BOZON Aloïs
BOUSQUET Alexandre

RENDU : IA ET JEUX

Préambule

Nous allons parler de la mise en place des listes et des difficultés rencontrées.
En effet, les listes sont la composante essentielle pour pouvoir mettre en place les différents algorithmes de calcul de solution de jeu.
Notre principale difficulté dans la mise en place des listes a été l'implémentation des fonctions suivantes :

- *onList* :
  Le problème rencontré avec la fonction onList (qui a pour but de retourner un item recherché ou renvoyer NULL si nous le trouvons pas) était lié avec les fonctions des bibliothèques standards du C et en particulier les fonctions *strcmp* et *strncmp*.
  Ces fonctions comparent des chaînes et pas des tableaux, ces fonctions attendaient le caractère de fin de chaîne '\0' et par conséquent nous renvoyaient tout le temps -1 (pas les mêmes chaînes) alors que les tableaux pouvaient être identiques.
  Pour pallier à ce problème, nous avons implémentés la fonction *isSameTab* qui compare deux tableaux (renvoie 1 si les tableaux sont identiques, 0 sinon).
- delList
  Nous avons eu des problèmes par rapport à la compréhension de la fonction, puis par rapport à la gestion des différents cas que pouvait rencontrer la fonction notamment lorsque *delList* devait supprimer un maillon donné par *popBest.*

File de Priorité :

C'est une structure de données qui permet de gérer certaines opérations sur les ensembles et qui sont les suivantes :

• créer une file vide ;

• d'ajouter à la file un élément et sa priorité ;

• d'extraire de la file l'élément de plus haute priorité ; et

Pour être plus précis, une clé c est associée à chaque élément v permettant de déterminer la priorité de l'élément. Pour notre utilisation, l'élément de plus haute priorité est celui avec la plus petite clé. C'est donc le couple (v,c) qui est inséré dans la file.

Ainsi popBest nous a permis de créer des files de priorité afin de nous faciliter la tâche pour UCS.

Une façon simple d'implémenter une file de priorité est d'utiliser un tas. Avec un tas classique implémenté par un arbre binaire quasi-complet (qui est lui-même un simple tableau). On a implémenté un arbre avec notre structure dans item.h qui nous permet le backtracking et de créer des tas depuis nos files de priorité.

DFS, BFS, UCS et A* :

Avant tout résultat présentons rapidement chacun des algorithmes ce qui nous permettra d'avoir des idées sur notre complexité et des résultats asymptotiques avec ceux que l'on a :

DFS :

L'algorithme de parcours en profondeur (ou parcours en profondeur, ou DFS, pour Depth-First Search) est un algorithme de parcours d'arbre, et plus généralement de parcours de Graphe. Son application la plus simple consiste à déterminer s'il existe un chemin d'un sommet à un autre. La complexité en temps du parcours en profondeur est en O(n+m) avec n et m respectivement le nombre de sommets et d'arêtes du graphe.

BFS :

L'algorithme de parcours en largeur permet le parcours d'un graphe ou d'un arbre de la manière suivante : on commence par explorer un nœud source, puis ses successeurs, puis les successeurs non explorés des successeurs, etc. L'algorithme de parcours en largeur permet de calculer les distances de tous les nœuds depuis un nœud source dans un graphe. La complexité en temps du parcours en largeur est en O(n+m) avec n et m respectivement le nombre de sommets et d'arêtes du graphe.

UCS :

UCS est un algorithme de recherche qui utilise le coût cumulé le plus bas pour trouver un chemin de la source à la destination. Les nœuds sont développés, en partant de la racine, en fonction du coût cumulé minimum.UCS est ensuite mise en œuvre à l'aide d'une file de priorité. Il permettra donc d'avoir une complexité en mémoire beaucoup plus basse que DFS et BFS car il ne re visitera pas des sommets déjà visités. Ainsi sa complexité en temps sera bien sur plus élevée du fait que popBest est bien plus coûteux que popFirst et popLast. On implémente UCS avec une file de priorité en ajoutant au début car on veut optimiser notre parcours dans la file de priorité. Ainsi, avec un peu de chance, le sommet le moins coûteux se trouvera au début.

A* :

Il est identique à celui de Dijkstra (croissance d'un arbre de racines, la source, par ajout de feuilles) sinon que le choix du sommet u se fait selon score[u], une valeur qui tient compte non seulement de coût[u] (du coût du chemin dans l'arbre de s à u), mais aussi d'une estimation de la distance entre u et la cible t. Cette estimation sera évaluée par des heuristiques qui permettront de nous faire tendre et prendre la direction vers le sommet cible. De manière générale on nomme heuristique tout algorithme supposé efficace en pratique qui produit un résultat sans garantie de qualité par rapport à la solution optimale.

Knights

Rappel du problème :
Le problème du knights est de déterminer le chemin d'un cavalier entre deux positions d'un échiquier.

Implémentations et résultats :
Voici ci-dessous un tableau comparant et synthétisant les performances de chaque algorithme en termes de solution obtenue et des complexités mémoire et cpu utilisés pour plusieurs tailles de planche de jeu.

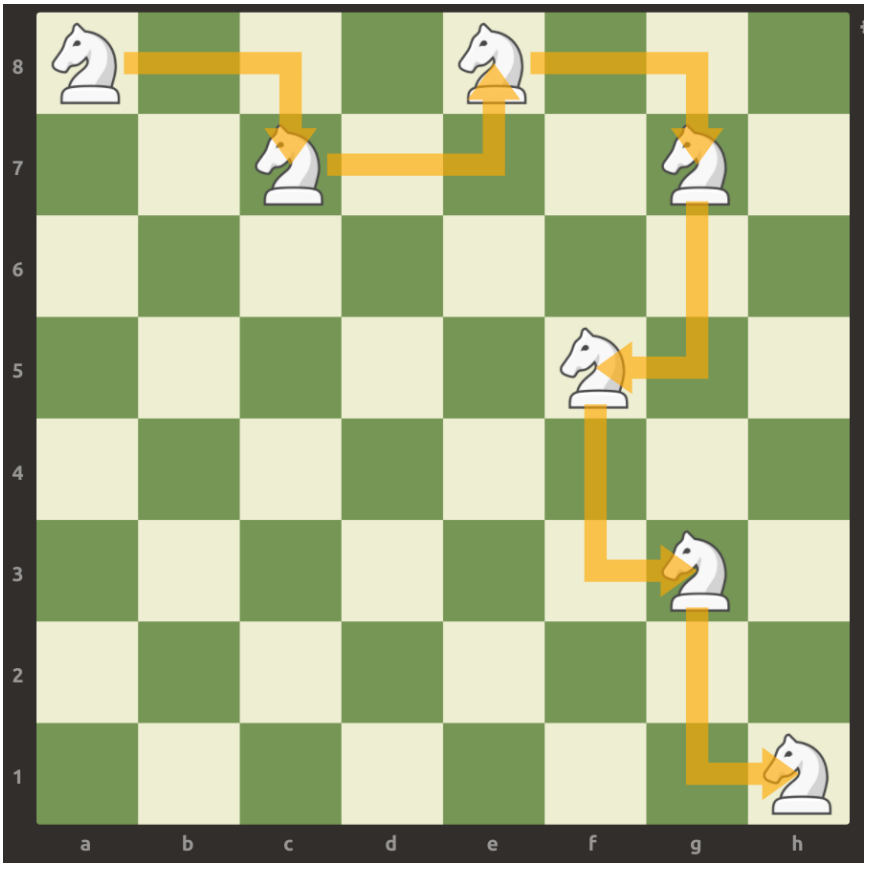
Taille Board	Algorithme utilisé	Complexité mémoire	Temps et Cpu utilisé	Taille solution
6	BFS	Size of open list = 61 Size of closed list = 78	0,0s 91%	4
6	DFS	Size of open list = 16 Size of closed list = 7	0,0s 89%	6
6	UCS	Size of open list = 4 Size of closed list = 32	0,0s 90%	4
7	BFS	Size of open list = 134 Size of closed list = 100	0,0s 94%	4
7	DFS	Size of open list = 44 Size of closed list = 19	0,0s 85%	18
7	UCS	Size of open list = 11 Size of closed list = 37	0,0s 89%	4
8	BFS	Size of open list = 107 Size of closed list = 584	0,04s 98%	6
8	DFS	Size of open list = 42	0,0s 93%	14

		Size of closed list = 15		
8	UCS	Size of open list = 0 Size of closed list = 64	0,0s 92%	6

Illustrations :

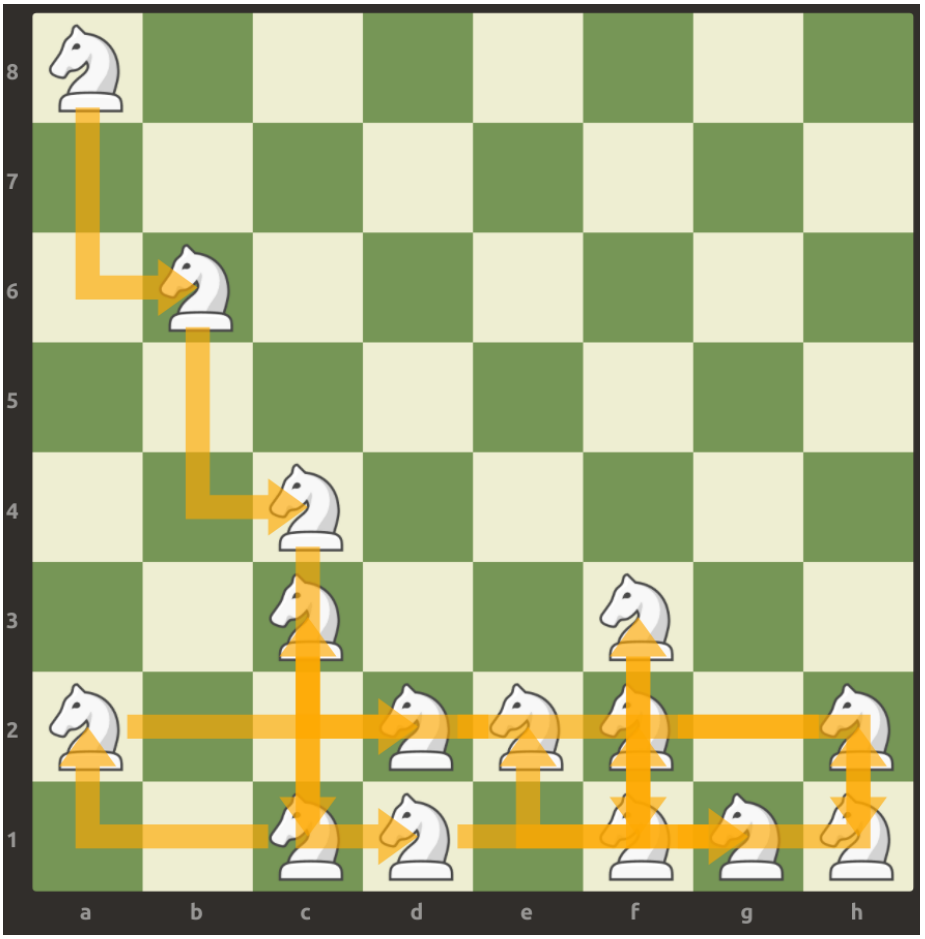
Voici les solutions pour les algorithmes *DFS*, *BFS* et *UCS* sur un échiquier 8x8.
Nous avons représenté les résultats graphiquement.

BFS :



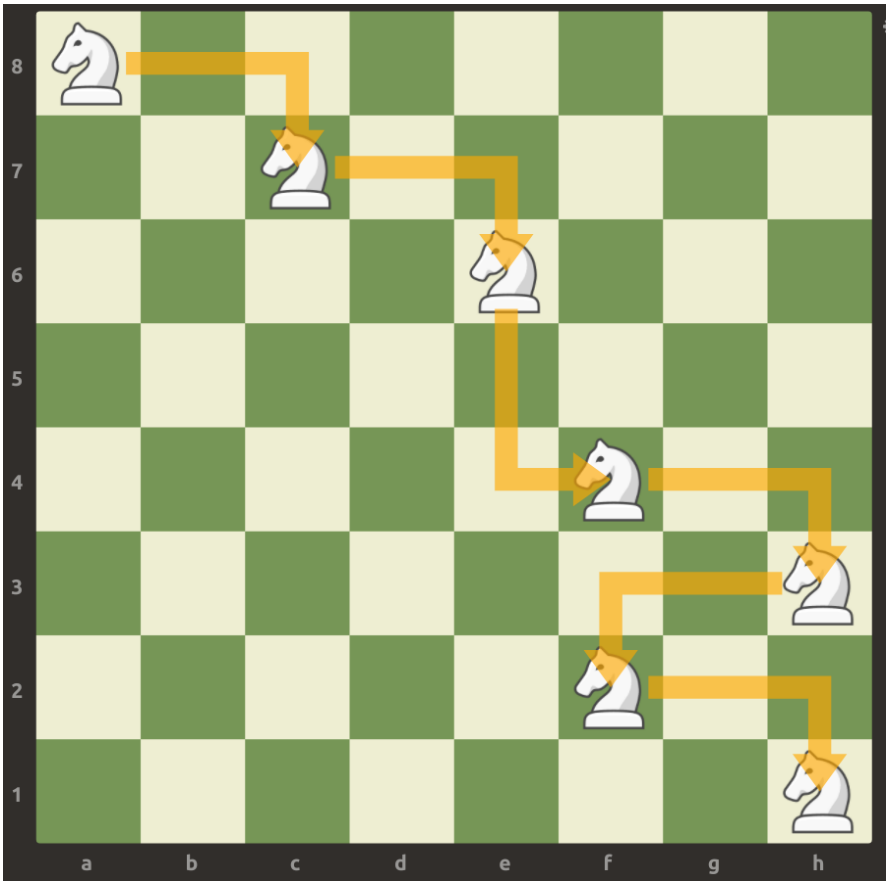
On constate que la solution proposée par l'algorithme BFS est plutôt bonne car le cavalier ne revient pas en arrière ou fait des coups que l'on peut considérer comme inutile.

DFS :



Dans notre problème du knights, on peut constater que la solution proposée par DFS n'est pas du tout optimale car le cavalier fait des coups en arrière et tourne autour du but ce qui fait que la solution est plus longue que celle proposée par BFS ou UCS.

UCS :



La solution proposée par *UCS* est la meilleure parmi les 2 autres. En effet, on peut voir qu'il n'y a pas de retour en arrière ou de coups inutiles.

Conclusion :

On peut voir à l'aide du tableau ainsi qu'avec les illustrations que *UCS* est meilleur en complexité mémoire et offre une solution optimale. Il est évident que UCS est meilleur du fait qu'il ne rejoue pas un coup ou ne visite plus de sommets déjà visité on peut le voir sur la Closed List qui est bien plus conséquente. BFS est ici meilleur (en termes de longueur de solution) que DFS car on peut trouver la solution à des profondeurs beaucoup moins profondes sans jouer trop de coup puisque la solution optimale semble ne pas être trop profonde. Néanmoins, la complexité en mémoire du DFS est nettement meilleure que BFS.

Nqueens

Rappel du problème :
Le problème des Nqueens consiste à positionner les N reines sur un échiquier NxN sans qu'aucune ne s'attaque.

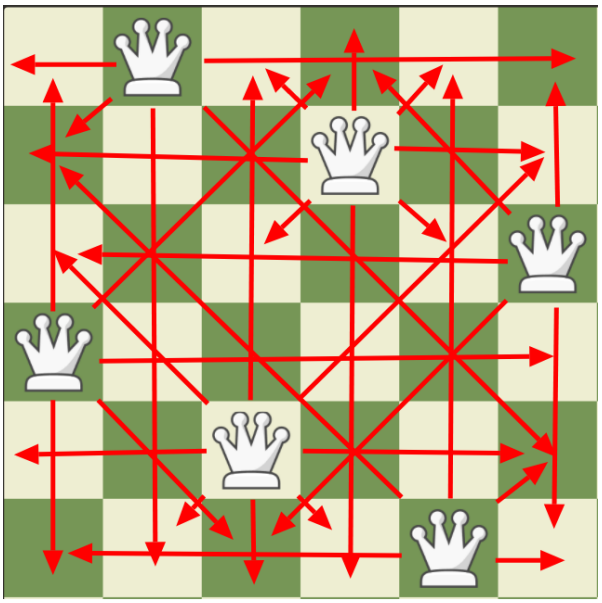
Implémentations et Résultats :
Voici ci-dessous un tableau comparant et synthétisant les performances de chaque algorithme en termes de solution obtenue et des complexités mémoire et cpu utilisés pour plusieurs tailles de planche de jeu.

Taille Board	Algorithme utilisé	Complexité mémoire	Temps et Cpu utilisé	Taille solution
6	BFS	Size of open list = 2879 Size of closed list = 60190	0,12s 97%	6
6	DFS	Size of open list = 68 Size of closed list = 1440	0,01s 96%	6
6	UCS	Size of open list = 3 Size of closed list = 2632	0,18s 99%	6
7	BFS	Size of open list = 201599 Size of closed list = 1344467	3,30s 99%	7
7	DFS	Size of open list = 103 Size of closed list = 12	0,00s 90%	7
7	UCS	Size of open list = 39 Size of closed list = 16831	8,78s 99%	7
8	BFS	Opened List : ? Closed List : ? Insolvable	77,19s 98%	?
8	DFS	Size of open list = 149 Size of closed list = 25944	0,12s 98%	8

8	UCS	Opened List : ? Closed List : ? Insolvable	95s 98%	?

Illustrations :

DFS, BFS et UCS :



(Les flèches rouges indiquent les différents déplacements que peuvent faire chaque reine)

Conclusion :

La solution est du même type, la différence réside dans le nombre de reines qui changent selon la taille de l'échiquier (NxN) pour les 3 algorithmes. Ce qui compte le plus sont les performances des 3 algorithmes, c'est-à-dire, leur temps d'exécution et leur complexité mémoire. On peut remarquer que l'algorithme DFS est le plus efficace (plus rapide que les 2 autres) et il arrive à trouver une solution là où les 2 autres n'y arrivent pas. Le problème de UCS ici est qu'il perd beaucoup de temps à évaluer si le sommet appartient à la Closed List avec la fonction OnList qui est utilisée deux fois dans UCS mais seulement une fois dans le DFS.

Voilà pourquoi DFS est meilleur que UCS dans ce cas particulier.

Taquin :

Rappel du problème :
Sur un plateau de 9 cases, il faut replacer, en les glissant, 8 cases dans le bon ordre avec le moins de mouvements possibles.

Le bon ordre ici est:

1	2	3
4	5	6
7	8	

Trois niveaux possibles :

Facile :

Etat initial

1	2	5
3		4
7	8	6

Moyen :

Etat initial

7	4	8
2	5	6
3	1	0

Difficile :

Etat initial

8		7
5	6	1
3	2	4

Regardons maintenant les résultats correspondant aux algorithmes implémentés pour chaque niveau de difficulté.

Implémentations et Résultats :

Voici ci-dessous un tableau comparant et synthétisant les performances de chaque algorithme en termes de solution obtenue et des complexités mémoire et cpu utilisés pour plusieurs tailles de planche de jeu.

Niveau Difficulté	Algorithme utilisé	Complexité mémoire	Temps et Cpu utilisé	Taille solution
Facile	BFS	Size of open list = 1218 Size of closed list = 1912	0,04s 3% cpu	12
Facile	DFS	Size of open list = 18265 Size of closed list = 23121	4,85s 58% cpu	22734
Facile	UCS	Size of open list = 1106 Size of closed list = 1816	0,05s 3% cpu	12
Facile	A*	Size of open list = 31 Size of closed list = 41	0,00s 0% cpu	12
Moyen	BFS	Opened List : Closed List :	Ne termine Pas	Ne termine Pas
Moyen	DFS	Opened List : Closed List :	Ne termine Pas	Ne termine Pas
Moyen	UCS	Opened List : Closed List :	Ne termine Pas	Ne termine Pas
Moyen	A*	Size of open list = 1037 Size of closed list = 1920	0,05s 3% cpu	26
Difficile	BFS	Opened List : Closed List :	Ne termine pas	Ne termine Pas
Difficile	DFS	Opened List : Closed List :	Ne termine Pas	Ne termine Pas
Difficile	UCS	Opened List : Closed List :	Ne termine Pas	Ne termine Pas
Difficile	A*	Size of open list = 2389 Size of closed list = 4813	0,36s 16% cpu	29

On utilise ici l'heuristique de Manhattan pour A* et pas l'heuristique qui renvoie zéro car sinon A* avec l'heuristique nulle devient Dijkstra.

Conclusion :

Nous pouvons constater que l'algorithme A* est l'algorithme à utiliser si l'on veut une solution dans toutes les situations (facile, intermédiaire, difficile). De plus, on peut constater que sa complexité mémoire n'est pas conséquente (seulement plusieurs milliers de cas traités par l'algorithme) et que son temps d'exécution est très bon (<0,4s) même sur des taquins difficiles. On obtient des résultats aussi performants du fait que A* possède une heuristique donc une opération dite élémentaire qui nous fait tendre vers la solution optimale. On aura ainsi une sorte de "direction" dans nos ajouts dans la file de priorité qui nous permettra de tendre de plus en plus rapidement vers la solution optimale. Ainsi comme UCS, A* ne considère pas certains chemins qui sont trop coûteux et de plus, il connaît à n'importe quel moment la distance au sommet cible donné par l'heuristique ce qui facilite grandement la tâche.

Gestion de projet

Durant toute la longueur du projet, nous avons travaillé en collaboration.
Nous avons chacun de notre côté implémenté chaque module en s'entraidant en cas de problème. Cela nous a permis de pratiquer et de comprendre correctement toutes les notions à saisir.
Voici un calendrier qui se rapproche de ce qu'on a pu produire durant les heures de cours.
Il faut savoir que nous avons aussi beaucoup travailler en dehors des heures de cours.
C'est pour cela que nous avons fini le projet en avance (courant mi décembre).

Date	24/11	30/11	01/12	02/12	07/12	09/12	14/12
Avancement	Implém. du module liste	Implém. module nqueens (bfs,dfs)	Implém. module knights (bfs, dfs)	Problème sur UCS	Résolution problème UCS (onList)	Implém. module taquin (bfs, dfs, ucs)	Implém. algo A* *FIN PROJET*

Conclusion

Cette UE nous a permis d'assimiler correctement toutes les notions de File, listes, arbres ainsi que tous les algorithmes de recherche de solution (bfs, dfs, ucs, A*). On a pu développer ses compétences sur des jeux ce qui a été pédagogiquement plus simple. De plus, nous avons pu constater les différents problèmes liés à ces algorithmes et leurs limites.
Pour approfondir notre travail, nous aurions pu travailler sur des heuristiques plus fines, qui nous auraient probablement données des solutions moins longues et moins coûteuses en mémoire.
Nous aurions aussi pu utiliser l'algorithme Double A* qui consiste à faire A* depuis la cible et la source, des tas plus avancés comme le tas de fibonacci qui est un exemple de tas - minimum.