Algorithmique
Correction Partiel n° 2

Info-Spé – Epita

12 mai 2009 - 09 :00

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

1. l'implication (ii)⇒(i) :
   Si G satisfait (ii), alors il est connexe et ne peut contenir de cycle puisque deux sommets distincts d'un cycle sont reliés sont reliés par deux chaînes du cycle.

2. la double implication(i)⇒(v),(vi) :
   Notons m le nombre d'arêtes de G. Comme G est connexe, on a $m \geqslant n-1$. Comme G est sans cycles, on a $m < n$. D'où le résultat : connexe et sans cycle avec $n-1$ arêtes.

3. l'implication (v)⇒(iii) :
   La suppression d'une arête quelconque de G conduit à un graphe d'ordre n possédant $n-2$ arêtes, trop peu pour qu'il puisse être encore connexe.

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

1. Une façon simple est de créer le graphe partiel T du graphe G en sélectionnant une par une les arêtes de G qui n'existe pas dans T et qui ne créent pas de cycle. On les compte, Lorsque l'on arrive à $N-1$ arêtes dans T, le graphe G est connexe. Si au bout du traitement, T ne contient pas $N-1$ arêtes, il ne l'est pas. **On utilisera pour cela une variante de l'algorithme de Kruskal ne tenant pas compte des coûts**.

2.
```
algorithme fonction EstConnexe : Booléen
parametres locaux
  Graphe G
variables
  Graphe T
  Ensemble M
  Entier i,x,y,rx,ry
  T_VectNEnt pere
debut
  /* Initialisation */
  T ← GrapheVide
  M ← EnsembleVide
  pour x ← 1 jusqua N faire
    pour i ← 1 jusqua d° (x,G) faire
      y ← i ème-succ-de x dans G
      M ← Ajouter({x,y},M)
    fin pour
    pere[x] ← -1
  fin pour
  /* Enrichissements successifs */
  i ← 1
```

```
    tant que (M<>EnsembleVide) et (i<N) faire
      {x,y} ← Choisir(M)
      M ← Supprimer({x,y},M)
      rx ← trouver(x,pere)
      ry ← trouver(y,pere)
      si rx<>ry alors
        reunir(rx,ry,pere)
        T ← Ajouter-l-arete {x,y} dans T
        i ← i+1
      fin si
    fin tant que
      retourne(M<>EnsembleVide)
  fin algorithme fonction EstConnexe
```

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

1. Le graphe doit être **bi-connexe**.

2. **algorithme procedure** MarqueSommets

```
  parametres locaux
    entier              src,dst
    t_vect_entiers      pere
  parametres globaux
    t_vect_booleens     M
  variables
debut
  faire
    M[dst] ← vrai
    dst ← pere[dst]
  tant que dst<>src
fin algorithme procedure MarqueSommets
```

3. **Principe de l'algorithme de Dijkstra :** On sélectionne à chaque itération le prochain sommet non-traité qui a la distance minimale, puis on *relâche* ses successeurs (arcs sortants), c'est à dire on met à jour la distance et le père de chaque successeur s'il est plus intéressant d'y accéder par le sommet courant. On s'arrête lorsque le sommet de destination est sélectionné.

4. **Principe de Dijkstra adapté au problème de l'aller/retour :** Il suffit d'effectuer la recherche du trajet retour en ayant préalablement marqué les sommets qui sont dans le chemin aller pour ne pas les re-sélectionner.

5. Plus court chemin aller/retour de 1 à 8 sur la figure 1 :
   aller : 1,7,8
   retour : 8,5,3,1

6. **algorithme procedure** dijkstra

```
    parametres locaux
      t_graphe_d g
      entier src,  dst
    parametres globaux
      t_vect_entiers pere
      t_vect_booleens M
    variables
      entier i
      t_vect_reels dist
      t_tas t
      t_listsom ps
```

```
    t_listadj pa
  debut
    pour i ← 1 jusqu'a g.ordre faire
      pere[i] ← 0
      dist[i] ← +∞
    fin pour
    pere[src] ← src
    dist[src] ← 0
    t ← tas_vide()
    ajout(t, dist[src], recherche(src, g))
    faire
      ps ← prend_min(t)
      src ← ps↑.som
      M[src] ← vrai
      pa ← ps↑.succ
      tant que (pa <> NUL) faire
        i ← pa↑.vsom↑.som
        si (non M[i] et (dist[i] > (dist[src] + pa↑.cout))) alors
          pere[i] ← src
          dist[i] ← (dist[src] + pa↑.cout)
          maj(t, dist[i], pa↑.vsom)
        fin si
      fin tant que
    tant que (non est_vide(tas) et (ps↑.som <> dst))
  fin algorithme procedure dijkstra
```

7.
```
algorithme procedure pccAR
  parametres locaux
    t_graphe_d          g
    entier              src, dst
  parametres globaux
    t_vect_entiers      pereA, pereR
  variables
    entier              i
    t_vect_booleens     M
debut
  pour i ← 1 jusqu'a g.ordre faire
    M[i] ← faux
  fin pour
  dijkstra(g, src, dst, pereA, M)
  pour i ← 1 jusqu'a g.ordre faire
    M[i] ← faux
  fin pour
  MarqueSommets(src, dst, pereA, M)
  M[src] ← faux
  dijkstra(g, dst, src, pereR, M)
fin algorithme procedure pccAR
```

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

1. Les sommets représentent les états du taquin. Chaque sommet contient la matrice complete du taquin avec les positions de chaque pièce.

2. Les arêtes représentent le déplacement d'une pièce sur le taquin et donc le passage d'un état à un autre. Depuis un sommet il y a donc au plus quatre états sortants possibles qui correspondent à l'échange entre le trou et son voisin en haut, en bas, à gauche ou à droite.

3. L'algorithme A^* est construit comme l'algorithme de Dijkstra : on itère sur un ensemble de sommets ouverts triés selon la somme entre la distance à la source et la valeur d'une fonction heuristique pour ce sommet. A chaque itération on prend le premier sommet parmi les ouverts et on effectue un relâchement sur ses successeurs (non traités.)

4. **Principe** la fonction distance va calculer pour chaque pièce la distance à sa place : si sa place dans la matrice est (i, j) et que la pièce se trouve dans la case $(i'; j')$ alors la distance est : $|i - i'| + |j - j'|$. Enfin, la distance de tout le taquin non résolu avec le taquin résolu est la somme des distances de chaque pièce. Enfin, observe que la piece k doit se trouver dans le taquin résolu à la case $(1 + k \texttt{ div } n, 1 + (5 \texttt{ mod } n))$.

5. algorithme :

```
algorithme fonction distance : entier
  parametres locaux
    t_mat_entiers      t
    entier             n
  variables
    entier             d,i,j,x,y
debut
  d ← 0
  pour i ← 1 jusqu'a n faire
    pour j ← 1 jusqu'a n faire
      x ← 1 + t[i,j] div n
      y ← 1 + t[i,j] mod n
      d ← d + abs(x - i) + abs(y - j)
    fin pour
  fin pour
  retourne d
fin algorithme fonction distance
```