# Optimisation combinatoire et convexe : projet voyageur de commerce

Nathanaël Courant et David Saulpic

## 1 Commentaires généraux

Nous avons codé deux heuristiques :

**nearest_neig.py** Une gloutonne qui consiste à partir d'un noeud aléatoire, choisir le plus proche voisin de ce noeud, et itérer ce procédé (en considérant seulement les voisins non encore visité).

**heuristic_kruskal.py** Une autre qui part d'un arbre couvrant minimal et qui construit le cycle associé au parcours en profondeur de cet arbre.

La première des deux heuristiques est la meilleure dans la grande majorité des exemples que nous avons testé. C'est donc elle que nous utilisons pour initialiser la recherche dans le programme linéaire dual.

En dehors de cela, nous avons également codé :

**parser.py** S'occupe de charger les données de TSPLIB et de produire les matrices d'adjacence.

**exact.py** Résolution exacte du voyageur de commerce en $O(n^2 2^n)$, fonctionne pour les instances de taille inférieure ou égale à 24 sans problème.

**min_cut.py** Implémente l'algorithme de Stoer-Wagner pour calculer une coupe minimum.

**separation.py** Implémente la méthode du primal pour trouver une borne inférieure.

**separation_dual.py** Implémente la méthode du dual.

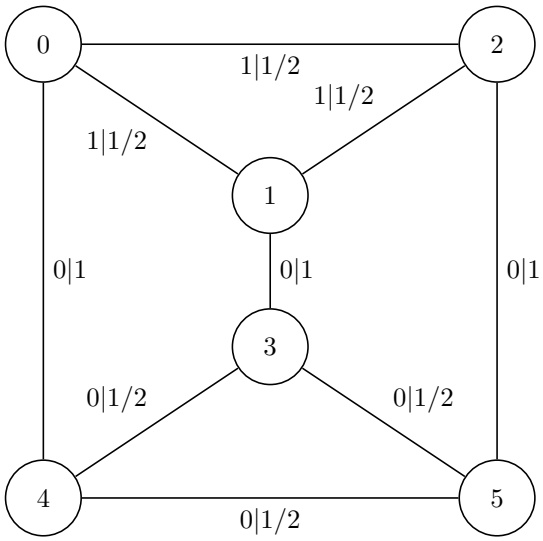
## 2 Résultats sur TSPLIB

| test | nearest_neig | heuristic_kruskal | Primal | Dual | Valeur exacte |
|---|---|---|---|---|---|
| burma14.tsp | 4048 | 4271, 0s | 3323.0, 0s | 3323.0, 0s | 3323 |
| gr17.tsp | 2187 | 2523 | 2085.0, 0.6s | 2085.0, 0.6s | 2085 |
| gr21.tsp | 3333 | 3841 | 2707.0, 0.3s | 2707.0, 0.2s | 2707 |
| eil51.tsp | 511 | 542 | 422.5, 28.9s | 422.5, 3.6s | 426 |
| gr24.tsp | 1553 | 1660 | 1272.0, 0.8s | 1272.0, 0.7s | 1271 |
| gr48.tsp | 6098 | 7297 | 4959.0, 43s | 4959.0, 8.5s | 5046 |
| dantzig42.tsp | 956 | 960 | 697.0, 16s | 697.0, 6.6s | 699 |
| brazil58.tsp | 30774 | 29438 | 25354.5, 84s | 25354.5, 21s | 25395 |
| berlin52.tsp | 8980 | 10096 | 7542.0, 15s | 7542.0, 7s | 7542 |
| bayg29.tsp | 2005 | 2117 | 1608.0, 4.4s | 1608.0, 1.8s | 1610 |
| bays29.tsp | 2258 | 2514 | 2013.5, 2.7 | 2013.5, 0.9s | 2020 |
| ulysses22.tsp | 10586 | 8399 | 7013.0, 1.6s | 7013.0, 1.7s | 7013 |
| st70.tsp | 830 | 866 | - | 670.0, 31.8s | 675 |
| pr76.tsp | 153462 | 140738 | - | 105120.0, 74s | 108159 |
| rd100.tsp | 9938 | 10660 | - | 7873, 186s | 7910 |
| pr107.tsp | 46680 | 58544 | - | 44176, 345s | 44303 |
| bier127.tsp | 135737 | 154509 | - | 117164.5, 692s | 118282 |
| ch150.tsp | 8191 | 8869 | - | 6476.5, * | 6528 |
| brg180.tsp | 12360 | 80970 | - | 961,42, 24349s | 1950 |
| d198.tsp | 18240 | 19629 | - | 15490, * | 15780 |

* : pour ces deux tests, le solver LP que nous utilisons (scipy.otimize) a eu un problème. La valeur que nous donnons est donc la dernière borne inférieure prouvée que nous avons.

# 3 Réponses aux questions

Même avec les contraintes supplémentaires sur les coupes, le programme linéaire n'est pas entier : en effet, considérons le graphe suivant :




Les annotations $a|b$ sur les arêtes siginifient que le coût de l'arête est de $a$, et que la variable $x_e$ correspondante a pour valeur $b$ dans la solution optimale du programme linéaire. On s'aperçoit que la valeur optimale de ce programme linéaire est donc de $3/2$ : il ne peut donc pas être entier.

On peut également remarquer que pour toute coupe $S$ de $V$, et pour toute solution entière $x$ du problème initial, $\sum_{e\in\delta(S)} x_e$ est paire. En particulier, si $|\delta(S)|$ est impair, $\sum_{e\in\delta(S)} x_e \leq |\delta(S)| - 1$. On vient donc d'obtenir une classe d'inégalités linéaires satisfaites par toute solution entière du problème, mais pas par les solutions fractionnaires comme on a pu le constater avec l'exemple ci-dessus.