Rapport BE-Graphe

Baudoint Emma 3MIC D

Sommaire

[Introduction 1](#_Toc41142835)

[Bilan de l’étape 1 et de l’étape 2 (finie) 2](#_Toc41142836)

[1.Diagramme UML 2](#_Toc41142837)

[2. Test sur les chemins 3](#_Toc41142838)

[a) Préparation des différents chemins 3](#_Toc41142839)

[b) Test des fonctions : 3](#_Toc41142840)

[Bilan Etape 3 et 4 ( reste Binary Heap non traité ) 4](#_Toc41142841)

[1. Diagramme UML 4](#_Toc41142842)

[2. Implémentation de Dijkstra 4](#_Toc41142843)

[Bilan étape 5 – A\* 6](#_Toc41142844)

[Test Unitaire, vérification de Dijkstra et A\* 7](#_Toc41142845)

[1. Test 1 et 2 : Chemin existant 7](#_Toc41142846)

[2. Test 3 et 4 : Chemin non-existant ou nul 7](#_Toc41142847)

[Test d’optimalité 9](#_Toc41142848)

[1. Hypothèse sur les performances 9](#_Toc41142849)

[2. Idées d’implémentation 10](#_Toc41142850)

[Problème ouvert 12](#_Toc41142851)

# Introduction

L’objectif de ce rapport est de comparer, analyser et expliquer les performances des algorithmes de plus court chemin. Ces algorithmes s’effectuent entre deux sommets sur une carte. J’ai étudié trois algorithmes :

Un premier déjà codé qui est l’algorithme de Bellman-Ford qui utilise un graphe pondéré orienté. Les deux suivants m’ont été donnés à coder. Le premier algorithme à coder à été l’algorithme de Dijkstra publié par Edsger Dijkstra en 1959. Le second est une application de la méthode A-Star (ou A\*) qui est une extension de l’algorithme de Dijkstra. Cette extension consiste essentiellement en un ajout de l’estimation du coût restant tandis que l’algorithme classique ne rend en compte que le coût du sommet d’origine au sommet actuel.

A partir de ces trois algorithmes, nous allons donc comparer leurs vitesses d’exécutions et comparer les performances et les défauts de chacun des algorithmes. Le lien git ci-dessous donne accès au code utilisé :

<https://github.com/Emma-bau/be_graphe.git>

# Bilan de l’étape 1 et de l’étape 2 (finie)

## 1.Diagramme UML

Une image contenant capture d’écran

Description générée automatiquementLe diagramme UML suivant a été réalisé afin d’améliorer la compréhension des classes utilisées et des liens entre celles-ci.

Diagramme UML des classes Graph, Arc, Path, Nodes, et RoadInformation.

J’ai implémenté les fonctions suivantes dans la classe Path :

* getLength() : qui retourne la longueur du chemin
* getTravelTime (double speed) : qui retourne le temps de trajet sur le chemin à la vitesse donnée
* getMinimumTravelTime() : retourne le temps minimum nécessaire pour parcourir le chemin à la vitesse maximum
* isValid() : boolean renvoyant vrai si un path est valide
* createShortestPathFromNodes() : retourne le chemin le plus rapide trouvé

## Test sur les chemins

### Préparation des différents chemins

Afin de préparer les différents tests sur les fonctions écrites, il faut d’abord préparer un jeu de chemin sur lesquels faire les tests. Pour cela on va servir de l’interface @BeforeClass. Cela va permettre de construire une fois toutes les données nécessaires à nos tests.

@BeforeClass

**Public** **static** **void** initAll() **throws** IOException

Ainsi dans la fonction ci-dessus on va créer tous les types de chemins dont nous avons besoin pour tester nos fonctions. Cela comprend des chemins vides, des chemins courts, des chemins plus longs et plus compliqués et des chemins comportant des erreurs.

### Test des fonctions :

Les campagnes de test sur les chemins m’ont permis de vérifier la pertinence des algorithmes. Elles servaient en effet à tester les fonctions que nous venions de créer sur différents types de chemins et de situations. Ces tests servent à trouver le maximum de problème dans notre programme afin de pouvoir les régler et de rendre à la fin un logiciel qui fonctionne presque parfaitement.

On va alors utiliser l’interface Test et l’annotation @Test. Cette interface nous permet de réaliser des tests unitaires. L’annotation indique à JUnit que la méthode à laquelle elle est attachée peut être utilisée en tant que test. A ce moment-là, JUnit construit une nouvelle instance de la classe et appelle la méthode qui a été annotée. Toutes les exceptions levées par le test seront signalées comme un échec. Par exemple, sur la classe Path, concernant les tests sur GetLength(), la fonction assertsEquals est utilisée pour comparer deux résultats attendus. Ainsi on compare un résultat attendu avec le résultat réellement obtenu par notre fonction quand on l’applique au chemin :

*assertEquals*(0, *emptyPath*.getLength(), 1e-6);

Dans l’assertion ci-dessus, on va comparer pour un chemin vide la longueur attendue : 0 avec la longueur obtenue par notre fonction quand on l’applique à un chemin vide. Cette campagne de test nous permet donc de vérifier que notre fonction marche et renvoie le résultat attendu pour toutes les différentes possibilités de chemin possible.

# Bilan Etape 3 et 4 ( reste Binary Heap non traité )

## Diagramme UML

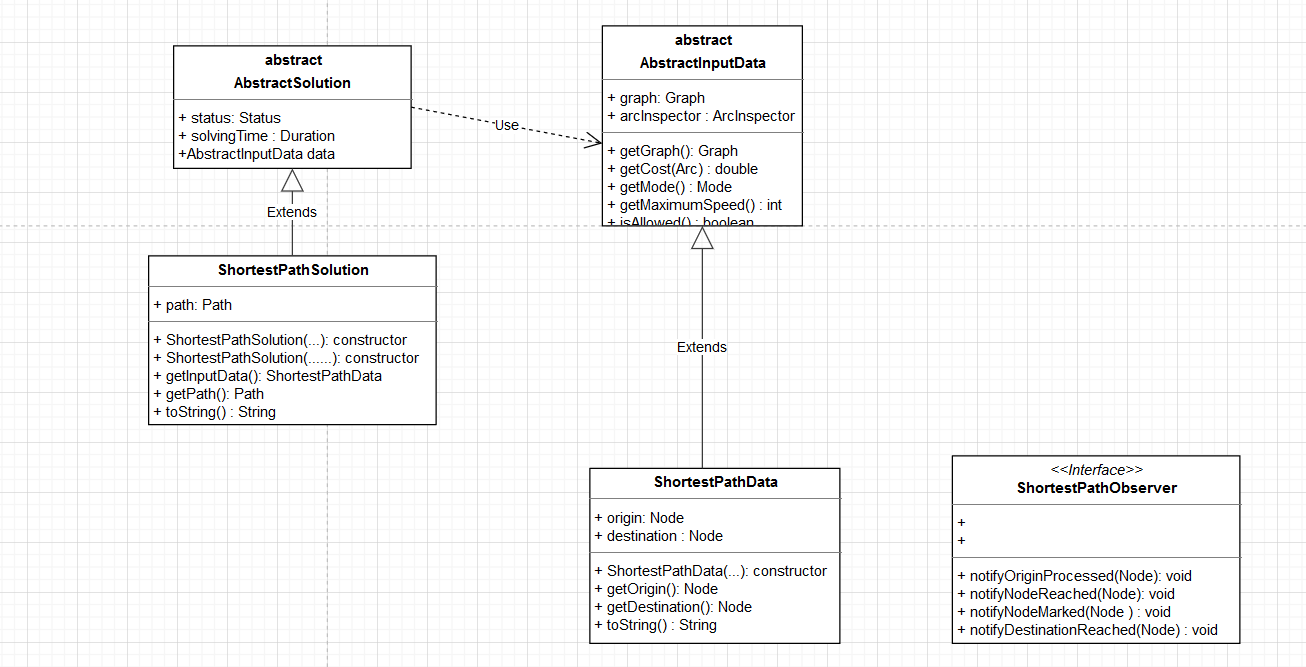


Diagramme UML de la forme des données utilisées pour ShortestPath.

## Implémentation de Dijkstra

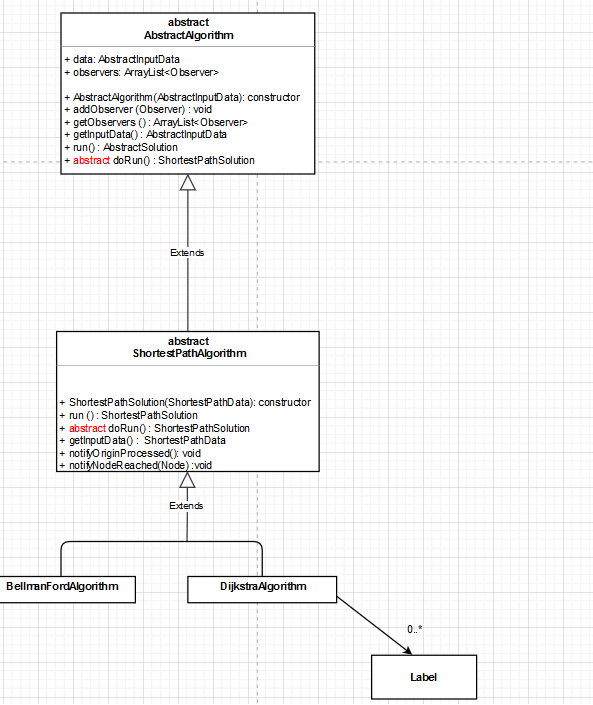


Diagramme UML pour l’algorithme Dijkstra avec l’ajout de la classe Label.

La classe Label a été implémentée avec les fonctions suivantes :

* **public** Label() : constructeur de Label par défaut

* **public** Label(Node node ) : constructeur de Label avec un nœud

* **public** **void** setPreviousArc(Arc arc) : setter de previousArc

* **public** **void** setPreviousLabel(Label label) : setter de Label

* **public** Label getPreviousLabel() : getter de previousLabel

* **public** **double** getCost() : getter de cost

* **public** **void** setCost(**double** cost) : setter de cost

* **public** **void** setHasBeenVisited(**boolean** b) : setter de HasbeenVisited

* **public** **boolean** hasBeenVisited() : vérifie si un nœud à été marqué

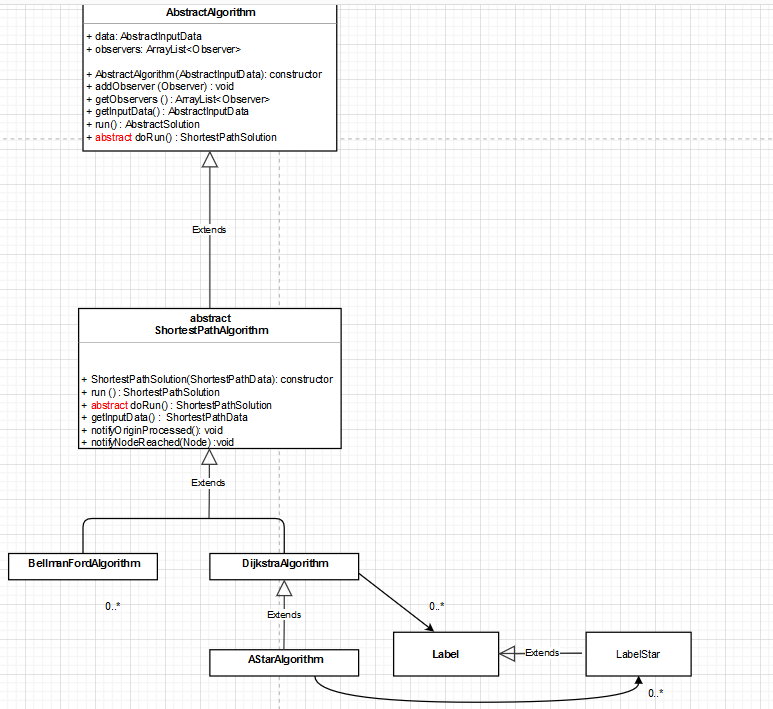
* **public** Node getNode() : getter de Node

* **public** Arc getPreviousArc() : getter de previousArc

* **public** **boolean** hasPreviousArc() : vérifie si un noeud a un prédécesseur
* **public** **double** getTotalCost() : getter de totalCost
* **public** **int** compareTo(Label o) : implémentation de compareTo

La vérification de l’algorithme se fait grâce au test unitaire (voir partie Test Unitaire, vérification de Dijkstra et A star ) .

# Bilan étape 5 – A\*



L’algorithme A\* est une extension de Dijkstra, il est conçu pour être plus rapide. Il est implémenté en utilisant une classe LabelStar qui hérite de la classe Label. Elle réimplémente la fonction getTotalCost() et prend en compte le coût estimé vers le commet destination, au moment du placement du nœud dans le tas. J’ai aussi implémenté un nouveau constructeur LabelStar qui contient en plus le calcul de ce cout estimé.

Ainsi l’algorithme A\* contient simplement la ligne suivante :

**protected** LabelStar newLabel(Node node, ShortestPathData data) {

**return** **new** LabelStar(node,data);}

Les seules fonctions qu’on réimplémente est newLabel et updateTas. Afin d’avoir désormais un tri dans le tas par ordre de coût estimé, j’ai mis en place une nouvelle fonction UpdateTas() et qui réalise un tri à bulle sur les labels présents dans notre tas, grâce à la fonction getTotaCcost(). A la fin de cette fonction, on a donc notre tas qui est ordonné par cout estimé. Dans le cas de Dijkstra, le tas est juste oridnné selon le cout réel puisque getTotalCost() est égal à getCost() pour la classe Label.

# Test Unitaire, vérification de Dijkstra et A\*

Une campagne de test a été réalisée afin de vérifier que nos deux algorithmes sont corrects, en temps et en longueur. On va aussi vérifier les cas limites tels que des chemins de longueur nulle ou inexistant. Afin de réaliser les tests de distances, je me suis servie des résultats de Bellman-Ford comme référence. Comme pour les tests unitaires sur Path et sur BinaryHeap, j’ai utilisé BeforeClass et Test afin de réaliser ces tests.

## Test 1 et 2 : Chemin existant

Dans ces deux premiers tests, on va vérifier si les deux algorithmes codes, Dijkstra et AStar renvoient les bonnes distances et les bons temps sur des chemins courts et simples sur différentes cartes routières.

Caractéristique :

|  |  |  |
| --- | --- | --- |
| Carte | Nœud de départ | Nœud d’arrivé |
| Carre  Guyane | 1  1 | 23  23 |

Sur un même chemin, on compare le résultat obtenu avec l’algorithme Bellman-Ford à celui obtenu avec les deux algorithmes que nous avons codés. On va utiliser la classe Asser :

Exemple d’utilisation de la fonction assert :

*assertEquals*((**long**) (*shortPathA*.getLength()), (**long**) (*shortPathB*.getLength()));

shortPathA étant issu de Dijkstra et shortPathB de Bellman-Ford. Ici on compare la distance trouvée.

Les premiers tests unitaires passent. On peut donc en déduire que sur tous les chemins courts routiers présents sur les cartes, nos algorithmes fonctionnent. Il faut donc maintenant s’intéresser aux cas plus spécifiques.

## Test 3 et 4 : Chemin non-existant ou nul

Il faut maintenant vérifier le comportement de notre algorithme s’il doit traiter des cas particuliers comme un chemin inexistant ou de longueur nulle. Tout d’abord en observant l’algorithme de Bellman-Ford, on s’aperçoit que si notre chemin n’a pas pu être créer ou s’il est nul, son statut est mis à INFEASIBLE. Il va donc falloir vérifier si les algorithmes Dijkstra et sa version améliorée ont eux aussi ce statut mis à jour. C’est ainsi qu’on va traiter ces deux types de chemins.

Caractéristique :

|  |  |  |  |
| --- | --- | --- | --- |
| Carte | Nœud de départ | Nœud d’arrivé | Type |
| Carre  Guyane | 1  11752 | 1  576 | Longueur nulle  Inexistant |

Afin de trouver un chemin inexistant, j’ai lancé notre application et est testé différent chemin sur la carte Guyane avec Bellman-Ford jusqu’à en trouver un inexistant. Pour les deux types de chemins, j’ai utilisé la fonction AssertsTrue avec le statut INFEASIBLE comme dans l’exemple ci-dessous :

*assertTrue*(*emptyPathA*.equals(AbstractSolution.Status.***INFEASIBLE***))

Ces deux tests passent. On peut donc en déduire que nos deux algorithmes ont un comportement semblable à celui de Bellman-Ford dès qu’ils sont en présence d’un chemin de longueur nulle ou d’un chemin inexistant.

1. Tester sans Bellman-Ford

Si l’algorithme de Bellman-Ford n’avait pas été codé, il n’aurait pas été possible de vérifier nos propres algorithmes de la même façon. Un des méthodes qu’on pourrait envisagée concernant les chemins existants seraient de comparées la valeur trouvée par nos algorithmes avec celle de Google Maps sur des cartes existantes. Par exemple sur la carte Guyane :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Carte | Nœud de départ | Nœud d’arrivé | Longueur | Temps |
| Guyane | 63 | 89 | 34,591 km | 29 min 13 s |

Avec Google Maps on obtient 35 km et 31 min, on peut donc en conclure que notre résultat est à peu près juste et que la légère erreur est dû à l’imprécision de notre logiciel comparé à celui de Google Maps.

Concernant les cas particuliers, l’algorithme de Bellman-Ford n’est pas requis pour vérifier Dijkstra et A\* car nous savons par avance quels résultats sont attendus pour les statuts des chemins et les couts de ceux-ci.

# Test d’optimalité

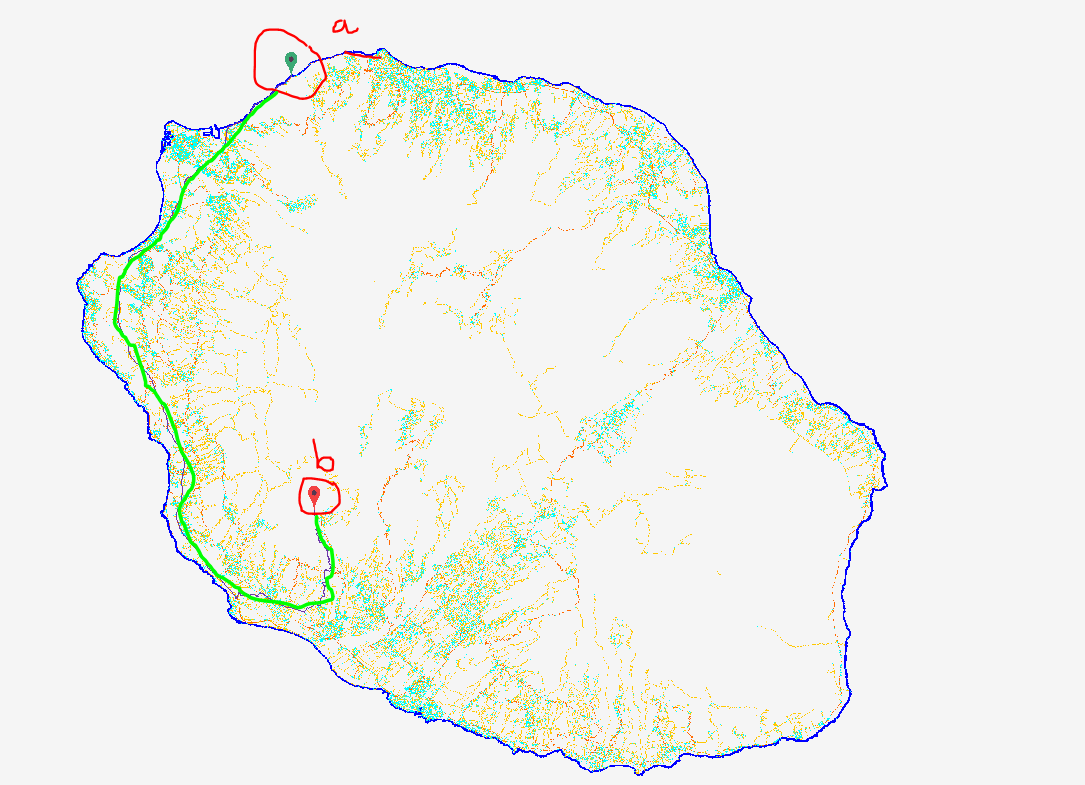
## Hypothèse sur les performances

A\* étant une version améliorée de Dijkstra, on s’attend à ce qu’il aille plus rapidement que celui-ci. Afin de vérifier cette hypothèse, on va prendre un chemin sur une carte globalement homogène comme la carte de la Côte d’ivoire. On observe ainsi que sur un chemin pris au hasard le temps vaut :

* Bellman-Ford : environ 6,997 s
* Dijkstra : environ 238 s
* A\* : environ 90 s

On s’aperçoit donc sur cet unique test qu’il existe des situations pour lesquelles A\* est plus performant que Dijkstra. Cependant, si la complexité de l’algorithme de Bellman-Ford semble élevé par rapport aux deux autres algorithmes, la différence entre les deux autres est beaucoup moins importante. Il doit donc exister des situations où Dijkstra est plus rapide que A\*.

A\* cherche un plus court chemin en tendant « à aller tout droit ». Ainsi, sur des cartes qui seraient plutôt de type circulaire nuiraient à l’efficacité de notre algorithme. J’ai ainsi réalisé un test sur la carte de la Réunion :



Entre le point a et le point b, le plus court chemin fait le tour de l’île à cause du volcan en plein milieu. Sur ce test, Dijkstra était environ deux fois plus rapide que A\*. A\* a perdu du temps à cause de cette tendance à vouloir aller « à vol d’oiseau ».

## Idées d’implémentation

Afin de tester les performances de nos algorithmes respectifs, il nous faut un jeu de donné extrêmement important. On va donc générer deux types de fichiers, un fichier d’entrée contenant toutes les données, c’est-à-dire une liste de couples de points pour une carte précise points, et un fichier de sortie qui contiendra les résultats des performances de nos algorithmes.

Afin de gérer ces fichiers, on pourrait générer deux nouvelles classes, DataGenerateur et DataReader pour lire et écrire dans les fichiers. On va enfin créer une dernière classe PerformanceTest qui utilise les deux classes ci-dessus.

Le format imaginé du fichier utilisé par la classe DataGenerateur est le suivant :

/Maps/guyane.mapgr

83;69

1024;459

…

On retrouverait aussi dans le fichier de sortie le format suivant :

Maps : Guyane ; Nœud de départ : 1024, ; Nœud d’arrivé 459

Temps Djikstra : 238s ; Nombre de nœud dans le tas : 5023

Temps A\* : 90s ; Nombre de nœud dans le tas 3026

Finalement dans notre classe PerformanceTest on ferait appel à nos deux algorithmes sur un jeu de données comme celui-ci et on mettrait en place un chronomètre qui nous permettrait d’évaluer le temps mis par chaque algorithme. De même, on récupèrerait le nombre de nœud traité dans le tas et on pourrait finalement analyser ces données.

# Problème ouvert

## Analyse de l’énoncé

J’ai choisi le problème ouvert numéro 1.1. L’énoncé est le suivant : Deux robots R1 et R2 habitent en 01 et 02 et doivent atteindre leurs destinations respectivement D1 et D2. En chemin, ils se croisent pour échanger leurs colis :

En orange est symbolisé le point de rencontre.

Le but de ce problème est de déterminer R.

Il faut que le coût = c(01R) +c( RD1) + c(O2R)+c(RD2)+ attente soit la plus petite possible. La variable attente est rajoutée car si les deux robots n’arrivent pas exactement en même temps au point R, un des robots perdra du temps à attendre l’autre.

Tout d’abord, afin de minimiser cette attente, il semble important que minimiser l’attente de ces deux robots. Il faut donc trouver un point à quasi-équidistance de O1 et de O2, qui soit bien sûre atteignable depuis les deux origines.

Ainsi on va tout d’abord observer quels sont les sommets atteignables à partir de O1 et quels sont ceux atteignables à partir de O2 :

Ici, R2 et R3 sont atteignables par O1 et par O2.

Afin de déterminer le meilleur point de rencontre, il faut donc aussi prendre en compte le temps entre le point de rencontre et les destinations respectives des deux robots.

## Algorithme général

Afin de rentabiliser tous les trajets, il faudrait lancer quatre algorithmes de Dijkstra simultanément chacun ayant pour origine respective O1, O2, D1 et D2. Ces algorithmes évolueraient concentriquement chacun leurs tours. Le premier point atteint par les quatre parcours seraient donc notre point d’échange de colis. Avant tous, il faudrait chercher le plus court chemin entre O1, D1 et entre O2, D2 afin de pouvoir gérer les cas particuliers.

## Implémentation

Il est nécessaire de modifier la classe Label déjà créer de manière à mémoriser quatre marquages différents (en effet, il en faudra un pour chaque Dijkstra lancer). Il faudra rendre aussi commun le tableau de Label aux quatre parcours, pour que les quatre algorithmes travaillent sur la même structure. Nos algorithmes s’arrêtent quand un des Label possède les quatre marquages. Il faudra aussi déterminer le moment où un algorithme de Dijkstra s’arrête afin de laisser la main au suivant.

## Cas Particulier

* Si l’une des origines n’est pas accessibles à partir de l’autre, il faudra lever une exception ou renvoyer un Label Null. De même si les destinations ne peuvent être atteintes par les robots (sauf s’ils pratiquent la téléportation)
* Si pour l’un des robots, sa destination et son origine son les mêmes, le point de rencontre sera donc sur cette origine :
* Si l’une des origines est situées sur le chemin de le plus court entre l’autre et sa destination, comme sur la figure ci-dessous.

Il faudra alors que le robot en O2 attende le robot qui provient de O1 avant de partir vers sa destination.

* Si la destination est la même pour le robot 1 et le robot 2, il faudra qu’ils se rendent directement à leurs destinations, l’échange se fera sue place.

# Conclusion