INFO8006: Project 1 - Report

Aldeghi Florian - s183157
Baguette Brice - s181482

September 28, 2023

1 Problem statement

a. Un **état de jeu** à ce niveau ci est la position de pacman dans le labyrinthe (pas son orientation N-S-E-O) et l'emplacement des nourritures dans ce labyrinthe. Si le labyrinthe comprend 6 cases où pacman peut se trouver et 3 nourriture, il y a $6 * 3 = 18$ états (certains sont in-atteignable, comme être sur une case où il n'y avait pas de nourriture et plus aucune nourriture en jeu, ou n'existe pas, comme être sur une case nourriture et avoir encore toutes les nourritures en jeu.) L'état initial correspond a l'emplacement initial de pacman ainsi que toutes les nourritures en jeu.

Une **action légale** est un déplacement de pacman (N-S-E-O) qui ne va pas dans un mur du jeu. A chaque état de jeu, sans prendre en compte l'emplacement des nourriture donc dans mon exemple précédent '6', correspond au plus 4 actions légales. Il y aurait donc au plus $6 * 4 = 24$ actions légales dans l'exemple (il y a déjà bien sur tout le contour du labyrinthe qui diminue ce nombre.)

Un **modèle de transition** est le résultat d'une action sur un certain état, ça veut dire ici qu'on regarde l'endroit où pacman se trouve et on envisage chaque déplacement légal ce que ça va apporter. Est-ce qu'on mange une nourriture ou une capsule ? Ou n'est-ce qu'un déplacement ? Cela montre l'état dans lequel on arrive.

Le **goal test** est un test qui vérifie si l'état correspond à l'objectif final, dans ce jeu il s'agit de regarder si toutes les nourritures sont mangées.

Le **coût d'un pas** est le nombre de point qu'une action engendre sur le score final. Pour ce niveau de jeu, nous avons fait correspondre une action de se déplacer à un coût d'un point, et l'objectif final est de minimiser le coût total avant d'arriver à l'objectif final (goal test.) Un déplacement vers une capsule coûte 6 (un pour le déplacement et 5 pour la capsule. En effet, tous ces coûts viennent de notre score qui prend en compte : les déplacement, le nombre de nourriture et de capsule mangée, et de si on gagne (ou perd) la partie. Dans ce score, on ne prend pas en compte les nourritures mangée et la victoire (ou défaite) car à ce niveau ci, ce sont des points fixes qu'on va d'office gagner. Le coût d'un pas est toujours positif.

2 Implementation

a. L'erreur dans implémentassions de DFS est dans la fonction *key*, elle ne renvoi pas un état mais une position. A cause de cela, la liste *closed*, qui vérifie qu'on ne passe pas deux fois par un même état, associe plusieurs états à la même clé car elle ne prend pas en compte la nourriture restante associée à l'état. Pour fixer cette erreur, il suffit que notre *key* nous renvoi la position de Pacman et la position des nourritures dans le labyrinthe.

b. *Leave empty.*

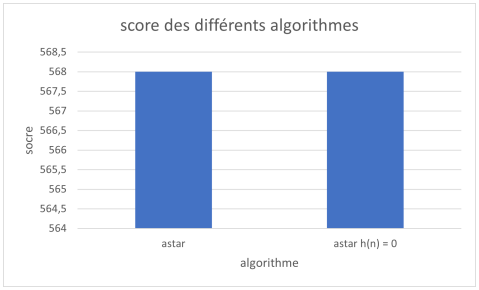
c. Notre fonction g(n) est une fonction qui calcul le coût atteint lorsqu'on est au noeud n qui correspond à un état de jeu. g(n) = nombre de déplacements + 5 fois le nombre de capsule mangée. L'objectif étant d'avoir le plus de point, on se dirige vers un état qui coûte le moins. Notre fonction h(n) estime le coût minimum qu'il reste à faire à partir d'un état. Il s'agit donc de la distance de Manhattan jusqu'au point le plus loin car on devra encore au moins parcourir cette distance avant de gagner la partie. L'heuristique vérifie bien la condition $0 \leq h(n) \leq h^*(n)$. Le coût estimé final est donc $f(n) = g(n) + h(n)$

d. Dans le cas où la fonction h(n) vaut 0 pour tous les noeuds, la fonction g(n) assure la 'completeness' de la fonction. En effet, nous avons un arbre fini grâce a la liste *closed* qui empêche les cycles et donc cette fonction g(n) suffit a parcourir tous les noeuds de l'arbre dans le pire des cas. De plus, par la description du problème, on sait qu'au moins un noeud correspond à une victoire (en réalité, autant de noeud que de nourriture initial.) Ensuite, cette fonction assure l'*'optimality'* de la recherche car s'il existait un noeud plus optimal que celui où on trouve une victoire, on y sera déjà passé avant. Effectivement, la fonction g(n) ne peut que augmenter en descendant dans l'arbre (par rapport à ses ancêtres) et donc s'il y a un noeud B meilleur que le noeud A où on est, c'est qu'un parent de B déjà visité a un $g(B_{parent}) < g(B) < g(A)$. Si lui même n'a pas été visité, on remonte par le même raisonnement vers la racine jusqu'à trouver un noeud déjà visité. Ainsi, dans tous les cas, on atteindrait le noeud B avant A.

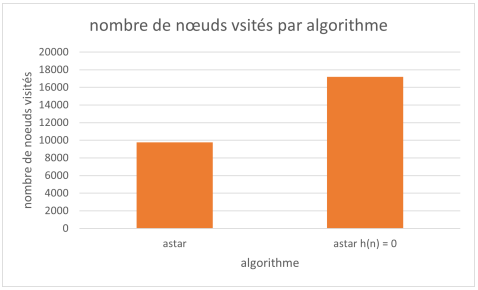
e. *Leave empty.*

f. Le but du BFS est de parcourir les noeuds dans l'ordre de découverte. En mettant le nombre de pas qu'il faut pour atteindre un noeud dans la fonction g(n) (g(n) = n), on est sur de les faire dans l'ordre car on voit d'abord tous les noeuds a distance 1 puis tous ceux à distance 2 à partir de ceux à distance 1 et ainsi de suite. La fonction h(n) = 0 car on ne doit pas utiliser d'heuristique. Pour une recherche en large, on ne doit pas viser une direction plus qu'une autre.

3 Experiment 1



(a) Score



(b) noeud

Figure 1: astar & astar h(n)=0

a.

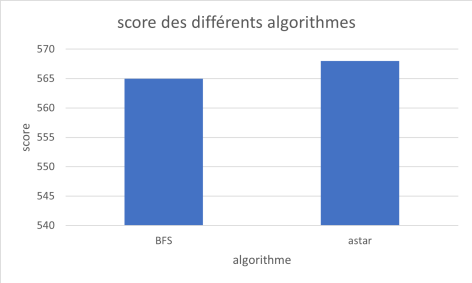
b. Les scores sont identiques et optimal. Pour les noeuds visités, le astar avec notre heuristique visite presque deux fois moins de noeuds que le astar à h(n) = 0.

c. Le score est le même et est le score optimal car comme décrit dans le point 2.d, la fonction g(n) se suffit à elle même pour assurer l'"optimality' de la recherche. Cependant, lorsqu'on utilise un heuristique h(n) qui n'est pas nul, ça aide à se diriger plus vite vers la bonne solution, ce qui explique

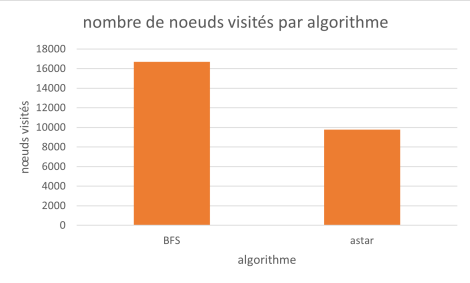
que la première méthode passe par moins de noeud. On peut dire que le premier heuristique domine celui qui vaut 0 car il lui est $\geq$ pour tous les noeuds

d. L'algorithme correspondant au astar avec h(n) = 0 est l'UCS (Uniformed Cost Search)

4 Experiment 2

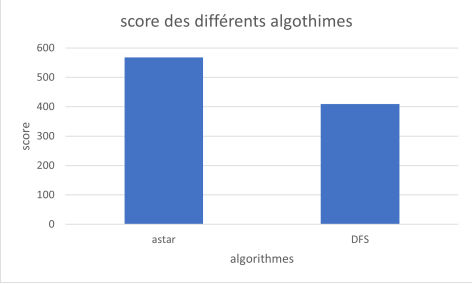


(a) Score

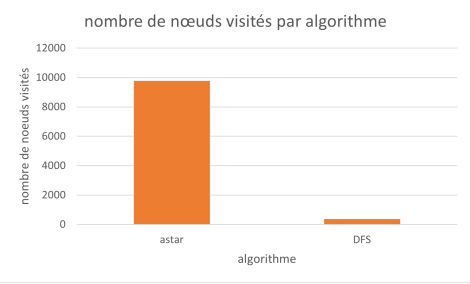


(b) noeud

Figure 2: BFS & Astar



(a) Score



(b) noeud

Figure 3: Astar & DFS

a.

b. - Les score score de astar et DFS sont très différents avec le score de DFS beaucoup plus petit que celui de astar. Cependant, le nombre de noeuds visités par le DFS est presque 30 fois inférieurs a celui du astar.
- Les scores de astar et BFS sont quasi identique avec celui du astar qui est optimal. Pour le nombre de noeuds visités, celui du BFS est presque deux fois plus élevé que celui de astar.

c. - Le score du DFS est largement inférieur car on prend la première solution trouvée, et comme on cherche en profondeur, on arrive très vite dans le fond de l'arbre ce qui signifie que le nombre d'actions effectuées est très grand, ce qui coûte beaucoup sur le score final. Le faible nombre de noeuds visités est expliqué par le fait qu'on trouve très vite une solution vu qu'on se trouve dans le fond de l'arbre qui correspond à beaucoup d'état visités et ainsi une plus grande chance de passer par toutes les nourritures. Le nombre de noeuds visités pour astar est du au fait qu'il cherche une solution optimal sans trop s'enfoncer dans l'arbre.
- Le score du BFS qui n'est pas optimal mais presque s'explique par le fait qu'on ne prend pas en compte le coût de manger une capsule, ce coût étant de 5, ça explique cette différence d'ordre 5

(3). Quant aux noeuds visités, cette différence s'explique par le fait que la fonction astar a une recherche similaire a BFS du point de vue de la fonction g(n) mais astar a une fonction h(n) qui lui permet de visiter les noeuds les moins coûteux seulement dans une direction précise et non pas dans toutes les directions comme le fait BFS. On pourrait voir les noeuds visités par BFS comme un rond autour de l'état initial alors que pour astar, c'est plutôt un ovale avec l'état initial d'un côté.