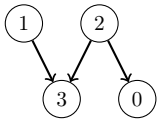
INF4705 — Analyse et conception d'algorithmes
TP 2

Ce travail pratique se répartit sur deux séances de laboratoire, traite de l'algorithmique des graphes et porte sur l'analyse et la conception d'algorithmes développés suivant différents patrons de conception.

1 Problématique

Un tri topologique d'un graphe orienté sans cycle, DAG ("directed acyclic graph"), est une liste des sommets du graphe qui maintient l'ordre défini par les arcs. Formellement, soit $G = (S, A)$ un DAG défini par l'ensemble de sommets $S = \{1, \ldots, n\}$ et l'ensemble d'arcs $A \subset S \times S$; un tri topologique de G est une permutation σ de S telle que $(i, j) \in A \rightarrow \sigma(i) < \sigma(j)$. On peut calculer un tri topologique du graphe G par un parcours de graphe qui s'exécute en temps linéaire par rapport au nombre d'arcs dans le graphe (voir section 7.6 des notes du cours). Pour ce travail pratique, on s'intéresse au nombre total des permutations possibles appelé aussi nombre d'extensions linéaires qu'on notera $\#P(G)$(ou simplement $\#P$). La figure 1 énumère les extensions linéaires (à droite) pour un petit graphe (à gauche).



- (2,1,3,0)
- (2,1,0,3)
- (2,0,1,3)
- (1,2,0,3)
- (1,2,3,0)

FIGURE 1 – DAG G=(S,A), S={0,1,2,3}, A={(1,3),(2,0),(2,3)}. #P=5

Calculer le nombre d'extensions linéaires d'un DAG est utile pour les problèmes d'ordonnancement où on cherche la probabilité de l'occurrence d'un élément x avant l'élément y dans une solution. Dans ce cas, il suffit de calculer le ratio

$$Pr(x < y) = \frac{\#P((S, A \cup \{(x, y)\}))}{\#P((S, A))}.$$

2 Implantation

Trois algorithmes seront implantés, mettant en pratique les patrons de conception : vorace, programmation dynamique et retour arrière ("backtracking").

2.1 Algorithme vorace

L'algorithme vorace décompose le graphe G en chaînes de taille maximum. Une chaîne $c \subset S$ est un sous ensemble de S tel que toute paire d'éléments

$u, v \in c$ sont comparables (reliés par un chemin de u vers v ou de v vers u dans G). À chaque itération, l'algorithme construit une chaîne avec un maximum de sommets (candidats). Les sommets de la chaîne ainsi que tous les arcs incidents aux sommets sont retirés de G avant de passer à l'itération suivante. L'algorithme s'arrête lorsqu'il n'y plus de sommets dans G. L'entropie de la distribution obtenue par la décomposition de l'algorithme vorace sert à approximer le nombre d'extensions linéaires de G par $2^{\frac{1}{2}nH(G)}$ où $H(G) = \sum_{i=1}^{k}(-\frac{|c_i|}{|S|}\log\frac{|c_i|}{|S|})$ et $c_1, c_2, \ldots, c_k$ est le résultat de la décomposition en chaînes vorace.

Algorithm 1: longestChain

input: directed acyclic graph $G(V, A)$
output: longest chain c in G

forall the $v \in V$ **do**
  pred[v] $\leftarrow -1$;
initialize queue Q with $\{v \in V : \text{indegree}(v) = 0\}$;
last $\leftarrow -1$;
while Q *is not empty* **do**
  remove vertex u from the front of Q;
  last $\leftarrow u$;
  forall the *arcs* $(u, v) \in A$ **do**
    pred[v] $\leftarrow u$;
    if Q *contains* v **then**
      move v to the end of Q;
    else
      add v at the end of Q;
$c \leftarrow \langle\rangle$;
while *last* $\neq -1$ **do**
  push last on c;
  last $\leftarrow$ pred[last];

Algorithm 2: Décomposition vorace en chaînes

input: Poset $(P, <)$ represented as a reduced DAG G
output: list L of chains

$L \leftarrow \langle\rangle$;
$c \leftarrow$ longestChain(G);
while $c \neq \langle\rangle$ **do**
  remove all vertices in c and incident arcs from G;
  push c on L;
  $c \leftarrow$ longestChain(G);
forall the $v \in G$ **do**
  push $\langle v\rangle$ on L;

2.2 Algorithme de retour arrière

L'algorithme de tri topologique sera adapté pour faire un retour arrière pour générer les permutations possibles.

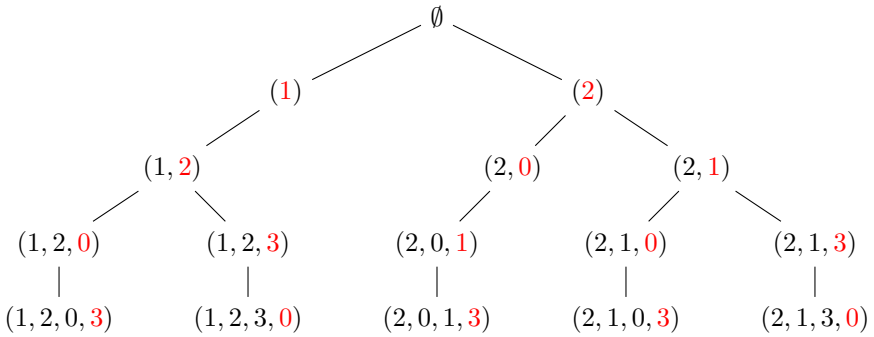


FIGURE 2 – Trace d'exécution de l'algorithme de retour arrière appliqué sur le graphe de la figure 1

Il ne sera pas nécessaire de mémoriser les permutations car on s'intéresse uniquement au nombre total de permutations. Pour chaque permutation trouvée, l'algorithme incrémente le nombre d'extensions linéaires de 1.

2.3 Algorithme de programmation dynamique

Le nombre d'extensions linéaires d'un DAG G est égal à la somme du nombre d'extensions linéaires des graphes obtenus en retirant les sommets n'ayant pas d'arcs entrants (ou les sommets n'ayant pas d'arcs sortants). Par exemple, pour le graphe de la figure 1, on a $\#P(G) = \#P(G \setminus \{1\}) + \#P(G \setminus \{2\})$. Cette propriété nous permet donc de calculer le nombre d'extensions linéaires d'un DAG par la programmation dynamique. Soit $c^1 = \langle c^1_1, \ldots, c^1_{m_1}\rangle, c^2 = \langle c^2_1, \ldots, c^2_{m_2}\rangle, \ldots, c^k = \langle c^k_1, \ldots, c^k_{m_k}\rangle$ le résultat de la décomposition en chaînes du graphe G par l'algorithme vorace. On pose $\#P_{i_1,i_2,\ldots,i_k}$ le nombre d'extensions linéaires du sous-graphe induit de G dont les sommets sont les éléments des chaînes partielles $\langle c^1_1, \ldots, c^1_{i_1}\rangle, \langle c^2_1, \ldots, c^2_{i_2}\rangle, \ldots, \langle c^k_1, \ldots, c^k_{i_k}\rangle$ et $\#P_{0,0,\ldots,0} = 1$. La formule récursive pour calculer le nombre d'extensions linéaires d'un DAG est comme suit

$$\#P_{i_1,i_2,\ldots,i_k} = \sum_{j=1}^{k} (\delta_j \times \#P_{i_1,\ldots,i_{j-1},i_j-1,i_{j+1},\ldots,i_k})$$

où $\delta_j = 1$ si $\nexists \ell \in \{1, 2, \ldots, k\}$ tel que $(c^j_{i_j}, c^\ell_{i_\ell}) \in A$ et 0 sinon. Le nombre total d'extensions linéaires est $\#P_{m_1,m_2,\ldots,m_k}$.
Considérons la décomposition en chaînes $c^1 = \langle 1, 3\rangle$ et $c^2 = \langle 2, 0\rangle$ du graphe de la figure 1. L'algorithme de programmation dynamique effectue les calculs suivants

$\#P_{2,2} = \#P_{1,2} + \#P_{2,1}$
$\#P_{1,2} = \#P_{0,2} + \#P_{1,1}$
$\#P_{1,1} = \#P_{0,1} + \#P_{1,0}$
$\#P_{2,1} = \#P_{1,1}$
Ses valeurs sont conservées dans le tableau suivant

		2	0
	1	1	1
1	1	2	3
3	1	2	**5**

Notez que la dimension du tableau correspond au nombre de chaînes dans le résultat obtenu par la décomposition en chaînes de l'algorithme vorace.

Attention : Avant d'appliquer cet algorithme de programmation dynamique, vous devez calculer la *fermeture transitive* du graphe donné en exemplaire, par exemple au moyen de cette modification à l'algorithme de Floyd :

FloydFermetureTransitive($A[1..n, 1..n]$:matrice d'adjacence)
pour $k = 1$ **à** n **faire**
pour $i = 1$ **à** n **faire**
pour $j = 1$ **à** n **faire**
$A[i, j] \leftarrow \max(A[i, j], A[i, k] \times A[k, j])$;

3 Jeu de données

Vous trouverez sur le site Moodle du cours tous les exemplaires du problème. Chaque fichier représente un graphe. Les graphes sont regroupés par paire (N,W) où N est le nombre de sommets du graphe et W est la largeur du graphe (nombre maximum de sommets mutuellement non comparables). Le format d'un fichier est "poset<N>-<W>.<L>" . La lettre "L" identifie l'exemplaire parmi dix avec une lettre de "a" à "j". Par exemple, le graphe "poset10-4.a" est le premier exemplaire de la série de graphes de taille 10 et de largeur 4. La première ligne du fichier indique le nombre de sommets du graphe et le nombre d'arcs séparés par un espace. Les lignes qui suivent sont les extrémités de chaque arc.

4 Résultats

Exécutez chacun des trois algorithmes en notant leur temps d'exécution et le nombre d'extensions linéaires trouvé, mais ne rapportez dans un tableau que la moyenne de chaque série de dix exemplaires. (Note : Les temps de calcul peuvent être élevés dépendamment de l'algorithme utilisé.)

5 Analyse et discussion

1. Tentez une analyse asymptotique du temps de calcul pour chaque algorithme.
2. Servez-vous de vos temps d'exécution pour confirmer et/ou préciser l'analyse asymptotique théorique de vos algorithmes avec la méthode hybride de votre choix (cette méthode peut varier d'un algorithme à l'autre). Justifiez ces choix.
3. Discutez des trois algorithmes en fonction de la qualité respective des solutions obtenues, de la consommation de ressources (temps de calcul, espace mémoire) et de la difficulté d'implantation.
4. Indiquez sous quelles conditions vous utiliseriez l'un de ces algorithmes plutôt que les deux autres.

6 Remise

Avant votre cinquième séance de laboratoire, vous devez faire une *remise électronique* en suivant les instructions suivantes :

1. Le dossier remis doit se nommer matricule1_matricule2_tp1 et doit être compressé sous format zip.
2. À la racine de ce dernier, on doit retrouver un rapport sous format PDF comprenant :
   — une brève description du sujet et des objectifs de ce travail (svp pas de redite de l'énoncé),
   — la description des jeux de données,
   — les résultats expérimentaux,
   — l'analyse et discussion.
3. Finalement, un script nommé *tp.sh* doit aussi être présent à la racine. Ce dernier sert à exécuter les différents algorithmes du TP. Le fonctionnement du script est décrit à la prochaine section.

7 tp.sh

Utilisation
tp.sh -a [vorace|dynamique|retourArriere] -e [path_vers_exemplaire]

Arguments optionnels :
-p Imprime le nombre d'extensions linéaires
-t Imprime le temps d'exécution

8 Barème de correction

1 pts : exposé du travail pratique
2 pts : présentation des résultats
5 pts : analyse et discussion
3 pts : les programmes (corrects,structurés,commentés,...)
2 pts : présentation générale et qualité du français