Structures de données
Et
Reines contre Cavaliers

Ce TP consiste à résoudre des problèmes simple d'échecs (8 reines, cavaliers) en utilisant les structures de données et algorithmes vus en cours (Piles, Files, BFS, DFS, UCS). L'implémentation se fera en C et s'appuiera sur des algorithmes et codes vus dans les UE *Algorithmique* et *Programmation*. Téléchargez le kit de démarrage qui contient les fichiers suivants :

- `list.c` et `list.h` : implémentation à compléter d'une liste chainée
- `board.c` et `board.h` : implémentation à compléter de la modélisation du jeu
- `nqueens.c` : implémentation à compléter de l'algorithme BFS pour le problème des N reines

1. Listes : FIFO, LIFO

La première structure dont nous avons besoin est une liste (pile ou file) qui nous permettra d'implémenter les algorithmes de recherche en profondeur et en largeur d'abord. Par intérêt pédagogique et dans un soucis d'efficacité, nous viserons une implémentation par liste chainée. Une version doublement chainée pourra aussi s'avérer pratique par la suite.

a. La structure *« Maillon »*

Un maillon est un élément d'une liste chainée. En C, c'est une structure qui contient une une valeur ou plusieurs valeurs, ainsi qu'un pointeur vers l'élément suivant et éventuellement un pointeur vers l'élément précédent. Cette structure sera utilisée de manière commune avec la structure *« Nœud »* utilisée pour l'implémentation du graphe de recherche. Nous appellerons cette structure `Item`.

b. La structure *« Liste »*

Une liste est une structure qui contient, à minima, un pointeur vers le premier élément de la liste. Dans le cas d'une liste doublement chainée, on peut aussi stocker le pointeur vers le dernier élément de la liste. On stockera aussi le nombre d'éléments contenus dans la liste.

c. Les fonctions à implémenter

Récupérez et adaptez votre implémentation élaborée lors des UEs Algorithmique et Programmation. Pour les besoins des jeux que nous développerons, la structure `Item` contiendra un tableau qui représentera la planche de jeu (échiquier, …). Rassemblez tout cela dans des fichiers `list.h` et `list.c` :

- `newNode(…)` : alloue et crée un nœud
- `freeNode(…)` : libère la mémoire d'un nœud
- `initList(…)` : initialise et retourne une liste vide
- `cleanList(…)` : vide la liste et libère la mémoire

- `push(…)` : ajoute un élément en tête (ou queue) de liste
- `popFirst(…)` : retire et retourne l'élément tête
- `popLast(…)` : retire et retourne l'élément queue
- `onList(…)` : recherche un élément (par sa valeur) et le retourne
- `delList(…)` : supprime un élément de la liste
- `printList(…)` : affiche le contenu d'une liste

Testez et re-testez bien votre liste. Une fois que vous êtes sûr, passez à la partie suivante.

2. Modélisation du problème des 8 reines

Cette partie consiste à mettre en œuvre le mécanisme de simulation par graphe de recherche pour résoudre un problème visé. Nous allons, dans un premier temps, se concentrer sur le problème des 8 reines. Implémentez les fonctions définies ci-dessous, en complétant les fichiers fournis `board.h` et `board.c` :

Le problème des N reines consiste à positionner les reines sur un échiquier NxN sans qu'aucune ne s'attaquent. L'idée est de partir d'un échiquier vide, puis d'ajouter une reine sur une position valide à chaque nouveau nœud de l'arbre de recherche. Une solution est trouvée lorsque les N reines sont positionnées. On considèrera l'échiquier comme un tableau (à une seule dimension) de N*N éléments, les positions vont de 0 à (N*N – 1). Les coordonnées *(i, j)* (démarrant à l'indice 0) s'obtiennent à partir de la position *pos* de la manière suivante :

```
i = (int)(pos / N);
j = (int)(pos % N);
```

La position *pos* est déterminée à partir de *(i, j)* par l'instruction : `pos = i*N+j;`

La valeur du tableau à la coordonnée *(i, j)* indique l'absence (0) ou la présence (1) d'une reine à cette position.

- `initGame(…)` : Construit le nœud initial et initialise la structure correspondant au monde modélisé pour le problème visé. Dans le cas du problème des 8 reines, l'état initial est un échiquier 8x8 vide (ou un tableau de 64 éléments rempli de 0). Dans le cas du problème de déplacement de cavalier, l'état initial est un échiquier où toutes les cases sont à 0 sauf celle où se trouve le cavalier.

- `evaluateBoard(…)` : renvoie le coût d'un état = le nombre de reines restant à placer (la distance du cavalier à sa position cible, pour le problème du cavalier..). Cette fonction retourne 0 lorsque l'état évalué est une solution.

- `isValidPosition(…)` : retourne vrai si la position testée est valide et atteignable en une action à partir de l'état du nœud courant. C'est ici que la connaissance à priori du problème/jeu est définie : 2 reines ne peuvent pas être positionnées sur la même ligne/colonne/diagonale ; le cavalier se déplace toujours de 2 cases sur un axe et 1 case sur l'autre axe.

- `getChildNode(…)` : retourne un nouveau nœud, dont l'état résulte de la simulation d'une action valide à partir du nœud parent. La structure *« Nœud »* à considérer ici est la même que celle utilisée pour l'implémentation d'un *« Maillon »* de la liste chainée : la structure `Item`. Cette fonction incrémente de 1 la profondeur du nœud fils et attache ce dernier au nœud parent afin de permettre le *backtracking* de la solution.

- `printBoard(…)` : affiche l'état du monde correspondant à un nœud du graphe.

3. Algorithme de recherche simple

Cette partie consiste à implémenter les algorithmes de recherches simple (largeur d'abord, profondeur d'abord) qui utiliseront les fonctions décrites précédemment. Pour rappel, une recherche en largeur d'abord s'implémente avec une File, tandis que la recherche en profondeur d'abord utilise une Pile.

L'essentiel du code de recherche en largeur d'abord est fourni dans le fichier `nqueens.c`.

a. Complétez-le pour qu'il fonctionne entièrement.
b. Ecrivez la fonction `dfs()` qui implémente une recherche en profondeur d'abord.
c. Comparez

4. Déplacement de cavalier

Cette partie vise à implémenter une fonction qui détermine le chemin d'un cavalier entre deux positions de l'échiquier.

- Copiez/dupliquez le fichier `nqueens.c` vers `knights.c`
- Ajoutez une entrée supplémentaire dans le Makefile

Modifiez les fonctions nécessaires dans le fichier `board.c` pour résoudre le problème de déplacement de cavalier. L'objectif est de déterminer la séquence de déplacements nécessaires pour se rendre à la dernière case (en bas à droite) de l'échiquier, en partant de la première case.

5. Liste avec priorité

Cette partie consiste à implémenter une version simple d'une liste avec priorité. L'implémentation consiste à modifier légèrement notre structure `Item`, et ajouter une fonction dédiée dans les fichiers `list.c` et `list.h` :

- Considérez l'attribut `f` de la structure `Item` pour y stocker le coût d'un nœud
- Implémentez la fonction `popBest(…)` qui retourne l'élément de coût minimal contenu dans la liste.

Cette structure de données nous permet à présent de résoudre les problèmes précédents avec des algorithmes de recherche plus efficaces, tels que UCS (Dijkstra) et A*.

6. Algorithme UCS

Pour finir ce TP, implémentez l'algorithme UCS (slide 40) en tenant compte des remarques suivantes :

- UCS utilise une liste avec priorité basé sur le coût (utilisez popBest !)
- Pour le problème considéré, le coût d'une action est constant et égal à 1.
- Le coût du nœud *n* correspond au coût depuis la racine jusqu'au nœud *n*. Lorsque le coût des actions est 1, le coût d'un nœud est égal à la profondeur du nœud..
- Déterminez le coût d'un nœud fils retourné par la fonction `getChildNode`. Stockez le coût dans son attribut `f`.
- Lorsqu'un nœud fils est ajouté à la liste, il remplace un élément correspondant au même état (même configuration de la planche de jeu, utilisez la fonction onList…) si son coût est inférieur à celui figurant dans la liste