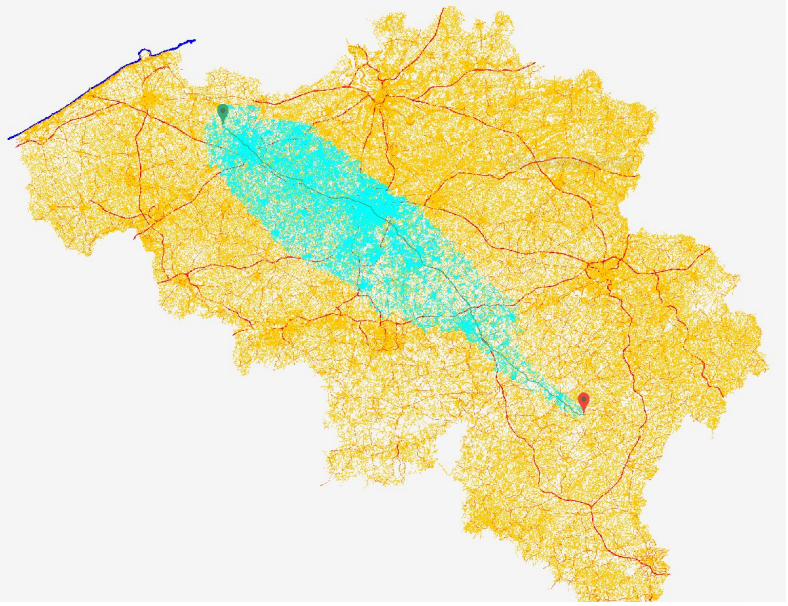
RIZZI Emma
ROUSSEAU Patrick
3MIC C

31 mai 2019

BE Graphes

- Rapport -

https://github.com/rizziemma/be-graphes/



A Star - Belgique

Introduction **2**

I. Implémentation des algorithmes de plus court chemin **3**

Conception 3

Dijkstra 4

A Star 5

II. Tests de validité **5**

Méthode utilisée 5

Résultats : 6

Méthode sans oracle 6

III. Tests de performance **7**

Méthode utilisée 7

Création des jeux de données 7

Lancement des tests et écriture des résultats 7

Lecture des résultats 7

Résultats : 8

IV. Problème ouvert : point de rencontre **8**

Conclusion **8**

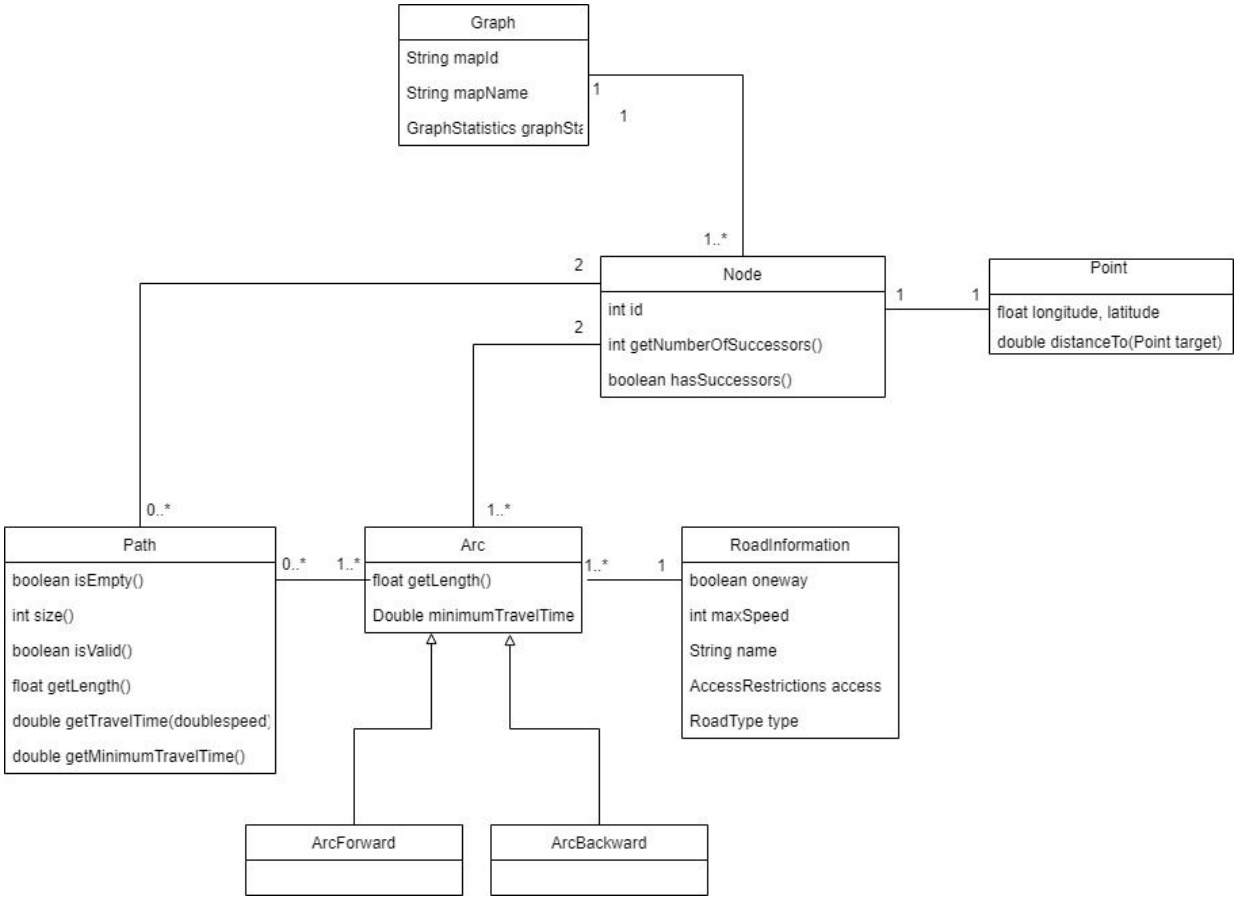
Introduction

Dans le cadre de l'UF de Graphes et POO de 3ème année, nous avons implémenté, prouvé la validité et testé les performances de différents algorithmes de recherche de plus courts chemin. Premièrement, nous expliquerons comment nous avons codé les algorithmes de Dijkstra puis une version suivant l'heuristique d'A Star. Nous détaillerons ensuite la méthode suivie pour prouver leur validité avec comme oracle l'algorithme de Bellman-Ford fourni. Enfin, nous comparerons les performances d'A Star et Dijkstra dans différentes conditions d'utilisations selon les cartes et paramètres de recherche de chemin.

I. Implémentation des algorithmes de plus court chemin

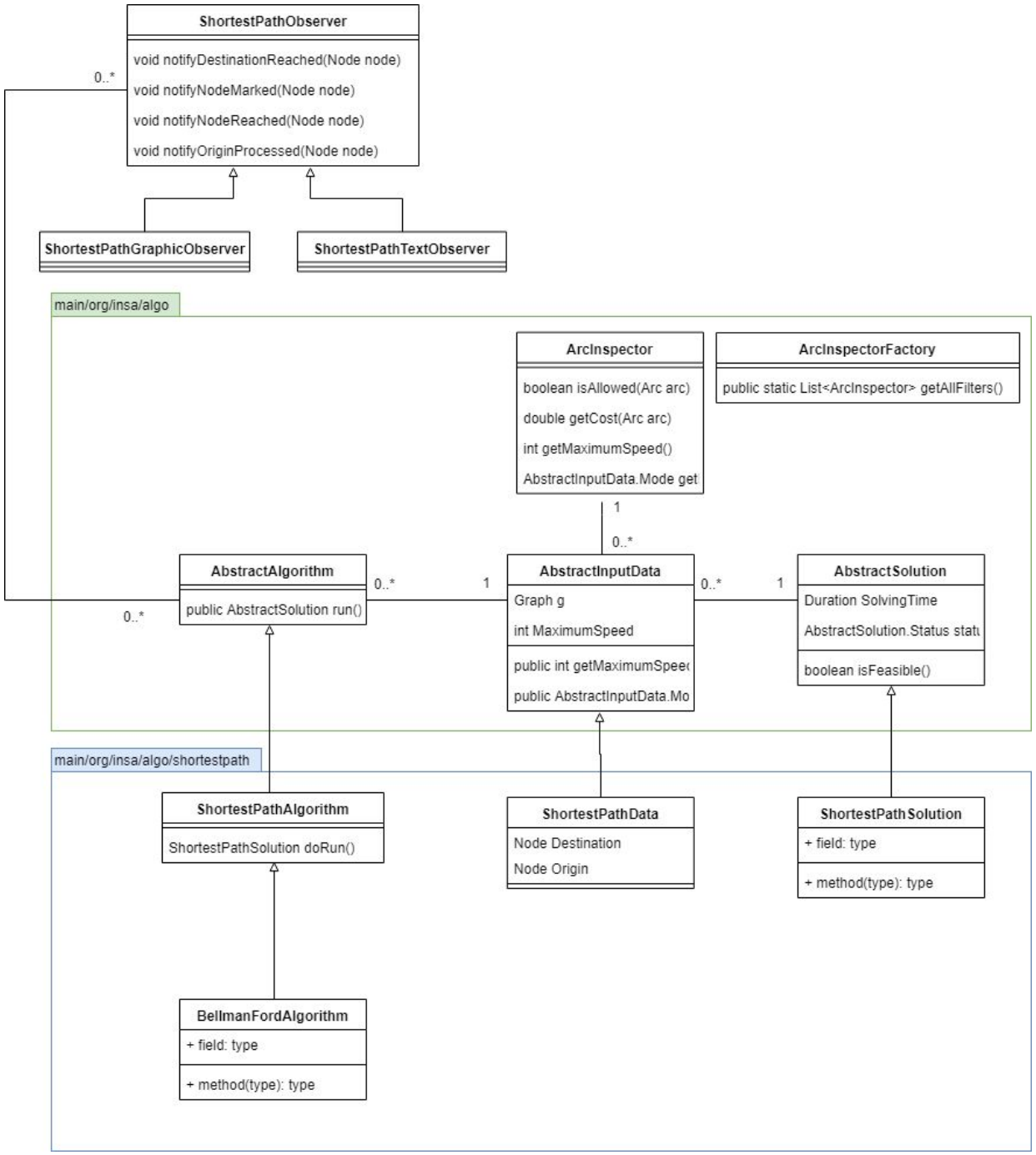
A. Conception

Avant d'implémenter nos algorithme de plus courts chemin, nous avons commencé par la compréhension des classes fournies. Voici le diagramme UML du package insa/graph permettant de visualiser les liens entre les différents éléments d'un graph. Nous avons ensuite complété la classe Path en implémentant les fonctions initialement *@deprecated* afin de passer les tests unitaires associés.



UML insa/graph

Voici le diagramme correspondant aux packages insa/algo et insa/algo/shortestpath, permettant de comprendre la construction d'un algorithme de plus court chemin et ses interactions avec les autres classes :



UML insa/algo et insa/algo/shortestpath

Nous avons ainsi pu compléter la classe BinaryHeap en implémentant la fonction remove(), nécessaire pour le fonctionnement de l'algorithme de Dijkstra.

B. Dijkstra

Pour cette algorithme, nous avons besoin d'un tas binaire trié par coût croissant. Pour cela nous avons créé la classe Label, qui associe un node à un coût, un marquage, et l'arc du node prédécesseur afin de reconstituer le chemin jusqu'au point de départ.

L'algorithme commence par la fonction InitRun(), qui crée un Label pour chaque Node du graphe, initialisé à un coût infini sauf pour le point de départ.
Ensuite, l'algorithme boucle jusqu'à ce que la destination soit marquée, ou qu'il n'y a plus de node dans le tas (et donc que le chemin est impossible). A chaque itération, on extrait le sommet de plus faible coût du tas, puis on parcourt tous ses successeurs dont on met à jour les coûts si ce chemin permet de réduire le coût actuel.

Pour reconstituer la solution, nous parcourons tous les arc prédécesseurs stockés dans les Labels à partir du Node destination jusqu'à l'origine, et l'on enregistre chaque Node parcouru dans une ArrayList solution. De cette manière, les Nodes seront stockés dans l'ordre inverse, d'où l'usage de *Collections.reverse(nodesSol)* avant de retourner la solution.

C. A Star

Nous avons implémenté A Star en héritant de Dijkstra, seule la fonction InitRun() diffère. Au lieu d'un tas de Label, nous utilisons un tas de LabelStar contenant un coût (depuis l'origine) et un coût estimé (jusqu'à la destination). Ce coût estimé est initialisé à la création du Label car il ne change pas au court de l'algorithme, avec la fonction distance(n1, n2) de la classe Point permettant d'évaluer la distance à vol d'oiseau entre le Node actuel et la destination.

Le tas binaire est trié selon la somme du coût actuel + coût estimé, ainsi à chaque itération le sommet à explorer est choisi en fonction du coût minimal estimé jusqu'à la destination, ce qui permet d'orienter la recherche vers la destination, plutôt que d'explorer dans toutes les directions comme le fait Dijkstra. Le reste de l'algorithme est identique à Dijkstra.

II. Tests de validité

A. Méthode utilisée

Nous avons développé des tests unitaires pour prouver la validité de nos algorithmes. Ces tests reposent sur la comparaison avec la solution donnée par l'algorithme de Bellman, que l'on considère optimale.
Voici les différents jeux de données que nous avons choisi :

- *emptyPath* : chemin vide (origine et destination à null),
- *singleNodePath* : chemin où origine = destination,
- *infeasiblePath* : chemin impossible entre deux îles sur la carte du Chili,
- *PathLength_i*, *PathLengthCars_i*, *PathTime_i*, *PathTimeCars_i*, *PathTimePedestrian_i* : sur trois cartes différentes (Carré dense, Belgique, Chili) nous avons choisi un chemin faisable pour les différentes options de recherche (choix de l'option par la sélection de l'ArcInspector).

Ces chemins sont initialisés par l'appel à la fonction *@before class,* en récupérant le graphe des trois cartes choisies par un *GraphReader,* et déclarant les différents *ShortestPathData* avec les nodes de départ, arrivée, le graph et l'arc inspector associé.

Pour chaque chemin, voici les tests effectués :

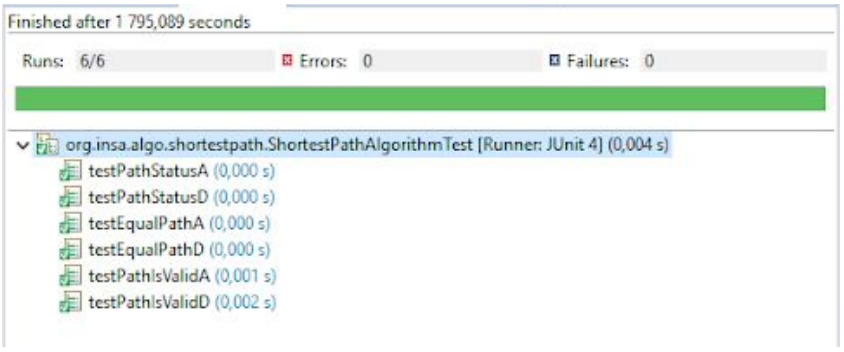
- testPathIsValid() : vérifie si le chemin retourné est valide selon la donnée en entrée
- testPathStatus() : vérifie le statut du chemin (Feasible ou Infeasible)

Pour les chemins possibles (Feasible), nous testons ainsi leur optimalité :

- testEqualPath() : Compare la longueur/temps du chemin retourné à celui trouvé par l'algorithme de Bellman pour la même entrée, à 10^{-6} près.

B. Résultats

Après plusieurs essai et correction de nos algorithmes, notamment de la fonction remove() du BinaryHeap, nous avons à présent des résultats optimaux pour Dijkstra et A Star. Nous pouvons maintenant tester leurs performances.



C. Méthode sans oracle

La méthode envisagée pour effectuer les tests sans oracle repose sur la propriété suivante : tout sous-chemin d'un plus court chemin est un plus court chemin. Ainsi, pour un chemin résultat donné d'un algorithme, nous pouvons choisir un certain nombre de nodes situés sur ce chemin et lancer le même algorithme en remplaçant l'origine ou la destination par un de ces nodes (en respectant bien la direction de recherche).

Chacun de ces sous chemins doit être un plus court chemin inclu dans le premier chemin solution. En parcourant ainsi le sous-chemin et le chemin solution node par node, nous pouvons déterminer s'ils sont bien égaux et prouver donc que le chemin solution est optimal.

III. Tests de performance

A. Méthode utilisée

a. Création des jeux de données

Afin de tester les performances de nos algorithmes nous avons créé des jeux de données. Le but était de montrer les différences entre les algorithmes A Star et Dijkstra. Pour ce faire on a choisi des cartes très différentes dans leurs nature afin de montrer les forces et les faiblesses de ces deux algorithmes, le choix s'est porté sur quatre cartes : la Réunion, la Haute-garonne, le Carrée Dense et la Belgique.
Pour avoir des résultats moyens nous avons décidé de faire un grand nombre de tests, ici cent.

Pour réaliser ces tests on a besoin de connaître la carte, le mode de parcours (temps ou distance) et le sommet de départ et d'arrivée. Donc pour une carte donnée on va générer aléatoirement 200 paires de sommets (pour avoir au moins 100 chemins faisables parmis ces paires). Ces informations sont écrites dans différents fichiers texte.

b. Lancement des tests et écriture des résultats

Pour lancer les tests, nous parcourons tous les fichiers texte générés, récupérons la carte associée au fichier courant par un *GraphReader*, et pour chaque paire de sommet on exécute l'algorithme Dijkstra ou A Star. Si le chemin est faisable nous sauvegardons dans un autre fichier texte la longueur (en Node) du chemin, le temps d'exécution de l'algorithme, et le nombre de sommets entrés dans le tas binaire. On passe au fichier de données suivant lorsque 100 tests ont retourné un chemin faisable.

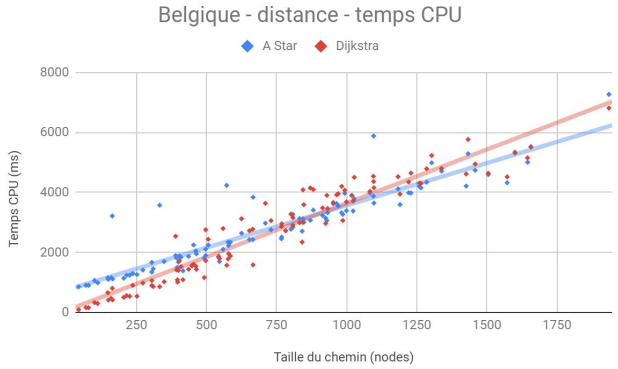
c. Lecture des résultats

En stockant les résultats de ces tests dans des fichiers texte, nous pouvons facilement les exporter vers un tableur pour générer des graphiques, plus facilement interprétables. Nous avons choisi de comparer les résultats de Dijkstra et A Star, en temps et distance, sur chaque carte selon le temps d'exécution et le nombre de sommets visités, en fonction de la taille du chemin solution en nombre de Nodes.

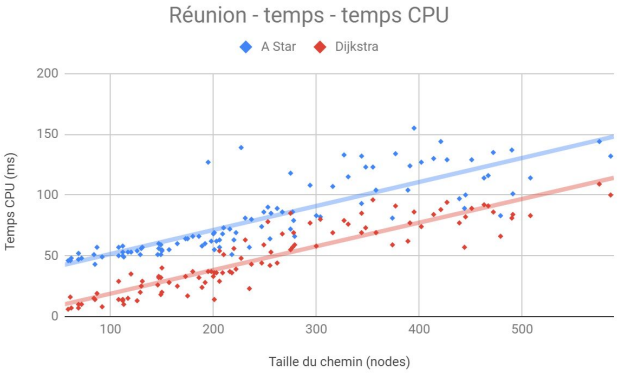
B. Résultats :

L'intégralité des résultats de ces tests est disponible en annexe. Nous présentons ici les résultats les plus significatifs que nous avons eu.

a. Temps d'exécution



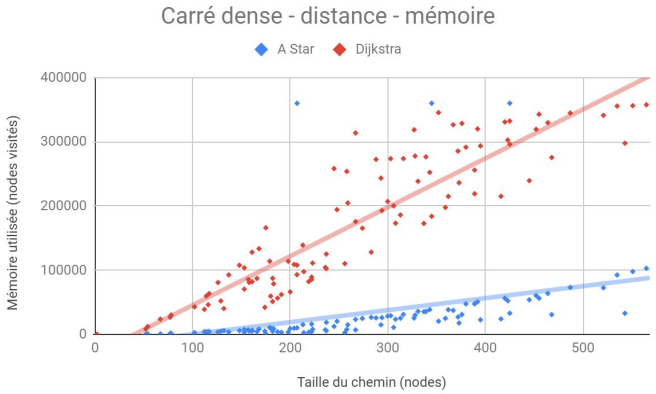
On remarque que Dijkstra est plus rapide pour la recherche de chemin courts, mais A Star est plus performant pour les grandes distances car le temps d'initialisation des Labels est plus long pour A Star et donc réduit ces performances sur des courts chemin. Cela vient notamment de notre choix d'initialiser tous les Labels au début de l'algorithme en calculant le coût estimé, plutôt que de calculer ce coût uniquement lorsque le sommet est visité, ce qui serait une piste d'amélioration pour notre algorithme.



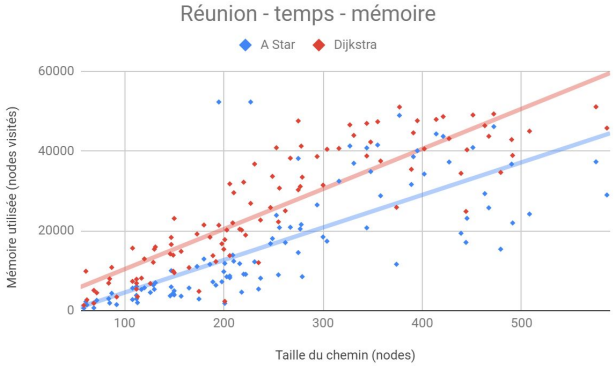
Nous avons choisi la carte de la Réunion car elle présente un obstacle naturel qui oblige de contourner le centre de la carte pour aller d'un côté à l'autre de l'île. Nous pouvons voir que A Star est moins performant sur les grandes distances également (donc les chemins qui doivent contourner cet obstacle).

Ainsi, nous pouvons en conclure que A Star est plus rapide pour des recherches de longs chemin sur des cartes urbanisées uniformément, alors qu'il faudra préférer Dijkstra pour rechercher des chemins courts ou devant contourner des obstacles.

b. Mémoire utilisée



Ici, on peut voir que A Star est bien plus performant que Dijkstra, et la différence est d'autant plus importante que le chemin est long. A Star envisage moins de sommet que Dijkstra grâce à l'orientation des recherches vers la destination par le coût estimé.



Bien que l'écart soit moins marqué dans le cas de l'île de la Réunion, A Star reste plus performant que Dijkstra en nombre de sommets visités. Ainsi, de manière générale, si l'on cherche à minimiser l'espace mémoire utilisé, A Star sera plus intéressant à utiliser que Dijkstra.

IV. Problème ouvert : point de rencontre

Le problème consiste à déterminer un ensemble de points de rencontre M entre deux noeuds de départ (O1 et O2) tels que :

- les coûts des trajets MO1 et MO2 sont égaux à +/- 15%
- MO1 et MO2 sont tous deux environs égaux à la moitié du chemin O1O2 à +/-30%

Voici la méthode que nous avons envisagée :

- Déterminer le coût **C = O1O2 par A*** (plus rapide).
- Exécuter une version modifiée de Dijkstra depuis O1 et O2 telle que la **nouvelle condition d'arrêt** soit que le prochain sommet à marquer (celui que l'on extrait de la pile à chaque itération) ait un **coût supérieur à C+30%**. Ainsi, on arrête l'algorithme une fois que tous les sommets marqués Mi ont un coût MiOi < C+30% avec i=1 ou 2 selon le point de départ. Au lieu de reconstituer un plus court chemin, cette version retourne l'ensemble des nodes marqués associé à leur coût MiOi dans un label par exemple.
- On parcourt la liste M1 (boucle for(Node N : M1)), si N appartient à M2 ET les coûts respectent les deux conditions :
    - |O1N - O2N| / O1N <= 0.15
    - |O1N - O2N| / O2N <= 0.15

  On place le sommet N dans l'ensemble des sommets M solutions. (Pour avoir MO1 et MO2 égaux à +/- 15%). Pour visualiser cet ensemble sur la carte, au lieu de notifier l'observeur que les sommets sont atteints lorsqu'ils sont marqués par Dijkstra, on les notifie lors de l'insertion dans N.

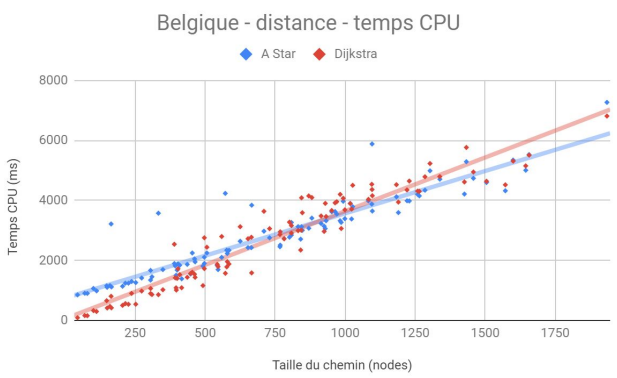
Conclusion

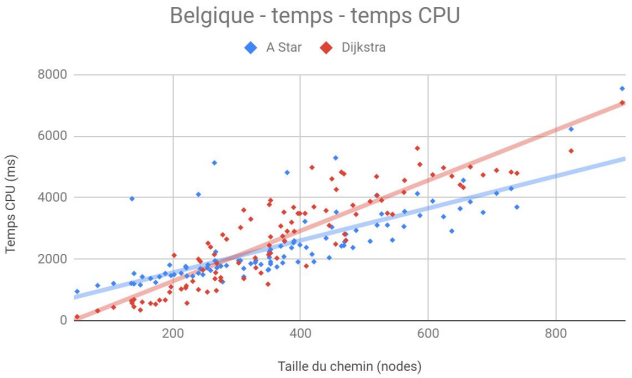
Au travers de ce BE nous avons pu implémenter et appréhender de manière concrète des algorithmes de plus courts chemins, étudiés théoriquement plus tôt dans l'année. Par l'affichage graphique, nous avons pu mieux comprendre la méthode de recherche utilisée par chacun de ces algorithmes. Nos tests ont montré que l'algorithme de Dijkstra explorait toutes les possibilités, de façon circulaire autour du point de départ, et ce, jusqu'au point d'arrivée. L'algorithme A Star, oriente ses recherches dans la direction de la destination par estimation de la distance à vol d'oiseau pour réduire le nombre de sommets explorés.

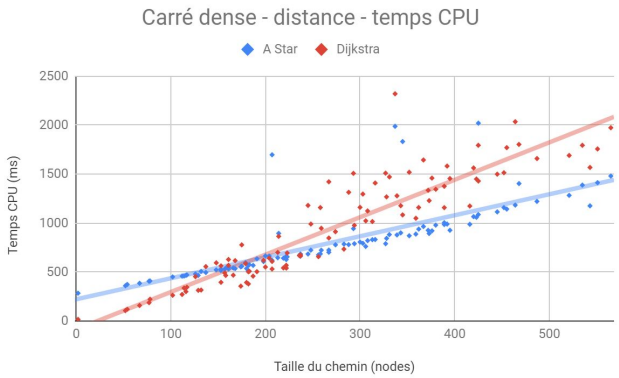
Sur une carte urbanisée de façon homogène, c'est A Star dont il faut privilégier l'utilisation. Il trouvera le chemin plus rapidement, en utilisant le moins de mémoire possible. Par contre selon l'endroit où on se trouve, il peut être préférable d'utiliser l'algorithme de Dijkstra classique comme par exemple pour un chemin très court, sur une île ou lorsqu'un obstacle qu'il faut contourner empêche de trouver un chemin proche d'une droite entre l'origine et la destination.

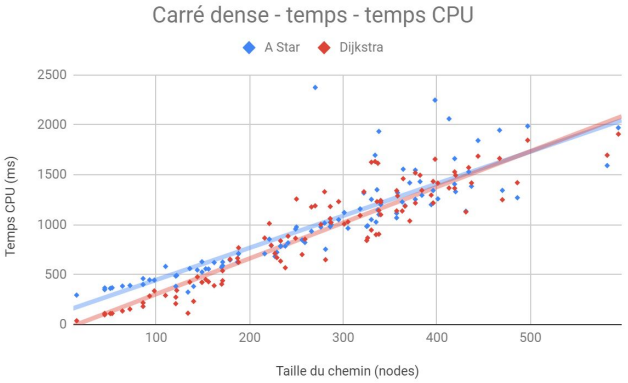
Plus généralement, les algorithmes de plus court chemin peuvent servir dans de nombreuses situation comme des problèmes de covoiturage, de livraison ou la recherche de point de rencontre comme le problème ouvert dont nous avons fourni un pseudo code. Leur application et adaptation doit être réfléchie mais elle permet de trouver des solutions optimales à tous les problèmes.

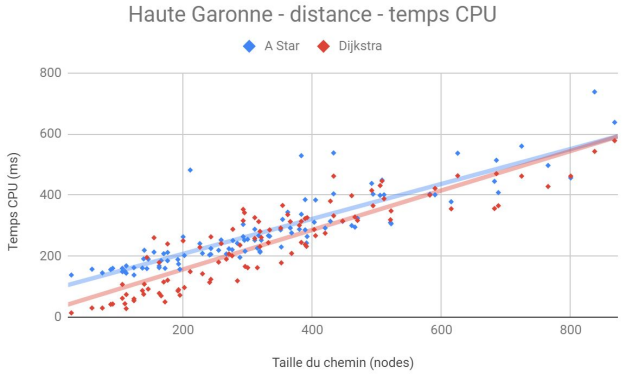
Annexe 1 : Performance en temps CPU en fonction de la taille du chemin

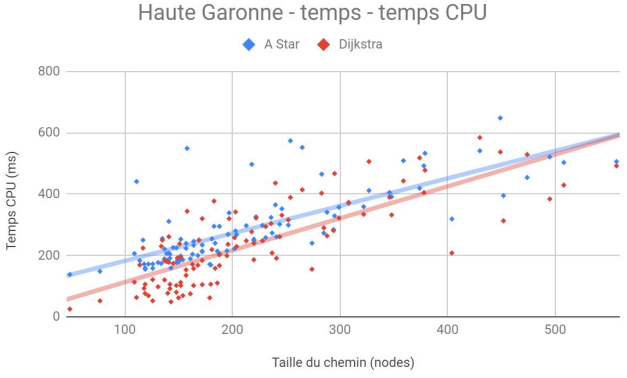


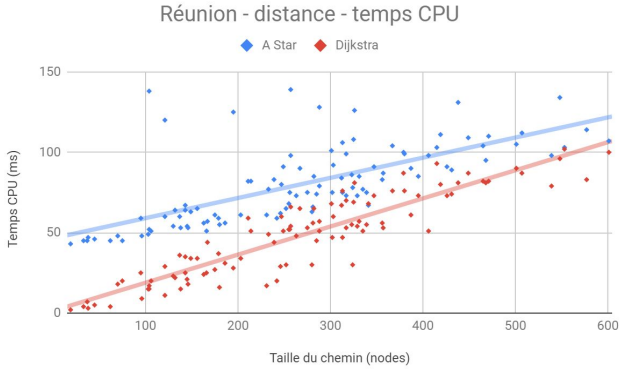


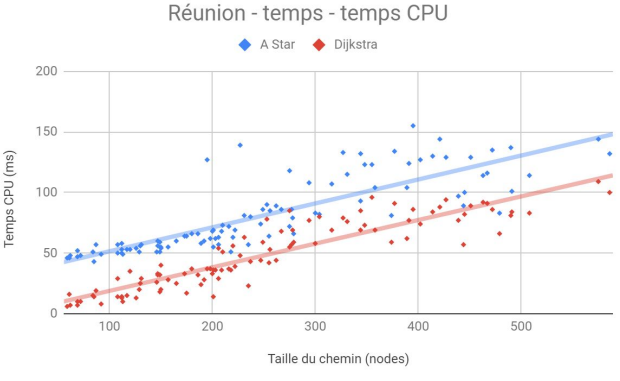












Annexe 2 : Performance en temps CPU en fonction de la taille du chemin