Tri topologique [gp03] - Exercice

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 21 mai 2018

Table des matières

1 Tri topologique 2
1.1 Tri topologique . . . 2
1.2 Algorithmique . . . 2
1.3 Tris topologiques linéaire . . . 4
1.4 Programmation . . . 6

2 Références générales 8

C++ - Tri topologique (Solution)

Mots-Clés Structures relationnelles ■
Requis Axiomatique objet, Modèles, Gestion des exceptions ■
Difficulté • • ◦ (1 h) ■

Objectif
Cet exercice réalise le tri topologique d'un graphe orienté acyclique.

1 Tri topologique

1.1 Tri topologique

Définition

Un **tri topologique** d'un graphe orienté acyclique $G = (S, A)$ est un ordre linéaire des sommets de G tel que si G contient l'arc (u, v), u apparaît avant v.

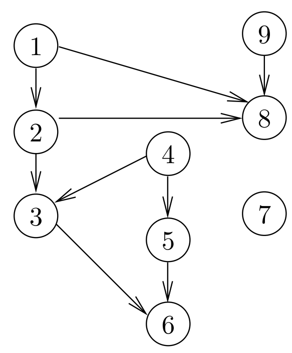
Remarque

Le tri topologique d'un graphe peut être vu comme un alignement de ses sommets le long d'une ligne horizontale tel que tous les arcs soient orientés de gauche à droite.

Attention

Le tri topologique d'un graphe orienté acyclique n'est pas forcément unique.

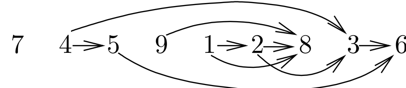
Proposez un tri topologique du graphe orienté acyclique de la figure suivante :



Solution simple

Voici trois tris topologiques possibles.

9 4 5 1 2 3 6 8 7



1.2 Algorithmique

Modifiez l'algorithme du parcours en profondeur (vu en cours) pour qu'il calcule pour chaque noeud `u` sa date de fin de traitement `ft[u]` (la date à laquelle lui et ses fils ont été traités).

Solution simple

```
PP(G)
Début
 Pour chaque sommet u de G Faire
   couleur[u] <- Blanc
 FinPour
 temps <- 0
 Pour chaque sommet u de G Faire
   Si couleur[u] = Blanc Alors
     VisiterPP(G,u,couleur,temps)
   FinSi
 FinPour
Fin

VisiterPP(G,s,couleur,temps)
Début
  couleur[s] <- Gris
  Pour chaque voisin v de s Faire
    Si couleur[v] = Blanc Alors
      VisiterPP(G,v,couleur)
    FinSi
  FinPour
  couleur[s] <- Noir
  temps <- temps+1
  ft[s] <- temps
Fin
```

Quel lien pouvez-vous faire entre les dates de fin de traitement et un tri topologique ?

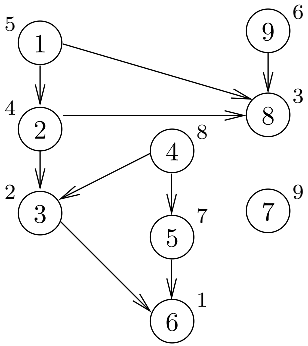
Solution simple

Si le graphe G contient l'arc (u, v), le traitement de u finira **après** le traitement de v, et on aura donc $ft[u] > ft[v]$. Trier les sommets par date de fin de traitements décroissantes nous fournit donc un tri topologique du graphe.

Dans le graphe de la première question, annotez les noeuds avec des fins de date de traitement lors d'un parcours en profondeur (« racines » successives) : noeuds 1, 9, 4 et 7.

Solution simple

On obtient le graphe suivant :



On retrouve alors le troisième des tris topologiques de la figure de la première question.

Proposez un algorithme de tri topologique.

Solution simple

On commence par parcourir le graphe en profondeur avec l'algorithme de la deuxième question, puis on trie les sommets par date de fin décroissante.

Quelle est sa complexité ?

Solution simple

La complexité de l'ensemble est donc celle du parcours $O(|S| + |A|)$ plus celle du tri $O(|S| \log(|S|))$, soit $O(|S| \log(|S|) + |A|)$.

1.3 Tris topologiques linéaire

Ce problème définit des tris topologiques de complexité linéaire.

Comment pouvez-vous améliorer votre algorithme pour qu'il soit de complexité linéaire ?

Solution simple

On stocke dans une pile les éléments dans l'ordre dans lequel leur traitement se termine. On retirera les éléments de la pile dans l'ordre inverse de celui dans lequel ils ont été insérés (une pile fonctionne toujours sur le mode « premier entré, dernier sorti »).

Écrivez l'algorithme correspondant.

Solution simple

```
PP(G)
Début
  Soit P une pile initialement vide
  Pour chaque sommet u de G Faire
    couleur[u] <- Blanc
  FinPour
  Pour chaque sommet u de G Faire
    Si couleur[u] = Blanc Alors
      VisiterPP(G,u,couleur,P)
    FinSi
  FinPour
  TantQue non PileVide(P) Faire
    u <- Dépiler(P)
    Afficher(u)
  FinTantQue
Fin
```

Avec

```
VisiterPP(G,s,couleur,P)
Début
  couleur[s] <- Gris
  Pour chaque voisin v de s Faire
    Si couleur[v] = Blanc Alors
      VisiterPP(G,v,couleur,P)
    FinSi
  FinPour
  couleur[s] <- Noir
  Empiler(P,s)
Fin
```

Montrez que votre algorithme est de complexité linéaire.

Solution simple

Utiliser une pile plutôt que simplement les dates de fin de traitement nous évite d'avoir à trier ces dates (ce qui nous coûterait $O(|S|\log(|S|))$. L'algorithme complet a ici un coût de $O(|S|+|A|)$ (le parcours est de coût $O(|S|+|A|)$ et la pile contient $|S|$ éléments, donc les dépiler est de coût $O(|S|)$).

Proposez un autre algorithme de tri topologique, basé cette fois-ci sur le fait qu'un sommet de degré entrant nul peut être placé en tête d'un tri topologique.

Solution simple

```
TriTopologique(G=(S,A))
Début
  Soit F une file initialement vide
  Pour chaque sommet u de G Faire
    degré(u) <- degré entrant de u
    Si degré(u) = 0 Alors
      Enfiler(F,u)
```

```
    FinSi
  FinPour
  TantQue non FileVide(F) Faire
    u <- Defiler(F)
    Afficher(u)
    Pour chaque voisin v de u faire
      degré(v) <- degré(v) - 1
      Si degré(v) = 0 Alors
        Enfiler(F,v)
      FinSi
    FinPour
  FinTantQue
Fin
```

Justifiez votre algorithme.

Solution simple

Un sommet qui est placé dans la file, est un sommet dont tous les prédécesseurs ont déjà été ordonnés : on peut donc le placer à son tour dans l'ordre sans risquer de violer un arc.

Donnez sa complexité.

Solution simple

La complexité de l'ensemble est une fois de plus en $O(|S| + |A|)$.

1.4 Programmation

Écrivez une procédure `topologicalSort(g,lt)` qui réalise le tri topologique d'un `Graphe g` supposé acyclique dans une liste `lt`.

Validez votre procédure avec la solution.

Solution C++ @[topologicalSort.cpp]

```cpp
#ifndef TOPOLOGICAL_SORT
#define TOPOLOGICAL_SORT
#include <list>
using namespace std;
#include "Graphe.hpp"

// find a topological sort of an acyclic graph
template<typename T>
void topologicalSort(Graphe<T>& g, list<T>& tlist)
{
  // clear the list that will contain the sort
  tlist.erase(tlist.begin(), tlist.end());
  VertexInfo<T> vinfo;
```

```
  for (int ix = 0; ix < g.getvInfoSize(); ++ix)
  {
    vinfo = g.getvInfo(ix);
    if (vinfo.occupied)
    {
      vinfo.color = VertexInfo<T>::WHITE;
      g.setvInfo(ix, vinfo);
    }
  }
  // cycle through the vertices, calling dfsVisit() for each
  // WHITE vertex. check for a cycle
  try
  {
    for (int ix = 0; ix < g.getvInfoSize(); ++ix)
    {
      vinfo = g.getvInfo(ix);
      if (vinfo.occupied && vinfo.color == VertexInfo<T>::WHITE)
      {
        g.dfsVisit((*(vinfo.vtxMapLoc)).first, tlist, true);
      }
    }
  }
  catch(GraphError&)
  {
    throw GraphError("Graph topologicalSort(): graph has a cycle");
  }
}
#endif
```

Écrivez un programme de test.

Validez votre programme avec la solution.

Solution C++ @[pgtoposort.cpp]

```
// Test de topologicalSort
#include <iostream>
#include <string>
#include <list>
using namespace std;
#include "Graphe.hpp"
#include "topologicalSort.cpp"
#include "UtilsCtneur.cpp"
using UtilsCtneur::afficherContainer;

int main()
{
  string fname;
  cout << "Nom du fichier? ";
  cin >> fname;
  ifstream is(fname.c_str());
  // graph specifying the courses and prerequisite edges
  Graphe<string> g;
  is >> g;
  // a list holding the topological order of courses
```

```
  list<string> tlist;
  // execute a topological sort; store results in list
  topologicalSort(g, tlist);
  // output the list of possible courses
  cout << "==> Possible schedule of courses" << endl << "  ";
  afficherContainer(tlist.begin(), tlist.end());
  cout << endl;
}
```

2 Références générales

Comprend [] ■