Projet module RP Autour du google hashcode 2019

Partie 1 : Première approche du problème, analyse

NB : Certaines parties du projet sont réalisées en Python, d’autres en Javascript (nodeJs). Nous avons utilisé 2 langages de programmation, car Javascript, pour des raisons personnelles permettait de plus rapidement tester certaines idées, mais aussi parce que Python s’avérait être sensiblement plus lent sur certains algorithmes. Python reste cependant utile notamment pour utiliser Gurobi que nous avions déjà utilisé en MOGPL.

Pour donner un sens à nos résultats tout au long du projet, nous avons cherché sur le leaderboard de google les résultats du top pour avoir un ordre de grandeur : 1 222 000 points pour la première place (en faisant la somme du score de chaque dataset).

Pour chaque dataset, les meilleurs résultats tournaient en général autours de :

* B\_lovely\_landscapes : 235 000 points
* C\_memorable\_moments : 1800 points
* D\_pet\_pictures : 440 000 points
* E\_shiny\_selfies : 550 000 points

Ensuite nous avons codé la fonction pour créer une présentation linéaire classique qui respecte l’ordre d’apparition des photos, pour obtenir ainsi une borne inférieure des résultats :

* B\_lovely\_landscapes : 12 points
* C\_memorable\_moments : 152 points
* D\_pet\_pictures : 190 961 points
* E\_shiny\_selfies : 112 468 points

On a ainsi à ce stade une bonne idée du score qu’il faut atteindre au maximum et au minimum.

On remarque que le score du dataset C est particulièrement faible.

On réalise alors une analyse des features sur chaque dataset :

On remarque que les dataset ont chacun leur particularités, c’est-à-dire que la réparition des features est très différentes d’un dataset à l’autre, on pourrait sûrement adapter un algorithme particulier pour chaque dataset.

Dans le B, on a près d’un million de features, qui ne sont présentes que 2 fois maximum, dans le C on a 2000 features présentes 2, 10 ou 20 fois chacunes de même que dans le D on a 200 features présentes 3000 ou 17500 fois la plupart du temps. Enfin le dataset E est plus régulier avec 500 features présentes 3000 +/- 100 fois chacunes.

Les résultats de la présentation linéaire de B s’expliquent donc très bien après avoir regardé les features présentes dans son dataset.

Partie 2 : Résolution par diverses techniques

Ici on commence par tenter une méthode gloutonne qui donne souvent des résultats assez acceptables sur des problèmes de type TSP. La méthode gloutonne utilisée est cependant en O(n²), ce qui pose un problème puisque les dataset ont environ 100 000 images, soit sur une double boucle 10 milliards d’itérations à faire en ordre de grandeur (sans compter que le calcul de la qualité de la transition à chaque itération est non négligeable en temps de calcul).

Pour réduire ce temps, on peut utiliser la même méthode gloutonne avec une profondeur de recherche fixée, c’est-à-dire qu’une fois qu’une image est sélectionnée, on ne maximise la transition en ne choisissant que parmi les 50 prochaines images pout une profondeur de recherche de 50.

On tend ainsi vers un algorithme en O(n) pour une profondeur qui tend vers 1.

Résultats pour une profondeur de recherche de 500 :

* B : 7374 points en 314 secondes
* D : 409070 points en 376 secondes

A la fin du rapport on verra une autre méthode pour résoudre b plus rapidement.

Etant donné les particularités du dataset B, on peut tenter une résolution avec un autre algorithme où l’on va travailler sur les mots-clé plutôt que sur les images :

On commence par construire une table de hashage qui associe à chacune des 1 millions de features du dataset B une liste correspondant aux index des images possédant cette feature.

Puis on construit la présentation ainsi :

Pour chaque transition on sélectionne les images qui ont au moins une feature en commun avec la slide actuelle, puis on maximise la transition parmi ces images. Enfin on met à jour la table de hashage.

Résolution par descente de gradient

Résolution par algorithme génétique

Partie 3 : Résolution par programmation linéaire en nombre entiers

95628 glout améliorée

Utiliser une autre fonction de transition qui calcule plus vite ? (car celle-là très couteuse)

Notes de fin : il pourrait être intéressant de comparer les résultats en mélangeant les vignettes puisque google a dû les mélanger d’une certaines façon (surtout pour B)

résultat terrible, on remarque que ca peut être uine bonne idée hashtable

gloutonne depth 10, 100% : dataE : 227584 en 95 secondes

parler de la profondeur, qui change pas tant que ça la longueur

parler dune gloutonne par nombre de features, ou calcul transition moins complique, ou critere de nombre de features decroissants

voir idée téléphone : laisser les cycles qui sont acceptable : 1 cycle implique juste qu’on se retrouve avec une mauvaise transition quelque part mais n’a pas d’impact sur le reste

La descente de gradient est également difficile, puisque on va chercher a améliorer la presentation en trouvant une

nouvelle presentation meilleure que la courante, mais pour ça il faut tester toutes les permutations jusqu'à en trouver une meilleure,

au début ça va mais au bout d'un moment ça devient long puisqu'on fait du O(ncarre) sur 80000 données => 10^5\*10^5 => 10 milliard, pour une iteration..

Pour améliorer la vitesse : ne pas recommencer la boucle a chaque iteration mais la finir (pour éviter de se retrouver dans une configuration où

tout le début de la présentation ets bon mais pas la fin, ici on va améliorer l'ensemble de la présentation à chaque itération)

La descente de gradient paraît être une bonne technique cependant vu le type de probleme, voir cone des solutions avec pas trop de local optima ?

Sans étude, vu qu'une vignette n'a une influence que sur celle d'avant et d'après