UNIVERSITÉ DU QUÉBEC À MONTRÉAL

REPLANIFICATION EFFICACE DE CHEMIN DANS UN MONDE DE JEU FORTEMENT DYNAMIQUE

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE - CONCENTRATION EN INTELLIGENCE ARTIFICIELLE

PAR

KATIA BEAUVAIS-MONESTIME

MARS 2026

À Marc-Antoine,

pour ton support constant.

REMERCIEMENTS

J'aimerais remercier mon directeur de recherche, le professeur Éric Beaudry, qui m'a acceptée comme étudiante. Son expertise et son soutien m'ont permis de persévérer et d'approfondir mes connaissances en recherche. C'est grâce à sa flexibilité que j'ai pu avoir la confiance de continuer.

Je remercie aussi mes collègues de mon laboratoire de recherche, qui m'ont donné des conseils afin de diriger ma recherche. Un merci particulier à Guillaume Gosset, qui m'a conseillé sur les améliorations possibles de la contraction hiérarchique dans un contexte dynamique et qui a toujours été ouvert à partager ses recherches passées.

Je veux également remercier mon employeur, Ubisoft Montréal, qui m'a offert l'opportunité de poursuivre mes études tout en travaillant à temps partiel au début de ma maîtrise. Malgré le changement de direction du projet à la mi-parcours, je n'aurais pas pu compléter mes cours sans cet arrangement.

Finalement, je tiens à dire merci à ma famille pour leur support tout au long de ma maîtrise. Spécifiquement à ma mère, qui m'a toujours encouragée et poussée à continuer et à mon conjoint, qui m'a offert un environnement propice à la concentration en prenant en charge le plus de distractions possibles, vous avez mon éternelle gratitude.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ALGORITHMES

ACRONYMES

NPC Personnage non-joueur, ou *non-playable character* en anglais.

FPS Images par seconde, ou *frame per second* en anglais.

DFS Recherche en profondeur, ou *Depth-First Search* en anglais.

BFS Recherche en largeur, ou *Breadth-First Search* en anglais.

SPT Arbre des chemins les plus courts, ou *Shortest Path Tree* en anglais.

WA* *Weighted A**.

DWA* *Dynamic Weighted A**.

MHA* *Multi-Heuristic A**.

IMHA* *Independant Multi-Heuristic A**.

SMHA* *Shared Multi-Heuristic A**.

DMHA* *Dynamic Multi-Heuristic A**.

IDA* *Iterative Deepening A**.

IDS *Iterative-Deepening Search*.

D* A* Dynamique.

ADA* *Anytime Dynamic A**.

ARA* *Anytime Repairing A**.

SREA* *Static Rectangle Expansion A**.

HPA* *Hierarchical Path-Finding A**.

CH Contraction hiérarchique.

ED *Edge Difference*.

DN *Deleted Neighbor*.

OET *Original Edge Term*.

CoC *Cost of Contraction*.

SD Sommet Dynamique.

NVD Nombre de Voisins Dynamiques.

2D Deux Dimensions.

CSV *Comma Separated Values*.

RÉSUMÉ

Dans l'univers du jeu vidéo, les agents intelligents servent à enrichir le monde et l'expérience pour le joueur. Ils doivent être en mesure de se déplacer dans un monde réinterprété en tant que graphe de navigation. Il doit le faire de manière efficace et crédible tout en s'adaptant aux changements qui peuvent survenir. En planification de chemin dans les jeux vidéo, l'algorithme A* est souvent la solution utilisée pour répondre à ce besoin. Mais dans un contexte où le monde de jeu est constitué d'une grille 2D dont le facteur de branchement est très élevé, l'algorithme A* perd de son efficacité. Dans le cadre de ce mémoire, nous explorons différents algorithmes de planification de chemin ainsi que des techniques permettant de planifier un chemin plus efficacement dans un contexte dynamique, principalement la contraction hiérarchique. C'est une technique de prétraitement de graphe de navigation permettant d'avoir des raccourcis et d'accélérer la recherche de chemin. Le présent mémoire s'est basé sur cette dernière. Nous proposons une heuristique pour la contraction hiérarchique qui ajoute deux nouveaux critères dans le calcul de l'ordre de priorité des sommets, soit si le sommet est dynamique (SD) et le nombre de voisins directs du sommet courant étant dynamique (NVD). Nous comparons la recherche de chemin avec et sans la contraction hiérarchique en prétraitement sur le graphe. Nous constatons que le graphe hiérarchique est nettement plus efficace pour une carte en grille 2D. Nous évaluons la performance d'un graphe hiérarchique utilisant notre heuristique proposée face à un graphe hiérarchique utilisant l'heuristique de base. Ceci mène à la conclusion que, pour de grandes cartes, la contraction hiérarchique et la planification de chemin avec notre heuristique étaient moins efficaces que celle utilisant l'heuristique de base. Néanmoins, dans de plus petites cartes avec un plus petit nombre de sommets dynamiques, notre heuristique permet une contraction hiérarchique créant moins de raccourcis aidant à des planifications de chemin plus efficace. En réglant les limitations observées lors de ce mémoire, soit la sélection manuelle des sommets dynamiques et la grosseur des cartes de jeu causant une contraction hiérarchique trop longue, il serait possible d'avoir de meilleures performances avec la contraction hiérarchique qui utilise l'heuristique que nous avons proposée.

Mots-clés : Planification de chemin, Contraction Hiérarchique, A*, Jeux Vidéo, Heuristique, Dynamisme

INTRODUCTION

Dans l'univers du jeu vidéo, les agents virtuels sont des entités animées essentielles qui contribuent à l'immersion du joueur dans le monde virtuel (Rabin, 2007). Ces entités, communément appelées «personnages non-joueurs» ou «*non-playable characters*» (NPC) en anglais, peuvent être des fantômes dans *Pac-Man*, des créatures robotiques dans *Horizon : Zero Dawn*, ou encore des villageois dans *Assassin's Creed : Origins*. Chacun de ces êtres artificiellement intelligents contribue à créer l'illusion d'un monde autonome et réactif, capable d'interagir avec son environnement et les actions du joueur.

Dans un jeu vidéo, un agent peut avoir différents rôles en fonction de la situation. Il peut être un personnage impartial permettant une interaction avec le joueur, comme un commerçant avec lequel il est possible d'acheter, de vendre et d'échanger des articles, ou encore proposer des missions principales pour avancer dans l'histoire, ou des quêtes annexes, offrant ainsi des défis additionnels. Un NPC peut également être un personnage neutre n'ayant pas d'interaction avec le joueur. Ce genre d'agent sert principalement à ajouter de la vie dans le monde en faisant des tâches spécifiques, comme se réveiller le matin et se rendre à la rivière pour y puiser de l'eau à l'aide d'un sceau. Il peut aussi être un personnage qui interagit avec ou pour le joueur, que ce soit de manière positive ou négative. Les interactions positives sont par exemple un NPC étant l'allié du joueur, comme les membres d'un escadron tactique assistant le joueur et suivant ses ordres (e.g. : *Ghost Recon : Breakpoint*). Les interactions négatives sont représentées par les NPC ennemis du joueur, qui peuvent le poursuivre, l'attaquer ou défendre une zone précise. La plupart des rôles d'un NPC nécessitent des déplacements dans le monde.

Pour pouvoir se déplacer, les NPC ont besoin d'algorithmes de planification de chemin. Les principales approches de planification de chemin sont décrites dans la section 1.2. Ces algorithmes ont besoin d'avoir accès à une représentation du monde (voir le sous-chapitre 1.1). Ceci indique l'environnement navigable selon les capacités et la dimension de l'agent. Les mouvements possibles de l'agent dépendent donc de la géométrie du monde. Cela peut entraîner des problèmes de performances lorsque l'environnement est dynamique, puisqu'il doit attendre que les changements soient terminés pour effectuer une nouvelle planification de chemin.

Lorsqu'un jeu est en train de s'exécuter, beaucoup de différentes composantes doivent travailler en même temps afin d'offrir l'expérience de jeu la plus performante possible. Cela peut aller de la création de gra-

phiques visuels à l'interprétation des commandes du joueur pour faire bouger son personnage, en passant par l'adaptation de l'environnement selon des évènements déclenchés par le joueur ou le jeu. Ceci signifie que la recherche de chemin d'un ou plusieurs NPC ainsi que leur adaptation dans un monde dynamique ne peuvent pas monopoliser toutes les ressources de calcul et l'espace en mémoire, car cela affecterait d'autres aspects du jeu, comme le graphisme, l'expérience du joueur et la qualité du jeu. C'est pour cela que la partie gestion des NPC reçoit un budget limité de temps et de ressource afin d'effectuer son calcul, mais on s'attend tout de même à ce que le résultat soit rapide afin d'avoir un comportement fluide.

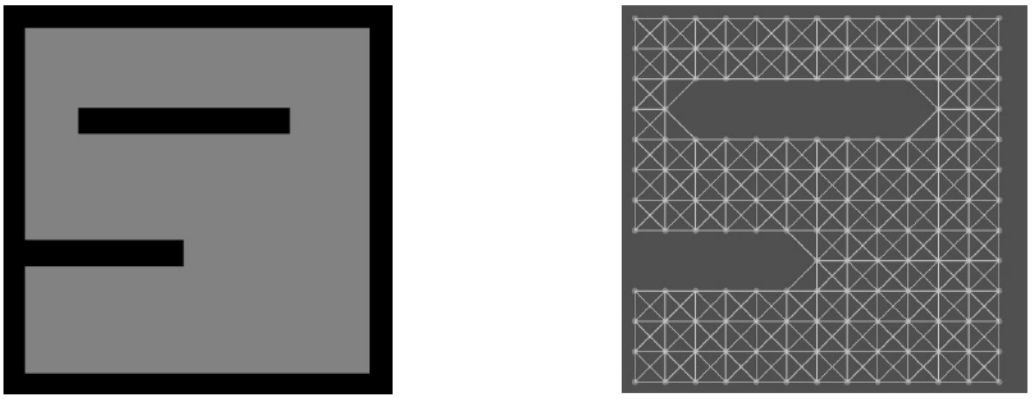
0.1 Problématique

La navigation dynamique regroupe deux axes, soit la mise à jour de la représentation du monde et le rajustement de la planification de chemin dans ce monde changeant si nécessaire. Le monde fait référence à la topologie dans un jeu comme montré à la figure 0.1, qui est le regroupement d'objets statiques et dynamiques que le joueur peut voir.

Figure 0.1 – Exemple de topologie d'un environnement de jeu. Vue de haut d'une plateforme avec deux murs parallèles.

Le joueur peut planifier son parcours en voyant l'espace libre à l'aide de la topologie, mais c'est la génération d'une représentation interne sur la topologie qui détermine les sections traversables du monde. De même que pour un NPC, il faut avoir accès à cette interprétation interne du monde afin qu'il puisse planifier son

chemin et, pour ce faire, il existe différentes méthodes possibles selon la situation. La figure 0.2a illustre un exemple de représentation du monde traversable. Afin qu'un NPC puisse parcourir l'environnement, il faut pouvoir traduire la représentation interne du monde. Ceci est généralement représenté sous forme d'un graphe orienté ou non orienté, appelé graphe de navigation (voir le sous-chapitre 1.1), où les sommets font référence aux cases et les arêtes les branchements entre elles. Un exemple peut être vu dans la figure 0.2b.



(a) (b)

Figure 0.2 – Représentation interne du monde traversable en 2D sous forme grille 2D. Les cases noires représentent les murs et les cases grises représentent l'espace traversable (a). La grille est ensuite traduite en graphe de navigation où chaque case est un sommet (b).

Un monde est dynamique à cause de facteurs changeants affectant la topologie du monde. Ceci signifie que l'espace traversable est mis à jour. Pour un joueur, ce n'est qu'une question d'un changement visuel et physique auprès duquel il doit ajuster ses actions, mais, en réalité, il faut d'abord appliquer les changements de la topologie dans les données internes du monde (voir la figure 0.3). La modification de la topologie de l'espace traversable affecte la représentation interne, ce qui à son tour peut affecter les sommets valides, les arêtes ainsi que leurs coûts associés comme vus à la figure 0.4.

Le processus actuel pour la planification de chemin d'un NPC lors d'un changement dans la topologie est qu'il recommence sa recherche, peu importe le changement dans le monde. Ceci est dû au fait qu'il n'y a aucun moyen dans le système courant de savoir si le changement affecte le chemin trouvé ou de conserver ce qui a été déjà planifié. Ce processus est exigeant et prend plus de temps, car il y a des situations où le NPC n'a pas besoin de repenser sa planification si le changement est mineur ou n'apporte aucun bénéfice. Cela signifie que le NPC refait sa planification pour avoir le même résultat qu'au départ, ce qui cause une perte de ressource qu'on pourrait transférer ailleurs.

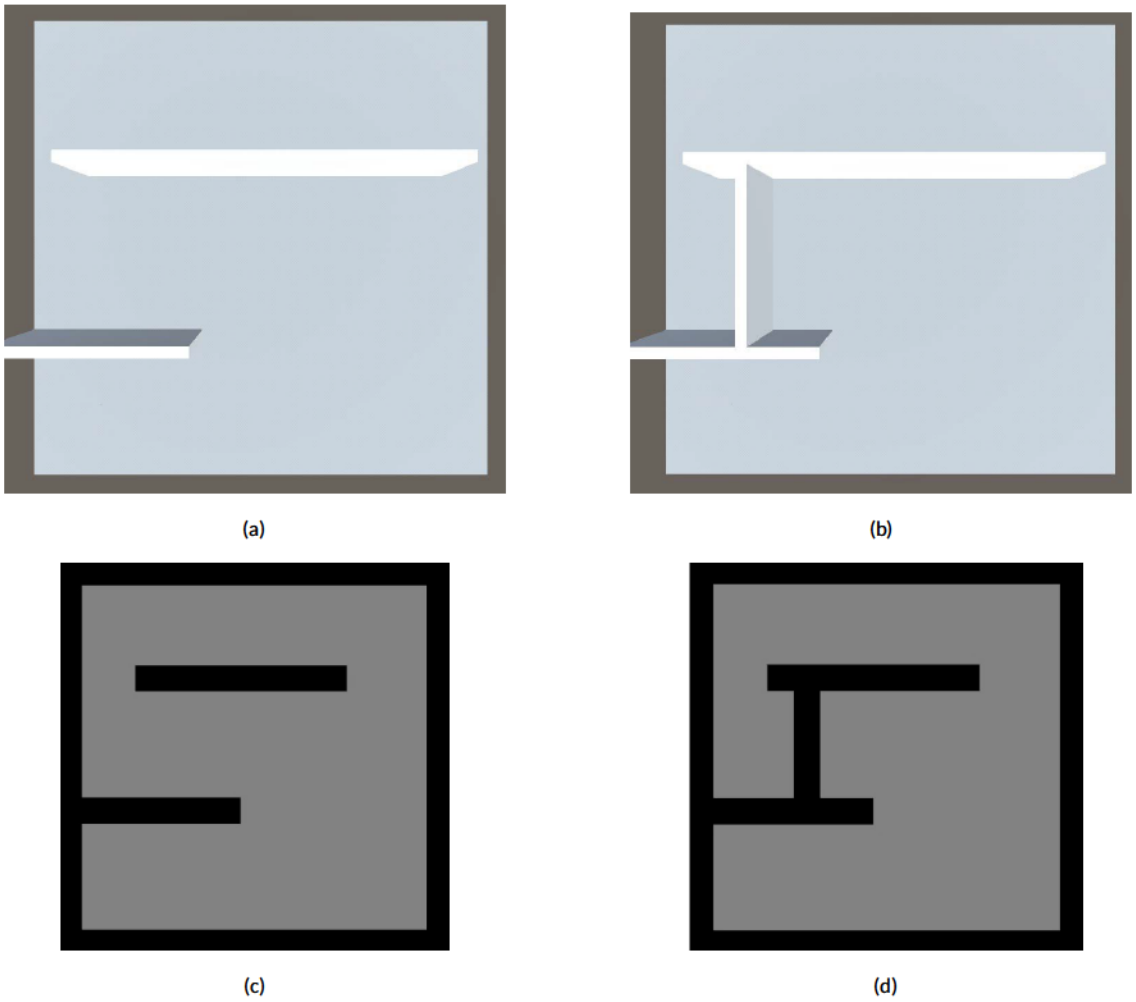


Figure 0.3 – Changement de topologie où un mur apparaît.

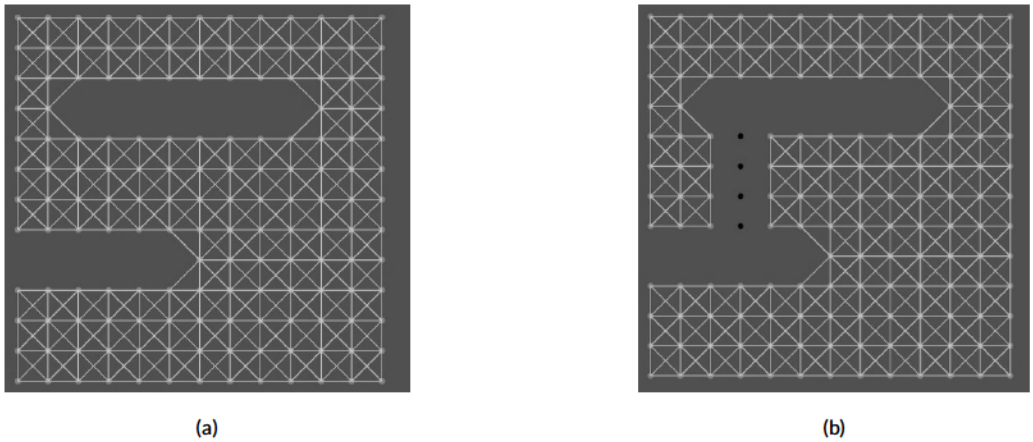


Figure 0.4 – À cause du mur ajouté le nombre de sommets et d'arêtes du graphe de navigation changent.

Pour un jeu avec une performance de 60 images par seconde, ou *frame per second* (FPS) en anglais, où une image est une image, chaque trame dure environ 16 millisecondes (ms). Le sous-système relié à l'intelligence artificielle dans un jeu a droit à typiquement 1,5 ms par image, dont 0,5 ms sont réservées à la planification de chemin des NPC. L'engin de jeu appartenant à Ubisoft donne à plusieurs systèmes des ressource pour l'exécution de leur calcul. Que ce soit à l'animation, effets visuels, son, génération de l'image, etc, généralement, le budget alloué à l'intelligence artificielle tourne autour de 1.5 ms par image. Cette information provient de source experte dans le domaine chez Ubisoft qui travaillent présentement sur des marques reconnues, telle que Rainbox Six Siege et Assassin's Creed. Ceci signifie qu'en projetant sur une seconde, le système a droit à 30 ms afin de planifier son chemin et que, si le monde change, il perd le temps utilisé au préalable et doit attendre la prochaine trame afin de recommencer sa recherche alors qu'il aurait pu continuer la même recherche dans certains cas.

Le processus désiré pour la replanification de chemin d'un NPC serait donc qu'il puisse garder en mémoire ce qui a déjà été planifié et déterminer s'il a besoin de modifier sa trajectoire ou non, ce qui a pour avantage de ne pas recalculer un chemin déjà trouvé. Tout ceci avec un espace en mémoire restreint et un temps d'exécution limité. Le but vise donc à pouvoir replanifier un chemin plus rapidement que ce qui est actuellement en place, soit de recommencer la planification de recherche de chemin à chaque changement du monde.

0.2 Objectif de la recherche

L'objectif de ce mémoire est d'apporter une amélioration à une technique de recherche de chemin permettant de s'adapter à un monde fortement dynamique dans un contexte de carte de jeu en grille. Pour ce faire, il faut analyser les techniques existantes de recherche de chemin et les optimisations courantes. Ensuite, il faut analyser les impacts dans un contexte de monde fortement dynamique afin d'identifier leurs forces et leurs limitations pour finalement offrir une amélioration sur une technique existante.

0.3 Méthodologie de la recherche

Afin d'analyser et d'identifier une technique de recherche de chemin prometteuse, il faut commencer par une revue de la littérature courante sur la recherche de chemin et les adaptations connues dans un contexte de monde dynamique. Cette étape sert à couvrir aussi la comparaison et l'évaluation des algorithmes trouvés. Après avoir sélectionné la technique à améliorer, il faut donc planifier et implémenter un projet afin de

comparer les performances. Ce projet permet de charger un monde de jeu en grille, d'y faire des modifications dynamique de l'espace navigable et d'appliquer des algorithmes de recherche de chemin. Il permet aussi de collecter les données et de déterminer si l'amélioration proposée est une solution qui répond au problème présenté.

0.4 Structure du mémoire

Le chapitre 1 fait une vue d'ensemble des différentes techniques de planification de chemin existante dans la littérature. Le chapitre 2 présente une proposition se basant sur une technique existante afin d'offrir une solution à la problématique. Le chapitre 3 est une évaluation de la proposition et une analyse des résultats en comparaison avec ce qui est en ce moment utilisé dans le domaine du jeu.

0.5 Conclusion

Le but de ma recherche est d'offrir une nouvelle technique permettant d'avoir une recherche de chemin dans un monde dynamique efficace utilisant le moins de ressources possibles dans un jeu. Cela permet de rediriger les ressources non utilisées vers d'autres parties du jeu afin d'y améliorer sa qualité générale. À l'aide des comparaisons et expériences, il est aussi possible de pousser la recherche vers une nouvelle direction pour trouver d'autres axes de solutions.

CHAPITRE 1
ÉTAT DE L'ART

Le chapitre suivant commence par une explication de ce que représente le monde de navigation pour la planification de chemin. Par la suite, une vue d'ensemble des algorithmes de planification de chemin existant ainsi que les modifications apportées à ceux-ci dans le but de répondre à différents besoins de performance, soit en temps de calcul ou en espace mémoire.

1.1 Monde de navigation

Il existe plusieurs différentes manières de représenter l'espace navigable dans un monde de jeu. Il est possible de représenter la topologie à l'aide de polygones en deux dimensions, des grilles uniformes et des voxels, soit des pixels volumétriques, qui peuvent être utilisés autant en deux qu'en trois dimensions. Ces représentations du monde sont ensuite utilisées afin de créer le graphe de navigation pour les NPC.

Dans un contexte de jeu vidéo, la navigation fait référence à un graphe de navigation G, appelé *NavGraph* (Buckland, 2004), représentant le monde traversable ainsi que l'algorithme de planification de chemin appliqué sur celui-ci. Le graphe de navigation est habituellement noté $G = (V, E)$, où V désigne les sommets, qui correspondent à des emplacements dans le monde, et E représente les arêtes, qui sont les tronçons indiquant la capacité de se rendre d'un sommet à un autre en associant un coût $c_{v_i,v_{i+1}}$ au déplacement. Comme illustré à la figure 1.1, G peut être un graphe non orienté 1.1a où les arêtes sont bidirectionnels ou un graphe orienté 1.1b où des arêtes peuvent être à sens unique.

Pour un sommet d'origine $v_0 \in V$ et un sommet de destination $v_k \in V$, un chemin est une séquence de sommets $\langle v_0, v_1, ..., v_k \rangle$ où chaque v_{i+1} est le sommet suivant v_i dans la séquence, v_0 est le sommet d'origine et v_k est le sommet de destination. Le chemin a un coût C qui est calculé en additionnant le coût individuel de chaque arête utilisée, $c_{v_i,v_{i+1}}$, et est considéré comme optimal si la séquence trouvée a le plus petit coût possible entre le sommet d'origine et le sommet de destination dans le graphe de navigation G.

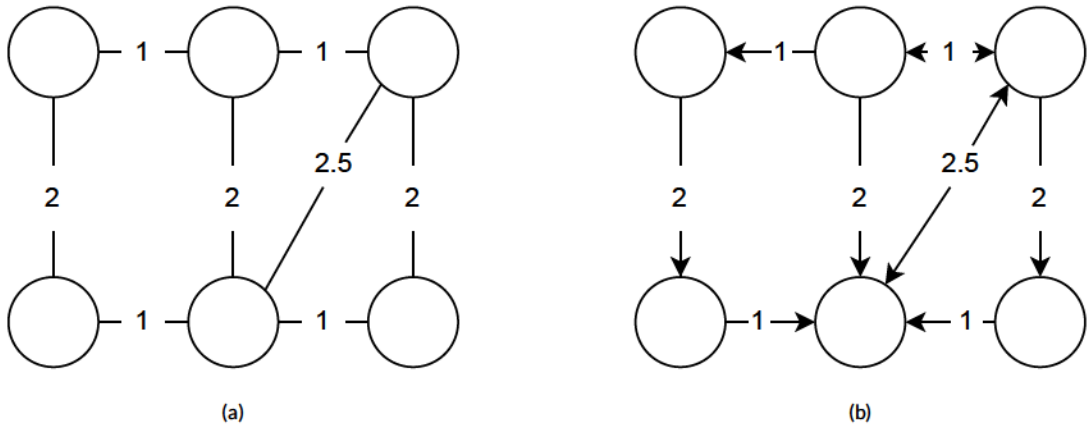


Figure 1.1 – Exemple d'un graphe non orienté (a) et d'un graphe orienté (b)

1.2 Algorithmes de planification de chemin

1.2.1 Recherche en largeur

L'algorithme de recherche en largeur (Russell et Norvig, 2009), ou *Breadth-First Search* (BFS) en anglais, est une technique parcourant tous les sommets adjacents avant d'aller plus loin dans le graphe. BFS est pratique lorsque le poids des arêtes sont uniformes et dans les cas où le sommet d'origine et le sommet de destination sont à proximité l'un de l'autre, mais est contraignant sur une longue distance puisqu'il va dans toutes les directions du graphe de manière égale. Tout comme DFS, sa complexité est de $O(n + m)$, ce qui est l'équivalent de traverser tous les sommets et toutes les arêtes.

1.2.2 Algorithme de Dijkstra

L'algorithme de Dijkstra (Dijkstra, 1959) porte le nom de son inventeur le professeur Edsger Wybe Dijkstra et est un algorithme ayant pour but de trouver le chemin le plus court entre un sommet d'origine et un sommet de destination dans un graphe avec coûts. Il est aussi possible de trouver les chemins les plus courts à partir d'une origine en créant un arbre des chemins les plus courts, *shortest path tree* (SPT) (Ghallab *et al.*, 2016), dont le premier élément est le sommet origine. SPT permet d'avoir le chemin le plus court de l'origine vers n'importe quel sommet, donc si on parcourt les arêtes dans le sens inverse, on peut avoir les chemins les plus courts entre diverses origines vers une destination. Si un point de destination est fourni, l'algorithme crée le SPT jusqu'à atteindre la destination. En considérant les sommets ($n = |V|$) et les arêtes à visiter ($m = |E|$), le temps de calcul de cet algorithme gourmand peut atteindre une complexité de $O(n \log(n) + m \log(n))$ dans le pire des cas.

1.2.3 Floyd-Warshall

L'algorithme Floyd-Warshall (Floyd, 1962)(Cormen *et al.*, 2009) est une technique de recherche de chemin qui calcule le chemin le plus court entre chaque paire de sommets dans un graphe orienté avec coûts. Il peut donc connaitre les chemins les plus courts entre multiple origines vers multiple destinations. Il garde en mémoire une matrice de coûts du chemin le plus court entre chaque sommet et une matrice de direction selon le chemin le plus court. Dans le cas d'un graphe orienté statique, c'est un bon moyen d'avoir accès rapidement aux chemins les plus courts pour chaque combinaison de sommet d'origine et de sommet de destination, mais dans un contexte dynamique cela devient problématique, car il faut recalculer la matrice de coûts de chemin et devient coûteux dans un grand graphe. De plus, comme seuls les sommets ($n = |V|$) impactent ses complexités, son temps de calcul est de complexité $O(n^3)$ et son espace en mémoire est de complexité $O(n^2)$. Le temps de calcul dans un monde de jeu dynamique devient rapidement problématique dans le cas où il faut modifier les deux matrices en temps réel. Pour ce qui est de l'espace en mémoire, puisqu'en général les graphes de navigation sont de grandes tailles et l'espace mémoire est limité, cet algorithme n'est pas souvent considéré. En revanche, si une section du monde est petite et statique Floyd-Warshall pourrait être une option à considérer.

1.2.4 Bellman-Ford

L'algorithme Bellman-Ford (Bellman, 1958)(Cormen *et al.*, 2009) est semblable à l'algorithme de Dijkstra vu à la section 1.2.2 en plus de permettre des arêtes à coûts négatifs dans le graphe de navigation. Il analyse G à partir du sommet d'origine et indique s'il y a un cycle négatif accessible à partir de celui-ci, soit qu'il n'y a pas de solution à la recherche de chemin. Pour les sommets $n = |V|$ et les arêtes $m = |E|$ du graphe, son temps de calcule est de complexité $O(nm)$ en moyenne et $O(n^3)$ dans le pire cas. Dans le cas de le présente problématique, les coûts des arêtes sont toujours positifs, donc Bellman-Ford ne sera pas considéré.

1.3 Algorithmes de planification de chemin avec optimisation du temps de recherche

1.3.1 Recherche Bidirectionnel

La planification de chemin bidirectionnel (Russell et Norvig, 2009) est une technique de planification de chemin connu pour permettre d'atteindre une solution plus rapidement en faisant la recherche de chemin dans les deux directions, soit de la source à la destination et de la destination à la source, en simultané (voir la figure 1.2). Ce n'est pas un nouvel algorithme, mais plutôt une façon d'évaluer un algorithme déjà

existant avec cette nouvelle perspective. Par exemple, il est possible d'appliquer l'algorithme de planification de chemin utilisant l'algorithme de Dijkstra 1.2.2 dans les deux directions afin d'accélérer la recherche.

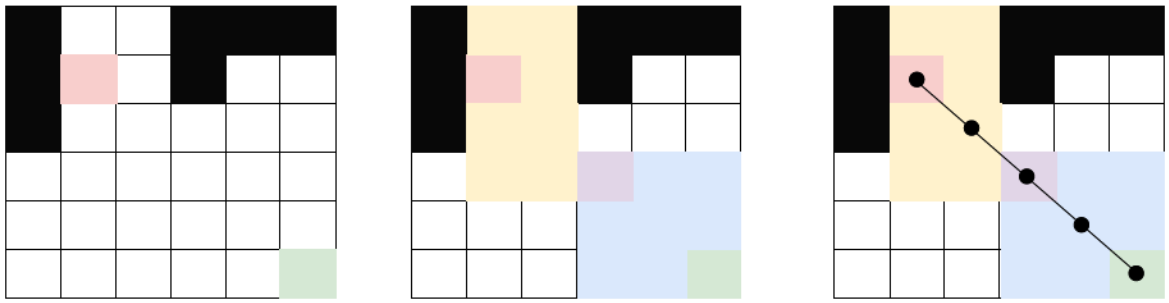


Figure 1.2 – Carte en grille 2D où la case verte est la source et la case rouge est la destination. La recherche bidirectionnel planifie un chemin en cherchant de la source et de la destination (b), où les cases bleus sont les sommets explorés à partir de la source et les cases jaunes sont les sommets explorés à partir de la destination. La planification arête lorsqu'une intersection est atteinte (case mauve), correspondant à un sommet étant exploré par les deux directions. Le chemin de la source à l'intersection et de la destination à l'intersection forment le chemin complet de la source à la destination (c).

Dans le domaine de la planification de chemin, c'est une manière de savoir si un chemin est possible entre deux points en visitant moins de sommets. Si la destination est confinée dans une section fermée et que la source est dans un monde ouvert, la recherche de chemin partant de la destination se termine sans succès plus tôt que celle partant de la source. Ceci permet d'arrêter la recherche de chemin plus vite que si on cherche seulement dans une direction. Dans le cas où un chemin serait possible, la recherche se termine lorsque les deux directions se connectent à un sommet commun.

1.3.2 A*

L'algorithme A* (Hart *et al.*, 1968)(Barr, 1981)(Ghallab *et al.*, 2016) est une technique améliorant l'algorithme de Dijkstra pouvant trouver le chemin optimal dans un graphe sans avoir à visiter autant de sommets que ce dernier. En revanche, l'algorithme A* ne peut effectuer la recherche que pour un sommet de destination déterminé alors que Dijkstra peut en avoir plusieurs. Ceci vient du fait que, pour choisir le prochain sommet à visiter, l'algorithme A* utilise une fonction d'évaluation $f(n)$ où $n \in V$, tel que démontré par notre réécriture formelle de l'algorithme A* 1. Le prochain sommet à visiter choisi est celui qui a le plus petit $f(n)$ parmi les sommets candidats. $f(n)$ est la somme de deux parties :

$$f(n) = g(n) + h(n) \tag{1.1}$$

où $g(n)$ est le coût réel du chemin optimal de v_0 à n et $h(n)$ est l'heuristique du chemin optimal de n à v_k. Une heuristique est une estimation du coût restant pour atteindre le but selon les paramètres du monde et est considéré *admissible* si h n'est jamais plus élevé que le coût optimal. L'estimation de l'heuristique dépend du paramètre du coût utilisé dans l'algorithme, par exemple, la distance euclidienne.

L'algorithme A* va toujours terminer puis retourner une solution si elle existe et la solution est optimale si l'heuristique est admissible. Par contre, il peut avoir besoin de beaucoup d'espace mémoire pour garder trace des sommets visités et l'algorithme A* visite tous les sommets du graphe si la destination n'est pas accessible.

L'algorithme A* est utilisé dans les jeux et est plutôt efficace pour répondre à leurs besoins. La planification de chemin pour un agent est principalement dans de courtes distances, ce qui minimise l'espace en mémoire utilisé, et des mesures supplémentaires sont mises en place afin d'éviter d'effectuer une planification vers une position inaccessible. En revanche, lors de changements dynamiques du monde l'agent doit recalculer son chemin au complet afin de connaitre les nouveaux coûts et heuristiques et n'a pas la possibilité de garder des informations d'une ancienne recherche afin d'accélérer le processus.

1.3.3 Weighted A*

Comme l'algorithme A* comporte des risques d'avoir un temps d'exécution et une utilisation d'espace mémoire exponentiel par rapport à la longueur du chemin à rechercher, il existe plusieurs extensions de l'algorithme A* qui se concentrent sur la mémoire et d'autres qui se penchent sur le temps d'exécution. Les techniques Weighted A*, mettent l'accent sur réduire le temps d'exécution en permettant un résultat sous-optimal selon un certain degré (Ebendt et Drechsler, 2009).

Algorithme 1 A*

Pour un graphe G avec un ensemble de sommets V, un sommet d'origine source $v_0 \in V$ et un sommet de destination $v_k \in V$
$OUVERT$ est une liste de sommets
$FERMER$ est une liste de sommets
$OUVERT \leftarrow VIDE$
$FERMER \leftarrow VIDE$
ajout v_0 à $OUVERT$
Tant que $OUVERT$ n'est pas vide **faire**
    $v \leftarrow$ sommet dans $OUVERT$ avec le plus petit $f(n)$
    enlève v de $OUVERT$
    si $v = v_k$ **alors**
        fin de la recherche et retour du chemin trouvé
    Fin si
    ajout v dans $FERMER$
    Pour chaque $voisin$ de v **faire**
        si $voisin \in FERMER$ **alors**
            retour en haut de la boucle
        sinon si $voisin \in OUVERT$ **alors**
            calcule nouveau $g(voisin)$
            si nouveau $g(voisin) <$ ancien $g(voisin)$ **alors**
                assigne v en tant que sommet parent de $voisin$
                nouveau $g(voisin)$ remplace ancien $g(voisin)$
                calcule nouveau $f(voisin)$
            Fin si
        sinon si $voisin \notin OUVERT$ **alors**
            assigne v en tant que sommet parent de $voisin$
            calcule nouveau $f(voisin)$ et $g(voisin)$
            $OUVERT \leftarrow voisin$
        Fin si
    Fin pour
Fin Tant que

1.3.3.1 Inflation Constante

Cette technique venant de Pohl (Pohl, 1970) applique un multiplicateur constant sur l'heuristique de la fonction d'évaluation $f(n)$ lors de la sélection du sommet à visiter, soit $1 + \epsilon$ où $\epsilon > 0$. $f(n)$ devient donc l'équation (1.2) dans l'algorithme usuel du A*.

$$f^{WA^*}(n) = g(n) + (1 + \epsilon)h(n) \tag{1.2}$$

Le premier variant Weighted A* proposé est communément nommé WA*. WA* ne garanti pas la préservation d'un h admissible après l'inflation de celui-ci, mais ceci permet d'avoir une version relaxée du A* pouvant diriger rapidement la recherche vers une direction plus prometteur en mettant plus d'importance sur les sommets ayant un petit h. Même si WA* n'a pas de h admissible, il est possible d'avoir un h ϵ-admissible, ce qui signifie que la solution trouvée est au plus ϵ-optimale. En revanche, il n'y a aucune garantie contre une fin de recherche précoce.

1.3.3.2 Dynamic Weighting

Cette variation de Pohl (Pohl, 1973) est une extension du WA*, connu sous le nom de Dynamic Weighting (DWA*), qui permet d'affecter le multiplicateur ϵ appliquer sur l'heuristique lors du calcul de $f(n)$. Il commence par appliquer un multiplicateur élevé au début de la recherche pour aider à trouver une direction prometteuse et le diminue graduellement au fur et à mesure. Ceci permet de ne pas avoir une fin de recherche précoce. La fonction d'évaluation est représentée par l'équation (1.3), où $d(n)$ représente la profondeur du sommet dans la recherche et M représente la longueur optimale de la solution. Comme M est rarement connu au préalable, une approximation ou un seuil maximal peut être utilisé. Tout comme WA*, DWA* est ϵ-admissible si h est admissible.

$$f^{DWA^*}(n) = g(n) + (1 + \epsilon)\frac{1 - d(n)}{M}h(n) \tag{1.3}$$

Les techniques Weighted A* permettent d'atteindre la destination plus rapidement que l'algorithme A* classique avec possiblement moins d'espace mémoire utilisé en échange d'un chemin sous-optimal. En revanche, elles dépendent de la précision des heuristiques. Si h a un risque d'avoir un minimum strictement

local, la performance est grandement affectée.

WA* et DWA* sont similaires au A* et offre une manière d'accélérer la recherche vers la destination donc l'intégration dans un système utilisant le A* est rapide et facile. Tout dépendant de la sous-optimalité du chemin trouvé le résultat peut être assez acceptable dans un contexte de jeu.

1.3.4 IDA*

Le *Iterative-Deepening A** (Koenig *et al.*, 2004) (IDA*) est une variation de la technique de planification de chemin récursive *Iterative-Deepening Search* (IDS).

Le IDS effectue une recherche en profondeur, mais au lieu d'aller au bout le l'arbre comme le *Depth-First Search* (DFS) il applique une limite de profondeur à chaque itération et retourne au point de départ. La limite de profondeur augmente graduellement jusqu'à trouver une solution.

Le IDA* fonctionne comme le IDS à l'exception du fait que la limite de profondeur est maintenant une limite de coût qui incrémente à chaque itération. Le IDA* n'est pas mieux que l'algorithme A* sur une longue distance, mais à la possibilité d'être plus performant que celui-ci sur de courte distance dans un graphe dense. Néanmoins, dans le contexte présent, il ne serait pas judicieux d'explorer davantage cet algorithme, car nous voulons une technique de recherche de chemin avec de bonnes performances pour des courtes et des longues distances.

1.3.5 Dynamic Multi-Heuristic A*

Dû au fait que les techniques *Weighted A** sont à risque de perdre de la performance à cause d'heuristique sujet à de local minima, le *Multi-Heuristic A** (MHA*) a été pensé pour contrer cela. Il utilise des heuristiques possiblement inadmissibles pour diriger la recherche hors de minimum locaux (Islam *et al.*, 2015).

MHA* prend en paramètre une heuristique consistante h_0, un ensemble arbitraire d'heuristiques inadmissibles $\{h_1, h_2, ..., h_k\}$ et deux facteurs de poids w_1 et w_2. Il lance plusieurs planifications de chemin WA* avec w_1 et les heuristiques inadmissibles en séquence round-robin sans ordre particulier tout en effectuant une recherche à part utilisant l'heuristique constante. Cette planification de chemin, considéré comme l'ancre, est le critère pour contrôler quelle heuristique inadmissible est utilisée en ne permettant que celles qui ont

un $f(n)$ au plus w_2 fois le plus petit $f(n)$ présent dans la liste ouverte de l'ancre.

Il existe deux variations du MHA*, le *Independent Multi-Heuristic A** (Islam *et al.*, 2015) (IMHA*) qui utilise des valeurs $g(n)$ et $h(n)$ indépendantes à chaque recherche et le *Shared Multi-Heuristic A** (Islam *et al.*, 2015) (SMHA*) qui utilise des valeurs $h(n)$ indépendantes, mais partage un même $g(n)$ à chaque recherche ce qui lui permet de faire une combinaison de plusieurs chemins partiels afin de contrer les minimum locaux.

Le *Dynamic Multi-Heuristic A** (DMHA*) (Islam *et al.*, 2015) est une variation du SMHA* qui s'active seulement lorsque que minimum local est atteint pendant la recherche en SMHA*. Lors d'une des recherches, après l'expansion du sommet courant en évaluant tous ses sommets voisins, si l'heuristique du sommet courant n'est pas plus petit que le plus petit heuristique trouvé pour au moins un sommet visité au préalable, ceci indique que la recherche se dirige vers un minimum local. Le DMHA* tente de régler ce problème en générant un sommet d'attraction proche du minimum local en choisissant celui qui a au moins une heuristique plus petite que la plus petite heuristique observée dans la recherche courante.

Le DMHA* est une technique utilisée dans le cadre du déplacement d'un robot dans des espaces étroits nécessitant un changement dans la position de ses membres afin de passer. Pour le contexte d'un jeu, il ne semble pas y avoir un besoin pour une solution aussi spécifique.

1.4 Algorithmes de planification de chemin dans un monde dynamique

1.4.1 D*

Le A* dynamique (D*) (Stentz, 1993) est semblable au A* en plus de pouvoir planifier un chemin optimal avec une connaissance partielle ou nulle du monde. Dans un graphe de navigation orienté, D* est en mesure de planifier un chemin où les coûts des arêtes peuvent changer dynamiquement. En plus des paramètres présents dans l'algorithme A*, soit une liste ouverte, une liste fermée, une heuristique par sommet, un coût par arête et une fonction d'évaluation, il y a aussi de nouveaux paramètres pour gérer l'aspect dynamique du D* : une étiquette sur chaque sommet $t(n)$ qui détermine son état entre *NEW* s'il n'a jamais été dans la liste ouverte, *OPEN* s'il est actuellement dans la liste ouverte et *CLOSED* s'il est dans la liste fermée, et l'heuristique précédente sur chaque sommet $p(n)$ qui est équivalent à l'heuristique de départ lors de l'insertion du sommet dans la liste ouverte et est utilisé afin de propager une augmentation du coût du chemin si $p(n) < h(n)$, soit *RAISE*, ou une diminution du coût du chemin si $p(n) \geq h(n)$, soit *LOWER*. Cette

propagation est appliquée à chaque retrait de sommet de la liste ouverte, en passant l'information aux voisins directs du sommet et en les ajoutant dans la liste ouverte. Tout comme le A*, le D* retourne une séquence entre le sommet d'origine et le sommet de destination si un chemin existe, mais fait sa planification au sens inverse, soit que l'origine est le sommet de destination et que la destination est le sommet d'origine.

L'algorithme est réparti en deux fonctions principales : *PROCESS-STATE*, qui sert à planifier le chemin optimal comme le A* avec quelques modifications, et *MODIFY-COST*, qui propage les changements de coût sur les arêtes affectées et ajoute les sommets impactés dans la liste ouverte. À l'initialisation, le sommet de destination est inséré dans la liste ouverte avec une heuristique nulle et son étiquette à $OPEN$.

Dans *PROCESS-STATE*, le sommet avec la plus petite valeur heuristique entre $p(n)$ et $h(n)$ est retiré de la liste ouverte. Ce sommet évalue ensuite ses voisins présents dans la liste fermée afin de savoir si l'un d'eux peut diminuer son heuristique, sachant que l'heuristique d'un sommet est $h(n) = h(n+1) + c(n, n+1)$ où n est le présent sommet et $n+1$ est son sommet voisin dans la liste fermée. Il évalue ensuite tous ses voisins et propage le changement d'heuristique à ceux qui sont déjà présents dans la liste ouverte puisqu'ils utilisent ce sommet comme prédécesseur. Si des sommets voisins n'utilisent pas le sommet courant comme prédécesseur, leur heuristique est affectée et le prédécesseur est changé seulement si le sommet courant est considéré *LOWER*. Si le sommet est un *RAISE*, il est réinséré dans la liste ouverte. Pour les sommets voisins présents dans la liste fermée, ils sont réinsérés dans la liste ouverte s'ils n'ont pas le présent sommet comme prédécesseur et qu'ils permettent d'avoir une plus petite heuristique pour le sommet en tant que prédécesseur de celui-ci. Pour *MODIFY-COST* propage le changement de coût sur le sommet et réinsère celui-ci dans la liste ouverte s'il était dans la liste fermée.

L'avantage de cet algorithme est qu'il est possible d'effectuer la planification de chemin en parallèle avec la modification locale du monde. C'est une option supplémentaire comparée au A* qui peut seulement planifier dans un monde statique ou après la modification totale de celui-ci. De plus, puisque la planification se fait à l'inverse, il est possible d'avoir plusieurs sommets de destination pour un sommet source, comme l'algorithme de Dijkstra (voir le sous-chapitre 1.2.2).

1.4.2 *Focussed* D*

Le *Focussed* D* (Stentz, 1995) est une extension de l'algorithme D* qui concentre sa mise à jour du coût des sommets selon la position courante de l'agent au moment de la détection d'un changement dans le monde. La propagation de la mise à jour des sommets s'arrête lorsque la plus petite valeur dans la liste ouverte est plus grande ou égale au coût du chemin trouver précédemment. Ceci permet d'arrêter la propagation plus tôt que le D* classique.

Comme la propagation se base sur la position courant de l'agent et non du monde en général, la modification des coûts est elle aussi spécifique à la position de l'agent. Ceci signifie que, si l'agent détecte un autre changement dans un sommet autour de lui, il ne sera pas possible de se baser sur l'information courante dans la liste ouverte, car celle-ci correspond à une propagation locale précédente. *Focussed* D* assume que l'agent ne fait que quelques mouvements en direction du sommet dont le changement a été détecté avant de devoir refaire sa planification de chemin, ce qui veut dire que la valeur de la fonction de coût des sommets dans la liste ouverte correspondent à leur fonction de coût courante moins l'estimation de la distance entre l'ancienne position de détection et la courante, et une valeur ϵ arbitraire.

Cet algorithme est préférable à D* parce qu'il permet d'avoir les mêmes résultats de chemins calculés, mais est plus rapide lors de la modification des données lorsqu'un changement dans le monde est détecté en temps réel.

1.4.3 D* *Lite*

D* *Lite* (Koenig et Likhachev, 2002) se base sur le *Lifelong Planning A** (Koenig *et al.*, 2004), LPA*, et est au moins aussi efficace que le *Focussed* D*. LPA* est une version du A* avec incrémentation et s'applique sur des graphes finis et connus où le coût des connexions entre sommets varie. Le LPA* garde en mémoire les heuristiques de chaque connexion, comme l'algorithme A*. Cependant, s'il y a un changement de coût dans l'environnement, LPA* modifie localement les coûts de seulement les sommets qui sont jugés pertinent pour la recherche du chemin le plus court en fonction des heuristiques. Il a donc une liste de priorité. Au lieu de calculer le coût du chemin de la source à la destination, comme le LPA*, D* *Lite* calcule de la destination à la source comme le D*. Ceci lui donne la possibilité de changer de source pendant la planification de chemin.

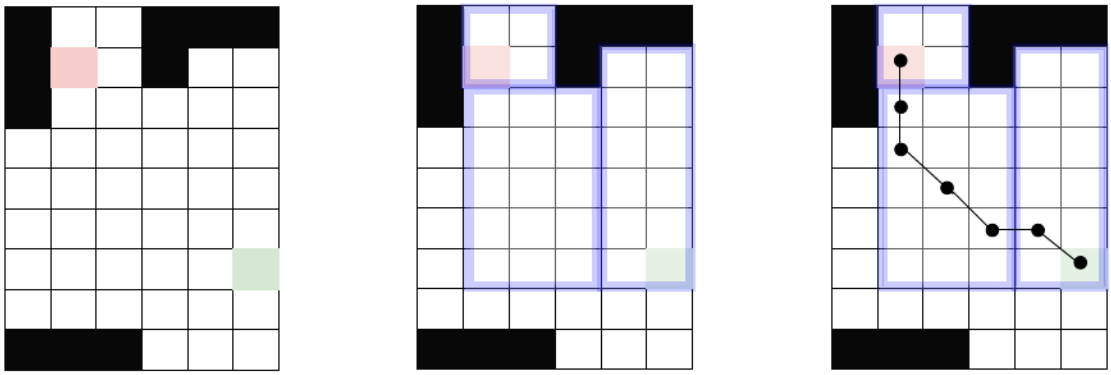
1.4.4 Anytime Dynamic A*

Le *Anytime Dynamic A** (Likhachev *et al.*, 2005) (ADA*) est un algorithme de planification de chemin dans un contexte dynamique qui est un mélange de D* *Lite* en faisant une recherche de la source à la destination et de *Anytime Repairing A** (Likhachev *et al.*, 2003) (ARA*) pour son comportement lors d'un changement dynamique dans l'environnement. Le ARA* est un algorithme de planification de chemin incrémental qui priorise trouver un chemin le plus tôt possible, même si le chemin n'est pas optimal. En effet, il commence par une planification de chemin en WA* (voir le sous-chapitre 1.3.3) avec un poids très élevé. Il répète ensuite la même requête de planification de chemin avec un poids plus petit à chaque nouvelle requête en réutilisant le résultat précédent pour diriger la recherche. En cas de changement dans le monde, les sommets et arêtes affectés sont gardés en mémoire pour la prochaine itération. Le ADA* fait donc une planification de la destination à la source et en cas de changements dans le monde, il utilise un poids élevé afin de trouver un chemin rapidement et recommence de manière incrémentale avec un plus petit poids tout en réutilisant les résultats de planification trouvés au préalable. Cet algorithme a un facteur de branchement moindre que le D* Lite et le ARA*. Par contre, s'il y a trop de changement dynamique dans le monde ou si la destination change, le ADA* est moins performant que le A*.

1.4.5 Static Rectangle Expansion A*

Le *Static Rectangle Expansion A**(Chong *et al.*, 2022) (SREA*) est une technique de planification incrémentale sur des cartes en grille 2D qui a pour but de limiter le facteur de branchement d'un algorithme A* classique et a été prouvée plus rapide que celui-ci après la succession de plusieurs requêtes de recherche de chemin. Comme démontré à la figure 1.3, SREA* crée des bordures rectangulaires de la source à la destination selon la configuration de l'environnement, puis limite sa planification de chemin aux rectangles créés.

L'avantage de cette technique est qu'elle garde les rectangles créés en mémoire durant la durée de vie du processus afin qu'ils puissent être réutilisés pour les autres requêtes de planification de chemin. Ceci signifie qu'il économise du temps de calcul au fil des requêtes. En cas de changement dynamique dans la carte, SREA* modifie les rectangles affectés en les sectionnant en plus petits rectangles ou bien en les combinant avec un rectangle voisin.



(a) (b) (c)

Figure 1.3 – Carte en grille 2D où la case verte est la source et la case rouge est la destination. SREA* créé des rectangles (en bleu) de la source à la destination délimités par les obstacles dans l'environnement (b). La planification de chemin se fait en cherchant un chemin de bordure en bordure jusqu'à la destination (c).

1.5 Approches hiérarchiques

Les planifications de chemin dans un environnement hiérarchique se concentrent sur la modification du graphe de navigation. Ce sont en effet des techniques applicables au moment de la génération d'un graphe avant l'application des processus de planification de chemin.

1.5.1 Near Optimal Hierarchical Path-Finding

Le Hierarchical Path-Finding A* (Botea *et al.*, 2004), HPA*, est une technique hiérarchique qui crée en prétraitement une abstraction d'un monde à l'aide de sous-groupes dans le but d'avoir une recherche de chemin plus rapide. HPA* a 3 principales étapes :

— La création d'une abstraction d'une carte quadrillée en un graphe de navigation abstrait.
— L'ajout de la source et de la destination dans l'abstraction.
— La recherche d'un chemin qui sera abstrait.

L'abstraction d'une carte quadrillée se fait à l'étape de prétraitement et est gardée en mémoire. Pour une carte, on commence par séparer la grille en sous-groupes rectangulaires de taille égaux. Par la suite, pour chaque bordure entre sous-groupes, on détermine une liste d'entrées de connexions qui seront les sommets du graphe de navigation. Une entrée doit être strictement entre deux bordures symétriques sans obstacle et ne doit pas excéder une certaine longueur, déterminée arbitrairement. Dans le cas où la longueur de l'entrée est plus élevée que la valeur indiquée, on se permet de mettre deux sommets d'entrées à chaque extrémité de la bordure. Lorsque tous les sommets ont été trouvés, les arêtes sont déterminées en ajoutant des connexions entre les sommets. Les sommets entre bordures ont des arêtes inter-groupes, et chaque sous-groupe a un ensemble d'arêtes intragroupe s'il y a un chemin possible entre chaque pair de sommets dans le sous-groupe. Le coût des arêtes peut être partiel à l'utilisation, mais, dans le cas de l'article, les arêtes inter-groupes ont un coût de 1 et les arêtes intragroupes ont un coût représentant la longueur du chemin optimal entre les sommets. Ceci résulte en un graphe de navigation abstrait de premier niveau. Il est possible d'ajouter une abstraction en-dessus le graphe afin d'avoir plusieurs niveaux d'abstraction.

À chaque requête de recherche de chemin, il faut tout d'abord connecter la source et la destination dans le graphe de navigation abstrait. Ceci est fait en créant une arête vers le sommet le plus proche dans le graphe. Lorsque les points sont connectés, une recherche entre la source et la destination est appliquée sur

le graphe en utilisant l'algorithme A*(Hart *et al.*, 1968)(Barr, 1981)(Ghallab *et al.*, 2016). Le chemin retourné peut être vu en trois parties : le chemin détaillé entre la source et le sommet le plus proche dans le graphe de navigation, le chemin abstrait entre le sommet le plus proche de la source et le sommet le plus proche de la destination dans le graphe et le chemin détaillé entre le sommet le plus proche de la destination et la destination.

Dans un cas de changement dynamique dans la carte, on a besoin de modifier le graphe de navigation seulement si les bordures sont affectées. La technique est un domaine agnostique et permet de trouver rapidement des chemins semi-optimaux. Il est possible d'avoir plusieurs niveaux d'abstraction selon les besoins et la grosseur de la carte, mais il faut prendre en considération l'espace en mémoire utilisé et disponible. Dans un contexte de jeu, le chemin détaillé total n'est pas nécessaire, donc on peut rapidement avoir un chemin partiel et raffiner au fur et à mesure entre chaque sommet du graphe abstrait.

Le coût d'ajout de la source et de la destination dans un graphe a un certain coût, mais dans le cas où un point est souvent utilisé, exemple plusieurs agents qui se dirigent vers le même endroit, il est possible de garder en mémoire ces points dans le graphe afin d'accélérer le processus en échange d'espace mémoire. L'article (Botea *et al.*, 2004) propose d'évaluer la possibilité d'ajouter la source et la destination dans le graphe de manière plus efficace afin d'économiser plus de temps de calcul. Une piste proposée est de connecter la source à un sous-ensemble creux des sommets de bordure. Une autre solution, considérée comme plus risquée, consiste à choisir les bords en direction de la destination.

1.5.2 TRANSIT

Le TRANSIT (Bast *et al.*, 2006)(Antsfeld *et al.*, 2012) est une technique rapide et optimale pour le calcul du plus court chemin dans un réseau routier. Cette technique d'abstraction a été analysée dans un contexte de jeu vidéo pour un monde de jeu en grille 2D et, après comparaisons, TRANSIT peut être plus rapides que l'algorithme A* standard, à condition d'accepter une utilisation plus élevée de l'espace mémoire.
TRANSIT suit l'intuition que, si on veut parcourir de longues distances, tout comme dans le monde réel, un sous-ensemble d'arêtes est plus souvent utilisé que d'autre en général. Les sommets associés à ces arêtes sont donc appelés des sommets de transit.
La planification de chemin à l'aide de la technique de TRANSIT se fait en deux étapes principales :

— Le précalcul sur la carte afin d'accélérer le processus.

— La planification du chemin détaillée.

L'étape de précalcul a en elle-même deux sous étapes, soit trouver les sommets de transit et construire une base de données de coûts exacts. La recherche de sommets transit les identifie en séparant la grille en sous-groupes carrée d'une taille prédéfinie. En considérant pour chaque groupe C, il existe un périmètre I de taille arbitraire qui l'entoure et un plus gros périmètre O de taille arbitraire qui entoure ce dernier, comme dans la figure 1.4. Il faut trouver pour chaque sommet du graphe de navigation dans C le chemin le plus court vers chaque sommet du graphe de navigation dans O qui utilise au moins un sommet présent dans I. Pour tous les chemins trouvés, on garde dans un conteneur V_c les sommets dont l'arête va de la zone C vers I, dans un conteneur V_i les sommets dont l'arête va de la zone I vers O ou C et dans un conteneur V_o les sommets dont l'arête va de la zone O vers I.

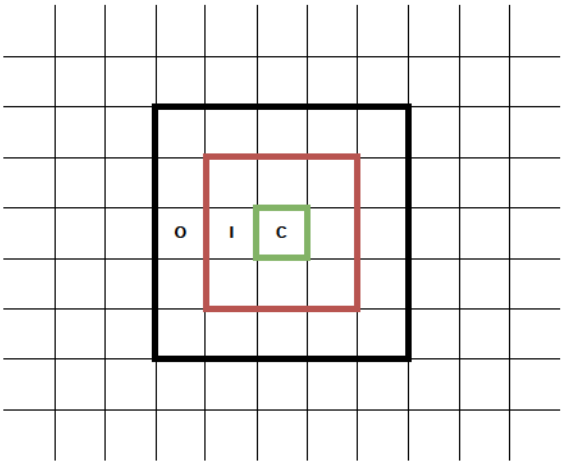


Figure 1.4 – Exemple d'une grille TRANSIT d'une carte où chaque sous-groupe sera considéré dans la recherche de sommets de transit en tant que groupe C avec ses périmètres I et O.

Les sommets transit pour tous les sommets dans la section C sont les sommets présents dans le conteneur V_i qui permettent de trouver le plus court chemin entre chaque sommet dans V_c vers chaque sommet dans V_o.

La base de données construite correspond au plus court chemin entre chaque sommet dans le groupe C vers chaque sommet transit et les plus courts chemins pour chaque sommet transit vers les autres sommets transit.

Pour la planification de chemin, TRANSIT sépare les requêtes en deux catégories : locale ou globale. Si la distance entre deux points est plus courte qu'un certain nombre de sous-groupes, la requête est considérée locale et l'algorithme de base est utilisé. La taille de I plus la distance entre I et O comme valeur afin de différencier local et global est proposée (Bast *et al.*, 2006). Pour la recherche globale, on commence par chercher les sommets transit de la source et de la destination respectivement, puis on sélectionne les sommets transit qui minimise le coût des chemins : source vers Transit source, Transit source vers Transit destination et Transit destination vers destination. On applique un algorithme de planification de chemin détaillée dans chaque sous-groupe, exemple A*.

L'utilisation d'une valeur afin de briser le problème de symétrie qui peut être présent dans une grille 2D est proposée. L'application de TRANSIT dans une carte symétrique provoque une grande utilisation de l'espace mémoire et un long temps de calcul de chemin. Pour un graphe de navigation $G = (V, E)$, on définit L comme étant la longueur du plus long plus court chemin du graphe. Pour chaque arête dans le graphe, on ajoute à son coût une valeur aléatoire dans l'intervalle $(0, \epsilon)$ où $\epsilon = \frac{1}{L}$.

TRANSIT est une technique testée sur des environnements statiques en grille 2D. L'aspect hiérarchique est intéressant, mais il faut donc évaluer la possibilité d'appliquer des changements dynamiques dans la base de données de sommets transit et des chemins précalculés. L'ajout d'une valeur dans le coût des arêtes pour régler le problème de symétrie est nécessaire pour les cartes venant de *Moving AI* (Sturtevant, 2012).

1.5.3 Contraction Hiérarchique

La contraction hiérarchique (Geisberger *et al.*, 2012) est un algorithme de prétraitement qui sert à diminuer le facteur de branchement lors de la recherche dans un graphe, ce qui accélère substantiellement la recherche de chemin par rapport a une recherche de chemin dans un graphe n'ayant subi aucune processus de prétraitement. Elle a été utilisée comme base pour la proposition présentée dans le chapitre 2.

1.5.3.1 Construction du graphe hiérarchique

La construction d'une hiérarchie s'effectue en contractant les sommets selon ordre de priorité déterminé au préalable par des critères formant une heuristique. L'ordre des sommets doit soigneusement être choisi pour que la techniques soit efficace. Cette heuristique est appliquée sur chaque sommet dans le but d'avoir un score de priorité, où le score le moins élevé indiquera quel sommet contracter en premier. (Geisberger

et al., 2012) propose plusieurs critères afin de former l'heuristique. Le plus important est le *Edge Difference* (ED) correspondant à la différence entre le nombre d'arêtes dans le graphe avant et après la contraction d'un sommet. Il y a aussi des critères d'uniformité, visant à faire en sorte que la contraction des sommets ne se fasse pas que dans une section du graphe, mais partout dans le graphe. Par exemple, le *Deleted Neighbor* (DN), qui comptabilise le nombre de voisins contractés d'un sommet, et le *Original Edges Term* (OET), qui comptabilise le nombre d'arêtes original de chaque arête. Afin d'éviter d'avoir un temps de contraction du graphe trop lent, il est aussi possible d'avoir une heuristique calculant le nombre de sommets visités lors de la contraction d'un sommet, appelé le *Cost of Contraction* (CoC). La combinaison de ces critères ainsi que l'utilisation d'une valeur de poids sur chacun afin de mitiger leur impact représente l'heuristique utilisée (voir équation 1.4). Un nombre infini de critères pour l'heuristique peut être implémenté lors de la création de la liste de priorité, mais (Geisberger *et al.*, 2012) souligne que cela entraîne une inflation de paramètres à ajuster. La complexité de l'heuristique est de trouver celle qui avantage les besoins de la carte tout en gardant la liste de critères simple et spécifique à la situation du graphe.

Après avoir sélectionné l'heuristique pour la contraction, l'ordre de contraction des sommets est déterminé en faisant une simulation de contraction de chaque sommet du graphe tout en gardant le score heuristique associé dans une liste de priorité ordonné du plus petit score au plus grand.

$$score = ED + \alpha CoC + \beta DN + \eta OET \tag{1.4}$$

1.5.3.2 Contraction d'un sommet

Lors de la contraction du graphe, chaque sommet se voit attribuer un niveau de 0 à $N - 1$, où N est le nombre de sommets dans le graphe. On commence par sélectionner le sommet v avec le plus petit score heuristique, soit le premier dans la liste de priorité, et on fait la recherche d'un chemin témoin pour toutes les paires de sommets entrants vers les sommets sortant de celui-ci s'il existe un chemin plus court ne nécessitant pas le sommet v. Un chemin témoin (ou *witness path* en anglais) correspond au plus court chemin entre un sommet entrant vers un sommet sortant ne nécessitant pas le sommet v si son coût est plus petit que le coût du chemin entre le sommet entrant et le sommet sortant passant par le sommet v. Si le chemin le plus court est celui passant par le sommet à contracter v, une arête raccourcie est créée afin de garder l'information dans le graphe. Une arête raccourcie est donc un indicateur qu'il existe un chemin

possible entre deux sommets.

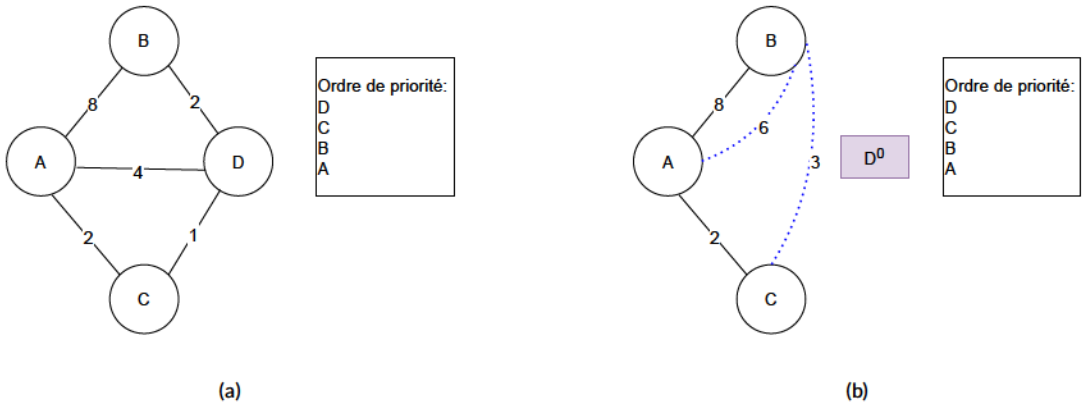


Figure 1.5 – Contraction du graphe de navigation (a) selon l'ordre de priorité. Puisque le sommet D est contenu dans le plus court chemin entre A-B et B-C, des raccourcis sont ajoutés au graphe (b).

De plus, (Geisberger *et al.*, 2012) ajoute un critère d'arrêt de recherche si la distance du chemin est au-dessus d'une certaine limite. Ceci accélère la contraction tout en n'ajoutant que quelques arêtes raccourcies superflues. Dans un contexte où le graphe représente un monde entre grilles avec des arêtes de poids uniformes, (Storandt, 2013) propose une autre technique de contraction qui accélère le processus. En effet, si les paires de sommets à évaluer et le sommet à contracter forment une ligne droite et qu'il n'y a aucun obstacle entre les trois sommets, on peut éviter de faire une recherche de chemin pour la paire de sommets et ne pas créer le raccourci associé. De plus, si le coût du chemin de la paire de sommets passant par le sommet à contracter correspond à la différence de position entre les deux sommets de la paire, on peut limiter la zone de la recherche de chemin par un rectangle délimité par la paire de sommets. Ceci rend la recherche plus courte en la dirigeant dans une direction précise.

Suivant la contraction de chaque sommet, le score de ses sommets voisins est mis à jour selon la nouvelle composition du graphe courant. Une mise à jour du score du prochain sommet potentiel dans la liste de priorité est aussi appliquée. Cette étape se nomme la mise à jour paresseuse, *Lazy Update* en anglais, puisqu'il met à jour que le score d'un sommet dans la liste. S'il y a eu un trop gros nombre de mise à jour paresseuse appliquée sur la liste de priorité, une mise à jour complète de tous les sommets restant dans la liste est faite. Le nombre de mise à jour paresseuse allouée entre chaque mise à jour entière est une valeur déterminée au préalable, tout comme les critères heuristiques.

La figure 1.6 illustre un exemple de graphe contenant plus de sommets démontrant où l'ordre de priorité

établie par une heuristique simple, où ?? montre le graphe de base et ?? indique l'ordre dans lequel les sommets seront contractés. Dans la figure 1.7, nous constatons à l'aide de la représentation hiérarchique que 5 arêtes raccourcies sont ajoutées dans le processus de contraction. La figure 1.8 démontre le graphe hiérarchique final qui est utilisé lors de la recherche de chemin contenant les nouveaux raccourcies ainsi que les niveaux de chaque sommet dans la hiérarchie.

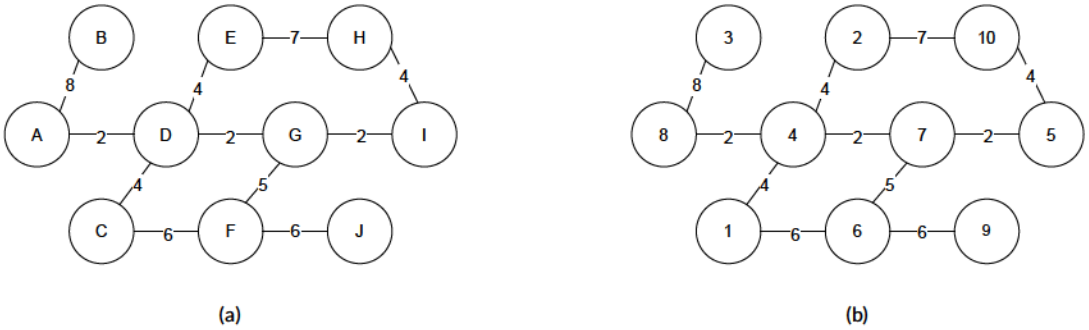


Figure 1.6 – Exemple d'un graphe de navigation (a). La contraction hiérarchique créé un ordre de sommets à contracter selon un score heuristique de priorité (b).

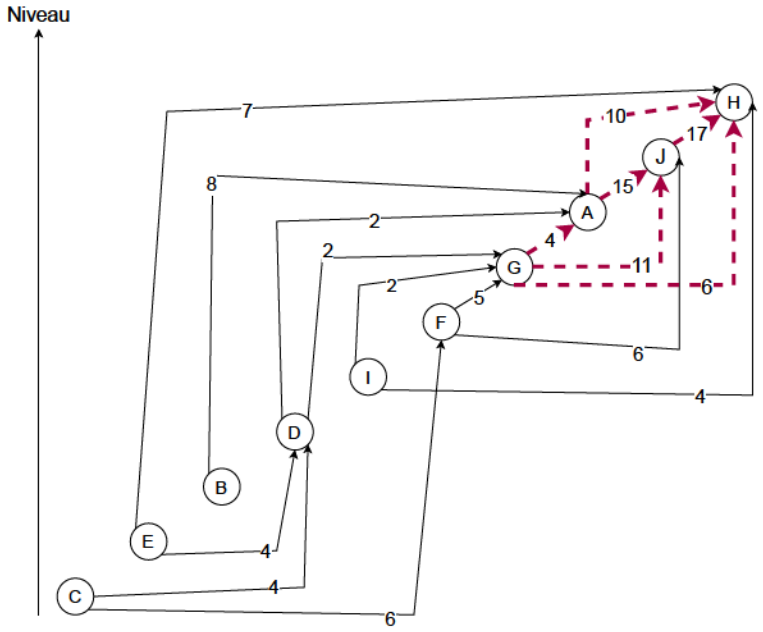


Figure 1.7 – La contraction hiérarchique créé selon l'ordre de sommets contracté. Le plus petit nombre est à la base de la hiérarchie ainsi que les raccourcis nécessaires afin de préserver l'optimalité.

L'utilisation de ces arêtes raccourcies permet de trouver un chemin vers la destination plus rapidement puisqu'il a la possibilité de visiter moins de sommets tout en préservant l'optimalité. La représentation

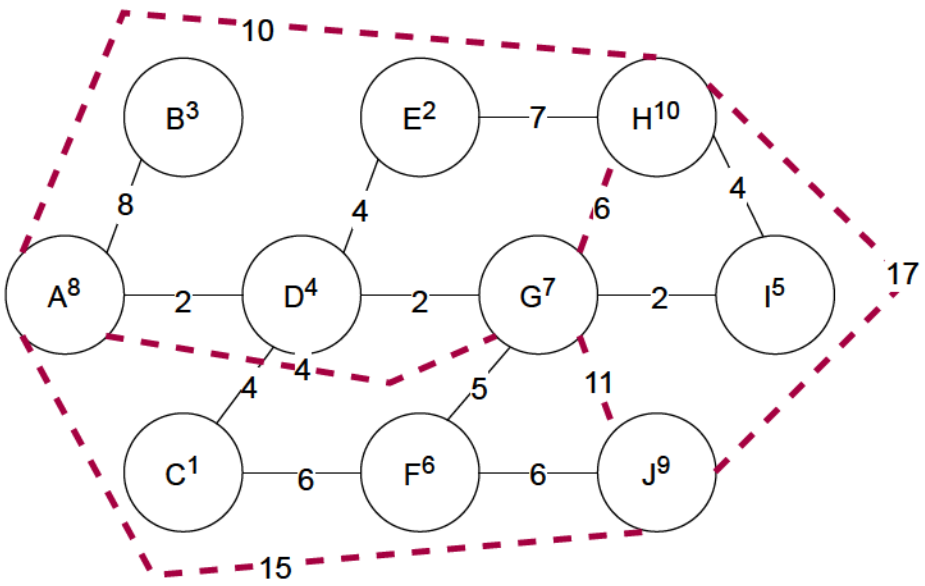


Figure 1.8 – La représentation en graphe hiérarchique du graphe de navigation contient le niveau des sommets, les raccourcis et les arêtes de base.

finale du graphe contient donc les nouveaux raccourcis ainsi qu'un indicateur de niveaux hiérarchique sur chaque sommet.

1.5.3.3 Recherche de chemin dans le graphe hiérarchique

La planification de chemin dans le graphe hiérarchique final est appliquée en utilisant l'algorithme de recherche bidirectionnelle, comme vu à la section 1.3.1, avec la particularité supplémentaire qu'un sommet ne peut pas visiter un sommet voisin de niveau inférieur. La recherche de chemin se fait donc de la source et de la destination jusqu'à ce qu'il y a une intersection à un niveau supérieur si un chemin existe. Si un chemin existe, un déroulement des raccourcis utilisés est appliqué afin d'avoir le vrai chemin dans le graphe d'origine, puisque les raccourcis ne correspondent pas à des arêtes applicables pour un agent. Si on prend comme exemple le graphe hiérarchique à la figure 1.8, en faisant une recherche de chemin du sommet A comme source au sommet J comme destination, un raccourci direct de A à J permet de trouver le chemin rapidement.

L'article de Geisberger et al., (Geisberger *et al.*, 2012), fait une comparaison entre la recherche de chemin dans un graphe avec le prétraitement afin d'avoir une contraction hiérarchique en utilisant l'algorithme

de Dijkstra et la recherche de chemin avec l'algorithme de Dijkstra dans un graphe sans aucun prétraitement. Cette comparaison démontre que pour un réseau routier de l'Europe, l'algorithme de Dijkstra dans un graphe sans prétraitement le temps moyen de recherche de chemin est environ 40 000 plus lent que lorsque ce même algorithme est appliqué dans un graphe hiérarchique. Ceci démontre que lorsque la taille du graphe est élevé, l'utilisation de la contraction hiérarchique pour le prétraitement est avantageux afin de limiter le facteur de branchement. Sachant que l'algorithme A* améliore l'algorithme de Dijkstra en permettant de trouver le chemin optimal en parcourant moins de sommets, voir section 1.3.2, la recherche de chemin à l'aide de l'algorithme A* dans un graphe ayant eu un prétraitement de contraction hiérarchique parcours moins de sommets que la recherche de chemin à l'aide de l'algorithme A* dans un graphe sans prétraitement.

1.5.3.4 Mise à jour du graphe hiérarchique

Afin de réparer le graphe hiérarchique face à des changements dynamiques mineurs affectant le poids des arêtes dans la carte, (Geisberger *et al.*, 2012) offre une approche permettant de le mettre à jour efficacement sans devoir recommencer le processus de contraction du graphe complet. Pour ce faire, il garde en mémoire l'ordre de priorité des sommets afin d'identifier les sommets et les raccourcis impactés.

Le but est que, pour une arête changée, nous devons identifier tous les raccourcis qui l'utilisent et les réparer. Si une arête est supprimée, les raccourcis l'utilisant sont supprimés aussi. Après la réparation des arêtes, il faut contracter de nouveau les sommets connectés à celles-ci dans les cas où l'arête a été ajoutée au graphe, a un poids plus petit ou si un raccourci l'utilisant a été supprimé ou a eu son poids augmenté. Ceci limite le nombre de contractions en ne recommençant pas le processus du début.

1.5.3.5 Contraction Hiérarchique dans une carte en grille 2D

À cause de l'uniformité des sommets dans une carte en grille 2D (voir la figure 1.9), (Storandt, 2013) propose une différente approche lors de la contraction d'un sommet afin de contourner se problème et ne pas avoir un nombre exponentiel de raccourcis ajoutés.

Dans le cas de la contraction d'un sommet v (voir la figure 1.10), tous les sommets connectés au sommet courant sont autant un sommet entrant qu'un sommet sortant. Lors de la recherche de chemins témoins entre les paires de sommets connectés, u et w, si les deux sommets sont sur le même axe, il n'est pas

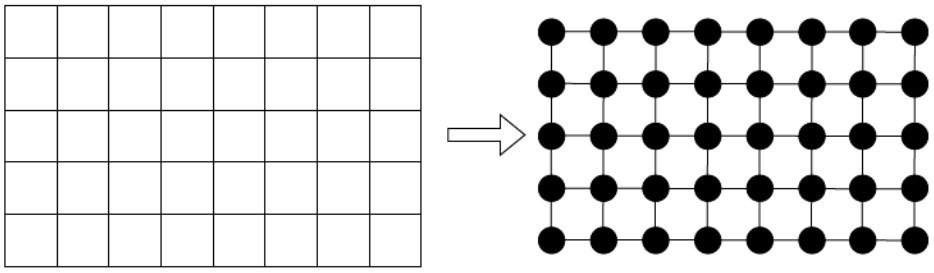
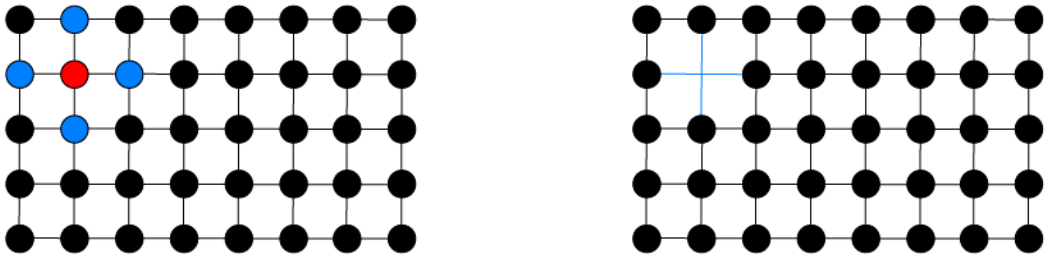


Figure 1.9 – Exemple de carte en grille 2D. Le graphe résultant contient des sommets de distances uniformes.

nécessaire de faire la recherche d'un chemin témoin ni d'ajouter une arête raccourcie dans le graphe s'il n'y a aucun obstacle entre eux. En cas d'obstacle entre u et w, la recherche de chemin témoin est appliquée et, si le chemin n'est pas plus court que le chemin (u, v, w), (u, v, w) est ajouté dans le graphe en tant que raccourci. Si u et w ne partagent aucun axe et que la différence positionnelle entre eux correspond au coût du chemin (u, v, w), la recherche de chemin témoin est limité par le rectangle formé par u et w, ce qui permet d'accélérer le processus. Un raccourci est ajouté seulement si le coût de (u, v, w) est strictement plus petit que le chemin témoin trouvé.



(a) (b)

Figure 1.10 – Exemple de contraction du sommet en rouge. Les sommets en bleus n'ont pas besoin de faire la recherche de chemin témoin s'ils sont sur le même axe parce qu'il n'y a pas d'obstacle et nous n'ajoutons pas de raccourci. Dans le cas des sommets ne partageant pas un axe, la recherche de chemin témoin est limité par le rectangle formé par les deux sommets. Un chemin témoin existe et à le même coût que le chemin utilisant le sommet à contracter, donc nous ne créons pas de raccourci.

1.6 Génération et mise à jour dynamique de données

Afin d'illustrer l'importance de la recherche de chemin dans un environnement dynamique, il faut mentionner que plusieurs angles sont explorés en tout temps. Cette sous-section met de l'avant l'idée d'utiliser le traitement d'image afin de mettre à jour dynamiquement les données de l'environnement.

1.6.1 FAR Planner

Les auteurs de l'article (Yang *et al.*, 2022) présentent l'architecture d'un robot se déplaçant dans un environnement réel pouvant avoir des changements dynamiques. La représentation interne du robot telle qu'affichée à la figure 1.11 se construit à travers deux étapes :

— Extraction des polygones obstacles par les senseurs

— Création et mise à jour du graphe de visibilité de l'environnement

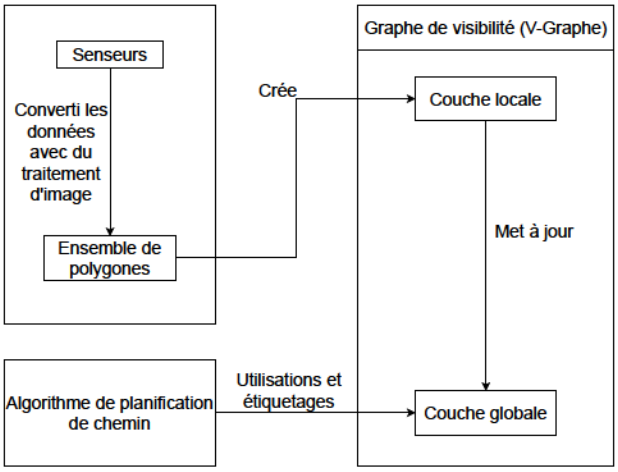


Figure 1.11 – Vue d'ensemble de la représentation interne du robot

1.6.1.1 Extraction des polygones obstacles par les senseurs

Cette étape utilise des techniques de traitement d'image. Le senseur commence par collecter un ensemble de points S, représentant les obstacles autour du robot. Ensuite, il projette S sur une image binaire, I, avec des pixels noirs et blancs, où les pixels noirs représentent l'espace vide et les pixels blancs les obstacles. Cette image est centrée sur le robot et l'ensemble de points S est multiplié par la taille de celui-ci. Un filtre de moyenne est appliqué sur I afin d'avoir une image en niveaux de gris, I_{blur}. Les points de bordure sont extraits à partir de I_{blur}, ce qui nous donne un ensemble de polygones fermés avec des sommets denses sur les contours, $P^k_{contour}$. Pour chaque polygone, on diminue le nombre de sommets en éliminant ceux qui ont un angle interne inférieur à un seuil, déterminé par la connexion de deux arêtes sur le contour de l'obstacle. Ceci donne pour finir les polygones extraits, notés $\{P^k_{local}\}$.

1.6.1.2 Création et mise à jour du graphe de visibilité de l'environnement

Le graphe de visibilité, *V-Graph*, contient deux couches : la couche locale, L_{local}, entourant le robot, et la couche globale, L_{global}, couvrant l'environnement observé. Ces deux couches sont mises à jour à chaque trame de données.

En utilisant $\{P^k_{local}\}$, on construit un *V-Graph* réduit partiellement sur L_{local}. E_{local} représente les arêtes de visibilité dans le *V-Graph*. Parmi ceux-ci, on ignore les arêtes plus longues qu'un certain seuil s'ils connectent deux polygones en passant par l'angle interne d'un ou des deux sommets et on conserve ceux qui passent

« autour ». Tous les arêtes plus courtes que le seuil sont conservées. La figure 1.12 illustre cette situation où les lignes rouges sont les arêtes représentant deux polygones et les lignes jaunes et cyans sont des arêtes plus longues qu'un seuil prédéterminé. Les arêtes jaunes sont ignorées parce qu'elles passent par l'angle interne des sommets des polygones, en gris. Seule l'arête cyan est conservée parce qu'elle passe autour.

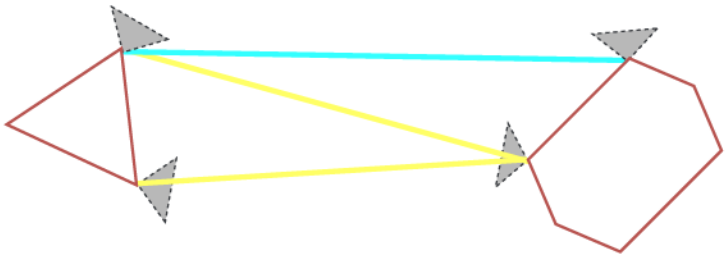


Figure 1.12 – Exemple de réduction d'arêtes

La mise à jour de la couche globale, L_{global}, se fait en fusionnant L_{local} dans celle-ci pour les parties qui se chevauchent. On commence par associer les sommets entre ceux des polygones dans la couche locale, $\{P^k_{local}\}$, et ceux de la couche globale, $\{P^k_{global}\}$. Pour chaque sommet de $\{P^k_{local}\}$, si le sommet le plus proche dans $\{P^k_{global}\}$ à une distance euclidienne plus petite qu'un seuil, il est associé à celui-ci. Par la suite, on applique du *robust fitting* pour chaque groupe de sommets $\{P^k_{local}\}$, recueillis sur plusieurs trames de données, associé à un sommet $\{P^k_{global}\}$. Les sommets sont filtrés selon un processus itératif où on calcule la moyenne et la covariance de ceux-ci. Ces valeurs sont ensuite utilisées afin de retirer les sommets considérés comme des données aberrantes. Le processus s'arrête si les sommets restent les mêmes pendant deux itérations ou que le nombre maximal d'itérations est atteint. La position moyenne des sommets restants correspond à la mise à jour du sommet global associé. Les sommets globaux n'ayant pas été associés après un certain nombre d'images de données sont retirés de l'ensemble. Les sommets locaux n'ayant pas été associés sont ajoutés comme nouveaux sommets dans l'ensemble global. Les arêtes locales existantes dans l'ensemble d'arêtes globales sont mises à jour et les autres sont ajoutés comme nouvelles arêtes globales.

1.6.1.3 Algorithme de planification de chemin

Le planificateur ajoute la position courante du robot et la destination en tant que sommets dans la couche globale et les connecte aux sommets n'ayant aucune arête non visible, sans obstacle, autour d'eux. Il applique ensuite l'algorithme de recherche en largeur sur la couche globale en se propageant à travers les arêtes globales afin de trouver le chemin le plus court si possible.

Lors de la navigation dans l'environnement, les sommets avec arêtes directement visibles par le robot font partie de l'espace libre tandis que les autres sont dans l'espace inconnu. À la fin de la navigation, le *V-Graph* est enregistré avec une étiquette pour chaque sommet indiquant son type d'espace. Le *V-Graph* peut être chargé dans de future recherche comme carte globale de départ. Avec une carte chargée, il est possible d'indiquer si la recherche de chemin se fait seulement sur l'espace ouvert ou sur les deux espaces.

Le traitement d'image peut être un procédé coûteux dans un contexte de jeu. En revanche, la segmentation des données par agent dans une couche locale et une couche globale peut permettre de mettre à jour le monde plus rapidement.

CHAPITRE 2

HEURISTIQUE POUR LA CONTRACTION HIÉRARCHIQUE DANS UN MONDE FORTEMENT DYNAMIQUE

Dans ce chapitre, nous proposons une adaptation de la méthode de contraction hiérarchique (Geisberger *et al.*, 2012) présentée au sous-chapitre 1.5.3 afin de mieux considérer les changements dynamiques dans une carte de jeu vidéo en introduisant des nouveaux critères dans le calcule d'heuristique pour les sommets du graphe. Notre hypothèse est que cette nouvelle heuristique permet d'accélérer la mise à jour de la carte et de la planification de chemin.

La contraction hiérarchique présentée par (Geisberger *et al.*, 2012) a été initialement conçue pour des routes n'ayant pas beaucoup de changements. Ces routes stables sont affectées par des changements de manière sporadique, soit par des travaux de construction peu fréquents ou du trafic affectant les mêmes arêtes périodiquement. Malgré son adaptation pour s'ajuster à des changements dynamiques, elle comporte des lacunes de performance lors de changements dynamiques à fréquence plus élevée. De plus, la création de la hiérarchie et des raccourcis dans le graphe est influencée par les propriétés intrinsèques d'une carte urbaine, soit la présence de grandes artères routières et de petites rues moins utilisées, ce qui fait en sorte que la plupart des raccourcis représentent les boulevards et les autoroutes. La modification de la contraction hiérarchique pour l'adapter à un monde de jeu en grille 2D par (Storandt, 2013) (voir le sous-chapitre 1.5.3.5) permet d'avoir une alternative sur laquelle nous pouvons nous baser pour notre proposition.

En utilisant la contraction hiérarchique comme telle, nous pouvons supposer que la planification de chemin A* dans le graphe en utilisant les raccourcis précalculés est plus efficace que la planification de chemin A* dans le graphe d'origine. En effet, comme le démontre la figure 2.1, grâce aux raccourcis présents dans le graphe hiérarchique et à la limitation du facteur de branchement d'un sommet vers seulement des sommets de niveaux supérieurs, le nombre de sommets explorés est moindre, ce qui accélère la planification d'un chemin.

Comme vu à la figure 2.1, le nombre de sommets visités lors de la planification de chemin avec l'algorithme A* dans un graphe classique (8 sommets) est plus élevé qu'avec la planification de chemin bidirectionnel en utilisant l'algorithme A* (3 sommets). La planification de chemin utilisant un graphe hiérarchique serait donc plus efficace que la planification de chemin utilisant un graphe classique, car elle permet de diminuer le facteur de branchement des sommets. Par contre, la contraction hiérarchique présentée par (Geisberger

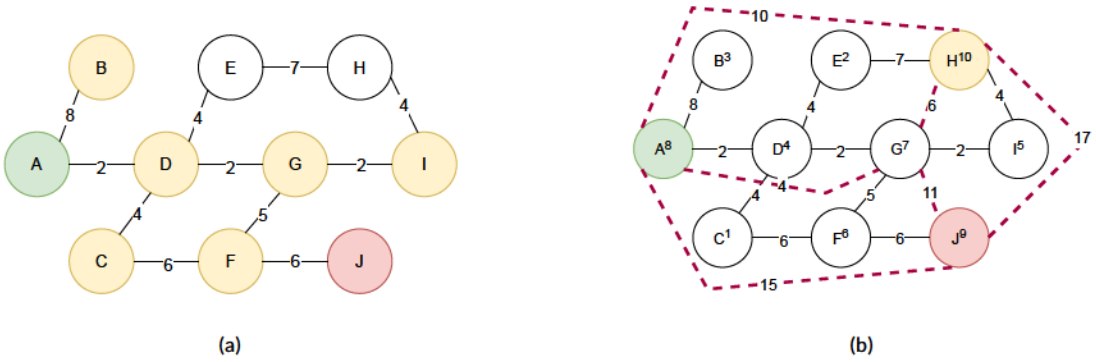


Figure 2.1 – Comparaison de la planification entre l'algorithme A* dans un graphe classique (a) et la planification bidirectionnel avec A* dans un graphe hiérarchique (b). Le sommet vert est la source, le sommet rouge est la destination et les sommets jaunes sont les sommets explorés lors de la planification de chemin.

et al., 2012) et (Storandt, 2013) ne se concentre pas sur l'aspect dynamique lors de la création du graphe hiérarchique. Notre proposition se base sur l'idée que changer l'ordre de contraction des sommets en prenant en considération le dynamisme de chacun d'entre eux peut affecter positivement les performances en lien avec la réparation du graphe hiérarchique lors de changements dynamiques dans le monde.

Dans un contexte où le monde de jeu en grille peut avoir multiples changements dynamiques et simultanés durant sa durée de vie, il faut avoir un moyen de modifier le graphe hiérarchique de façon rapide et efficace. En effet, il est possible d'avoir un mur qui se détruit ou l'ajout d'obstacles qui bloquent des chemins. Ceci implique que le graphe peut avoir de nouveaux sommets et arêtes ou bien avoir des sommets et des arêtes qui deviennent soit inactifs, soit marqués avec un coût très grand ou infini. Le but de la proposition est donc d'avoir des critères heuristiques lors de la création de la hiérarchie des sommets qui prennent en considération l'aspect dynamique des sommets dans une carte en grille.

Ce mémoire s'est concentré sur l'amélioration de la mise à jour du graphe de contraction lors de changements dynamiques dans un monde de jeux en grille. L'hypothèse est que le résultat de la recherche de chemin n'est pas affecté puisque l'algorithme de planification de chemin reste le même, mais les modifications apportées sur l'heuristique utilisée par la contraction hiérarchique sont plus efficaces lors de la mise à jour de celle-ci durant la durée de vie du monde de jeu.

2.1 Heuristique d'ordonnancement pour le dynamisme

Comme expliqué dans la section 1.5.3.4, lors du changement de poids d'une arête dans le graphe hiérarchique, Geisberger (Geisberger *et al.*, 2012) ajuste récursivement tous les raccourcis utilisant ladite arête. Il garde la liste de priorité faite au préalable lors de la création du graphe hiérarchique afin de savoir quels sommets, et par conséquent quels raccourcis, sont affectés par ce changement. Par contre, il n'y a aucune mention de critère de dynamisme pouvant influencer l'ordre de priorité des sommets. Ceci signifie que si une arête associée à un sommet placé au début de la liste de priorité, et donc qu'il a été contracté tôt dans le processus, le nombre de raccourcis l'utilisant devant être modifié est potentiellement plus élevé que s'il est contracté plus tard dans le processus.

Les propositions suivantes s'appuient sur l'hypothèse qu'en connaissant au préalable quels sommets sont susceptibles d'être mise à jour, il est possible de modifier l'ordre de priorité des sommets nous permettant d'avoir une réparation de graphe hiérarchique plus efficace, tout en conservant la performance de la recherche de chemin.

Il est possible d'annoter les sommets plus portés à être modifié comme étant dynamique à l'aide d'une valeur binaire. Avec cette information, on peut l'ajouter comme critère heuristique qui affectera le score du sommet lors de la création de la liste de priorité de contraction. Un sommet dynamique signifie que, durant le cycle de vie du graphe de navigation, ce sommet et ses arêtes vont être modifiés, ce qui va aussi affecter ses voisins.

Sachant que certains sommets sont dynamiques et vont affecter la hiérarchie, il faut aussi prendre en considération les voisins directs de ces sommets et comment ils vont aussi affecter le graphe final. En ayant un compteur du nombre de voisins direct dynamique de chaque sommet, on peut faire en sorte que ces sommets aient aussi un score de priorité élevé. Le but est que les sommets statiques ayant des voisins dynamiques soient impliqués dans le moins de raccourcis possible, facilitant donc la modification du graphe hiérarchique lors des changements dynamiques du monde parce qu'on aura moins d'arêtes a exploré afin de propager l'information.

La motivation derrière cette proposition est d'avoir les sommets dynamiques ainsi que leur voisins directs à la fin de la liste de priorité de contraction des sommets. En suivant ce principe, lors de contraction hiérarchique, le nombre de raccourcis créé en fonction d'un sommet dynamique ou d'un voisin d'un sommet

dynamique est limité. Ceci permet de limiter le nombre d'arêtes raccourcies à modifier au moment de la mise à jour dynamique du graphe hiérarchique à la destruction ou à la création d'un mur. C'est donc pour cela qu'afin de minimiser le nombre de changements dans le graphe hiérarchique, il faut que les sommets dynamiques soient contractés en dernier ou du moins le plus tard possible.

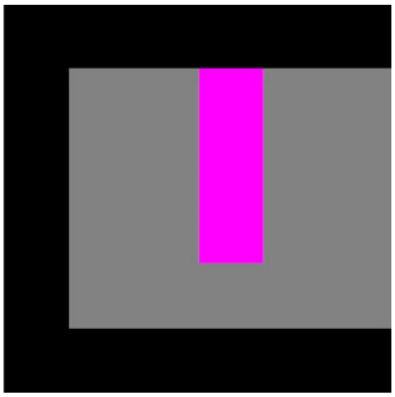
La formule heuristique que nous proposons dans cette recherche est donc l'équation (2.1), qui se base sur l'équation heuristique de base (1.4) en ajoutant les critères de dynamismes sur le sommet SD, qui est un indicateur vrai ou faux selon que le sommet courant est marqué comme dynamique ou non, et NVD, qui est le nombre de sommets dynamiques directement voisin au sommet courant. Ces deux critères ont eux aussi un poids afin d'ajuster leur impact sur le score de priorité.

$$score = ED + \alpha CoC + \beta DN + \eta OET + \gamma SD + \sigma NVD \tag{2.1}$$

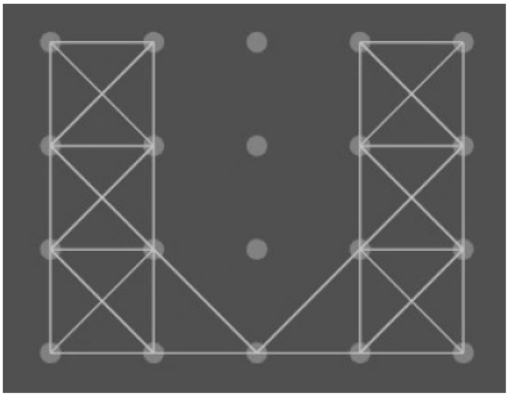
En suivant ces heuristiques, le résultat attendu est d'avoir une contraction hiérarchique dans un contexte dynamique qui a un équilibre entre l'efficacité de la planification de chemin bidirectionnelle dans le graphe hiérarchique et de son ajustement au moment d'un changement dynamique dans la carte. Avec la nouvelle heuristique, nous supposons que l'ordre de priorité des sommets sera différent et que l'ordre de priorité suivant l'heuristique proposée dans cette recherche place les sommets dynamiques ainsi que leur voisin le plus loin dans la liste tout en gardant une recherche de chemin la plus rapide possible. En utilisant comme exemple une petite carte de jeu théorique (voir figure 2.2), nous pouvons constater que les listes de priorité des sommets suivant l'heuristique de base et l'heuristique proposée n'ont pas la même séquence de sommets et de ces faits les raccourcis créés sont aussi différents (voir figure 2.3).

La procédure de mise à jour dynamique des données de la carte diffère de la méthode utilisée dans (Geisberger *et al.*, 2012) à cause du type de carte. Puisque la carte est en grille avec des cases de taille uniforme avec des connexions uniformes avec leurs voisins, le dynamisme représente deux situations, soit l'ajout ou le retrait d'arêtes dans la carte. Ceci implique que l'algorithme de mise à jour du graphe hiérarchique n'a pas besoin de déterminer si un raccourci est invalide ou non selon le nouveau poids d'une arête. Si une arête existante change, ceci signifie qu'elle a été désactivée et que tous les raccourcis l'utilisant sont maintenant invalides. Dans le cas de l'ajout d'une arête, tout comme dans (Geisberger *et al.*, 2012) on sait qu'aucun raccourci ne l'utilise et donc il n'est pas nécessaire d'identifier les raccourcis affectés.

(a) Carte de base où chaque case noire représente un obstacle

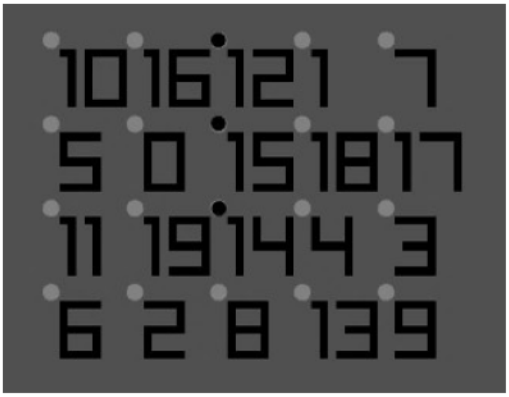


(b) Carte avec les sommets dynamiques en magenta



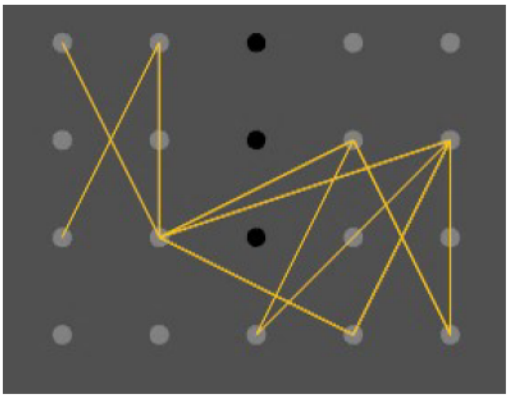
(c) Graphe de navigation représentant la carte avant tout changement dynamique

Figure 2.2 – Carte de jeu théorique de taille $6x6$ contenant 20 sommets dont 3 étant marqués comme sommet dynamique.

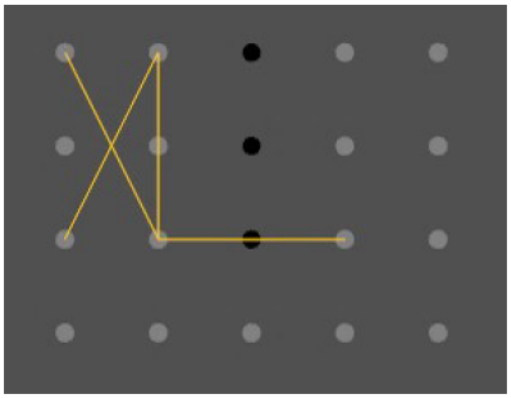


(a) Ordre des sommets avec l'heuristique de base

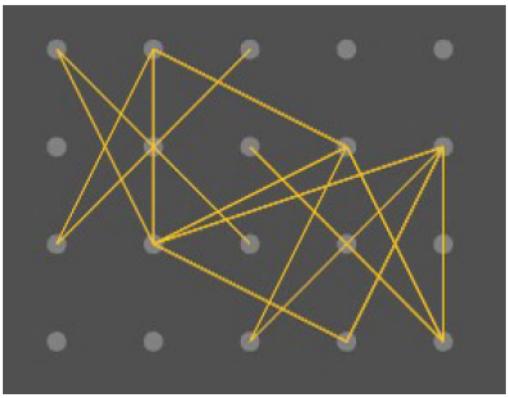
(b) Ordre des sommets avec l'heuristique proposée



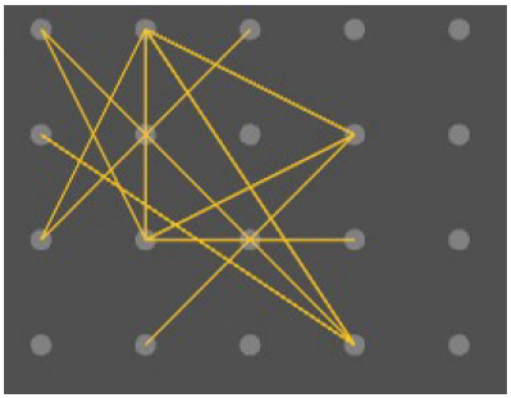
(c) Raccourcis créés avec l'heuristique de base



(d) Raccourcis créés avec l'heuristique proposée



(e) Mise à jour du graphe hiérarchique avec l'heuristique de base



(f) Mise à jour du graphe hiérarchique avec l'heuristique proposée

Figure 2.3 – On peut voir ici que l'ordre de priorité des sommets diffère selon la formule heuristique. Ce changement affecte le nombre de raccourcis nécessaires au graphe hiérarchique, soit 11 arêtes raccourcis avec l'heuristique de base (non dynamique) et 4 arêtes raccourcis avec l'heuristique proposée (dynamique). Lors de la mise à jour du graphe hiérarchique après la destruction des murs (sommets en noir), l'heuristique de base à 15 raccourcis et l'heuristique proposée en a 11. Ceci va aussi affecter le nombre d'arêtes à visiter lors de la recherche de chemin ainsi que le nombre de raccourcis à modifier lors d'un changement dynamique dans la carte.

CHAPITRE 3
EXPÉRIMENTATION ET ÉVALUATION

Afin de comparer la méthode proposée au chapitre 2 avec différents algorithmes de planification de chemin, nous les avons implémentés dans un projet logiciel[1]. Les divers algorithmes de planification de chemin, les mondes de jeu en grille venant de (Sturtevant, 2012), leurs séries de requêtes de planification de chemin correspondantes ainsi que les techniques de collecte de performances sur le temps de calcul et l'espace en mémoire utilisé ont été intégrés dans celui-ci. De plus, il y a un système permettant d'apporter des modifications dans le monde pendant la requête de planification de chemin. (Sturtevant, 2012) comprend un ensemble de cartes en grille en deux et trois dimensions provenant de divers jeux, tels que *Dragon Age Origins* et *Warframe*, qui sont utilisées dans le cadre de recherches sur la planification de chemin. Pour cette recherche, seules les cartes en deux dimensions (2D) ont été considérées.

3.1 Architecture du projet d'évaluation

L'architecture du projet d'évaluation, présentée à la figure A.1 de l'Annexe A correspond aux composantes qui ont permis d'évaluer notre heuristique proposée en lien avec le dynamisme à la contraction hiérarchique. Elle a aussi servi à faire la comparaison entre la proposition, la contraction hiérarchique avec l'heuristique de base et les autres techniques existantes.
Le système est réparti selon les parties suivantes :

- Le module Interface visuelle est l'engin qui utilise une base de code offert en ligne[2] faisant l'intégration de Raylib[3] pour afficher la carte de jeu et Dear ImGui[4] pour avoir des fenêtres interactives.
- Le module Évaluateur de performance qui permet de lancer les étapes de prétraitement et de recherche de chemin sans devoir afficher l'interface graphique.
- Le module Gestionnaire qui s'occupe de la carte, sa transition en graphe de navigation et la modification dynamique de celle-ci.
- Le module Graphe qui est la carte de jeu en graphe de navigation contenant les sommets, les arêtes

1. https://gitlab.info.uqam.ca/beauvais-monestime.katia/planificationdynamique
2. https://github.com/raylib-extras/rlImGui
3. https://www.raylib.com/
4. https://github.com/ocornut/imgui

et les scénarios de planification de chemin s'ils existent.

— Le module Prétraitement qui applique la contraction hiérarchique.

— Le module Planification de chemin qui permet d'utiliser divers algorithmes de planification de chemin.

— Le module Sauvegarde de performance qui exporte les résultats des planifications de chemin ainsi que du prétraitement avec les modifications dynamiques du graphe.

Le module Sauvegarde de performance crée, pour chaque carte de test, un ensemble de fichiers textes et CSV contenant les résultats du prétraitement et de recherche de chemin. Les cartes de test utilisées par l'interface graphique et le système de test de performance sont des fichiers textes provenant de la base de données de *Moving AI*[5] (Sturtevant, 2012) créé par Nathan Sturtevant. Ces cartes viennent de plusieurs jeux, donc le contexte d'utilisation est représentatif de ce qui existe dans l'univers du jeu vidéo. Pour tester les changements dynamiques, nous soutenons l'hypothèse que nous connaissons d'avance les cases dynamiques dans la grille de jeux. Nous avons manuellement modifié ces fichiers afin d'ajouter un indicateur de dynamisme, soit des murs qui peuvent être détruits.

3.2 Implémentation de la contraction hiérarchique dans un monde en grille avec changements dynamiques

Lors de la contraction hiérarchique du projet, le graphe de navigation correspond à la traduction d'une carte en grille où chaque sommet correspond à une case de celle-ci et peut avoir jusqu'à 8 voisins directs sur les axes et en diagonale qui sont connectés à l'aide d'une arête. Le poids d'une arête fait référence à la distance d'un sommet à l'autre. Le poids d'une arête liant deux sommets partageant un axe est de 1 tandis que, pour deux sommets en diagonale il est de $\sqrt{2}$. Lors du chargement d'une grille de jeu, une case peut avoir l'un des quatre états de départ suivants : obstacle, libre, obstacle avec indicateur de dynamisme ou libre avec indicateur de dynamisme. Cette distinction a été ajoutée manuellement dans les fichiers de carte afin de simuler des murs pouvant être détruits et des portes qui se ferment dans le contexte de cette recherche. Un sommet sait s'il est dynamique et s'il est libre ou s'il constitue un obstacle. Cette information est utilisée lors de la contraction dynamique et modifiée lors de changements dynamiques de la carte de jeu.

L'ordre de priorité des sommets lors de la contraction hiérarchique utilise les mêmes critères pour l'heuristique que ceux présentés dans (Geisberger *et al.*, 2012), soit le *Edge Difference*, le *Deleted Neighbour*, le

5. `https://movingai.com/`

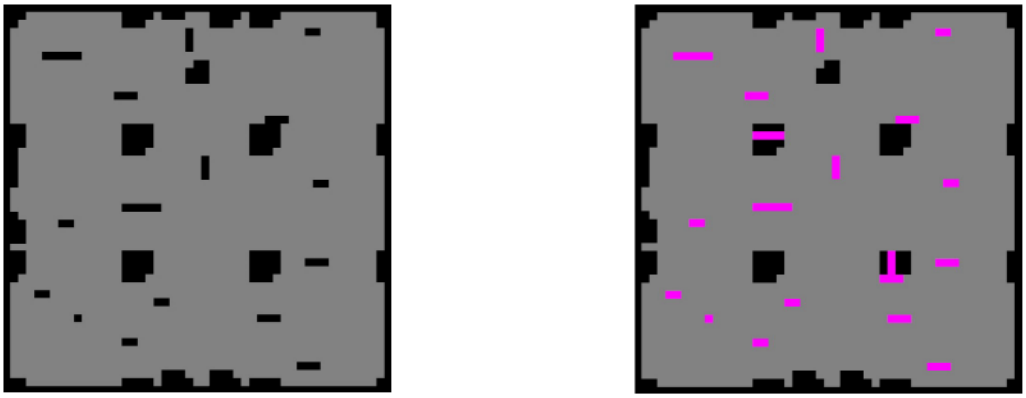
Original Edge Term et le *Cost of Contraction* qui est illustré par la formule (1.4). En plus de ceux-ci, comme expliqués par notre proposition au chapitre 2, nous ajoutons deux autres critères à l'heuristique dans le contexte de la contraction : si le sommet est dynamique et le nombre de voisins directs étant dynamiques. Le but de cette heuristique (équation (2.1)) est d'influencer l'ordre de priorité afin que les sommets dynamiques ainsi que leurs voisins directs soient contractés le plus tard possible. De ce fait, lors de la modification de la carte, le nombre de raccourcis à réparer est limité et les raccourcis ajoutés au préalable ont moins de chance d'être invalidés, ce qui minimise le nombre d'actions à appliquer dans le graphe hiérarchique. Ceci permet d'accélérer le temps de réparation de la hiérarchie.
Comme le contexte est dans une carte en grille et non un graphe de route, déterminer s'il faut ajouter un raccourci ou non lors de la contraction d'un sommet suit la même démarche que celle présentée dans (Storandt, 2013), qui soutient accélérer le processus. Premièrement, si deux sommets, u et w, et le sommet à contracter, v, sont alignés sur un axe et que la somme des poids des arêtes (u, v) et (v, w) est semblable au coût d'un lien direct entre u et w, alors il n'est pas nécessaire de chercher un chemin témoin ni de créer un raccourci. Tel qu'expliqué à la section 1.5.3.2, le chemin témoin désigne un chemin entre u et w sans passer par v qui est plus court que le chemin de u vers w passant par v, soit le chemin (u, v, w). Deuxièmement, si le coût du chemin (u, v, w) correspond exactement à la différence positionnel entre u et w, il est possible de limité la recherche d'un chemin témoin à une zone plus restreinte, soit le rectangle délimité par u et w. Ceci permet d'accélérer la planification de chemin entre u et w puisque si un chemin plus court existe, il fait partie du rectangle établi.

3.3 Expériences

Le projet a été fait à l'aide de l'environnement de développement Visual Studio 2022 en utilisant la version de C++ 17. Le développement et la collecte des résultats ont été faits sur un ordinateur ayant un processeur Intel(R) Core(TM) i7-9700K CPU @ 3,6GHz avec 16 Go de mémoire principale (RAM).

Afin de déterminer les poids de chaque critère pour l'heuristique de base et l'heuristique proposée, nous avons utilisé une carte de jeu en grille simple venant de (Sturtevant, 2012) (voir figure 3.1). Sur cette carte, nous avons testé toutes les combinaisons possibles pour les poids de chaque critère. Les poids utilisés sont les suivants : 0.25, 0.3, 0.5, 0.6, 0.75, 0.9, 1.5. Cette ensemble de poids est utilisé afin de couvrir les possibilités sans surcharger l'ordinateur. Nous avons gardé la combinaison qui avait la meilleure performance moyenne entre la mise à jour du graphe hiérarchique et la recherche de chemin dans la carte de jeu. L'heu-

ristique de base qui sera utilisée pour l'expérimentation est celle de l'équation (3.1) et l'heuristique proposée qui sera utilisée est représentée par l'équation (3.2).



(a) Carte de base

(b) Carte avec les sommets dynamiques en magenta

Figure 3.1 – Carte de jeu Arena utilisée pour déterminer les valeurs les plus efficaces pour l'heuristique de base et l'heuristique proposée. Elle est de dimension 49×49, contient 2 064 sommets et 7 844 arêtes. 54 des sommets ont été marqués arbitrairement comme étant des sommets dynamiques.

$$score = ED + 1.5CoC + 1.5CN + 0.6OET \tag{3.1}$$

$$score = ED + 1.5CoC + 1.5CN + 0.6OET + 0.5SD + 0.25NVD \tag{3.2}$$

Pour chaque carte de jeu utilisée, la contraction hiérarchique a été faite deux fois, soit avec la formule heuristique de base montrée par l'équation (3.1) et avec la formule heuristique proposée contenant des critères sur le dynamisme des sommets montrés à l'équation (3.2). Par la suite, le système simule des changements dynamiques dans la carte et applique la mise à jour de la hiérarchie et des raccourcis du graphe. Pour les deux situations, les données suivantes sont évaluées :

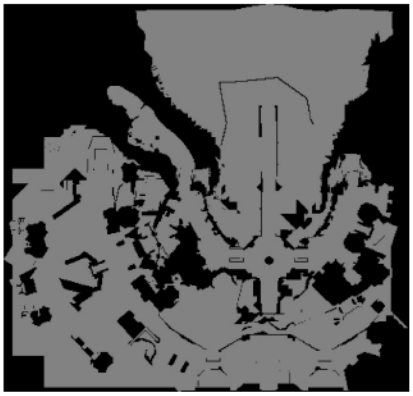
- nombre de niveaux hiérarchiques ;
- nombre de raccourcis créés ;
- temps total de prétraitement ;
- nombre de niveaux hiérarchiques après la mise à jour du graphe hiérarchique ;
- nombre de raccourcis créés après la mise à jour du graphe hiérarchique ;
- temps total de la mise à jour du graphe hiérarchique.

L'évaluation de performance de la recherche de chemin dans le graphe hiérarchique a été faite pour chaque version du graphe hiérarchique et regroupe les données suivantes :

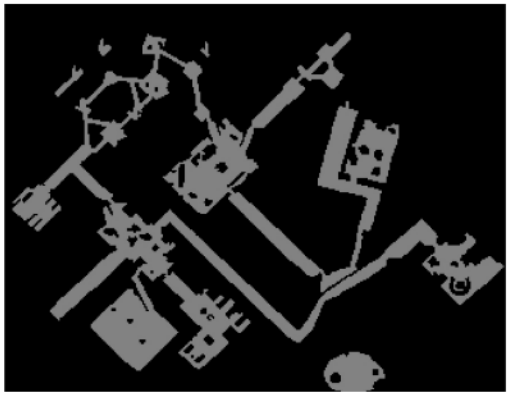
— cumulatif de temps de recherche de chemin pour toutes les requêtes ;

— cumulatif du nombre de sommets visités pour toutes les requêtes.

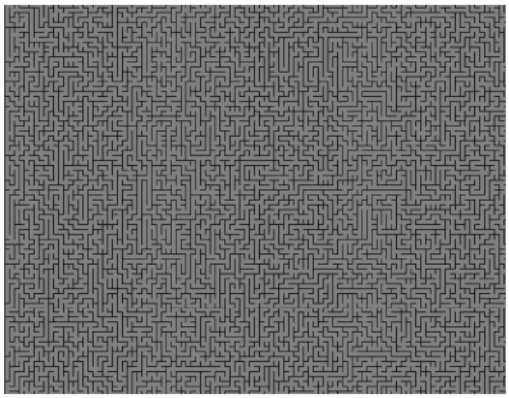
L'expérimentation et l'analyse de résultats ont été faites en utilisant quatre cartes de tailles différentes venant de Nathan Sturtevant(Sturtevant, 2012). La figure (3.2) illustre les cartes de base utilisées. Afin de simuler des changements dynamiques, nous avons marqué manuellement et de manière aléatoire plusieurs cases de la carte en tant que mur pouvant être détruit (figure 3.3), ce qui crée de nouveaux passages. Cette information est prise en considération par l'heuristique proposée. Les informations sur chaque carte d'évaluation sont détaillées dans le tableau 3.1. À noter que le nombre de sommets fait référence au nombre de sommets navigable dans la carte et non le nombre total de sommets possibles qui peut être calculé selon la dimension de la carte. Notons aussi que le nombre de sommets dynamiques est inclus dans le nombre de sommets. Le nombre d'arêtes fait référence au nombre d'arêtes présent avant tout changement dans la carte de jeu. Ceci signifie que cette valeur peut changer au cours du cycle de vie de la carte tout selon les changements dynamiques.



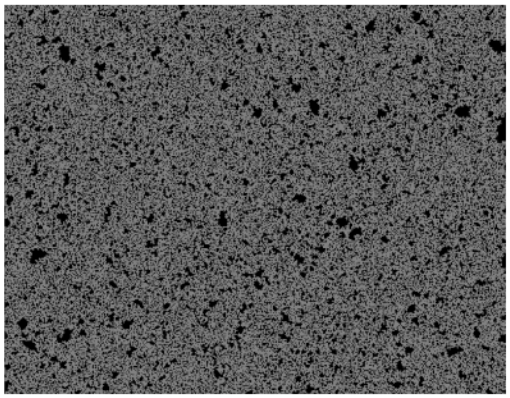
(a) Orz100d



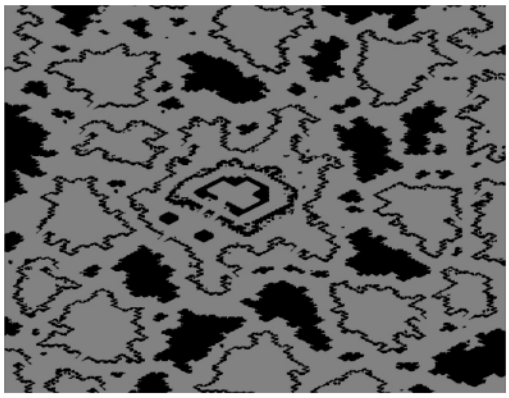
(b) AR0307SR



(c) Big_Maze

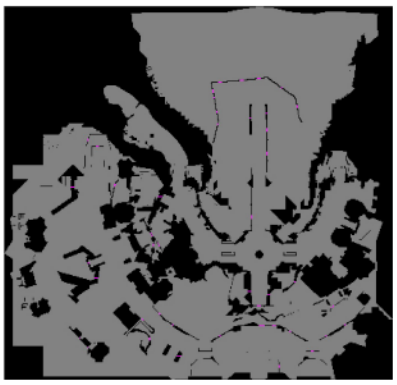


(d) Random512-35-0

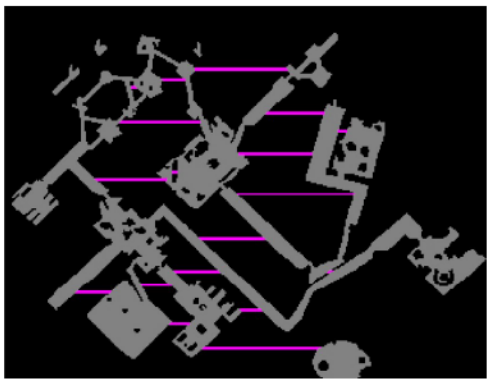


(e) TheFrozenSea

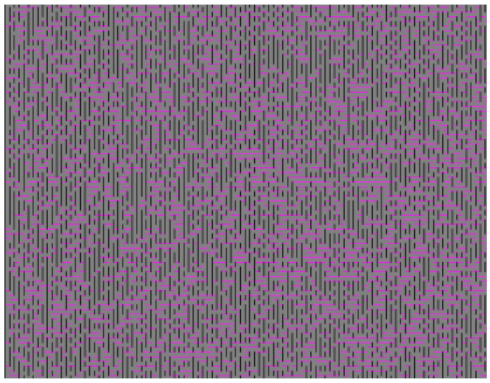
Figure 3.2 – Cartes de jeu en grille utilisées pour les expériences.



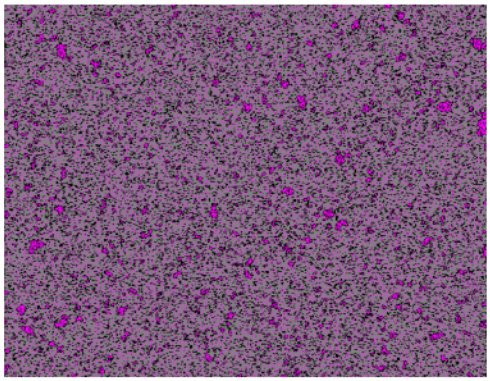
(a) Orz100d



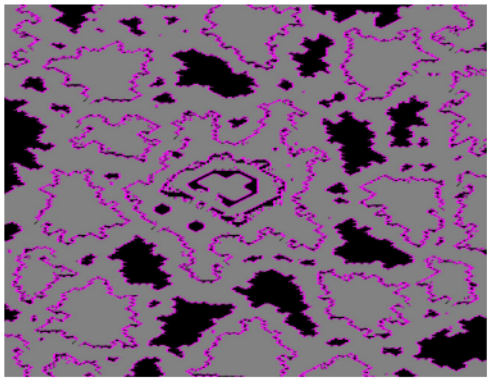
(b) AR0307SR



(c) Big_Maze



(d) Random512-35-0



(e) TheFrozenSea

Figure 3.3 – Cartes de jeu en grille avec marqueurs en magenta sur les cases représentants les obstacles pouvant être détruits.

Tableau 3.1 – Informations par carte

Carte	Dimension	Sommets	Arêtes	Sommets Dynamiques
Orz100d	492×312	99 716	383 626	134
AR0307SR	267×320	15 983	55 762	1 082
Big_Maze	512×512	229 891	820 285	20 625
Random512-35-0	512×512	197 413	614 981	35 873
TheFrozenSea	1024×1024	849 758	3 295 233	95 454

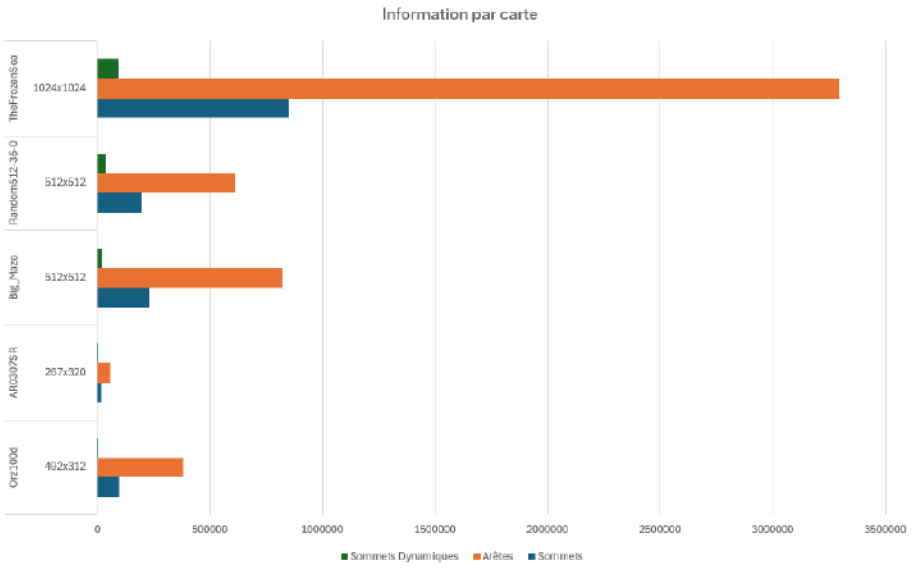


Figure 3.4 – Vue en graphe de l'information par carte

Les expériences ont été effectuées comme suit. Pour chaque carte de jeu, nous créons le graphe de navigation en gardant en mémoire les sommets qui peuvent changer dynamiquement. Dans la carte de base, tous les sommets dynamiques sont des murs. Par la suite, on choisit aléatoirement 5%, 10%, 15% et 20% des sommets dynamiques à changer et on les garde en mémoire. Ceci est fait pour qu'on puisse avoir exactement la même mise en situation entre la contraction hiérarchique avec l'heuristique de base et celle avec l'heuristique proposée. Avec ses informations, on applique la contraction hiérarchique sur le graphe de base afin d'avoir le graphe hiérarchique. Le calcul du score de priorité pour chaque sommet est présenté par les équations heuristiques (3.1) pour l'heuristique de base et (3.2) pour l'heuristique proposée. Nous détruisons 5% des murs choisis aléatoires. Ensuite, nous appliquons le changement dynamique des 10% des sommets prédéterminés. Cela peut être une destruction ou une construction d'un mur selon le sommet. La même

opération est finalement faite pour 15%, 20% et 100% des sommets prédéterminés. Chaque mise à jour d'un groupe de sommets modifie dynamiquement le graphe hiérarchique en suivant le procédé de mise à jour détaillé au sous-chapitre 1.5.3.4. Entre chaque étape de changement dynamique du graphe hiérarchique ainsi qu'à sa création, les scénarios de recherche de chemin offert par (Sturtevant, 2012) pour chaque carte sont utilisés afin de faire des recherches bidirectionnelles pour les 500 plus longues requêtes. À cause de la taille de la carte *TheFrozenSea*, la carte est divisée en sections de 256×256 afin de ne pas avoir de débordement de pile lors de la contraction hiérarchique. Ceci affecte la performance de la contraction de cette carte, qui est plus rapide que des cartes plus petites malgré son nombre élevé de sommets comme nous le voyons dans le graphe 3.4.

3.4 Résultats et analyse

On peut constater dans le tableau 3.3 que la contraction hiérarchique des cartes de départ de petites et moyennes tailles est légèrement plus rapide en utilisant l'heuristique proposée et, dans certains cas, à la même vitesse. Il y a aussi moins de raccourcis créés avec notre proposition. Nous pouvons supposer que ceci est causé par l'ordre de priorité des sommets qui place les sommets dynamiques vers la fin de la liste, ce qui fait en sorte que moins de raccourcis sont nécessaires lors de la contraction. En effet, si un sommet dynamique est contracté plus tard dans le processus de prétraitement et que celui-ci est connecté à plusieurs sommets, comme une entrée (porte ou tunnel), il fait plus souvent partie des chemins témoin que de nouveaux raccourcis. Nous pouvons aussi constater que, puisqu'il y a moins de raccourcis créés, nous économisons aussi de l'espace en mémoire. En revanche, pour les grosses cartes avec beaucoup de sommets dynamiques, le temps de contraction et le nombre de raccourcis sont beaucoup plus élevés avec la proposition. Ceci est peut-être dû au fait qu'avec un nombre très élevé de sommets dynamiques, les critères ajoutés par la proposition n'ont pas autant d'impact et d'influence. Il faut aussi noter que le nombre de sommets par carte vu dans le tableau 3.1 est identique au nombre de niveaux par carte vu dans le tableau 3.3 car tous les sommets ont été contractés et qu'un sommet correspond à un niveau dans la hiérarchie du graphe final. Le tableau 3.4 démontre que la mise à jour du graphe hiérarchique est en moyenne égale entre l'heuristique proposée qu'avec l'heuristique de base, avec quelques cas où l'heuristique de base est plus rapide. Nous pouvons aussi voir qu'avec l'ordre des sommets dans la hiérarchie du graphe, le nombre de sommets à contracter à chaque mise à jour est plus élevé avec l'heuristique de contraction hiérarchique proposée. Bien que la proposition crée moins de raccourcis, la modification du graphe hiérarchique lors de l'activation des sommets prolonge le temps de calcul, certainement à cause de l'ajout de raccourcis nécessaire pour

maintenir l'optimalité du graphe hiérarchique. Pour chaque raccourci à créer, il faut s'assurer qu'il n'y a pas un chemin témoin déjà existant. Par contre, comme on peut le constater dans le tableau 3.5, la recherche de chemin est visiblement plus efficace dans le graphe hiérarchique utilisant l'heuristique proposée. Ceci est provoqué par le fait qu'il y a moins de raccourcis au total, donc un facteur de branchement lors de la recherche bidirectionnelle moins élevée.

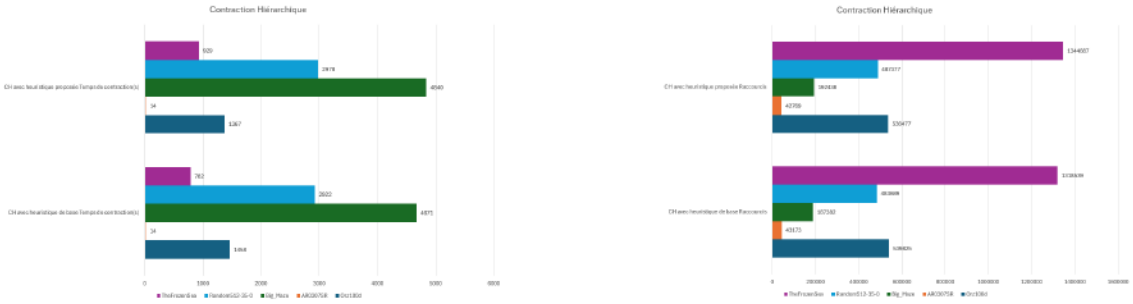
L'avantage de l'heuristique proposée est donc surtout sur l'efficacité de la création du graphe hiérarchique et de la planification de chemin dans celui-ci sur de petites et moyennes cartes ou avec un nombre limité de sommets dynamiques. Il est moins efficace en moyenne pour la modification dynamique, mais la différence du temps de calcul et l'espace en mémoire gagné par la différence de raccourcis créés peut être acceptable selon le cas.

Tableau 3.2 – Planification de chemin utilisant l'algorithme A* dans le graphe sans contraction hiérarchique

Carte	Sommets explorés en moyenne	Temps en moyenne(ms)
Orz100d	63 588,80	13 020 300
AR0307SR	10 263,90	333 952
Big_Maze	194 537,00	58 299 800
Random512-35-0	44 651,10	4 504 050
TheFrozenSea	588 068,00	529 016 000

Tableau 3.3 – Contractions hiérarchiques sur les cartes de départ

Carte	Niveaux	CH avec heuristique de base		CH avec heuristique proposée	
		Raccourcis	Temps de contraction(s)	Raccourcis	Temps de contraction(s)
Orz100d	99 716	539 825	1 458	536 477	1 367
AR0307SR	15 983	43 173	14	42 769	14
Big_Maze	229 891	187 382	4 671	192 438	4 840
Random512-35-0	197 413	483 689	2 922	487 377	2 979
TheFrozenSea	849 758	1 318 539	782	1 344 687	929



(a) Comparaison du temps de contraction en secondes.

(b) Comparaison du nombre de racourcis.

Figure 3.5 – Vue en graphe de la contraction hiérarchique sur les cartes de départ.

Tableau 3.4 – Ajustements de graphes hiérarchiques après des changements dynamiques

Carte	CH avec heuristique de base			CH avec heuristique proposée		
	Raccourcis	Temps d'ajustement (s)	Nombre de sommets à contracter	Raccourcis	Temps d'ajustement (s)	Nombre de sommets à contracter
Orz100d 5%	539 861	0,16	16	536 513	0,18	17
Orz100d 10%	539 860	0,63	23	536 511	0,66	24
Orz100d 15%	539 912	0,66	28	536 563	0,66	32
Orz100d 20%	539 947	168,19	27	536 601	168,67	30
Orz100d 100%	540 094	5,85	177	536 761	6,13	150
AR0307SR 5%	43 175	1,95	120	42 773	2,56	121
AR0307SR 10%	43 189	4,22	250	42 788	9,51	249
AR0307SR 15%	43 231	3,08	302	42 833	3,84	302
AR0307SR 20%	43 290	2,31	367	42 895	2,08	370
AR0307SR 100%	43 186	17,21	798	42 796	17,32	802
Big_Maze 5%	193 353	9,89	2 286	198 364	10,09	2 430
Big_Maze 10%	203 190	24,68	4 330	208 174	25,78	4 623
Random512-35-0 5%	486 625	5 792,68	2 929	490 447	7 424,88	2 921
Random512-35-0 10%	491 814	7 647,75	5 388	496 024	11 017,50	5 413
TheFrozenSea 5%	1 319 409	1 781,85	7 689	1 346 175	2 013,36	8 117
TheFrozenSea 10%	1 320 916	2 561,99	11 166	1 348 514	2 628,10	11 662

Tableau 3.5 – Planification de chemin selon les graphes hiérarchiques

Carte	CH avec heuristique de base		CH avec heuristique proposée	
	Sommets explorés en moyenne	Temps de planification en moyenne(ms)	Sommets explorés en moyenne	Temps de planification en moyenne(ms)
Orz100d 0%	1 539,42	27 346	1 432,54	26 278
Orz100d 5%	1 539,42	27 368	1 432,54	26 189
Orz100d 10%	1 539,42	27 239	1 432,54	26 019
Orz100d 15%	1 539,42	27 355	1 432,54	25 928
Orz100d 20%	1 540,17	27 487	1 433,45	25 976
Orz100d 100%	1 540,70	27 310	1 433,78	25 986
AR0307SR 0%	148,47	413	146,86	404
AR0307SR 5%	148,47	416	146,91	403
AR0307SR 10%	148,47	406	146,91	392
AR0307SR 15%	148,56	403	146,95	394
AR0307SR 20%	148,57	407	146,96	390
AR0307SR 100%	148,47	405	147,22	390
Big_Maze 0%	43,88	104	45,08	105
Big_Maze 5%	45,40	99	46,52	101
Big_Maze 10%	47,34	103	48,69	108
Random512-35-0 0%	1765,88	24 522	1 768,55	24 388
Random512-35-0 5%	1766,52	24 642	1 769,39	24 403
Random512-35-0 10%	1768,47	24 515	1 770,89	24 345
TheFrozenSea 0%	227,64	667	258,51	764
TheFrozenSea 5%	227,64	642	258,51	810
TheFrozenSea 10%	227,92	674	258,53	766

3.5 Limites

Cette technique est limitée par la nécessité d'avoir un concepteur de niveaux devant faire une liste de murs pouvant être changés au préalable. Ceci empêche d'avoir des situations entièrement représentatives de la réalité d'un jeu, ce qui peut impliquer trop ou pas assez de sommets dynamiques identifiés. En effet, dans le contexte de ces expériences, les sommets modifiables ont été choisis arbitrairement et les changements appliqués dans ceux-ci ont été fait aléatoirement afin de simuler le dynamisme.

Une autre limite observée est que, dans le cas de cartes à grille de grande taille, c'est-à-dire comportant un grand nombre de sommets et d'arêtes, comme les cartes *Big_Maze* et *Random512-35-0* (voir la figure 3.2), le temps de contraction de la carte est trop long pour un jeu, voir le tableau 3.3 et la figure 3.5. Par contre, si les développeurs du jeu sont prêts a créer des sous-sections de la grilles de jeu, tel que fait avec la carte *TheFrozenSea*, il est possible de sauver du temps de contraction. Nous pouvons constater en regardant la figure 3.5a que la contraction de départ du graphe hiérarchique est plus rapide pour la carte *TheFrozenSea*, qui a été découpé en sous-sections, que les cartes *Big_Maze* et *Random512-35-0*, qui ont été traitées en entier.

CONCLUSION

Notre proposition est une alternative de l'heuristique utilisée dans le calcul du score de chaque sommet pour l'ordre de priorité fait lors de la contraction hiérarchique d'une carte de jeu. Elle offre deux nouveaux critères à considérer à chaque sommet, soit si le sommet courant est dynamique (SD) et le nombre de voisins directs qui est un sommet dynamique (NVD). La nouvelle formule heuristique est celle démontrée par l'équation (2.1). Cette proposition se base sur l'hypothèse que les sommets dynamiques dans la carte de jeu sont marqués au préalable par un concepteur de niveaux.

En comparant les planifications de chemin utilisant le graphe hiérarchique et celle de l'algorithme A* classique, nous avons conclu que, dans le contexte d'une carte de jeu en grille 2D la planification de chemin dans le graphe hiérarchique est grandement plus efficace en temps et en mémoire, car elle diminue le facteur de branchement grâce à ses raccourcis et sa limitation de voisins en bloquant les sommets de niveaux inférieurs.

Avec les données de comparaison de la contraction hiérarchique utilisant l'heuristique de notre proposition et l'heuristique de base comme vue par l'équation (1.4), nous avons constaté que pour de grosses cartes de jeu 2D ayant un nombre élevé de sommets identifiés comme étant dynamique, le temps de la création du graphe hiérarchique avec notre proposition est plus long et ajoute plus de raccourcis, comme vus pour les cartes *Big_Maze* et *Random512-35-0* dans le tableau 3.3. Le graphe hiérarchique étant plus volumineux provoque un temps de réparation lors d'un changement dynamique beaucoup plus long et une planification de chemin nécessitant plus de sommets à parcourir, comme démontré aux tableaux 3.4 et 3.5 pour les cartes *Big_Maze* et *Random512-35-0*.

En revanche, dans le cas de cartes de jeu de plus petites dimensions avec un nombre de sommets dynamiques moindre, la contraction hiérarchique utilisant l'heuristique de notre proposition est plus efficace à défaut de quelques cas. En effet, le graphe hiérarchique résultant contenait moins de raccourcis et se terminait plus rapidement (voir les cartes *Orz100d* et *AR0307SR* dans le tableau 3.3). Pour la réparation du graphe hiérarchique après un changement dynamique, comme illustré dans le tableau 3.4, le temps de réparation est plus élevé parce qu'il y a plus de raccourcis à réparer. Pour la planification de chemin, parce qu'il y a en tout moins de raccourcis dans le graphe hiérarchique de notre proposition, le temps de calcul et le nombre de sommets parcouru sont moins élevés (voir le tableau 3.5). Donc, si un jeu ne contient pas un très grand

nombre de sommets dynamiques et est prêt à sacrifier un peu de performance lors de la réparation du graphe hiérarchique, notre proposition permet une recherche de chemin plus efficace.

Afin de contrer les limitations mentionnées au sous-chapitre 3.5, plusieurs pistes de solutions peuvent être explorées. Pour contrer la nécessité d'avoir un concepteur de niveaux devant identifier manuellement les sommets dynamiques dans la carte de jeu, il est possible de créer un système qui, pour une carte de jeu donnée, détermine les murs stratégiques à détruire dans la carte d'après un ensemble de requêtes de planification chemin, permettant ainsi d'avoir une probabilité de dynamisme sur chaque sommet au lieu d'un indicateur binaire de dynamisme. Pour régler le souci d'une carte trop volumineuse pour la contraction hiérarchique, il est possible d'avoir un système faisant un découpage de la carte en sous-groupes afin d'y appliquer la contraction hiérarchique parallèlement. Ce faisant, il est ensuite possible de reconnecter les sous-groupes en suivant un procédé semblable au HPA* (voir le sous-chapitre 1.5.1) en ayant un graphe abstrait de navigation d'un sous-groupe à l'autre afin de couvrir les cas où la planification de chemin a besoin de changer de sous-groupes. Le changement dynamique de la carte se fait donc sur une sous listes de sommets, accélérant le processus.

ANNEXE A

PROJET D'ÉVALUATION

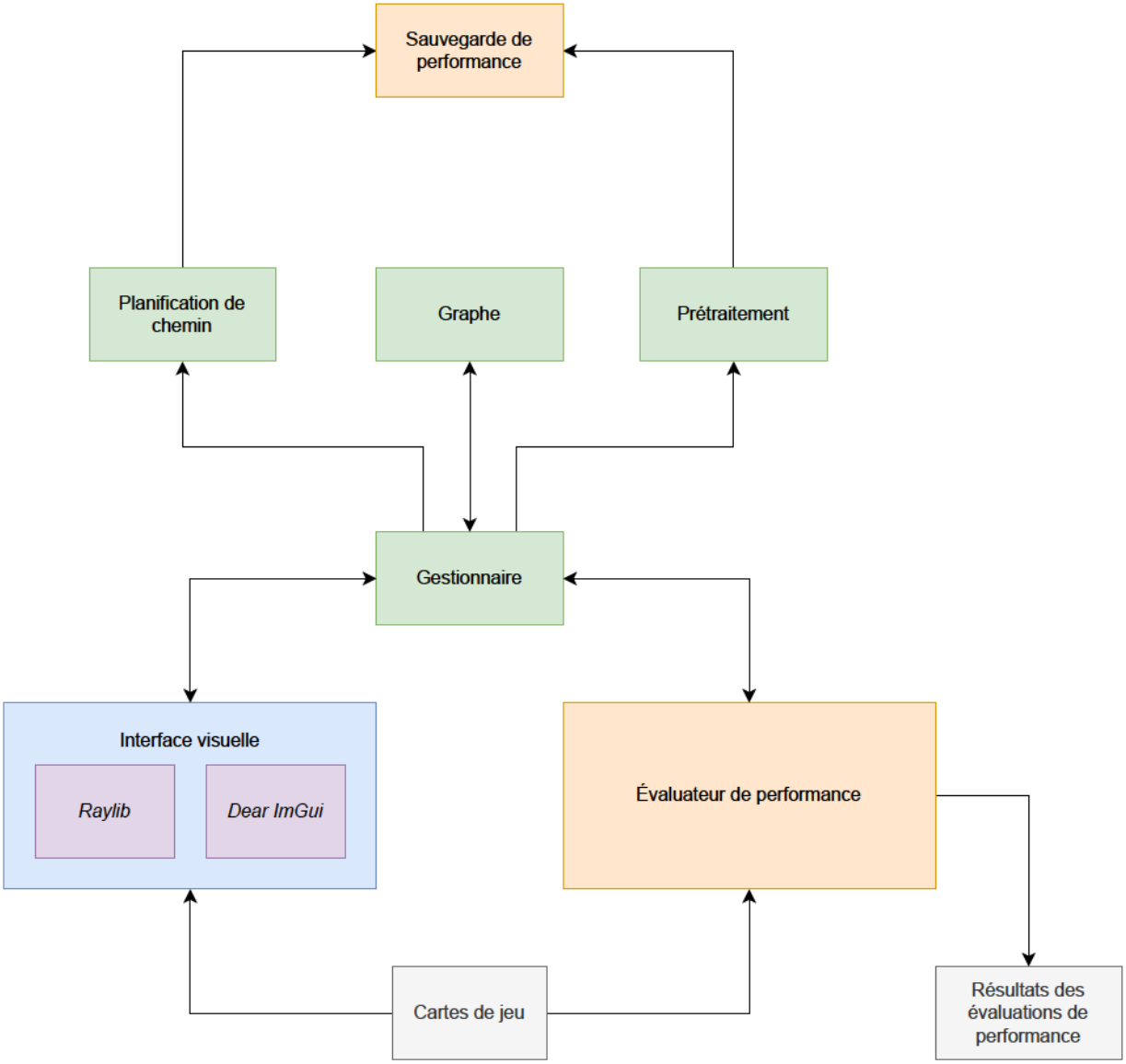


Figure A.1 – Architecture du projet d'évaluation

BIBLIOGRAPHIE

Antsfeld, L., Harabor, D., Kilby, P. et Walsh, T. (2012). TRANSIT Routing on Video Game Maps. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, *8*(1), 2–7. http://dx.doi.org/10.1609/aiide.v8i1.12509

Barr, A. (1981). *Handbook of Artificial Intelligence Volume 1*. Kaufmann, William Inc.

Bast, H., Funke, S., Matijevic, D., Demetrescu, C., Goldberg, A. V. et Johnson, D. S. (2006). TRANSIT : Ultrafast Shortest-Path Queries with Linear-Time Preprocessing. Récupéré de https://api.semanticscholar.org/CorpusID:1838728

Bellman, R. (1958). On a routing problem. *Quarterly of Applied Mathematics*, *16*(1), 87–90. http://dx.doi.org/10.1090/qam/102435

Botea, A., Müller, M. et Schaeffer, J. (2004). Near Optimal Hierarchical Path-Finding. http://dx.doi.org/10.1.1.112.314

Buckland, M. (2004). *Programming Game AI by Example*. Wordware Publishing, Inc.

Chong, L., Jian, L. et XueQuan, L. (2022). Static Rectangle Expansion A* Algorithm for Pathfinding. *IEEE Transactions on Games*, *14*(1), 23–35. http://dx.doi.org/10.1109/tg.2020.3012602

Cormen, T. H., Leiserson, C. E., Rivest, R. L. et Stein, C. (2009). *Introduction to Algorithms*. The MIT Press.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. http://dx.doi.org/10.1145/3544585.3544600. Récupéré de https://www.semanticscholar.org/paper/45786063578e814444b8247028970758bbbd0488

Ebendt, R. et Drechsler, R. (2009). Weighted A* Search - Unifying View and Application. *Artificial Intelligence*, *173*(14), 1310–1342. http://dx.doi.org/10.1016/j.artint.2009.06.004

Floyd, R. W. (1962). Algorithm 97 : Shortest path. *Communications of the ACM*, *5*(6), 345–345.

Geisberger, R., Sanders, P., Schultes, D. et Vetter, C. (2012). Exact Routing in Large Road Networks Using Contraction Hirerarchies. volume 46 388–404.

Ghallab, M., Nau, D. et Traverso, P. (2016). *Automated Planning and Acting*. Cambridge University Press.

Hart, P., Nilsson, N. et Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*, 100–107. http://dx.doi.org/10.1109/tssc.1968.300136

Islam, F., Narayanan, V. et Likhachev, M. (2015). Dynamic Multi-Heuristic A. 2376–2382., Seattle, WA, USA. IEEE. http://dx.doi.org/10.1109/ICRA.2015.7139515

Koenig, S. et Likhachev, M. (2002). D* Lite. http://dx.doi.org/10.1.1.18.7250

Koenig, S., Likhachev, M. et Furcy, D. (2004). Lifelong Planning A*. *155*, 93–146.

http://dx.doi.org/10.1016/j.artint.2003.12.001

Likhachev, M., Ferguson, D., Gordon, G., Stentz, A. et Thrun, S. (2005). Anytime Dynamic A* : An Anytime Replanning Algorithm. http://dx.doi.org/10.1.1.324.2316

Likhachev, M., Gordon, G. J. et Thrun, S. (2003). ARA* : Anytime A* with Provable Bounds on Sub-Optimality. Dans S. Thrun, L. K. Saul, et B. Schölkopf (dir.). *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003, December 8-13, 2003, Vancouver and Whistler, British Columbia, Canada]*, 767–774. MIT Press. Récupéré de https://proceedings.neurips.cc/paper/2003/hash/ee8fe9093fbbb687bef15a38facc44d2-Abstract.html

Pohl, I. (1970). Heuristic Search Viewed as Path Finding in a Graph. http://dx.doi.org/10.1016/0004-3702(70)90007-X. Récupéré de https://www.semanticscholar.org/paper/bba5a510c44fe6679249d9a939cbc0bc4ba9f658

Pohl, I. (1973). The Avoidance of (Relative) Catastrophe, Heuristic Competence, Genuine Dynamic Weighting and Computational Issues in Heuristic Problem Solving. Récupéré de https://www.semanticscholar.org/paper/62e27eca44ba6a533a8e306233143e56cf913553

Rabin, S. (dir.) (2007). *AI game programming wisdom* ([nachdr.] éd.). Hingham, Mass. : Charles River Media.

Russell, S. et Norvig, P. (2009). *Artificial Intelligence : A Modern Approach* (4th éd.).

Stentz, A. (1993). Optimal and Efficient Path Planning for Unknown and Dynamic Environments. http://dx.doi.org/10.1.1.15.3683

Stentz, A. (1995). The Focussed D* Algorithm for Real-Time Replanning. http://dx.doi.org/10.1.1.26.6615

Storandt, S. (2013). Contraction hierarchies on grid graphs. http://dx.doi.org/10.1007/978-3-642-40942-4_21

Sturtevant, N. (2012). Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games*, *4*(2), 144 – 148. Récupéré de http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf

Yang, F., Cao, C., Zhu, H., Oh, J. et Zhang, J. (2022). FAR Planner : Fast, Attemptable Route Planner using Dynamic Visibility Update