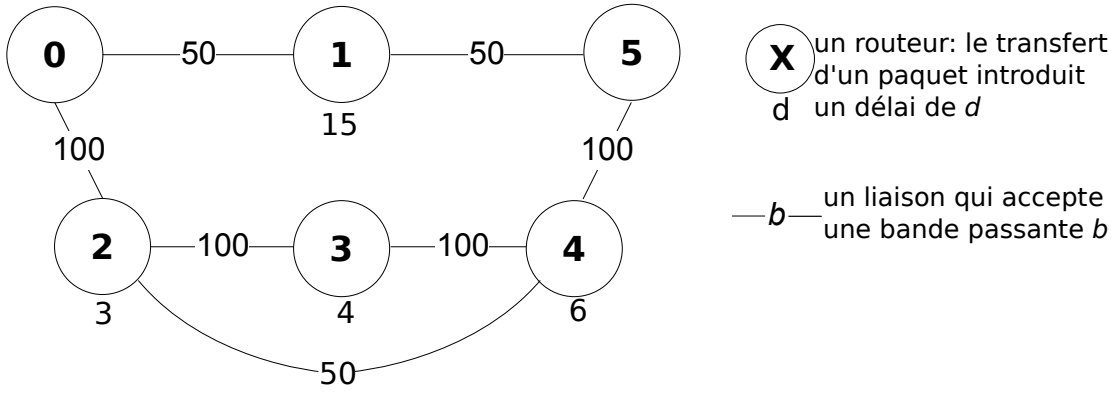
Mini-Projet (TP4) “Consolidation des bases de la programmation”

Ce mini-projet porte sur un problème issu de l'ingénierie du trafic réseau. Il faut déterminer comment allouer les ressources d'un réseau pour assurer un routage optimal des paquets, tout en respectant des contraintes de bande passante. Soit l'exemple du réseau ci-dessous.



Étant donnés $n = 6$ routeurs, il faut déterminer le chemin optimal de paquets du routeur 0 (la source) au routeur $n - 1 = 5$ (la destination). Le transfert de tout paquet par un routeur introduit un délai noté d. Les liaisons possèdent une bande passante (capacité) maximale noté b. En fonction de différentes contraintes (voir ci-dessous), il peut y avoir plusieurs solutions optimales :

A. S'il faut assurer une bande passante de minimum $bmin = 100$, la seule solution faisable est
   - $0 \to 2 \to 3 \to 4 \to 5$ avec un coût (délai) de 3+4+6 = 13.

B. S'il faut assurer une bande passante de $bmin = 50$, il y a trois solutions candidates :
   1. $0 \to 1 \to 5$ de coût (delai) 15 ;
   2. $0 \to 2 \to 4 \to 5$ de coût 3+6 = 9 ;
   3. $0 \to 2 \to 3 \to 4 \to 5$ de coût 3+4+6 = 13.

   La deuxième solution est optimale, car le coût 9 est le plus petit.

C. Si on impose $bmin = 50$ et ainsi *une contrainte de TTL* (Time To Live) maximal de $ttlmax = 2$, alors la seule solution faisable devient $0 \to 1 \to 5$, car c'est le seul chemin avec un TTL de 2 (avec uniquement 2 liens utilisés).

Étape 1 : lecture des données du problème Écrire une routine qui permet de lire les données du problème à partir d'un fichier qui respecte les conventions ci-dessous : *a*) le nombre de routeurs n ; *b*) la bande passante minimale noté $bmin$; *c*) le TTL maximal noté $ttlmax$; *d*) un tableau d avec les délais de tous les n routeurs ; *e*) une matrice b avec la bande passante (capacité) de chaque lien. Voici un exemple de fichier d'entrée pour le cas noté “C.” plus haut.

```
6                          <--- n
50                         <--- bmin
2                          <----ttlmax
0 15 3 4 6 0               <--- tableau d; pas de délai pour 0 (source) et 5 (destination)
0   50  100 0   0   0      <--- matrice 6 X 6 de bande passante
50  0   0   0   0   50
100 0   0   100 50  0
0   0   100 0   100 0
0   0   50  100 0   100
0   50  0   0  100  0
```

Étape 2 : écrire une fonction `calcCout(`*tab*`)` **qui calcule le cout d'une solution candidate codée par un tableau** *tab* Par exemple, `calcCout([0,1,5])` devrait renvoyer 15.
Astuce : Si le tableau ne représente pas une solution faisable (par exemple, la contrainte de TTL n'est pas satisfaite), le coût pourrait prendre en compte une pénalité pour toute contrainte violée. Par exemple, si on considère $ttlmax = 1$, alors $[0, 1, 5]$ n'est pas une solution faisable. Dans ce cas, `calcCout([0,1,5])` pourrait renvoyer $15 + 1000 \cdot 1$. Pour le même $ttlmax = 1$, `calcCout([0,2,4,5]` devrait renvoyer $9 + 1000 \cdot 2$, car $ttlmax$ serait dépassé de deux unités.
Etape 3 : écrire une fonction `changeSol(`$tab1, tab2, i, j$`)` qui copie le contenu du tableau $tab1$ en $tab2$ et **qui inverse les positions** i **et** j. Le tableau $tab1$ ne doit pas être modifié, mais il est important de modifier $tab2$.

Par exemple,

```
1 s1=[0,10,20,30,40,50]
2 s2=Array.new(6)
3 changeSol(s1,s2,1,3)
4 p s2
```

devrait afficher :

```
[0,30,20,10,40,50]
```

Etape 4 : écrire une fonction `deltaChangeSol(`$tab1, tab2, i, j$`)` qui appele `changeSol(`$tab1, tab2, i, j$`)` et renvoie différence de coût entre $tab2$ et $tab1$. La valeur renvoyée est `calcCout(`$tab2$`)-calcCout(`$tab1$`)`.
Etape 5 : écrire une fonction `trouverMeilleureInversion(`$tab1, tab2$`)` qui permet de trouver le valeurs de i et j pour lesquelles `deltaChangeSol(`$tab1, tab2, i, j$`)` renvoie le plus petit coût. On dit que le tableau résultant $tab2$ représente la *meilleure solution voisine* à $tab1$, en considérant un opérateur d'inversion.
Etape 6 : écrire une fonction `trouverMeilleureSolVoisine(`$tab1, tab2$`)` qui cherche la meilleure solution $tab2$ voisine à $tab1$. A partir de $tab1$, on peut obtenir $tab2$ avec les opérateurs suivantes :

- l'inversion d'un élément i avec un élément j : on suit les étapes 4-5 et la fonction `trouverMeilleureInversion(`*ta*
- l'ajout d'un élément i à la position j de $tab1$. Vous pouvez avoir besoin d'une routine `trouverMeilleureInsertio` écrite dans le même esprit que `trouverMeilleureInversion(`$tab1, tab2$`)`. La différence est que la nouvelle méthode ne cherche pas d'inverser i avec j mais d'insérer i la position j
- la suppression de l'élément à la position j.

Etape 7 : utiliser la fonction `trouverMeilleureSolVoisine(`$tab1, tab2$`)` de manière itérative. L'objectif est de faire diminuer progressivement le cout, i.e., itération par itération. À chaque itération, on passe de la solution courante à une solution voisine en choisissant l'opération qui minimise le coût. Cette opération peut être une inversion, une insertion ou une suppression, voir l'étape précédente.
Utiliser toute astuce et toute idée pour faire minimiser le coût à long terme. Par exemple, il faut faire attention à éviter de choisir le même i et le même j à chaque itération. [1]
Etape 8 : Exécuter 100 itérations (voir étape ci-dessus) et afficher la solution finale, ainsi que son coût.

Attention : il faut respecter strictement le format du fichier d'entrée. La notation va prendre en compte la qualité de la solution renvoyée pour de différents fichiers d'entrée.

1. Voir, par exemple, les principes de la "recherche locale" ou de la "recherche Tabou". Il suffit de cherchez ces mots clés sur Internet, voir par exemple `http://en.wikipedia.org/wiki/Tabu_search`