BOUTELDJA Wassim
FURNON Clément

Rapport Projet Algorithmique

Intelligence Artificielle

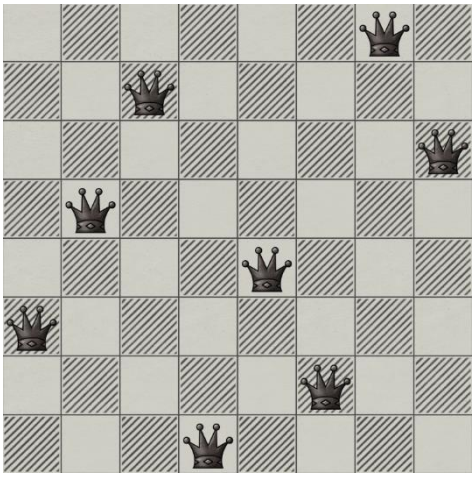
Au cours des TPs nous avons résolu des problèmes d'échecs simples comme celui des NQueens ou du Cavalier ainsi que la résolution du jeu de taquin. Nous nous sommes appuyés sur des structures de données et algorithmes vu en cours (Piles, Files, BFS, DFS, UCS).

Nous avons commencé par implémenter les fonctions et procédures nécessaires à l'utilisation des listes doublement chainées (cf. list.c et list.h) :

- nodeAlloc : alloue et crée un nœud
- freeItem : libère la mémoire d'un nœud
- initList : initialise et retourne une liste vide
- cleanupList : vide la liste et libère la mémoire
- listCount : retourne le nombre d'éléments dans la liste
- addLast // addFirst : ajoute un élément en queue/tête
- popLast //popFirst : retire et retourne l'élément en queue/tête
- popBest : retire et retourne le meilleur élément de la liste
- onList : recherche un élément (par sa valeur) et le retourne
- delList : supprime un élément de la liste
- printList : affiche le contenu d'une liste

Problème des N-Queens

Le problème des N-Queens consiste à placer N reines sur un échiquier de taille NxN en faisant en sorte qu'aucune reine ne puisse en prendre une autre. C'est-à-dire qu'il ne peut pas y avoir plus d'une reine sur la même ligne, la même colonne et la même diagonale.



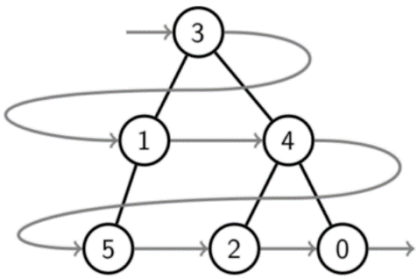
Nous avons ensuite implémenté les fonctions relatives à l'utilisation de l'échiquier :

- initGame : construit le nœud initial, ici c'est un échiquier de taille NxN rempli de 0
- evaluateBoard : renvoie le nombre de reine qu'il reste à placer
- isValidPosition : teste si on peut placer une reine sur une certaine case, renvoie 0 ou 1

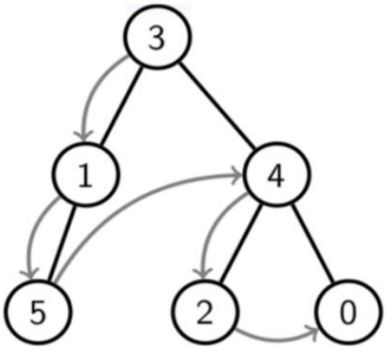
- getChildBoard : si la position testée est valide on crée le fils correspondant et on le lie à son parent
- printBoard : affiche l'état de l'échiquier du nœud donné en paramètre

Nous avons ensuite implémenté les algorithmes de recherche simple :

- Le parcours en largeur (BFS - Breadth First Search)
  Cette fonction effectue une recherche en largeur d'abord. Elle utilise une file pour stocker les nœuds à explorer et elle prend le premier élément de la file à chaque tour de boucle. Les nouveaux nœuds sont ajoutés à la fin de la file pour être explorés plus tard. Les nœuds visités sont ajoutés au début de la liste "visités". Si un nœud est trouvé avec une valeur égale à zéro en utilisant la fonction evaluateBoard, la fonction showSolution est appelée pour afficher la solution.



- Le parcours en profondeur (DFS - Depth First Search)
  Cette fonction est une implémentation de la recherche en profondeur d'abord. La différence entre cette fonction et la fonction BFS est que DFS considère le dernier nœud ajouté à la liste ouverte alors que BFS considère le premier nœud ajouté à la liste ouverte. Cela change la manière dont les nœuds sont explorés et peut avoir un impact sur la performance et la qualité de la solution trouvée



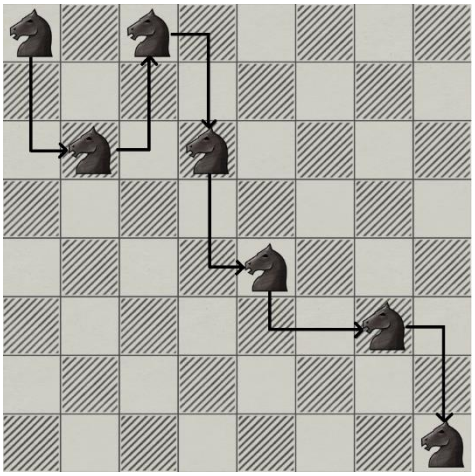
Durant les tests effectués sur des tailles d'échiquiers différents, nous avons constaté qu'il y a une différence au niveau de la mémoire utilisé et le temps mis pour implémenter les deux recherches.

N-Queens		BFS	DFS
6x6	Profondeur	6	6
	Open List	2879	68
	Temps (s)	0,0664	0,002
8x8	Profondeur	8	8
	Open List	3709439	149
	Temps (s)	74,952	0,057
10x10	Profondeur	x	10
	Open List	x	284
	Temps (s)	Killed	0,553

Conclusion :

Il vaut mieux donc éviter le parcours en largeur pour le problème de N-Queens car cet algorithme est plus long et plus coûteux en mémoire, alors on opte pour un parcours en profondeur.

Problème du Cavalier



Le problème du Cavalier que nous avons étudié consiste à trouver un chemin possible sur un échiquier (de taille NxN) entre un le coin supérieur gauche et le coin inférieur droit. On doit respecter les contraintes de déplacement d'un cavalier (déplacement en L).

Pour réaliser la solution de ce problème on se basera sur les algorithmes utilisés pour résoudre le problème des N-Queens. Afin de ne pas entrer dans un cycle, chaque fois que nous visitons un état possible de l'échiquier nous l'ajoutons dans la closedList_p afin d'être sûrs de ne pas repasser par cet état.

Nous avons ensuite implémenté un nouvel algorithme :

- UCS (Uniform Cost Search)
  Cette fonction implémente l'algorithme de recherche en profondeur d'abord (UCS) pour trouver une solution à un problème donné. Elle utilise une liste ouverte pour stocker les états du problème à explorer et une liste fermée pour stocker les états déjà visités. La fonction boucle tant qu'il y a des états dans la liste ouverte, extrait le premier état de la liste ouverte, l'ajoute à la liste fermée et évalue si cet état est la solution. Si ce n'est pas la solution, la fonction génère des états adjacents à cet état et les ajoute à la liste ouverte si ceux-ci ne sont pas déjà sur la liste ouverte ou fermée. Si une solution est trouvée, la fonction appelle la fonction showSolution pour afficher la solution.

Voici les résultats que l'on a obtenu avec les différents algorithmes toujours sur des tailles d'échiquiers différents.

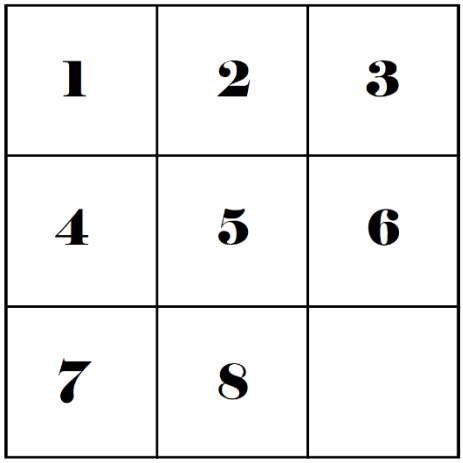
Knight		BFS	DFS	UCS
6x6	Profondeur	4	8	4
	Open List	2	17	2
	Closed List	34	12	34
	Temps (s)	0,0001	0,0005	0,0003
8x8	Profondeur	6	10	6
	Open List	0	21	0
	Closed List	64	13	64
	Temps (s)	0,001	0,0005	0,001
10x10	Profondeur	6	14	6
	Open List	2	13	2
	Closed List	98	68	98
	Temps (s)	0,002	0,003	0,004

Conclusion :

L'algorithme DFS est plus rapide et moins coûteux en mémoire que les autres, ce qui est normal puisqu'il trouve la première solution possible. Les algorithmes UCS et BFS sont plus longs car ils parcourent toute la largeur des solutions possibles, ils accèdent à la profondeur n+1 seulement quand ils ont parcouru tous les états possibles de la profondeur n. Dans notre cas on ne remarque pas de différence entre BFS et UCS en termes de temps d'exécution ou de mémoire. Cependant dans le cas d'un échiquier plus grand UCS permettrait d'accéder plus rapidement à la solution optimale.

Jeu du Taquin

Le problème du jeu du taquin consiste à ordonner des tuiles numérotées dans un cadre carré avec une case vide utilisée pour déplacer les tuiles adjacentes. L'objectif est d'aligner les tuiles en ordre croissant avec la case vide en position finale bas droite.



Partie 1

Nous avons repris les structures de données et algorithmes des problèmes des N-Queens et du Cavalier en modifiant le nécessaire pour les besoins de ce jeu.

De même que pour le problème du cavalier nous avons choisi de considérer que la case vide pouvait être intervertie avec au plus 4 positions adjacentes (haut, droite, bas, gauche) plutôt que de balayer tout le tableau à la recherche des inversions possibles.

Nous avons repris les différents types d'algorithme de recherche que nous avions utilisés dans les problèmes précédents et nous les avons testés sur différents niveaux de difficulté :

Taquin		BFS	DFS	UCS
Easy	Profondeur	12	46544	12
	Open List	1240	33147	1276
	Closed List	2198	51586	2319
	Temps (s)	0,043	92,591	0,0948
Medium	Profondeur	26	58192	26
	Open List	10172	39409	10889
	Closed List	164144	70097	162953
	Temps (s)	586,857	223,537	840,842
Difficult	Profondeur	29	56533	x
	Open List	228	38539	x
	Closed List	181210	67019	x
	Temps (s)	755,88	233,879	Killed

Conclusion :

UCS et BFS permettent de trouver l'enchainement de mouvements le plus court pour atteindre l'état final. Cependant, dans le cas de planches considérées difficile les algorithmes de recherches BFS et UCS ne semblent pas optimisés. En effet, ils sont beaucoup trop longs. DFS nous permet de trouver une solution, cependant la taille des listes est assez conséquente, et il ne permet pas de trouver le chemin le plus court pour atteindre l'état final.

Partie 2

Dans cette partie, nous avions pour objectif de mettre au point l'algorithme A* basé sur l'algorithme de Dijkstra associé à une fonction qui estime le coût entre un nœud quelconque du graphe et un nœud cible. Cette fonction dite heuristique doit sous-estimer le coût du chemin optimal pour que A* soit efficace.

Nous avons donc implémenté une nouvelle fonction :

- getManhattanHeuristic : somme des distances de Manhattan des cases à leur position cible

Taquin		A*	UCS
Easy	Profondeur	12	12
	Open List	51	1276
	Closed List	77	2319
	Temps (s)	0,0003	0,0948
Medium	Profondeur	26	26
	Open List	2227	10889
	Closed List	4192	162953
	Temps (s)	0,198	840,842
Difficult	Profondeur	29	x
	Open List	6133	x
	Closed List	11608	x
	Temps (s)	2,082	Killed

Conclusion :

A* est basé sur UCS donc il permet également de trouver le chemin le plus court pour atteindre l'état final, d'où le fait que quel que soit l'algorithme utilisé on trouve la même profondeur. On remarque que plus la fonction heuristique associée à child_p->g est précise sur le coût réel, plus le temps d'exécution sera faible et la place occupée en mémoire sera également moins importante. En effet, utiliser A* avec une fonction heuristique plus précise permet de dissocier deux nœuds qui avait le même coût avec une fonction heuristique moins précise. On trouvera donc plus facilement un nœud de coût inférieur ce qui augmentera l'efficacité de notre algorithme.