|  |
| --- |
|  |
| Projet d’Algorithmes et Structures de Données |
| L’algorithme de Bellman-Ford |
|  |
| **Ben Slimane Amir & Delerce-Mauris Marina** |
| **18/05/2014** |

|  |
| --- |
|  |

Introduction

L’algorithme de Bellman-Ford fût inventé par Richard Bellman, Samuel End et Lester Ford à la fin des années 1950. Richard Bellman [1920-1984] était un mathématicien américain de grande renommée, il s’est vu décerner plusieurs récompenses telles le prix de théorie John Von Neumann en 1976 ainsi que l’IEEE Medal of Honor (la plus haute récompense de l’Institute of Electrical and Electronics Engineers qui récompense une carrière exceptionnelle dans les domaines de l’IEEE) en 1979. Samuel End [1886-1967] était quant à lui également un mathématicien très connu, puisque sa contribution en mathématique fût récompensée par la Mathematical Association of America en créant un prix portant son nom.

Dans ce rapport nous allons tout d’abord présenter l’algorithme et démontrer toutes les informations qu’il nous permet de recueillir sur un graphe. Puis nous verrons quelques exemples d’utilisations sur des graphes dotés de propriétés différentes. Et enfin, nous détaillerons les calculs de la complexité de cet algorithme.

Présentation de l’algorithme

Cet algorithme est utilisé en informatique afin de déterminer le chemin emprunté par les messages à travers le protocole d’information de routage (RIP).

Il permet de construire une arborescence de plus courts chemins d’un graphe orienté pondéré enraciné en r et couvrant tous les sommets accessibles à partir de r si G ne contient pas de circuit de poids négatifs. En effet, contrairement à l’algorithme de Dijkstra le graphe auquel on applique l’algorithme de Bellman-Ford peut contenir des arcs de poids négatifs et il permet alors de détecter si le graphe possède un circuit de poids négatif également appelé circuit absorbant. Effectivement, si à la fin de l’exécution de l’algorithme il existe un arc (u,v) tel que :

Avec le poids du sommet v, le poids du sommet u et le poids de l’arc (u,v)

# Pseudo-code :

boolean BellmanFord(G,s){

1

2

Initialisation

}

3

Recherche des plus courts chemins à partir de s vers tous les autres sommets

3.1

}

}

}

4

Recherche de cycles absorbants

}

}

}

}

Avec :

* s : le sommet source
* G=(S, A) : le graphe que l’on souhaite parcourir, S : les sommets, A : les arêtes

**Initialisation :**

On initialise tout d’abord le poids du sommet source à 0. En effet, comme il s’agit du sommet duquel nous partons cela ne coûte rien pour y arriver. On initialise également les poids de tous les autres sommets à .

**Recherche du plus court chemin :**

Pour chaque sommet du graphe sauf le sommet source, on regarde toutes ces arêtes. On compare alors le poids du sommet en question à la somme du poids du sommet père et du poids de l’arête. Si cette somme est plus petite que le poids du sommet que l’on regarde alors on remplace le poids du sommet par le résultat de la somme.

Une fois tous les sommets parcourus de cette manière il nous faut vérifier que le graphe ne contient pas de cycle absorbant.

Un **cycle absorbant** est un cycle dans lequel se trouve une ou plusieurs arête(s) à poids négatif(s). Si l’on cherche un chemin minimum dans ce cycle on peut boucler indéfiniment. Or l’algorithme fait que si l’on parcourt au plus |S|-1 fois une même arête on considère qu’elle est de poids minimal.

**Détection de cycles absorbants :**

Pour chaque arête du graphe on regarde si l’on peut trouver un poids encore inférieur au résultat obtenu avant. Si l’on en trouve alors le graphe en question contient un cycle absorbant. Dans ce cas, on retourne vrai, sinon on retourne faux.

Programmation de l’algorithme

L’algorithme ci-dessus a été implémenté grâce à une classe Graphe contenant une liste de sommets et une liste d’arcs.

|  |
| --- |
| public boolean BellmanFord() {  /\*\*  \* Initialisation  \*/  graphe.getSommets().get(depart).setCout(0);  for (int i = 0; i < graphe.getSommets().size(); i++) {  if (i != depart) {  graphe.getSommets().get(i).setCout(2147483647);  }  }  /\*\*  \* Relaxation  \*/  int sommetDepart, sommetArrivee;  double coutTemp, coutArrivee;  for (int i = 0; i < graphe.getSommets().size() - 1; i++) {  for (int j = 0; j < graphe.getArcs().size(); j++) {  sommetDepart = graphe.getArcs().get(j).getDepart();  sommetArrivee = graphe.getArcs().get(j).getArrivee();  coutArrivee = graphe.getSommets().get(sommetArrivee).getCout();  coutTemp = graphe.getSommets().get(sommetDepart).getCout()  + graphe.getArcs().get(j).getPoids();  if (coutArrivee > coutTemp) {  graphe.getSommets().get(sommetArrivee).setCout(coutTemp);  }  }  }   /\*\*  \* Verification  \*/  for (int j = 0; j < graphe.getArcs().size(); j++) {  sommetDepart = graphe.getArcs().get(j).getDepart();  sommetArrivee = graphe.getArcs().get(j).getArrivee();  coutArrivee = graphe.getSommets().get(sommetArrivee).getCout();  coutTemp = graphe.getSommets().get(sommetDepart).getCout()  + graphe.getArcs().get(j).getPoids();  if (coutArrivee > coutTemp) {  return true;  }  }  return false; } |

Détermination de la complexité

Pour un graphe contenant n sommets et m arcs la complexité est la suivante.

Initialisation : 1 + 2 (d’après le pseudo-code)

Recherche du plus court chemin à partir de s vers tous les autres sommets : 3 (3.1)

Recherche d’un cycle absorbant : 4

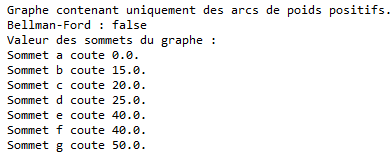
Complexité totale :

La complexité calculée ci-dessus est la complexité au pire des cas. L’algorithme parcourt dans tous les cas l’ensemble des arcs A du graphe G |S| - 1fois même si à partir d’une certaine itération, les sommets ne subissent plus de modifications. La complexité au meilleur des cas est donc la même. Pour un graphe complet, c’est-à-dire que chaque sommet est relié à tous les autres sommets, chaque sommet a donc n-1 arcs. Sachant qu’il y a n sommets, il y a donc arcs dans le graphe. Comme la complexité de l’algorithme est , avec ici alors la complexité est en .

Exemples de fonctionnement

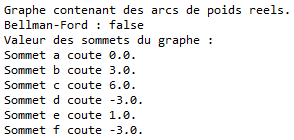
Dans cette partie nous allons appliquer l’algorithme de Bellman-Ford à plusieurs graphes différents.

# Graphe contenant des valeurs positives

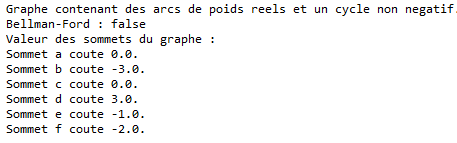


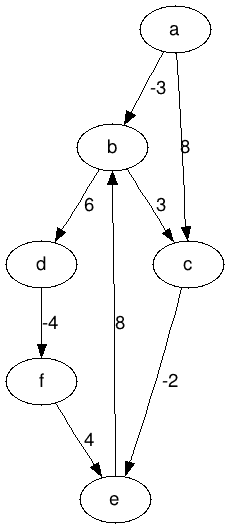
Comme nous pouvons le voir dans l’exemple ci-dessus, l’algorithme calcule les plus courts chemins partant du sommet a. Dans ce cas, le résultat de l’application de l’algorithme de Bellman-Ford est équivalent à celui de Dijkstra, cependant dans ce cas-là, l’algorithme de Dijkstra est plus performant.

# Graphe contenant des valeurs réelles

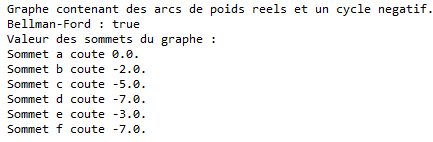
Dans cet exemple, l’algorithme de Dijkstra ne donne pas de résultat car ce graphe contient des valeurs négatives. En revanche, l’algorithme de Bellman-Ford fonctionne parfaitement. Ce graphe ne contient pas de cycle.

# Graphe contenant un cycle et des valeurs réelles



Dans ce graphe on observe bien qu’il y a deux cycles qui contiennent des valeurs négatives. Cependant ce ne sont pas des cycles absorbants. En effet, dans les cycles, la somme des valeurs négatives est inférieure à la somme des valeurs positives. On ne trouve donc pas de chemin de plus en plus court au fur et à mesure des itérations.

# Graphe contenant un cycle absorbant



Dans ce graphe nous pouvons voir que le graphe contient un cycle absorbant et qu’il a été détecté par l’algorithme. En effet, ici,, on peut donc toujours trouver un chemin puisque le poids des sommets diminue à chaque itération.