Projet d'Intelligence Artificielle

Introduction :

Le projet à pour but de nous faire implémenter trois types de recherche de résolution de chemin pour les labyrinthes du jeu Pacman. Un algorithme de recherche en profondeur, un algorithme de recherche en largeur et un algorithme de recherche par heuristique : A*.

Ce rapport présente les différentes solutions proposées et va permettre de les comparer entres elles pour permettre de déterminer la plus efficace s'il y en a une.

Les fonctions implémentées sont présentes dans le fichier search.py où elles sont commentées et ne seront pas expliquées en détail ici.

I-Exploration en profondeur d'abord (dfs : depthFirstSearch)

L'objectif d'un tel algorithme de recherche est d'explorer à fond chaque branche du graphe avant de passer aux suivantes. Afin de répondre à cet objectif, la structure utilisée ici est une pile. Quand on explore un nœud, on observe ses successeurs et on les ajoute dans la pile suivant l'ordre suivant qui a été défini dans le programme original : d'abord le nœud situé au nord s'il y en a, puis celui au sud, est et enfin ouest. Si un nœud a déjà été exploré, on l'ignore. Quand un on a fini d'explorer un nœud, on dépile le dernier nœud empilé, on regarde si c'est le nœud destination sinon, on l'explore à son tour.

En résumé, l'exploration se déroule ainsi : Quand on explore un nœud, le prochain nœud à explorer sera dans les successeurs de ce premier nœud avec comme ordre de priorité : d'abord le successeur situé à l'ouest, puis celui à l'est, sud et enfin nord. Si le nœud n'a pas de successeurs, on explore le dernier successeur découvert lors des explorations précédentes.

Démarche :

Notre but a été de réaliser un algorithme implémentant les caractéristiques suivantes :

-Quand on explore un nœud, on regarde si ce nœud est le nœud de l'objectif auquel cas on s'arrête et on retourne la liste d'action accomplies pour aller à ce nœud.

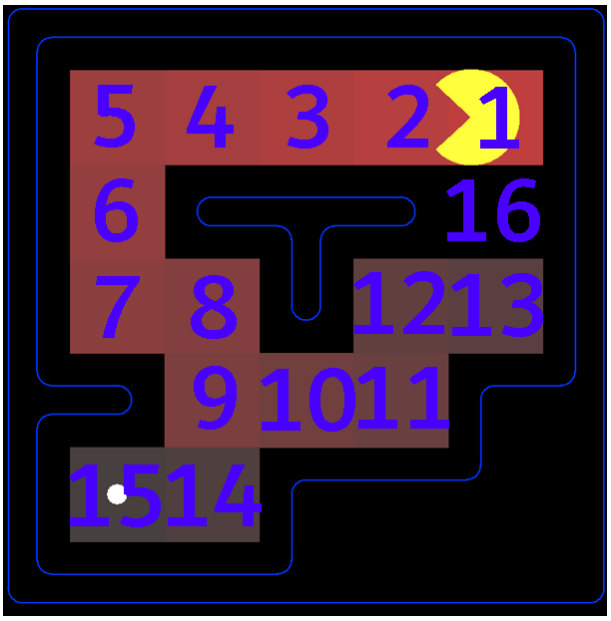
-Sinon on observe ses successeurs

-Si un successeur a déjà été exploré on ne fait rien, sinon on l'ajoute à la pile.

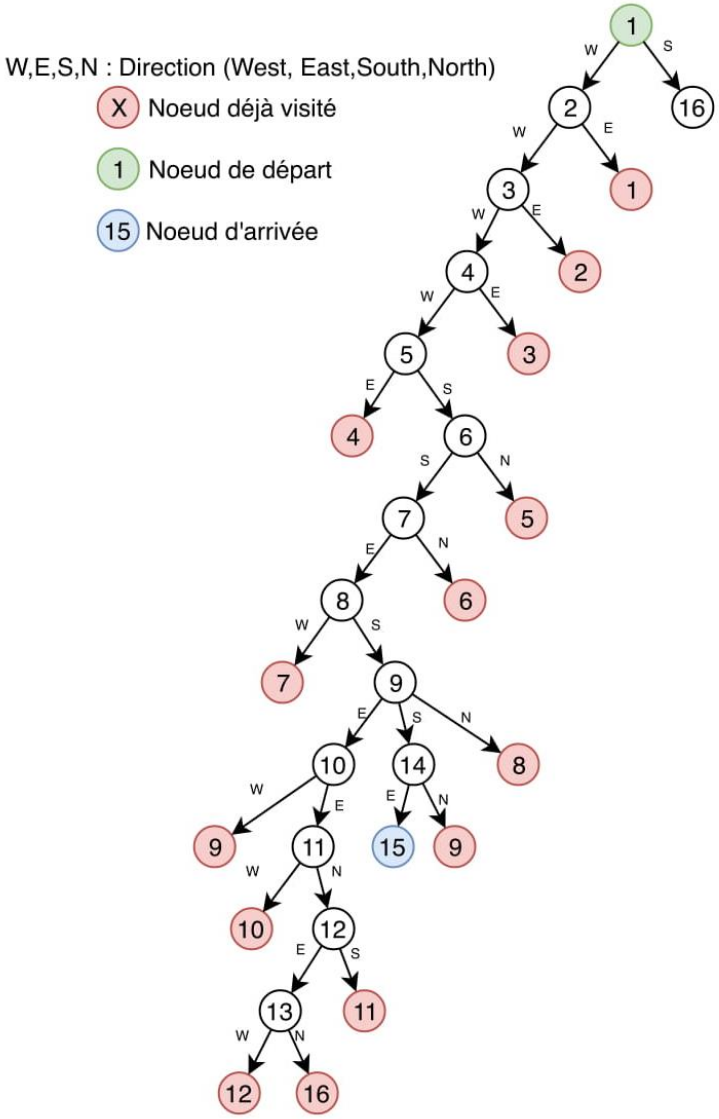
-On dépile le dernier nœud de la pile et on l'explore

-Si la pile est vide, on a exploré tous les nœuds sans trouver de solution et on arrête l'algorithme

Exemple illustré avec le parcours de tinyMaze :



Parcours tinyMaze



L'ordre d'exploration de ce labyrinthe est celui correspondant à la numérotation sauf pour le nœud 16 qui n'est jamais exploré car le nœud destination (15) est atteint avant la découverte de 16. On constate bien que à chaque fois que l'on explore un nœud, le prochain nœud exploré est choisi parmi ses successeurs en respectant la priorité ouest, est, sud, nord. Si un nœud n'a pas de successeurs ou si tous ses successeurs ont déjà été explorés (comme c'est le cas avec le nœud 13), on retourne au dernier nœud qui avait un successeur non déjà exploré (ici 9 et 14).

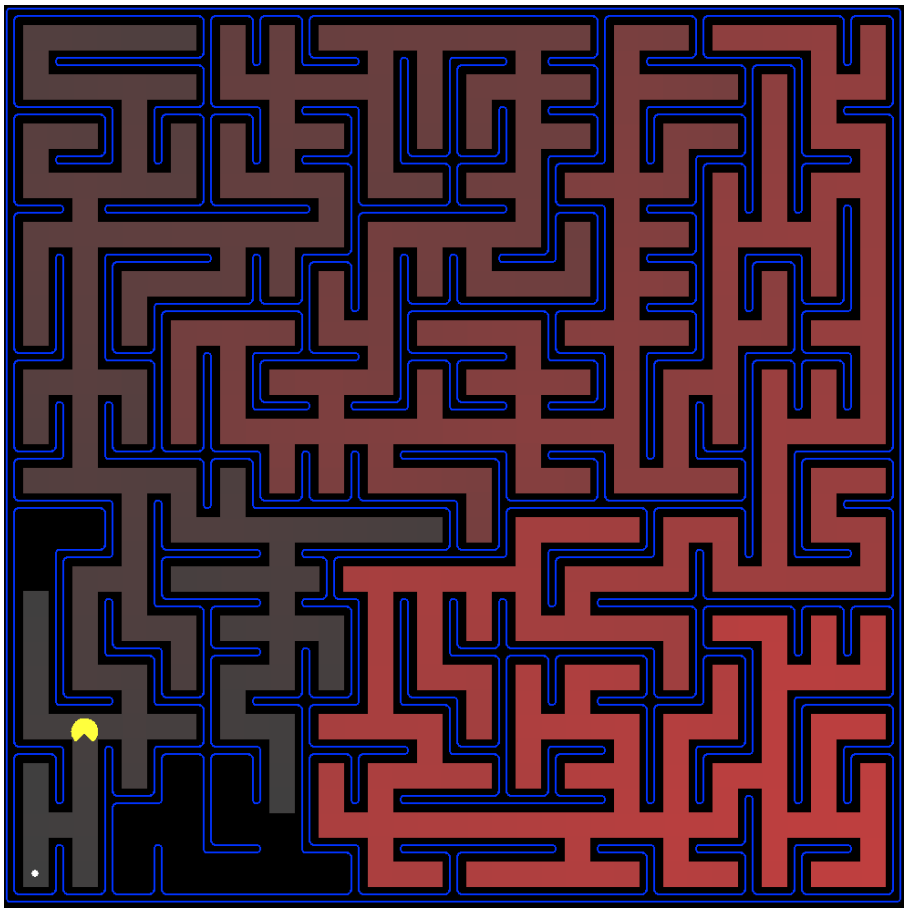
II-Exploration en largeur d'abord (bfs : breadthFirstSearch)

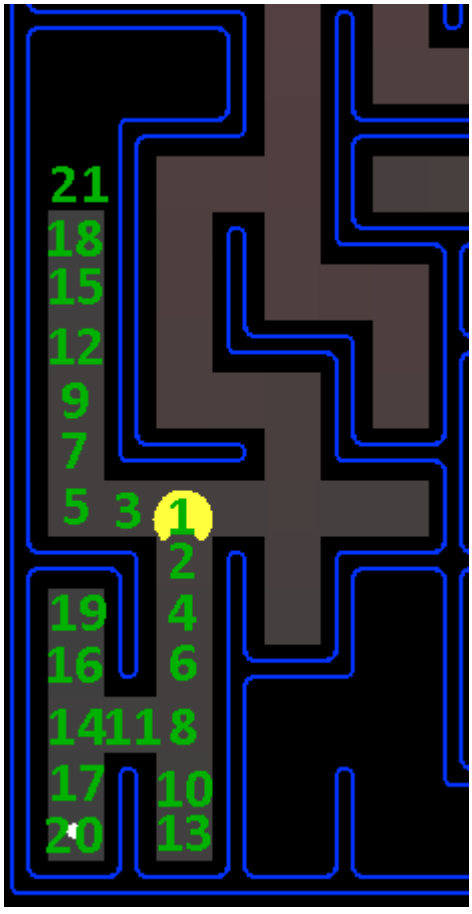
L'objectif d'un tel algorithme de recherche est d'explorer toutes les branches en même temps : lors de l'exploration d'un nœud, on explore tous ses successeurs avant d'explorer ces derniers à leur tour. Pour satisfaire cet objectif, la structure utilisée ici pour l'ordonnancement des nœuds est une file, quand on explore un nœud, on ajoute ses successeurs à la file (toujours en respectant l'ordre Nord, Sud, Est, Ouest). Le prochain nœud exploré est le prochain nœud qui va être défilé qui est également le premier qui a été enfilé.

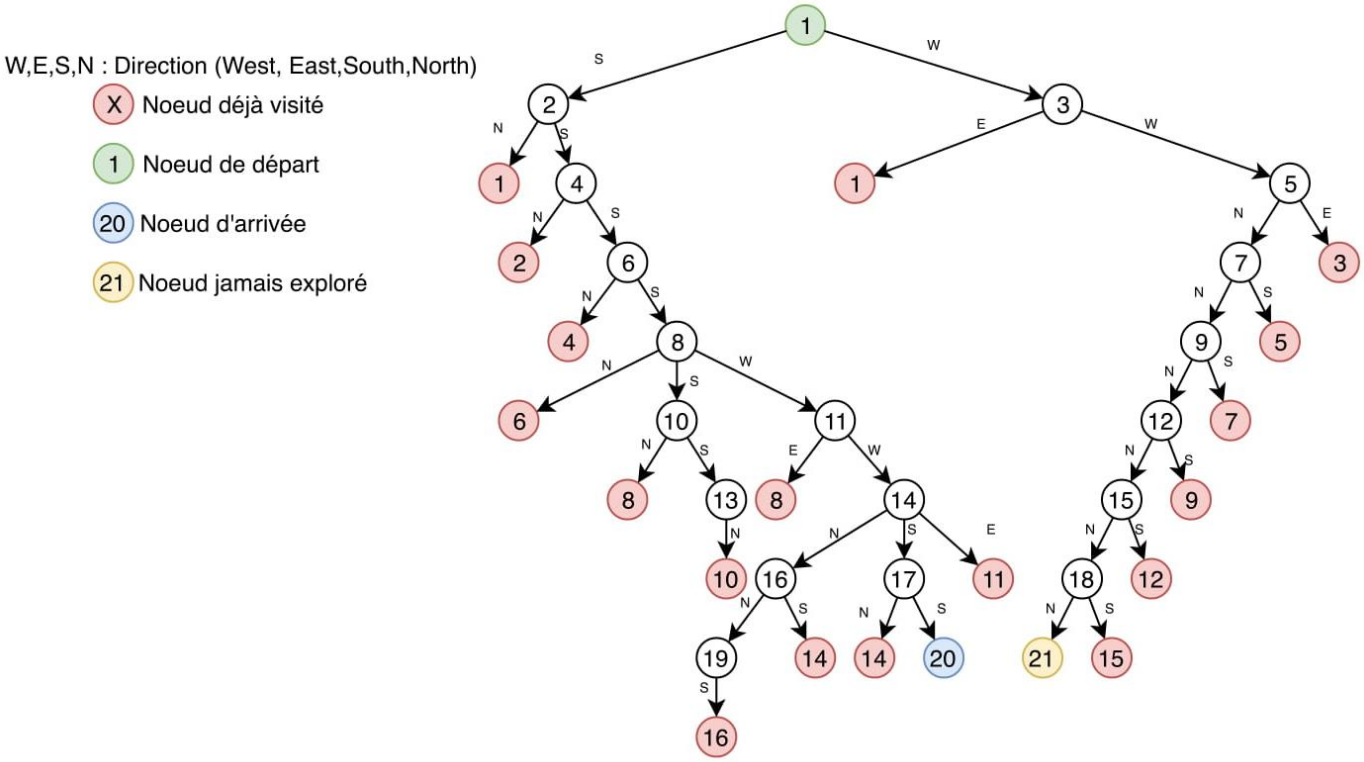
En résumé, cet algorithme est strictement identique au depthFirstSearch mis à part que la structure utilisée est une file à la place d'une pile.

Exemple illustré avec un extrait du parcours de bigMaze :

Dans cet exemple, on utilisera un extrait du labyrinthe bigMaze en considérant le point où est situé le pacman comme point de départ.







Les nœuds sont explorés dans l'ordre dans lequel ils sont numérotés sauf pour le numéro 21 qui n'est jamais exploré car l'exploration est arrêtée après la découverte du nœud 20 qui est le nœud destination. On retrouve bien dans notre exemple les caractéristiques d'un parcours en largueur, toutes les branches sont découvertes progressivement : on explore tous les successeurs d'un nœud en même temps plutôt que d'en explorer un en profondeur puis les autres.

On peut noter que l'on conserve cependant la priorité nord, sud, est, ouest, par exemple quand on explore le nœud 1, on enfile d'abord le nœud 2 qui est au sud, puis le 3 qui est à l'ouest et donc le prochain nœud exploré sera le premier nœud enfilé soit le 2. Quand on explore le nœud 2, on enfile 4 et le prochain nœud à explorer sera le dernier nœud qui reste dans la file soit le nœud 3 et ainsi de suite.

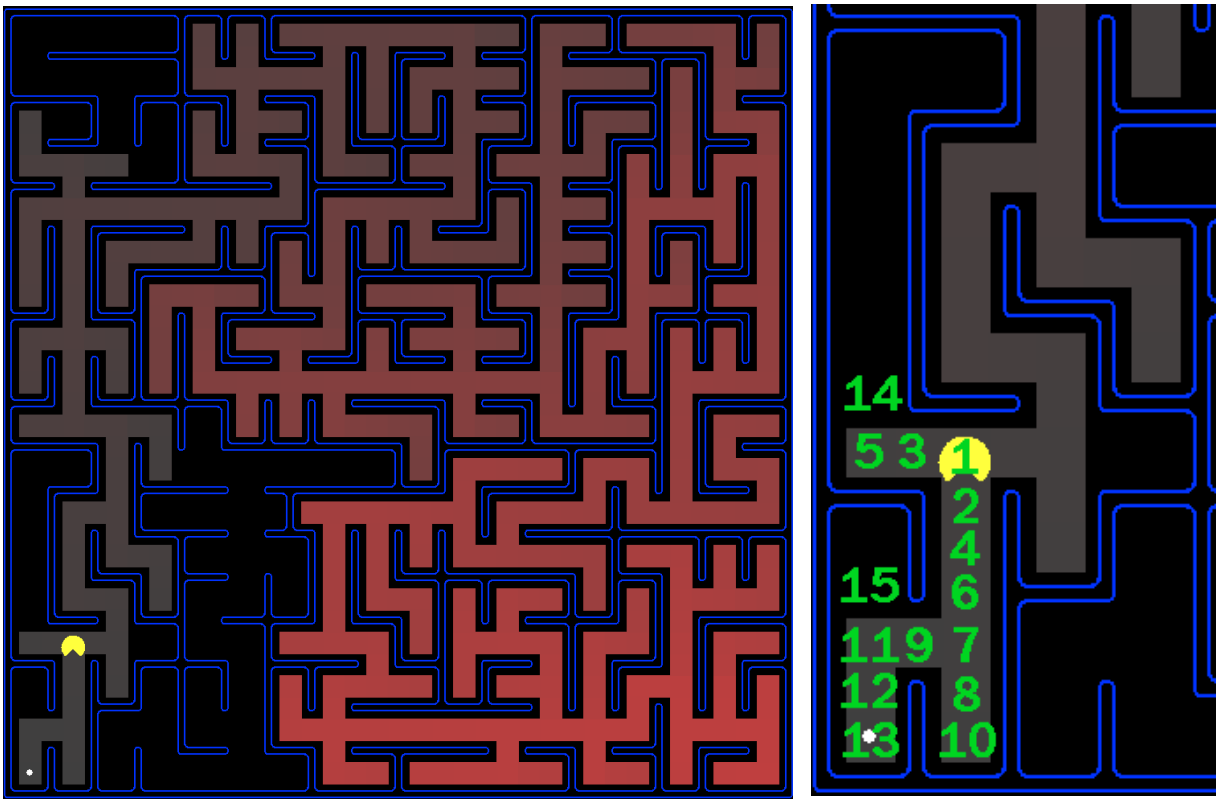
III-Exploration A* (aStar)

L'objectif d'un tel algorithme de recherche est d'éviter le développement des chemins couteux, chaque nœud est évalué par la somme du coût nécessaire à atteindre ce nœud et de l'heuristique à ce nœud qui est l'évaluation du chemin le moins coûteux pour aller de ce nœud jusqu'au but. On va choisir d'explorer en priorité des nœuds avec une évaluation faible plutôt que les autres car on considère que leur exploration nous éloigne de l'objectif.

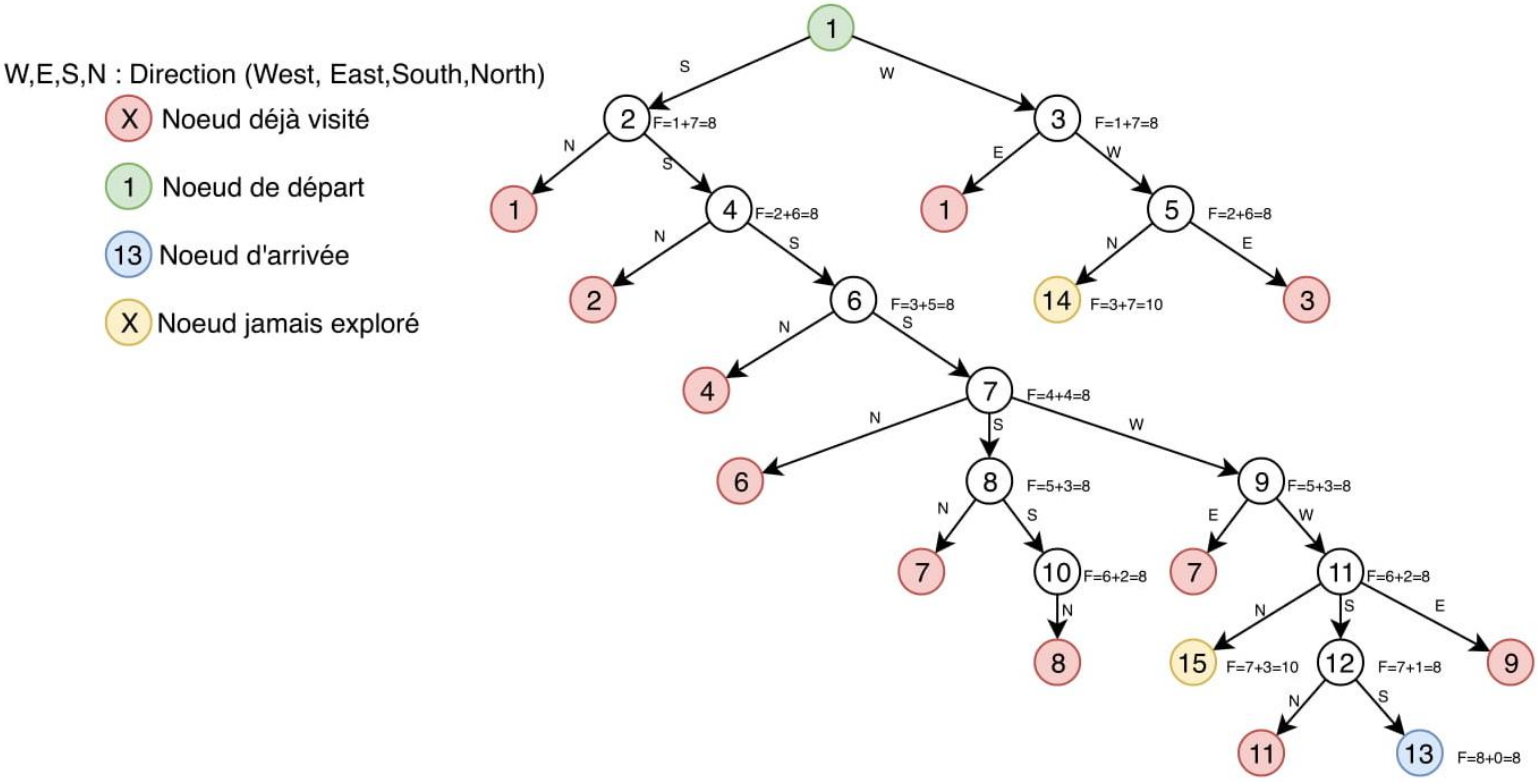
Pour l'implémentation de cet algorithme, on utilise une structure de file avec priorité, qui permet de réorganiser la pile des nœuds en attente d'exploration en fonction de leur priorité afin de faire en sorte que les nœuds avec une plus faible évaluation soient toujours défilés en premier. Lorsque les nœuds de la file ont tous la même priorité, la structure agit comme une file classique de type first in first out et donc l'algorithme revient à effectuer un parcours en largeur classique. Les deux algorithmes sont d'ailleurs identiques mis à part cette variante dans la structure.

Etant donné que le coût de chaque étape est la même pour chaque passage de case (coût = 1), utiliser l'algorithme aStar avec l'heuristique nulle n'a pas vraiment d'intérêt puisque tous les nœuds auront la même heuristique et donc l'exploration reviendra à effectuer un simple parcours en largeur. En revanche en utilisant l'heuristique fournie correspondant à la distance de Manhattan qui associe à l'heuristique de chaque nœud sa distance (coordonnée en x + coordonnée en y) par rapport au but, on devrait constater que les chemins qui s'éloignent de l'objectif ne seront plus explorés en priorité.

Illustration avec le même extrait du parcours de bigMaze que pour bfs :



Sans même se plonger dans l'exploration effectuée, on peut constater sur ces images que en effet, les chemins qui ont tendance à s'éloigner de l'objectif (représentés par les cases 14 à 15) ne sont pas explorés.



Chaque nœud est évalué ici par la fonction F=G + H avec G est le coût nécessaire pour atteindre le nœud depuis le nœud initial et H l'heuristique associée à ce nœud. On constate bien que les nœuds 14 et 15 sont abandonnés par l'exploration car leur évaluation est plus élevée que les autres du fait de leur éloignement de l'objectif. Cet algorithme permet donc de bien optimiser l'exploration en abandonnant les chemins qui s'éloignent de l'objectif comme prévu.

Conclusion sur les performances des algorithmes

Comparaison des différentes explorations pour la labyrinthe bigMaze:

	DFS	BFS	aStar (nullHeuristic)	aStar(ManhattanHeuristic)
Nœuds explorés	390	620	620	549
Nombre d'étapes du chemin trouvé	210	210	210	210

On constate que l'algorithme le plus efficace est l'algorithme d'exploration en profondeur car par chance, le premier chemin exploré en profondeur s'est avéré être le chemin optimal. On constate également que l'algorithme aStar avec heuristique nulle est bien de la même efficacité que l'algorithme d'exploration en largeur puisqu'ils agissent en réalité de la même manière. Enfin

l'algorithme A* avec l'heuristique de la distance de Manhattan grâce à l'exclusion des chemins qui s'éloignent de l'objectif, arrive à trouver la même solution en explorant moins de nœuds.

Comparaison des différentes explorations pour la labyrinthe mediumMaze :

	DFS	BFS	aStar (nullHeuristic)	aStar(ManhattanHeuristic)
Nœuds explorés	144	269	269	221
Nombre d'étapes du chemin trouvé	130	68	68	68

Ce deuxième échantillon confirme les observations faites précédemment, cette fois on constate que l'algorithme d'exploration par profondeur arrive à trouver un chemin en explorant très peu de nœuds mais cependant on s'aperçoit que cette solution n'est pas optimale. En revanche les autres algorithmes arrivent à trouver le chemin optimal et encore une fois c'est l'algorithme A* avec heuristique de Manhattan qui arrive à supprimer des explorations inutiles et donc à trouver la solution le plus rapidement.

Pour conclure on peut dire que dans certains cas particuliers, l'algorithme le plus efficace sera l'algorithme d'exploration en profondeur car il se peut que le chemin optimal soit découvert très rapidement mais cet algorithme est beaucoup moins stable car dans certaines autres situations, il trouvera des chemins qui ne sont pas optimaux ou alors les chemins qu'il explorera en profondeur au début ne seront pas fructueux et il prendra beaucoup plus de temps à se résoudre. L'algorithme A* en revanche sera beaucoup plus régulier et arrivera toujours à trouver un chemin optimal tout en supprimant des étapes par rapport à un algorithme d'exploration en largueur.