

Projet interdisciplinaire ou de recherche

Comparateur d'algorithmes de plus court chemin

Vincent ALBERT
Nicolas BÉDRINE

Année 2015–2016

Projet réalisé pour l'équipe Maia du laboratoire Loria

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : ALBERT, Vincent

Élève-ingénieur(e) régulièrement inscrit(e) en 2^e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 1205033068

Année universitaire : 2015–2016

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Benchmarking d'algorithmes de plus court chemins sur grilles générées aléatoirement

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Villers-lès-Nancy, le 19 mai 2016

Signature :

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : BÉDRINE, Nicolas

Élève-ingénieur(e) régulièrement inscrit(e) en 2^e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) :

Année universitaire : 2015–2016

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Benchmarking d'algorithmes de plus court chemins sur grilles générées aléatoirement

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Villers-lès-Nancy, le 19 mai 2016

Signature :

Projet interdisciplinaire ou de recherche

Comparateur d'algorithmes de plus court chemin

Vincent ALBERT
Nicolas BÉDRINE

Année 2015–2016

Projet réalisé pour l'équipe Maia du laboratoire Loria

Vincent ALBERT
Nicolas BÉDRINE
vincent.albert@telecomnancy.eu
nicolas.bedrine@telecomnancy.eu

TELECOM Nancy
193 avenue Paul Muller,
CS 90172, VILLERS-LÈS-NANCY
+33 (0)3 83 68 26 00
contact@telecomnancy.eu

Loria
615, Rue du Jardin botanique
54506, Vandœuvre-lès-Nancy
03 83 59 20 00



Encadrant : M. Olivier Buffet

Remerciements

Nous tenons à remercier toutes les personnes nous ayant accompagnés durant la réalisation de ce projet, et en particulier de M. Buffet pour nous avoir encadré durant toute la durée du projet et pour nous avoir prodigué ses précieux conseils.

Nous remercions aussi M. Jean-François SCHEID qui a supervisé l'organisation des PIDRs et de Mme Isabelle CHENET, secrétaire des PIDRs.

Enfin, nous remercions toute l'équipe pédagogique et administrative de TELECOM Nancy pour nous fournir un cadre d'études adéquat à la réalisation de ce projet.

Avant-propos

Ce rapport est le résultat d'un projet de découverte de la recherche de quatre mois effectué dans le cadre du sujet fourni par M. Olivier BUFFET, chercheur au Loria.

Le projet a été réalisé pour l'équipe Maia (MAchines Intelligentes Autonomes), groupe de recherches centré sur les comportements décisionnels intelligents. Ses principaux champs de recherches sont les systèmes multi-agents et complexes ainsi que les systèmes décisionnels incertains.

N'ayant que peu d'expérience dans le domaine de l'Intelligence Artificielle mais étant très intéressé par cette dernière, nous avons choisi ce sujet afin d'en approfondir nos connaissances. Étant donné notre méconnaissance de la plupart des algorithmes de plus courts chemin, nous nous sommes demandés quelles en sont les principales différences.

Notre étude portant sur l'implémentation des algorithmes ainsi que de leurs comparaison au travers de batteries de tests était donc l'occasion d'en apprendre plus sur leur fonctionnement.

Table des matières

Remerciements	iv
Avant-propos	v
Table des matières	vi
1 Introduction	1
1.1 État actuel du domaine de recherches et légitimité du sujet	1
1.2 Présentation du fonctionnement global et de l'utilisation du logiciel	1
1.2.1 Fonctionnement du projet	1
1.2.2 Les environnements	1
1.2.3 Les algorithmes	1
1.2.4 Lancer le projet	2
2 Matériel et Méthode	3
2.1 Déroulement du projet	3
2.1.1 Son organisation	3
2.1.2 Les technologies utilisées	3
2.2 Outils développés pour tester les algorithmes	3
2.2.1 Les environnements	3
2.2.2 Les différents algorithmes implémentés	6
2.2.3 Objet Évaluation et fichiers de log	11
2.2.4 Vue graphique	11
3 Résultats et interprétation	12
3.1 Un logiciel de test	12
3.2 Exécution des algorithmes sur les environnements	12
3.2.1 Présentation des résultats et comparaison de IDA* et RBFS	12
3.2.2 Interprétation des résultats	12
4 Discussion globale	14
4.1 Notre approche	14
4.2 Notre contribution	14
4.3 Les améliorations possibles	14

4.4 Les voies de recherche possibles	15
Glossaire	16
Annexes	18
A Première Annexe	18
B Seconde Annexe	19
C Troisième Annexe	20
D Quatrième Annexe	21
E Cinquième Annexe	22
Résumé	23

1 Introduction

1.1 État actuel du domaine de recherches et légitimité du sujet

Les algorithmes de plus court chemin appartiennent à la théorie des graphes et constituent un vaste champ de recherches de l'Intelligence Artificielle. Il existe de nombreux algorithmes qui ont été développés depuis les débuts de la Recherche Opérationnelle au début des années 1940, les plus connus étant Dijkstra et A*.

Cependant, bien que ces algorithmes soient connus et maîtrisés, il est difficile de déterminer *a priori* leur comportement sur différents types d'environnements, et donc de savoir lequel sera le plus efficace.

1.2 Présentation du fonctionnement global et de l'utilisation du logiciel

Le projet se propose donc de permettre de réaliser des séquences de tests de différents algorithmes sélectionnés par l'utilisateur sur des environnements différents générés aléatoirement.

1.2.1 Fonctionnement du projet

1.2.2 Les environnements

Les environnements correspondent aux graphes sur lesquels les algorithmes sont appliqués. Il existe deux types d'algorithmes : les algorithmes générés sous forme de grilles à N dimensions avec $N > 1$ (hypercubes) et des environnements générés de manière totalement aléatoires.

1.2.3 Les algorithmes

Plusieurs algorithmes de calcul de plus court chemin ont été choisis parmi différents algorithmes proposées afin de tester leur efficacité sur différents types d'environnements de différentes tailles. Nous avons choisis des algorithmes ayant des démarches et des volontés d'optimisations très différentes. Si certains vont trouver un compromis entre mémoire et temps de calcul, d'autres vont rechercher l'économie en mémoire. Certains de ces algorithmes ont d'ailleurs une recherche objective. Le but a été de rassembler des algorithmes au profil différent pour comparer leur comportement. Leur point de départ et d'arrivée sont définis aléatoirement. Les algorithmes parcourent le graphe en explorant ses nœuds par accès directs.

1.2.4 Lancer le projet

Afin de répondre à tout type de besoins, le jar exécutable du logiciel prend en paramètre plusieurs options permettant différents modes d'expérimentation.

En vue graphique

En lançant le logiciel avec la commande : `java -jar AlgoComparator.jar -view`, celui-ci se lance en mode graphique. Cependant cette option a été implémentée au début du développement, avant que nous nous rendions compte qu'elle ne nous était pas utile. Elle a donc été délaissée et n'est plus fonctionnelle.

En console avec dialogue utilisateur

En lançant le logiciel avec la commande : `java -jar AlgoComparator.jar -console`, celui-ci se lance en mode console et initie un dialogue avec l'utilisateur. Il lui permettra de créer un environnement, de sauvegarder la graine, de lancer une expérience et d'enregistrer les résultats dans un fichier de logs.

En console en mode de benchmarking

En lançant le logiciel avec la commande : `java -jar AlgoComparator.jar -bench`, celui-ci se lance en mode benchmarking. Il va charger la configuration dans le fichier `bench.conf` afin de déterminer la taille et le nombre de dimensions des environnements à générer. Un seul dialogue se lance avec l'utilisateur en début d'exécution pour lui demander les algorithmes qu'il veut lancer. Tous les résultats sont stockés dans le fichier `"logs.txt"`.
Note : l'option `-bench all` permet de lancer un benchmarking sur tous les algorithmes existants et supprime la séquence de dialogue.

Tester la génération d'environnements

Etant donné que nous avons rencontré des difficultés lors de la génération des environnements durant une certaine période du développement, nous avons aussi ajouté une option permettant de lancer un benchmarking sur la création d'environnements de 10 à plusieurs millions de points. Cette option est accessible avec la commande `java -jar AlgoComparator.jar -test consenv`. Il est aussi possible d'avoir le test de l'affichage graphique d'un environnement en remplaçant l'option `CONSENV` par l'option `GRAPHENV`. Ceci peut-être pratique pour vérifier que la création des liens entre les points se fait correctement.

2 Matériel et Méthode

Nous allons maintenant voir les procédés mis en place pour le bon déroulement du projet et les outils développés afin de parvenir au résultat final.

2.1 Déroulement du projet

2.1.1 Son organisation

Afin d'avoir le meilleur suivi possible de l'avancement du projet, une réunion a été organisée avec l'encadrant toutes les deux semaines durant toute la période de développement du projet. En dehors de ces réunions, nous nous rencontrons afin de développer le projet en commun. Le reste du temps, nous avons programmé à distance grâce à l'utilitaire git.

2.1.2 Les technologies utilisées

Le projet a été réalisé en Java sous Eclipse. Afin de travailler au mieux en équipe et de pouvoir gérer les versions de notre projet, nous avons utilisé le gestionnaire de version git.

2.2 Outils développés pour tester les algorithmes

Afin de tester les algorithmes, nous avons codé des outils nous permettant de tester l'efficacité des algorithmes que nous allons présenter ci-dessous.

2.2.1 Les environnements

Le premier et plus important de ces outils est le générateur d'environnements permettant de tester un algorithme sous plusieurs configurations possibles.

Présentation générale de la conception des environnements : listes d'adjacence

Tous les environnements sont conçus de la même façon. Ce sont des ensembles de places (représentées par N coordonnées pour N dimensions) reliées entre elles par des liens pondérés. Afin de représenter ces liens, nous possédons une liste d'adjacence associant à chacun des points la liste de ses successeurs. Chacune des dimensions possède une borne supérieure et inférieure.

Les environnements aléatoires

Au début du projet, nous voulions générer aléatoirement des graphes orientés. Cependant nos premiers échecs furent infructueux ; nos graphes n'étaient en effet jamais suffisamment connexes pour qu'une solution existe. Nous avons alors recherché un moyen de générer un environnement aléatoire, orienté et fortement connexes, mais ils semblerait que cela soit encore du domaine de la recherche. Ceci n'étant pas l'axe principal de notre projet, nous avons décidé d'abandonner ce type de graphe afin de nous concentrer sur la génération de grilles.

Les grilles et hypercubes

Le principal type d'environnement utilisé sont donc les grilles à N dimensions. A partir des bornes données par l'utilisateur et le nombre de places appartenant à chaque dimension du graphe, les places sont générées de manière régulière sur chaque dimension. En parallèle de la création des points, chacun d'entre eux est relié à son voisin. On sait que dans un hypercube, tous les points ont pour voisins directs les points ayant exactement un coordonnées différente. A partir de là il est facile de générer le lien entre la place courante et tous les points qui ont été précédemment générés.

Les liens sont pondérés pseudo-aléatoirement à partir d'un RNG alimenté par une graine spécifique (dans notre cas, le timestamp).

Génération d'environnement par graine : l'objet Seed

Afin d'enregistrer facilement les données relatives à un graphe sans enregistrer la totalité des points (ce qui pourrait être une perte considérable de place dans le cas d'un environnement à plusieurs millions de points), nous enregistrons toutes les données nécessaires à la génération d'un environnement dans un fichier texte, ce qui nous permettra par la suite de recréer exactement le même environnement sans sauvegarder tous les points.

Les données enregistrées sont la graine (le timestamp lors de la première création de l'environnement), le nombre de points pour chacune des dimensions (nous sommes dans le cas des grilles et hypercubes) ainsi que la borne min et la borne inf des points (toutes les bornes sont considérées comme identiques, les hypercubes sont donc réguliers).

L'objet Seed de notre programme permet, à partir d'un fichier texte formaté, de reproduire à l'identique le graphe d'un environnement.

Les dimensions non-uniformes

Il a été décidé d'implémenter un système de dimension non uniforme. La valeur d'un chemin pouvant varier selon sa position sur la carte, quand bien même sa distance euclidienne n'aurait pas changer. Cela permet de simuler la présence sur une carte d'un relief difficile.

Les refactorisations du modèle des environnements

Étant donné qu'au début nous désirions créer des environnements aléatoires, l'ensemble du code de nos Environnements se basait sur les positions des places dans le graphe. Or de cette manière, des parcours supplémentaires étaient obligatoires afin de rechercher des places par leurs coordonnées, ce qui ralentissait considérablement notre code.

Cependant une fois les graphes aléatoires abandonnés, nous nous sommes rendus qu'il était bien plus facile et rapide de générer les graphes en grille à partir de leurs indices. Nous avons donc développé une seconde version plus efficace (étant donné que le nombre de dimensions et de points par dimension est connu à l'avance).

Cependant en effectuant des recherches, nous avons vu le principe de génération d'hypercubes de dimension N récursivement par fusion d'hypercubes de dimension $N-1$. Nous avons donc décidé d'adapter ce principe à nos grilles en N dimensions et à M points.

Voici les images exposant ce principe et dont nous nous sommes inspirés :

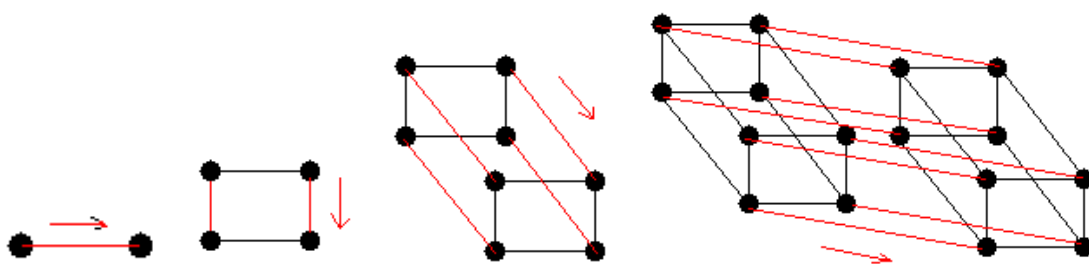


FIGURE 2.1 – Génération d'un hypercube en 4 dimensions

Nous avons donc refactorisé le code des Environnements afin d'appliquer cette modification, et la différence fut flagrante. Alors qu'il était impossible de générer un environnement d'un million de points dans la première version (même après plusieurs heures), le nouvel algorithme était capable de le faire en moins d'une dizaine de secondes.

Voici l'algorithme que nous avons implémenté :

Data: M : tableau qui associe à chaque dimension i son nombre de points, N : nombre de dimensions du graphe, borne_inf : borne inférieure de toutes les dimensions, borne_sup : borne supérieure de toutes les dimensions, i : compteur correspondant à la dimension courante, j : compteur correspondant au nombre de points de la dimension courante, graph : graph actuel que l'on fait évoluer

Result: Hypercube de N dimensions à M[i] points chacune

graph = new Point(nb_dim = N, coordonnées = {borne_min, ...}) //On crée le point extremum
i = 0

while i < N **do**

 //On calcule la distance entre les points de la dimension courante

 distance = calculerDist(M[i], borne_inf, borne_sup)

 //On duplique le graphe le nombre de fois qu'il faut

for j=0; j<M[i]; j++ **do**

 tmp = dupliquer(graph)

 //On décale les coordonnées de la copie de la distance qui existe entre les points

 glisser_coordonnées(tmp, distance)

 copies_graph.push(tmp)

end

 graph = fusion(graph, copies_graph) //On fusionne toutes les copies ensembles

end

Algorithm 1: Algorithme de génération d'une grille à N dimensions

Voici un tableau comparatif des différentes versions de l'algorithme de création de cartes à N dimensions :

Version	10 ²			10 ⁴			10 ⁶			10 ⁷		
	2D	3D	4D	2D	3D	4D	2D	3D	4D	2D	3D	4D
1 ^{ère} version	5	-	-	10 ⁷	10 ⁸	10 ⁸	+∞			+∞		
2 ^{ème} version	2	-	-	34	36	41	16 864	24 247	32 589	53 789	74 252	83 736
3 ^{ème} version	2	-	-	13	16	23	368	512	746	14 713	21 697	37 250

TABLE 2.1 – Tableau récapitulatif des benchmarkings

2.2.2 Les différents algorithmes implémentés

Nous allons maintenant présenter les différents algorithmes mis en place dans notre projet. Pour chaque algorithme, nous commencerons par une présentation générale de l'algorithme, puis nous détaillerons son fonctionnement. Le pseudo-code exhaustif du premier algorithme est présenté. Celui des autres est disponible en annexe.

A*

L'algorithme A* lance une recherche orientée vers une destination précise. Cette recherche est biaisée par une heuristique qui va permettre de développer certain chemin plutôt que d'autre.

Les calculs s'effectuent sur deux listes, représentant les nœuds parcourus et la frontière, ensemble des nœuds candidat pour les futures recherches. Les nœuds possèdent un coût $f=g+h$, où g est la distance courante parcouru lors de l'exploration, et h la distance heuristique du nœud à la destination. Ce coût sera initialisé pour tout les nœuds à l'infini et affiner lorsque les nœuds seront rencontrer.

Pour la suite du descriptif de A^* , la liste des nœuds explorés sera appeler nœuds fermés et la frontière, les nœuds ouverts. Initialement, il n'y a pas de nœud fermés et seul la source est nœud ouvert. Jusqu'à ce que la liste des nœuds ouverts soit vide, ou que la destination soit atteint, on sélectionne dans les nœuds ouverts, le nœuds présentant le meilleur coût f . On extrait ce nœud des nœuds ouverts, puis est ajouté aux nœuds fermés. Des traitements sont ensuite effectués sur les nœuds adjacents à ce nœud, sauf si ces nœuds sont des nœuds fermés. Si le nœud n'est pas dans la liste des nœuds ouverts, il y est ajouté. Son coût est aussi calculé, par somme du coût du chemin courant effectué par l'algorithme (égal au coût du nœud considéré en premier lieu additionné au coût du chemin entre nœud considéré et nœud adjacent), et l'approximation de la distance du nœud à la source. Si le nœud est déjà dans les nœuds ouverts, on met à jour son coût,

si le coût du chemin actuelle est plus intéressant que le coût du nœud.

Data: NoeudOuvert : liste des nœuds ouverts, NoeudFermé : liste des nœuds explorés,
Prédécesseur : tableau du nœud à atteindre pour atteindre un nœud dans le cas du plus court chemin, N : nombre total de point du graphe, Source : nœud source, Destination : nœud destination

Result: Parcours : Liste ordonnée du plus court chemin de la Source à la Destination

for $i=0; i<N; i++$ **do**

 Prédécesseur[noeud(i)] = noeud[i]

end

NoeudOuvert <- Source

$g(\text{Source}) = 0$

while *NoeudOuvert n'est pas vide* **do**

 NoeudCourant = minf(*NoeudOuvert*) //On sélectionne le nœud de Noeud ouvert qui a le plus faible coût $f=g+h$

if *NoeudCourant = Destination* **then**

 Current = Prédécesseur[Destination] **while** *Current != Source* **do**

 Parcours <- Current

 Current = Prédécesseur[Current]

end

 //Fin de l'algorithme

end

 Retirer(*NoeudOuvert*, NoeudCourant)

 NoeudFermé <- NoeudCourant

for *NoeudAdjacent dans noeudadjacent(NoeudCourant) AND NoeudAdjacent n'est pas dans NoeudFermé* **do**

 Nouveaug = $g(\text{NoeudCourant}) + \text{Chemin}(\text{NoeudCourant} + \text{NoeudAdjacent})$

if *NoeudAdjacent n'est pas dans NoeudOuvert* **then**

 NoeudOuvert <- NoeudAdjacent

else if *Nouveaug > g(NoeudAdjacent)* **then**

 continue

$g(\text{NoeudAdjacent}) = \text{Nouveaug}$

$f(\text{NoeudAdjacent}) = \text{Nouveaug} + f(\text{NoeudAdjacent})$

 Prédécesseur[NoeudAdjacent] = NoeudCourant

end

end

print("Plus de noeud ouvert ! Chemin non trouve !")

Algorithm 2: Algorithme A*

Dijkstra

L'algorithme de Dijkstra nous a servis de référence tout au long de ce projet. C'est un algorithme simple à implémenter, mais sa recherche non orientée devient un inconvénient. Lors d'un parcours, il ne recherche pas le chemin le plus court vers un point précis, mais analyse la totalité du graphe, et calcule le plus court chemin pour tous les points du graphe. On perd ainsi beaucoup de calcul pour des informations non nécessaires. L'algorithme renvoie un tableau de résultat, qui possède une entrée pour chaque nœud du graphe. A chaque nœud dans le tableau correspond le nœud à atteindre avant ce premier dans le cadre du plus court chemin.

L'utilisation d'une liste des nœuds explorés permet à l'algorithme de connaître sa progression. A chaque nœud est associé une distance à la source, initialisée à l'infini pour tous les nœuds, sauf pour le nœud source lui-même, initialisée à 0. Jusqu'à ce que tous les nœuds du graphe soit dans liste des nœuds explorés, on va considérer le nœud qui à la plus petite distance et qui n'a pas encore exploré. On va mettre ce nœud dans la liste des nœuds explorés. On met à jour la distance à la source de tous ces nœuds adjacents qui ne sont pas dans la liste des nœuds explorés. Cette mise à jour consiste à attribuer à cette distance le minimum entre sa valeur actuelle et l'addition de la distance à la source du nœud considéré et la distance entre le nœud considéré et le nœud adjacent. Ce calcul permet de savoir si le nœud considéré offre ou non un chemin plus court vers ses nœuds adjacents. En cas de modification de la distance à la source d'un nœud adjacent, on sait alors que le plus court chemin pour atteindre ce nœud passera par le nœud considéré.

IDA* (Iterative deepening A*)

L'algorithme IDA* est une modification de l'algorithme A* étudié précédent. Il donc pour objectif d'établir le plus court chemin vers une destination précise. Il est aidé pour cela d'une heuristique qui le conduit à une recherche biaisée. Ce qui le différencie d'A*, c'est l'absence de liste pour retenir les nœuds visités et il ne possède pas de réel frontière.

En absence des ces informations, l'algorithme doit parcourir plusieurs fois le graphe à partir du départ vers la destination. Il possède un maximum, correspondant au meilleur chemin trouvé sur le parcours précédent. A chaque parcours, l'algorithme tout les chemins jusqu'à ce que le maximum soit dépassé. Les coûts de tous les chemins sont alors remontés, le plus petit est considéré comme nouveau maximum, et les recherches sont réinitialisées au départ. On répète le processus jusqu'à atteindre la destination.

UCS (Uniform Cost Search)

L'algorithme Uniform Cost Search peut être perçu comme une fusion entre les algorithmes Dijkstra et A*. Il garde la démarche de Dijkstra dans l'exploration du graphe, c'est-à-dire une exploration objectif de tout le graphe, mais qui va s'arrêter une fois la destination trouvée. Il exploite deux structures de donnée, des listes qui représente les nœuds explorés et une frontière, comme A*. L'algorithme s'avère assez coûteux en mémoire, car chaque nœud explorée doit contenir le plus chemin qui mène à lui, partant de la source.

L'algorithme débute son exploration à partir de la source, qui est alors le seul nœud présent dans la frontière. Il considère le nœud en question et le place dans la liste des nœuds explorés. Tous les nœuds adjacents au nœud source sont alors placés dans la frontière, leur coût d'accès mis à jour avec le coût du chemin courant, et leur plus court chemin sont uniquement constitué du nœud source. Jusqu'à ce que le nœud considéré soit la destination, on considère dans la frontière le nœud le plus accessible, pour le placer dans la liste des nœuds explorés. Les nœuds qui lui sont adjacents sont ajoutés à la frontière avec un court plus chemin égal au plus court chemin du nœud considéré auquel on ajoute ce dernier, sauf si ils ont été déjà explorés. Dans ce dernier cas, on met à jour leur plus court chemin si il s'avère que l'on vient de trouver un meilleur chemin pour ceux-ci.

RBFS (Recursive Best First Search)

L'algorithme RBFS se focalise sur la recherche du plus court chemin vers une destination précise à l'aide d'une exploration biaisée par une heuristique. Les structures de donnée de l'algorithme se base sur une liste des nœuds du graphe. La structure de donnée utilisée pour les nœuds permet le stockage d'un plus court chemin vers celui-ci qui pourra être mis à jour au fils des recherches.

La particularité de cet algorithme est d'être récursif. Cette récursivité est utile à l'exploration à chaque de plusieurs voies possibles et terminer quel choix est le meilleur. A partir d'un nœud considérée, on met à jour les valeurs les coûts d'accès à ces successeurs (constituer de la valeur nécessaire pour atteindre le nœud lors du chemin considéré et d'une approximation de la distance du nœud à la destination), ainsi que la liste du plus court chemin pour atteindre ces successeurs. On va ensuite développer la voie qui présente le coût le plus intéressant (c'est à dire considérer le nœud successeur ouvrant sur cette voie), tout en gardant en mémoire la deuxième meilleur possibilité. Si il s'avère qu'au fils du développement du chemin son coût devient supérieur à la seconde possibilité, on arrête de développer ce chemin, et on remonte le coût du chemin développé, et on développe le second chemin, et on garde en mémoire le nouveau seconde meilleur coût (qui peut être le coût remonté de la première tentative). On considère au début la source et on termine lorsque la destination est considérée.

SMA* (Simplified Memory Bounded A*)

L'algorithme SMA* a pour objectif de palier au problème de mémoire que peut entraîner A*, dans le sens au celui-ci peut remplir la mémoire au fils de ses recherches. Si cela arriver dans la cas de SMA*, l'algorithme est capable d'effacer les informations de certaines nœuds, en retenant les coûts de chemins possibles en passant par ces nœuds. Il sera ainsi possible de développer le nœud si la recherche démontre que passer par ce nœud devient intéressant. Dans le modèle que nous avons développé, nous utilisé une mémoire virtuelle pour simuler plus facilement la saturation de celle-ci. Outre cela, l'algorithme utilise une pile et une structure de données pour le stockage des nœuds en mémoire. Cette structure de donnée permet de stocker le coût pour atteindre ce nœud depuis la source, mais aussi l'approximation du coût pour atteindre la destination, une liste de ces successeurs et de ces ancêtres. SMA* propose aussi une profondeur maximum, pour n'exploiter les nœuds qui ne sont qu'à une certaine distance de la source.

La queue est initialement remplis par le seul nœud source. Cette queue est toujours trié par coût d'accès au nœud. (Selon la formule $f=g+h$, où g est le coût du chemin pour atteindre le nœud calculé pendant l'exploration, et h une approximation de la distance du nœud à la destination). C'est le nœud avec le plus petit coût d'accès qui sera considéré (l'algorithme s'arrête quand on arrive ce nœud est la destination), on l'appellera nœud courant. On commence à considérer son prochain successeur, c'est-à-dire dans la liste de ces successeurs, celui qui n'a pas encore été considéré. On utilise un pointeur attaché au nœud pour savoir cela. Si la profondeur du nœud dépasse la limite (la profondeur étant la taille du chemin courant depuis la source vers ce nœud), on le place dans la queue, et on lui attribut la valeur infini pour f , sinon on met à jour cette valeur normalement (pour rappel $f=g+h$, où ici g est le coût pour aller jusqu'au nœud courant, additionné au coût du chemin entre le nœud courant et le nœud considéré, et h , une approximation de la distance du nœud considéré à la destination). Si ce nœud est le dernier successeur à considérer, on peut informer le nœud courant et ces ancêtres de la meilleur possibilité de chemin trouvée, et si tout les successeurs du nœud courant sont en mémoire, on peut enlever le nœud courant de la queue.

En cas de saturation de la mémoire, on va décharger le nœud de la queue avec la plus mauvais coût f , et l'enlever de la liste des successeurs de son ancêtre, tout en faisant remonter son coût. On peut ainsi reconsidérer ce chemin si nécessaire. On recharge l'ancêtre en mémoire si celui-ci a été déchargé.

2.2.3 Objet Évaluation et fichiers de log

Le second outil nous permettant de comparer les algorithmes entre eux sont les objets Évaluation récoltant les données relatives à l'exécution de l'algorithme. Ces données sont les suivantes :

- Le nombre de nœuds "visités" avant de trouver la première solution
- Le nombre total de nœuds "visités".
- Le nombre de nœuds "explorés".
- Le nombre de solutions trouvées par l'algorithme.
- La liste du coût pour toutes les solutions trouvées.
- Le nombre de nœuds appartenant au chemin de chacune des solutions.
- La liste du nombre de nœuds "visités" pour chaque solution.

Ce sont sur elles que se baseront les comparaisons entre les algorithmes.

2.2.4 Vue graphique

Lors de la première phase de développement, nous avons réalisé une interface graphique permettant à l'utilisateur de modifier toutes les options possibles et de lancer le déroulement d'un algorithme avec une interface graphique. Cependant cela n'était que fonctionnel pour les environnements en deux dimensions et ne correspondaient pas à nos besoins réels, nous avons donc abandonné le développement de cette fonctionnalité qui est désormais obsolète.

3 Résultats et interprétation

3.1 Un logiciel de test

L'ensemble du projet a pris la forme d'un logiciel permettant la création d'environnement avec génération aléatoire d'une source et d'une destination, l'exécution d'algorithme sur ces environnements, et le calcul de performances de ces exécutions. Parmi les algorithmes cités plus haut, seul SMA* n'a pas passé les tests avec succès, et reste pour le moment en développement.

Le programme nous offre la possibilité d'effectuer des séries de tests sur des environnements différents. Cela nous est utile pour établir un descriptif des avantages et inconvénients des algorithmes et décrire leur comportement face à différentes situations et cela grâce à l'évaluation des performances.

3.2 Exécution des algorithmes sur les environnements

Nous avons décidé de comparer entre eux les algorithmes partageant certaines caractéristiques communes.

Dans un premier temps, nous comparerons donc IDA* et RBFS, ces deux algorithmes ayant une distance limite. Étant donné la vitesse d'exécution de RBFS, nous limiterons la dimension des grilles sur lesquelles ils seront déroulés à 25 noeuds. Ensuite, nous comparerons les trois derniers algorithmes restants, Dijkstra, A* et Uniform Cost Search. UCS est semblable à un A* avec une heuristique à 0, et UCS est semblable à Dijkstra mais s'arrête quand il atteint l'objectif.

3.2.1 Présentation des résultats et comparaison de IDA* et RBFS

Résultats et comparaison de IDA* et RBFS

Version	Critère 1			Critère 2			Critère 3			Critère 4		
	2D	3D	4D	2D	3D	4D	2D	3D	4D	2D	3D	4D
IDA*												
RBFS version												

TABLE 3.1 – Tableau récapitulatif des benchmarkings sur IDA* et RBFS

Résultats et comparaison de Dijkstra, A* et UCS

3.2.2 Interprétation des résultats

Version	Critère 1			Critère 2			Critère 3			Critère 4		
	2D	3D	4D	2D	3D	4D	2D	3D	4D	2D	3D	4D
Dijkstra												
A*												
UCS												

TABLE 3.2 – Tableau récapitulatif des benchmarkings sur Dijkstra, A* et UCS

4 Discussion globale

4.1 Notre approche

Ce que nous devons réaliser est un outil capable de calculer les performances sur certains algorithmes d'intelligence artificielle, afin de pouvoir comparer ces performances. Nous avons donc besoin de créer un outil de création de carte, permettant de créer des environnements aux propriétés diverses (nombre de dimensions, uniformité ou non des dimensions), où les algorithmes peuvent s'exécuter, et ainsi révéler les spécificités de chacun, et leur réaction face à des situations non triviales.

4.2 Notre contribution

Nous avons mis au point un logiciel capable à la fois de créer des cartes de tests aléatoires à partir de graine gardée en mémoire, puis d'exécuter les algorithmes sur ces cartes dans la recherche d'un plus court chemin entre une source et une destination. Ce programme possède une multitude de configuration qui permette d'effectuer une série de test à la suite pour des cartes différentes. Il est donc possible de le lancer sur des machines dotées de fortes puissance de mémoire et/ou de calcul, comme des clusters, pour générer des environnements de million de points pour y effectuer des tests.

Nous avons effectués nos propres tests, liés à notre propre système de calcul de performance. Ceux-ci nous ont permis de vérifier les propriétés théoriques des algorithmes, ainsi que les avantages et inconvénients qu'ils peuvent apporter. Ainsi nous avons prouvé que même si IDA* semble être une bonne alternative à A* pour des systèmes ne disposant pas de beaucoup de mémoire, celui-ci devra être muni d'une bonne puissance de calcul en compensation, que des algorithmes comme RBFS offre peu d'alternative à des algorithmes comme A*, du fait de sa forte récursion.

4.3 Les améliorations possibles

Du fait de son architecture respectant cloisonnement les différents types de modèle, notamment les cartes, les algorithmes d'intelligence artificielle et les critères d'évaluation de performance, il est tout à fait possible d'implémenter de nouveaux algorithmes, de nouveaux critères à l'évaluation, mais aussi de nouveaux moyens de générer des cartes. On peut penser à l'ajout de critères à la création de ces cartes, mais aussi de nouveaux algorithmes pour générer les cartes.

Une vue graphique est disponible, mais inachevée, car son développement a été abandonné pour se concentrer sur une vue en console. Elle permettait alors de voir les cartes générées. Notre objectif était d'afficher le parcours de l'algorithme sur la carte en mettant en surbrillance les nœuds considérés et les nœuds mis-à-jour. Il est toujours possible de compléter cette interface,

elle serait très utile pour comprendre l'interaction d'algorithme comme A^* , face à certains types de cartes complexes.

L'algorithme SMA* est implémenté, mais n'a pas encore passé avec succès les tests. Une mémoire simulée est déjà disponible, ainsi que le reste du code, et le calcul des critères de performance. Il ne reste qu'à déboguer la démarche de l'algorithme pour que celui-ci retourne des résultats exploitables.

4.4 Les voies de recherche possibles

Nous avons effectués de recherche sur les méthodes de génération de carte aléatoire. Nous n'avons pas trouvé de réel méthode à la création de graphe aléatoire connexe, la connexité étant une propriété nécessaire à nos graphes. Sans cela, il serait possible de n'avoir aucun chemin entre deux points pris aléatoirement dans le graphe. Développer ces applications nous seraient utiles pour supprimer notre dépendance aux grilles, et en apprendre d'avantage sur les comportements de nos algorithmes d'intelligence artificielle.

Nous avons remarqué que nos algorithmes d'intelligence artificielle pouvait privilégier la mémoire à la puissance de calcul, comme c'est le cas pour IDA* et RBFS développé à la fin des années 80, quand la mémoire était encore cher. Il serait intéressant de voir si la création d'algorithme de recherche du plus court chemin orientée (c'est-à-dire avec une heuristique) privilégiant la puissance de calcul au détriment de la mémoire, qui serait alors surexploitée.

Glossaire

- UCS : Uniform Cost Search
- RBFS : Recursive Best For Search
- IDA* : Iterative Deepening A*
- SMA* : Simplified Memory-bound A*

Annexes

A Première Annexe

Data: Exploration : liste des nœuds explorés, Prédécesseur : tableau du nœud à atteindre pour atteindre un nœud dans le cas du plus court chemin, Distance : tableau de la distance des nœuds à la source, N : nombre total de point du graphe, Source : nœud source

Result: Tableau Distance et Prédécesseur remplis

```
for  $i=0; i<N; i++$  do
    Prédécesseur[noeud(i)] = noeud[i]
    Distance[noeud(i)] = infini
end
Distance[Source] = 0

while  $size(Exploration) < N$  do
    NoeudCourant = mindistance(Exploration, Distance) //On sélectionne le nœud qui a la plus
    courte distance à la source et qui n'a pas encore été exploré

    for NoeudAdjacent dans noeudadjacent(NoeudCourant) do
        if NoeudAdjacent n'est pas dans Exploration then
            if  $Distance(NoeudCourant) + Chemin(NoeudAdjacent, NoeudCourant) <$ 
             $Distance(NoeudAdjacent)$  then
                Distance(NoeudAdjacent) = Distance(NoeudCourant) + Chemin(NoeudAdjacent,
                NoeudCourant)
                Prédécesseur(NoeudAdjacent) = NoeudCourant
            end
        end
    end
end
```

Algorithm 3: Algorithme de Dijkstra

B Seconde Annexe

Data: limite : valeur que l'on s'impose à chaque recherche, Source : nœud source, Destination : nœud destination

Result: Parcours : Liste ordonnée du plus court chemin de la Source à la Destination
limite = h(Source)

```
while true do
  retour = recherche(Source, 0, limite)
  if retour = -1 then
    break
  end
  //Fin de l'algorithme
  limit = retour
end

Function recherche (noeud, g, limite)
  f = g + h(noeud)
  if f > limit then
    return f
  end
  if noeud = Destination then
    Parcours <- noeud
    return -1
  end
  min = infini
  for NoeudAdjacent dans noeudadjacent(noeud) do
    retour = Recherche(NoeudAdjacent, g+Chemin(noeud, NoeudAdjacent), limit)
    if retour = -1 then
      Parcours <- NoeudAdjacent
      return -1
    end
    if retour < min then
      min = retour
    end
  end
  end
  return retour
```

Algorithm 4: Algorithme IDA*

C Troisième Annexe

Data: Exploration : liste des nœuds explorés, Frontière : liste des nœuds ouverts, Distance : tableau de la distance des nœuds à la source, Parcours : tableau des plus court chemin pour atteindre un nœud, N : nombre total de point du graphe, Source : nœud source, Destination : nœud destination

Result: Parcoursfinal : Liste ordonnée du plus court chemin de la Source à la Destination

Frontière <- Source

while true **do**

 NoeudCourant = mindistance(Frontière) //On sélectionne le nœud qui a la plus courte distance à la source

if NoeudCourant = Destination **then**

 | Parcoursfinal = Parcours[NoeudCourant]

end

 //Fin de l'algorithme

 Retirer(Frontière, NoeudCourant)

 Exploration <- NoeudCourant

for NoeudAdjacent dans noeudadjacent(NoeudCourant) **do**

 NouvelleDistance = Distance[NoeudCourant] + Chemin(NoeudCourant, NoeudAdjacent)

 NouveauParcours = Parcours[NoeudCourant] + NoeudCourant

if NoeudAdjacent n'est ni dans Exploration ni dans Frontière **then**

 | Frontiere <- NoeudAdjacent

 | Parcours[NoeudAdjacent] = NouveauParcours

 | Distance[NoeudAdjacent] = NouvelleDistance

else if NoeudAdjacent est dans Frontière et Distance[NoeudAdjacent] > NouvelleDistance **then**

 | Parcours[NoeudAdjacent] = NouveauParcours

 | Distance[NoeudAdjacent] = NouvelleDistance

end

end

Algorithm 5: Algorithme Uniform Cost Search

D Quatrième Annexe

Data: RETOUR : structure de donnée de deux type décrivant si la recherche est un échec et la taille du chemin, limite : limite de recherche, Parcours : tableau du plus court chemin, Noeuds : liste de tous les noeuds, Source : nœud source, Destination : nœud destination

Result: Parcoursfinal : Liste ordonnée du plus court chemin de la Source à la Destination

$g(\text{source}) = 0$

RBFS(Noeuds, Source, infini)

Function *RBFS(Listenoeud, noeud, limite)*

```
    if noeud = Destination then
        Parcoursfinal = Parcours[Destination]
    end
    for NoeudAdjacent dans noeudadjacent(noeud) do
         $g(\text{NoeudAdjacent}) = g(\text{noeud}) + \text{Chemin}(\text{NoeudAdjacent}, \text{noeud})$ 
         $f(\text{NoeudAdjacent}) = g(\text{NoeudAdjacent}) + h(\text{NoeudAdjacent})$ 
    end
    Successeurs = noeudadjacent(noeud)
    while true do
        tri(Successeurs) //On tri les noeuds par valeur croissante de f
        if isEmpty(Successeurs) then
            return RETOUR(true, infini)
        else
            Parcours[Successeurs[0]] = Parcours[noeud] + noeud
        end
        if  $f(\text{Successeurs}[0]) > \text{limite}$  then
            return RETOUR (true, Successeurs[0])
        end
        if  $\text{size}(\text{Successeurs}) = 1$  then
            result = RBFS(Listenoeud, Successeurs[0], limite)
        end
        result = RBFS(Listenoeud, Successeurs[0], min(limite, f(Successeurs[1])))
        if estSucces(result) then
            return result
        end
    end
    return retour
```

Algorithm 6: Algorithme RBFS

E Cinquième Annexe

Data: Queue : liste des nœuds ouverts, Mémoire : liste à capacité limitée, T : Taille de la mémoire, DepthMax : Taille maximum d'un chemin en nombre de nœud, Source : nœud source, Destination : nœud destination

Result: Parcours : Liste ordonnée du plus court chemin de la Source à la Destination

Queue <- Source

Mémoire <- Source

while true **do**

Tri(queue) // On tri la queue par f croissant NoeudCourant = Queue(0) //On sélectionne l'élément de la queue qui a le plus faible coût $f=g+h$

if NoeudCourant = Destination **then**

Current = Ancêtre(Destination)

while Current != Source **do**

Parcours <- Current

Current = Ancêtre(Current)

end

end

//Fin de l'algorithme

if Tous les successeurs de NoeudCourant ont été générés **then**

Retirer(Queue, NoeudCourant)

else

NoeudAdjacent = SuccesseurNonConsidéré(NoeudCourant)

Mémoire <- NoeudAdjacent

Queue <- NoeudAdjacent

$g(\text{NoeudAdjacent}) = g(\text{NoeudCourant}) + \text{Chemin}(\text{NoeudCourant}, \text{NoeudAdjacent})$

if Profondeur(NoeudAdjacent) > DepthMax **then**

$f(\text{NoeudAdjacent}) = \text{infini}$

else

$f(\text{NoeudAdjacent}) = g(\text{NoeudAdjacent}) + h(\text{NoeudAdjacent})$

end

if NoeudAdjacent est le dernier successeur de NoeudCourant à être considéré **then**

//Informer NoeudCourant et ses ancêtres du meilleur f des successeurs de NoeudCourant

end

if Tous les successeurs de NoeudCourant sont en mémoire **then**

//Informer NoeudCourant et ses ancêtres du meilleur f des successeurs de NoeudCourant

end

if size(Mémoire) == T **then**

Tri(queue)

Asupprimer = Queue(size(Queue)-1)

Retirer(Queue, Queue(size(Queue)-1))

Retirer(Successeurs(Ancêtre(Asupprimer)), Asupprimer)

if Ancêtre(Asupprimer) n'est pas dans la queue **then**

Queue <- Ancêtre(Asupprimer)

end

Retirer(Mémoire, Asupprimer)

end

end

end

Algorithm 7: Algorithme SMA*

Résumé

Le sujet en bref. Le sujet qui a été choisi porte sur la réalisation d'un logiciel de benchmarking permettant de tester et de comparer différents algorithmes de recherche de plus court chemin sur des environnements diversifiés générés aléatoirement sur des séquences de tests paramétrées dans un fichier de configuration.

Les fonctionnalités implémentées.

Les résultats.

Interprétation. Mots-clés : algorithmes de plus court chemin, théorie des graphes, environnements aléatoires, benchmarking, interface utilisateur, graines.