Ecole Normale Supérieure
Algorithmique et programmation 2008–2009

Projet 1:
Arbres couvrants minimaux par l'algorithme de Kruskal

Rappels

Soit $G = (S, A, w)$ un graphe connexe, pondéré, non orienté de sommets S et d'arêtes $A \subseteq S^2$ de poids $w : A \to \mathbb{R}$. Un arbre couvrant minimal est un sous-ensemble de A' de A tel que :
- chaque sommet de S est touché par une arête de A' (A' couvre S),
- (S, A') est connexe et sans cycle (c'est un arbre),
- le poids total de l'arbre $\sum_{a \in A'} w(a)$ est minimal.

L'algorithme de Kruskal (voir [CLR, §24.2]) construit un arbre couvrant minimal en maintenant une partition $P \subseteq \mathcal{P}(S)$ des nœuds S. Pour $s \in S$, on note $P(s)$ l'unique partie de P contenant s : $P(s) \in P$ et $s \in P(s)$. Voici le pseudo-code de l'algorithme de Kruskal :

$A' = \emptyset$
$P = \{ \{s\} \mid s \in S \}$
`trier` A `par ordre croissant de poids` w
`pour chaque` $\{s, t\} \in A$
  `si` $P(s) \neq P(t)$
    $A' = A' \cup \{\{s, t\}\}$
    `fusionner` $P(s)$ `et` $P(t)$ `dans` P
  `fin si`
`fin pour`

On rappelle également que les opérations de comparaison $P(s) = P(t)$ et de fusion de $P(s)$ et $P(t)$ peuvent s'implanter efficacement grâce à la structure de données *union-find* (voir [CLR, §22.3]). Si on utilise une telle structure, ainsi qu'un algorithme de tri en $\mathcal{O}(n \log n)$, l'algorithme de Kruskal est en $\mathcal{O}(|A| \log |A|)$.

À faire

Vous écrirez une fonction C qui calcule un arbre couvrant minimal par l'algorithme de Kruskal. L'argument de la fonction sera un graphe connexe de taille arbitraire sous forme de listes d'adjacence (efficace pour représenter les graphes peu denses). Les poids seront de type `int`. La sortie sur écran sera la liste des arêtes qui forment l'arbre (dans un ordre quelconque) suivie du poids de l'arbre. Votre fonction utilisera la structure de données *union-find* ainsi qu'un algorithme de tri que vous devrez également programmer.

Vous écrirez également un programme de démonstration qui calcule des arbres couvrants minimaux sur des graphes connexes générés aléatoirement. Le programme prendra en argument en ligne de commande le nombre de sommets S et le nombre d'arêtes A des graphes à générer.

Enfin, vous ferez quelques expériences en variant $|A|$ et $|S|$ pour vérifier expérimentalement le coût théorique (attention au coût de la génération du graphe et de la sortie sur écran).

Bonus

Cette partie est optionnelle. Il s'agit de trouver une solution approchée au problème du voyageur de commerce grâce à un arbre couvrant minimal.

Soit $G = (S, A, w)$ un graphe complet ($A = S^2$), pondéré, non orienté et qui vérifie l'inégalité triangulaire : $\forall s, t, u \in S$, $w(\{s, u\}) \leq w(\{s, t\}) + w(\{t, u\})$. On cherche à trouver un circuit, c'est à dire un chemin $s_1, \ldots, s_n \in S$ passant une et une seule fois par chaque sommet, de poids $\sum_{i=1}^{n} w(\{s_i, s_{i+1}\})$

minimal (avec la convention $s_{n+1} = s_1$). On peut montrer (ce qu'on ne demande pas, voir [CLR, §37.2.1]) que le parcours préfixe d'un arbre couvrant minimal de G est un circuit de poids au plus double du poids minimal.

Vous écrirez une fonction C qui prend en argument un graphe complet de taille arbitraire et calcule un circuit grâce à cette heuristique. La sortie sur écran sera la liste, dans l'ordre, des sommets à visiter suivie du poids du circuit. Les poids seront de type flottant `double`.

Vous écrirez également un programme de démonstration qui teste votre fonction sur des graphes aléatoires. Le programme prendra en argument en ligne de commande le nombre de sommets des graphes à générer. Les graphes seront générés de la manière suivante (afin de s'assurer que l'inégalité triangulaire est vérifié) : pour chaque sommet $s_i \in S$, on choisit une position aléatoire $(x_i, y_i) \in \mathbb{R}^2$ dans le plan ; chaque arête a alors pour poids la distance euclidienne entre ses extrémités :

$$w(\{s_i, s_j\}) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Références

[CLR] T. Cormen, C. Leiserson, and R. Rivest. *Introduction à l'algorithmique.* Dunod, 1994.

Ecole Normale Supérieure
Algorithmique et programmation 2008–2009

Projet 2:
Arbres couvrants minimaux par l'algorithme de Prim

Rappels

Soit $G = (S, A, w)$ un graphe connexe, pondéré, non orienté de sommets S et d'arêtes $A \subseteq S^2$ de poids $w : A \to \mathbb{R}$. Un arbre couvrant minimal est un sous-ensemble de A' de A tel que :

- chaque sommet de S est touché par une arête de A' (A' couvre S),
- (S, A') est connexe et sans cycle (c'est un arbre),
- le poids total de l'arbre $\sum_{a \in A'} w(a)$ est minimal.

L'algorithme de Prim (voir [CLR, §24.2]) construit un arbre couvrant minimal de manière incrémentale "gloutonne". Il part d'un sommet r arbitraire puis fait grossir A' en y adjoignant une arête de poids minimal qui le laisse connexe et sans cycle. Il utilise pour cela une file de priorité $P : S \to \mathbb{R} \cup \{\infty\}$ des sommets qu'il reste à ajouter pour que l'arbre soit couvrant. P est une fonction partielle : on notera $s \in P$ pour indiquer que P est définie sur s, et $P \neq \emptyset$ pour indiquer que P est définie sur au moins un élément. Voici le pseudo-code de l'algorithme de Prim :

```
A' = ∅
choisir r ∈ S
P(r) = 0 et ∀s ≠ r, P(s) = ∞
tant que P ≠ ∅
  s = extraire-min P
  pour chaque t tel que {s,t} ∈ A
    si t ∈ P et w({s,t}) < P(t)
      A' = A' ∪ {{s,t}}
      P(t) = w({s,t})
    fin si
  fin pour
fin tant que
```

Si la file de priorité est implantée à l'aide de la structure de données des *tas de Fibonacci* (voir [CLR, §21]), les fonctions `extraire-min` et $P(t) = w(\{s,t\})$ (qui ne peut que diminuer la priorité d'un nœud) se font respectivement en $\mathcal{O}(\log |S|)$ et $\mathcal{O}(1)$. On obtient un coût total $\mathcal{O}(|A| + |S| \log |S|)$ pour l'algorithme de Prim.

À faire

Vous écrirez une fonction C qui calcule un arbre couvrant minimal par l'algorithme de Prim. L'argument de la fonction sera un graphe connexe de taille arbitraire sous forme de listes d'adjacence (efficace pour représenter les graphes peu denses). Les poids seront de type `int`. La sortie sur écran sera la liste des arêtes qui forment l'arbre (dans un ordre quelconque) suivie du poids de l'arbre. Votre fonction utilisera la structure de données des *tas de Fibonacci* que vous devrez également programmer.

Vous écrirez également un programme de démonstration qui calcule des arbres couvrants minimaux sur des graphes connexes générés aléatoirement. Le programme prendra en argument en ligne de commande le nombre de sommets S et le nombre d'arêtes A des graphes à générer.

Enfin, vous ferez quelques expériences en variant $|A|$ et $|S|$ pour vérifier expérimentalement le coût théorique (attention au coût de la génération du graphe et de la sortie sur écran).

Bonus

Cette partie est optionnelle. Il s'agit de trouver une solution approchée au problème du voyageur de commerce grâce à un arbre couvrant minimal.

Soit $G = (S, A, w)$ un graphe complet ($A = S^2$), pondéré, non orienté et qui vérifie l'inégalité triangulaire : $\forall s, t, u \in S$, $w(\{s, u\}) \leq w(\{s, t\}) + w(\{t, u\})$. On cherche à trouver un circuit, c'est à dire un chemin $s_1, \ldots, s_n \in S$ passant une et une seule fois par chaque sommet, de poids $\sum_{i=1}^{n} w(\{s_i, s_{i+1}\})$ minimal (avec la convention $s_{n+1} = s_1$). On peut montrer (ce qu'on ne demande pas, voir [CLR, §37.2.1]) que le parcours préfixe d'un arbre couvrant minimal de G est un circuit de poids au plus double du poids minimal.

Vous écrirez une fonction C qui prend en argument un graphe complet de taille arbitraire et calcule un circuit grâce à cette heuristique. La sortie sur écran sera la liste, dans l'ordre, des sommets à visiter suivie du poids du circuit. Les poids seront de type flottant `double`.

Vous écrirez également un programme de démonstration qui teste votre fonction sur des graphes aléatoires. Le programme prendra en argument en ligne de commande le nombre de sommets des graphes à générer. Les graphes seront générés de la manière suivante (afin de s'assurer que l'inégalité triangulaire est vérifié) : pour chaque sommet $s_i \in S$, on choisit une position aléatoire $(x_i, y_i) \in \mathbb{R}^2$ dans le plan ; chaque arête a alors pour poids la distance euclidienne entre ses extrémités :

$$w(\{s_i, s_j\}) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Références

[CLR] T. Cormen, C. Leiserson, and R. Rivest. *Introduction à l'algorithmique.* Dunod, 1994.

Ecole Normale Supérieure
Algorithmique et programmation 2008–2009

Projet 3:
Plus court chemins par l'algorithme de Dijkstra

Rappels

Soit $G = (S, A, w)$ un graphe pondéré, orienté de sommets S et d'arêtes $A \subseteq S^2$ de poids positifs ou nuls $w : A \to \mathbb{R}^+$. Étant donné un sommet $s \in S$, on cherche à calculer les plus court chemins dans G avec pour origine s, c'est à dire, la fonction : $\delta_s : S \to \mathbb{R}^+ \cup \{\infty\}$ définie par :

$$\delta_s(t) = \min \left\{ \sum_{i=1}^{n-1} w(s_i, s_{i+1}) \mid s_1 = s,\ s_n = t,\ \forall i\, (s_i, s_{i+1}) \in A \right\}$$

Ce minimum est ∞ si il n'y a aucun chemin de s à t, et 0 si $s = t$.

L'algorithme de Dijkstra (voir [CLR, 25.2]) est un algorithme glouton permettant de calculer δ_s. Il utilise une liste de priorité $P : S \to \mathbb{R}^+ \cup \{\infty\}$ des sommets restant à traiter. P est une fonction partielle : on notera $s \in P$ pour indiquer que P est définie sur s, et $P \neq \emptyset$ pour indiquer que P est définie sur au moins un élément. Voici le pseudo-code de l'algorithme de Dijkstra :

```
∀t, P(t) = w(s,t) si (s,t) ∈ A ; P(t) = ∞ sinon
δ_s(0) = 0 et ∀t ≠ s, δ_s(t) = ∞
tant que P ≠ ∅
  u = extraire-min P
  pour chaque v tel que (u,v) ∈ A
    si δ_s(v) > δ_s(u) + w(u,v)
      δ_s(v) = δ_s(u) + w(u,v)
      P(v) = δ_s(v)
    fin si
  fin pour
fin tant que
```

Si la file de priorité est implantée à l'aide de la structure de données des *tas de Fibonacci* (voir [CLR, §21]), les fonctions `extraire-min` et $P(v) = \delta_s(v)$ (qui ne peut que diminuer la priorité d'un nœud) se font respectivement en $\mathcal{O}(\log |S|)$ et $\mathcal{O}(1)$. On obtient un coût total $\mathcal{O}(|A| + |S| \log |S|)$ pour l'algorithme de Dijkstra.

À faire

Vous écrirez une fonction C qui calcule le plus court chemin à origine unique par l'algorithme de Dijsktra. L'argument de la fonction sera un graphe de taille arbitraire sous forme de listes d'adjacence (efficace pour représenter les graphes peu denses) ainsi qu'un nœud s. Les poids seront de type `int`. La sortie sur écran sera, pour chaque $t \in S$, la liste ordonnée des sommets formant un chemin de poids minimum entre s et t ainsi que son poids, ou `+oo` si t n'est pas accessible depuis s. Votre fonction utilisera la structure de données des *tas de Fibonacci* que vous devrez également programmer.

Vous écrirez également un programme de démonstration qui calcule les plus court chemins sur des graphes générés aléatoirement. Le programme prendra en argument en ligne de commande le nombre de sommets S et le nombre d'arêtes A des graphes à générer.

Enfin, vous ferez quelques expériences en variant $|A|$ et $|S|$ pour vérifier expérimentalement le coût théorique (attention au coût de la génération du graphe et de la sortie sur écran).

Références

[CLR] T. Cormen, C. Leiserson, and R. Rivest. *Introduction à l'algorithmique.* Dunod, 1994.

Ecole Normale Supérieure
Algorithmique et programmation 2008–2009

Projet 4:
Plus court chemins par l'algorithme de Johnson

Rappels

Soit $G = (S, A, w)$ un graphe pondéré, orienté de sommets S et d'arêtes $A \subseteq S^2$ de poids $w : A \to \mathbb{R}$. On cherche à calculer le plus court chemin dans G entre tout couple de sommets, c'est à dire, la fonction : $\delta : S^2 \to \mathbb{R}^+ \cup \{\infty\}$ définie par :

$$\delta(s,t) = \min \left\{ \sum_{i=1}^{n-1} w(s_i, s_{i+1}) \mid s_1 = s,\ s_n = t,\ \forall i\, (s_i, s_{i+1}) \in A \right\}$$

Ce minimum est ∞ si il n'y a aucun chemin de s à t, et 0 si $s = t$. On suppose de plus que G ne contient aucun cycle $s_1, \ldots, s_n = s_1$, $(s_i, s_{i+1}) \in A$ de poids total $\sum_{i=1}^{n-1} w(s_i, s_{i+1})$ strictement négatif, sinon cette fonction n'est pas définie !

L'idée de l'algorithme de Johnson (voir [CLR, 26.3]) est de se ramener à $|S|$ calculs de plus court chemins à origine unique c'est à dire, de fonctions $\delta_s : t \mapsto \delta(s,t)$. L'algorithme de Bellman–Ford calcule un δ_s en $\mathcal{O}(|A||S|)$. L'algorithme de Dijkstra calcule un δ_s en $\mathcal{O}(|A| \log |S|)$, ce qui est mieux, mais ne fonctionne que si les poids w sont tous positifs. De plus, l'algorithme de Dijkstra utilise une liste de priorité $P : S \to \mathbb{R}^+ \cup \{\infty\}$ des sommets restant à traiter. P est une fonction partielle : on notera $s \in P$ pour indiquer que P est définie sur s, et $P \neq \emptyset$ pour indiquer que P est définie sur au moins un élément.

On rappelle maintenant ces deux algorithmes :

```
BELLMAN–FORD(s)
δ_s(0) = 0 et ∀t ≠ s, δ_s(t) = ∞
pour i de 1 à |S| − 1
  pour chaque (u, v) ∈ A
    δ_s(v) = min(δ_s(v), δ_s(u) + (u, v))
  fin pour
fin pour
```

```
DIJKSTRA(s)
∀t, P(t) = w(s, t) si (s, t) ∈ A ; P(t) = ∞ sinon
δ_s(0) = 0 et ∀t ≠ s, δ_s(t) = ∞
tant que P ≠ ∅
  u = extraire-min P
  pour chaque v tel que (u, v) ∈ A
    si δ_s(v) > δ_s(u) + w(u, v)
      δ_s(v) = δ_s(u) + w(u, v)
      P(v) = δ_s(v)
    fin si
  fin pour
fin tant que
```

Si la file de priorité P est implantée à l'aide de la structure de données des *tas binaires* (voir [CLR, §7.5]), les fonctions `extraire-min` et $P(v) = \delta_s(v)$ (qui ne peut que diminuer la priorité d'un nœud) se font en $\mathcal{O}(\log |S|)$. Il est possible de faire mieux grâce aux *tas de Fibonaci*, ce qu'on ne demandera pas (cela donnerait un coût $\mathcal{O}(|A| + |S| \log |S|)$ pour l'algorithme de Dijkstra).

L'astuce de l'algorithme de Johnson consiste à n'utiliser qu'une seule fois l'algorithme de Bellman–Ford, pour calculer δ_0 sur le graphe $G' = (S', A', w')$ dérivé de G de la manière suivante. Un sommet 0 est ajouté $S : S' = S \cup \{0\}$. Ce sommet est lié à tous les sommets de S par une arête de poids nul : $A' = A \cup \{(0, s) \mid s \in S\}$ et $w'(0, s) = 0$. δ_0 est donc la fonction de plus court chemins avec pour origine ce nouveau sommet. On construit alors le graphe $G'' = (S, A, w'')$ par *repondération* de G : on pose $w''(u, v) = w(u, v) + \delta_0(u) - \delta_0(v)$. G'' a la propriété remarquable suivante : ses arêtes sont toutes de poids positifs. On peut donc calculer les plus court chemins δ'' sur G'' par $|S|$ applications de l'algorithme de Dijkstra. Enfin, les plus court chemins δ sur G peuvent se déduire de δ'' en annulant la repondération : $\delta(u, v) = \delta''(u, v) + \delta_0(v) - \delta_0(u)$.

À faire

Vous écrirez une fonction C qui calcule les plus cours chemins entre tout couple de sommets par l'algorithme de Johnson. L'argument de la fonction sera un graphe de taille arbitraire sous forme de listes d'adjacence (efficace pour représenter les graphes peu denses). Les poids seront de type `int`. La sortie sur écran sera, pour chaque $s, t \in S$, la liste ordonnée des sommets formant un chemin de poids minimum entre s et t ainsi que son poids, ou `+oo` si t n'est pas accessible depuis s. Votre fonction utilisera la structure de données *tas binaire* que vous devrez également programmer.

Vous écrirez également un programme de démonstration qui calcule les plus court chemins sur des graphes générés aléatoirement. Le programme prendra en argument en ligne de commande le nombre de sommets S et le nombre d'arêtes A des graphes à générer.

Enfin, vous ferez quelques expériences en variant $|A|$ et $|S|$ pour vérifier expérimentalement le coût théorique (attention au coût de la génération du graphe et de la sortie sur écran).

Références

[CLR] T. Cormen, C. Leiserson, and R. Rivest. *Introduction à l'algorithmique.* Dunod, 1994.

Ecole Normale Supérieure
Algorithmique et programmation 2008–2009

Projet 5:
Plus court chemins par l'algorithme de Gabow

Rappels

Soit $G = (S, A, w)$ un graphe pondéré, orienté de sommets S et d'arêtes $A \subseteq S^2$ de poids entiers positifs ou nuls $w : A \to \mathbb{N}^+$. Étant donné un sommet $s \in S$, on cherche à calculer les plus court chemins dans G avec pour origine s, c'est à dire, la fonction : $\delta_s : S \to \mathbb{N}^+ \cup \{\infty\}$ définie par :

$$\delta_s(t) = \min \left\{ \sum_{i=1}^{n-1} w(s_i, s_{i+1}) \mid s_1 = s,\ s_n = t,\ \forall i\, (s_i, s_{i+1}) \in A \right\}$$

Ce minimum est ∞ si il n'y a aucun chemin de s à t, et 0 si $s = t$.

Nous rappelons d'abord l'algorithme de Dijkstra pour le calcul de δ_s :

$\forall t,\ P(t) = w(s,t)$ `si` $(s,t) \in A$; $P(t) = \infty$ `sinon`
$\delta_s(0) = 0$ `et` $\forall t \neq s,\ \delta_s(t) = \infty$
`tant que` $P \neq \emptyset$
  u `= extraire-min` P
  `pour chaque` v `tel que` $(u,v) \in A$
    `si` $\delta_s(v) > \delta_s(u) + w(u,v)$
      $\delta_s(v) = \delta_s(u) + w(u,v)$
      $P(v) = \delta_s(v)$
    `fin si`
  `fin pour`
`fin tant que`

L'algorithme de Dijkstra utilise une liste de priorité $P : S \to \mathbb{N}^+ \cup \{\infty\}$. Une remarque importante est que, si on connaît une borne M à priori sur les valeurs de δ_s, P peut s'implanter efficacement par un couple (m, T) où $m = \min_u P(u)$ et le tableau $T : [m, M] \cup \{\infty\} \to \mathcal{P}(S)$ associe à chaque priorité la liste (chaînée) des nœuds qui ont cette priorité. L'algorithme de Dijkstra a alors un coût en $\mathcal{O}(|A| + M)$.

On note maintenant W le poids maximum : $W = \max\{w(u,v) \mid (u,v) \in A\}$, et $k = \lceil \log W \rceil$ le nombre de bits qu'il faut pour représenter W. On note de plus $w^i(u,v) = \lfloor 2^{i-k} w(u,v) \rfloor$ les i bits de poids forts de la représentation binaire de $w(u,v)$. L'idée de l'algorithme de Gabow (voir [CLR, exercice 25-4]) est de calculer itérativement les fonctions δ_s^1 à δ_s^k correspondant aux plus court chemins d'origines s dans les graphes $G^1 = (S, A, w^1)$ à $G^k = (S, A, w^k)$. Si on note $\hat{w}^i(u,v) = w_i(u,v) + 2\delta_s^{i-1}(u) - 2\delta_s^{i-1}(v)$, et $\hat{\delta}_s^i$ les plus court chemins d'origines s dans les graphes $\hat{G}^i = (S, A, \hat{w}^i)$, alors on a (ce qu'on ne démontrera pas) :

- $\delta_s^i(u) = \hat{\delta}_s^i(u) + 2\delta_s^{i-1}(v)$
- $\hat{\delta}_s^i(u) \leq |A|$

On note alors que le calcul de δ_s^1 ainsi que celui de δ_s^i à partir de δ_s^{i-1} se ramène à un calcul de plus court chemin dans un graphe dont les plus court chemins (d'origine s) sont bornés par $|A|$. D'où un coût total en $\mathcal{O}(|A|k)$ pour l'algorithme de Gabow.

À faire

Vous écrirez une fonction C qui calcule le plus court chemin à origine unique par l'algorithme de Gabow. L'argument de la fonction sera un graphe de taille arbitraire sous forme de listes d'adjacence (efficace pour représenter les graphes peu denses) ainsi qu'un nœud s. Les poids seront de type `int`. La sortie sur écran

sera, pour chaque $t \in S$, la liste ordonnée des sommets formant un chemin de poids minimum entre s et t ainsi que son poids, ou `+oo` si t n'est pas accessible depuis s.

Vous écrirez également un programme de démonstration qui calcule les plus court chemins sur des graphes générés aléatoirement. Le programme prendra en argument en ligne de commande le nombre de sommets S et le nombre d'arêtes A des graphes à générer, ainsi que le poids maximal W autorisé sur les arêtes.

Enfin, vous ferez quelques expériences en variant $|A|$, $|S|$ et W pour vérifier expérimentalement le coût théorique (attention au coût de la génération du graphe et de la sortie sur écran).

Références

[CLR] T. Cormen, C. Leiserson, and R. Rivest. *Introduction à l'algorithmique.* Dunod, 1994.

Ecole Normale Supérieure
Algorithmique et programmation 2008–2009

Projet 6:
Flot maximal par l'algorithme de Malhotra, Kumar, Maheshwari

Rappels

Soit $G = (S, A, c)$ un graphe pondéré, orienté de sommets S et d'arêtes $A \subseteq S^2$ de capacités positives ou nulles $c : A \to \mathbb{R}^+$. On suppose A symétrique : $(u, v) \in A \Longrightarrow (v, u) \in A$ (en ajoutant au besoin des arêtes de capacité nulle à A). De plus, deux sommets de S sont distingués : la source s et le puits t. Un flot est une valuation des arêtes $f : A \to \mathbb{R}^+$ qui vérifie les contraintes de capacité et la conservation du flot :

$$\forall a \in A,\ f(a) \leq c(a)$$
$$\forall u \in S \setminus \{s, t\},\ \textstyle\sum_{(v,u)\in A} f(v, u) = \sum_{(u,v)\in A} f(u, v)$$

On chercher à maximiser le débit total du flot, c'est à dire :

$$\sum_{(s,u)\in A} f(s, u) = \sum_{(u,t)\in A} f(u, t)$$

Étant donné un flot f et un nœud $u \neq s, t$, on appelle *potentiel* en u le débit supplémentaire $\rho(u)$ qu'il est possible de faire passer par u :

$$\rho(u) = \min\left(\sum_{(v,u)\in A} c(v, u) - f(v, u),\ \sum_{(u,v)\in A} c(u, v) - f(u, v)\right)$$

De plus, on appelle *potentiel de référence* ρ la valeur minimale de ce potentiel $\rho = \min_u \rho(u)$ et *nœud de référence* un nœud r qui atteint ce minimum : $\rho(r) = \rho$.

L'algorithme de Malhotra, Kumar et Maheshwari (voir [Wilf, §3.7]) construit un flot maximal de la manière suivante. Il part du flot nul $\forall a \in A,\ f(a) = 0$ et détermine un nœud de référence r. Le flot est mis à jour pour faire passer un débit ρ supplémentaire par r. Cela se fait en distribuant le flot ρ sur toutes les arêtes sortantes (r, u_i) de r. Si on note $u_1, \ldots, u_n$ les nœuds successeurs de r, on commencera par donner le maximum possible $\rho_1 = \min(\rho, c(r, u_1) - f(r, u_1))$ à u_1 puis, le maximum de ce qui reste $\rho_2 = \min(\rho - \rho_1, c(r, u_2) - f(r, u_2))$ à u_2, jusqu'à épuisement du flot. Le flot arrivant à chaque voisin u_i de $\rho(f)$ est alors distribué à ses voisins et, de proche en proche (par un parcours en largeur), jusqu'à t. Le flot ρ est distribué de manière similaire depuis r jusqu'à s, en utilisant les arêtes entrantes au lieu des arêtes sortantes. L'opération est répétée tant qu'il reste un potentiel de référence non nul. Comme à chaque étape le nœud de référence voit son potentiel annulé, il n'est pas possible d'obtenir deux fois le même nœud de référence. L'algorithme a un coût total en $\mathcal{O}(|S|^3)$.

À faire

Vous écrirez une fonction C qui calcule le flot maximal par l'algorithme de Malhotra, Kumar, Maheshwari. L'argument de la fonction sera un graphe de taille arbitraire sous forme de listes d'adjacence (efficace pour représenter les graphes peu denses) ainsi que deux nœuds s et t distincts. Les capacités seront de type `int`. La sortie sur écran sera un flot maximal f : vous donnerez la liste des arêtes (u, v) telles que $f(u, v) \neq 0$, avec leur débit associé $f(u, v)$ et leur capacité $c(u, v)$, ainsi que le débit total de f.

Vous écrirez également un programme de démonstration qui calcule des flots maximaux sur des graphes générés aléatoirement. Le programme prendra en argument en ligne de commande le nombre de sommets S et le nombre d'arêtes A des graphes à générer. De plus, le programme vérifiera que la sortie de l'algorithme est bien toujours un flot (conditions de capacité et conservation du flot).

Enfin, vous ferez quelques expériences en variant $|A|$ et $|S|$ pour vérifier expérimentalement le coût théorique (attention au coût de la génération du graphe, de la vérification et de la sortie sur écran).

Références

[Wilf] H. Wilf. *Algorithms and Complexity.* A K Peters, Ltd 1994. http://www.cis.upenn.edu/~wilf/AlgComp3.html

Ecole Normale Supérieure
Algorithmique et programmation 2008–2009

Projet 7:
Multiplication de matrices

Multiplication de deux matrices

Étant données deux matrices M de taille $m \times n$ et N de taille $n \times p$, la méthode naïve calcule $M \times N$ en $\mathcal{O}(m \times n \times p)$ (coût cubique). L'algorithme récursif de Strassen (voir [CLR, 31.2]) est plus malin. Supposons d'abord m, n, p pairs. Les matrices M et N sont coupées chacune en quatre blocs de taille $(m/2) \times (n/2)$ et $(n/2) \times (p/2)$:

$$M = \begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix} \qquad N = \begin{bmatrix} N_{11} & N_{12} \\ N_{21} & N_{22} \end{bmatrix}$$

On calcule ensuite récursivement les 7 sous-produits suivants :

$$\begin{cases} X_1 = (M_{11} + M_{22}) \times (N_{11} + N_{22}) \\ X_2 = (M_{21} + M_{22}) \times N_{11} \\ X_3 = M_{11} \times (N_{12} - N_{22}) \\ X_4 = M_{22} \times (N_{21} - N_{11}) \\ X_5 = (M_{11} + M_{12}) \times N_{22} \\ X_6 = (M_{21} - M_{11}) \times (N_{11} + N_{12}) \\ X_7 = (M_{12} - M_{22}) \times (N_{21} + N_{22}) \end{cases}$$

Pour obtenir le produit $M \times N$, les sous-produits X_i sont composés de la manière suivante :

$$\begin{cases} P_{11} = X_1 + X_4 - X_5 + X_7 \\ P_{12} = X_3 + X_5 \\ P_{21} = X_2 + X_4 \\ P_{22} = X_1 - X_2 + X_3 + X_6 \end{cases}$$

On vérifie que l'on a bien :

$$M \times N = \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{bmatrix} \quad \text{car} \quad \begin{cases} P_{11} = M_{11} \times N_{11} + M_{12} \times N_{21} \\ P_{12} = M_{11} \times N_{12} + M_{12} \times N_{22} \\ P_{21} = M_{21} \times N_{11} + M_{22} \times N_{21} \\ P_{22} = M_{21} \times N_{12} + M_{22} \times N_{22} \end{cases}$$

Dans le cas où m, n ou p est impair, on se ramène au cas pair en ajoutant une ligne ou une colonne en bas ou à droite de M ou N (ou les deux). La récursivité s'arrête dès qu'une dimension est 1.

Séquence de multiplications

On se donne maintenant n matrices $M_1, \ldots, M_n$ de taille différentes $a_1 \times b_1, \ldots, a_n \times b_n$, mais avec la contrainte $1 \leq i < n \Longrightarrow b_i = a_{i+1}$. Ainsi, le produit $M = M_1 \times \cdots \times M_n$ est bien défini.

Pour se ramener à $n-1$ multiplication de deux matrices, il faut définir un parenthésage. L'associativité de la multiplication garantit que tous les parenthésages donnent le même résultat.[1] Par contre, ils n'ont pas tous le même coût. L'algorithme suivant (voir [CLR, 16.4]) trouve le coût optimal en $\mathcal{O}(n^3)$ par une méthode de programmation dynamique :

[1] C'est vrai dans $\mathbb{R}$, $\mathbb{Z}$, mais aussi avec le type `int` qui calcule modulo 2^{32} ou 2^{64}. Par contre cela n'est pas vrai pour les nombres à virgule flottante.

```
pour i de 1 à n
  c[i,i] = 0
fin pour
pour j de 1 à n-1
  pour i de 1 à n-j
    c[i,i+j] = min { c[i,i+k] + c[i+k+1,i+j] + t(a_i, b_{i+k}, b_{i+j}) | k ∈ [0,j-1] }
  fin pour
fin pour
retourner c[1,n]
```

où la fonction auxiliaire $t(m, n, p)$ (à déterminer !) calcule le coût de la multiplication d'une matrice $m \times n$ par une matrice $n \times p$. Au cours de l'algorithme, on maintient dans $c[i, j]$, pour $i \leq j$, le coût optimal de $M_i \times \cdots \times M_j$.

À faire

Vous programmerez une fonction qui multiplie deux matrices de dimensions quelconques par la méthode de Strassen. Les coefficients seront de type `int`.

Vous programmerez ensuite une fonction qui prend en argument un nombre arbitraire de matrices de tailles arbitraires, calcule le parenthésage optimal, puis calcule la matrice produit par la méthode de Strassen. La sortie sur écran sera la matrice produit ainsi que le parenthésage choisi par l'algorithme dynamique.

Vous écrirez également un programme de démonstration qui calcule le produit de plusieurs matrices aléatoires de dimensions aléatoires. Le programme prendra en argument en ligne de commande le nombre de matrices à multiplier ainsi que les dimensions minimales et maximales des matrices.

Enfin, vous programmerez une fonction qui calcule le produit de manière naïve (produit de deux matrices par l'algorithme cubique, et parenthésage du produit de gauche à droite). Vous écrirez un programme de test automatique qui vérifie que l'implantation maligne et l'implantation naïve donnent le même résultat sur des jeux de matrices aléatoires.

Références

[CLR] T. Cormen, C. Leiserson, and R. Rivest. *Introduction à l'algorithmique.* Dunod, 1994.