Algorithmique
Partiel n° 2

Info-Spé – Epita

D.S. 314446.3 BW (12 mai 2009 - 09 :00)

Consignes (à lire) :

- Vous devez répondre sur **les feuilles de réponses prévues à cet effet**.
  - Aucune autre feuille ne sera ramassée (gardez vos brouillons pour vous).
  - Répondez dans les espaces prévus, **les réponses en dehors ne seront pas corrigées** : utilisez des brouillons !
  - Ne séparez pas les feuilles à moins de pouvoir les ré-agrafer pour les rendre.
  - Aucune réponse au crayon de papier ne sera corrigée.
- La présentation est notée en moins, c'est à dire que vous êtes noté sur 20 et que les points de présentation (2 au maximum) sont retirés de cette note.
- **Les algorithmes :**
  - Tout algorithme doit être écrit dans le langage Algo (pas de C, Caml ou autre).
  - Tout code Algo non indenté ne sera pas corrigé.
  - Tout ce dont vous avez besoin (types, routines) est indiqué en **annexes** (dernière page) !
- Durée : 3h.

Exercice 1 (Graphes et arbres... – *4.5 points*)

Définitions et théorème de caractérisation des arbres

Soit G un graphe d'ordre n. On dira que G est un ***arbre*** s'il vérifie les conditions équivalentes suivantes :

(i) G est connexe et sans cycle,

(ii) pour toutes paires de sommets x et y de G, il existe dans G une chaîne et une seule d'extrémités x et y,

(iii) G est connexe et minimum au sens des arêtes pour cette propriété, c'est-à-dire qu'il n'est plus connexe si on lui supprime l'une quelconque de ses arêtes,

(iv) G est sans cycle et maximum au sens des arêtes pour cette propriété, c'est-à-dire qu'on crée un cycle en ajoutant une arête rendant adjacents deux quelconques de ses sommets qui ne le sont pas,

(v) G est connexe et possède $n-1$ arêtes,

(vi) G est sans cycle et possède $n-1$ arêtes.

La démonstration la plus courte de ce théorème se ferait en 6 étapes, par exemple les implications suivantes :

$$\text{(i)}\Rightarrow\text{(ii)}\Rightarrow\text{(iii)}\Rightarrow\text{(iv)}\Rightarrow\text{(v)}\Rightarrow\text{(vi)}\Rightarrow\text{(i)}$$

Nous n'allons pas utiliser ce cheminement, notre but n'étant pas de démontrer le théorème. En revanche, nous allons montrer certaines implications qui permettraient de faire cette démonstration par un autre chemin. Les implications choisies sont les suivantes :

1. l'implication (ii)⇒(i)
2. la double implication(i)⇒(v),(vi)
3. l'implication (v)⇒(iii)

Exercice 2 (Couvrant et donc... Connexe ? – *6.5 points*)

Les arbres permettent une nouvelle caractérisation des graphes connexes.

1. Donner un principe algorithmique basé sur celui de l'algorithme de Kruskal qui permette de déterminer si un graphe G de N sommets est connexe.
2. Ecrire l'algorithme ***abstrait*** de la fonction *(EstConnexe)* correspondant au principe de la question précédente.

Exercice 3 (L'aller, puis le retour ... – *14 points*)

L'objectif de cet exercice est de construire un plus court chemin aller/retour d'un sommet vers lui même en passant par un sommet particulier. On désire que le chemin de retour **ne passe pas** par les mêmes sommets qu'à l'aller.

On cherche à adapter l'algorithme de Dijkstra pour ce problème. En première approximation, on pourrait considérer qu'il suffit de répéter l'opération de la source à la destination, puis de la destination à la source. Malheureusement, les chemins ainsi trouvés ne garantissent pas que les sommets de l'aller n'apparaissent pas au retour.

L'algorithme que nous allons écrire cherchera donc à construire un plus court chemin pour l'aller, puis un plus court chemin pour le retour qui ne passe pas par les sommets du plus court chemin de l'aller. Dans la suite, nous supposerons qu'un tel chemin existe toujours.

Les chemins seront représentés comme d'habitude à l'aide d'un vecteur de père, où la case `i` donne le père du sommet `i` dans le chemin. Notre algorithme devra construire **deux** vecteurs de pères, un pour l'aller et un autre pour le retour.

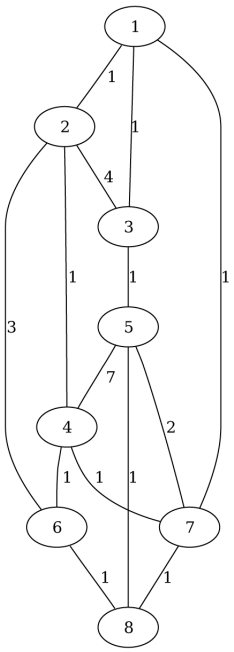

FIG. 1 – Graphe 1

Pour les algorithmes de plus courts chemins nous supposerons définis un tas répondant aux spécifications suivantes :

types
`t_tas` */* tas de t_ listsom (valeur/index) avec clefs réelles */*
Opérations
/ le tas vide */*
`tas_vide() : t_tas`
/ test du vide */*
`est_vide(t_tas t) : booleen`
/ ajout d'une valeur avec sa clef au tas */*
`ajout(t_tas, reel clef, t_listsom val)`
/ récupération et suppression du minimum */*
`prend_min(t_tas) : t_listsom`
/ mise à jour de la clef d'une valeur (ou insertion au besoin) */*
`maj(t_tas, reel clef, t_listsom val)`
/ réinitialisation (suppression de tous les éléments) */*
`vide_tas(t_tas t)`

1. Quelle propriété d'un graphe non-orienté nous assure que nous pouvons trouver un tel chemin aller/retour ?

2. Écrire la procédure `MarqueSommets(src,dst,pere,M)` qui met à vrai la case `M[i]` pour tous les sommets `i` qui sont dans le chemin allant de `src` à `dst` décrit par le vecteur de père `pere`.

3. Donner le principe général de l'algorithme de Dijkstra.

4. Comment peut on adapter ce principe pour résoudre notre problème ?

5. Donner un plus court chemin aller/retour (selon la définition précédente) pour le graphe de la figure 1 partant de 1 à 8 et revenant à 1.

6. Écrire la procédure `dijkstra(g,src,dst,pere,M)` qui trouve le plus court chemin entre `src` et `dst` dans `g` (sans passer par les sommets marqués dans `M`.)

7. Écrire la procédure `pccAR(g,src,dst,pereA,pereR)` qui trouve le plus court chemin aller/retour (satisfaisant la définition précédente) en partant de `scr` et en passant par `dst`. Le vecteur de père `pereA` correspond au chemin aller tandis que le vecteur de père `pereR` correspond au chemin retour. Cette procédure utilisera les procédures écrites précédemment (`MarqueSommets` et `dijkstra`.)

Exercice 4 (Jouons un peu ! – *5 points*)

Dans cet exercice nous allons essayer de résoudre le problème du jeux du taquin. Le taquin est un puzzle assez classique. Il se présente sous la forme d'un damier où peuvent coulisser des pièces carrées. Le but est de positionner les pièces de manière à ce qu'elles reforment le dessin d'origine (où comme dans la version d'origine, de manière à ce qu'elles se retrouvent dans l'ordre). Les pièces ne peuvent se déplacer que s'il l'une des cases adjacentes (haut, bas, gauche ou droite) est vide. Il n'y a qu'une case libre dans tout le taquin.

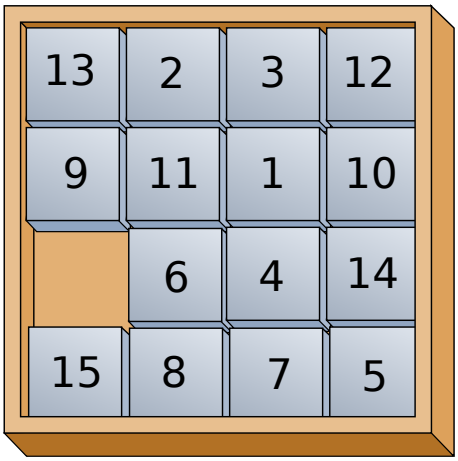

FIG. 2 – Exemple de taquin

Nous allons représenter le taquin par une matrice d'entier $N \times N$ les pièces seront numérotées de 1 à $N^2 - 1$ et le trou aura la valeur 0.

Pour résoudre le taquin nous nous appuierons sur la recherche de plus court chemin dans un graphe. L'algorithme choisit étant A^*, nous ne nous intéresserons pas à son implantation mais plutôt à la représentation du problème et la définition d'une heuristique intéressante.

Taquin 3×3 avant résolution :

0	4	2
1	7	5
3	6	8

Taquin 3×3 après résolution :

0	1	2
3	4	5
6	7	8

1. Dans le graphe que nous utiliserons que représentent les sommets ?

2. Que représentent les arêtes ?

3. Donner le principe de l'algorithme A^*.

4. On cherche à construire une bonne heuristique pour notre problème. En règle générale, une telle heuristique s'appuie sur la définition d'une distance *minimale* à parcourir. En considérant la distance d'une pièce à sa position finale comme la somme des déplacements horizontaux et verticaux à effectuer, donner le principe de la fonction `distance` qui détermine la distance globale entre un taquin mélangé et un taquin résolu ?

5. Écrire la fonction `distance(t_mat_entiers t, entier n) :entier` qui calcule la distance globale entre un taquin mélangé et le taquin résolu où les cases sont triées dans l'ordre croissant de gauche à droite et de haut en bas. On pose que le taquin est carré et qu'il fait `n` cases de côté.

Annexes

Les graphes non orientés

SORTE Graphe

UTILISE Sommet, Entier, Booléen

OPERATIONS

graphe-vide	: → Graphe
ajouter-le-sommet _ à _	: Sommet × Graphe → Graphe
ajouter-l'arête <_,_> à _	: Sommet × Sommet × Graphe → Graphe
_ est-un-sommet-de _	: Sommet × Graphe → Booléen
<_,_> est-une-arête-de _	: Sommet × Sommet × Graphe → Booléen
d°	: Sommet × Graphe → Entier
_ ème-succ-de _ dans _	: Entier × Sommet × Graphe → Sommet
retirer-le-sommet _ de _	: Sommet × Graphe → Graphe
retirer-l'arête <_,_> de _	: Sommet × Sommet → Graphe

Les ensembles

SORTE Ensemble

UTILISE Elément, Booléen

OPERATIONS

ensemble-vide	: → Ensemble
ajouter	: Elément × Ensemble → Ensemble
supprimer	: Elément × Ensemble → Ensemble
_ ∈ _	: Elément × Ensemble → Booléen

Représentation dynamique des graphes

```
types
    t_listsom = ↑ s_som
    t_listadj = ↑ s_ladj
    s_som     = enregistrement
       entier      som
       t_listadj   succ
       t_listadj   pred
       t_listsom   suiv
    fin enregistrement s_som
    s_ladj      = enregistrement
       t_listsom   vsom
       reel        cout
       t_listadj   suiv
    fin enregistrement s_ladj
    t_graphe_d  = enregistrement
       entier     ordre
       booleen    orient
       t_listsom  lsom
    fin enregistrement t_graphe_d
```

La fonction `recherche (entier` s`, t_graphe_d` G`)` qui retourne le pointeur sur le sommet s dans G (de type `t_listsom`) est considérées comme prédéfinies.

Autres types utiles

```
constantes
    Max = /* une valeur suffisante !*/
types
    t_vect_entiers  = Max entier
    t_vect_booleens = Max booleen
```

Vous pouvez utiliser d'autres types de vecteurs, des piles ou des files, ou autres structures, à condition de spécifier le type des éléments.