ASI3 Mercredi 14 Janvier 2015

Algorithmique et Base de la programmation

Durée : *3h00*
Documents autorisés : **AUCUN**

Remarques :
- Veuillez lire attentivement les questions avant de répondre.
- Le barème donné est un barème indicatif qui pourra évoluer lors de la correction.
- Rendez une copie propre.

1 Question de cours (4 points)

Pour rappel résoudre le problème du sac à dos de taille W avec n objets de valeur v_i et de poids w_i, revient à résoudre le problème mathématique suivant :

Soit $X \in [0,1]^n$ tel que $x_i = 1$ si l'objet est dans le sac à dos, 0 sinon. Choisir X afin de maximiser $\sum_{i=1}^{n} x_i v_i$ tel que $\sum_{i=1}^{n} x_i w_i \leq W$.

Nous avons vu en cours que le problème du sac à dos pouvait être résolu assez efficacement grâce à un algorithme de programmation dynamique. L'élément central de cet algorithme est le calcul d'une matrice V tel que $V_{i,j}$ ($0 \leq i \leq n$ et $0 \leq j \leq W$) représente la somme des valeurs des i premiers objets retenus ($\sum_{k=1}^{i} x_k v_k$) pour un sac à dos de contenance maximale j.

1. Donnez la formule de récurrence permettant de calculer les $V_{i,j}$ sachant que $V_{0,j} = 0$
   Solution proposée:
   $V_{i,j} = max(V_{i-1,j}, v_i + V_{i-1,j-w_i})$
2. Donnez la matrice V sur l'exemple suivant, tel que les lignes de la matrice représente les objets et les colonnes les tailles du sac à dos :
   - Taille maximale du sac à dos : 9
   - Objets :

objets	1	2	3	4	5
valeurs	5	3	4	6	5
poids	2	3	1	8	3

   Solution proposée:

i	w_i	v_i	0	1	2	3	4	5	6	7	8	9
0			0	0	0	0	0	0	0	0	0	0
1	2	5	0	0	5	5	5	5	5	5	5	5
2	3	3	0	0	5	5	5	8	8	8	8	8
3	1	4	0	4	5	9	9	9	12	12	12	12
4	8	6	0	4	5	9	9	9	12	12	12	12
5	3	5	0	4	5	9	9	10	13	13	13	18

3. Quels objets choisir pour résoudre ce problème ? Justifiez.
   Solution proposée:
   Objets retenus :
   - 5 car $V_{5,9} \neq V_{4,9}$

— 3 car $V_{3,6} \neq V_{2,6}$
— 2 car $V_{2,5} \neq V_{1,5}$
— 1 car $V_{1,2} \neq V_{0,2}$

2 Exercice de TD (3 points)

Écrire une fonction, `rechercheDichotomique`, qui détermine par dichotomie le plus grand indice d'un élément (dont on est sûr de l'existence) dans un tableau t de n entiers significatifs triés par ordre croissant. Il peut y avoir des doubles (ou plus) dans le tableau.

Solution proposée:

fonction rechercheDichotomique (t : **Tableau**[1..MAX] **d'Entier** ; n : **Naturel** ; element : **Entier**) **: Naturel**

⌊**précondition(s)** ∃ 1 ≤ i ≤ n / t[i] = element

Déclaration a,b,m resultat : **Naturel**

```
debut
   si t[1] = element alors
      resultat ← 1
   sinon
      a ← 1
      b ← n
      m ← (a + b) div 2
      tant que a < b faire
         si t[m] = element alors
            a ← m
         sinon
            si t[m] < element alors
               a ← m+1
            sinon
               b ← m-1
            finsi
         finsi
      fintantque
      resultat ← b
   finsi
   retourner resultat
fin
```

3 Problème (13 points)

Pour répondre aux questions de cet exercice, vous pouvez utiliser les fonctions et procédures données en annexe. Vous donnerez l'algorithme de toute autre fonction ou procédure que vous utiliserez.

3.1 Partie 1 : un arbre recouvrant minimal (6 points)

Nous avons vu en cours que l'algorithme de Dijkstra permet d'obtenir un arbre recouvrant de coût minimal sur un graphe valué avec des nombres positifs. Cet algorithme est le suivant :

procédure dijkstra **(E** g : Graphe<Sommet,**ReelPositif**>, s : Sommet, **S** arbreRecouvrant : Arbre <Sommet>, coutDepuisS : Dictionnaire<Sommet,**ReelPositif**>**)**

  Déclaration l : Liste<Sommet>
  sommetDeA, sommetAAjouter : Sommet

debut
  arbreRecouvrant ← creerRacine(liste(),s)
  coutDepuisS ← dictionnaire()
  ajouter(coutDepuisS,s,0)
  l ← sommetsAccessiblesDepuisArbre(g,arbreRecouvrant)
  tant que non estVide(l) **faire**
    sommetLePlusProche(l, arbreRecouvrant, coutDepuisS, sommetDeA, sommetAAjouter)
    ajouter(coutDepuisS, sommetAAjouter, rechercher(coutDepuisS, sommetDeA) + obtenirValeur(g, sommetDeA, sommetAAjouter))
    ajouterCommeFils(arbreRecouvrant, sommetDeA, sommetAAjouter)
    l ← sommetsAccessiblesDepuisArbre(g, arbreRecouvrant)
  fintantque
fin

Tel que :
- la fonction `sommetsAccessiblesDepuisArbre` permet d'obtenir la liste des sommets présents dans le graphe G, non présents dans l'arbre recouvrant A, mais accessibles depuis un sommet de l'arbre ;
- la procédure `sommetLePlusProche` qui permet d'identifier le prochain sommet à ajouter dans l'arbre ainsi que son futur « père » dans ce dit arbre ;
- la procédure `ajouterCommeFils` qui permet d'ajouter un sommet dans l'arbre en spécifiant son père.

3.1.1 Signatures

Donnez les signatures des trois opérations précédentes.

Solution proposée:

- **fonction** sommetsAccessiblesDepuisArbre **(**g : Graphe<Sommet,**ReelPositif**>, a : Arbre<Sommet>**)** **:** Liste<Sommet>
- **procédure** sommetLePlusProche **(E** Liste<Sommet>,a : Arbre,cout : Dictionnaire<Sommet,**ReelPosi** **S** sommeDeA, sommetAAjouter : Sommet**)**
- **procédure** ajouterCommeFils **(E/S** a : Arbre<Sommet>, **E** sommetPere, sommetFils : Sommet**)**

3.1.2 Application

Donnez l'arbre recouvrant minimal pour le graphe présenté par la figure 1 depuis le sommet 1.

Solution proposée:

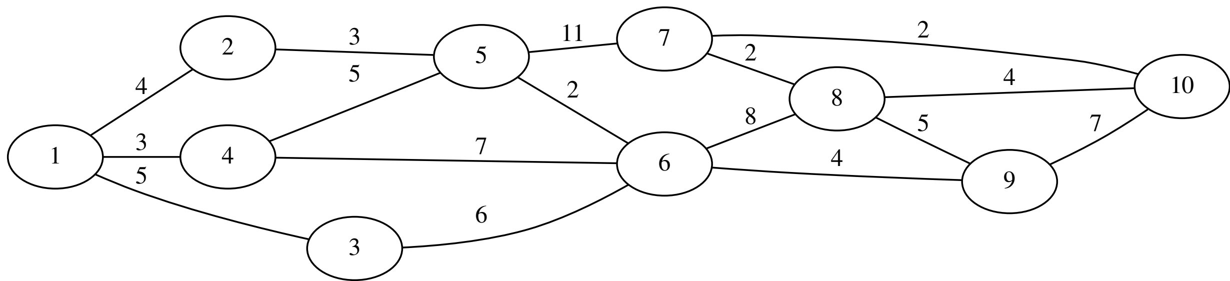
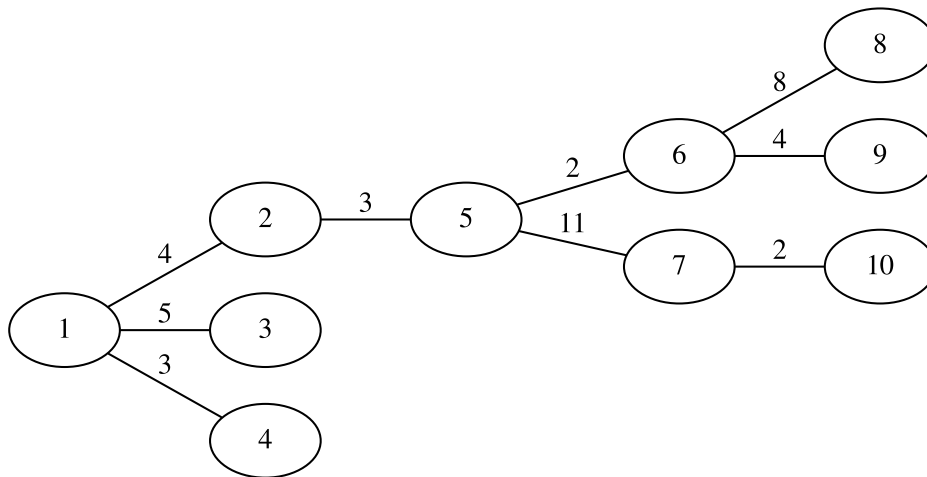


FIGURE 1 – Un graphe valué positivement



3.1.3 Algorithme

Donnez l'algorithme de la fonction `sommetsAccessiblesDepuisArbre` (n'oubliez pas de décomposer le problème si besoin).

Solution proposée:

Analyse :

- sommetsAccessiblesDepuisArbre : Graphe × Arbre<Sommet> → Liste<Sommet>
  - valeurs : Arbre<Sommet> → Liste<Sommet>
  - estPresent : Liste<Sommet> × Sommet → **Booleen**

Conception préliminaire :

- **fonction** valeurs (a : Arbre<Sommet>) **:** Liste<Sommet>
- **fonction** estPresent (l : Liste<Sommet>, s : Sommet) **: Booleen**

Conception détaillée :

fonction valeurs (a : Arbre<Sommet>) **:** Liste<Sommet>
 Déclaration temp : Liste<Sommet>
debut
 si estVide(a) **alors**

```
        retourner liste()
    sinon
        temp ← liste()
        inserer(temp,1,obtenirElement(a))
        pour chaque f de obtenirFils(a)
            pour chaque v de valeurs(a)
                inserer(temp,1,v)
            finpour
        finpour
        retourner temp
    finsi
fin
fonction estPresent (l : Liste<Sommet>, s : Sommet ) : Booleen
    Déclaration   i : Naturel
debut
    i ← 1
    tant que i≤longueur(l) et obtenirElement(l,i)≠s faire
        i ← i+1
    fintantque
    retourner i>longueur(l)
fin
fonction sommetsAccessiblesDepuisArbre (g : Graphe<Sommet,ReelPositif>, a : Arbre<Sommet>)
: Liste<Sommet>
    Déclaration   res : Liste<Sommet>
debut
    res ← liste()
    pour chaque s2 de obtenirSommets(g)
        si non estPresent(res,s2) alors
            pour chaque s1 de res
                si arcPresent(g,s1,s2) alors
                    inserer(res,1,s1)
                finsi
            finpour
        finsi
    finpour
fin
```

3.2 Partie 2 : chemin le plus court (4 points)

Donnez l'algorithme de la fonction suivante qui permet d'obtenir le chemin (une liste de sommets) le plus court permettant d'aller d'un sommet $s1$ à un sommet $s2$ d'un graphe valué avec des nombres positifs :

— **fonction** cheminPlusCourt (g :Graphe, s1,s2 : Sommet) **:** Liste<Sommet>
  ⌊**précondition(s)** sommetPresent(g,s1) et sommetPresent(g,s2)

Solution proposée:

```
procédure cheminPlusCourtR (E a : Arbre<Sommet>, sCible : Sommet, S ch : Liste<Sommet>)
debut
```

```
  si estVide(a) alors
    ch ← liste()
  sinon
    si obtenirElement(a)=sCible alors
      ch ← liste()
      inserer(ch,1,sCible)
    sinon
      pour chaque f de obtenirFils(a)
        cheminPlusCourtR(f,sCible,temp)
        si non estVide(temp) alors
          inserer(temp,1,obtenirElement(a))
        finsi
      finpour
    finsi
  finsi
fin
fonction cheminPlusCourt (g : Graphe, s1,s2 : Sommet) : Liste<Sommet>
  ⌊précondition(s)  sommetPresent(g,s1) et sommetPresent(g,s2)
  Déclaration  a : Arbre<Sommet>
               c : Dictionnaire<Sommet,ReelPositif>
               res : Liste<Sommet>
debut
  dijkstra(g,s1,a,c)
  cheminPlusCourtR(a,s2,res)
  retourner res
fin
```

3.3 Partie 3 : Skynet le virus (3 points)

Le site Web `www.codingame.com` propose des exercices ludiques de programmation. L'un des exercices, « Skynet le virus » est présenté de la façon suivante :

« Votre virus a créé une *backdoor* sur le réseau Skynet vous permettant d'envoyer de nouvelles instructions au virus en temps réel. Vous décidez de passer à l'attaque active en empêchant Skynet de communiquer sur son propre réseau interne. Le réseau Skynet est divisé en sous-réseaux. Sur chaque sous-réseau un agent Skynet a pour tâche de transmettre de l'information en se déplaçant de noeud en noeud le long de liens et d'atteindre une des passerelles qui mène vers un autre sous-réseau. Votre mission est de reprogrammer le virus pour qu'il coupe les liens dans le but d'empêcher l'agent Skynet de sortir de son sous-réseau et ainsi d'informer le *hub* central de la présence de notre virus. »

Bref, l'agent Skynet (S) est sur un graphe (par exemple celui de la figure 2 où les identifiants des sommets ne sont pas indiqués) valué (avec la valeur 1 pour chaque arc) dont certains sommets sont des passerelles (P). Le but du jeu est d'empécher l'agent skynet d'atteindre une des passerelles en supprimant le moins d'arcs du graphe.

L'algorithme de ce jeu est proposé par la procédure `skynet`. L' `agentSkynet` parcourt le graphe (grâce à la fonction `seDeplace`) de sommet en sommet à chaque itération. Pour résoudre ce problème, il faut couper un arc du graphe à chaque itération de façon à ce que l'agent Skynet ne puisse pas atteindre l'une des passerelles. De plus il faut faire le moins de coupures possibles (le score est fonction de ce paramètre). Pour cela il suffit de supprimer le premier arc du chemin le plus court entre l'`agenSkynet` et la plus proche passerelle.

Complétez l'algorithme de la procédure `skynet` (remplacer les ... par une ou plusieurs instructions).

procédure skynet **(E/S** g : Graphe<Sommet>, **E** agentSkynet : Sommet, **S** agentSkynetAAtteindPasserelle : **Booleen)**

Déclaration passerelles : Liste<Sommet>
s : Sommet
...

debut
passerelles ← sommetsDesPasserelles(g)
tant que agentSkynetPeutAtteindreUnePasserelle(g,agentSkynet) et non estPresent(passerelles, agentSkynet) **faire**
...
supprimerArc(g,agentSkynet,s)
agentSkynet ← seDeplace(g, agentSkynet)
fintantque
agentSkynetAAtteindPasserelle ← estPresent(agentSkynet,passerelles)
fin

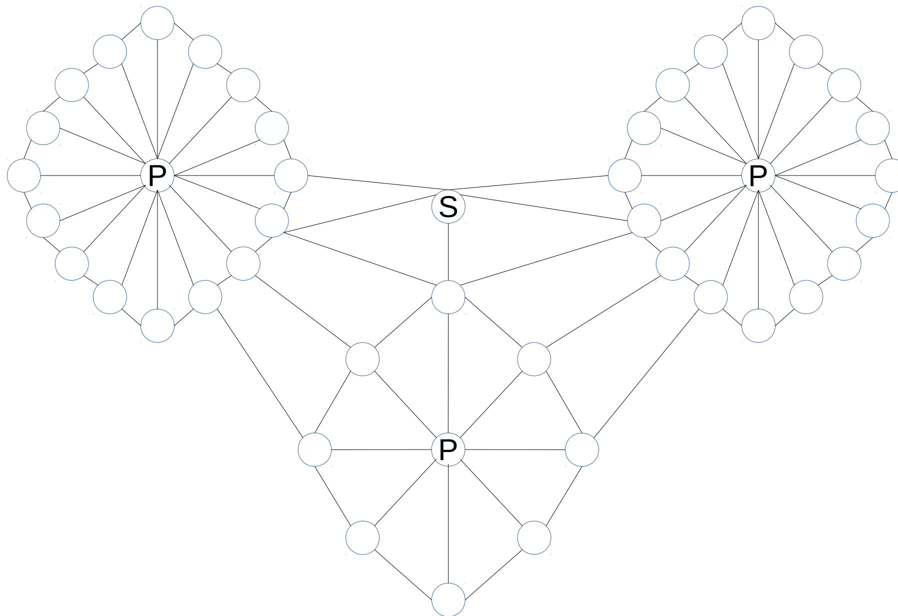


FIGURE 2 – Un sous réseau Skynet

Solution proposée:

procédure skynet **(E/S** g : Graphe<Sommet>, **E** agentSkynet : Sommet, **S** agentSkynetAAtteindPasserelle : **Booleen)**

Déclaration passerelles : Liste<Sommet>
s1,s2,p : Sommet
chMin,temp : Liste<Sommet>
lmin : **Naturel**

debut
passerelles ← sommetsDesPasserelles(g)
tant que agentSkynetPeutAtteindreUnSommet(g,agentSkynet) et non estPresent(passerelles, agentSkynet) **faire**

```
    lmin ← longueur(obtenirSommets(g))
    pour chaque p de passerelles
      ch ← cheminPlusCourt(g,agentSkynet,p)
      si non estVide(ch) et longueur(ch)<lmin alors
        lmin ← longueur(ch)
        chMin ← ch
      finsi
    finpour
    s ← obtenirElement(ch,2)
    supprimerArc(g,agentSkynet,s)
    agentSkynet ← seDeplace(g, agentSkynet)
  fintantque
  agentSkynetAAtteindPasserelle ← estPresent(agentSkynet,passerelles)
fin
```

Annexe : Conception préliminaire de certains TAD Collection

Vous trouverez dans cette annexe les signatures des fonctions et procédures que vous pouvez utiliser dans vos algorithmes.

Type Liste (paramétré par `Element`)

- **fonction** liste **() :** Liste
- **fonction** estVide **(**uneListe : Liste**) : Booleen**
- **procédure** insérer **(E/S** uneListe : Liste, **E** position : **Naturel**, element : Element**)**
  ⌊**précondition(s)** $1 \leq position \leq longueur(uneListe) + 1$
- **procédure** supprimer **(E/S** uneListe : Liste, **E** position : **Naturel)**
  ⌊**précondition(s)** $1 \leq position \leq longueur(uneListe)$
- **fonction** obtenirElement **(**uneListe : Liste, position : **Naturel) :** Element
  ⌊**précondition(s)** $1 \leq position \leq longueur(uneListe)$
- **fonction** longueur **(**uneListe : Liste**) : Naturel**

Type Dictionnaire (paramétré par `Element`)

- **fonction** dictionnaire **() :** Dictionnaire
- **procédure** ajouter **(E/S** unDictionnaire : Dictionnaire, **E** clef : Clef, element : Valeur**)**
- **procédure** retirer **(E/S** unDictionnaire : Dictionnaire, **E** clef : Clef**)**
- **fonction** estPrésent **(**unDictionnaire : Dictionnaire, clef : Clef**) : Booleen**
- **fonction** rechercher **(**unDictionnaire : Dictionnaire, clef : Clef**) :** Valeur
  ⌊**précondition(s)** estPresent(unDictionnaire, clef)

Arbre (paramétré par `Element`)

- **fonction** arbre **() :** Arbre
- **fonction** estVide **(**a : Arbre**) : Booleen**
- **fonction** creerRacine **(**fils : Liste<Arbre>,e : Element**) :** Arbre
- **fonction** obtenirElement **(**a : Arbre**) :** Element

⌊**précondition(s)** non estVide(a)

— **fonction** obtenirFils **(**a : Arbre**) :** Liste<Arbre>

⌊**précondition(s)** non estVide(a)

— **procédure** fixerFils **(E/S** a : Arbre, **E** fils : Liste<Arbre>**)**

⌊**précondition(s)** non estVide(a)

Graphe (paramétré par `Cle et Valeur (une valeur par arc)`)

— **fonction** graphe **() :** Graphe
— **fonction** estVide **(**g : Graphe**) : Booleen**
— **procédure** ajouterSommet **(E/S** g : Graphe, **E** s : Cle**)**

⌊**précondition(s)** non sommetPresent(g,s)

— **procédure** ajouterArc **(E/S** g : Graphe, **E** s1,s2 : Cle**)**

⌊**précondition(s)** non arcPresent(g,s1,s2)

— **fonction** sommetPresent **(**g : Graphe, s : Cle**) : Booleen**
— **fonction** arcPresent **(**g : Graphe, s1,s2 : Cle**) : Booleen**
— **procédure** supprimerSommet **(E/S** g : Graphe, **E** s : Cle**)**

⌊**précondition(s)** sommetPresent(g,s)

— **procédure** supprimerArc **(E/S** g : Graphe, **E** s1,s2 : Cle**)**

⌊**précondition(s)** arcPresent(g,s1,s2)

— **fonction** obtenirSommets **(**g : Graphe**) :** Liste<Cle>
— **fonction** obtenirValeur **(**g : Graphe, s1,s2 : Cle**) :** Valeur

⌊**précondition(s)** arcPresent(g,s1,s2)

— **procédure** fixerValeur **(E/S** g : Graphe, **E** s1,s2 : Cle, v : Valeur**)**

⌊**précondition(s)** arcPresent(g,s1,s2)