMCQueue` est l'implémentation d'une file supportant plusieurs producteurs et plusieurs consommateurs. Une telle file est utilisée dans la méthode `SPPT4asyncmd` dans le cadre de la communication entre le thread dédié au puits et les autres threads : les autres threads sont les producteurs, ils ajoutent dans la file des étiquettes à placer dans le front de Pareto ; le thread dédié au puit est l'unique consommateur, il extrait de la file les étiquettes et vérifie qu'elles sont non-dominées avant de les ajouter au front de Pareto.

Cette file MPMC pourra être remplacée par tout autre file MPMC.

3 Implémentation

Voici des détails concernant l'implémentation du patron de conception Stratégie et des différentes stratégies.

3.1 Implémentation du patron de conception Stratégie

La stratégie à adopter pour réaliser la seconde phase est définie par l'argument de template donné lors de l'appel de la méthode `runSecondPhase` de `LSDPF`.

Cette implémentation du patron de conception ne permet pas d'utiliser des paramètres connus lors de l'exécution sans passer par des variables globales ou des ersatz.

Une implémentation utilisant du polymorphisme aurait permis de faire passer des paramètres spécifiques à chaque stratégie sans nécessiter des variables globales.

Le coût engendré par le polymorphisme lors de l'appel aux méthodes de l'interface `SndPhase` serait négligeable puisque les méthodes de `SndPhase` ne sont appelées qu'une seule fois lors d'une seconde phase.

3.2 Stratégies

Dans cette section, les différentes stratégies implémentées pour réaliser la seconde phase sont présentées.

Comme dans l'existant, toutes les stratégies sont dépendantes du paramètre `ITERATION_LABEL_MAJ_DYN_PF` défini au lancement de l'application et non à la compilation. Ce paramètre correspond au nombre de générations entre deux mises à jour dynamiques du front de Pareto. Autrement dit, au lieu de réaliser une mise à jour dynamique du front de Pareto pour chaque étiquette ajoutée dans une file d'exploration, on ne les réalise que pour les étiquettes dont la génération est divisible par `ITERATION_LABEL_MAJ_DYN_PF`.

La génération d'une étiquette est définie ainsi : la première étiquette a pour génération 0 ; lorsqu'une étiquette de génération g est extraite d'une file d'exploration, les étiquettes qu'elle engendrera auront pour génération $g+1$.

Les stratégies séquentielles `SndPhaseSequentiel` et `SndPhaseSequentielT` acceptent une valeur de 0 pour `ITERATION_LABEL_MAJ_DYN_PF`, néanmoins une contre-performance de `SndPhaseSequentiel` face à `SndPhaseSequentielT` a lieu avec cette valeur. Cet incident est détaillée dans l'analyse des résultats des mesures en annexe.

3.2.1 SndPhaseSequentiel

La première stratégie, `SndPhaseSequentiel`, abrégée SPS, doit réaliser la seconde phase comme le faisait auparavant LSDPF avant que cette responsabilité ne lui soit retirée.

`SndPhaseSequentiel` utilise donc les mêmes structures de données que l'ancienne LSDPF pour stocker les étiquettes de chaque nœud et pour représenter la file d'exploration : `boost::multi_index_container`.

`SndPhaseSequentiel` est dépendante de `EXPLORATION_STRATEGY`, tout comme l'était LSDPF.

3.2.2 SndPhaseSequentielT

La deuxième stratégie, `SndPhaseSequentielT`, abrégée SPST, a pour objet de faire exactement ce que `SndPhaseSequentiel` fait, mais en retirant la dépendance à la macro `EXPLORATION_STRATEGY`.

A cette fin, SPST est un template, dont les deux arguments sont la structure servant de file d'exploration (`multi_index_container` attendu) et la stratégie employée.

3.2.3 SPPT3md

La première stratégie parallèle est nommée SPPT3md, pour Second Phase Parallel Template 3 Mise (à jour) Dynamique.

La boucle principale de SPPT3md suit les étapes décrites dans la section "Boucle principale des stratégies parallèles" du chapitre "Analyse et conception".

La mise à jour dynamique a lieu lors de la création d'une requête d'ajout d'étiquette. Si l'étiquette d'une requête satisfait les conditions pour qu'une mise à jour dynamique se produise, des étiquettes candidates au front de Pareto sont générées, leur non-domination est vérifiée, puis elles sont insérées dans le buffer de requêtes du thread responsable du puits.

Le thread responsable du puits n'est responsable que du puits, on parle de thread dédié. Son buffer de requêtes a une taille plus grande que les autres.

Le reste des nœuds est partagé entre les autres threads.

Le thread responsable du puits réalise les étapes de la boucle principale comme les autres threads.

3.2.4 SPPT4asyncmd

La deuxième stratégie parallèle est nommée SPPT4asyncmd, pour Second Phase Parallel Template 4, avec "async" pour "thread dédié détaché de la boucle principale" et "md" pour "Mise (à jour) Dynamique".

La boucle principale de SPPT4asyncmd suit les étapes décrites dans la section "Boucle principale des stratégies parallèles" du chapitre "Analyse et conception".

La mise à jour dynamique a lieu lors de l'étape de lecture des buffers de requêtes, lorsqu'une requête est acceptée et est ajoutée à une liste d'étiquettes d'un nœuds. Les étiquettes candidates au front de Pareto qui sont alors générées voient leur non-domination vérifiée, puis sont placées dans la file des requêtes à destination du puits.

Un thread est dédié au puits, les autres se partagent le reste des nœuds.

Le thread dédié au puits ne réalise pas les étapes de la boucle principale comme les autres threads. Il défile les éléments présents dans la file des requêtes à destination du puits, vérifie leur non-domination, puis les insère dans un conteneur spécifique (nommé maladroitement "StaticCriteriaValuesContainer") pour stocker le front de Pareto.

Les autres threads vérifient que les étiquettes candidates au front de Pareto sont non-dominées avant de les insérer dans la file des requêtes. Néanmoins, le front de Pareto est susceptible de changer, ce qui pourrait rendre une étiquette candidate dominée. Ce n'est pas un problème, puisque le thread dédié vérifie la non-domination de chaque étiquette candidate avant leur insertion dans le front de Pareto. Un problème menaçant l'exactitude de la stratégie se pose si un des autres threads décide qu'une étiquette candidate est dominée alors qu'elle ne l'est pas. Un tel cas pourrait se produire si la lecture des composantes de l'étiquette était partielle, et qu'une réécriture des composantes de l'étiquette avait lieu au milieu de la lectures des différentes composantes de l'étiquette. Si une étiquette candidate est jugée à tort dominée, il risque de manquer une solution au front de Pareto, et des solutions non-optimales pourraient se glisser dans le front. Pour empêcher ce cas de se produire, le conteneur mal-nommé est spécifié ainsi :

Ce conteneur spécifique offre la possibilité à plusieurs threads de lire son contenu simultanément (afin de vérifier la non-domination d'étiquettes), ainsi qu'un à un seul thread d'écrire des étiquettes, le tout sans verrou :

- s'il n'y a pas d'étiquettes marquées (ou purgées), le thread dédié ajoute les nouvelles étiquettes à la suite des étiquettes déjà présentes. En cas de déplacement de la capacité du conteneur, le thread dédié signalera l'incident et provoquera l'arrêt de la recherche ;
- lorsque le thread dédié juge qu'une étiquette doit être purgée, parce qu'elle est dominée, il l'a marque avec le numéro de l'itération de la boucle principale en cours. Le contenu de l'étiquette n'est pas écrasé immédiatement ;
- lorsque qu'un thread quelconque consulte les étiquettes, il ignore les étiquettes marquées ;
- lorsque qu'un thread quelconque consulte une étiquette qui vient d'être marquée, car il a réalisé la vérification de l'absence de marquage juste avant que le marquage ne soit posé, il lira les composantes d'une étiquette purgée, mais ce n'est pas un problème : si l'étiquette dont la domination est vérifiée est dominée par une étiquette elle-même dominée par une autre, alors celle subissant la vérification est dominée ; la vérification n'est pas mise en péril ;

- pour que le thread dédié écrase une étiquette purgée avec le contenu d'une nouvelle étiquette, il faut que le numéro de l'itération de la boucle principale avec lequel l'étiquette purgée a été marquée soit inférieur strictement à celui de l'itération en cours. Ainsi on est assuré qu'aucun autre thread ne lit les composantes de l'étiquette purgée.

L'utilisation d'un thread dédié qui ne suit pas la boucle principale comme les autres ne sert qu'à faire apparaître des problèmes de concurrences qui étaient résolus par la boucle principale et les barrières à la fin de chaque étape. Cela crée un besoin pour une structure complexe et fragile, qui prend du temps à décrire et qui est certainement mal conçue.

3.2.5 Limites des méthodes parallèles implémentées

Les méthodes SPPT3md et SPPT4asyncmd ont le même défaut : elles ne font gagner du temps que sur les instances qui ne prenaient déjà pas beaucoup de temps à se résoudre avec la version séquentielle. Elles donnent de plus des résultats très similaires, et SPPT3md est légèrement meilleure en moyenne.

Il est regrettable qu'il n'y ait pas de stratégie sans thread dédié avec laquelle on pourrait comparer les stratégies SPPT3md et SPPT4asyncmd.

3.3 Structures pour les files d'exploration et les listes d'étiquettes par nœud

L'existant utilise des `boost::multi_index_container` en tant que files d'exploration et listes d'étiquettes par nœud.

Les stratégies SPS et SPST utilisent ces structures.

Les stratégies SPPT3md et SPPT4asyncmd utilisent `ExplorationArray` et `LabelArray` en tant que file d'exploration et liste d'étiquettes par nœud.

3.3.1 LabelArray

La structure `LabelArray` est un conteneur (`std::vector`) d'étiquettes non ordonné, tel que les étiquettes restent à la même position une fois après leur insertion. Les étiquettes purgées sont simplement marquées afin d'être ignorées lors des lectures et être finalement écrasées lors de l'insertion d'une nouvelle étiquette.

3.3.2 ExplorationArray

La structure `ExplorationArray` n'est pas une file d'exploration, c'est simplement un tableau (`std::vector`) de références à des étiquettes présentes dans des `LabelArray`. Il est systématiquement parcouru entièrement lors de la détermination des IN- et OUT-Criteria, l'extraction des tâches respectant les critères, et la purge des étiquettes dominées. Cette structure a été envisagée après la lecture de l'article [3] dans lequel l'utilisation d'un simple tableau au lieu d'un tas de Fibonacci (file de priorité) donnait de meilleurs résultats.

Puisque `ExplorationArray` stocke des références à des étiquettes au sein de structures `LabelArray`, les composantes des étiquettes ne sont pas stockées plusieurs fois.

On peut déplorer l'absence d'alternatives à `ExplorationArray`.

3.4 Limites

On peut déplorer des arrêts prématurés du programme causés par un dépassement de capacité mémoire pour de nombreuses instances des plus grands graphes.

Il est inévitable que certaines instances ne puissent pas être traitées à cause de ce problème.

Il a été jugé coûteux en temps de développement de se prémunir de ce problème, c'est pourquoi il a été ignoré, et les instances provoquant des dépassements de capacité ont été supprimées des campagnes de mesure.

4 Qualité et performances

Des tests unitaires ont été implémentés, ainsi qu'un ensemble de scripts permettant de réaliser des campagnes de résolution d'instances, afin de vérifier l'exactitude des résultats de chaque stratégie et les temps pris par chaque stratégie.

Ces éléments se concentrent uniquement sur la seconde phase de la méthode, puisque le développement de stratégies pour cette phase est l'objet du projet dont ce document est le rapport.

4.1 Tests unitaires

Les tests unitaires sont peu nombreux, et ne concernent que deux fonctions vérifiant la domination entre deux séquences et deux conteneurs d'étiquettes.

La solution technique choisie pour réaliser les tests unitaires présente des faiblesses, puisqu'il est laborieux d'ajouter des tests supplémentaires : il faudrait modifier le fichier servant à la compilation des exécutables réalisant les tests unitaires. Cette solution relève beaucoup du fait-maison, malgré l'utilisation d'un framework (Boost, `unit_test_framework`).

4.2 Lancement de campagnes de résolution d'instances

Un ensemble de scripts python et shell a été développé afin de vérifier l'exactitude des stratégies et de mesurer leurs performances sur plusieurs graphes.

Ces scripts permettent de commander le lancement de la résolution d'instances de différents graphes avec différentes stratégies. Les instances à résoudre et les résultats des résolutions sont stockés dans une base de données locale (Sqlite). Parmi les résultats sont présents la taille du front de Pareto trouvé ainsi que les temps de résolution nécessaires.

Un des principaux intérêts de ce système est la possibilité d'interrompre et de reprendre la campagne de test sans devoir la recommencer entièrement.

4.2.1 Vérification de la correction des stratégies

Afin de vérifier que les stratégies sont exactes, on compare seulement la taille du front de Pareto trouvé pour chaque instance par toutes les stratégies. Si toutes les stratégies ne donnent pas la même taille de front de Pareto, alors nécessairement l'une d'entre elles au moins n'est pas correcte.

Une telle approche a permis de constater que la version séquentielle existante contenait un vice : le prétraitement ne garantit pas que les étiquettes servant pour l'initialisation du front de Pareto ne sont pas non-dominées. Ainsi, pour l'instance #23 du graphe COL (COL_nodes.csv, COL_edges.csv, COL_instances_Raith.csv) le front de Pareto final pour SPS et SPST était (187655, 317538), (187655, 317628) au lieu d'être (187655, 317538) puisque (187655, 317538) domine (187655, 317628).

Des graphes rcsp ont été utilisés uniquement à cette effet. Leur petite taille permet d'avoir une campagne de tests rapide. Ces graphes ont permis de tester l'exactitude des stratégies avec 3, 5 et 11 objectifs, alors que les autres graphes utilisés pour la mesure des performances n'offrent que 2 objectifs.

Les campagnes de tests associées aux graphes rcsp peuvent être considérées comme faisant parti des tests unitaires.

Toutes les campagnes de tests ont fait l'objet d'une vérification de la cohérence des tailles de front de Pareto trouvé par chaque stratégie.

4.2.2 Mesure des temps de résolution des instances

Les temps de résolution des instances sont mesurés par le programme, et sont écrits dans un fichier conformément à la spécification de l'existant.

Les graphes utilisés uniquement pour la mesure des temps de résolution sont tous des graphes à deux objectifs, et sont les suivants : BAY, berlin, NW, NY et paris.

D'autres graphes ont été fournis mais n'ont pas été utilisés pour les mesures car un grand nombre de leurs instances provoquaient des arrêts prématurés de la résolution à cause d'un manque de mémoire vive sur l'ordinateur utilisé pour les mesures.

6

Bilan et conclusion

1 Fait et Reste à faire

Ce qui a été fait :

- ajout d'un patron de conception Stratégie pour la seconde phase de la méthode :
 - extraction de la stratégie originelle pour la seconde phase de la classe LSDPF ;
 - déplacement de l'implémentation de cette stratégie dans une autre classe (SPS) ;
 - ajout d'une stratégie séquentielle s'affranchissant de la macro EXPLORATION_STRATEGY (SPST) ;
 - ajout d'une stratégie parallèle inspirée de [2] avec tous les threads suivant le cycle (SPPT3md) ;
 - ajout d'une stratégie parallèle inspirée de [2] dont le thread dédié au puits ne suit pas le cycle (SPPT4asyncmd) ;
- vérifier l'exactitude des stratégies sur les graphes mis à disposition ;
- réaliser des mesures afin de comparer les différentes stratégies ;

Ce qui reste à faire :

- envisager une stratégie parallèle exploitant un GPU ;
- envisager une stratégie parallèle exploitant les positions géographiques des nœuds ;
- implémenter une nouvelle stratégie en prenant l'une des stratégies parallèles ajoutées et en retirant la notion de thread dédié au puits.

La stratégie utilisant un GPU a été abandonnée, ainsi que celle mentionnant les positions géographiques des nœuds.

Améliorations envisageables :

- changer le patron de conception Stratégie : utiliser l'interface SndPhase et le polymorphisme pour passer une instance configurée au lieu d'utiliser un paramètre de template ; se libérer ainsi des dépendances aux variables globales ;
- tester une autre implémentation d'une file MPMC dans la méthode parallèle asynchrone, notamment une qui ne demanderait pas d'attente active ;
- renommer la structure StaticCriteriaValuesContainer ;
- renommer les stratégies ;

2 Aléas

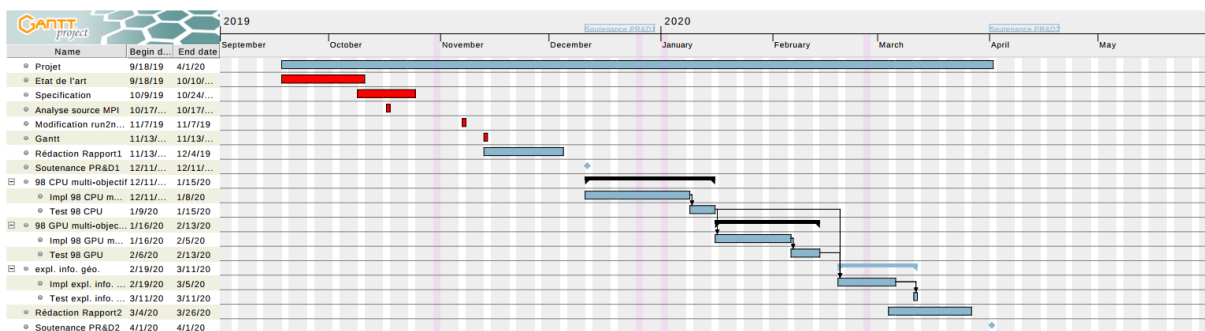
Après la soutenance de décembre 2019, l'ordinateur sur lequel le développement avait lieu a été rendu inopérant. La panne irréversible a eu lieu en dehors des horaires dédiés au projet, et un remplacement a été trouvé rapidement. La réinstallation de l'environnement de développement a été réalisée en dehors des horaires dédiés, par conséquent aucune heure n'a été perdue.

C'est néanmoins à l'occasion de cet incident que l'utilisation de CMake a été abandonnée, et a été remplacée par les fichiers Makefile et config présents dans solution/.

3 Planning du second semestre

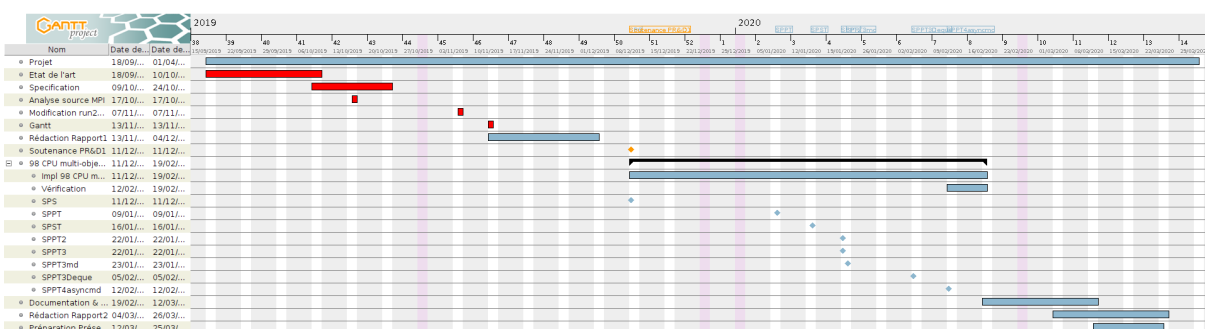
Au début du second semestre, le plan suivant a été choisi : développer deux stratégies parallèles en combinant le parallélisme décrit dans [2] et les éléments multi-objectifs de l'existant, un sur CPU et un sur GPU, puis de développer une autre stratégie parallèle exploitant les positions géographiques des nœuds. Et réaliser des mesures et tester la correction des stratégies.

L'exploitation des informations géographiques risquait d'être ignorée, et le temps disponible finirait alors absorbé par la rédaction du rapport final.



A la fin du second semestre, l'utilisation d'un GPU ainsi que l'exploitation des informations géographiques ont été abandonnées. Le temps qui leur était accordé a été absorbé par le développement et le test des stratégies parallèles sur CPU.

Voici le diagramme de Gantt final.



4 Qualité

L'ajout de tests unitaires a permis de déterminer la cause des premières difficultés rencontrées lorsque de la conception de SPPT4asyncmd. On peut regretter que les tests unitaires aient été ajoutés après la découverte d'un problème, pour tenter d'en identifier plus précisément la

cause, plutôt que d'être conçus dès le début. De plus, la couverture de code assurée par les tests unitaires implémentés est très faible, et l'ajout de nouveaux tests unitaires n'est pas simple.

Le développement d'un ensemble de scripts shell et python pour le lancement automatique de la résolution d'instances a permis la vérification superficielle de l'exactitude des stratégies sur une multitude de graphes et avec différentes configurations.

Ces vérifications ont permis de découvrir un problème mineur dans la stratégie séquentielle originelle : les étiquettes fournies par le prétraitement pour initialiser le front de Pareto ne sont pas forcément strictement non-dominées, et l'initialisation du front de Pareto se faisait sans prendre en compte cette non-garantie.

Les mesures réalisées ont permis de mettre en évidence les gains ou l'absence de gain apportées par chaque stratégie.

- les deux stratégies séquentielles ont des performances similaires, si on omet une certaine anomalie décrite en annexe, ce qui est rassurant puisqu'elles sont presque identiques ;
- les deux stratégies parallèles ont également des performances similaires, et ont la même limite, à savoir que les instances pour lesquelles le temps de résolution est diminué sont celles qui avaient déjà un temps de résolution faible. Néanmoins l'une est en moyenne plus rapide que l'autre. Cette limite est due aux structures utilisées pour représenter les files d'exploration et les listes d'étiquettes par nœud.

5 Gestion de projet

La gestion de projet n'a pas été exemplaire.

Le diagramme de Gantt n'a pas été mis à jour fréquemment, et il aurait pu être plus détaillé. Une liste de tâches à faire dynamique (Trello par exemple) aurait pu être utilisé. Les interfaces exactes des classes utilisées n'ont pas toutes été définies au premier semestre.

Annexes

A

Spécifications fonctionnelles de l'existant

1 Constantes globales

Les constantes globales sont définies dans Const.h.

- `EXPLORATION_STRATEGY` définit le critère à minimiser pour retirer une étiquette de la file d'exploration. On utilise soit l'ordre lexicographique (`ES_DIJKSTRA`), soit on minimise une combinaison linéaire à coefficients positifs des composantes de l'étiquette (`ES_LC`), soit on utilise une heuristique (`ES_A_STAR`);
- `NB_CRITERIA` est une macro qui définit $|K|$ le nombre de critères;
- `std::vector<std::vector<double>> const firstPhaseWeights` contient les barycentres à coefficients positifs utilisés par la première phase de la méthode de résolution.
- `std::vector<u_int8_t> const secondPhaseWeightIndexes` contient les identifiants des barycentres de `firstPhaseWeights` qui sont utilisés dans la deuxième phase pour la mise à jour dynamique du front de Pareto.
- `extern const std::map<u_int8_t, u_int8_t> &weightsMap` est utilisée pour le calcul des chemins idéaux pendant la seconde phase.
- `extern int ITERATION_LABEL_MAJ_DYN_PF` n'est pas constant, car il est un des paramètres d'entrée du programme. Néanmoins, après son initialisation, il n'est pas modifié. Il est utilisé par la seconde phase pour limiter le nombre de mises à jour dynamiques du front de Pareto.

2 Les sorties

2.1 Sortie standard

La première écriture dans la sortie standard est une description synthétique du graphe :
"Graph data :

Vertices : [nombre de nœuds]

Edges : [nombre d'arcs]".

Ensuite, pour chaque instance traitée seront affichés les résultats de la résolution de l'instance :
"new LSDPF :

starts :

```
# [instance_id] : t|S| = [taille du front de Pareto], time = [temps pris par la résolution, mesuré par LSDPF et non dans main.cpp]ms
```

ends and write

```
delete lsdpf :".
```

L'affichage des deux premières lignes est forcé avant le traitement de l'instance :

```
"new LSDPF :
```

```
starts :".
```

2.2 Sortie standard d'erreur

Si une exception est levée, "Exception caught : [message de l'exception]" sera écrit dans la sortie standard d'erreur.

2.3 Sous-dossier des résultats

Un sous-dossier est créé dans le dossier défini par un paramètre d'entrée du programme, dont le nom est la date de l'exécution courante du programme, au format YYYYMMDDhhmmss.

2.4 Fichier results.csv

Un fichier results.csv est créé dans le sous-dossier défini ci-dessus. Il contient :

- la liste des barycentres utilisés dans la première phase (dans la constante globale firstPhaseWeights);
- la liste des barycentres utilisés dans la seconde phase (dans la constante globale firstPhaseWeights dont l'identifiant se trouve dans la constante globale secondPhaseWeightIndex);
- la stratégie de sélection des étiquettes de la seconde phase;
- la valeur de ITERATION_LABEL_MAJ_DYN_PF;
- l'entête "#;Solutions;First phase (ms);Second phase (ms);Total (ms);Labels created;Labels added to PF";

Puis pour chaque instance, on écrit le résultat de la résolution : "[identifiant de l'instance];[taille du front de Pareto];[temps pris par la première phase];[temps pris par la seconde phase];[somme des temps pris];[nombre d'étiquettes créées];[nombre d'étiquettes ajoutées au front de Pareto]".

2.5 Fichiers results_[identifiant de l'instance].csv

Pour chaque instance résolue, on crée un fichier nommé results_[identifiant de l'instance, commençant par 0 si inférieur à 10].csv. Dans ce fichier est écrit la liste des solutions du front de Pareto. Chaque solution est séparée par un saut de ligne. Pour chaque solution, les composantes sont séparées par des points-virgules.

3 Fichiers d'entrée

Le programme a besoin de trois fichiers d'entrée : un pour la liste de nœuds, un pour la liste des arcs, et un pour la liste des instances. Dans ces trois fichiers, les entités sont séparées par un saut de ligne. Les attributs d'une entité sont séparés par une tabulation. Les attributs d'une entité sont soit entier, soit réel, et sont écrit en base décimale.

Les attributs d'un nœud sont son identifiant (entier), sa latitude (réel) et sa longitude (réel).

Les attributs d'un arc sont l'identifiant du nœud d'origine (entier), l'identifiant du nœud d'arrivée (entier), puis les $|K|$ composantes du poids de l'arc, toutes réelles, avec $|K|$ le nombre d'objectifs.

Les attributs d'une instance sont son identifiant (entier), l'identifiant de la source (entier), et l'identifiant du puits (entier).

4 La classe Parser

La classe Parser permet de lire les trois fichiers d'entrée. Cette classe a trois propriétés privées qui sont accessibles en copie ou en lecture seule avec des getters : une liste de nœuds (Vertex), une liste d'arcs (Edge), et une liste d'instance (Instance).

4.1 void Parser::parseVerticesFile(std::string const &)

Parser::parseVerticesFile prend en argument un nom de fichier. Parser::parseVerticesFile ne retourne pas de valeur. Si l'ouverture en lecture seule du fichier échoue, l'erreur `std::invalid_argument("Vertices file not found")` est levée. Sinon, cette méthode ajoute les nœuds lus à la liste interne des nœuds.

4.2 void Parser::parseEdgesFile(std::string const &)

Parser::parseEdgesFile prend en argument un nom de fichier. Parser::parseEdgesFile ne retourne pas de valeur. Si l'ouverture en lecture seule du fichier échoue, l'erreur `std::invalid_argument("Edges file not found")` est levée. Sinon, cette méthode ajoute les arcs lus à la liste interne des arcs.

$|K|$ est connu à la compilation. Si le nombre d'objectifs du fichier d'arcs ne correspond pas au nombre d'objectifs pour lequel le programme a été compilé, alors le fichier d'arcs sera mal lu, ce qui rendra le résultat de la recherche de plus court chemin inutilisable.

4.3 void Parser::parseInstancesFile(std::string const &)

Parser::parseInstancesFile prend en argument un nom de fichier. Parser::parseInstancesFile ne retourne pas de valeur. Si l'ouverture en lecture seule du fichier échoue, l'erreur `std::invalid_argument("Instance file not found")` est levée. Sinon, cette méthode ajoute les instances lues à la liste interne des instances.

5 La classe Graph

La classe Graph permet de construire une liste d'adjacence (`graph_t`) à partir d'une liste d'arcs (Edge) et de nœuds (Vertex). La classe Graph permet également de convertir les identifiants de nœuds en identifiants utilisés par la liste d'adjacence. Un getter permet de récupérer la liste d'adjacence en lecture seule, nommé `graph_t const & getInstance() const`. La conversion d'un identifiant de nœuds vers un identifiant utilisé par la liste d'adjacence est réalisée par la méthode `vertex_t const & getVertex(int64_t const vertexId) const`.

5.1 void Graph::setVertices(std::vector<Vertex>)

Graph::setVertices prend en argument une liste de nœuds par valeur. Le passage par valeur dans ces conditions n'est pas justifié. Graph::setVertices ne retourne pas de valeur. Cette méthode ajoute les nœuds passés en argument à la liste d'adjacence.

5.2 void Graph::setEdges(std::vector<Edge>)

Graph::setEdges prend en argument une liste d'arcs par valeur. Le passage par valeur dans ces conditions n'est pas justifié. Graph::setVertices ne retourne pas de valeur. Cette méthode ajoute les arcs passés en argument à la liste d'adjacence.

$|K|$ est connu à la compilation. Si le nombre d'objectifs du fichier d'arcs ne correspond pas au nombre d'objectifs pour lequel le programme a été compilé, alors le fichier d'arcs sera mal lu, ce qui rendra le résultat de la recherche de plus court chemin inutilisable.

5.3 void Graph::printData() const

Graph::printData écrit dans la sortie standard le nombre d'arcs et le nombre de nœuds présents dans la liste d'adjacence, dans le format défini ci-après, puis en force l'affichage.

"Graph data :

Vertices : [nombre de nœuds]

Edges : [nombre d'arcs]"

6 La classe LSDPF

La classe LSDPF permet, en connaissant une liste d'adjacence, une source et un puits, de résoudre le problème de plus court chemin multi-objectif, c'est-à-dire de déterminer le front de Pareto. Le constructeur de la classe LSDPF a besoin d'une instance de la classe Graph, de l'identifiant du nœud source et de l'identifiant du nœud puits. La classe LSDPF mesure le temps pris par les deux phases de la méthode, ainsi que le nombre d'étiquettes créées et le nombre d'étiquettes ajoutées au front de Pareto lors de la seconde phase. Ces données sont organisées dans une structure nommée Stats, et la classe LSDPF dispose d'un getter pour accéder en lecture seule à l'instance de cette structure. Les lancements de la première et de la deuxième phase doivent être déclenchés par un acteur extérieur à la classe. L'acteur extérieur a la responsabilité d'appeler ces méthodes dans le bon ordre.

6.1 void LSDPF::runFirstPhase()

LSDPF::runFirstPhase ne prend pas d'arguments. La propriété criteriaValues nécessaire au déroulement de la deuxième phase est initialisée par cette méthode. Le temps pris par la première phase est mesuré. On notera l'utilisation de la constante globale firstPhaseWeights pour obtenir les barycentres à coefficients positifs nécessaires à l'exécution de la première phase.

6.2 void LSDPF::runSecondPhase()

LSDPF::runSecondPhase ne prend pas d'arguments. Néanmoins, cette méthode utilise la propriété `criteriaValues` qui a normalement été initialisée par LSDPF::runFirstPhase. Le temps pris par la seconde phase est mesuré. Le nombre d'étiquettes créées, et le nombre d'étiquettes ajoutées au front de Pareto sont également mesurées. On notera l'utilisation des constantes globales `weightsMap`, `firstPhaseWeights` pour concevoir les chemins idéaux à partir de `criteriaValues`, et de `secondPhaseWeightIndexes` et `ITERATION_LABEL_MAJ_DYN_PF` pour la mise à jour dynamique du front de Pareto.

6.3 vertex_label_list_t const & LSDPF::getParetoFront() const

Cette méthode est un getter pour récupérer le front de Pareto, une fois qu'il a été calculé dans le cadre de l'exécution de la méthode `runSecondPhase`. Le type `vertex_label_list_t` est le nom donné à un conteneur de Boost paramétré pour contenir des étiquettes.

6.4 void LSDPF::printStatistics() const

Cette méthode n'a pas de paramètres. Elle écrit dans la sortie standard des informations dans le format suivant :

"LSDPF statistics :

Solutions : [taille du front de Pareto]

First phase time : [temps pris par la première phase]ms

Second phase time : [temps pris par la seconde phase]ms

Created labels : [nombre d'étiquettes créées]

Labels added to PF : [nombre d'étiquettes ajoutées au front de Pareto]".

Puis elle force l'affichage de ces lignes.

6.5 void LSDPF::printStatistics(int64_t) const

Cette méthode prend pour paramètre l'identifiant de l'instance. Elle écrit dans la sortie standard des informations dans le format suivant :

"#[identifiant de l'instance] : |S| = [taille du front de Pareto], time = [somme des temps pris par les deux phases]ms".

Puis elle force l'affichage de ces lignes.

7 La classe Writer

La classe `Writer` permet d'écrire tous les résultats après la résolution d'une instance. C'est notamment elle qui permet d'écrire le contenu du front de Pareto dans un fichier.

7.1 Writer::Writer(std::string const &)

Le constructeur de cette classe prend en argument un nom de dossier, dans lequel sera créé un dossier ayant pour nom la date de l'exécution courante, dans lequel seront créés les différents fichiers associés aux différentes instances.

Si le dossier ne peut pas être créé, alors une exception `std::invalid_argument("Result directory cannot be created")` est levée.

Sinon, le dossier est créé, et un fichier `results.csv` est créé. Le fichier `results.csv` reçoit l'entête suivant :

- la liste des barycentres utilisés dans la première phase (dans la constante globale `firstPhaseWeights`);
- la liste des barycentres utilisés dans la seconde phase (dans la constante globale `firstPhaseWeights` dont l'identifiant se trouve dans la constante globale `secondPhaseWeightIndex`);
- la stratégie de sélection des étiquettes de la seconde phase;
- la valeur de `ITERATION_LABEL_MAJ_DYN_PF`;
- l'entête `"#;Solutions;First phase (ms);Second phase (ms);Total (ms);Labels created;Labels added to PF"`;

7.2 `void Writer::writeResults(Instance const &, LSDPF const *, double, double)`

`Writer::writeResults` prend en argument une instance, une instance du solveur, le temps pris par la première phase, et le temps pris par la seconde phase. Cette méthode écrit à la suite du fichier `results.csv` la ligne suivante : `"[identifiant de l'instance];[taille du front de Pareto];[temps pris par la première phase];[temps pris par la seconde phase];[somme des temps pris];[nombre d'étiquettes créées];[nombre d'étiquettes ajoutées au front de Pareto]"`.

7.3 `void Writer::writeParetoFrontFile(Instance const&, LSDPF const*)`

`Writer::writeParetoFrontFile` prend en argument une instance et une instance du solveur. Cette méthode crée un nouveau fichier dans le dossier créé lors de la construction de l'instance, avec pour nom `"/results_[identifiant de l'instance, commençant par 0 si inférieur à 10].csv"`. Dans ce fichier, il écrit toutes les étiquettes présentes dans le front de Pareto, en les séparant par des sauts de ligne. Pour chaque étiquette, les composantes de l'étiquettes sont séparées par des points-virgules.

B

Spécifications fonctionnelles des modifications apportées à l'existant

Les modifications apportées se concentrent sur la classe LSDPF, qui implémente la méthode de résolution. Pour permettre de choisir entre différentes implémentations de la seconde phase, on implémente le patron de conception Stratégie afin de pouvoir faire coexister plusieurs implémentations de la deuxième phase de la résolution sans dupliquer l'implémentation de la première phase. Toutes les propriétés et méthodes liées à la deuxième phase sont déplacées hors de LSDPF et on les retrouve dans SndPhaseSequentiel.

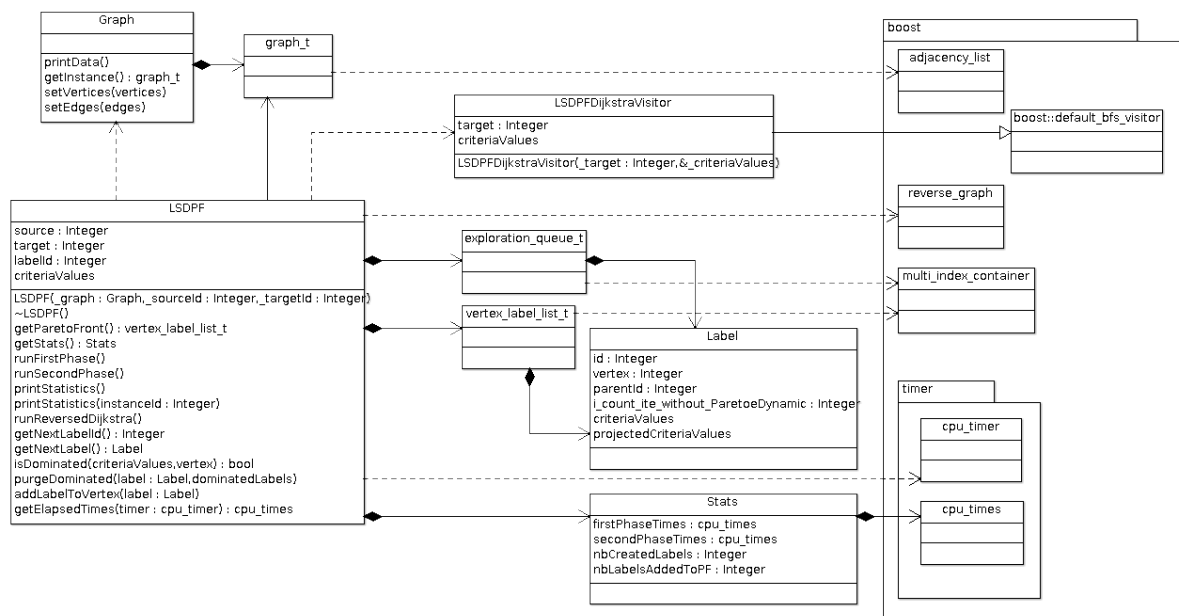


Figure 1 – Classes en relation avec LSDPF, dans l'existant

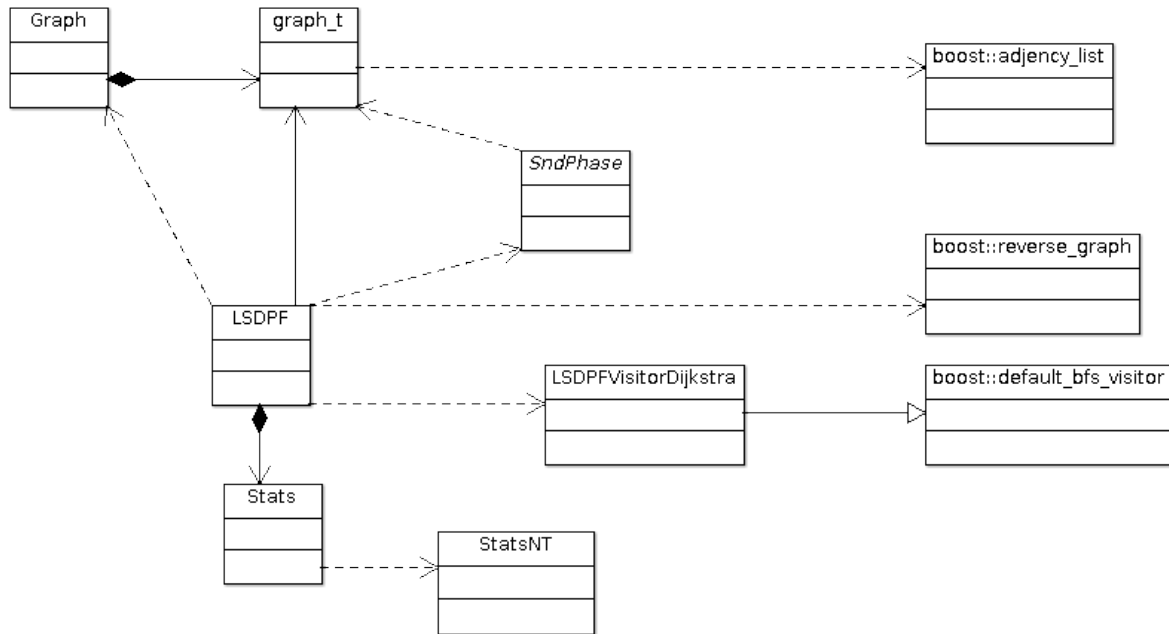


Figure 2 – Classes en relation avec LSDPF, dans la proposition

vertex_label_list_t et exploration_queue_t sont des conteneurs de Boost qui ont été renommés. On constate que LSDPF n'est plus dépendant d'eux dans la proposition. Les dépendances de LSDPF liées à la première phase sont conservées. Les liaisons entre LSDPF et Writer ont été ignorées dans ce diagramme, ainsi que les dépendances liées à la première phase. Une des sections suivantes est dédiée à l'interface SndPhase. Cette interface ne sera pas dans le code source, mais cela n'empêchera pas SndPhaseSequentiel de l'implémenter.

1 Modification de la définition de la classe Label

Les instances de la classe Label existante sont immuables. On souhaite remédier à cela en les rendant mutables, ce qui implique notamment l'ajout d'accesseurs non constants sur presque toutes les propriétés de la classe, et l'ajout d'un opérateur d'affectation.

2 Modification de la définition de la classe LSDPF

La classe LSDPF est modifiée afin de pouvoir choisir entre plusieurs implémentations de la deuxième phase, en faisant de runSecondPhase un template. Toutes les propriétés et méthodes dédiées à la seconde phase sont supprimées.

Les propriétés de LSDPF utilisées par l'ancienne version de la méthode runSecondPhase sont supprimées. Les propriétés de LSDPF subissent les modifications suivantes :

- const graph_t &graph;
- const vertex_t source;
- const vertex_t target;
- std::vector<std::vector<std::vector<double>>> criteriaValues;
- vertex_label_list_t* vertexLabelList; // supprimée. Les étiquettes de chaque nœud seront gérées par l'implémentation de SndPhase
- exploration_queue_t explorationQueue; // supprimée. gérée par l'implémentation de SndPhase
- vertex_label_list_t paretoFront; // ajoutée
- Stats stats;

De nombreuses méthodes privées de LSDPF sont supprimées, il ne reste que celles-ci, dont l'implémentation ne change pas :

- `void runReversedDijkstra(const boost::reverse_graph<graph_t> &reverseGraph, u_int8_t n);`
- `boost::timer::cpu_times getElapsedTimes(const boost::timer::cpu_timer &timer) const;`

Voici la liste des méthodes publiques de LSDPF :

- `LSDPF(const Graph *_graph, int64_t _sourceId, int64_t _targetId);`
- `LSDPF();`
- `const std::vector<std::vector<double>> &getParetoFront() const;`
- `const Stats &getStats() const;`
- `void runFirstPhase();`
- `template<T> void runSecondPhase<T>();`
- `void printStatistics() const; // même implémentation que dans LSDPF`
- `void printStatistics(int64_t instanceId) const; // même implémentation que dans LSDPF`

Les méthodes publiques de LSDPF restent inchangées, sauf `runSecondPhase` et `getParetoFront`. L'implémentation de `getParetoFront` doit nécessairement changer puisqu'elle dépendait de la propriété `vertexLabelList`, et que son type de retour a changé (afin de diminuer la dépendance à Boost au niveau de LSDPF). `runSecondPhase` devient une méthode template et a un paragraphe dédié à sa spécification.

3 Définition de la méthode `LSDPF::runSecondPhase<T>`

La méthode `runSecondPhase` est désormais une méthode template avec comme paramètre une classe implémentant l'interface `SndPhase`. Le corps de la méthode `runSecondPhase<T>` est réduit à mesurer le temps mis à calculer le front de Pareto, et récupérer les statistiques, puisque la responsabilité du calcul du front de Pareto lui a été enlevée. Voici son pseudo code :

- Lancer le chronomètre;
- Créer une instance de la classe `T`;
- Appeler la méthode `run` de cette instance;
- Arrêter le chronomètre;
- Récupérer le front de Pareto, le nombre d'étiquettes créées et le nombre d'étiquettes ajoutées dans le front de Pareto à partir de l'instance de `T`.

4 Définition de l'interface `SndPhase`

Il a été fait mention de classes implémentant l'interface `SndPhase`. Il est important de préciser que l'interface `SndPhase` n'existe pas dans le code source. Mais les classes utilisées en paramètre de template de la méthode `LSDPF::runSecondPhase` doivent avoir les méthodes publiques décrites par l'interface `SndPhase`, puisque ces méthodes seront appelées dans le corps de `LSDPF::runSecondPhase`.

Une classe implémentant l'interface `SndPhase` définit une implémentation de la deuxième phase de la recherche de plus court chemin multi-objectif. Une classe implémentant l'interface `SndPhase` a un constructeur avec les paramètres suivants :

- `const graph_t &_graph, // le graphe`
- `const vertex_t _source,`
- `const vertex_t _target, // le puits`
- `const std::vector<std::vector<std::vector<double>>> &criteriaValues // résultat de la première phase`

Une classe implémentant l'interface `SndPhase` a les méthodes publiques suivantes :

- `void run();`
- `std::vector<std::vector<double>> getParetoFront() const;`

- `int64_t getNbCreatedLabels() const;`
- `int64_t getNbLabelsAddedToPF() const;`

Une implémentation de cette interface s'utilise ainsi : la méthode `run` est appelée, puis les getteurs sont appelées une fois que l'exécution de `run` est terminée.

C

Spécifications fonctionnelles concernant les nouvelles stratégies

Toutes les stratégies suivantes implémentent l'interface fictive `SndPhase` décrite à la fin du chapitre précédent.

1 `SndPhaseSequentiel`

Par la suite, on appellera `SndPhaseSequentiel` SPS.

C'est l'adaptation directe de la stratégie d'origine permettant de résoudre la seconde phase. SPS reprend tous les éléments de l'ancienne classe `LSDPF` qui ne servait qu'à la résolution de la seconde phase.

Durant le développement, il a été découvert que les étiquettes fournies par le prétraitement ne sont pas strictement non-dominées. La seule modification apportée à SPS par rapport à la stratégie originelle est une vérification et une purge des étiquettes dominées lors de l'initialisation du front de Pareto. Cet incident entraînait des étiquettes non strictement non-dominées dans le front de Pareto final, ce qui faussait la taille du front de Pareto.

2 `SndPhaseSequentielT`

Par la suite, on appellera `SndPhaseSequentielT` SPST.

C'est une tentative d'affranchissement de la macro `EXPLORATION_STRATEGY`. La stratégie d'exploratoire est alors définie par le couple d'arguments donné au template. Mis à part cela, c'est exactement la stratégie SPS.

3 Structures utiles pour les stratégies parallèles

3.1 Définition de `Buffer98T`

Le patron de classe `Buffer98T` est un conteneur, qui permet à plusieurs threads de le remplir simultanément. Une fois le remplissage terminé, le contenu du buffer peut être lu. La lecture ne doit être effectuée que par un seul thread, il est donc libre de modifier le contenu du buffer pendant sa lecture. Une fois la lecture terminée, le buffer est déclaré vide, et son contenu sera écrasé lors du prochain remplissage.

`Buffer98T` ne prend comme argument de template que le type d'élément `T` qu'il contient.

Son constructeur demande en argument une capacité. Le nombre d'éléments stockés dans le buffer ne peut pas dépasser cette capacité, et la capacité ne peut pas être modifiée une fois que l'instance est créée.

Cette classe possède les propriétés suivantes :

- `size_t m_capacity;`
- `std::atomic<size_t> m_size;`
- `T * m_content;`

L'ajout simultané d'éléments dans le buffer par plusieurs threads est permis en rendant la variable représentant le nombre d'éléments atomique. Au moment de l'ajout, cette variable est incrémentée de façon atomique, et cette opération retourne le nombre d'éléments avant l'incrément. On a la garanti que le nombre d'éléments avant l'incrément sera toujours différent, puisque l'incrément est atomique, et ce nombre d'éléments avant l'incrément correspond à la position de la case dans laquelle le thread peut écrire sans risque.

Buffer98T est le patron de classe utilisé pour les buffers de tâches et de requêtes eux-mêmes utilisés par les stratégies parallèles.

3.2 LabelArray

LabelArray est un conteneur de Label, pour représenter la liste des étiquettes d'un nœud. Les étiquettes ne sont pas ordonnées dans le conteneur.

Lorsqu'une étiquette est purgée de la liste, elle est seulement invalidée. Lorsqu'une étiquette est insérée dans la liste, s'il y a une étiquette invalidée, l'étiquette invalidée est remplacée par le contenu de la nouvelle étiquette; sinon, l'étiquette est insérée à la fin de la liste.

L'interface de ce conteneur est la suivante :

- `LabelArray();` // constructeur
- `bool isDominated(std::vector<double> const & cv) const;` // vérifie la domination de composantes par les étiquettes de la liste;
- `std::vector<unsigned int> purgeDominated(std::vector<double> const & cv);` // purge les étiquettes dominées par les composantes, et renvoie la liste des positions des étiquettes purgées;
- `unsigned int addLabel(Label const & l);` // insertion d'une étiquette. renvoie la position de l'étiquette dans la structure;
- `Label const & operator[](unsigned int i) const;` // accesseur
- `Label & operator[](unsigned int i);` // accesseur
- `std::vector<std::vector<double>> getFront() const;` // renvoie la liste des composantes des étiquettes qui ne sont pas invalidées

3.3 LabelPosition

LabelPosition est une simple structure représentant la position d'une étiquette au sein d'une liste de LabelArray.

Ses attributs sont :

- `vertex` : identifiant du nœud de l'étiquette;
- `labelIndex` : position de l'étiquette au sein du LabelArray du nœud de l'étiquette.

Les deux attributs sont des entiers naturels. Si `vertex` a pour valeur le plus grand entier naturel pour son type, cela signifie que la structure est invalide. Cela est utile dans le conteneur ExplorationArray pour marquer les LabelPosition purgées.

3.4 ExplorationArray

ExplorationArray est un conteneur de LabelPosition, pour représenter une file d'exploration.

ExplorationArray ne contient pas les composantes des étiquettes présentes dans la file d'exploration. ExplorationArray doit connaître un tableau de LabelArray (en lecture seule), et Exploration ne contient que des références à certaines étiquettes au sein des LabelArray.

L'interface de ce conteneur est la suivante :

- ExplorationArray(LabelArray const * labelLists); // constructeur, pour connaître le tableau de LabelArray;
- std::vector<double> getOutStaticCriterion(std::vector<std::vector<double>> const & out) const; // calcule l'OUT-Criterion, out contient l'arc sortant minimum de chaque nœud;
- std::vector<double> getInStaticCriterion() const; // calcule de l'IN-Criterion;
- void purgeDominated(unsigned int vertex, std::vector<unsigned int> const & indexes); // invalide les LabelPosition représentant des étiquettes qui viennent d'être purgées. Toutes ces étiquettes ont pour nœud celui d'identifiant vertex, et se trouvent dans le LabelArray correspondant aux positions contenues dans indexes;
- void insert(unsigned int vertex, unsigned int labelIndex); // insertion d'une nouvelle étiquette dans la file, qui a pour nœud vertex et qui se trouve à la position labelIndex dans le LabelArray correspondant;
- void consume(unsigned int i); // l'étiquette de LabelPosition en position i dans la file d'exploration est extraite, le LabelPosition en position i est alors invalidé;
- LabelPosition const & operator[](unsigned int i) const; // accesseur
- LabelPosition & operator[](unsigned int i); // accesseur
- bool empty() const; // renvoie "aucune LabelPosition valide n'est stockée"

A cela s'ajoute trois fonctions suivantes pour identifier des étiquettes à extraire de la file d'exploration, dans le cas où on ne prend en compte que l'OUT-Criterion, ou l'IN-Criterion, ou les deux. L est l'OUT-Criterion, M est l'IN-Criterion, et in contient l'arc entrant minimum de chaque nœud. n est le nombre maximum d'étiquettes à identifier pour l'extraction.

```
std::vector<unsigned int> getPositionsPerOutStaticCriterion(unsigned int n,
    std::vector<double> const & L) const;
std::vector<unsigned int> getPositionsPerInStaticCriterion(unsigned int n,
    std::vector<double> const & M,
    std::vector<std::vector<double>> const & in) const;
std::vector<unsigned int> getPositionsPerInOrOutStaticCriterion(unsigned int n,
    std::vector<double> const & L,
    std::vector<double> const & M,
    std::vector<std::vector<double>> const & in) const;
```

3.5 Description des classes de préfixe Worker

Dans l'algorithme décrit dans [2], chaque thread est responsable d'une partie des nœuds, a sa propre file d'exploration, et doit écrire et lire dans des buffers de tâches et de requêtes et maintenir les listes d'étiquettes des nœuds qu'il a sous sa responsabilité.

Les opérations sur la file d'exploration, sur les buffers et sur les listes d'étiquettes sont réalisées par un objet Worker. Le Worker possède la file d'exploration, des références aux buffers et aux listes d'étiquettes de chaque nœuds, ainsi que le numéro du thread auquel il est associé et une référence au graphe, afin d'accomplir sa mission.

Chaque thread n'a donc qu'à créer son objet Worker, lui fournir tous les paramètres nécessaires, puis respecter le flot d'exécution de l'algorithme de [2] en appelant les méthodes du Worker correspondant à chaque étape de l'algorithme.

Les deux classes ayant pour préfixe Worker sont Worker98Amd et Worker98A4md.

Elles partagent l'interface suivante («minimum» et «plus petit» sont au sens de l'ordre lexicographique) :

- int64_t getNbCreatedLabels() const; // pour les informations liées à la résolution.

- `int64_t getNbLabelsAddedToPF() const; // pour les informations liées à la résolution.`
- `bool readTaskBuffer(); // lecture du buffer de tâches de même numéro que le thread associé au Worker, et génération des requêtes, et placement de chaque requête dans le buffer du thread responsable du nœuds associé à la requête; le résultat est le booléen valant : "aucun ajout de requête dans un buffer n'a échoué (à cause de leur capacité limitée)"`
- `unsigned int writeTaskBuffersOutStaticCriterion(unsigned int n, std::vector<double> const & L); // extraction d'au plus n étiquettes de la file d'exploration vérifiant l'OUT-Criterion (étiquette plus petite que L), et ajout d'autant de tâches dans les buffers de tâches de chaque thread; le résultat est le nombre de tâches ajoutées.`
- `unsigned int writeTaskBuffersInStaticCriterion(unsigned int n, std::vector<double> const & M, std::vector<std::vector<double>> const & in); // extraction d'au plus n étiquettes de la file d'exploration vérifiant l'IN-Criterion (somme d'une étiquette avec le plus petit arc entrant plus petite que M), et ajout d'autant de tâches dans les buffers de tâches de chaque thread; le résultat est le nombre de tâches ajoutées.`
- `unsigned int writeTaskBuffersInOrOutStaticCriterion(unsigned int n, std::vector<double> const & L, std::vector<double> const & M, std::vector<std::vector<double>> const & in); // extraction d'au plus n étiquettes de la file d'exploration vérifiant l'OUT-Criterion ou l'IN-Criterion, et ajout d'autant de tâches dans les buffers de tâches de chaque thread; le résultat est le nombre de tâches ajoutées.`
- `void readRequestBuffer(); // lecture du buffer de requêtes du thread, et mise à jour de la file d'exploration et des listes des étiquettes des nœuds.`
- `std::vector<double> getOutStaticCriterion(std::vector<std::vector<double>> const & out) const; // calcul de l'OUT-Criterion partiel, c'est-à-dire le minimum parmi les sommes d'une étiquette dans la file d'exploration avec le poids de l'arc sortant minimal du nœud associé à l'étiquette.`
- `std::vector<double> getInStaticCriterion() const; // calcul de l'IN-Criterion partiel, c'est-à-dire le minimum parmi les étiquettes dans la file d'exploration.`

4 Stratégie parallèle : SPPT3md

La stratégie parallèle SPPT3md a une boucle principale dont les étapes sont décrites dans la section "Boucle principale des stratégies parallèles et condition d'arrêt" du chapitre "Analyse et conception". Comme décrit dans cette section, chaque étape doit être réalisée par tous les threads avant qu'ils ne puissent passer à l'étape suivante.

Les structures mentionnées dans la section "Boucle principale des stratégies parallèles et condition d'arrêt" du chapitre "Analyse et conception" sont les structures suivantes :

- les buffers de tâches sont des `Buffer98T<LabelPosition>` ;
- les buffers de requêtes sont des `Buffer98T<Label>` ;
- les listes d'étiquettes des nœuds sont des `LabelArray` ;
- les files d'exploration sont des `ExplorationArray`.

La recherche est terminée lorsqu'aucun thread n'a inséré de tâches dans les buffers lors de l'étape correspondante.

Les nœuds sont répartis entre les threads ainsi (on note T le nombre de threads) : le puits est le seul nœud dont le thread 0 est responsable ; le nœud n différent du puits est sous la responsabilité du thread $n \% (T-1) + 1$.

Lors de l'extraction des tâches depuis les files d'exploration, l'IN- et l'OUT-Criteria sont utilisés.

4.1 Spécificités du worker associé à SPPT3md

Les listes d'étiquettes de chaque nœud sont des LabelArray. La file d'exploration est un ExplorationArray.

SPPT3md réalise des mises à jour dynamiques, et c'est dans une méthode du worker qu'elles se produisent, pendant l'étape d'insertion des requêtes dans les buffers de requêtes.

Le worker sait quel thread est responsable de chaque nœud.

5 Stratégie parallèle : SPPT4asyncmd

La stratégie parallèle SPPT4asyncmd a une boucle principale dont les étapes sont décrites dans la section "Boucle principale des stratégies parallèles et condition d'arrêt" du chapitre "Analyse et conception". Comme décrit dans cette section, chaque étape doit être réalisée par tous les threads avant qu'ils ne puissent passer à l'étape suivante.

Les structures mentionnées dans la section "Boucle principale des stratégies parallèles et condition d'arrêt" du chapitre "Analyse et conception" sont les structures suivantes :

- les buffers de tâches sont des Buffer98T<LabelPosition> ;
- les buffers de requêtes sont des Buffer98T<Label> ;
- les listes d'étiquettes des nœuds sont des LabelArray ;
- les files d'exploration sont des ExplorationArray.

La recherche est terminée lorsqu'aucun thread n'a inséré de tâches dans les buffers lors de l'étape correspondante.

Les nœuds sont répartis entre les threads ainsi (on note T-1 le nombre de threads réalisant la boucle principale, T étant le nombre total de threads) : le puits est sous la responsabilité d'un thread à part ; les autres nœuds n sont respectivement sous la responsabilité du thread $n \% (T-1)$.

Le thread qui a la responsabilité du puits ne participe pas à l'exécution de la boucle principale comme les autres. Il ne fait que recevoir les requêtes d'ajout d'étiquettes dans le front de Pareto et mettre à jour ce front.

Les requêtes d'ajout d'étiquettes sont stockées dans une file supportant plusieurs producteurs et plusieurs consommateurs concurrents. Le thread dédié au puits est l'unique consommateur de cette file, et les autres threads sont tous des producteurs.

Le front de Pareto est représenté par la structure StaticCriteriaValuesContainer, décrite dans une prochaine sous-section. Tous les threads ont accès à l'instance de cette structure, afin de vérifier la domination d'étiquettes, ou pour simplement mettre à jour le front dans le cas du thread dédié (ce qui inclut des vérifications de domination).

La structure StaticCriteriaValuesContainer a comme faiblesse que la taille maximale du front de Pareto doit être définie au début de la recherche, et ne peut pas être modifiée. Cela impose la gestion de l'erreur de dépassement de la capacité du front de Pareto, et de l'arrêt de la recherche dans ce cas de figure. Lorsque la recherche est interrompue ainsi, on écrit dans la sortie standard : "Le front de Pareto a dépassé sa taille maximale", et l'exécution du programme s'arrête prématurément.

5.1 Spécificités du worker associé à SPPT4asyncmd

Les listes d'étiquettes de chaque nœud sont des LabelArray. La file d'exploration est un ExplorationArray.

SPPT3md réalise des mises à jour dynamiques, et c'est dans une méthode du worker qu'elles se produisent, pendant l'étape d'insertion des étiquettes dans les listes d'étiquettes des nœuds, après la vérification de non-domination entre les requêtes partageant le même nœud.

Le worker sait quel thread est responsable de chaque nœud.

Le worker possède une référence à la file des requêtes du puits, afin de pouvoir y insérer des requêtes, et une référence à l'instance de StaticCriteriaValuesContainer.

Le thread dédié ne possède pas d'instance de la classe worker spécifique à SPPT4asyncmd.

5.2 StaticCriteriaValuesContainer

StaticCriteriaValuesContainer est un conteneur maladroitement nommé qui représente la liste des étiquettes d'un nœud. Cette structure n'est utilisée que pour le puits.

Ce conteneur résout le problème de synchronisation entre la lecture du front de Pareto et sa mise à jour, problème qui est causé par le fait que le thread dédié au puits ne suit pas les étapes de la boucle principale avec les autres threads.

La capacité de la structure StaticCriteriaValuesContainer, c'est à dire la taille maximale du front de Pareto, est définie par un argument du constructeur.

StaticCriteriaValuesContainer stocke dans un tableau les composantes des étiquettes du front de Pareto. Un autre tableau permet de marquer les cases du premier tableau qui ont été purgées et dont le contenu n'a pas encore été remplacé.

Pour éviter qu'une écriture de composantes d'une étiquette ne se produisent au milieu d'une lecture, et que les composantes lues se mettent à être celles d'une étiquette qui n'a jamais existé, risquant ainsi de tromper la vérification de non-domination des étiquettes et mettant en péril l'exactitude de la stratégie, on choisit de faire ainsi :

- lorsque les composantes d'une étiquette sont marquées par le thread dédié au puits (le marquage n'altérant pas les composantes mais le contenu du second tableau dit de marquage), l'entier naturel utilisé est le numéro de l'itération de la boucle principale en cours. 0 signifie que l'étiquette n'est pas marquée et que ses composantes peuvent être lues ;
- lorsqu'un thread réalise une consultation en lecture seule, et que l'étiquette qu'il consulte est marquée, il l'ignore ; sinon, il commence sa lecture. Une écriture des composantes qu'il commence à lire ne peut avoir lieu même si l'étiquette devient marquée, grâce au point suivante :
- lorsque le thread dédié au puits tente d'insérer les composantes d'une nouvelle étiquette non dominée, il écrase les composantes d'une étiquette marquée seulement sur le numéro de l'itération de la boucle principale en cours est strictement supérieur à celui de l'itération pendant laquelle le marquage a eu lieu.

L'interface publique de StaticCriteriaValuesContainer est la suivante :

- `StaticCriteriaValuesContainer(unsigned int capacity, unsigned int nb_criteria, std::atomic<unsigned int> const & currentPhase);` // constructeur, précise la capacité en nombre d'étiquettes, le nombre de composantes par étiquette, et une référence au numéro de l'itération courante de la boucle principale ;
- `unsigned int capacity() const;` // renvoie la taille maximale de la liste d'étiquettes ;
- `unsigned int size() const;` // renvoie la taille de la liste d'étiquettes, en comptant les étiquettes marquées ;
- `double const * raw() const;` // renvoie le tableau des composantes
- `unsigned int const * rawIsPurged() const;` // renvoie le tableau de marquage
- `bool isPurged(unsigned int i) const;` // renvoie "l'étiquette à la position i est purgée (=marquée)";
- `std::vector<double> operator[](unsigned int i) const;` // renvoie les composantes de l'étiquette en position i sous la forme d'une vector ;
- `bool isDominated(std::vector<double> const & cv) const;` // renvoie "l'étiquette qui a les composantes données en entrée est dominée par une étiquette de la liste";
- `bool isDominated(CriteriaValues const & cv) const;` // renvoie "l'étiquette qui a les composantes données en entrée est dominée par une étiquette de la liste";

- `bool isDominatedIgnoreEquality(std::vector<double> const & cv) const; // renvoie "l'étiquette qui a les composantes données en entrée est dominée strictement par une étiquette de la liste";`
- `bool purgeAndIsDominatedIgnoreEquality(CriteriaValues const & cv, unsigned int & rewrite_position) const; // renvoie "l'étiquette qui a les composantes données en entrée est dominée par une étiquette de la liste"; si l'étiquette était non-dominée, alors les étiquettes dominées par celui-ci ont été purgées (=marquées), et dans ce cas, si la valeur de rewrite_position est différente de la capacité, alors les composantes de l'étiquette peuvent être écrites par dessus celles de l'étiquette purgée précédemment à la position rewrite_position. Si l'étiquette était non-dominée, mais que rewrite_position vaut la capacité, il n'y a pas de position à écraser, donc il faudra ajouter l'étiquette à la fin de la partie utilisée du tableau; cette méthode n'insère pas l'étiquette;`
- `void purge(CriteriaValues const & cv); // purge (= marque) les étiquettes dominées par les composantes données en entrée;`
- `bool add(CriteriaValues const & cv); // ajoute les composantes d'une nouvelle étiquette à la fin de la partie utilisée du tableau des composates; renvoie "l'ajout est un succès", c'est-à-dire "la capacité n'a pas été dépassée";`
- `void rewrite(CriteriaValues const & cv, unsigned int i); // écrase le contenu de la position i des tableaux des composantes et de marquage, pour y placer la nouvelle étiquette;`
- `std::vector<std::vector<double>> getFront() const; // renvoie la liste des composantes des étiquettes non marquées.`

D

Spécifications non fonctionnelles

1 Contraintes de développement et conception

Le langage de programmation adopté est le C++, puisque c'est le langage dans lequel l'existant est écrit.

L'utilisation d'une carte graphique sera envisagée comme piste exploratoire, pour :

- Réaliser toute la seconde phase ;
- Réaliser seulement une partie du traitement.

Il faudra alors se procurer une machine disposant d'une telle carte.

OpenMP et MPI peuvent être utilisés également.

L'existant utilise une partie de la bibliothèque Boost, on continuera de les utiliser, et on s'autorise à étendre les dépendances de l'existant à d'autres parties de Boost.

2 Contraintes de fonctionnement et d'exploitation

2.1 Modes de fonctionnement

Il est possible de configurer la stratégie exploratoire de la méthode de résolution du programme en modifiant la macro `EXPLORATION_STRATEGY` définie dans `Const.h`. On peut choisir entre `ES_DIJKSTRA` (retirer la plus petite étiquette selon l'ordre lexicographique), `ES_A_STAR` (retirer la plus petite projection d'une étiquette selon l'ordre lexicographique) et `ES_LC` (retirer l'étiquette dont la somme des poids est la plus petite ; LC comme Linear Combination).

Choisir `ES_A_STAR` rend la méthode label-correcting, choisir `ES_DIJKSTRA` ou `ES_LC` rend la méthode label-setting, si on omet la mise à jour dynamique des étiquettes du puits.

2.2 Contrôlabilité

On peut suivre l'exécution d'un traitement en lisant la sortie standard. Ce qui est écrit dans la sortie standard est défini dans la sous-section consacrée aux sorties de la section *Spécifications fonctionnelles de l'existant*.

E

Guide d'installation

1 Environnement

Le système d'exploitation utilisé est la distribution Linux Mint 19.2 Tina. Le compilateur est gcc v7.4.0.

2 Librairies

2.1 Boost

Voici les étapes à suivre pour installer les composants et les headers de Boost :

- visiter la page <https://www.boost.org/users/download> ;
- télécharger boost_1_72_0.tar.gz ou une version plus récente ;
- lire le contenu d'INSTALL ;
- ouvrir boost_1_72_0/index.html ;
- lire la section Getting Started, et le Getting Started guide ;

Pour la dernière version du programme décrite par ce document, vous avez besoin des composants filesystem, thread et unit_test_framework. Pour la version originelle du programme, vous aurez également besoin des composants system, timer et chrono.

- `$./bootstrap.sh -help`
- `$./bootstrap.sh --includedir=/usr --with-libraries=filesystem,thread,unit_test_framework`
- `$./b2 install`
- déplacer éventuellement les headers ;
- déplacer le fichier libboost_unit_test_framework.so.1.72.0 dans /lib/, ou dans un dossier équivalent ;
- déplacer les fichiers libboost_unit_test_framework.so* dans le dossier lib/ du projet ;
- déplacer les fichiers *.a dans le dossier lib/ du projet ;

Les composants filesystem, thread, system, timer et chrono sont des librairies statiques. Le composant unit_test_framework est une librairie dynamique.

2.2 Osmium

La librairie Osmium a un dépôt ici : <https://github.com/osmcode/libosmium/>.

Un seul fichier d'entête d'Osmium est nécessaire. Sa version la plus récente se trouve ici : <https://github.com/osmcode/libosmium/blob/master/include/osmium/osm/location.hpp>.
Une copie de ce fichier se trouve dans le projet, dans `include/`.
Il suffit de créer le répertoire `/usr/include/osmium/osm/` puis d'y copier `location.hpp`.

2.3 Rigport::MPMCQueue

Cette structure a pour dépôt : <https://github.com/rigtorp/MPMCQueue/>.
Un seul fichier d'entête de ce dépôt est nécessaire. Sa version la plus récente se trouve ici : <https://github.com/rigtorp/MPMCQueue/blob/master/include/rigtorp/MPMCQueue.h>.
Une copie de ce fichier se trouve dans le projet, dans `include/`.
Il suffit de créer le répertoire `/usr/include/rigtorp/` puis d'y copier `MPMCQueue.h`.

3 Doxygen

Le guide suivi pour l'installation d'une interface graphique pour Doxygen est le suivant : <https://franckh.developpez.com/tutoriels/outils/doxygen/>

4 Python

Si vous choisissez d'utiliser les scripts en python, vous aurez besoin de `scipy`, `matplotlib`...
Les scripts fonctionnent avec `python2.7`, aucune vérification n'a été faite pour savoir s'ils fonctionnent en `3.7`.

F

Guide d'utilisation

1 Utilisation courante

Le programme permet de résoudre des instances d'un graphe, c'est-à-dire de calculer l'ensemble des plus courts chemins entre des paires de nœuds.

1.1 Configuration

Configurer les paramètres utilisées pour la résolution des instances du graphe revient à choisir la valeur des macros des variables globales dans `include/Const.h`.

Le nombre d'objectifs du graphe dont les instances sont à résoudre doit être la valeur de la macro `NB_CRITERIA`.

La variable globale `firstPhaseWeights` doit des vecteurs de taille `NB_CRITERIA`, et les vecteurs contenus représentent des barycentres. Au moins tous les barycentres dont un coefficient vaut 1 et tous les autres valent 0 doivent être présents dans `firstPhaseWeights`.

La variable globale `secondPhaseWeightIndexes` contient les positions au sein de `firstPhaseWeights` des barycentres utilisés lors de la seconde phase. Par défaut, tous les barycentres dont un coefficient vaut 1 et tous les autres valent 0 doivent être référencés dans `secondPhaseWeightIndexes`.

```
#define NB_CRITERIA 2
const std::vector<std::vector<double>> firstPhaseWeights =
{{1,0},{0,1}};
const std::vector<uint8_t> secondPhaseWeightIndexes = {0,1};
```

```
#define NB_CRITERIA 3
const std::vector<std::vector<double>> firstPhaseWeights =
{{1,0,0},{0,1,0},{0,0,1}};
const std::vector<uint8_t> secondPhaseWeightIndexes = {0,1,2};
```

La macro `EXPLORATION_STRATEGY` peut prendre les valeurs `ES_DJKSTRA`, `ES_A_STAR`, ou `ES_LC`. Sa valeur définit la stratégie d'exploration employée, mais n'est pas suffisante. Cette macro n'est utilisée que par la stratégie séquentielle originelle, et par la classe `Writer`.

La macro `METHOD` définit la classe qui sera instanciée afin de résoudre la seconde phase. Cette macro n'est utilisée que dans `src/main.cpp`. Les valeurs prises par cette macro sont spécifiées dans les sous sections suivantes.

La macro `MAX_TASKS_EXTRACTED_PER_THREAD` est utilisée par les deux stratégies parallèles pour résoudre la seconde phase. Elle sert à dimensionner les buffers de tâches pour ces stratégies.

Les macro commençant par `SPPT3` et `SPPT4` sont utilisées spécifiquement par chaque stratégie parallèle, et sont expliquées dans les sous-sections correspondantes.

1.1.1 Stratégie séquentielle originelle

La valeur à donner à la macro `METHOD` pour utiliser la stratégie séquentielle originelle est `SndPhaseSequentiel`.

La stratégie exploratoire utilisée dépend de la valeur de la macro `EXPLORATION_STRATEGY`.

1.1.2 Stratégie séquentielle avec template

La valeur à donner à la macro `METHOD` pour utiliser la stratégie séquentielle avec template est `SndPhaseSequentielT<ExplorQueue, StratExplor>`, où `ExplorQueue` et `StratExplor` doivent être remplacés par un des couples suivants :

- `exploration_queue_DIJKSTRA_t`, `Label::ByCriteriaValues`
- `exploration_queue_ES_A_STAR_t`, `Label::ByProjectedCriteriaValues`
- `exploration_queue_ES_LC_t`, `Label::ByLCValue`

`ExplorQueue` étant la structure utilisée pour la file d'exploration, et `StratExplor` le type utilisé lors de l'extraction de la prochaine étiquette.

Il est bien vu de donner une valeur cohérente à la macro `EXPLORATION_STRATEGY`.

1.1.3 Stratégie parallèle avec thread dédié suivant le cycle

La stratégie parallèle avec thread dédié [au puits] suivant le cycle est également appelée `SPPT3md`.

La valeur à donner à la macro `METHOD` pour utiliser cette stratégie est `SPPT3md<NbThreads>`, où `NbThreads` doit être remplacé par le nombre de threads à utiliser. Ce nombre doit être supérieur ou égal à 2.

La stratégie `SPPT3md` dépend de la macro `SPPT3_WELL_REQUEST_BUFFER_SIZE_FACTOR` qui est un facteur pour la taille du buffer de requête du puits.

Il est bien vu de donner à la macro `EXPLORATION_STRATEGY` la valeur `ES_DIJKSTRA`, puisque c'est la stratégie exploratoire utilisée par `SPPT3md`.

1.1.4 Stratégie parallèle avec thread dédié en dehors du cycle

La stratégie parallèle avec thread dédié [au puits] en dehors du cycle est également appelée `SPPT4asyncmd`.

La valeur à donner à la macro `METHOD` pour utiliser cette stratégie est `SPPT4asyncmd<NbThreads>`, où `NbThreads` doit être remplacé par le nombre de threads à utiliser. Ce nombre doit être supérieur ou égal à 2.

La stratégie `SPPT4asyncmd` dépend des macro `SPPT4_PARETO_FRONT_CAPACITY` et `SPPT4_QUEUE_SIZE`. `SPPT4_PARETO_FRONT_CAPACITY` représente la capacité du front de Pareto, et si elle est dépassée la seconde phase échoue. `SPPT4_QUEUE_SIZE` est la taille de la file des requêtes d'ajout d'étiquettes pour le puits.

Il est bien vu de donner à la macro `EXPLORATION_STRATEGY` la valeur `ES_DIJKSTRA`, puisque c'est la stratégie exploratoire utilisée par `SPPT4asyncmd`.

1.2 Lancement du programme

Le programme prend 5 arguments lors de son appel :

- chemin du fichier définissant les nœuds du graphe ;
- chemin du fichier définissant les arcs du graphe ;
- chemin du fichier définissant les instances du graphe ;
- chemin du dossier dans lequel les résultats seront écrits ;
- le nombre de générations entre deux mises à jour dynamiques du front de Pareto.

Le nombre de générations doit être un entier naturel. Donner une valeur nulle à ce paramètre est déconseillé.

Pour les stratégies séquentielles, une valeur nulle signifie que toutes les générations d'étiquettes font l'objet de mises à jour dynamiques. Cela revient à donner la valeur 1 à ce paramètre, bien que les temps de résolution changent significativement lorsque ce paramètre passe de 0 à 1.

Pour les stratégies parallèles, une valeur nulle signifie qu'aucune mise à jour dynamique n'aura lieu.

1.3 Sortie du programme

Le chemin vers le dossier dans lequel sont écrits les résultats est un paramètre donné lors du lancement du programme. Dans ce dossier, un sous-dossier dont le nom est la date actuelle à la seconde près est créé. Dans ce sous-dossier, les fichiers suivants sont créés :

- results.csv : contient un entête avec les barycentres utilisés par le prétraitement, les barycentres utilisés par la seconde phase, la stratégie exploratoire utilisée, le nombre de générations entre deux mises à jour dynamiques, puis un bilan de la résolution de chaque instance ;
- results_.csv : un par instance résolue, est remplacé par l'identifiant de l'instance ; contient le front de Pareto qui est la solution de l'instance.

Le programme écrit également dans la sortie standard entre chaque traitement d'une instance pour permettre le suivi de l'avancement du traitement de la liste d'instances du graphe.

2 Benchmark

Un certain nombre de scripts en python sont présents dans solution/benchmark/. Ils permettent de réaliser des accès et des modifications sur une base de données Sqlite. Cette base de données permet de mémoriser des commandes, c'est-à-dire le souhait de résoudre les instances d'un graphe avec une certaine méthode sous une certaine configuration.

Voici la marche à suivre pour utiliser ces scripts :

- depuis la console, atteindre le dossier solution/ ;
- exécuter le script ./initbenchmark.sh, qui initialise la base de données ;
- exécuter le script ./runbenchmark.sh : tant qu'il reste des instances à résoudre, le programme est éventuellement compilé pour correspondre à une certaine configuration, puis le programme est lancé ;
- lorsque le programme termine correctement, les résultats calculés et les temps de résolution sont stockés dans une base de données ;
- interrompre ./runbenchmark.sh ne fait pas perdre tout le travail réalisé. ./runbenchmark.sh peut être relancé, et le programme reprendra sa dernière tâche ;
- quand il n'y a plus d'instances à résoudre, ./runbenchmark.sh s'arrête ;
- on utilise ensuite les scripts en python contenues dans solution/benchmark/ pour réaliser des traitements sur les données recueillies, notamment des scripts pour tracer des graphes. Un script important est showParetoFrontError.py, qui permet de détecter les incohérences dans les tailles de fronts de Pareto trouvées.

Lorsque le programme termine trop rapidement, il pourrait être lancé plusieurs fois dans la même seconde. Cela produira un plantage, puisque le programme tentera plusieurs fois de créer le même dossier (le nom du dossier est la date courante à la seconde près). Pour éviter cela, il y a deux options :

- utiliser `./runbenchmark.sh`, sans argument. Une pause d'une seconde sera effectuée entre chaque lancement du programme ;
- utiliser `./runbenchmark.sh 1`. Aucune pause ne sera effectuée, mais le sous-dossier de résultats sera supprimé systématiquement après l'assimilation des informations pertinentes dans la base de données.

La terminaison rapide du programme a lieu notamment avec les graphes rcsp.

3 Tests unitaires

Les tests unitaires se trouvent dans le dossier `unittest/` du projet. Il est fait usage du composant `unit_test_framework` de la librairie Boost.

3.1 Exécution des tests

Placez vous dans le dossier `unittest/` du projet et lancer l'exécutable `./cleanAndRun.sh`. Plusieurs exécutables sont alors compilés, puis exécutés.

Ils signalent la présence ou non d'erreurs dans les tests.

3.2 Ajouter un test unitaire

Pour ajouter des tests unitaires, il y a deux options :

- les ajouter dans un fichier existant de `unittest/src/` ;
- les ajouter dans un nouveau fichier de `unittest/src/` ;

Le premier cas est simple tant qu'aucune nouvelle dépendance n'est ajoutée.

Dans le deuxième cas, il faut réaliser des modifications à `unittest/Makefile` :

- ajouter le nom de l'exécutable à compiler sur la ligne commençant par `build` ;
- ajouter les lignes relatives à la création de l'exécutable à partir des fichiers `.o` et des librairies dans la section `# bin` ;
- ajouter les lignes relatives à la création des fichiers `.o` nécessaires à la compilation de l'exécutable.

1 Tests unitaires

Les deux fonctions `template dominate` et `doesLDominateR` de `dominate.h` sont testées dans 5 cas, avec des séquences de taille 2, pour un seul couple de types pour les premiers arguments alors que le programme utilise ce template avec deux couples.

- les deux séquences à comparer ne se dominent pas ;
- idem en inversant les deux séquences ;
- les deux séquences sont les mêmes, la première domine donc la seconde ;
- la première séquence domine strictement la seconde ;
- la deuxième séquence domine strictement la première.

La structure `LabelArray` est testée dans un seul scénario, l'ajout de trois étiquettes non-dominées entre elles.

La structure `StaticCriteriaValuesContainer` est testée dans quatre scénarios.

- lorsque le conteneur est vide ;
- lorsqu'un y ajoute deux éléments non-dominées entre eux ;
- la rencontre d'un élément égale à l'étiquette servant à la purge ;
- l'écrasement d'un élément purgé.

2 Vérification de la correction des stratégies

Toutes les stratégies doivent être exactes. Par conséquent, pour une instance donnée, quelque soit la stratégie utilisée et sa configuration, le front de Pareto trouvé doit toujours avoir la même taille.

Seule la taille du front de Pareto et non son contenu sera vérifiée.

Les graphes `rcsp5`, `rcsp6`, `rcsp7`, `rcsp8`, `rcsp13`, `rcsp14`, `rcsp15`, `rcsp16`, `rcsp21`, `rcsp22`, `rcsp23` et `rcsp24` ont été utilisées uniquement pour vérifier l'égalité des tailles de front de Pareto trouvée par instance, pour toutes les stratégies et leurs configurations.

Les instances de ces graphes ont l'intérêt d'être petites, donc le temps nécessaire pour les résoudre est ridiculement faible.

Les configurations testées sont les mêmes que celles détaillées dans la section suivante.

Ces graphes ont été mis à disposition avec l'existant, et sont disponibles dans trois versions différentes : avec 3 objectifs, 5 objectifs et 11 objectifs. Les trois versions sont utilisées.

Les temps de résolution des instances de ces graphes n'ont pas été analysés.

3 Mesure des temps de résolution

La machine utilisée pour la mesure des temps de résolution est composée d'un processeur Quad Core Intel Core i5-9300H, supportant 8 threads, avec 8 Go de RAM.

Lors de chaque résolution d'instance, on mesure les temps pris pour chaque phase, le nombre d'étiquettes créées, le nombre d'étiquettes ajoutées au front de Pareto, et la taille du front de Pareto final.

Pour vérifier qu'une méthode donne le bon résultat, on se contente de vérifier la taille du front de Pareto obtenu.

On ne mesurera les temps de résolution d'instances que pour des graphes avec deux objectifs. Les graphes qui serviront pour les mesures des temps de résolutions sont BAY, berlin, NW, NY et paris.

La durée digne d'intérêt est celle du temps pris par la résolution de la seconde phase en fonction de l'instance et de la stratégie.

Les composants des combinaisons linéaires utilisés lors de la phase de prétraitement seront les mêmes pour toutes les stratégies et leurs configurations : tous les barycentres dont un coefficient vaut 1 et dont les autres valent 0.

3.1 Stratégies séquentielles

La stratégie exploratoire choisie sera d'extraire de la file d'exploration les petites étiquettes au sens de l'ordre lexicographique, puisque c'est la seule stratégie exploratoire supportée par les stratégies parallèles avec lesquelles on souhaite réaliser la comparaison.

Si on omet la stratégie exploratoire, l'unique paramètre des deux stratégies séquentielles est `ITERATION_LABEL_MAJ_DYN_PF`, c'est-à-dire le nombre de générations entre deux mises à jour dynamiques du front de Pareto.

Les valeurs qu'on utilisera pour `ITERATION_LABEL_MAJ_DYN_PF` sont 1, 15, 60 et 150. On ajoute également la valeur 0, puisque les stratégies séquentielles supportent cette valeur du paramètre, et qu'il y a une différence significative dans les temps de résolution entre la valeur 0 et la valeur 1, malgré qu'il semblerait à première vue que le comportement des méthodes soit le même avec ces deux valeurs.

Après la réalisation des mesures pour les stratégies séquentielles, la meilleure d'entre elles sera déclarée stratégie séquentielle de référence et servira à la comparaison avec les stratégies parallèles.

3.2 Stratégies parallèles

Comme écrit précédemment, la seule stratégie exploratoire disponible pour les stratégies parallèles est d'extraire les plus petites étiquettes au sens de l'ordre lexicographique de la file d'exploration.

Les deux paramètres principaux des deux stratégies parallèles sont `ITERATION_LABEL_MAJ_DYN_PF`, défini dans la sous-section précédente, et le nombre de threads utilisés.

Les deux stratégies parallèles ont deux contraintes sur ces paramètres : `ITERATION_LABEL_MAJ_DYN_PF` doit être supérieur strictement à 0, sans quoi aucune mise à jour dynamiques n'aura lieu ; et le nombre de threads doit être au moins égal à deux.

Sachant que la machine sur laquelle sera réalisée les mesures supporte huit threads, on choisit d'utiliser deux, quatre, huit et seize threads, dans le but de tracer des courbes d'évolution de l'accélération lorsque le nombre de thread augmente.

On utilisera la valeur d'`ITERATION_LABEL_MAJ_DYN_PF` qui donnait les meilleurs temps pour la meilleure version séquentielle.

Une fois que le nombre de threads qui donne les meilleurs temps dans ces conditions est établi, on fait varier ITERATION_LABEL_MAJ_DYN_PF en lui donnant les valeurs 1, 15, 60 et 150.

4 Résultats

Dans cette partie sont décrits les résultats agrégés obtenus.
La seule durée exploitée est le temps de résolution de la seconde phase.

4.1 Comparaison des versions séquentielles

Dans les tableaux suivants, P signifie ITERATION_LABEL_MAJ_DYN_PF, TM signifie temps moyen pris par la seconde phase (en millisecondes), Acc signifie moyenne des accélérations par instance.

Voici les résultats pour la stratégie SPS, la moyenne de l'accélération étant calculée en comparant les temps de résolution de la seconde phase par instance avec ceux de la configuration où ITERATION_LABEL_MAJ_DYN_PF vaut 15 :

SPS	BAY		berlin		NW		NY		paris	
P	TM	Acc	TM	Acc	TM	Acc	TM	Acc	TM	Acc
0	6229	0.33	2894	0.20	6793	0.23	15810	0.27	238	0.69
1	653	0.96	379	0.97	830	0.96	2054	1.32	45	1.68
15	609	1.0	353	1.0	792	1.0	2579	1.0	73	1.0
60	1075	0.56	403	0.80	1375	0.57	2287	1.07	97	0.69
150	1297	0.46	871	0.32	1513	0.51	2876	0.78	208	0.44

Voici les résultats pour la stratégie SPST, la moyenne de l'accélération étant calculée en comparant les temps de résolution de la seconde phase par instance avec ceux de la configuration où ITERATION_LABEL_MAJ_DYN_PF vaut 15 :

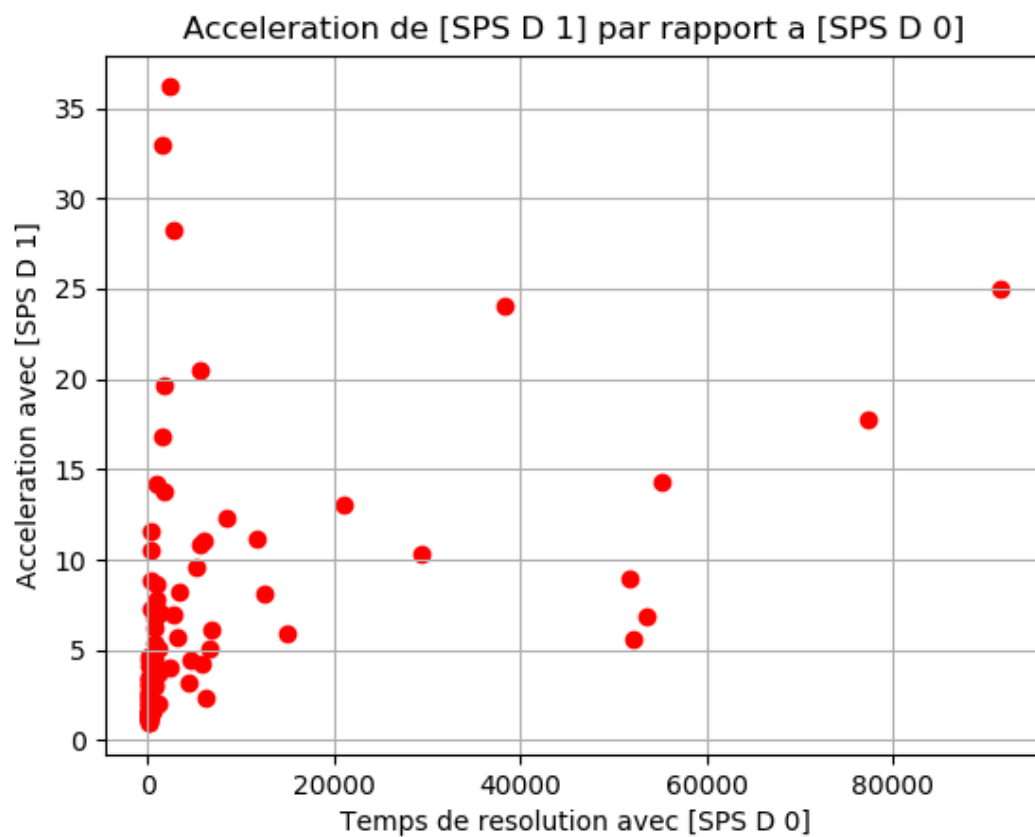
SPST	BAY		berlin		NW		NY		paris	
P	TM	Acc	TM	Acc	TM	Acc	TM	Acc	TM	Acc
0	1096	0.77	499	0.74	1399	0.57	2164	1.26	45	1.67
1	655	1.28	381	0.97	829	0.97	3103	0.83	77	1.03
15	839	1.0	354	1.0	793	1.0	2576	1.0	73	1.0
60	1073	0.75	404	0.80	1190	0.67	2696	0.90	97	0.69
150	1241	0.66	713	0.40	899	0.86	3275	0.67	210	0.44

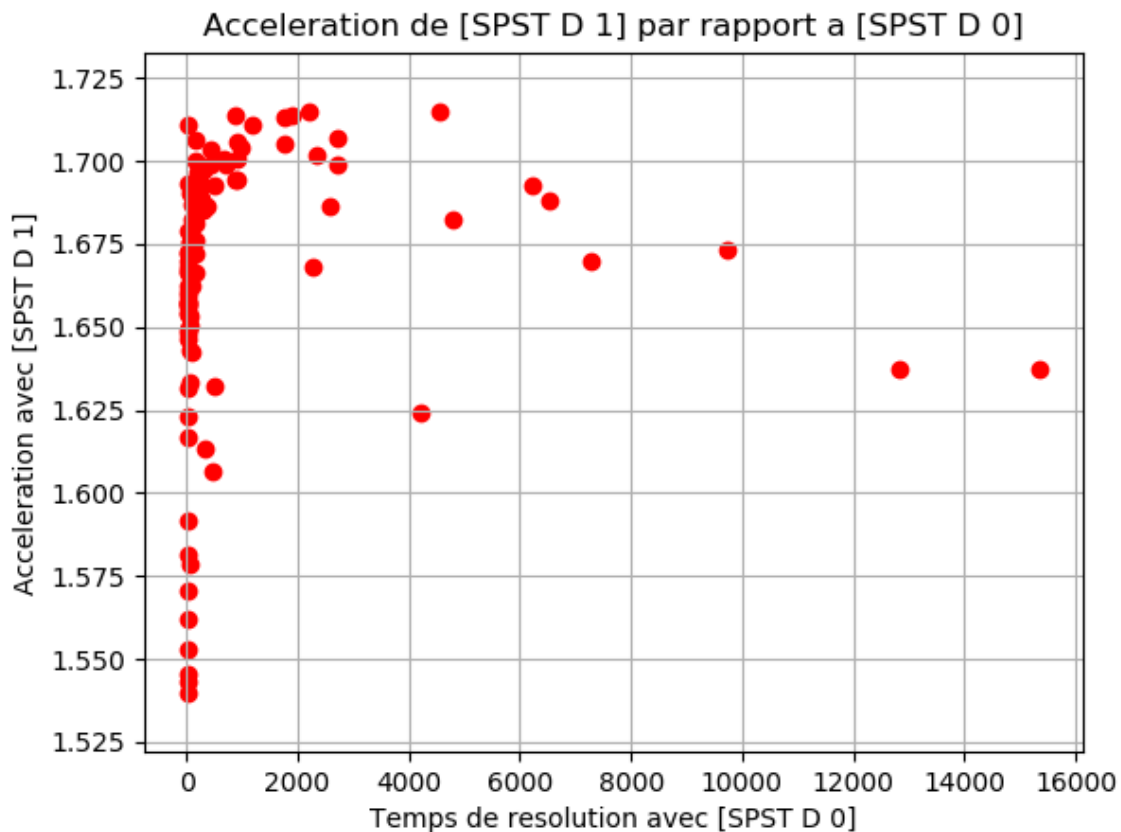
La stratégie SPST n'offre pas d'amélioration significative par rapport à SPS, donc SPS est conservée comme stratégie séquentielle de référence, avec ITERATION_LABEL_MAJ_DYN_PF valant 15.

La stratégie SPS subit des contre-performances lorsque ITERATION_LABEL_MAJ_DYN_PF vaut 0. Pourtant, après lecture du code source correspondant, le cas où ITERATION_LABEL_MAJ_DYN_PF vaut 0 ne devrait pas être très différent du cas où ITERATION_LABEL_MAJ_DYN_PF vaut 1.

Voici deux graphes de l'accélération obtenue par instance en utilisant ITERATION_LABEL_MAJ_DYN_PF = 1 au lieu de 0 en fonction du temps pris pour la résolution en

utilisant `ITERATION_LABEL_MAJ_DYN_PF = 0`, avec la stratégie SPS, puis avec la stratégie SPST, sur le graphe BAY :





On constate que le comportement de SPST est plus stable que celui de SPS dans cette situation. Cette anomalie n'a pas été analysée plus profondément.

En dehors de ce cas particulier, SPS et SPST donnent des résultats assez similaires, ce qui est une bonne chose puisque la différence entre SPS et SPST est simplement la définition d'une classe au lieu de la définition d'un patron de classe.

4.2 Comparaison des versions parallèles à la version séquentielle de référence

La stratégie séquentielle de référence utilisée pour évaluer les performances des stratégies parallèles en la stratégie SPS.

Dans les tableaux suivants, P signifie ITERATION_LABEL_MAJ_DYN_PF, TM signifie temps moyen pris par la seconde phase (en millisecondes), Acc signifie moyenne des accélérations par instance, comme pour les tableaux précédents.

Voici les résultats pour les stratégies SPPT3 et SPPT4, lorsque qu'on fait varier le nombre de threads et que ITERATION_LABEL_MAJ_DYN_PF est constant et égal à 15, la moyenne de l'accélération étant calculée en comparant les temps de résolution par instance avec ceux de la stratégie séquentielle SPS (avec ITERATION_LABEL_MAJ_DYN_PF = 15 également) :

SPPT3 [P=15]	BAY		berlin		NW		NY		paris	
Threads	TM	Acc	TM	Acc	TM	Acc	TM	Acc	TM	Acc
2	3134	1.67	2135	0.80	3955	1.13	21924	2.05	143	2.19
4	695	2.65	419	1.78	806	2.04	4227	3.31	37	3.81
8	435	3.25	265	2.44	469	2.65	2651	4.15	23	3.95
16	625	0.95	362	0.80	704	0.99	2718	1.61	46	0.83

SPPT4 [P=15]	BAY		berlin		NW		NY		paris	
Threads	TM	Acc	TM	Acc	TM	Acc	TM	Acc	TM	Acc
2	3242	1.64	1986	0.86	4083	1.45	22266	1.88	153	2.02
4	725	2.46	425	1.64	846	2.15	4288	3.06	38	3.23
8	444	3.15	264	2.34	478	2.65	2684	4.00	24	3.02
16	648	0.91	372	0.75	746	0.93	2711	1.54	50	0.77

La stratégie SPPT3 est presque toujours meilleure que SPPT4 lorsque ITERATION_LABEL_MAJ_DYN_PF est égal à 15.

L'accélération moyenne maximale pour chaque graphe est presque toujours atteinte avec huit threads, ce qui coïncide avec le nombre de threads supportés par la machine sur laquelle les mesures ont été réalisées.

Pour les mesures suivantes, on fixe le nombre de threads à huit, et on fait varier ITERATION_LABEL_MAJ_DYN_PF. Voici les résultats de ces variations pour les stratégies SPPT3 et SPPT4, la moyenne de l'accélération étant calculée en comparant les temps de résolution par instance avec ceux de la stratégie séquentielle SPS avec ITERATION_LABEL_MAJ_DYN_PF = 15 :

SPPT3 [T=8]	BAY		berlin		NW		NY		paris	
P	TM	Acc	TM	Acc	TM	Acc	TM	Acc	TM	Acc
1	441	3.22	270	2.46	471	2.58	2620	4.14	23	4.12
15	443	3.25	265	2.44	469	2.65	2651	4.15	23	3.95
60	475	3.01	304	1.95	488	2.54	2852	3.74	30	2.80
150	611	2.40	463	1.09	542	2.22	3450	2.82	83	1.74

SPPT4 [T=8]	BAY		berlin		NW		NY		paris	
P	TM	Acc	TM	Acc	TM	Acc	TM	Acc	TM	Acc
1	455	3.07	272	2.29	493	2.58	2672	3.91	24	2.91
15	444	3.15	264	2.34	478	2.65	2684	4.00	24	3.02
60	485	2.94	308	1.86	499	2.51	2889	3.56	31	2.11
150	629	2.36	476	1.04	556	2.34	3508	2.68	87	1.26

Prendre ITERATION_LABEL_MAJ_DYN_PF égal à 15 reste la meilleure option.

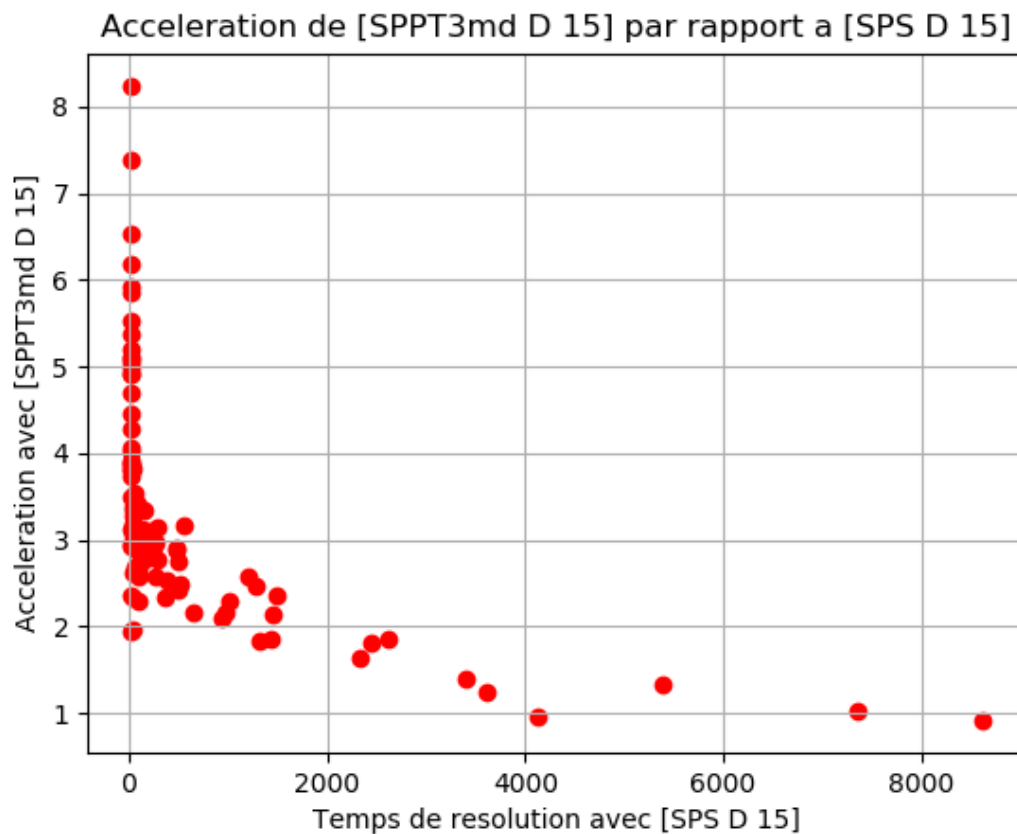
Quelque soit l'instance à résoudre, la stratégie qui aura en moyenne les meilleurs temps de

résolution est la stratégie SPPT3 avec huit threads et ITERATION_LABEL_MAJ_DYN_PF égal à 15.

4.2.1 Limites

Sur le graphe suivant, on trace un point par instance de BAY. L'abscisse de chaque point est son temps de résolution (en millisecondes) avec la stratégie SPS avec ITERATION_LABEL_MAJ_DYN_PF = 15. L'ordonnée de chaque point est l'accélération apportée par la stratégie SPPT3 avec huit threads par rapport à la stratégie SPS.

On constate que les instances dont la résolution est la plus accélérée sont celles dont la résolution ne prenait déjà que peu de temps.



On retrouve un nuage de points avec la même allure avec tous les graphes cités précédemment et toutes les stratégies parallèles.



Bibliographie

- [1] Guy E BLELLOCH, Yan GU, Yihan SUN et Kanat TANGWONGSAN. « Parallel shortest paths using radius stepping ». In : *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM. 2016, p. 443-454.
- [2] Andreas CRAUSER, Kurt MEHLHORN, Ulrich MEYER et Peter SANDERS. « A parallelization of Dijkstra's shortest path algorithm ». In : *International Symposium on Mathematical Foundations of Computer Science*. Springer. 1998, p. 722-731.
- [3] Michael KAINER et Jesper Larsson TRÄFF. « More Parallelism in Dijkstra's Single-Source Shortest Path Algorithm ». In : *arXiv preprint arXiv :1903.12085* (2019).
- [4] Yannick KERGOSIEN, Antoine GIRET, Emmanuel NÉRON et Gaël SAUVANET. *An efficient label-correcting algorithm for the multi-objective shortest path problem*. Rapp. tech. 2019.
- [5] Ulrich MEYER et Peter SANDERS. « Δ -stepping : A parallel single source shortest path algorithm ». In : *European symposium on algorithms*. Springer. 1998, p. 393-404.
- [6] Peter SANDERS et Lawrence MANDOW. « Parallel label-setting multi-objective shortest path search ». In : *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE. 2013, p. 215-224.

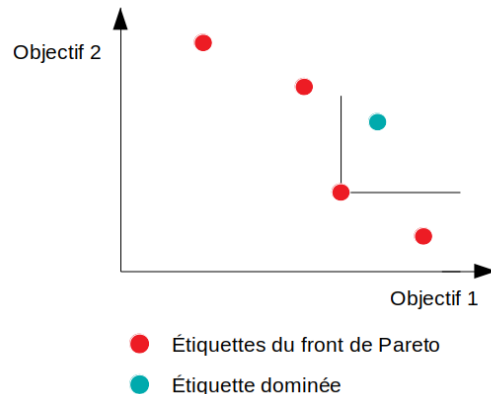
Parallélisation de la recherche de plus court chemin multi-objectif

Simon MOULARD

Encadrement : Emmanuel NERON

Objectifs

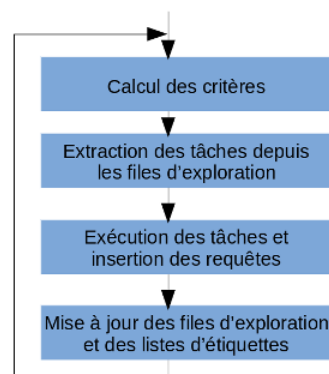
Ce projet consiste à paralléliser une partie d'une méthode exacte servant à déterminer l'ensemble des plus courts chemins multi-objectifs entre deux nœuds d'un graphe, cet ensemble étant appelé un front de Pareto.



Exemple de front de Pareto. Les solutions optimales ne se dominent pas entre elles.

Mise en œuvre

On implémente plusieurs stratégies parallèles en combinant les techniques présentes dans la stratégie séquentielle multi-objectif existante fournie, notamment la mise à jour dynamique du front de Pareto, avec la structure d'une algorithmes parallèle de recherche de plus court chemin mono-objectif, qui propose un moyen d'extraire plusieurs tâches des files d'exploration par itération.

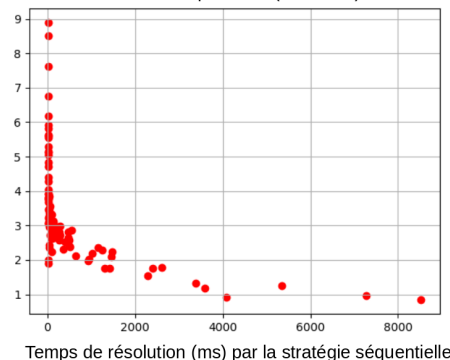


Boucle principale des stratégies parallèles. Chaque étape doit être terminée par tous les threads avant qu'ils passent à la suivante.

Résultats

Les stratégies parallèles implémentées permettent d'obtenir une réduction significative du temps nécessaire pour les résolutions d'instances, néanmoins ce gain n'est présent que pour les instances qui se résolvent déjà rapidement avec la stratégie séquentielle. Bien que deux stratégies parallèles aient été implémentées, elles sont très proches et donnent des résultats très similaires.

Accélération moyenne par instance avec une stratégie parallèle par rapport à celle séquentielle (8 threads)



La résolution des instances simples est grandement accélérée, mais les instances difficiles n'en profitent pas.

Parallélisation de la recherche de plus court chemin multi-objectif

Simon MOULARD

Encadrement : Emmanuel NERON

Objectifs

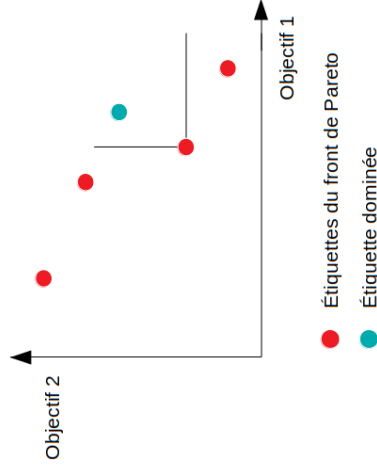
Ce projet consiste à paralléliser une partie d'une méthode exacte servant à déterminer l'ensemble des plus courts chemins multi-objectifs entre deux nœuds d'un graphe, cet ensemble étant appelé un front de Pareto.

Mise en œuvre

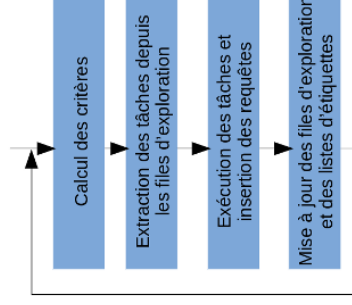
On implémente plusieurs stratégies parallèles en combinant les techniques présentes dans la stratégie séquentielle multi-objectif existante fournie, notamment la mise à jour dynamique du front de Pareto, avec la structure d'un algorithme parallèle de recherche de plus court chemin mono-objectif, qui propose un moyen d'extraire plusieurs tâches des files d'exploration par itération.

Résultats

Les stratégies parallèles implémentées permettent d'obtenir une réduction significative du temps nécessaire pour les résolutions d'instances, néanmoins ce gain n'est présent que pour les instances qui se résolvent déjà rapidement avec la stratégie séquentielle. Bien que deux stratégies parallèles aient été implémentées, elles sont très proches et donnent des résultats très similaires.



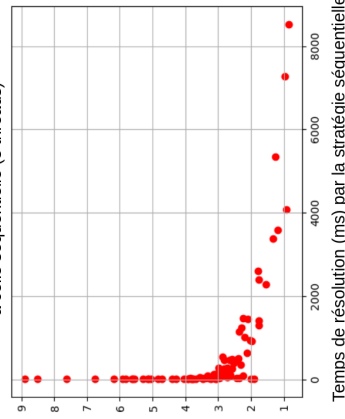
Exemple de front de Pareto. Les solutions optimales ne se dominent pas entre elles.



Boucle principale des stratégies parallèles.

Chaque étape doit être terminée par tous les threads avant qu'ils passent à la suivante.

Accélération moyenne par instance avec une stratégie parallèle par rapport à celle séquentielle (8 threads)



La résolution des instances simples est grandement accélérée, mais les instances difficiles n'en profitent pas.

Parallélisation de la recherche de plus court chemin multi-objectif

Résumé

Ce projet de recherche propose de rendre parallèle une méthode exacte séquentielle visant à résoudre le problème de plus court chemin multi-objectif, plus précisément la détermination de l'ensemble des chemins non-dominés entre deux nœuds d'un graphe qui minimisent plusieurs fonctions objectifs. La méthode séquentielle est basée sur un algorithme de label-correcting comprenant plusieurs techniques d'amélioration. Des résultats expérimentaux mesurent l'efficacité de la nouvelle méthode parallèle par rapport à la version séquentielle.

Mots-clés

di, prd, plus court chemin, correction d'étiquettes, multi-objectif, parallèle

Abstract

This research project's purpose is to make parallel a exact sequential method to solve the one-to-one multi-objective shortest path problem, which aims to determine a set of non-dominated paths between two given nodes in a graph that minimize several objective functions. The method is based on a label-correcting algorithm with several improvement techniques. Computational experiments measures the efficiency of the parallel method compared to the sequential one.

Keywords

di, prd, shortest path, label-correcting, multi-objective, parallel