UNIVERSITÉ PARIS DESCARTES

DUT Informatique - 1ère année
SDA - Structures de Données et Algorithmes

Equipe pédagogique
Marie-José Caraty, Denis Poitrenaud,
Julien Rossit, Camille Kurtz,
Jacques Alès-Bianchetti, Denis Jeanneau

PROJET
STRUCTURES DE DONNEES ET ALGORITHMES
Le taquin cylindrique

Le jeu du taquin inventé par Sam Loyd vers 1870 est à l'origine un damier (4 lignes et 4 colonnes) composé de 15 cases qui peuvent glisser (d'une case à la fois, à l'horizontal ou à la verticale) sur leur support grâce à une case vide. Les cases, numérotées de 1 à 15, sont initialement dans le désordre. Le but du jeu est de faire glisser les cases afin d'obtenir la liste ordonnée des nombres. Les nombres doivent être rangés par ordre croissant par lignes (de gauche à droite et de haut en bas) et la case vide doit être positionnée en bas à droite. Ce jeu présente de nombreuses variantes concernant par exemple la nature du puzzle (image), la taille du damier ou la position de la case vide. La variante utilisée dans le projet est le taquin cylindrique où une case peut glisser autour du cylindre (à gauche et à droite) et verticalement (en haut et en bas) d'une case à la fois. Le but du jeu est le même (obtenir liste ordonnée des nombres) mais il y a autant de solutions que de colonnes.

Le projet consiste à résoudre par programme un problème de taquin cylindrique modélisé par un damier de m lignes et n colonnes. Les ($m \times n$)-*1* cases sont numérotées de *1* à ($m \times n$)-*1*. Sur le damier initial, la case vide est située à une position quelconque. La trace de la solution comprendra (1) le nombre de lignes et de colonnes du damier, (2) le nombre de mouvements conduisant à la solution au problème et (3) du damier initial au damier but, la trace des mouvements successifs (nature du mouvement et damier résultant). Un algorithme (de recherche A*) de planification (domaine de l'Intelligence Artificielle) et des structures de données choisies pour l'implémentation vont nous permettre de résoudre le problème.

Le damier initial est décrit dans un fichier texte dont la première ligne indique les nombres de lignes (m) et de colonnes (n). Les lignes suivantes indiquent les m lignes du damier. La case vide, symbolisée par « # », peut être située à n'importe quelle case du damier.

```
2 3
  2  #  4
  5  1  3
```

Le damier sera traité comme une matrice d'entiers (tableau à 2 dimensions) de taille (m, n). Par convention, la position (*0*, *0*) est située en haut à gauche du damier. Vous développerez le composant correspondant à la structure de données représentant de telles matrices gérées en mémoire dynamique. Dans l'application développée, vous devez veiller à une bonne gestion de la mémoire dynamique.

Les mouvements (glissement des cases) seront repérés relativement au mouvement de la case vide. Les mouvements sont contraints par la position de la case vide dans le damier. Quatre mouvements au plus sont possibles : NORD, OUEST, SUD et EST.

Pour le damier donné en exemple, seuls trois mouvements sont possibles :

OUEST

#	2	4
5	1	3

SUD

2	1	4
5	#	3

EST

2	4	#
5	1	3

La solution d'un problème de taquin peut être représentée par une séquence d'états permettant de passer du damier initial au damier but (damier des chiffres ordonnés avec la case vide en bas à droite). Un état de la séquence correspond à une action de mouvement de la case vide ; il est caractérisé par un damier résultant d'un mouvement possible (NORD, OUEST, SUD, EST) appliqué à un damier précédent. Pour le premier état de la séquence, le damier est le damier initial, il ne correspond à aucun mouvement et c'est donc également le damier précédent. Pour l'état final de la séquence, le damier courant est l'un des damiers buts.

Une recherche exhaustive de la solution du problème de taquin est exclue. L'algorithme de recherche A* qui est à programmer consiste à explorer en parallèle un ensemble de séquences d'états (solutions partielles) ; l'exploration est guidée par une heuristique de recherche (f) fondée sur le coût (g) d'une solution partielle (séquence d'états allant de l'état initial à un état e) et sur l'estimation du coût minimum (h) pour passer de l'état e à l'un des états finaux/but.

L'algorithme utilisera deux listes L_{EAE} et L_{EE} qui permettent respectivement -de mémoriser les états à explorer et -de mémoriser les états explorés. La liste L_{EAE} contient les états en attente d'exploration et la liste L_{EE} permet de mémoriser l'ensemble des séquences d'états (solutions partielles) explorées. Parmi les états de L_{EAE}, on choisira d'explorer l'état ayant la valeur minimale de l'heuristique de recherche f. Une fois l'état sélectionné, son exploration consiste (1) à le mémoriser dans la liste L_{EE} (il fait partie d'une solution explorée), (2) à le supprimer de L_{EAE}, puis (3) à le dériver suivant chacun des mouvements possibles. Dans cette dernière étape, chacun des états dérivés (des mouvements) est ajouté à la liste L_{EAE} (états à explorer ultérieurement) à la condition que le

damier de l'état ne coïncide avec le damier d'aucun des états de L_{EAE} et de L_{EE}. Cette condition correspond à une détection de cycle dans la solution : un damier déjà exploré ou à explorer n'est plus à considérer pour la solution. L'algorithme à programmer est donné à la page 6.

Caractérisation d'un état

Pour guider votre implémentation, un état sera caractérisé par :

(1) un damier : une *matrice*(*m, n*),

(2) un mouvement : NORD, OUEST, SUD ou EST,

(3) un index de l'état précédent dans L_{EE} (lien vers le damier précédent auquel est appliqué le mouvement pour produire le damier de l'état),

(4) les deux valeurs (*g* et *h*) composant l'heuristique de recherche *f* (*f=g+h*). Vous pourrez ajouter toute information facilitant l'initialisation (cf. état initial) et favorisant l'efficacité du traitement.

Heuristique de recherche

L'heuristique de recherche *f(e)*, associée à un état *e*, est la somme de deux informations d'heuristique *g(e)* et *h(e)* :

$$f(e) = g(e) + h(e)$$

g(e) est le coût de la meilleure séquence d'états (solution partielle) connue pour aller de l'état initial à l'état *e*. Ce coût est le nombre de mouvements pour aller de l'état initial à l'état *e*.

h(e) est l'estimation du coût de la séquence d'états allant de l'état *e* à l'état final le plus proche. Ce dernier sera celui dont le nombre de cases mal placées du damier à l'état *e* est le plus petit. Le coût sera égal à ce nombre de cases mal placées.

f(e) est ainsi une estimation du coût de la meilleure solution (séquence d'états) passant par *e*.

Implémentation

Pour l'implémentation des listes L_{EAE} et L_{EE}, vous aurez à choisir les structures de données, les plus adaptées, vues en cours et en travaux pratiques.

Vous utiliserez une autre structure de donnée, à choisir, pour mémoriser la séquence d'états de la solution trouvée et permettre simplement l'affichage de la trace de la solution.

Cadre du développement logiciel

Le projet est à réaliser en binôme (les monômes ne sont pas autorisés). Les membres d'un binôme seront de préférence du même groupe. Vous devez programmer et tester l'application demandée.

L'un des objectifs de ce projet est la réutilisation des composants techniques (pile, file ou liste), vus en cours ou développés en travaux pratiques. Vous choisirez les composants les plus adaptés à la résolution des problèmes posés par l'application.

Le développement logiciel se fera par cycle de type agile au moyen de *sprints* (5 au total). Chaque *Sprint#n* ($1 \le n \le 5$) est défini par une spécification et un test par redirection correspondant à un jeu de données de test (JDT) (*in#n.txt*) et ses résultats de référence attendus (*ref#n.txt*). Les différents *sprints* seront donnés dans le cours Projet. Pour un *Sprint#n* donné, si le résultat de votre application (*out#n.txt*) coïncide avec le résultat de référence (*ref#n.txt*), votre application est 0-

défaut et le *Sprint#n* est validé. Vous passerez alors au développement du *Sprint#n+1* suivant. Sous Visual, vous recopierez au niveau des répertoires tous les fichiers sources nécessaires du projet du *Sprint#n* dans le nouveau projet du *Sprint#n+1*.

Votre travail sera évalué à partir :

(1) d'une recette (cf. spécifications p.4),
(2) d'un dossier de développement logiciel (cf. spécifications p. 5).

Tests et recette de l'application

Vous trouverez sous COMMUN, pour ($1 \le n \le 5$) un taquin (*in#n.txt*) et sa sortie de référence attendue (*ref#n.txt*) pour chacun des cinq sprints qui vous serviront aux phases de développement. Si vous avez bien respecté les spécifications, votre application devra donner les mêmes résultats que les sorties de référence.

Votre programme doit pouvoir résoudre tout taquin « résolvable » représenté par un damier *(m, n)* avec $m>1$, $n>1$ et $(m \times n) \le 16$. Les taquins de recette admettent des solutions dont la séquence d'états (de grande taille) est calibrée pour que l'algorithme rende un résultat dans un temps acceptable (vos choix des structures de données seront alors déterminants).

La **semaine du 7 janvier 2019**, vous passerez la recette de votre projet lors de votre séance **SDA3**. Il s'agit d'un test de recette qui testera le sprint de plus haut niveau atteint au cours de votre développement avec un nouveau JDT. Vous aurez à compiler et à exécuter votre programme sur le JDT de recette (type *in.txt*) qui vous sera communiqué. Votre enseignant vérifiera automatiquement le 0-défaut de votre application en comparant le fichier *out#n.txt* de votre application au fichier de référence *ref#n.txt*. Si les deux fichiers coïncident, le sprint#n sera validé à la recette.

Pour la comparaison des solutions, vous devez créer le fichier texte correspondant à la trace d'exécution demandée. Le format du résultat du sprint#5 est le suivant :

```
Damier : 2 lignes, 3 colonnes
Solution en 5 mouvements
  2  #  4
  5  1  3
SUD
  2  1  4
  5  #  3
EST
  2  1  4
  5  3  #
NORD
  2  1  #
  5  3  4
OUEST
  2  #  1
  5  3  4
SUD
  2  3  1
  5  #  4
```

Ligne 1 : le nombre de lignes et de colonnes du damier.

Ligne 2 : le nombre de mouvements.

A partir de la ligne 3, le damier initial à m lignes et n colonnes (nombres cadrés à droite sur 3 caractères), la case vide est symbolisée par « # »

Puis, pour les états de la solution :

(1) le mouvement (en majuscule),

(2) le damier (résultant).

Dossier du développement logiciel

Vous devez porter une attention particulière à la rédaction de votre dossier. Sa qualité est déterminante pour l'évaluation de votre travail. La composition de votre dossier doit être la suivante :

- Une page de garde indiquant le nom et le groupe des membres du binôme, l'objet du dossier. Une table des matières de l'ensemble du dossier (incluant les annexes) avec une pagination continue sur l'ensemble du dossier.
- Une présentation du projet en une page. Il s'agit de présenter les fonctionnalités de l'application et de décrire les entrées et sorties attendues.
- Le graphe de dépendance des fichiers sources de votre application. Tous les composants (réutilisés ou développés) de l'application devront figurer sur le graphe (cf. Cours 4).
- L'organisation des tests de l'application. Pour les tests que vous concevrez, vous donnerez les jeux de données de test (JDT) et les solutions en annexe.
- Un bilan du projet. Ce bilan comprend i) le bilan de validation de votre application à partir des résultats des tests effectués, ii) les difficultés rencontrées, iii) ce qui est réussi et ce qui peut être amélioré.

En annexe au dossier :

- le listing complet de vos sources. La présentation doit suivre l'ordre suivant : (i) le point d'entrée de l'application (programme principal), (ii) par ordre alphabétique, la liste des composants utilisés par l'application (tous les fichiers de spécification (.h), suivi de tous les fichiers corps (.cpp)).
- les jeux de données de test (JDT) de votre application (fichiers textes des taquins testés). Les traces d'exécution du programme associées aux JDT. Tout JDT doit être commenté de manière à illustrer le test effectué.
  Remarque **:** Il n'est pas demandé dans ce projet de tester la validité des taquins initiaux (considérés sans erreur). Les taquins de développement font partie de vos tests. A vous de compléter les tests.

Recommandations

Le code source doit être structuré en composants pour une compilation séparée.

Les codes sources doivent absolument être commentés (cartouches, structures de données et attributs, fonctions) suivant le formalisme Javadoc. Les conventions de nommage données en cours doivent être respectées. Les préconditions des fonctions doivent obligatoirement être documentées et testées par assertion dans le code.

Vous veillerez à ce que votre programme ne présente aucune fuite mémoire.

Suivez toutes les spécifications données sous peine de pénalisation :

(1) convention du référentiel,
(2) ordre d'exploration des états (cf. algorithme),
(3) format du fichier texte de la solution.

Date limite de remise de projet

La **date limite** de remise de projet est fixée au lundi **14 janvier 2019**.

- Déposez la version papier (brochée) de votre projet au secrétariat.
- Déposez dans le puits (pour chacun des répertoires de groupe du binôme SDA/Gr?) l'archive (.zip) de nom « Nom1Gr?Nom2Gr?.zip » constituée (1) du dossier de développement (.pdf) et (2) du source (sans l'exécutable .exe) du *sprint* de plus haut niveau (#n) que vous avez validé lors de la recette et éventuellement (3) du source du sprint (#n+1) en cours de développement.


```
initialiser étatInitial
insérer étatInitial dans LEAE
solutionTrouvée <- faux

tant_que (LEAE n'est pas vide)
    // Choisir l'état à explorer (étatCourant). Au premier passage : étatCourant = étatInitial
    si (un unique état (e) de LEAE est de valeur f minimale)
        étatCourant <- e
    sinon
        si (parmi les états de LEAE de même valeur f minimale,
            un unique état (e) de LEAE est de valeur h minimale)
            étatCourant <- e
        sinon
            étatCourant <- (l'état le plus récemment inséré dans LEAE
                            parmi les états de LEAE de même valeur f
                            minimale et de même valeur h minimale)
        fin_si
    fin_si

    si (étatCourant est l'état final) // le damier de étatCourant est le damier but
        solutionTrouvée <- vrai
        sortir de la boucle tant_que
    sinon    // explorer l'état courant
        insérer étatCourant dans LEE
        supprimer étatCourant de LEAE
        pour_tout (mouvement possible associé à étatCourant)
            // mouvement possible relatif à damier de étatCourant
            // ordre d'exploration (obligatoire) NORD, OUEST, SUD, EST
            initialiser un étatDérivé // d'après le mouvement
            si (damier de étatDérivé n'est égal au damier
                d'aucun état de LEAE et de LEE)
                insérer étatDérivé dans LEAE
            fin_si
        fin_pour_tout
    fin_si
fin_tant_que

si (solutionTrouvée = vrai)
    mémoriser la séquence solution du taquin
    // choisir la structure de données adaptée pour mémoriser les états de la séquence solution
    afficher la trace de la solution
    // cf. Tests et recette de l'application (format solution)
sinon
    afficher qu'il n'y a pas de solution au taquin
fin_si
```

Algorithme de résolution d'un taquin