VO Minh Tuan – Ramy CHEMAK – Mai 2019

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*).

Les fonctions implémentées sont présentes dans le fichier search.py.

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

Cet algorithme parcoure tous les nœuds du graphe en profondeur jusqu'à trouver le nœud destination. La technique de l'exploration fait que lorsqu'on explore un nœud S, on explore son premier fils et on le rajoute à une une pile des nœud en exploration. Mais avant de passer aux autres fils de S, on développe plutôt ses petits-fils issues du premier fils et ainsi de suite. Donc on essaie d'aller le plus possible en profondeur du graphe pour chaque nœud, d'où l'exploration en profondeur d'abord. A chaque fois qu'on explore un nœud, on teste si c'est le nœud but. Si non, on l'empile à la pile des nœuds en exploration, et on rajoute le chemin qui le relie à son prédécesseur à un dictionnaire des chemins. Si oui, on s'arrête et on retourne le chemin trouvé. Ce chemin est reconstitué à partir du dictionnaire des chemins qu'on met à jour à chaque itération. L'exploration commence à partir de la racine de l'arbre, et on initialise le chemin vers cette racine dans le dictionnaire des chemins à Null.

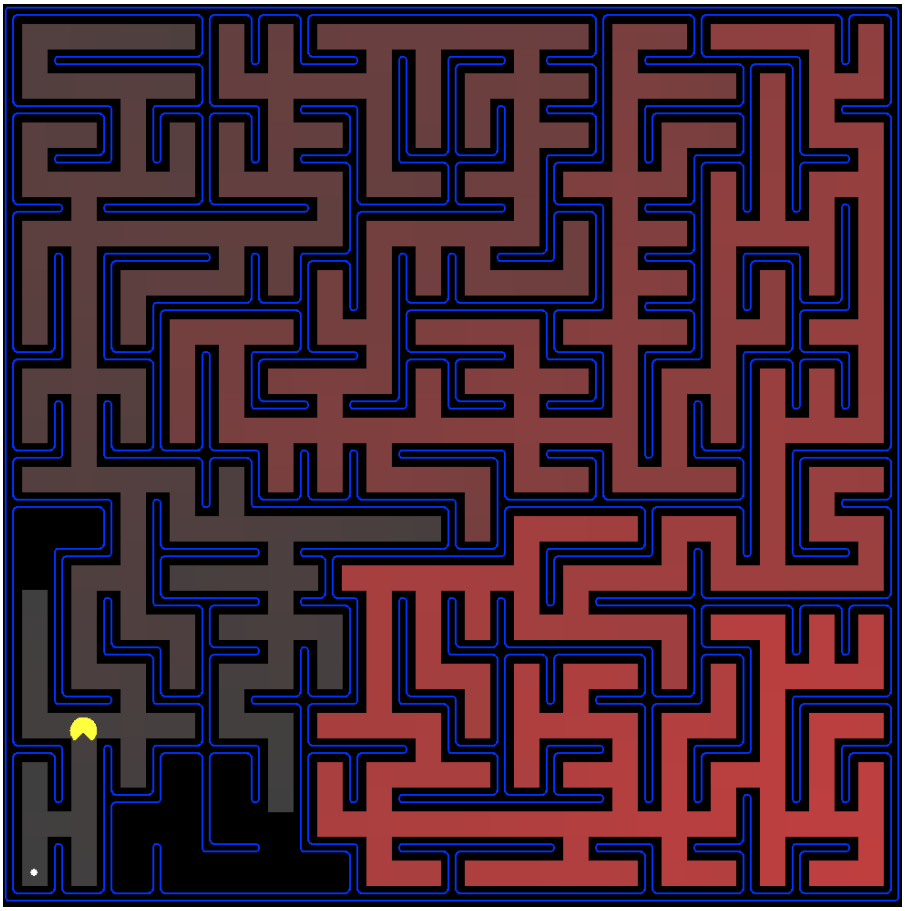
II- Exploration en largeur d'abord (bfs : breadthFirstSearch)

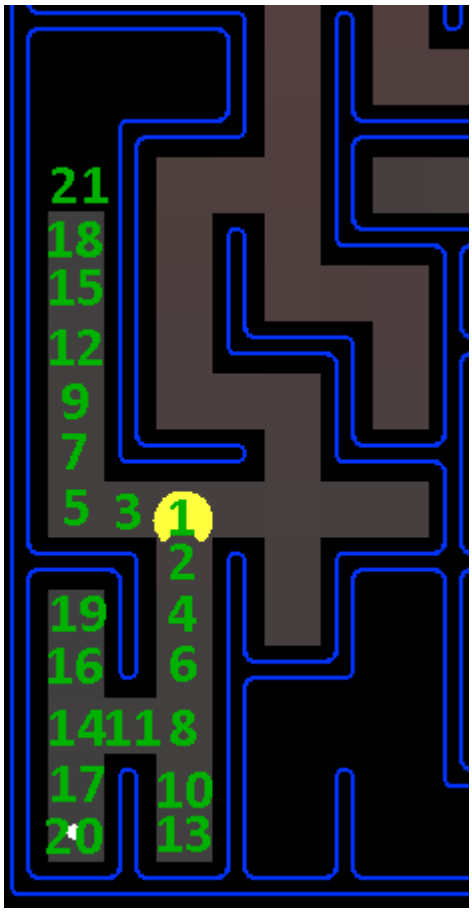
L'objectif d'un tel algorithme de recherche est d'explorer toutes les branches en même temps; lors de l'exploration d'un nœud, on explore tous ses successeurs avant d'explorer ces derniers à leur tour. Pour satisfaire cet objectif, la structure utilisée ici pour l'ordonnancement des nœuds est une file, 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.





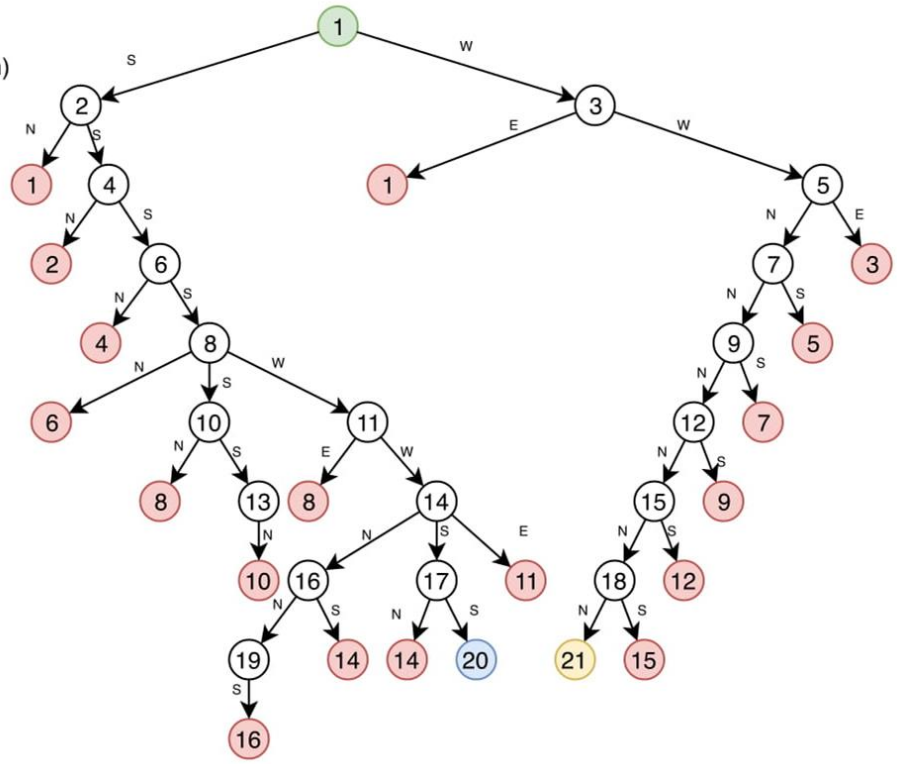
W,E,S,N : Direction (West, East,South,North)

X Noeud déjà visité

1 Noeud de départ

20 Noeud d'arrivée

21 Noeud jamais exploré



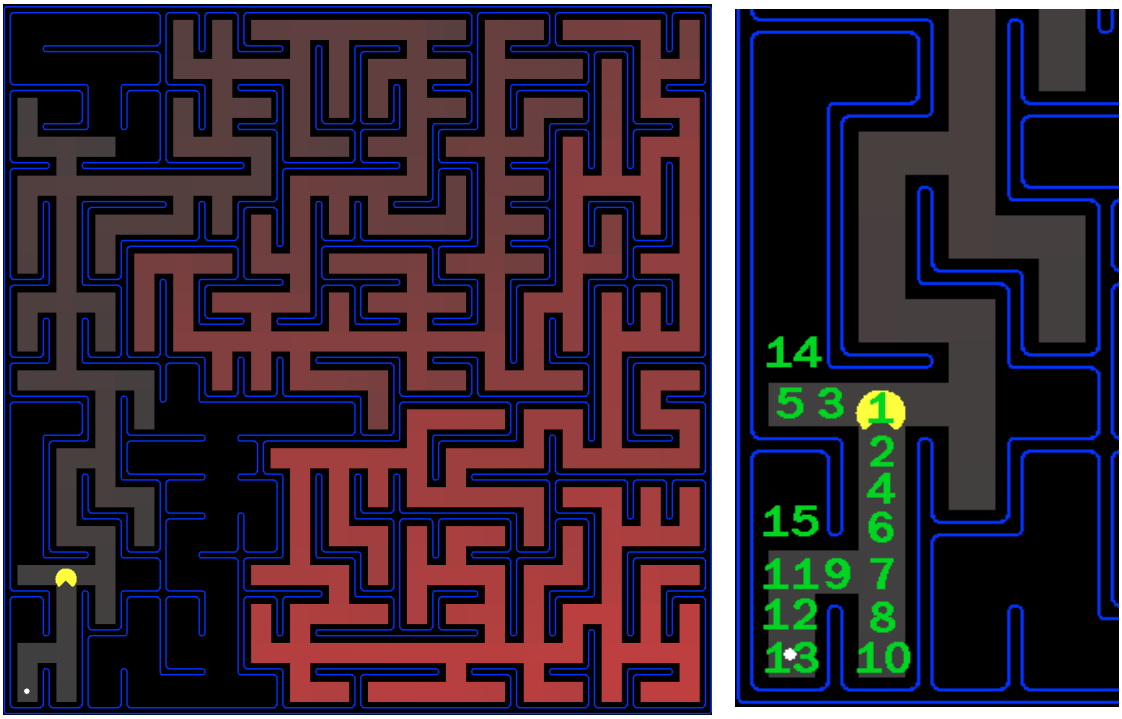
III- Exploration par A*

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.

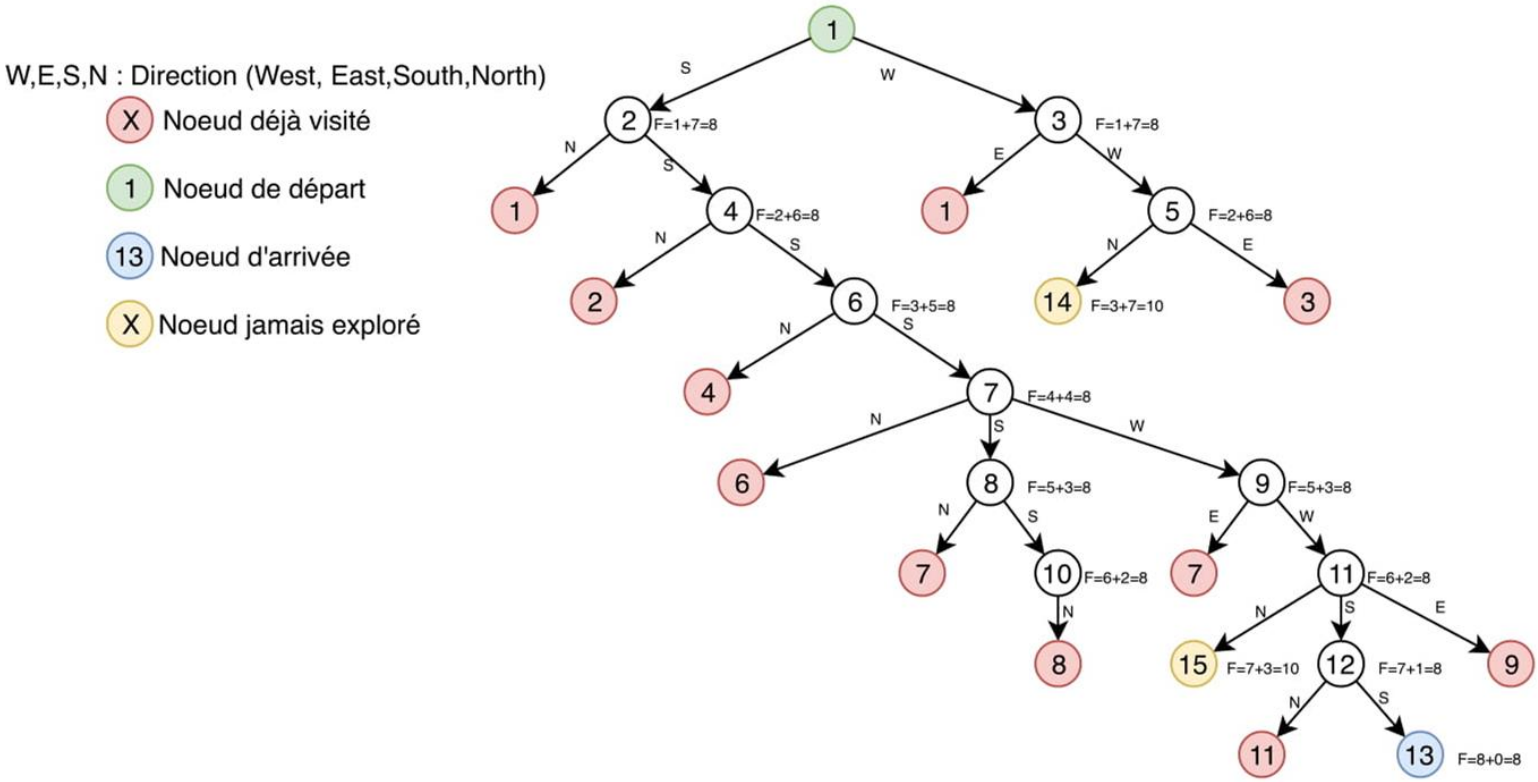
Pour l'implémentation de cet algorithme, on utilise une structure de file avec priorité, qui permet de réorganiser la pile des nœuds en attente d'exploration en fonction de leur priorité afin de faire en sorte que les nœuds avec une plus faible évaluation soient toujours défilés en premier. Lorsque les 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.

IV- Algorithme de Dijkstra - Uniform Cost Search (bonus)

Cet algorithme commence par parcourir le graphe en largeur, c.-à-d. les successeurs de la racine. Puis on explore leurs successeurs à leur tour mais on commence par le nœud le moins couteux, et on continue cet exploration en privilégiant toujours le nœud le moins éloigné de la racine. Pour implémenter cet algorithme, on utilise une file à priorité (PriorityQueue) dans laquelle on enfile chaque nœud qu'on arrive à explorer, mais en le mettant à l'index qui va avec son cout (ici c'est la distance parcourue jusque-là).

Cet algorithme est obsolète dans notre cas, car passer d'un nœud au suivant coute toujours un mouvement, ie toutes les actions ont un cout = 1. Ce qui fait que l'algorithme de Dijkstra appliqué à notre problème se comporte exactement comme un parcours en largeur (Breadth First Search).