Algorithmique — M1

Examen du 11/1/11 -corrigé

Université Paris Diderot

On applique un algorithme de cours

Exercice 1 – *Routage*

Le serveur S est connecté à la machine T par un réseau avec les noeuds A, B, C, D, les capacités de connexions entre les noeuds sont (en Mbit/s) :

	A	B	C	D	T
S	2	6	1		
A		3		7	
B				3	5
C		2		6	
D	3				4

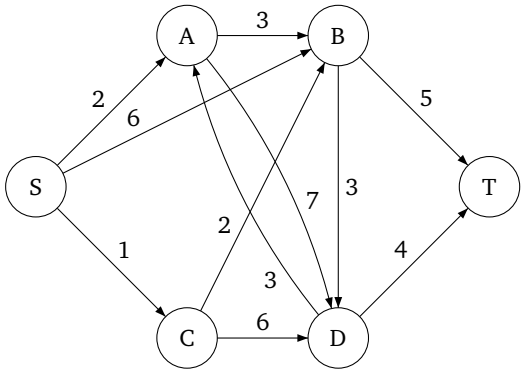
L'utilisateur de la machine T télécharge un très grand fichier du serveur S. On veut trouver le routage qui maximise le débit.

1. Quel problème algorithmique de cours correspond à ce problème de routage ? Comment s'appelle l'algorithme du cours ?

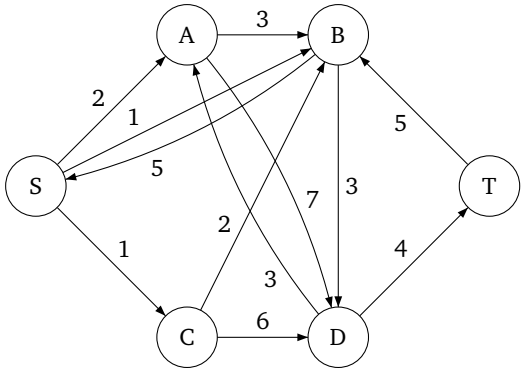
Correction. Problème de flot maximum dans un réseau. Algorithme de Ford-Fulkerson.

2. Appliquez cet algorithme. Donnez toutes les étapes de son application.

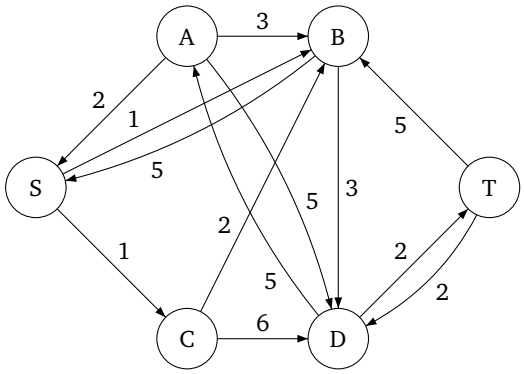
Correction. Dessinons d'abord le réseau :



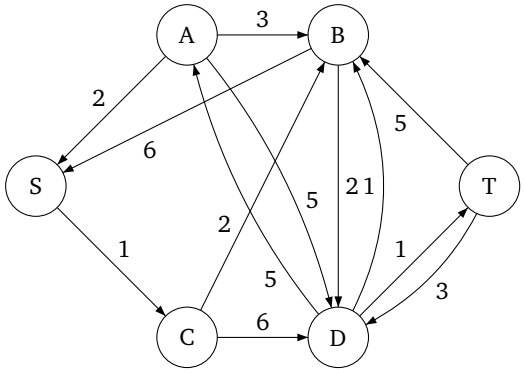
On trouve le premier chemin améliorant SBT de poids 5, et on dessine le réseau résiduel.



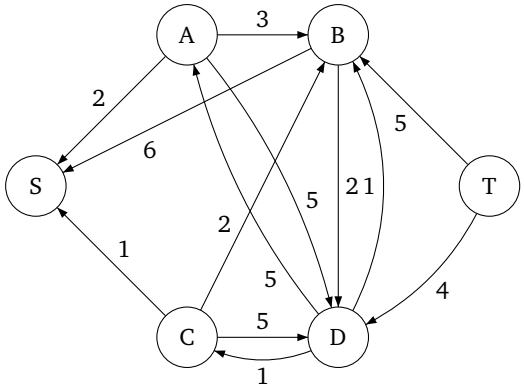
On trouve le second chemin améliorant SADT de poids 2, et on dessine le réseau résiduel.



On trouve le troisième chemin améliorant SBDT de poids 1, et on dessine le réseau résiduel.



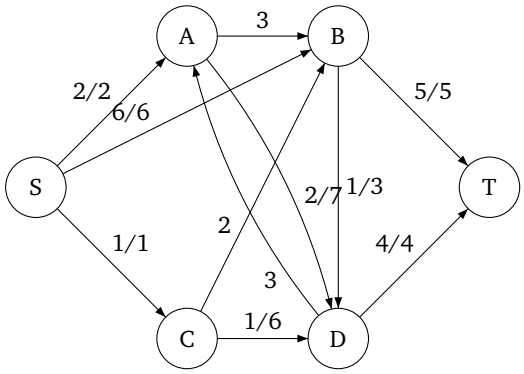
On trouve le quatrième chemin améliorant SCDT de poids 1, et on dessine le réseau résiduel.



Il ne reste plus de chemin améliorant, l'algorithme s'arrête.
Le flot maximal est montré dans le point suivant

3. Quel est le débit maximal, quel routage assure ce débit ?

Correction. Le débit optimal est 5+2+1+1, il est réalisé par le routage suivant(la somme de 4 flots : 5 sur SBT, 2 sur SADT, 1 sur SBDT, 1 sur SCDT) :



Un algorithme facile

Exercice 2 – *Valeur encadrée*

Étant donné un tableau trié d'entiers $A[s..f]$ et deux entiers ("bornes") $a \leqslant b$, on cherche s'il existe un élément $A[i]$ du tableau tel que $a \leqslant A[i] \leqslant b$ (s'il y en a plusieurs trouvez un)

Exemple Soit le tableau $A[1..5] = [3, 7, 8, 43, 556]$ et les bornes $a = 40, b = 50$. Dans ce cas-là la valeur encadrée existe : c'est $A[4] = 43$.

1. Donnez (en pseudocode) un algorithme "diviser-pour-régner" qui résout ce problème. Expliquez briévement.

Correction. Coupons le tableau en deux moitiés : $A[s..m]$ et $A[m+1..f]$. Les trois cas ci-dessous correspondent à trois positions de l'élément de milieu $A[m]$ par rapport à l'intervalle $[a, b]$.

À gauche : $A[m] < a$. Dans ce cas-là tous les éléments de la moitié gauche du tableau sont aussi $< a$, et ne peuvent pas appartenir à l'intervalle demandé $[a, b]$. On va chercher la valeur encadrée dans la moitié droite seulement.

Dedans : $a \leqslant A[m] \leqslant b$. On a déjà trouvé une valeur encadrée $A[m]$. On retourne son indice.

À droite : $b < A[m]$. Ce cas est symétrique au premier, la recherche peut être limitée à la moitié gauche du tableau.

Cette analyse mène à la fonction récursive suivante chercher(s,f) qui renvoie l'indice de la valeur encadrée dans le tableau A[s..f] (ou $\perp$ s'il n'y en a pas. On suppose que le tableau A, et les bornes a,b sont des variables globales.

```
fonction chercher(s,f)
        // cas de base
        si (s=f)
                si (A[s] ∈ [a; b])
                                retourner s
                sinon retourner ⊥

        // on divise pour régner
        m=(s+f)/2
        si A[m] < a
                ind= chercher(m+1,f)
        sinon si a ⩽ A[m] ⩽ b
                ind=m
        sinon
                ind=chercher(s,m)

        retourner ind
```

2. Analysez sa complexité.

Correction. On a réduit un problème de taille n à un seul problème de la taille $n/2$ et des petits calculs en $O(1)$, donc la complexité satisfait la récurrence :

$$T(n) = T(n/2) + O(1).$$

En la résolvant par Master Theorem on obtient que $T(n) = O(\log n)$.

Un jeu - deux puzzles - deux algorithmes

Un *Domino* est un rectangle qui contient 2 lettres, par exemple | A | H | (il ne peut pas être retourné).

Règle : Une séquence de dominos est une *chaîne*, si les lettres voisines coïncident sur chaque paire des dominos consécutifs, par exemple | A | H | | H | K | | K | A | | A | Z | | Z | V | .

Donnée initiale pour les deux puzzles : n pièces de dominos : $D_1 =$ | x_1 | y_1 |,..., $D_n =$ | x_n | y_n |

Puzzle 1 (permutation) : Il faut trouver une permutation de tous ces dominos qui forme une chaîne (selon la règle ci-dessus), si celle-là existe.

Puzzle 2 (sous-séquence) : Il faut trouver la plus longue sous-séquence de tous ces dominos qui forme une chaîne (selon la règle ci-dessus).

Exemple : Pour les pièces | A | H |, | A | Z |, | K | A |, | H | K |, | Z | V | :

– la solution du puzzle 1 est | A | H | | H | K | | K | A | | A | Z | | Z | V |;

– la solution du puzzle 2 est | A | H | | H | K |, ou bien | A | Z | | Z | V | .

Exercice 3 – *Premier puzzle - backtracking*

. On cherche à développer un algorithme de type backtracking qui résout le puzzle 1.

1. Écrivez l'algorithme (en pseudocode)

Correction. On suppose que les pièces sont représentées par deux tableaux $x[1..n]$ et $y[1..n]$. On représentera les solutions partielles du problème par un tableau Chaine[1..k] qui contiendra les indices des pièces de domino formant une chaîne.
La fonction de backtracking (recherche en profondeur) étant donné Chaine[1..k] cherche une solution de problème qui commence par cette Chaine.

```
fonction chercher(k)
        si k=n
                Afficher(Chaine)
                stop

        pour tout ind∉ Chaine
                si k=0 || compatible(Chaine[k],ind)
                   Chaine[k+1]=ind
                   chercher(k+1)
```

La primitive compatible(i,j) teste si on peut placer le dé i à gauche du dé j.

```
fonction booléenne compatible(i,j)
        return y[i] == x[j]
```

Le programme principal :

```
initialiser
chercher(0)
afficher "pas de solution"
```

2. Expliquez cet algorithme (vous pouvez vous inspirer de l'indication)

Correction. On définit une k-solution partielle comme une chaîne composée de k dominos (parmi les n pièces données). Une k-solution peut être étendue vers une $k+1$-solution en ajoutant à sa droite une pièce qui (1) n'est pas encore utilisée et (2) est compatible. On fait une recherche en profondeur dans l'arbre se solutions partielles, en commençant par sa racine (la 0-solution vide). On s'arrête si on trouve une chaîne qui utilise toutes s pièces (k=n).

3. Estimez le nombre d'opérations nécessaire

Correction. Au pire cas on teste toutes les permutations, il y en a $n!$ ce qui donne une complexité exponentielle.

4. Montrez le déroulement de votre algorithme pour les pièces
| A | H | , | A | Z | , | K | A | , | H | K | , | Z | V | .

Correction.
–vide ;
– | A | H |
– | A | H | | H | K |
– | A | H | | H | K | | K | A |
– | A | H | | H | K | | K | A | | A | Z |
– | A | H | | H | K | | K | A | | A | Z | | Z | V |

L'algorithme s'est donc déroulé sur une seule branche de l'arbre, sans retour arrière.

Indication : Essayez de répondre aux questions suivantes pour parvenir à un tel algorithme.
- Comment tester que deux dominos peuvent être adjacents ?
- Comment définir une solution partielle ?
- Quelle est une solution partielle de taille 0 ?
- Comment passer d'une solution partielle à une solution plus grande ?
- Quand s'arrêter ?
- Comment représenter une solution partielle par une structure de données ? Comment représenter les dominos déjà utilisés en chaîne et les dominos encore disponibles ?

Exercice 4 – *Deuxième puzzle - programmation dynamique*

On cherche à développer un algorithme de type programmation dynamique qui résout le puzzle 2.

1. Soit $c(i)$ une fonction entière qui donne la longueur de la chaîne la plus longue qui est une sous-séquence de $D_1, D_2, \ldots D_i$ et qui se termine par la pièce D_i. Écrivez les équations de récurrence pour cette fonction sans oublier les cas de base.

Correction. Cas de base $c(1) = 1$. Pour $i > 1$ la sous-chaîne qui nous intéresse est la plus longue parmi les chaînes suivantes :
–Le dé D_i seul
–Pour un $j < i$ et tel que D_j est compatible avec D_i on prend la chaîne la plus longue jusqu'à D_j, et on y ajoute le dernier dé D_i.
Ça donne la récurrence suivante

$$c(i) = \begin{cases} 1, & \text{si } i = 1 \\ \max\{1, c(j) + 1 | & j < i \text{ tel que compatible}(j,i)\} \text{si } i > 1 \end{cases}$$

2. Écrivez un algorithme efficace (récursif avec "marquage" ou itératif) pour calculer c .

Correction. L'algo itératif remplit le tableau C[i] par les valeurs de la fonction c(i) pour i de 1 jusqu'à n.

```
C[1]=1 ;
pour i de 2 à n faire
      meilleur=1
      pour j de 1 à i-1 faire
            si compatible(j,i) et C[j+1]> meilleur
              meilleur=C[j]+1
      C[i]=meilleur
```

3. En sachant calculer la fonction choisie c, comment trouver la longueur de la sous-chaîne la plus longue parmi $D_1, D_2, \ldots D_i$.

Correction. Le résultat final est l'élément maximum de tout le tableau C[1..n]

4. Analysez la complexité de votre algorithme.

Correction. $O(n^2)$ à cause de deux boucles imbriquées.

5. Montrez le déroulement de votre algorithme pour les pièces
| A | H |, | A | Z |, | K | A |, | H | K |, | Z | V |.

Correction.
–$C[1] = 1$
–$C[2] = 1$ (il n'y a pas de dé compatible à gauche de la 2-ème pièce).
–$C[3] = 1$ (même raison).
–$C[4] = \max\{1, C[1] + 1\} = 2$ (les dés 1 et 4 sont compatibles).
–$C[5] = \max\{1, C[2] + 1\} = 2$ (les dés 2 et 5 sont compatibles).

6. Comment modifier votre algorithme pour qu'il trouve la sous-chaîne la plus longue (et non seulement sa longueur).

Correction. Pour chaque i il faut mémoriser le meilleur dé j à mettre à gauche de D_i. Ça donne

```
C[1]=1 ; J[1]=0
pour i de 2 à n faire
    meilleur=1
    pour j de 1 à i-1 faire
        si compatible(j,i) et C[j+1]> meilleur
            meilleur=C[j]+1 ; J[i]=j
    C[i]=meilleur ;
```

On prend le i qui maximise $C[i]$ et on compose la sous-chaîne optimale de droite à gauche : i, puis $J[i]$, puis $J[J[i]]$ etc.