Master BioInformatique

Année : 2009/2010 Session de mars 2010

PARCOURS : Master 1
UE TBS221 : Algorithmique 2ème partie
Épreuve : Examen
Date : Lundi 29 mars 2010
Heure : 10 heures
Durée : 2 heures
Documents : autorisés
Épreuve de M. Alain Griffault

Code d'anonymat :

Avertissement

- La plupart des questions sont indépendantes.
- L'espace laissé pour les réponses est suffisant (sauf si vous utilisez ces feuilles comme brouillon, ce qui est fortement déconseillé).

Exercice 1 (Divisibilité : Variante de recherche de motifs (4 points))

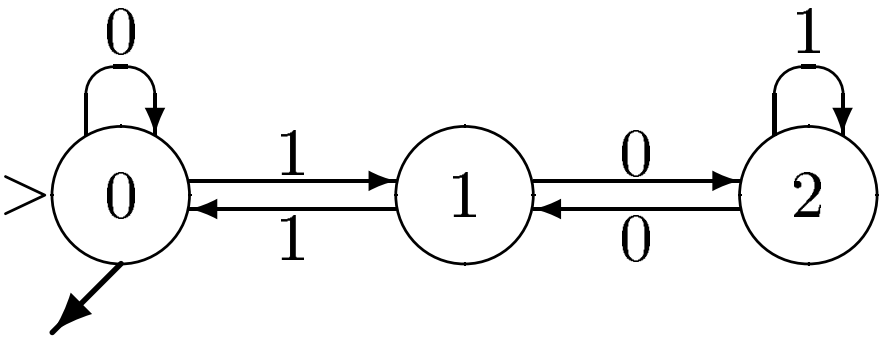

Figure *1 – Un automate de recherche de motifs*

Question 1.1 (2 points) *Compléter le tableau correspondant à la lecture du nombre binaire* 1101010110100011 *par l'automate de la figure 1.*

		1	*1*	*0*	*1*	*0*	*1*	*0*	*1*	*1*	*0*	*1*	*0*	*0*	*0*	*0*	*1*
état après	*0*																
état = 0	*Oui*																
nombre reconnu	*0*																

Question 1.2 (2 points) *Caractériser les nombres reconnus par l'automate de la figure 1.*

Exercice 2 (Chemins fiables : Variante des plus courts chemins pondérés (5 points))
Soit $G(S, A)$ un graphe orienté pondéré par une fonction de probabilité qui associe à chaque arc $(u, v) \in A$ un réel $p(u, v) \in [0..1]$. On interprète le réel $p(u, v)$ comme la probabilité que le moyen de communication entre les sommets u et v ne soit pas interrompu. On fera l'hypothèse que toutes ces valeurs sont indépendantes.

Propriété 1 (admise) *Soit $G(S, A)$ un graphe orienté pondéré par une fonction de probabilité p à valeurs indépendantes. Soit $s_0, \ldots, s_n$ un chemin de longueur non nulle dans G. La probabilité que le chemin $s_0, \ldots, s_n$ ne soit pas interrompu est égale à $\Pi_{0 \leq i < n} p(s_i, s_{i+1})$.*

Question 2.1 (4 points) *Adapter l'algorithme* `PCC_Dijkstra(G,p,r)` *vu en cours, pour obtenir un algorithme* `Chemin_Plus_Fiable(G,p,r,t)` *qui imprime à l'envers le chemin le plus fiable entre deux sommets r et t d'un graphe orienté pondéré.*

```
Chemin_Plus_Fiable(G,p,r,t) {
  pour s dans S-{r} faire {
    // s.dist = MAX_FLOAT;              // attribut dist pour PCC_Dijkstra
    s.proba =                           // INITIALISATION ATTRIBUT proba POUR CET ALGO

    s.pere := nil;
  }
  // r.dist := 0;
  r.proba :=

  // F := Creer_Tas(S);                 // un tas pour gerer l'ordre min
  F := Creer_Tas(S);     // PRECISER POUR QUEL ORDRE

  tant que (Tas_Vide(F)=false) faire {
    // u := Extraire_Min(F);            // dist minimal pour PCC_DIJKSTRA
    u :=

    //pour v dans u.adjacents faire {   // calcul des distances des adjacents
    //  si (v.dist > u.dist + p(u,v))
    //  alors {
    //    v.dist := u.dist + p(u,v);
    //    v.pere := u;
    //  }
    pour v dans u.adjacents faire {     // CALCUL DES PROBA DES ADJACENTS









      }
    }
  }
  // IMPRESSION A L'ENVERS DU CHEMIN DE r A t









}
```

Question 2.2 (1 point) *Donner la complexité de votre algorithme* `Chemin_Plus_Fiable(G,p,r,t)`.

Exercice 3 (Graphe biparti : Variante parcours en largeur (5 points))

Définition 1 *Un graphe biparti est un graphe non orienté $G(S, A)$ dans lequel S peut être partitionné en deux ensembles S_1 et S_2 tels que $(u, v) \in A$ implique soit $(u, v) \in S_1 \times S_2$, soit $(u, v) \in S_2 \times S_1$.*
En d'autres termes, toutes les arêtes passent d'un ensemble à l'autre.

Propriété 2 (admise) *Un graphe non orienté $G(S, A)$ est connexe si pour n'importe quel sommet $s \in S$, l'algorithme* `Parcours_Largeur(G,s)` *vu en cours colorie tous les sommets en noir.*
En d'autres termes, toutes les sommets sont accessibles depuis n'importe quelle racine.

Question 3.1 (4 points) *Compléter l'algorithme* `Parcours_Largeur(G,s)` *vu en cours, pour obtenir un algorithme* `Biparti_Connexe(G)` *qui retourne vrai si et seulement si le graphe G est biparti et connexe.*

```
Biparti_Connexe(G) {
  // on utilise comme racine un sommet quelconque du graphe
  r := S[random()];
  pour s dans S-{r} faire {
    s.couleur := blanc;
    s.d = -1;
    s.pere := nil;
  }
  r.couleur := gris;
  r.d := 0;
  r.pere := nil;
  F := Creer_File_Vide();
  Enfiler(F,r);
  tant que (File_Vide(F)=false) faire {
    u := Tete_File(F);
    pour v dans u.adjacents faire {
      si (v.couleur = blanc)
      alors {
        v.couleur := gris;
        v.d := v.d + 1;
        v.pere := u;
        Enfiler(F,v);
      }
      // COMPLETER PAR DU CODE QUI RETOURNE FAUX SI G N'EST PAS BIPARTI









    }
    Defiler(F);
    u.couleur := noir;
  }
  // COMPLETER PAR DU CODE QUI RETOURNE FAUX SI G N'EST PAS CONNEXE




  retourner true;
}
```

Question 3.2 (1 point) *Donner la complexité de votre algorithme* `Biparti_Connexe(G)`.

Exercice 4 (Nombre de chemins entre deux sommets : Variante du tri topologique (6 points))

Définition 2 *Un graphe orienté est acyclique si pour toutes paires $(s,t) \in S \times S$, l'existence d'un chemin $s \rightsquigarrow t$ implique la non existence d'un chemin $t \rightsquigarrow s$.*
En d'autres termes, il n'y a pas de circuit dans le graphe.

Question 4.1 (2 points) *Soit t un sommet d'un graphe orienté acyclique. Soit u un sommet quelconque, notons $u.nbCheminsVersT$ le nombre de chemins reliant le sommet u au sommet t. Par convention, nous poserons $t.nbCheminsVersT = 0$.*
Donner une propriété reliant $u.nbCheminsVersT$ aux nombres $v.nbCheminsVersT$ pour tous les sommets v adjacents de u.

Question 4.2 (3 points) *Inspirer vous de l'algorithme* `Tri_Topologique(G)` *vu en cours pour écrire un algorithme* `Dag_Comptage_Chemin(G,s,t)` *qui retourne le nombre de chemins reliant s à t.*

```
Visiter_PP(u, t) {
  // Le parametre t est ajoute pour l'algorithme de comptage de chemins
  // code classique de Visiter_PP
  u.couleur := gris;
  u.debut := date;
  date := date+1;
  pour v dans u.adjacent faire {
    si (v.couleur = blanc)
    alors {
      v.pere := u;
      Visiter_PP(v, t);
    }
  }
  u.couleur := noir;
  u.fin := date;
  date := date+1;
  // Ajout pour le tri topologique
  Inserer_Tete_Liste(F,u);
  // AJOUT A FAIRE POUR LE CALCUL DU NOMBRE DE CHEMINS DE u A t




}
```

```
Parcours_Profondeur(G) {
  pour u dans S faire {
    u.couleur := blanc;
    u.pere := nil;
  }
  date := 0;
  pour u dans S faire {
    si (u.couleur = blanc)
    alors Visiter_PP(u);
  }
}

Tri_Topologique(G) {
  pour u dans S faire {
    u.couleur := blanc;
    u.pere := nil;
  }
  date := 0;
  F := Creer_Liste_Vide();      // pour le resultat du tri topologique
  pour u dans S faire {
    si (u.couleur = blanc)
    alors Visiter_PP(u);
  }
  retourner F;
}

Dag_Comptage_chemin(G,s,t) {
  pour u dans S faire {
    u.couleur := blanc;
    u.pere := nil;
    u.nbCheminsVersT := 0;           // pour le comptage de chemins
  }
  date := 0;
  // A COMPLETER









  retourner s.nbCheminsVersT;
}
```

Question 4.3 (1 point) *Donner la complexité de votre algorithme* `Dag_Comptage_Chemin(G,s,t)`.