UNIVERSITÉ PARIS DESCARTES

SDA - Structures de Données et Algorithmes

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

Cours « Projet »
Analyse du projet
Le taquin

Recette en séance de SDA3, la semaine du 7 janvier 2019
Remise des dossiers (version papier) et dépôt des archives (électroniques) le lundi 14 janvier 2019 (au secrétariat)

Sommaire

1. PROBLEME DU TAQUIN

Expression de la solution

Problème

2	#	4
5	1	3

1	2	3
4	5	#

3 états finaux

2	3	1
5	#	4

3	1	2
#	4	5

Particularité du taquin cylindrique, *autant de solutions que de colonnes*

Mouvement repéré relativement à la case vide notée **#** suivant les directions (**NORD, OUEST, SUD, EST**) : également l'ordre d'exploration des solutions

Solution

2	#	4
5	1	3

état initial

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

état but

Solution : séquence d'états allant du damier initial au damier but

1. PROBLEME DU TAQUIN

Le principe de résolution

En Intelligence artificielle,

l'algorithme A* est une variante de **l'algorithme de Dijkstra** connu pour trouver les plus courts chemins dans un graphe depuis un état initial jusqu'à un état final

Son principe

utiliser une **évaluation heuristique** pour chaque **nœud** pour estimer le meilleur chemin passant par ce nœud et **accélérer ainsi la recherche de l'état final**

Notion d'état - Informations caractéristiques

un état e

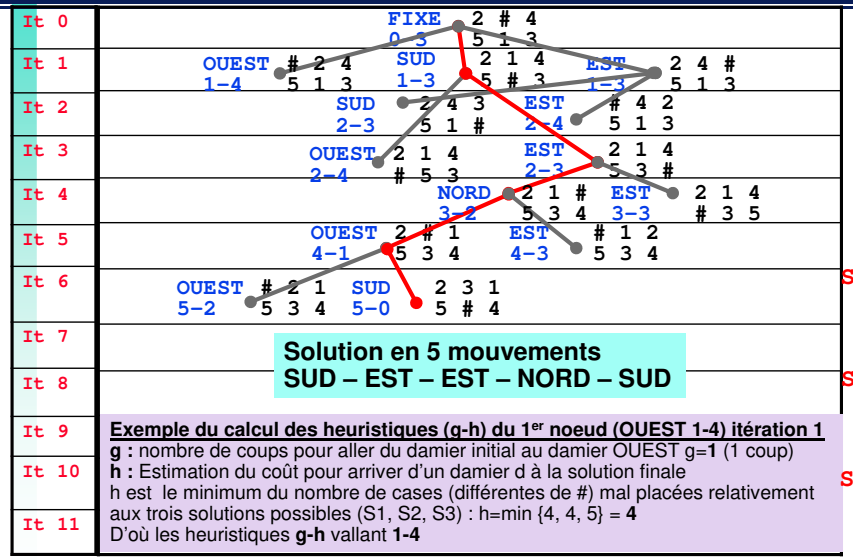
deux heuristiques

2 1 4 / 5 # 3	damier résultant d
SUD	du mouvement (du #)
2 # 4 / 5 1 3	à partir du damier
g	Nombre de coups pour passer du damier initial à d
h	Estimation du coût pour passer de d à l'un des états finaux **Estimation** : le nombre de cases numérotées mal placées (# exclus)

Informations caractéristiques d'un état

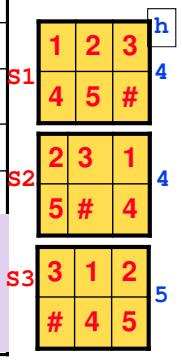
Arbre des solutions du taquin

2 # 4 / 5 1 3



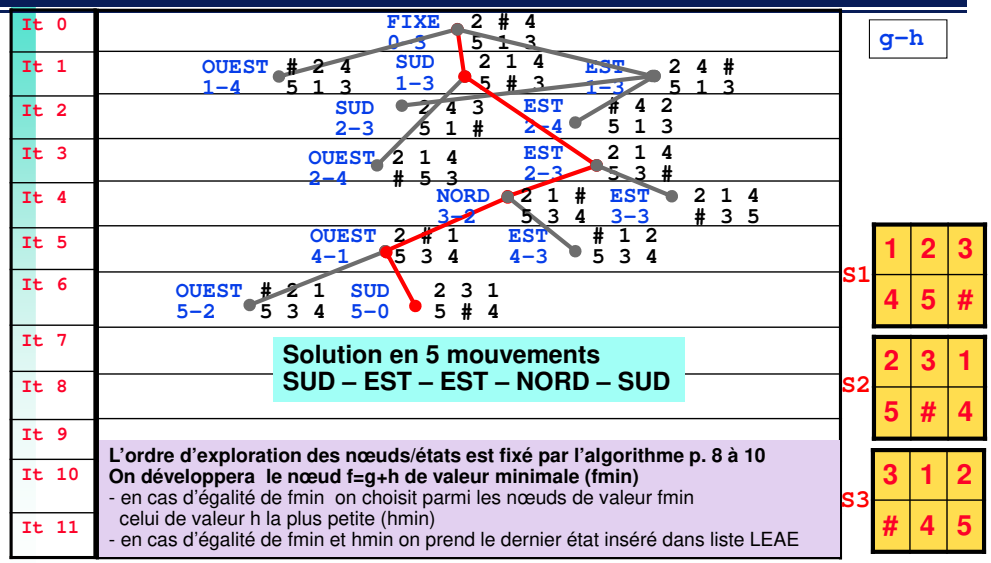
Exemple du calcul des heuristiques (g-h) du 1er noeud (OUEST 1-4) itération 1
g : nombre de coups pour aller du damier initial au damier OUEST g=**1** (1 coup)
h : Estimation du coût pour arriver d'un damier d à la solution finale
h est le minimum du nombre de cases (différentes de #) mal placées relativement aux trois solutions possibles (S1, S2, S3) : h=min {4, 4, 5} = **4**
D'où les heuristiques **g-h** vallant **1-4**

Trois solutions



Arbre des solutions du taquin

2 # 4 / 5 1 3



L'ordre d'exploration des nœuds/états est fixé par l'algorithme p. 8 à 10
On développera le nœud f=g+h de valeur minimale (fmin)
- en cas d'égalité de fmin on choisit parmi les nœuds de valeur fmin celui de valeur h la plus petite (hmin)
- en cas d'égalité de fmin et hmin on prend le dernier état inséré dans liste LEAE

Algorithme A* appliqué au problème du taquin (1/3)

```
Initialiser étatInitial
Insérer étatInitial dans LEAE    // liste des états à explorer
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 g minimale)
      étatCourant ← e
   sinon
      si (parmi les états de LEAE de même valeur g 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 g
                       minimale et de même valeur h minimale)
      fin_si
   fin_si
```

Algorithme du taquin (2/3)

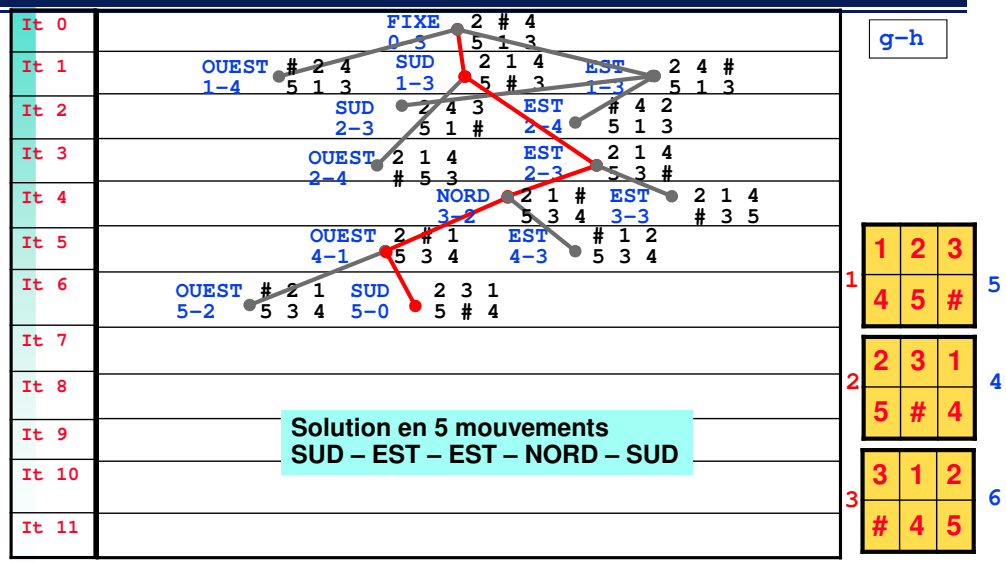
```
si (étatCourant est l'état final) // le damier de etatCourant est le but
   solutionTrouvée ← vrai
   sortir de la boucle tant_que
sinon     // explorer l'état courant
   insérer étatCourant dans LEE
   supprimer étatCourant de LEAE
   pour_tout (mouvement possible associé à étatCourant)
      // mouvement possible relatif à damier de étatCourant
      // impérativement considérés dans l'ordre 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
```


Algorithme du taquin (3/3)

```
si (solutionTrouvée = vrai)
   mémoriser la séquence solution du taquin
   // choisir la structure de données adaptée pour mémoriser les états de la 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
```


Arbre des solutions du taquin

2	#	4
5	1	3



g-h

1	2	3
4	5	#

1 — 5

2	3	1
5	#	4

2 — 4

3	1	2
#	4	5

3 — 6

Exemple d'exécution de l'algorithme (1/3)

2	#	4
5	1	3

Initialisation avant la boucle `tant_que`

Exemple d'exécution de l'algorithme (2/3)

2	#	4
5	1	3

Cf. Trace de l'éxécution de `ref#4.txt`

2ème itération de la boucle `tant_que`

LEAE

# 2 4 5 1 3 OUEST 2 # 4 5 1 3 g=1 h=4 *index 0*	2 1 4 5 # 3 SUD 2 # 4 5 1 3 g=1 h=3 *index 1*	2 4 3 5 1 # SUD 2 4 # 5 1 3 g=2 h=3 *index 2*	# 4 2 5 1 3 EST 2 4 # 5 1 3 g=2 h=4 *index 3*

--->

LEE

2 # 4 5 1 3 FIXE 2 # 4 5 1 3 g=0 h=3 *index 0*	2 4 # 5 1 3 EST 2 # 4 5 1 3 g=1 h=3 *index 1*

--->

3ème itération de la boucle `tant_que`

LEAE

# 2 4 5 1 3 OUEST 2 # 4 5 1 3 g=1 h=4 *index 0*	2 4 3 5 1 # SUD 2 4 # 5 1 3 g=2 h=3 *index 2*	# 4 2 5 1 3 EST 2 4 # 5 1 3 g=2 h=4 *index 3*	2 1 4 # 5 3 OUEST 2 1 4 5 # 3 g=2 h=4 *index 4*	2 1 4 5 3 # EST 2 1 4 5 # 3 g=2 h=3 *index 5*

--->

LEE

2 # 4 5 1 3 FIXE 2 # 4 5 1 3 g=0 h=3 *index 0*	2 4 # 5 1 3 EST 2 # 4 5 1 3 g=1 h=3 *index 1*	2 1 4 5 # 3 SUD 2 # 4 5 1 3 g=1 h=3 *index 2*

--->

Solution à extraire de LEE (3/3)

2	#	4
5	1	3

LEE

2 # 4 5 1 3 FIXE 2 # 4 5 1 3 g=0 h=5 *index 0*	2 4 # 5 1 3 EST 2 # 4 5 1 3 g=1 h=3 *index 1*	2 1 4 5 # 3 SUD 2 4 # 5 1 3 g=1 h=3 *index 2*	2 1 4 5 3 # EST 2 1 4 5 # 3 g=2 h=3 *index 3*	2 1 # 5 3 4 NORD 2 1 4 5 3 # g=3 h=2 *index 4*	2 # 1 5 3 4 OUEST 2 1 # 5 3 4 g=4 h=1 *index 5*	2 3 1 5 # 4 SUD 2 # 1 5 3 4 g=5 h=0 *index 6*

Dans cette implémentation,
l'état but n'est pas ajouté à liste LEE
Il est mémorisé dans la structure de données qui permet d'afficher simplement la solution

Quel est le TDA le mieux adapté ?

Implémentation et optimisation mémoire

1) **Gestion des damiers (tableaux à 2 dimensions) en mémoire dynamique**
   Composant Tableau2D à développer

2) **Le damier précédent est repéré par un index dans LEE**
   Par algorithme, tout damier est dérivé à partir d'un damier de LEE

Coût mémoire d'un état

Attributs du type `Tableau2D`	damier résultant
Une variable de type énuméré	mouvement
Un index dans LEE	damier précédent
entier non signé	nombre de coups g de l'état initial à e
entier non signé	heuristique h de e à l'état but
deux entiers non signés	position de la case vide dans le damier résultant

Architecture logicielle (1/4)

Un exemple d'Architecture

Composant `Taquin` (`.h` et `.cpp`) des parties de jeu de taquin

Composant `Etat` (`.h` et `.cpp`) des états du jeu de taquin

Composant `Tab2D` (`.h` et `.cpp`) des tableaux bidimensionnels (`m` lignes, `n` colonnes) d'éléments de type (généralisé) Item (à spécialiser pour l'application suivant le type des cases du damier)

Tous les composants du cours/TD/TP (ré-)utilisés
(pile, liste, chaine, …)
avec les items spécialisés

L'application (`le main` : point d'entrée de l'application)

Remarque : Toute application « monolithique » sera fortement sanctionnée

Architecture logicielle - `Taquin` (2/4)

Composant `Taquin` (`.h` et `.cpp`) des parties de jeu de taquin

Sa donnée concrète (structure nommée `Taquin`) **inclut**

- la mémorisation des états nécessaire à la planification de l'exploration des solutions (les listes LEE et LEAE
  LEE (liste des états explorés), LEAE (liste des états à explorer)
- les nombres de lignes et de colonnes caractéristiques du damier considéré

Spécification

- ✓ `initialiser` Allouer la variable `t` de type `Taquin`
  i) Initialiser les listes LEE et LEAE
  ii) lire les nombres de lignes et de colonnes à partir du flot d'entrée
  iii) créer l'état initial à partir de ce même flot
- ✓ `jouer` Jouer dans la configuration du taquin (itération de l'algorithme)
  Indiquer si la solution est alors trouvée
- ✓ `lire/afficher` lire et afficher un taquin en utilisant les flots standards
  Toute fonction utile (detruire, …)

Architecture logicielle - `Etat` (3/4)

Composant `Etat` (`.h` et `.cpp`) des états du jeu de taquin

Spécification

- ✓ `initialiser/detruire` Allouer/désallouer la variable `e` de type `Etat`
- ✓ `afficher` Afficher un état
- ✓ ….

Attention : `lire` et `afficher` un état en utilisant les flots standards

Architecture logicielle - `Tab2D` (4/4)

Composant `Tab2D` (`.h` et `.cpp`) des tableaux bidimensionnels de taille (m, n) d'éléments de type Item (spécialisé en `unsigned int`)

Sa donnée concrète (structure nommée `Tab2D`) **inclut**

- le tableau bidimensionnel géré en mémoire dynamique
- le nombre de lignes
- le nombre de colonnes

Spécification

- ✓ `initialiser` Initialiser la variable t de type Tab2D (tous ses attributs) entre autres initialiser à vide le conteneur (le tableau 2D), l'allouer en mémoire dynamique
- ✓ `detruire` Désallouer les attributs de la variable t de type Tab2D faisant un lien avec la mémoire dynamique
- ✓ …. Lecture et affichage d'une matrice en utilisant les flots standards

Spécialisation du type Item

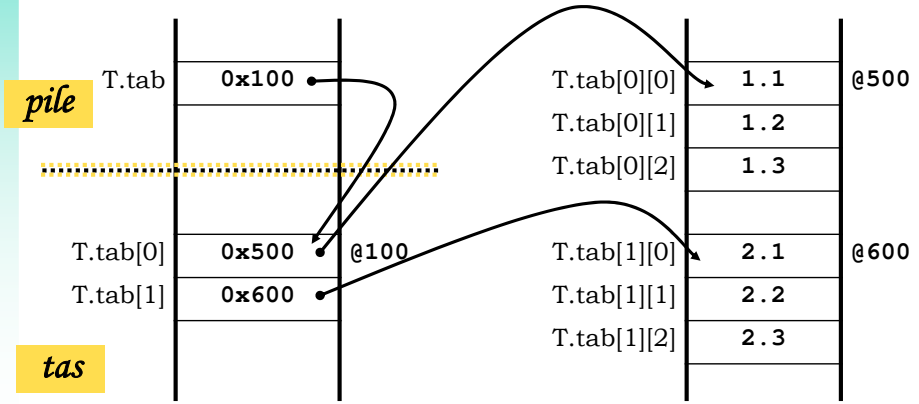
……

Tableau 2-D stocké en mémoire dynamique

Représentation du damier

Soit le tableau T(2, 3) à 2 lignes et 3 colonnes $T=\begin{bmatrix} 1.1 & 1.2 & 1.3 \\ 2.1 & 2.2 & 2.3 \end{bmatrix}$

Choix de stockage du tableau bidimensionnel T en mémoire dynamique



Définition de la donnée concrète

Définir une donnée concrète (structure de données) Tab2D
représentant la **famille** des **tableaux de taille (m, n)**
$\forall$ (m,n) $\in$ IN^{*2} (m lignes et n colonnes)
stockées en mémoire dynamique

```
struct Tab2D {
   Item tab**; // adresse du tableau bidimensionnel
               // en mémoire dynamique
   int nbL;    // nombre de lignes de la matrice
   int nbC;    // nombre de colonnes de la matrice
};
```


Développement de l'application par Sprints

Cycles de développement
appliqués à une fonctionnalité incrémentale
de l'application

Développement par Sprints (1/3)

Phase de développement

Deux fichiers `in.txt` et `ref#n.txt` sont donnés pour un `Sprint#n` sur COMMUN (1 ≤ n ≤ 5).
Le premier fichier correspond aux entrées de l'application testées appelées le Jeu de Données de Test (JDT),
le second correspond aux sorties attendues de l'application appelées le résultat de référence.

En fin de la phase de développement du `Sprint#n`, vous comparerez les sorties de votre programme (`run.txt`) résultant de l'exécution de votre application à partir du jeu de données de test `in.txt` avec le résultat de référence donné (`ref#n.txt`)

Pour comparer les deux solutions, vous utiliserez la comparaison de fichiers **Diff.jar** (archive de l'exécutable d'un programme Java de test de différence de fichiers)
si les deux fichiers (`run.txt` et `ref#n.txt`) sont identiques,
votre application est 0-défaut pour ce test,
vous pouvez passer au développement du Sprint suivant
sinon corrigez les erreurs de votre application

Développement par Sprints (2/3)

En référence à la programmation agile pour lequel le cycle de développement (analyse fonctionnelle, spécification, codage, test) est court, des Sprints vous sont proposés pour le développement de votre projet.

Attention (1) Chaque Sprint validé donne lieu à un développement matérialisé par un projet et représente une évolution dont vous devez garder la trace.
Lorsque vous passez à un autre Sprint, créez un nouveau projet où vous recopierez (sans partage de sources) au niveau des répertoires toutes les entités logicielles (validées) du précédent projet qui sont nécessaires au nouveau sprint ou à y adapter.

Attention (2) Dans le cas où votre programme ne fonctionne pas sur l'ensemble des fonctionnalités demandées
Les projets liés aux Sprints sont indispensables pour montrer l'état de développement de votre logiciel.
Chaque sprint validé correspond à un très bon travail que vous pouvez démontrer (tests réussis) et qui vous rapportera des points.

Développement par Sprints (3/3)

1. **Dans le cas où votre programme ne fonctionne pas sur l'ensemble des fonctionnalités demandées**

   Dans le dossier de développement logiciel demandé, vous donnerez les listings et traces d'exécution correspondant au sprint de plus haut niveau que vous avez passé avec succès.

2. **Dans le cas où votre programme fonctionne sur l'ensemble des fonctionnalités demandées**

   Dans le dossier de développement logiciel demandé, les listings et les traces d'exécution des deux derniers sprints sont à donner.

Développement par sprints – Sprint #1 (1/3)

Créez un projet de nom `Sprint1` au sein d'une `Solution/Projet`

Premier Sprint

- **Analyse fonctionnelle :** A partir d'un fichier texte décrivant le taquin (cf. `in.txt` sur COMMUN), lire et afficher le taquin lu (à partir du `tableau2D` créé).
- **Spécification :** Développez le composant (.h et .cpp) **`Tableau2D`** ainsi que le programme de test (**`main`**) correspondant à l'analyse fonctionnelle.
- **Codage :** Codez (a) les fonctions `initialiser`, `detruire`, `lire` et `afficher` de Tableau2D (spécialisez le type `Item` utilisé par `Tableau2D`) et (b) le test de bonne initialisation d'un damier de type Tableau2D à partir d'un flot standard.
- **Test :** Testez votre application par redirection des entrées à partir du fichier `in.txt (JDT du Sprint#1)` et sa sortie vers le fichier `run.txt`. Comparez votre fichier `run.txt` au fichier des résultats de référence `ref#1.txt`: si les deux fichiers coïncident, votre `Sprint#1` est validé, vous pouvez passer au `Sprint#2` sinon corrigez les erreurs.

Sprint#1 – Prototypage des fonctions (2/3)

Sprint#1

```
// Allouer en mémoire dynamique un Tableau2D
void initialiser(Tab2D& m, unsigned int nbL,
                 unsigned int nbC);

// Desallouer un Tableau2D
void detruire(Tab2D& m);

// Lire un Tableau2D
void lire(Tab2D& m);

// Afficher un Tableau2D
void afficher(const Tab2D& m);
```


Sprint#1 – Résultat de référence `ref#1.txt` (3/3)

2	#	4
5	1	3

```
Damier : 2 lignes, 3 colonnes
  2  #  4
  5  1  3
```

Développement par sprints – Sprint #2 (1/3)

Créez un projet de nom `Sprint2` au sein de la même Solution que le `Sprint1`.
Au niveau des répertoires des projets, recopiez (sans partage de sources) **les sources utiles du `Sprint1` dans le répertoire des sources de `Sprint2`. Adaptez les sources au `Sprint2`.**

Deuxième Sprint

- **Analyse fonctionnelle :** Donner la 1ère itération de l'algorithme (cf. Transp. 6 à 8). On ne calculera ni le nombre de coups (g), ni l'heuristique (h).
- **Spécification :** Développez les composants (.h et .cpp) **`Etat`** et `Taquin` ainsi que le programme de test de ces composants correspondant à l'analyse fonctionnelle.
- **Codage :** Codez (a) le type `Etat`, le type énuméré `Mouvement` et la fonction `afficher` un état dans un flot standard pour trace. Codez(b) les fonctions `initialiser` et `jouer` de Taquin sans prendre en compte l'heuristique h, la fonction `afficher` de Taquin (affichant les deux listes `LEE` et `LEAE` suivant le format `ref#2.txt, Transp. 28`). Codez toutes les fonctions utiles à l'itération.
- **Test :** Testez votre application par redirection des entrées à partir du fichier `in.txt (JDT du Sprint#2)` et sa sortie vers le fichier `run.txt`. Comparez votre fichier `run.txt` au fichier des résultats de référence `ref#2.txt` : si les deux fichiers coïncident, votre `Sprint#2` est validé, vous pouvez passer au `Sprint#3` sinon corrigez les erreurs.

Sprint#2 – Prototypage des fonctions (2/3)

Sprint#2

```
// Afficher un état du taquin
void afficher(const Etat& e);

// Créer le jeu de taquin avec l'état initial
void initialiser(Taquin& t);

// Itération de l'algorithme de recherche
void jouer(Taquin& t);

// Afficher le contenu des listes adev et dev
void afficher(Taquin& t);
```


Sprint#2 – Résultat de référence `ref#2.txt` (3/3)

```
Damier : 2 lignes, 3 colonnes
Iteration 0
*** LEE - long : 0

*** LEAE - long : 1
  2  #  4
  5  1  3
Fin iteration 0


Iteration 1
*** LEE - long : 1
  2  #  4
  5  1  3

*** LEAE - long : 3
EST
  2  4  #
  5  1  3
SUD
  2  1  4
  5  #  3
```

```
OUEST
  #  2  4
  5  1  3
Fin iteration 1
```

2	#	4
5	1	3

Développement par sprints – Sprint #3 (1/4)

Créez un projet de nom `Sprint3` au sein de la même Solution que les `Sprints 1` et `2`.
Au niveau des répertoires des projets, recopiez (sans partage de sources) **les sources utiles du `Sprint2` dans le répertoire des sources de `Sprint3`. Adaptez les sources au `Sprint3`.**

Troisième Sprint

- **Analyse fonctionnelle :** Donner une partie de l'arbre des solutions obtenu à partir de l'algorithme en prenant l'heuristique h=0. On se limitera aux 100 premières itérations de l'algorithme, sauf si l'état final est atteint avant.
- **Spécification :** Développez les composants (.h et .cpp) **`Etat`** et `Taquin` ainsi que le programme de test de ces composants correspondant à l'analyse fonctionnelle.
- **Codage :** Codez la fonction `appartient` qui vérifie si un état a déjà été développé, la fonction `but` qui vérifie si l'état final est atteint, la fonction `afficher` de Taquin (affichant les 2 listes `LEE` et `LEAE` avec les valeurs de `g` et `h` suivant le format `ref#3.txt`). Codez toutes les fonctions utiles à l'itération.
- **Test :** Testez votre application par redirection des entrées à partir du fichier `in.txt (JDT du Sprint#3)` et sa sortie vers le fichier `run.txt`. Comparez votre fichier `run.txt` au fichier des résultats de référence `ref#3.txt` : si les deux fichiers coïncident, votre `Sprint#3` est validé, vous pouvez passer au `Sprint#4` sinon corrigez les erreurs.

Sprint#3 – Prototypage des fonctions (2/4)

Sprint#3

```
// renvoie vrai si l'état existe déjà dans le taquin
bool appartient(const Etat& ef, Taquin& t);

// renvoie vrai s'il s'agir de l'état final, faux sinon
bool but(const Etat& e);

// affiche un état du taquin
void afficher(const Etat& e);

// itération de l'algorithme de recherche
// renvoie vrai si la solution a été trouvé, faux sinon
bool jouer(Taquin& t);

// affiche le contenu des listes adev et dev
void afficher(Taquin& t);
```


Sprint#3 – Résultat de référence `ref#3.txt` (3/4)

Début

```
Damier : 2 lignes, 3 colonnes
Iteration 0
*** LEE - long : 0

*** LEAE - long : 1
  2  #  4
  5  1  3
f=g+h=0+0=0
Fin iteration 0


Iteration 1
*** LEE - long : 1
  2  #  4
  5  1  3
f=g+h=0+0=0

*** LEAE - long : 3
EST
  2  4  #
  5  1  3
f=g+h=1+0=1
```

```
  SUD
  2  1  4
  5  #  3
f=g+h=1+0=1
OUEST
  #  2  4
  5  1  3
f=g+h=1+0=1
Fin iteration 1


Iteration 2
*** LEE - long : 2
  2  #  4
  5  1  3
f=g+h=0+0=0
EST
  2  4  #
  5  1  3
f=g+h=1+0=1

*** LEAE - long : 4
```

2	#	4
5	1	3

Sprint#3 – Résultat de référence `ref#3.txt` (4/4)

```
NORD
  4  #  3
  5  2  1
f=g+h=5+0=5
EST
  4  2  3
  1  #  5
f=g+h=5+0=5
NORD
  #  2  3
  4  1  5
f=g+h=5+0=5
EST
  4  1  2
  5  3  #
f=g+h=5+0=5
OUEST
  4  1  2
  #  5  3
f=g+h=5+0=5
EST
  5  4  #
  1  2  3
f=g+h=5+0=5
```

```
OUEST
  #  5  4
  1  2  3
f=g+h=5+0=5
NORD
  5  2  #
  1  3  4
f=g+h=5+0=5
EST
  #  2  5
  3  1  4
f=g+h=5+0=5
OUEST
  5  #  2
  3  1  4
f=g+h=5+0=5
OUEST
  5  2  4
  #  3  1
f=g+h=5+0=5
NORD
  5  #  4
  3  2  1
f=g+h=5+0=5
Fin iteration 61
```

2	#	4
5	1	3

Fin

Développement par sprints – Sprint #4 (1/4)

Créez un projet de nom `Sprint4` au sein de la même Solution que les `Sprints 1` à `3`. Au niveau des répertoires des projets, recopiez (sans partage de sources) **les sources utiles du `Sprint3` dans le répertoire des sources de `Sprint4`. Adaptez les sources au `Sprint4`.**

Quatrième Sprint

- **Analyse fonctionnelle :** Donner la trace de l'arbre des solutions, obtenu à partir de l'algorithme de recherche utilisant l'heuristique (algorithme donné dans le projet) sans l'affichage de la solution.
- **Spécification :** Complétez la fonction **`jouer`** de Taquin en intégrant l'heuristique ainsi que le programme de test (**`main`**) correspondant à l'analyse fonctionnelle précédente.
- **Codage :** a) Codez la fonction heuristique, modifiez la fonction jouer de Taquin et la fonction `afficher` de Taquin (affichant seulement la liste `LEAE` avec les valeurs de g et h suivant le format `ref#4.txt`) et b) ajouter toutes les fonctions utiles
- **Test :** Testez votre application par redirection des entrées à partir du fichier `in.txt (JDT du Sprint#4)` et sa sortie vers le fichier `run.txt`. Comparez votre fichier `run.txt` au fichier des résultats de référence `ref#4.txt` : si les deux fichiers coïncident, votre `Sprint#4` est validé, vous pouvez passer au `Sprint#5` sinon corrigez les erreurs.

Sprint#4 – Prototypage des fonctions (2/4)

Sprint#4

```
// calcul de l'heuristique
unsigned int heuristique(Etat& e);
```


Sprint#4 – Résultat de référence `ref#4.txt` (3/4)

```
Damier : 2 lignes, 3 colonnes
Iteration 0
*** LEAE - long : 1
  2  #  4
  5  1  3
f=g+h=0+3=3
Fin iteration 0

Iteration 1
*** LEAE - long : 3
EST
  2  4  #
  5  1  3
f=g+h=1+3=4
SUD
  2  1  4
  5  #  3
f=g+h=1+3=4
OUEST
  #  2  4
  5  1  3
f=g+h=1+4=5
Fin iteration 1
```

```
Iteration 2
*** LEAE - long : 4
EST
  #  4  2
  5  1  3
f=g+h=2+4=6
SUD
  2  4  3
  5  1  #
f=g+h=2+3=5
SUD
  2  1  4
  5  #  3
f=g+h=1+3=4
OUEST
  #  2  4
  5  1  3
f=g+h=1+4=5
Fin iteration 2
```

2	#	4
5	1	3

Début

Sprint#4 – Résultat de référence `ref#4.txt` (4/4)

```
Iteration 6
*** LEAE - long : 8
SUD
  2  3  1
  5  #  4
f=g+h=5+0=5
OUEST
  #  2  1
  5  3  4
f=g+h=5+2=7
EST
  #  1  2
  5  3  4
f=g+h=4+3=7
EST
  2  1  4
  #  3  5
f=g+h=3+3=6
OUEST
  2  1  4
  #  5  3
f=g+h=2+4=6
```

```
  EST
  #  4  2
  5  1  3
f=g+h=2+4=6
SUD
  2  4  3
  5  1  #
f=g+h=2+3=5
OUEST
  #  2  4
  5  1  3
f=g+h=1+4=5
Fin iteration 6
```

Fin

2	#	4
5	1	3

Développement par sprints – Sprint #5 (1/3)

Créez un projet de nom `Sprint5` au sein de la même Solution que les `Sprints 1 à 4`. Au niveau des répertoires des projets, recopiez (sans partage de sources) **les sources utiles du `Sprint4` dans le répertoire des sources de `Sprint5`. Adaptez le source au `Sprint5`.**

Cinquième Sprint

- **Analyse fonctionnelle :** Donner la solution du problème (suite de coups permettant de passer de l'état initial à l'état final)
- **Spécification :** Finalisez le développement des composants de manière à répondre à toutes les spécifications données dans le projet.
- **Codage :** Codez la fonction `afficherSolution` suivant le format donné dans `ref#5.txt`
- **Test :** Testez votre application par redirection des entrées à partir du fichier `in.txt (JDT du Sprint#5)` et sa sortie vers le fichier `run.txt`.
  Comparez votre fichier `run.txt` au fichier des résultats de référence `ref#5.txt` : si les deux fichiers coïncident, votre `Sprint#5` est validé…
  Bravo !!!
  sinon quelques corrections sont encore à faire !

Sprint#5 – Prototypage des fonctions (2/3)

Sprint#5

```
//affiche la solution
void afficherSolution(Taquin& t);
```

.

Sprint#5 – Résultat de référence `ref#5.txt` (3/3)

2	#	4
5	1	3

```
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
```