

Grégory LANG

Openclassrooms

Parcours « Ingénieur Machine Learning »

11 février 2020

Projet 8 IML : Compétition KAGGLE :



Développement d'un algorithme génétique

Résumé :

Ce rapport présente mes travaux sur le projet 8 Openclassrooms du parcours Ingénieur Machine Learning.

Dans le but de participer à une compétition Kaggle, j'ai développé un algorithme génétique pour tenter de répondre au problème d'optimisation d'un agenda fictif des visites de l'usine du père Noel par 5000 familles, sur une durée de 100 jours avant Noël.

Je la compare ou l'agrémente en partie avec la méthode du type «stochastique product search».

Je montre aussi la méthode MIP en utilisant un optimiseur du commerce tel que Gurobi.

Le github du code du projet est disponible ici :

<https://github.com/jeugregg/santa-workshop-tour-2019>

Le kernel Kaggle est disponible ici :

<https://www.kaggle.com/jeugregg/santa-s-2019-original-genetic-algorithm-method>

SOMMAIRE

1. La problématique et son Interprétation	3
1.1. Contexte et interprétation	3
1.2. Pistes envisagées	4
2. Exploration du dataset	4
3. Modèle Algo Génétique	6
3.1. Principe	6
3.2. Implémentation	6
4. Modèle stochastic product search	14
4.1. Principe	14
4.2. Implémentation	14
4.3. Résultats	16
5. Modèle MIP Gurobi	18
5.1. Principe	18
5.2. Mise en équation	19
5.3. Implémentation	21
5.4. Résultat	22
6. Conclusions	23
7. Axes d'améliorations	23
8. Sources bibliographiques	24

1. La problématique et son Interprétation

1.1. Contexte et interprétation

Pour ce projet, il faut réaliser une compétition Kaggle.

La compétition choisie est « Santa Workshop Tour 2019 » qui propose de répondre à un problème d'optimisation d'un calendrier de visites fictives de l'usine du Père Noël.

La compétition est hébergée ici :

<https://www.kaggle.com/c/santa-workshop-tour-2019>

Le problème est le suivant : 5000 familles veulent visiter le Père Noël. Mais il n'y a que 100 jours disponibles avant Noël. Vu le nombre important de personnes, il a fallu leur proposer de choisir 10 dates par ordre de préférence. Mais, il est possible que le choix final proposé de certaines familles ne soit pas parmi leurs 10 dates.

Le Père Noël devra payer des pénalités aux familles si le choix n'est pas le premier : le choix 0. Voici ces pénalités appelées : « **preference cost** » :

```

choice_0: $0
choice_1: $50
choice_2: $50 + $9 par membre de la famille
choice_3: $100 + $9 par membre de la famille
choice_4: $200 + $9 par membre de la famille
choice_5: $200 + $18 par membre de la famille
choice_6: $300 + $18 par membre de la famille
choice_7: $300 + $36 par membre de la famille
choice_8: $400 + $36 par membre de la famille
choice_9: $500 + $235 par membre de la famille
autre    : $500 + $434 par membre de la famille

```

En plus, les coûts de fonctionnement dépendent de la différence entre le nombre de personnes qui visitent entre le jour J et le jour J+1.

Cette pénalité est appelée « **accounting penalty** »:

$$\text{accounting penalty} = \sum_{d=100}^1 \frac{(N_d - 125)}{400} N_d^{(\frac{1}{2} + \frac{|N_d - N_{d+1}|}{50})}$$

where N_d is the occupancy of the current day, and N_{d+1} is the occupancy of the previous day (since we're counting backwards from Christmas!). For the initial condition of $d = 100$, $N_{101} = N_{100}$.

Aussi, il y a 2 contraintes supplémentaires : $125 \leq N_d \leq 300$

Donc le problème revient à optimiser les choix assignés aux familles pour obtenir le score minimum :

$$\text{score} = \text{preference cost} + \text{accounting penalty}$$

C'est un problème de minimisation sous contraintes.

1.2. Pistes envisagées

Pour résoudre ce problème il existe différentes méthodes :

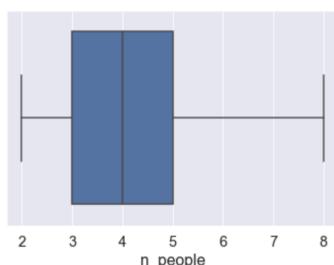
- du type stochastique : tel que algorithme génétique, ...
- du type Integer Programming / Mixed Integer Programming (MIP)

Je propose de développer une méthode d'algorithme génétique combinée avec une autre méthode : « stochastic product search ».

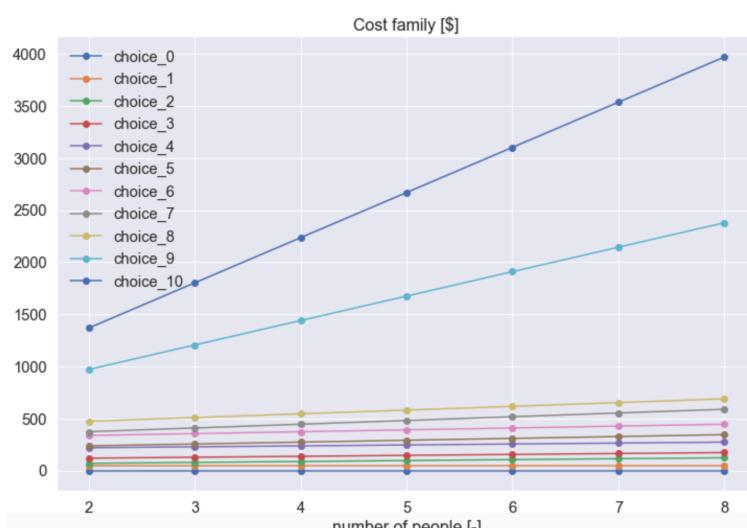
Le méthode MIP est aussi étudiée pour comparer ou en complément en utilisant un optimizer du commerce tel que Gurobi.

2. Exploration du dataset

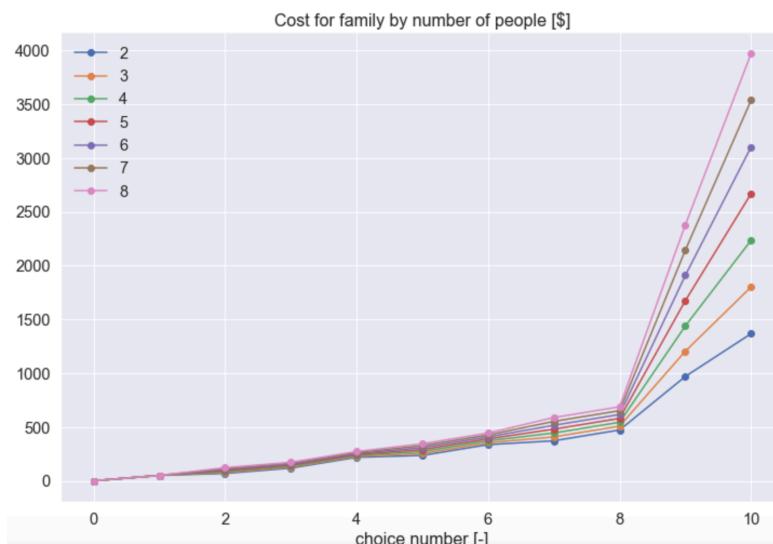
On constate que le nombre de personne par famille est entre 2 et 8 avec une médiane à 4 :



Ensuite on peut voir l'impact sur le « preference cost » :

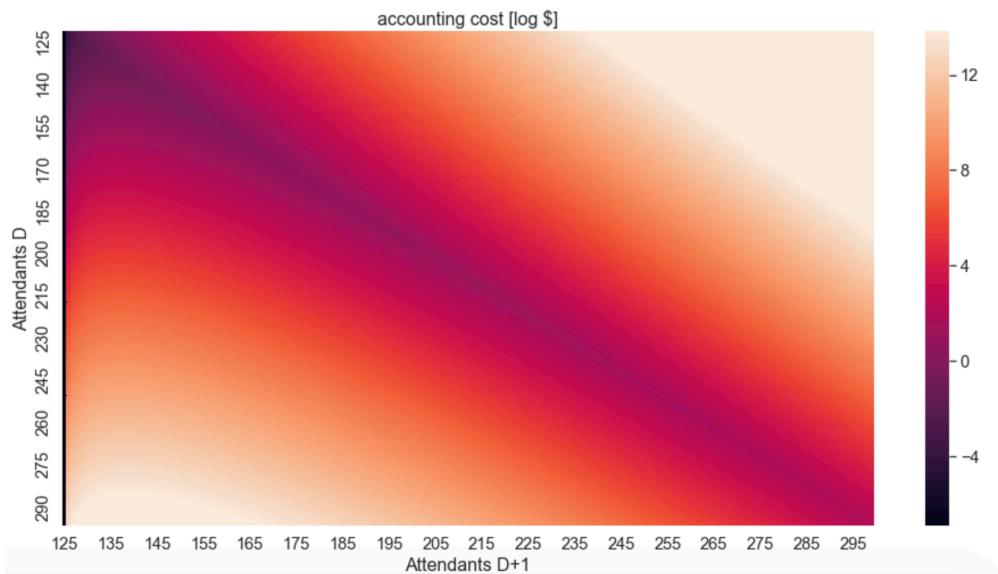


On constate que clairement les choix 9 et 10 seront à éviter surtout pour les grandes familles.

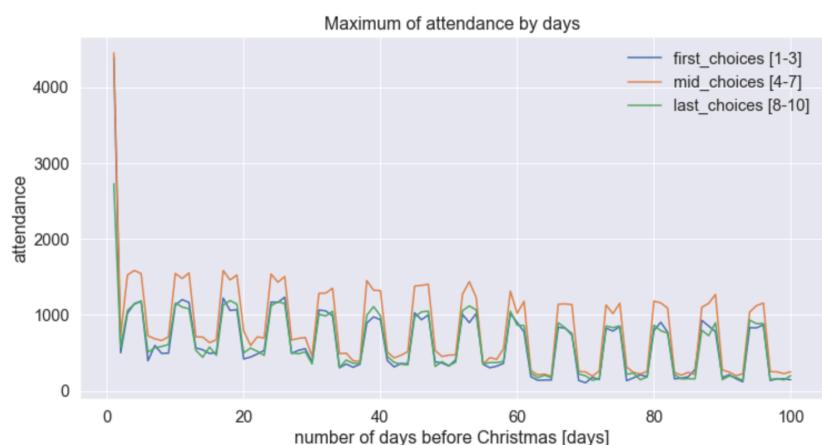


On constate que pour les choix entre 0 et 8, l'impact du nombre de personnes est un peu moins important que le numéro du choix.

En ce qui concerne le « accounting penalty », le delta d'affluence entre 2 jours peut avoir un impact exponentiel.



Si on s'intéresse aux jours choisis par les familles, on constate que le jour avant Noël est très demandé par rapport aux autres jours. Aussi on remarque que les week-end sont aussi le plus demandés.



3. Modèle Algo Génétique

3.1. Principe

Le principe de l'algorithme génétique se rapproche du principe de l'évolution des espèces vivantes sur terre. On part d'une population initiale constituée d'individus qui possèdent chacun des gènes.

Cette population est évaluée suivant un ou plusieurs critères.

Ensuite, une sélection est faite. Par exemple les individus les moins bons seront éliminés.

Puis, pour les meilleurs, des croisements entre les individus seront effectués.

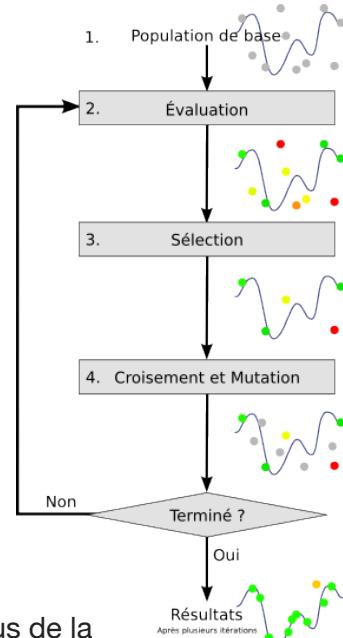
Enfin, des mutations de quelques gènes de certains individus de la population peuvent avoir lieu.

Dans notre cas,

- La population est un ensemble de plusieurs « agendas » des 5000 visites des familles.
- Un individu est donc un seul agenda de 5000 jours assignés aux familles.
- Les gènes sont les jours assignés par famille.

La mutation d'un gène correspond à la modification d'un jour dans l'agenda.

L'évaluation sera la pénalité à payer pour le Père Noël qui dépend uniquement de l'agenda.



3.2. Implémentation

Le notebook **santa-s-2019-starter-notebook.ipynb** contient toutes les routines de fonctionnement de l'algo génétique.

3.2.1. Création population initiale

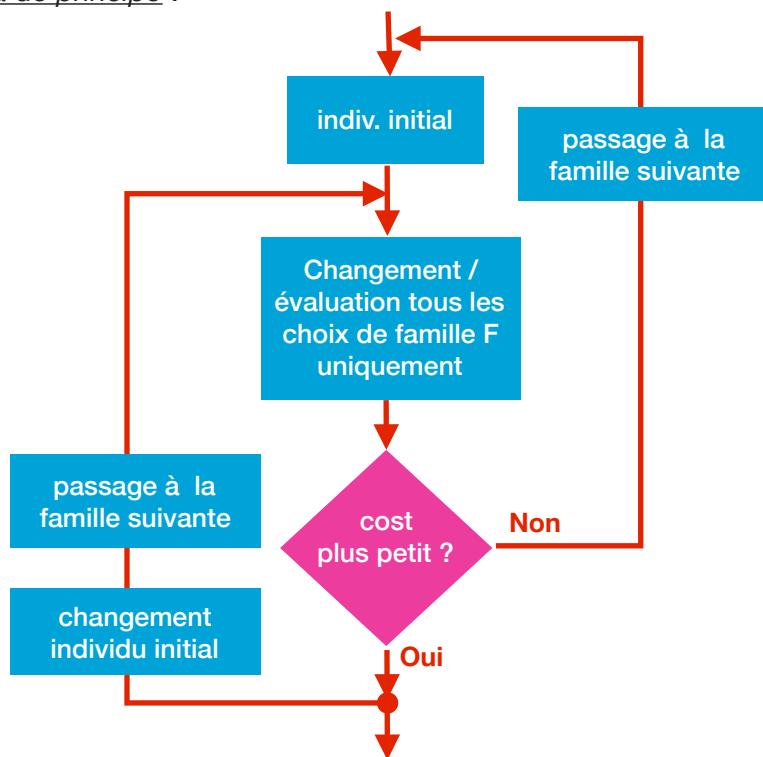
Il y a beaucoup de méthodes possibles.

Comme un agenda/individu « sample » est fourni, la plupart des méthodes partent de ce premier individu pour la constitution de la population.

La première méthode utilisée est d'optimiser avec une méthode simple depuis cet individu suivant plusieurs chemins.

La méthode simple appelée dans ce rapport « boost simple » est relativement simpliste.

Voici le schéma de principe :



Ce boost est séquentiel puisque qu'il prend toujours le meilleur individu pour en déduire le meilleur suivant. Le principe du calcul est issu du notebook Kaggle suivant :

<https://www.kaggle.com/inversion/santa-s-2019-starter-notebook>

Par contre, il y a plusieurs façons de réaliser le **passage à la famille suivante**. Cela va dépendre du chemin choisi.

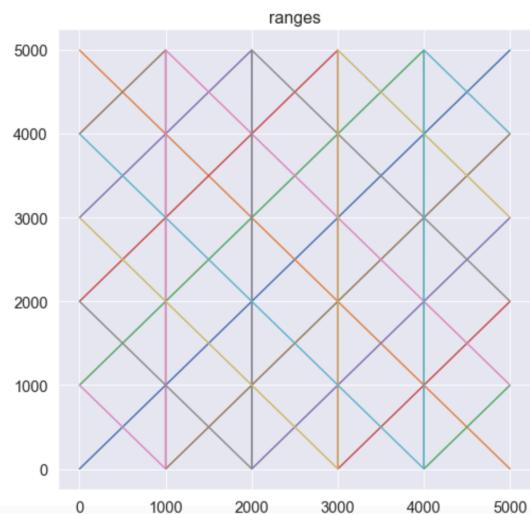
Voici les deux méthodes proposées.

- La méthode des **10 chemins différents** : en partant de l'individu « sample » fourni, on parcourt toutes la familles avec la méthode précédente, en décalant de 500 la première famille testée. ($500 = 5000 / 10$).

On trouve 10 meilleurs individus différents qui serviront de base pour créer la population totale.

Enfin, on ajoute le reste de la population en la générant par mutation de ces 10 meilleurs.

Pour une population de 1000 individus, on mute 99 fois chacun des 10 meilleurs.



- La méthode des **chemins aléatoires** : on parcourt *nb_pop* fois de façon aléatoire les familles pour créer *nb_pop* individus avec l'optimisation « boost simple ». Mais on part toujours du « sample » pour l'optimisation des individus. La méthode n'est cette fois-ci pas séquentielle.

3.2.2.Sélection

A chaque génération, on sélectionne les meilleurs. On calcule alors une probabilité de croisement suivant une loi. Elle permet de classer les individus par le critère de coût et de préparer la phase de croisement.

Les individus très mal classés ont de grandes chances de ne pas être croisés et ainsi de disparaître à la prochaine génération. Voici la loi utilisée :

C'est une fonction de probabilité croissante avec le rang inversé à la puissance POW_SELECT, normalisée.

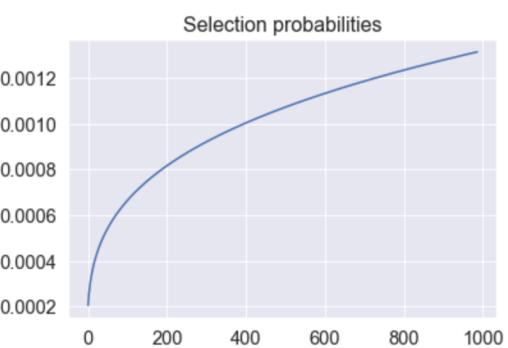
$$p_I = \frac{RANG_INVERSE_I^{pow_select}}{\sum RANG_INVERSE^{pow_select}}$$

exemple : le meilleur des 1000 candidats :

$$p_{1er} = \frac{1000^{0.3}}{\sum RANG_INVERSE^{0.3}} = 0.0013 \text{ avec } \text{POW_SELECT} = 0.3$$

Alors que le dernier à une probabilité de 0.0002.

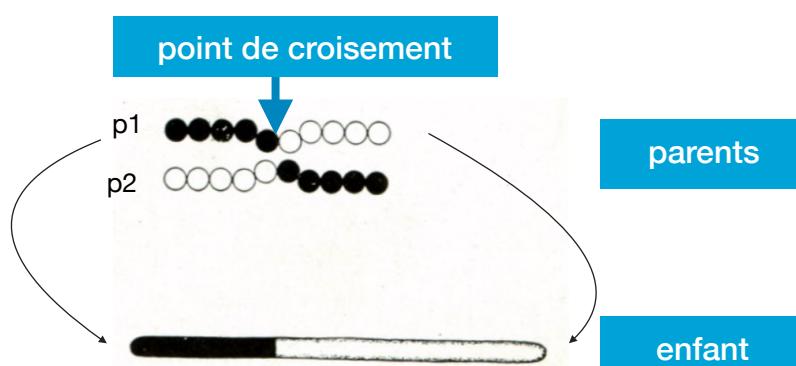
Enfin, à chaque génération, on conserve les NB_BEST_KEEP meilleurs. (entre 10 et 20 sur une population de 1000).



3.2.3.Croisement/enjambement

Le principe du croisement aussi appelé enjambement, est de mélanger une partie des « gènes » de deux individus pour en créer un nouveau.

On choisit aléatoirement un point de croisement :



On choisit de créer nb_pop enfants en sélectionnant 2 parents avec la loi de probabilité donnée précédemment. On régénère entièrement la population à chaque epoch.

3.2.4. Mutation

La mutation est faite dans en se limitant aux choix des familles plutôt que de choisir n'importe quel jour.

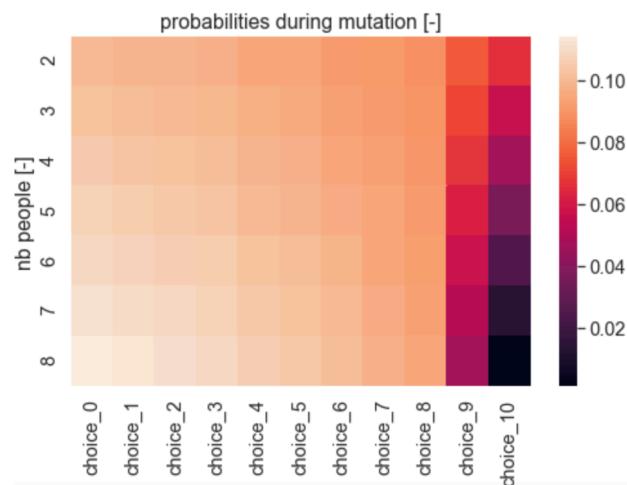
Pour muter le gène d'un individu, en d'autres termes, pour changer la date assignée à une famille, on ajoute un delta aléatoirement au numéro de son choix actuel dans un intervalle min/max. On répète cela pour un ratio R_POP_MUT des individus et sur un ratio R_MUT de leurs gènes (ou familles).

Voici les valeurs utilisées les plus pertinentes :

- pour $R_POP_MUT = 0.1$ (100 individus si $nb_pop = 1000$)
- et $R_MUT = 0.01$ (ce qui représente 50 familles).

Deux lois de probabilités sont utilisées lors de la mutation.

La première est celle qui permet de muter plutôt vers les choix de numéro faible et de façon plus prononcée si la famille est nombreuse dû au prix à payer plus important :



La seconde est dans le cas où aucun choix d'une famille n'est possible, on lui attribue une date aléatoire suivant une probabilité sur les jours en évitant le plus possible les week-ends et le jour avant Noël.

3.2.5. Ajout des meilleurs

A chaque génération, on réserve les meilleurs individus de la génération précédente en les conservant (20 sur 1000 individus).

3.2.6.Evaluation

L'évaluation est faite simplement en calculant la pénalité « preference cost » et la pénalité « accounting penalty » comme décrit auparavant pour toute la population, individu par individu.

Le calcul est donc répété de nombreuse fois. Tout comme les mutations et les croisements.

C'est pourquoi il est primordial d'utiliser des fonctions qui s'exécutent rapidement.

Mon choix s'est porté sur les routines optimisées avec « **numba** » .

La fonction d'évaluation du coût est disponible sur le Kernel suivant : <https://www.kaggle.com/xhlulu/santa-s-2019-faster-cost-function-24-s>

La fonction a été corrigée en partie en ce qui concerne les contraintes $125 \leq Nd \leq 300$.

Le même principe a été utilisé pour les autres routines utiles et en particulier celle pour la mutation et le croisement. Aussi, l'ajout de l'option de parallélisation de numba a réduit les temps de calcul en particulier pour le croisement :

```
@njit(parallel=True, fastmath=True)
def my_fun():
...

```

Avant optimisation, en utilisant des dataFrames de pandas : **6 s par epoch**

```
Timing cross:  3.284s
Timing mutation:  2.493s
Timing eval:  0.506s
```

Apres optimisation, passage en array, re-codage pour *numba* et l'activation du mode parallèle, une itération (epoch) dure **0.1s par epoch**

```
Timing cross:  0.0365 s
Timing mutation:  0.0409 s
Timing eval:  0.0233 s
```

3.2.7.Boosting

Le boosting est activable toutes les *boost_freq* epochs (=2000).

Le principe est le même que pour la création de la population en utilisant le boost simple séquentiel.

3.2.8. Etude hyper-paramètres

3.2.8.1. Première population

Voici la liste des hyper-paramètres lors de la création de la première population:

```
NB_FIRST_POP = 1000 # number of first population of choices
DELTA_CHOICE_FIRST_POP = 2 # +/- delta choice of mutated first population
R_FIRST_MUT = 0.05 # RATIO of mutation for first population
```

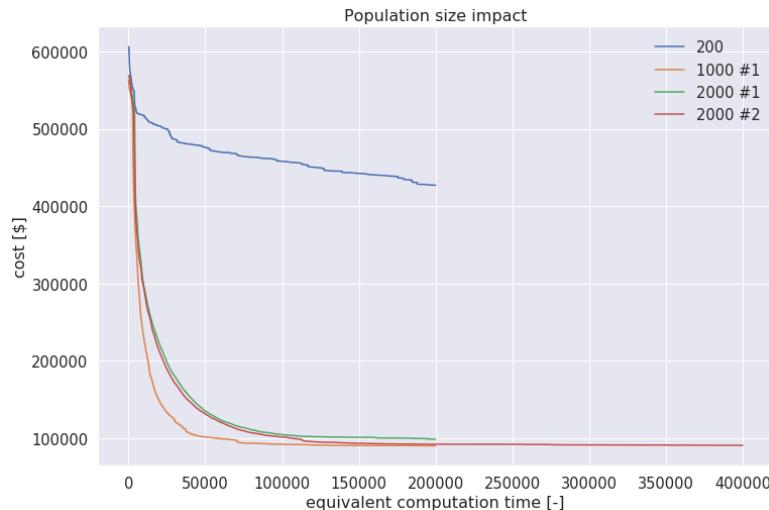
Les méthodes des *10 chemins* ou des *chemins aléatoires* donnent le même résultat en terme de vitesse de convergence.

Pour les deux méthodes ont utiliser un taux de mutation de 5% des gênes de chaque individus (5% des dates de chaque agenda), autour de +/- 2 numéros de choix des familles. Le choix est pris d'utiliser la méthode des *chemins aléatoires* car elle possède une déviation standard supérieure (1.5 au lieu de 0.5 pour l'autre méthode) pour apporter de la diversité au départ.

3.2.8.2. Taille de la population

La taille de la population est maintenue constante au cours du calcul.

Mais l'impact sur le temps de convergence est important comme on peut le constater sur la figure suivante :



Une population trop petite ne permet pas d'avoir assez de diversité pour trouver un meilleur individu à chaque génération. Une population trop grande n'apporte pas plus d'intérêt car le calcul est ralenti. On note toute de même que 1000 et 2000 semblent converger vers la même valeur.

La taille de la population plutôt optimale en vitesse de convergence est donc 1000.

3.2.8.3.Hyper-Paramètres des générations

Voici la liste des hyper-paramètres utilisés pour chaque itération, donc à chaque génération (partie **Loop over generations** du notebook **santa-s-2019-starter-notebook.ipynb**) :

```
NB_MAX_EPOCHS = 200000
R_POP_MUT = 0.1 # % population mutated at each gen. default : 0.10
R_MUT = 0.01 # % families mutated at each gen default : 0.01
DELTA_CHOICE = 2 # default : 2
NB_BEST_KEEP = 20 # how much best one to keep
POW_SELECTION = 0.3 # default 0.3
flag_boost = True
boost_freq = 2000
R_CROSSOVER = 1
CHOICE_RANGE_MAX = 10 # maximum choice number
```

Evidemment le nombre d'epochs est important. Le calcul peut être arrêter et relancer sans problème. Mais temps de calcul est de l'ordre de $0.12 \times \text{nb_epochs}$ en secondes sur MACBOOK PRO portable 2015. $100000 \text{ epochs} = 12000\text{s} = 3\text{h}20$

3.2.8.4.Taux de mutation

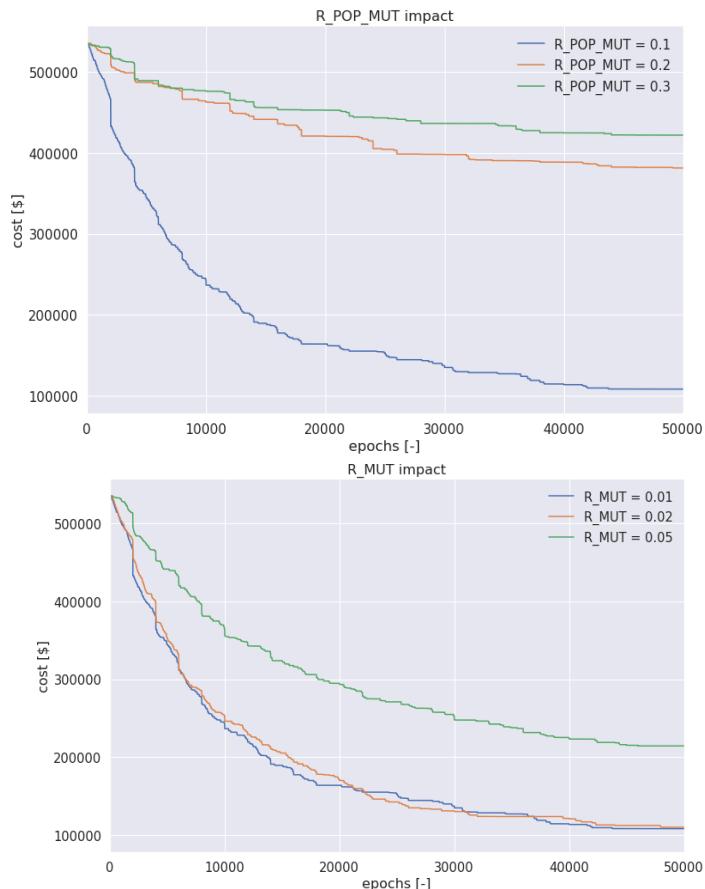
Le taux R_POP_MUT représentant le ratio d'individu à muter, impacte fortement la vitesse de convergence.

Un taux inférieur ralenti de même la convergence par le trop peu de diversité.

Le taux optimal **R_POP_MUT = 0.1** est la référence. (10% de $1000 = 100$ individus qui mutent à chaque epoch).

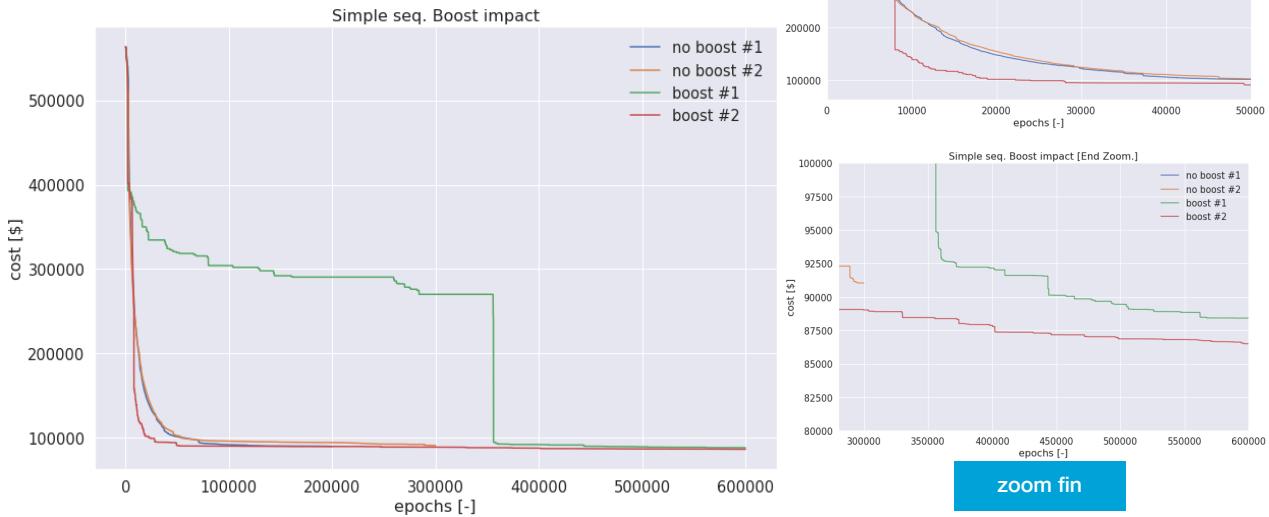
En ce qui concerne le taux de mutation R_MUT, l'impact est du même ordre. Un taux de 0.02 au lieu de 0.01, n'apporte pas de résultat vraiment différent. Un taux de 0.05 ralenti la convergence.

La référence est donc **R_MUT=0.01**



3.2.8.5. Activation du boost simple séquentiel

Le boost permet le plus souvent de faire converger plus vite. Mais comme l'évolution est aléatoire parfois cela peut avoir un effet de blocage (rarement observé) comme sur cette figure (courbe verte boost #1).

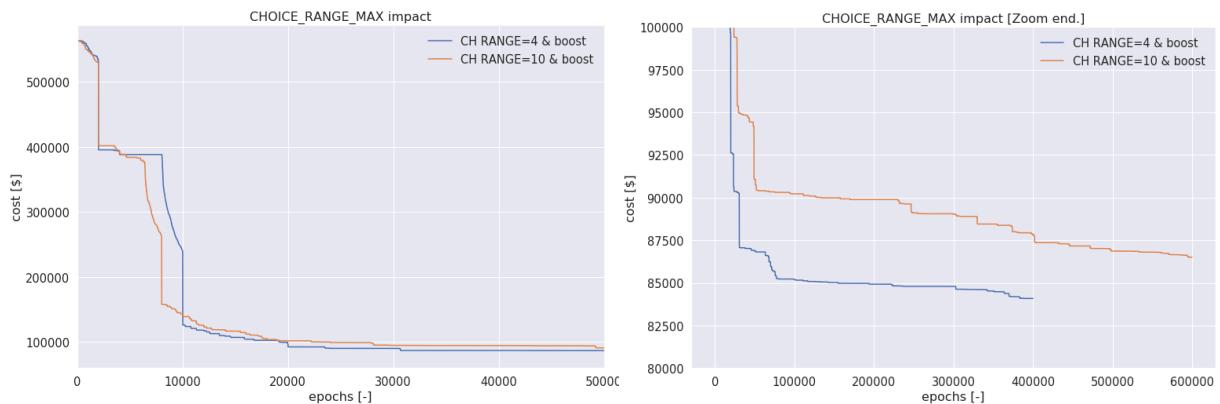


On observe que l'activation ou non ne change rien sur la valeur de convergence. De plus, elle ne semble pas donner de grand avantage en temps de calcul. Un boost simple entraîne un calcul de 10s tous les 2000 epochs (5% de temps de calcul en plus). Si on cherche l'optimum sur plus de 6000000 epochs, le boost simple ne sert à rien. Si on cherche un résultat rapidement, mieux vaut l'activer.

3.2.8.6. Meilleur résultat

Le meilleur résultat (cost=84089) est obtenu dans un temps relativement court de 15h. Mais en 7h, le cost= 84900. C'est aussi grâce à une limitation de la zone de recherche : **CHOICE_RANGE_MAX=4** (recherche limitée du choix #1 jusqu'au #5 uniquement).

Si on compare avec le range normal (1 à 11), on voit que l'on converge au début de la même manière puis on fini à un minimum plus bas. Avec le range de 10 on l'atteindrait sûrement mais beaucoup plus tard.



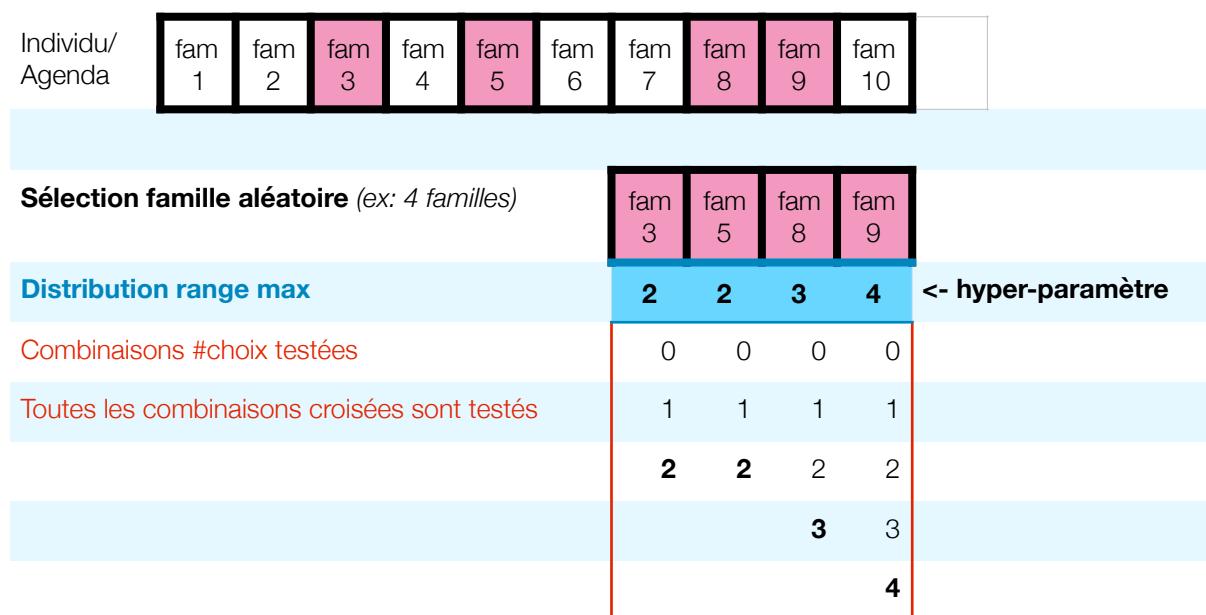
4. Modèle stochastic product search

Le notebook Kaggle suivant m'a permis d'appliquer cette méthode très rapidement :
<https://www.kaggle.com/dmtry/c-stochastic-product-search-in-few-threads>

4.1. Principe

Le principe de la méthode est de simplement trouver un nouveau candidat ayant un coût inférieur en cherchant à évaluer, comme le *boost simple*, tous les choix possibles des familles, mais en choisissant cette fois-ci plusieurs familles à la fois et de façon aléatoire à chaque epoch.

De plus, toutes les combinaisons possibles sont testées dans un range max à définir.



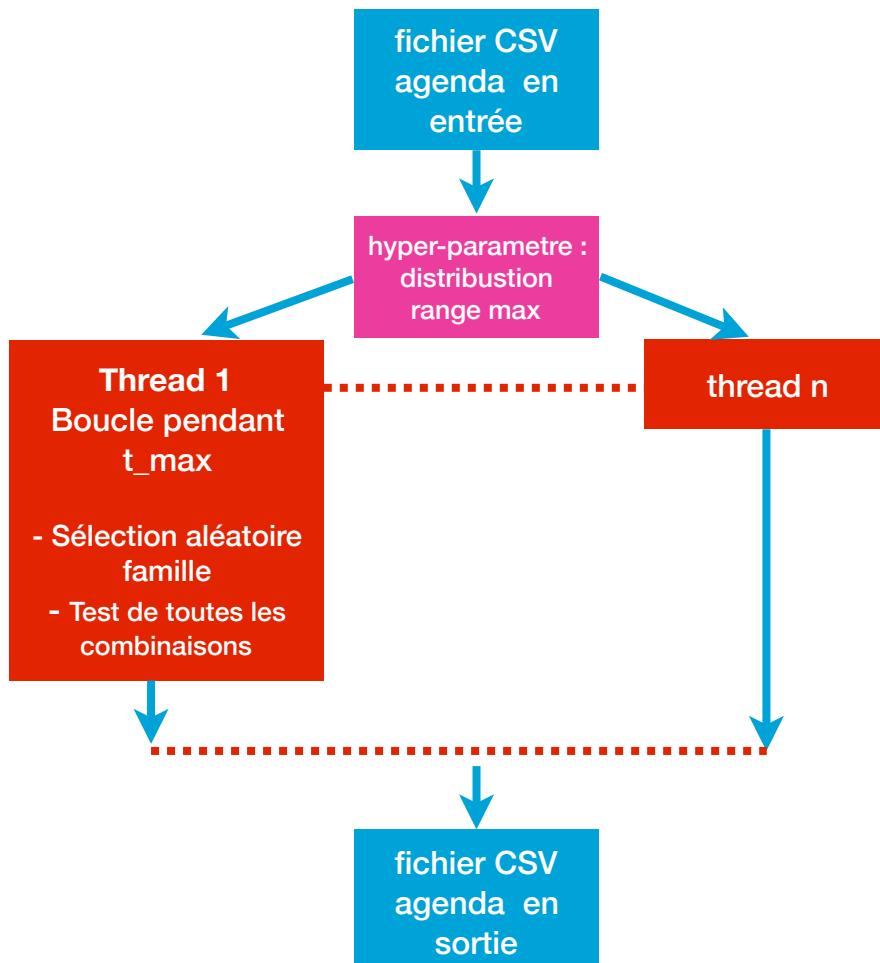
4.2. Implémentation

Le code original a été réalisé en C++. Il peut être compilé à la volée puis exécuté avec un délai maximum d'exécution.

L'individu initial est fourni par un fichier CSV en entrée.

Il calcule sur plusieurs threads. Donc la sélection aléatoire des familles à chaque itération est faite sur plusieurs threads. Pendant l'itération, si un meilleur individu est trouvé alors il est pris comme nouvelle référence pour ce thread. Mais les threads restent totalement indépendant jusqu'à la fin du calcul. Seul le meilleur résultat des threads est pris comme résultat finale exporté sous forme de fichier CSV.

Schéma de principe du programme:



Le programme C++ a été retravaillé pour accepter des paramètres issus de python avec une adaptation avec le module « `ctypes` ». Un soucis avec les arrays de numpy ne m'a pas permis d'éviter les fichiers CSV pour gérer les entrées/sorties des agendas.
 Néanmoins, 2 paramètres ont été rendus accessibles sous python : le temps d'exécution et le nombre de threads en parallèle.

L'implémentation a été effectué avec un script python qui est exécuté en ligne de commande.

Fonction python de lancement d'une recherche d'optimum Stochastic Product Search :

- `run_stochprodsearch(arr_best_curr, end_time=6, nb_jobs=4)`

Elle exécute en ligne de commande la fonction d'interface avec le routine de calcul :

- `stochprodsearch_03.py`

Elle exécute, grâce à `ctypes` la version compilée de la routine de calcul en C++ :

- `stochprodsearch_03.cpp`

Cette optimisation peut donc soit être lancée de façon séquentielle, à la fin d'une optimisation avec l'algorithme génétique, ou bien, en add-on pendant l'optimisation algo génétique, tel que le « *boost simple* » qui s'exécute toutes les 2000 epochs.

En add-on, l'implémentation actuelle est activable dans l'algo génétique. Le principe est le suivant :

- Si le calcul ne progresse plus pendant 1000 epochs par exemple,
 - alors on lance l'optimisation « *stochastic product search* » pendant 30 secondes sur 4 threads.
 - puis on attend les prochaines 1000 epochs pour vérifier la progression de nouveau.
 - On augmente, le temps d'exécution si un nouveau minimum n'est pas trouvé.

4.3. Résultats

4.3.1. Séquentiels

Les résultats montrent que ce modèle ne peut pas démarrer en partant du sample fourni. Il faut que le meilleur individu donné en entrée soit déjà relativement optimisé en *cost*. De même, aux alentours de 72000, le modèle n'arrive plus à progresser dans des temps raisonnables (8h).

Point de départ	Distribution	Temps de calcul	coût de départ	coût final
sample fourni	4 ou 6 ou 8 ou 15 fam.	8h	5600000	5600000
meilleur algo gén.	15 fam. {2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 5}	8h	83000	73800
...	8 fam. {2, 2, 2, 2, 2, 2, 3, 5}	8h	83000	72600
...	6 fam. {2, 2, 2, 2, 3, 5}	8h	83000	72060
...	4 fam. {2, 2, 3, 5}	8h	83000	72193
meilleur algo gén. +s.p.s	6 ou 15	8h	71500	71500

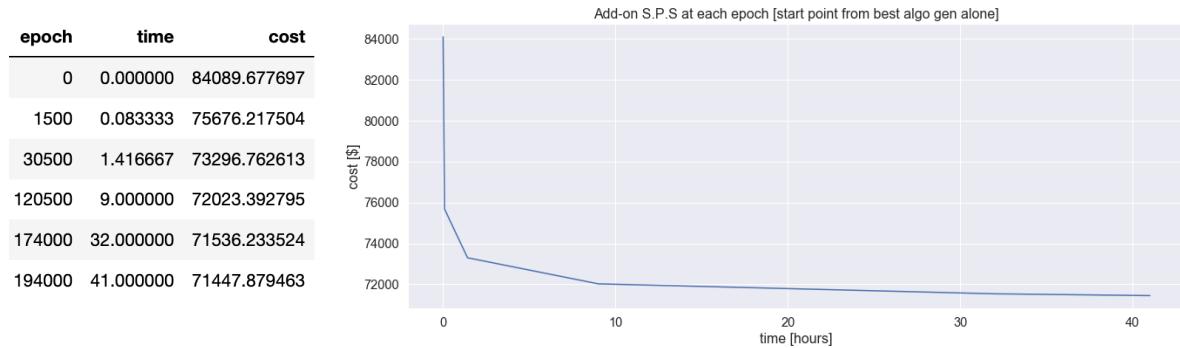
Le meilleur hyper-paramètre est donc une distribution sur 6 familles.

C'est donc la référence utilisée pour les calculs soit en séquentiel (à la suite de l'algo génétique) ou alors en add-on à la méthode algo génétique tout comme le boot simple.

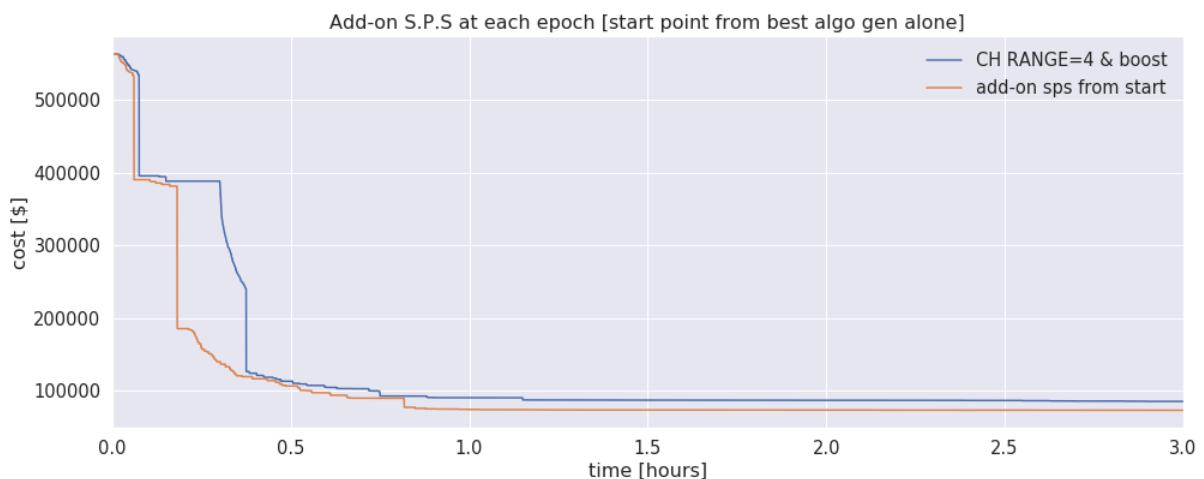
4.3.2. En add-on

Pour ce premier résultat, le add-on S.P.S a été activé à chaque epoch, en partant de la dernière population du calcul algo génétique seul ayant un coût de 84089.

Le temps de convergence est extrêmement long en dessous de 72000.



On voit que l'on améliore la convergence de départ si on commence directement avec l'add-on :



5. Modèle MIP Gurobi

Un **optimizer** Mixed-Integer-Program du commerce a été utilisé par les vainqueurs de cette compétition pour obtenir le score optimal.

Parmi les autres optimizers, Gurobi c'est révélé être le plus efficace.

Site officiel : <https://www.gurobi.com/>

5.1. Principe

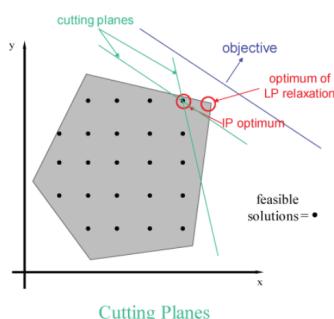
Le principe de cette méthode est de chercher sous contraintes à minimiser la fonction objectif : cost.

Comme ce problème est sur des choix de dates dans un agenda, il est aussi nécessaire d'imposer que nos variables de décision soient entières.

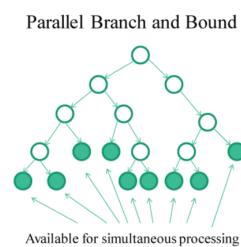
La première étape est de linéariser le problème. (cf chapitre suivant)

La résolution d'un problème MIP par Gurobi est composée des étapes suivantes :

- **Presolve** : qui procède à une simplification des contraintes
- **Cutting Planes** : on réduit l'espace des solutions possibles en fonction des contraintes et en considérant des variables entières



- **Heuristics** : Gurobi utilise certaines techniques pour trouver le prochain bon candidat au problème pendant le processus d'optimisation.
- **Primal simplex, Dual simplex, and Barrier** : Gurobi essaie de trouver un nouveau candidat mieux placé avec différentes méthodes du type Simplex, issu des techniques de « Linear Programming ».
- **Branch and Bound** : Ici il tente de trouver des variables entières avec cette méthode de « MIP » où l'on va chercher des variables entières, solution du problème en partant des solutions en variables non-entières. En gros, si on trouve un bon score avec une variable non-entiére, on va soit remplacer par la valeur entière supérieure ou bien par la variable entière inférieure. Du coup, on découpe le



problème en plusieurs sous-problèmes et si une solution est trouvée avec des variables toutes entières et avec un objectif plus bas que le précédent candidat, cela devient notre nouveau candidat, et ainsi de suite.

Cette phase peut être parallélisée sur plusieurs CPUs. C'est la phase la plus longue dans notre cas.

5.2. Mise en équation

Le problème doit être linéariser avant d'être implémenter. La fonction objectif de notre problème n'est pas linéaire, en particulier l'**accounting penalty** :

$$\text{accounting penalty} = \sum_{d=100}^1 \frac{(N_d - 125)}{400} N_d^{(\frac{1}{2} + \frac{|N_d - N_{d+1}|}{50})}$$

Pour représenter cette équation par un calcul linéaire, on peut créer la matrice **ACC[k][L]** qui donne l'**accounting penalty** en fonction de l'affluence **k** du jour J, et l'affluence **L** du jour J+1.

Ensuite, on crée une matrice binaire 3D **occ[d][k][L]** dont chaque valeur est à 1 si l'affluence du jour **d** est égale à **k** et l'affluence du jour suivant est égale à **L**. Cela devient une partie des variables du système à optimiser.

$$occ_{d,k,l} = \begin{cases} 1 & \text{occupancy}(d) = k \wedge \text{occupancy}(d+1) = l \\ 0 & \text{else} \end{cases}$$

The occupancy variables

La contrainte sous-jacente est que pour un jour **d**, doit avoir qu'une valeur « 1 » dans la matrice **occ** pour tous les **k** et **L**.

L'équation de l'**accounting penalty** précédente s'écrit comme cela : $\sum_{d=1}^{100} \sum_{k=1}^{176} \sum_{l=1}^{176} occ_{d,k,l} * ACC_{k,l}$

C'est donc une matrice binaire, qui, pour un jour **d** donné, n'a qu'un seul « 1 » suivant les axes **k** et **L**. Car il ne peut y avoir d'une seule affluence par jour.

Ensuite, pour la partie **preference cost**, de la même manière, on crée une matrice **PREF[f][c]** qui donne le coût en fonction des numéros de choix **c** des familles **f**.

Puis, on défini une matrice binaire **asm[f][c]** qui représente le fait d'avoir attribué ou non à une famille **f**, son numéro de choix **c**.

$$asm_{f,c} = \begin{cases} 1 & \text{family } f \text{ is assigned choice } c \\ 0 & \text{else} \end{cases}$$

The assignment variables

Une contrainte sous-jacente est qu'il n'y ai qu'un seul choix par famille.

Le cout global, donc la fonction objectif à minimiser devient :

$$\sum_{f=1}^{5000} \sum_{c=1}^{10} asm_{f,c} * PREF_{f,c} + \sum_{d=1}^{100} \sum_{k=1}^{176} \sum_{l=1}^{176} occ_{d,k,l} * ACC_{k,l}$$

The objective of our MIP

Il y a bien sur d'autre contraintes comme celle lié à l'affluence par jour Nd : $125 \leq Nd \leq 300$

L'affluence s'écrit comme ceci : $occupancy(d) = \sum_{f=1}^{5000} \sum_{c=1}^{10} asm_{f,c} * family_size_f * \mathbb{I}(choices_{f,c} = d)$

Calculating the occupancy from the assignment matrix

Elle dépend du vecteur **family_size** qui contient la taille des familles et de la matrice **choices** qui contient les jours choisi par les familles.

Pour un jour **d**, la valeur de $asm_{f,c} * family_size_f$ est ajouté à **occupancy(d)** uniquement si le choix de la famille **f** est bien le jour **d**.

Puis, il y a 2 contraintes supplémentaires qui vérifient que l'**occupancy** de chaque jour **d**, est égale à la somme de toutes les valeurs binaires multipliées par l'affluence **k** de la matrice **occ** à **d** fixé, et de même pour **d+1** avec l'affluence **L** :

$$occupancy(d) = \sum_{k=125}^{300} \sum_{l=125}^{300} occ_{d,k,l} * k$$

$$occupancy(d + 1) = \sum_{k=125}^{300} \sum_{l=125}^{300} occ_{d,k,l} * l$$

Enfin, on vérifie si **occ** indique que pour un jour **d** d'**occupancy k**, le jour d'après l'**occupancy** est **L**, alors il faut que **occ** indique pour le jour **d+1** qu'il y a une **occupancy** de **L**.

$$\text{Pour tout } d \text{ et } L \text{ de } occ : \sum_{K=125}^{300} occ_{d,K,L} = \sum_{L2=125}^{300} occ_{d+1,L,L2}$$

5.3. Implémentation

L'implémentation a été reprise du blog suivant :

<https://towardsdatascience.com/helping-santa-plan-with-mixed-integer-programming-mip-1951386a6ba5>

Le code python est disponible dans le notebook : **santa_gurobi_01.ipynb**

Une licence Gurobi est nécessaire pour faire fonctionner le modèle. Si vous êtes étudiant, c'est gratuit pour un an.

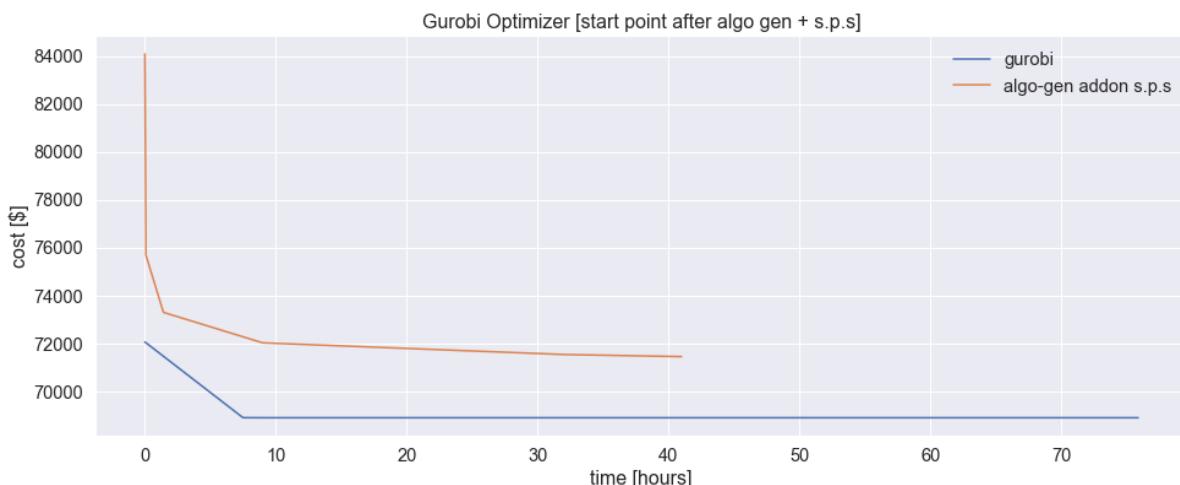
Le modèle nécessite juste comme entrée l'agenda de départ pour initialiser les variables de décision **asm** et **occ**.

5.4. Résultat

Le compétiteur Kaggle parle de 1 semaine de calcul sur un PC 2 coeurs 16Mo de RAM, mais il ne dit pas quel est son agenda de départ.

J'ai donc lancé l'optimisation en partant de mon meilleur résultat issu de l'algorithme génétique enchaîné par l'optimisation Stochastic Product Search.

Point de départ	Temps de calcul	coût de départ	coût final (avant optimal)	time	cost
meilleur résultat avec aglo gen + S.P.S	76h	72060	68895	0.000000	72060.800000
				7.491944	68899.094911
				8.796389	68895.761275
				75.886944	68895.761275



Le calcul Gurobi est donc assez long, en tout cas avec la formulation utilisée. Il n'a pas eu le temps de converger et tourne toujours à l'heure où j'écris ce rapport.

Si on compare avec la méthode algorithme génétique et SPS je pense qu'il est beaucoup plus performant pour démarrer la convergence même si je n'ai pas essayé de partir au même point que pour l'algorithme par faute de temps.

Ensuite, si on regarde sous le seuil de coût à 72000, le modèle Gurobi chute rapidement avec un bon résultat beaucoup plus rapidement que l'algorithme génétique, mais il n'a pas encore trouvé l'optimum au bout de 70h. La barre de la semaine est probablement le temps minimum nécessaire pour avoir l'optimum avec Gurobi. Je pense que le modèle peut être optimisé au niveau de ces bornes de recherche.

6. Conclusions

L' algorithme génétique développé est efficace en 1ère approche.

Sa performance de calcul est accrue avec Numba.

Par contre, il est amélioré avec l'add-on Stochastique Product Search.

L'optimiseur Gurobi est plus rapide pour trouver un meilleur minimum.

Mais, il reste très lent pour trouver l'optimum (1 semaine).

En l'état actuel, le modèle Gurobi est tout de même le meilleur et reste donc le modèle à recommander.

7. Axes d'améliorations

Il est possible d'améliorer l'implémentation S.P.S dans l'algorithme génétique en le déclenchant au bon moment.

Le code de l'algo génétique serait vraiment plus performant s'il avait été codé en C++.

Pour la méthode Stochastique Product Search, l'étude des hyper-paramètres étant très longues, elle peut être améliorée avec plus de temps.

Pour Gurobi, je pense que les bornes du problème pourrait être restreinte pour accélérer la convergence.

8. Sources bibliographiques

Généralités sur algo génétique : <https://khayyam.developpez.com/articles/algo/genetic/>

Documentation Optimiseur type Gurobi :

- Simplification avec Simplex : https://www.gurobi.com/documentation/9.0/refman/simplex_logging.html
- Barrier optimization : <https://pdfs.semanticscholar.org/a187/a63069e2478344cb1fe6aa71c849aecc3fcc.pdf>
- Primal & Dual : <https://www.youtube.com/watch?v=AMUlnn723l0>
- Simplex : https://www.youtube.com/watch?v=jh_kkR6m8H8
- schema explicatif : <https://www.gurobi.com/resource/mip-basics/>

Minimum-cost flow problem :

- https://en.wikipedia.org/wiki/Minimum-cost_flow_problem
- network simplex algorithm : https://en.wikipedia.org/wiki/Network_simplex_algorithm
- Explication des graphes : <http://www.4er.org/CourseNotes/Book%20B/B-IV.pdf>