Dorian Essamir

Le pathfinding A* et l'optimisation pour des grilles 2D

Master « jeux et médias interactifs numériques »
cohabilité par le Conservatoire national des arts et métiers,
l'Université de Poitiers et l'Université de La Rochelle

le cnam
enjmin

Campus Image Magelis Angoulême

UNIVERSITÉ La Rochelle

1431 Université de Poitiers

Ce mémoire a pour but d'expliquer le fonctionnement de l'algorithme de pathfinding A* pour les grilles 2D, très utilisées dans les jeux vidéos, et d'approfondir le sujet en expliquant quelques problèmes que peut rencontrer cet algorithme et quelles méthodes ont été trouvées pour y remédier. L'exposé commencera par un rappel du fonctionnement de l'algorithme pour une grille 2d, puis débattra des structures de données les plus adaptées pour l'application de l'algorithme, le document parle ensuite de deux algorithmes qui permettent d'améliorer nettement les performances du A*, le HPA* et le JPS.

le A* pour des grilles uniformes

Différentes choses sont nécessaires pour cet algorithme:

- un graphe représentant l'ensemble des chemins utilisables sur la map, dans le cas d'une grille 2d, chaque case de la grille représente un node du graphe et chaque déplacement s'effectue d'une case a la case adjacente sur la grille et coûte 1, diagonale ou ligne droite.

- deux listes de nodes :

◦ la liste ouverte est une liste ou on enregistre tous les nodes sur lesquels on n’est pas encore allé.

◦ la liste fermée, ou on enregistre tous les nodes déjà vérifiés et dont on n'a plus besoin de se soucier.

- trois valeurs calculées pour chaque node:

◦ G : c'est la distance parcourue depuis le départ jusqu'au node actuel en suivant le chemin généré pour y arriver, chaque déplacement coûtant 1, c'est simplement le nombre de cases parcourues depuis le départ dans le cas d'une grille 2d

◦ H: c'est la distance estimée qu'il faudra parcourir depuis le node actuel jusqu'à l'arrivée, on peut utiliser plusieurs formules pour calculer cette distance, selon la représentation qu'on possède de la map, dans le cas d'une grille 2d uniforme, la distance de Manhattan est tout ce dont on a besoin, elle se calcule en additionnant la distance en X et la distance en Y entre deux cases de la grille. Ce calcul de distance nous donne donc une distance plus grande que la distance réelle a parcourir étant donné que les déplacements en diagonale sont autorisés, et par conséquent cette valeur diminuera plus vite que la distance parcourue n'augmentera, mais ce n'est pas un problème, et la distance de Manhattan est plus rapide en temps de calcul que beaucoup d'autres formules.

◦ F: l'addition de G et H, c'est le score du node qui est utilisé pour déterminer quel chemin est le plus rapide.

La première chose qu'on fait est d'ajouter le node de départ à la liste ouverte.

Première Itération:

Maintenant, depuis le node de départ, qui est le node sur lequel on se tient actuellement (node actuel), on prend tous les nodes à portée (les 8 cases autour de la case ou on est actuellement, sauf si ces cases sont des murs, ou hors limite du tableau, ou le parent du node actuel) et on calcule pour chacun les valeurs F, G et H expliquées ci-dessus. On ajoute tous ces nodes à la liste ouverte et pour chacun de ces nodes, on enregistre le node actuel en tant que “parent”.

On met le node actuel dans la liste fermée et on l’enlève de la liste ouverte, puis on choisit le node de la liste ouverte avec le score F le plus petit, ce node est le nouveau node actuel.

Deuxième Itération:

Depuis le nouveau node actuel, on répète l'opération en testant tous les nodes à portée du node actuel, si un de ces nodes est déjà dans la liste ouverte, on vérifie si le chemin depuis notre node actuel est meilleur que celui testé auparavant, on vérifie donc que la valeur G qu'on calcule pour ce node depuis notre node actuel est inférieure à celle qu'il avait déjà, si c'est le cas, on change ses valeurs F et G par les nouvelles qu'on vient de calculer, et on change son parent pour le node actuel, sinon on ne fait rien.

Pour la première itération, le calcul de la valeur G était simple, il suffisait de calculer la distance entre le node actuel qui était celui de départ, et les nodes à portée, pour les itérations suivantes, la valeur G des nodes à portée sera calculée en additionnant le G du node actuel à la distance entre le node actuel et le node à portée, G est donc la distance parcourue entre chaque nodes depuis le point de départ et non pas la distance en ligne droite depuis le départ.

A partir de là il suffit d’enchaîner les itérations jusqu'à ce que le node d'arrivée soit inclus dans la liste fermée, à ce moment-là, il suffit de remonter les parents de chaque node depuis le node d'arrivée, ce qui nous fait arriver au node de départ, et nous avons notre chemin.

Structures de données

Pour la liste fermée, beaucoup utilisent un tableau 2d correspondant a la taille de la map, dans lequel on change la valeur des cases par une valeur prédéfinie qui correspond a l’état du node correspondant, par exemple 1 signifie que le node est en liste ouverte, 2 liste fermée. Pour éviter d'avoir réinitialiser le tableau a chaque nouvel appel de la fonction de pathfinding, on incrémente les deux valeurs a chaque nouvelle utilisation (3 pour liste ouverte et 4 pour fermée, puis 5 et 6 etc etc...), ainsi les anciennes valeurs déjà inscrites dans le tableau n'ont plus aucune influence sur le test, si la valeur de la case est inférieure a la valeur définie pour liste ouverte, le node correspondant a cette case n'a pas encore été testé, sinon on vérifie si le node est en liste ouverte ou fermée...

Une des opérations les plus lentes de l'algorithme de pathfinding A* est de trouver le node avec le plus petit F dans la liste ouverte. En effet selon la taille de la map on peut se retrouver avec des listes ouvertes de plusieurs centaines de nodes, voire plusieurs milliers, et aller chercher le node avec le plus petit F dans cette liste simplement en parcourant un tableau et en gardant le plus petit qu'on a trouvé a la fin peut se révéler très lent a chaque itération de l'algorithme, même si l'ajout d'un nouveau node ne sera jamais plus rapide.

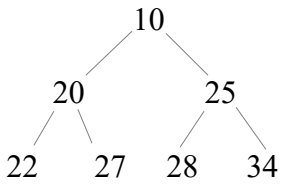
Pour accélérer un peu le calcul, il est préférable de garder sa liste ouverte bien triée, l'ajout d'un nouveau node a la liste sera plus lent, mais la récupération du plus petit F sera instantanée, il suffit de récupérer le premier node de la liste. Il existe différents algorithmes pour trier une liste de données, comme le quick sort, qui consiste a commencer par comparer le nouveau node a celui du milieu de la liste, si il est inférieur au milieu, on le compare au node a ¼ de la liste et ainsi de suite jusqu'à trouver sa place dans la liste. Il existe cependant une meilleure méthode.

Les binary heaps

Trier l'intégralité de sa liste ouverte simplement pour toujours avoir le node avec le plus petit F sous la main semble quand être un peu trop, c'est la que les binary heaps entrent en jeu.

Les binary heaps sont des arbres binaires qui ont comme propriété spéciale de toujours avoir toutes leurs branches remplies, sauf éventuellement les toutes dernières. Un binary heap aura toujours l'objet avec la valeur la plus basse (ou la plus haute selon le tri que l'on veut) au sommet, ses deux enfants seront tous les deux des objets de valeur supérieure ou égale, et chacun de ces enfants aura deux enfants de valeur supérieure ou égale a eux, et ainsi de suite...

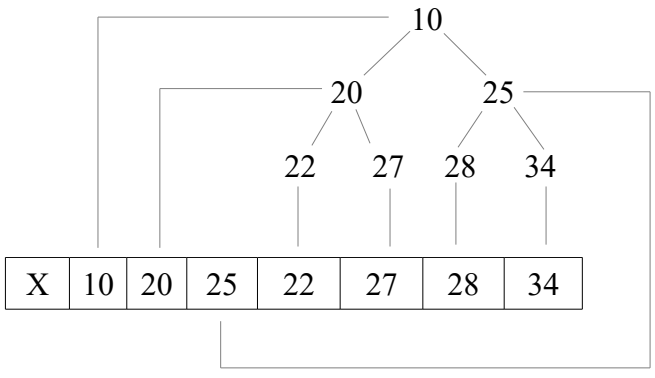
Exemple de binary heap :



Dans cet exemple, comme dans tous les binary heaps, l'objet le moins cher est en haut de l'arbre, et le deuxième moins cher est un de ses deux enfants, après ça, tous les autres objets n'ont aucune importance, par exemple dans le binary heap ci dessus, le deuxième enfant de 10 est 25 alors qu'il y a un 22 (moins cher) dans les enfant de 20, plus bas dans la hiérarchie, c'est tout a fait normal dans un binary heap.

Maintenant la vraie question est : comment utiliser le binary heap pour une liste de valeurs en pathfinding, et c'est ici que ça devient intéressant, les binary heaps peuvent être stockés dans de simples tableaux a une dimension. Dans le tableau ci dessous, on stocke la premier objet du heap en position 1 du tableau (pas en position 0, c'est possible de mettre des objets en position 0 d'un tableau, mais il faut commencer en position 1), ses deux enfants sont en position 2 et 3, et les quatre enfants de ces derniers en position 4 a 7, et ainsi de suite.

tableau final :



Dans cette configuration, les enfant de n'importe quel objet dans le heap peuvent être trouvés en multipliant la position du parent par deux pour trouver la position du premier enfant, et le second enfant est la position du premier plus un, par exemple les enfants de 10 qui est en position 1 sont en position 1x2 = 2 et 1x2+1 = 3, les enfants de 25 qui est en position 3 sont en position 3*2 = 6 et 3*2+1 = 7

Il nous faut maintenant pouvoir ajouter et enlever des objets dans le heap :

Ajouter un objet :

Pour ajouter un objet au heap, on place l'objet a la fin du tableau, juste après le dernier objet déjà existant, puis on trouve son parent, qui est a la position de l'objet dans le tableau divisée par 2, arrondi a l'entier inférieur, on compare le parent au nouvel objet, si le parent a une valeur supérieure a celle de l'enfant, on échange leur place, sinon on laisse le nouvel objet la ou il est, si il y a eu échange, on compare le nouvel objet a son nouveau parent, qui se trouve encore a la place de l'objet divisée par deux et arrondie a l'inférieur (la nouvelle place, celle de son ancien parent), et ainsi de suite jusqu'à ce que le test de supériorité soit faux ou que le nouvel objet soit a la première place dans le tableau.

Enlever un objet :

Lorsqu'on parle d'enlever un objet dans le heap pour le pathfinding, on veu bien sur enlever seulement le premier objet dans le tableau, celui tout en haut du heap. On l'enlève donc, et il faut ensuite réorganiser le heap sans cet objet, pour ce faire, on prend le dernier objet du heap, on le place en première position, on le compare ensuite a ses deux enfants (qui sont a la position courante de l'objet *2 et position courante *2 +1, donc 2 et 3 pour la première itération), si le dernier objet est plus petit que ses deux enfants, il reste ou il est, sinon, on l'échange avec son enfant le plus petit, on répète ensuite cette opération avec les nouveaux enfants de l'ancien dernier objet jusqu'à ce qu'il trouve sa place dans le heap, ou qu'il n'ait plus d'enfants, ce qui signifiera que le fond du heap a été atteint, et on arrête le tri.

Il existe un dernier cas qui n'a pas encore été traité, on peut parfois arriver dans le cas ou un objet est déjà dans la liste ouverte, mais que son F doit changer car il est maintenant inférieur, il faut alors retrouver l'objet dans le tableau et modifier son F, puis appliquer le même processus que pour l'ajoute d'un objet dans le heap, on le compare a son parent, si il est inférieur on échange de place et ainsi de suite.

L'utilisation d'un Binary Heap comme structure de données pour la liste ouverte en pathfinding A* permet de multiplier la vitesse de l'algorithme par 2 ou 3 et plus sur des maps plus grandes (100 * 100 et plus). C'est une amélioration non négligeable pour une solution relativement simple a mettre en place, et c'est donc la solution que j'ai choisi de présenter dans ce mémoire.

Hiérarchie et pathfinding, le HPA*

Lorsqu'on utilise le A* en jeux vidéos, on peut s'apercevoir de plusieurs problèmes dans l'algorithme, tout d'abord le nombre de nodes « étendus » (les nodes testés et ajoutés a la liste ouverte) est parfois très élevé, et c'est de pire en pire a mesure que la taille de la map augmente, et ensuite, l'algorithme se déroule sur une seule frame et par conséquent, si le calcul du chemin dure trop longtemps, on aura droit a un freeze le temps que le calcul se termine, a moins d'arrêter le calcul en cours si on dépasse un temps imparti, et de recommencer la ou on s'était arrêté a la frame suivante, ou simplement arrêter le calcul (c'est ce qui se passait dans Warcraft 2 quand on disait a une unité d'aller trop loin d'un coup, elle allait ailleurs, parfois dans un cul de sac qui se dirigeait vaguement vers l'objectif, mais s'arrêtait avant, c'est parce que le pathfinding s'était arrêté avant d'arriver au bout de ce cul de sac...).

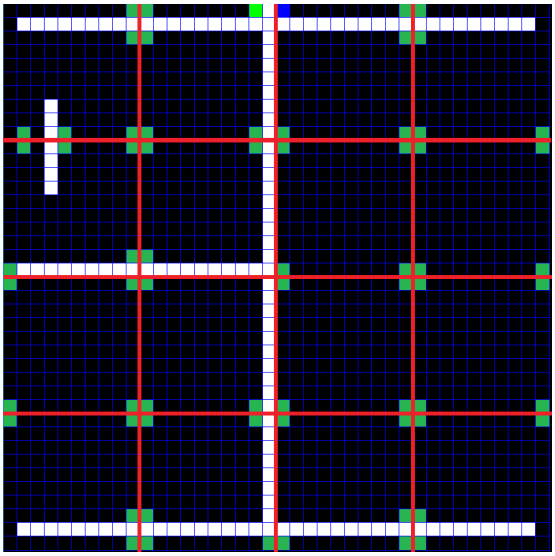
Pour remédier a ces problèmes, il y a le HPA* (Hierarchical Pathfinding A*), le principe de

cet algorithme consiste a utiliser des niveaux d'abstractions sur la map pour accélérer la recherche (selon le principe du divide and conquer), on sépare donc la grille en un ensemble de sous grilles, qui contiennent chacune plusieurs cases de la grille d'origine, et on calcule le chemin du départ a l'arrivée en utilisant ces sous grilles (qui sont en fait les nodes du niveau d'abstraction) puis on calcule dans chaque sous grille le chemin vers la sous grille suivante.

Comment procéder : deux étapes, le pre-processing, puis le calcul de chemin

Le pre-processing est exécuté une seule fois, pendant le chargement de la map, on y calcule toutes les données nécessaires a l'exécution du pathfinding HPA* et on les enregistre dans un graphe abstrait.

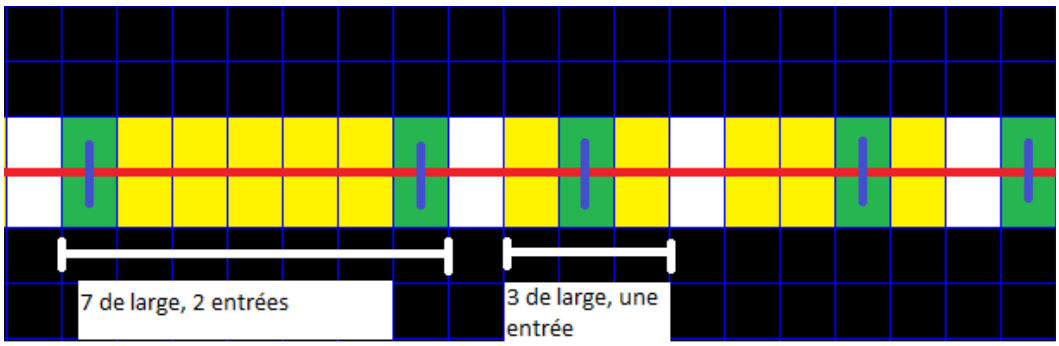
Dans un premier temps, il faut diviser notre grille, pour cela il suffit de couper la grille a intervalles réguliers en carrés de plus petite taille, dans la figure ci dessous, on découpe la map en carrés de 10 par 10 cases (les traits rouges) qu'on appelera des clusters.



Une fois la découpe faite, chaque cluster de 10 par 10 cases devient un nouveau node dans le niveau d'abstraction supérieur, et c'est sur ces nodes la que nous lanceront l'algorithme de pathfinding. Mais comment ces nodes sont ils reliés entre eux ?

On relie chaque node aux nodes adjacents (les clusters adjacent au cluster en cours) en précalculant pour chaque cluster une liste d'entrée/sorties vers les clusters adjacents, les entrées/sorties sont déterminées selon certaines règles :

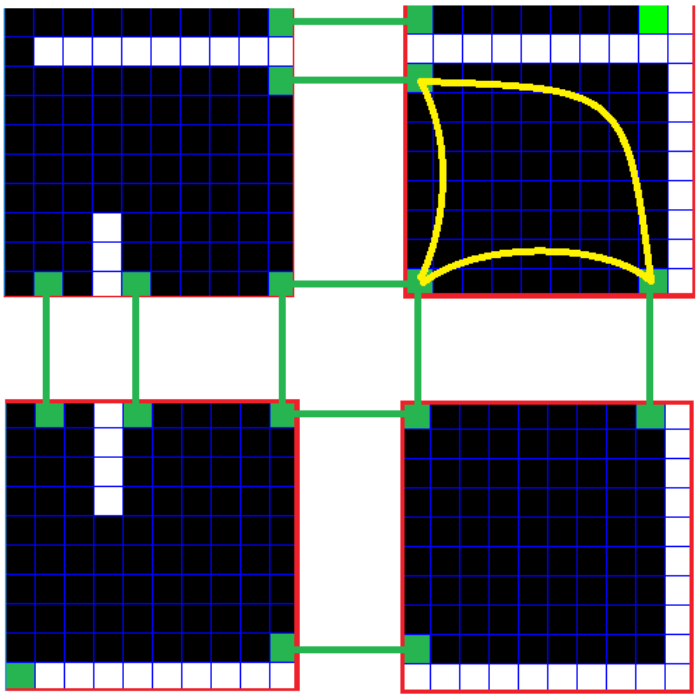
- une entrée/sortie se situe forcément au bord d'un cluster, a la frontière avec un autre cluster.
- Si une case au bord d'une frontière est vide et que la case symétrique a celle ci par rapport a la frontière avec le cluster d'à coté est vide aussi, alors on a une entrée sortie, dans le cas de plusieurs cases adjacentes qui forment des entrées sortie a la suite, on définit l'entrée sortie au milieu de ces cases si la largeur de l'entrée sortie est inférieure a 6, sinon on définit une entrée sortie a chaque extrémité.



Dans le schéma ci dessus les zones de sorties détectées sont en jaune et les entreés sortie finales en vert, la séparation entre deux clusters est en rouge

Une fois les entrées et sorties définies, on crée un graphe abstrait avec les différentes entrées sorties qui sera notre premier niveau d'abstraction, un coté d'une entrée/sortie (une case donc) représente un node, on relie les deux nodes d'une entrées sortie entre eux (schéma ci dessous en vert) la distance entre les deux nodes d'une entrée sortie est toujours de 1. On relie ensuite les nodes entre eux a l'intérieur de chaque cluster (schéma ci dessous en jaune) si il existe un chemin entre eux, a ce moment la on calcule la distance entre les deux nodes en appliquant un pathfinding A* entre les deux a l'intérieur du cluster pour obtenir le coût de déplacement optimal entre ces deux nodes, une fois toutes les entrées sorties reliées et enregistrées dans un graphe, on a notre graphe abstrait sur lequel on exécutera notre recherche de chemin. Chaque node du graph doit avoir enregistré d'une manière ou d'une autre (ID, position etc etc...) le cluster dans lequel il se trouve pour pouvoir facilement lancer la recherche après.

ci dessous la décomposition des 4 clusters en haut a gauche de la map, avec les entrées sorties reliées entre chaque cluster(en vert), et les liens des entrées a l'intérieur d'un cluster en haut a droite(en jaune).



Une fois le graphe abstrait pré-calculé et enregistré, on peut lancer des calculs de pathfinding, les calculs se déroulent en plusieurs étapes, il faut en premier rajouter les nodes de départ et d'arrivée au graphe abstrait, pour cela on trouve dans quel cluster se trouve le départ et l'arrivée, et on calcule le chemin jusqu'aux entrées/sorties du cluster, une fois les deux nodes ajoutés au graphe, on calcule le chemin dans le graphe abstrait, puis on ré-assemble le chemin final en mettant bout a bout les chemins que l'on calcule entre deux entrées sorties a l'intérieur de chaque cluster dans lequel on est passés.

Le chemin découvert de cette manière ne sera pas le chemin optimal, mais s'en rapproche grandement, et il est beaucoup moins long a calculer que le chemin calculé par un pathfinding A* basique, le chemin peut ensuite être affiné grâce a des algorithmes de path smoothing.

On peut ajouter plusieurs niveaux d'abstractions, qui représentent a chaque fois des clusters de plus en plus gros sur la map contenant eux memes plusieurs clusters, pour accélérer encore plus la recherche, mais le chemin découvert avec plusieurs niveaux d'abstractions est souvent encore moins optimisé que le chemin découvert avec un seul niveau d'abstraction, et requiert donc encore plus de post processing et d'affinage pour être optimisé, ce qui revient a rendre le temps final de calcul a peine mieux qu'avec un seul niveau d'abstraction, les chercheurs qui ont proposé cet algorithme recommandent d'ailleurs un seul niveau d'abstraction et des clusters de 10 par 10 cases pour appliquer leur algorithme.

Même si le chemin trouvé a la fin de l'algorithme n'est pas tout a fait optimal, cet algorithme présente de nombreux avantages sur le A* normal :

- il est beaucoup plus rapide a exécuter.
- On peut l'interrompre en plein milieu plus facilement, en effet rien ne nous empêche de calculer simplement le chemin abstrait de cluster en cluster, et de ne calculer le chemin intra-cluster qu'une fois que notre entité arrive dans le cluster en question, on peut donc décomposer le calcul en plusieurs calculs simples (un pathfinding sur une map de 10*10 cases est très rapide a calculer)

Cette méthode présente cependant quelques désavantages également :

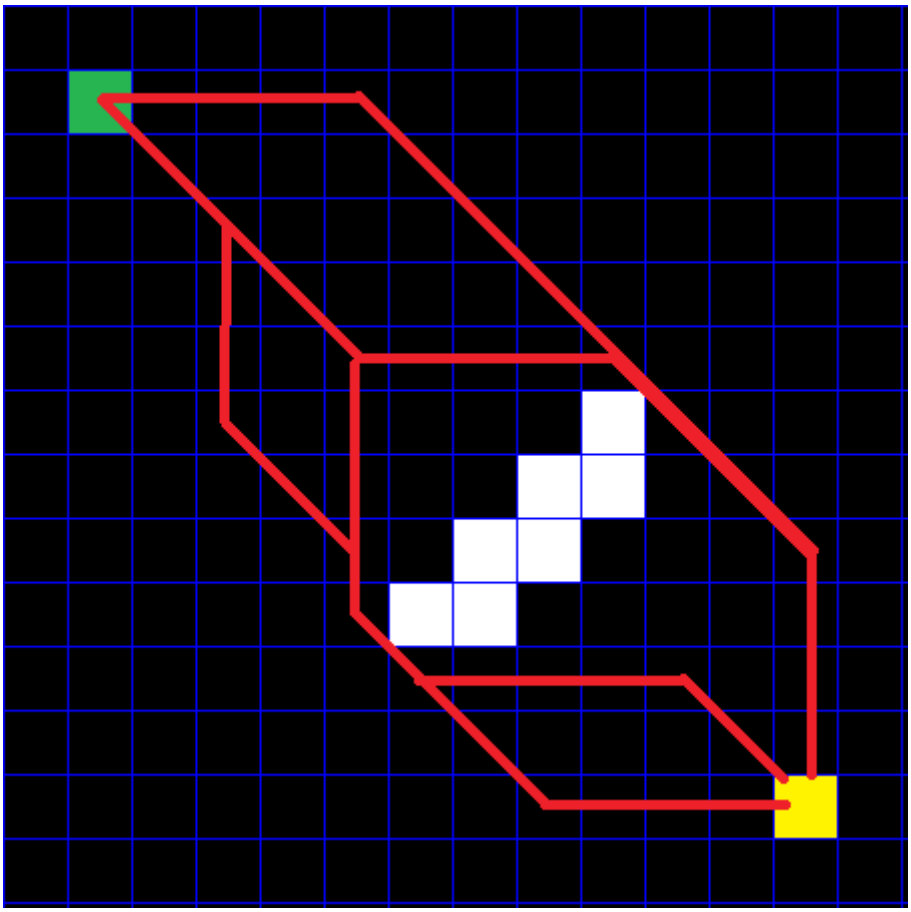
- le précalcul du graphe abstrait demande de réserver un peu de mémoire pour rester enregistré
- un chemin pas tout a fait optimal

Pour finir, on peut remarquer que si le développeur le veut, il peut aussi pré-calculer tous les chemins intra cluster et les garder en mémoire pour accélérer encore la recherche au prix de plus de mémoire utilisée, tout dépend de la taille des maps, de la mémoire disponible, etc etc...

Le problème de la symétrie, le JPS(Jump point search)

Un autre problème inhérent au pathfinding sur des grilles 2d est la symétrie, ce problème est lui aussi a l'origine d'une perte de performances lors de l'utilisation du A*, en effet lorsque l'on teste un chemin avec cet algorithme, il arrive très souvent que le node que l'on teste soit aussi atteignable par un ou plusieurs autres chemins, touts aussi optimaux que celui que l'on est en train de tester, et qui seront d'ailleurs peut être testés juste après, c'est le problème de la symétrie.

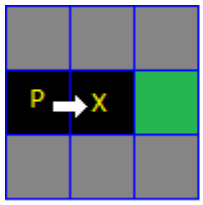
Par exemple dans le schéma ci dessous,tous les chemins (en rouge) entre le départ et l'arrivée (en vert et jaune) sont des chemins optimaux, et selon la manière dont est écrite notre A* tous peuvent très bien être testés presque jusqu'au bout, avant que l'un d'entre eux n'aboutisse a la solution.

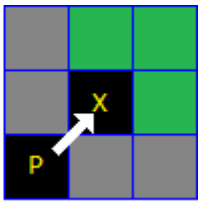


Pour régler ce problème, il y a le JPS (Jump point Search), c'est un ensemble de règles a appliquer au nodes qu'on ajoute a la liste ouverte du A* pour réduire le nombre de nodes testés pendant la recherche.

Lors du A* normal, on choisit un node dans la liste ouverte, puis on s'étend au possiblement sept nodes qui l'entourent (8 moins le parent), et ainsi de suite jusqu'à trouver l'arrivée, les règles du JPS permettent de réduire les nodes sur lesquels on s'étend a entre 1 et 3 nodes en règle générale, 5 dans le pire des cas, de la manière suivante :

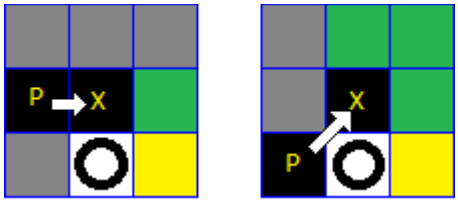
X est le node actuel qu'on vient de prendre de la liste ouverte
P est le node parent de X
au lieu de s'étendre sur tous les nodes autour de X, on va simplement s'étendre sur les nodes qui sont dans la direction vers laquelle on va, en cas de mouvement vertical ou horizontal, c'est seulement le node devant nous qu'on étend, et en cas de mouvement en diagonale, on prend le node devant nous en diagonale, et les deux nodes adjacents.





Les nodes etendus sont en vert, les ignorés en gris

Rien que ça réduit largement le nombre de nodes a tester par rapport au A* normal, en cas d'obstacle la règle est comme le montre le schéma suivant.

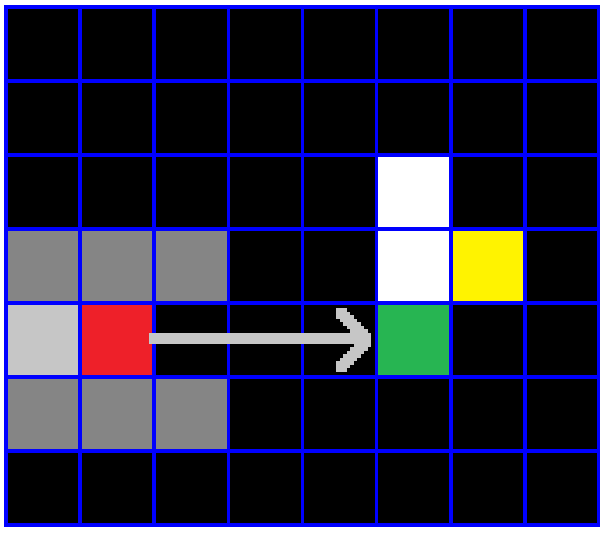


Dans ce schéma l'obstacle est représenté par la case blanche avec un rond noir dedans, les nodes étendus sont en vert et en jaune.

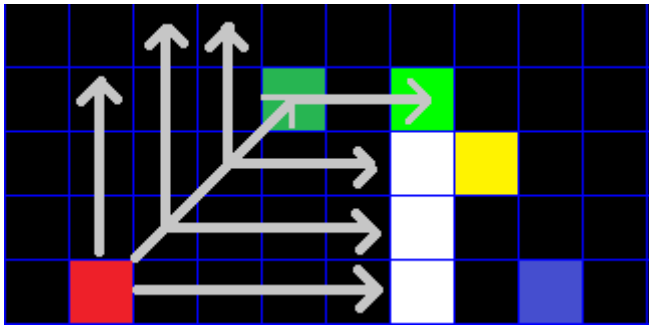
Avec JPS, les nodes étendus ont deux types, ce sont soit des voisins naturels de X(en vert dans le schéma), autrement dit la continuation naturelle du chemin depuis P vers X, soit des voisins forcés de X (en jaune dans le schéma), les voisins sont définis comme voisins forcés de X quand ce ne sont pas un des voisins naturels de X, mais qu'ils ne sont pas atteignables depuis P par un meilleur chemin que celui qui passe par X.

Maintenant qu'on a nos règles, il reste le plus important, au lieu d'ajouter a la liste ouverte les voisins de X selon la règle décrite au dessus, on continue le chemin de manière récursive jusqu'à tomber sur un voisin forcé, le node Y possédant un voisin forcé Z est alors ajouté a la liste ouverte, ce sont les fameux Jump Points du Jump Point Search, tous les nodes parcourus entre le point de depart et le point d'arrivée durant la récursion sont simplement ignorés, on n'enregistre que le node Y ayant un voisin forcé, qu'on appelle un point de saut, avec la direction de laquelle on vient, et on calcule la distance depuis le X initial jusqu'à Y comme coût de déplacement.

Dans le cas d'un déplacement en ligne horizontal ou vertical, on se contente d'étendre récursivement le chemin dans la direction P → X, dans le cas d'un déplacement en diagonale, on étend d'abord le chemin horizontalement et verticalement, avant d'étendre vers la diagonale, puis on recommence a chaque case parcourue en diagonale a étendre les chemins verticaux et horizontaux, si aucun voisin forcé n'est détecte, on ajoute rien, sinon on ajoute les parents des voisins forcés a la liste ouverte.



Dans le schéma ci-dessus on étend le node rouge vers la droite et on rencontre le node vert ayant pour voisin forcé le node jaune, on ajoute alors le node vert a la liste ouverte, et on arrête la récursion, on la relance du node vert qui est maintenant seul dans la liste ouverte et on s'étend encore vers la droite, on ne trouve rien, mais comme on a un voisin forcé au dessus, on s'étendra aussi en diagonale vers le node jaune.



Dans le schéma ci-dessus on étend en diagonale le node rouge, a la recherche de l'arrivée en bleu, on fait d'abord les sauts horizontaux et verticaux, mais on rencontre un mur ou le bord de la map, on avance alors de une case en diagonale et on recommence les sauts horizontaux et verticaux. Après le troisième saut en diagonale, sur le carré vert foncé, le saut horizontal rencontre un voisin forcé (en jaune) en passant sur la case en vert clair, on ajoute donc la case vert foncé a la liste ouverte et on arrête la récursion.

On repart ensuite du node vert foncé qui est maintenant le seul dans la liste ouverte et on l'étend en diagonale comme avec le rouge, on retombe alors sur le node vert clair qui a comme voisin forcé le node jaune, on ajoute le vert clair a la liste ouverte, du vert clair on part horizontalement vers la droite et on rencontre le node jaune en voisin forcé, on l'ajoute a la liste ouverte, etc etc, jusqu'à l'arrivée en bleu.

Cet algorithme permet d'améliorer entre 10 et 30 fois les performances du A* classique, ne demande pas le moindre pré-processing, et a pour seul défaut de n'être optimal que dans le cas d'une grille uniforme, avec des couts de déplacement de 1 dans toutes les directions.

Conclusion

Le pathfinding pour les grilles 2d ne se limite pas au A*, et il existe de nombreuses améliorations possibles de l'algorithme, ayant chacune leur défauts et qualités, ce mémoire résume seulement les deux plus puissantes de ces améliorations, avec d'un coté le HPA* qui permet non seulement d'optimiser grandement la vitesse de calcul d'un chemin, mais permet aussi d'étaler plus facilement ce calcul dans le temps sans pour autant se tromper de chemin, plutôt que de tout faire en une frame, au prix d'un chemin pas tout a fait optimal et d'un peu de mémoire réservée pour les pré calculs. Le JPS permet de son coté la meilleure vitesse de calcul de chemin et un chemin 100% optimal dans tous les cas, mais doit par contre s'effectuer d'un seul coup, avec toutes les récursions nécessaires pour l'exécuter...

Il existe bien sur beaucoup d'autres méthodes d'optimisation de pathfinding, comme par exemple les Swamps, mais le HPA* et le JPS sont les deux a fournir les meilleurs résultats en terme de vitesse et c'est pour ça que ce sont les algorithmes présentés dans ce mémoire. On peut d'ailleurs ajouter que les deux méthodes sont tout a fait compatibles et qu'on pourrait sans doute les combiner pour obtenir des résultats encore meilleurs.

Bibliographie

Near Optimal Hierarchical Path-Finding
Adi Botea Martin Muller Jonathan Schaeffer
https://webdocs.cs.ualberta.ca/~mmueller/ps/hpastar.pdf

Online Graph Pruning for Pathfinding on Grid Maps
Daniel Harabor and Alban Grastien

Un site avec le JPS bien expliqué
http://zerowidth.com/2013/05/05/jump-point-search-explained.html

un site avec les bases du A* et les binary Heaps
http://www.policyalmanac.org/games/aStarTutorial.htm
http://www.policyalmanac.org/games/binaryHeaps.htm