Branch and Bound Method

Paul-Alexis Dray and Jean-François Thai

December 21, 2017

Abstract

Dans ce rapport, nous nous proposons de mettre en oeuvre la méthode de Branch and Bound (**BB**) pour résoudre le Traveling Salesman Problem (**TSP**).

La bonne volonté racourcit le chemin

Proverbe brésilien

1 Introduction

Nous avons réalisé notre implémentation en **C++**, en utilisant la librairie **Lemon** qui permet de gérer les graphes de manière simple et efficace, et contenant divers algorithme pour résoudre le problème de l'arbre couvrant minmum dans un graphe donné.
Dans la suite, nous introduirons les principales classes de la librairie Lemon que nous avons utilisées, avant de présenter en détail l'algorithme que nous avons implémenté. Pour finir, nous discuterons des résultats de nos expérimentations.

2 La librarie Lemon

Lemon stands for: ***L****ibrary for* ***E****fficient* ***M****odeling and* ***O****ptimization in* ***N****etworks.*
Le coeur de la librairie Lemon réside dans les multiples classes existantes pour représenter efficacement des graphes : *ListGraph* (graphes non-orientés) et *ListDigraph* (graphes orientés) principalement.
Dans notre implémentation, nous utilisons :

- la classe *SmartDigraph*, qui est une variante de *Digraph* pour la représentation des graphes orientés plus efficaces (tant en espace qu'en temps) mais avec moins de fonctionnalités (la suppression de noeuds ou d'arcs dans le graphe n'est pas possible);
- les maps *SmartDigraph::NodeMap<value>* et *SmartDigraph::ArcMap<value>* pour représenter le poids des arcs (en indiquant *value* comme un *int*) et les différents chemins parcourus (en indiquant *value* comme un *bool* : 1 si l'arc fait partie du chemin, 0 sinon);
- l'algortime Kruskal implémenté dans Lemon pour résoudre le problème de l'arbre couvrant minimal sur un graphe donné;
- le format de fichier LGF pour importer un graphe depuis un fichier, qui permet de représenter les noeuds, les arcs entre chaque noeuds et les poids de chaque arcs.

3 Notre algorithme Branch and Bound

Pour illustrer l'explication de notre algorithme, nous nous placerons dans le graphe de la figure 1[1]

[1]Notons que le graphe 1 n'est pas orienté. Cependant, dans notre implémentation utilisons ce graphe comme s'il l'était : chaque arc est vu comme deux arcs orientés dans des sens contraires

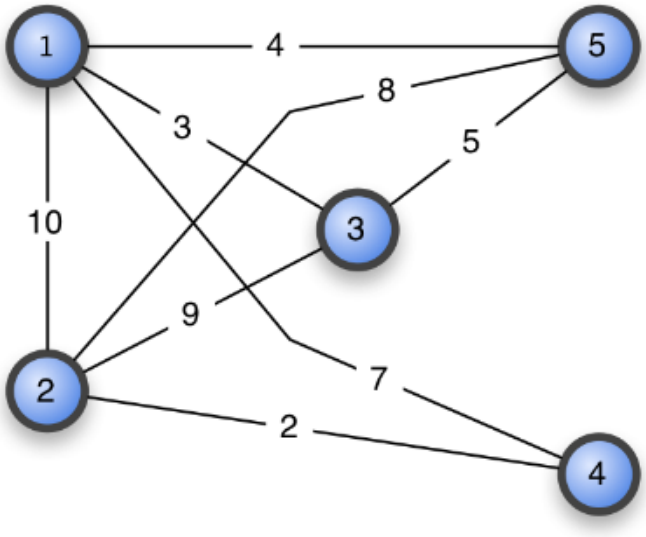

Figure 1: Graphe test G

3.1 L'algorithme

Notre algorithme Branch and Bound est présenté ci-dessous (Algorithme 1). Grossièrement, il s'agit de :

1. choisir un noeud de départ **n** et une arête **arc** sortante de **n**
2. calculer l'arbre couvrant minimal **ACM** sur le graphe privé de **arc**
3. calculer la LB = poids(**ACM** + **arc**)
4. si (**ACM** + **arc**) est un circuit hamiltonien, mettre à jour UB, sinon faire un appel récursif.

La vérification du circuit hamiltonien se fait avec l'algorithme 2.

Algorithm 1 Branch and Bound

Input: *graph* un graphe, *node* un noeud de *g*
Output: *path* un chemin

```
for all arc, arête sortante de node do
    path ← path + arc
    subgraph ← graph − arc                    //on enlève arc à graph
    minSpanTree ← kruskal(subgraph)
    LB = poids(minSpanTree + path)
    if (minSpanTree + path) est un circuit hamiltonien then
        UB = LB
        path ← path + minSpanTree
        return path
    else
        if LB < UB then
            BranchandBound(graph, target(arc))
        end if
    end if
end for
```

Algorithm 2 isHamiltonian

Input: *graph* un graphe, *node* un noeud de *g*
Output: *path* un chemin

```
for all noeud de graph do
  if OutArc(noeud) > 1 or InArc(noeud) > 1 then
    return false                //chaque noeud a au plus un arc entrant et un arc sortant
  end if
end for
currentNode ← noeud, arc ← OutArc(noeud)
visited[currentNode] = 1
while visited[target(arc)] = 0 do
  currentnode ← target(arc), arc ← OutArc(currentNode)
  visited[currentNode] = 1
end while
for all noeud de graph do
  if visited[noeud] != 1 then
    return false                //chaque noeud doit avoir été visité
  end if
end for
return true
```

3.2 L'algorithme pas à pas

Dans cette partie, nous nous proposons de suivre étape par étape notre algorithme sur le graphe 1, avec pour noeud de départ le noeud 3:

1. pour chaque arête sortante (*ie.* {31, 32, 35}), on calcule l'arbre couvrant minimal sur le graphe G privé de l'arête (*ie.* G-{31}, G-{32}, G-{35})
2. le chemin parcouru (*ie.* {31} + arbre couvrant minimal) n'est pas un circuit hamiltonien, on ne met donc pas à jour la borne supérieur on réitère en se plaçant sur le noeud cible de l'arête (*ie.* noeud 5)
3. pour chaque arête sortante (*ie.* {51, 52}), on calcule l'arbre couvrant minimal sur le graphe G privé de l'arête (*ie.* G-{35, 51}, G-{35, 52})
4. le chemin parcouru (*ie.* {35, 52} + arbre couvrant minimal est un circuit hamiltonien, on met à jour la borne supérieur)

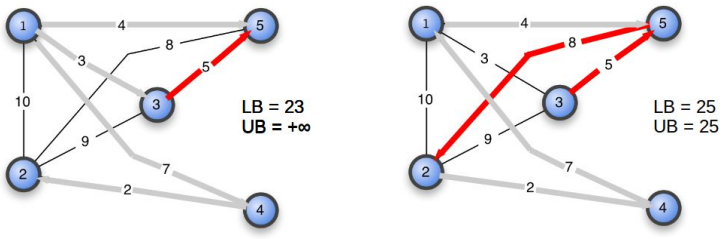

Figure 2: Etapes de l'algorithme

3.3 Idée d'algorithme

Un autre algorithme auquel nous avions pensé aurait été de :

1. calculer l'arbre couvrant minimal du graphe ;
2. construire le graphe eulérien à partir de l'arbre couvrant minimal ;
3. parcourir le graphe eulérien tant qu'il ne nous ramène pas en un noeud déjà visité, auquel cas, construire l'arbre couvrant minimal sur le sous-graphe restant (nous donnant ainsi une LB) et ainsi de suite.

Cependant, nous n'avons pas eu le temps d'implémenter cet algorithme et sa validité reste à démontrer.

4 Commentaires sur nos résultats

[2] Nous avons testé notre algorithme sur différentes instances données :

- graphe démo : UB = 25, temps d'éxécution = 0,001 sec
- graphe 17 noeuds : UB = 39, temps d'éxécution = 2654,69 sec

Nous avons également testé notre algorithme sur une instance comportant 33 noeuds : la machine a tourné pendant 25 heures avant d'être arrêtée (probablement dû à un acte malveillant).
Aussi, le temps d'exécution étant trop important, nous avons décidé pour les instances plus grosses de mettre une limite de temps de calcul fixée à deux heures. Les résultats suivants sont donc des solutions réalisables mais non optimales :

- graphe 33 noeuds : UB = 3011, temps d'éxécution = 7200,00 sec
- graphe 35 noeuds : UB = 3403, temps d'éxécution = 7200,00 sec

[2] Les tests ont été effectués sur le pc4040 de l'école

5 Annexes

5.1 Résultats détaillés

5.1.1 Graphe démo

Chemin: $3 \to 5 \to 2 \to 4 \to 1 \to 3$
Poids: 25

id	(u,v)	bool	weight	id	(u,v)	bool	weight
0	(1,1)	0	9999	13	(3,4)	0	9999
1	(1,2)	0	10	14	(3,5)	0	5
2	(1,3)	0	3	15	(4,1)	0	7
3	(1,4)	1	7	16	(4,2)	1	2
4	(1,5)	0	4	17	(4,3)	0	9999
5	(2,1)	0	10	18	(4,4)	0	9999
6	(2,2)	0	9999	19	(4,5)	0	9999
7	(2,3)	0	9	20	(5,1)	0	4
8	(2,4)	0	2	21	(5,2)	0	8
9	(2,5)	1	8	22	(5,3)	1	5
10	(3,1)	1	3	23	(5,4)	0	9999
11	(3,2)	0	9	24	(5,5)	0	9999
12	(3,3)	0	9999				

5.1.2 Graphe 17 noeuds

Poids: 39
(Voir le fichier results/result-atsp-17-1.txt)

5.1.3 Graphe 33 noeuds

Poids: 3011
(Voir le fichier results/result-atsp-17-1.txt)

5.1.4 Graphe 35 noeuds

Poids: 3403
(Voir le fichier results/result-atsp-17-1.txt)