TRAVAIL PRATIQUE 2
INF4705 – Automne 2016

POLYTECHNIQUE MONTRÉAL

LE GÉNIE EN PREMIÈRE CLASSE

Le 08 décembre 2016

Adrien Doumergue (1868995) & Robin Royer (1860715)

Introduction

Ce TP nous propose un problème combinatoire et nous invite à faire preuve de créativité et d'imagination pour le résoudre. Le but est ici de pouvoir, en un interval de temps fix, donner la meilleur solution possible et en ne connaissant pas a l'avance l'exemplaire considéré. Un des objectifs de ce TP est bien évidemment de se baser
Il s'agit d'un problème NP-difficile qui nécessite d'utiliser des algorithmes heuristiques pour atteindre le plus rapidement possible une solution proche de la solution optimale.

Choix d'implémentation

Le sujet précise que le fichier d'entrée notifie les compagnies **ne pouvant pas être** assis à la même tables. Nous en déduisons donc dans notre implémentation que si nous avons trop peu de table pour accueillir toutes les compagnies en prenant en compte les contraintes précédente, l'ensemble des solutions pour cet exemplaire est nul que nous envoyons un message d'erreur. (Bien que des exemplaires fournis présentent ce cas.)

Discussion de l'algorithme

Pour cette compétition, nous avons choisi d'implémenter notre algorithme en c++ afin de profiter de la compilation pour gagner en efficacité sur le temps d'exécution.
Afin de faciliter la lecture du code et l'implémentation de notre algorithme, nous avons décidé d'utiliser la programmation orientée objet en définissant des classes Table, Company et Solution. Afin de lire les données, un Parser fait aussi partie de notre projet.

Notre code va d'abord générer une solution qui appartient à l'ensemble des solutions du problème (pas de conflits avec les compagnies qui ne doivent pas être assises à la même table). Si on ne trouve pas de solutions, un message d'erreur est renvoyé.

A partir de cette solution, on va pouvoir générer d'autres solutions dans le voisinage suivant : on prend une compagnie aléatoire et on la change de table tout en vérifiant qu'il n'y ait pas de conflits. Ceci nous assure de rester dans l'ensemble des solutions dans un voisinage de complexité O(n*p) où n est le nombre de tables et p le nombre de compagnies. Nous avons aussi ajouté un moyen de placer cette compagnie à la table la plus vide possible, mais nous nous sommes rendus compte que l'écart-type influence peu le résultat final par rapport aux décompte d'amis/ennemis.

Afin d'éviter de rester coincé à un optimum local, nous avons implémenté une heuristique de recuit simulé. Une fois que que le recuit simulé n'arrive plus à obtenir de modifications, on appelle un algorithme génétique qui a pour population l'ensemble des solutions qui ont été la meilleure solution dans le recuit simulé. On agrandit cette population a une taille de 1000 et on applique les principes d'un algorithme génétique : on crée une nouvelle population à partir de l'ancienne et on ne garde que les 1000 meilleurs solutions (sélection naturelle) pour la génération suivante.

Pseudo-code du recuit simulé

```
Pour 100 itérations sans nouveau maximum
	tant que iteration < 2000
		s est choisi avec une permutation aléatoire légale avec une probabilité de
	1/2
		s est choisi avec une permutation aléatoire maximisant l’équilibre des
	tables avec une probabilité de 1/2

		si f(si) > f(s) alors
			si+1 <- s
		sinon
			si+1 <- s avec une probabilite de pi
			si+1 <- si avec une probabilite de 1 - pi
		i <- i+1
```

Conclusion

Notre algorithme est original dans le sens où il combine deux heuristiques : le recuit simulé (heuristique qui dépend de la trajectoire) et un algorithme génétique (heuristique qui ne dépend pas de la trajectoire). En point d’amélioration, nous pourrions implémenter du multi-threading sur la phase finale tout en appliquant un nombre aléatoire de mutations à nos solutions (dans le cas présent nous n’effectuons qu’une mutation). Nous pourrions aussi augmenter la taille de la population.

Ce TP a pu nous confronter avec les problématiques d’optimisation ainsi qu'à la problématique du choix de langage. En effet choisir un langage compilé tel que le c++ amène un complexité importante qu'il ne faut pas négliger (écriture et maintenance du code) en échange d’un gain en performance.