

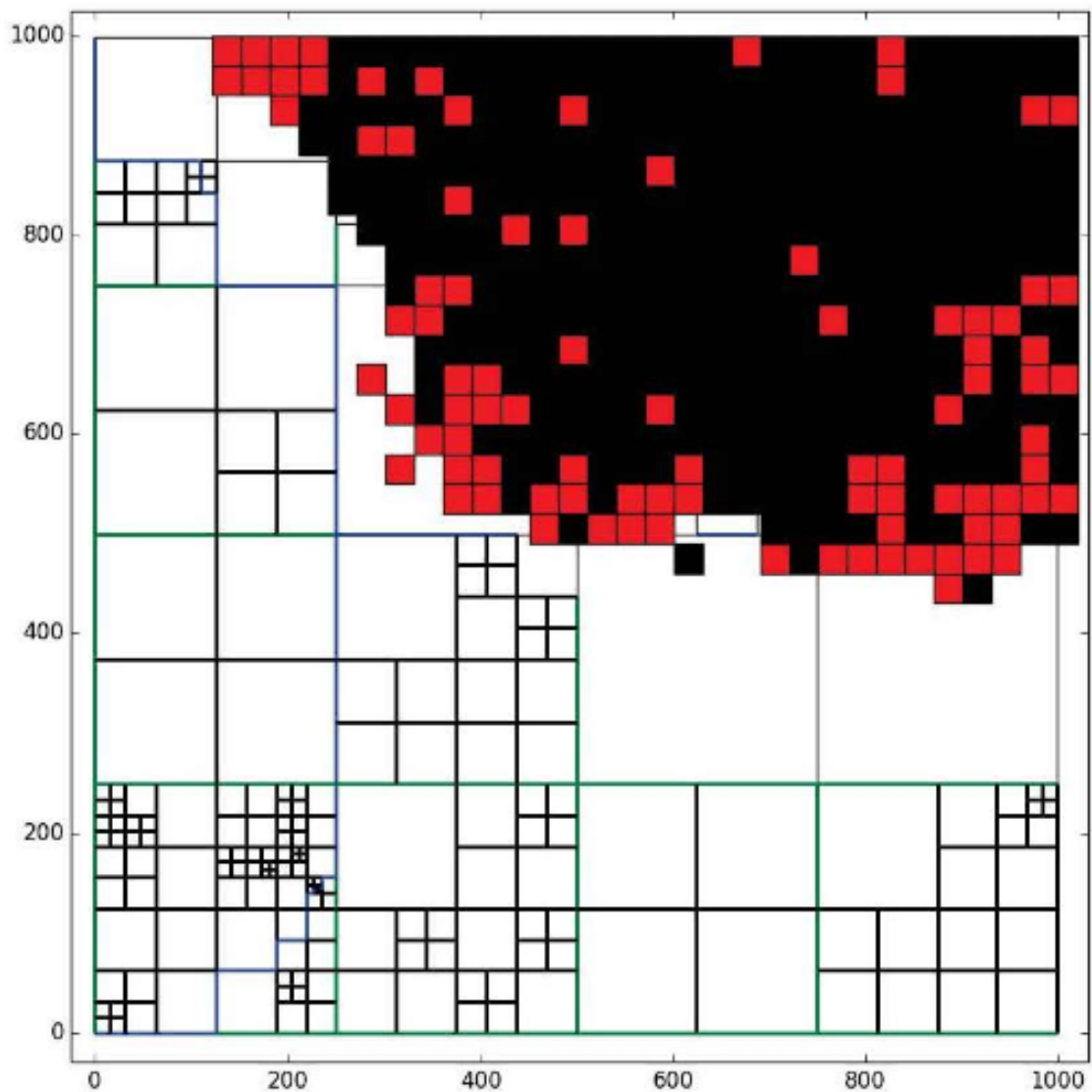
Rapport de projet Métaheuristique

version 6 juin 2019

Maxime Arens - 4IR-B2

Vincent Erb - 4IR-B2

<https://github.com/VincentErb/FireEvac>



1 Rappel du sujet

1.1 Objectifs

- Ce projet a mobilisé les compétences suivantes :
- Métaheuristique et optimisation combinatoire
 - Algorithmique, structure de données, graphes, complexité
 - Développement
 - Bon sens

1.2 Contexte

Nous allons nous intéresser à la planification d'évacuations pour sauver des vies humaines en cas d'incendie. Ce sujet s'inspire d'études menées au LAAS (équipe ROC) dans le cadre d'un projet collaboratif(GeoSafe) se déroulant entre la France et l'Australie : <http://geosafe.lessonsonfire.eu/>.

On supposera que la zone à évacuer et que le modèle de propagation d'incendie sont connus. Différents secteurs critiques à évacuer ont été identifiés et les trajets d'évacuation reliant ces secteurs critiques à un secteur sécurisé ont été définis. Ces trajets d'évacuation forment un arbre enraciné au sommet sécurisé. Ces données sont sous forme de fichiers textes comme ci-dessous :

```

1  c [evacuation info] format: header with <num evac nodes> <id of safe node> then one line per evac node with ...
2  <id of the node> <population> <max rate> <k> <v1> ... <vk> where v1,...,vk is the escape route for this node
3  3 6
4  1 48 8 3 4 5 6
5  2 30 5 3 4 5 6
6  3 33 3 2 5 6
7  c [graph] format: header with <num nodes> <num edges> then one line per edge <node 1> <node 2> <duedate> <length> <capacity>
8  7 8
9  0 1 9223372036854775807 101 50
10 0 2 9223372036854775807 101 50
11 0 3 9223372036854775807 101 50
12 1 4 9223372036854775807 7 8
13 2 4 9223372036854775807 4 5
14 3 5 9223372036854775807 6 3
15 4 5 9223372036854775807 9 10
16 5 6 9223372036854775807 12 11

```

Fig. 1: Fichier exemple de données instance

Le problème de planification d'évacuation revient à gérer l'utilisation partagée des différents tronçons des trajets afin de permettre l'évacuation de tous les secteurs critiques dans un délai minimal.

Certaines contraintes entourent une évacuation, une fois commencée, une évacuation ne peut pas s'arrêter à un noeud du trajet d'évacuation et la taille des convois d'évacuations ne peut changer. Ainsi une évacuation se modifie en jouant sur les dates de départ et la tailles des convois des différents noeuds à évacuer.

1.3 Choix de conception

Afin de résoudre ce problème nous avons décidé d'utiliser le langage informatique **Python** qui se prête assez bien à ce genre de projet. Nous avons utilisé l'IDE Pycharm de JetBrains afin de réaliser ce projet.

Afin de pouvoir se répartir le travail de manière plus simple, chaque partie du TP correspond à un fichier python différent. Nos différentes fonctions sont commentées afin de pouvoir être correctement utilisées à partir de n'importe quel autre fichier.

Nous avons traité toutes les parties du problème sauf la propagation au cours du temps de l'incendie (on ne priorise pas les trajets où le feu va arriver en premier).

2 Première étape - Lecture et vérification de solutions

2.1 Input reader

2.1.1 Commentaires

Après avoir réfléchi sur le sujet et avoir étudié les formats des fichiers de données nous avons décidé de travailler à deux sur cette première partie. La première séance entière fut dédiée à la rédaction d'une première version du lecteur de fichier. Cependant quelques heures de travail en externe furent effectuées afin de changer la structure dans laquelle nous rangions les données tirées des fichiers.

2.1.2 Explications

Notre choix de structure pour stocker les données tirées des fichiers fut l'objet **dictionnaire** de Python. En effet un dictionnaire python est une structure désordonnée de données où chaque donnée "values" est liée à un attribut "key". C'est une structure optimisée de données qui convient à notre situation.

Nous avons donc choisi de stocker les données d'un fichier "instance" (fichier dont les données représente un problème d'évacuation) dans deux dictionnaires : **evac_path** qui contient les chemins d'évacuations et **arcs** qui contient les informations sur les arcs du graphe (ou les branches de l'arbres). Afin de pouvoir tester ce parser nous avons donc implémenté une fonction capable d'afficher (print) ces dictionnaires. Par la suite comme il fut question d'exploiter un fichier solution, nous fîmes une fonction capable de créer un dictionnaire **solution** à partir d'un fichier solution.

2.1.3 Readme

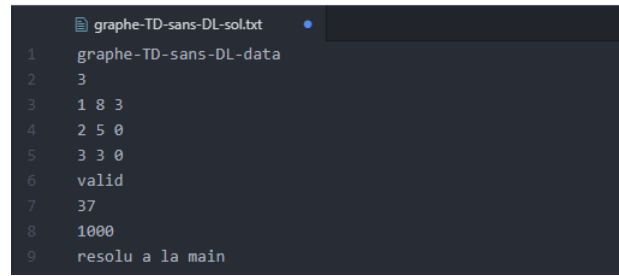
- **parse_instance** est une fonction qui prend pour argument le chemin vers un fichier de données et renvoie deux dictionnaires de données **evac_path** et **arcs**.

Afin de "parser" le fichier et de remplir les dictionnaires, la fonction lit les lignes une à une et mots par mots. Ainsi comme les fichiers de données ont une structure bien spécifique on sait suivant le placement de chaque mot où le placer dans le dictionnaire. Pour le dictionnaire **evac_path** nous attribuons à chaque noeud de départ (keys) les valeurs suivantes (values) : **pop**, **max_rate**, **route_length** et **route_nodes**. Pour le dictionnaire **arcs** nous attribuons à chaque tuple de noeud (keys) les valeurs suivantes (values) : **duedate**, **length** et **capacity**.

- **parse_solution** est une fonction qui prend pour argument un chemin vers un fichier solution et qui renvoie un dictionnaire **solution**.

- **print_input_data** est une fonction qui prend pour arguments un dictionnaire de type **evac_path** et un de type **arcs** et les print sur le terminal à l'aide d'une boucle **for**.

Le parser de solution fonctionne de manière similaire à celui qui parse les instances. Pour le dictionnaire **solution** nous attribuons à chaque **node_data** (keys) les valeurs suivantes (values) : **instance_name**, **nb_evac_nodes**, **validity**, **aim_function**, **time** et **method**. **node_data** est un tuple composé de : **id** (de départ), **evac_rate** et **start**.



```
graphe-TD-sans-DL-sol.txt
1  graphe-TD-sans-DL-data
2  3
3  1 8 3
4  2 5 0
5  3 3 0
6  valid
7  37
8  1000
9  resolu a la main
```

Fig. 2: Fichier exemple de données solution

2.2 Vérificateur de solution

2.2.1 Commentaires

L'implémentation du vérificateur de solutions, abrégé checker, a été sujet d'une très grande phase de réflexion. En effet, notre compréhension du sujet ainsi que nos compétences en python ne nous permettaient pas d'appréhender le problème facilement. Nous avons passé deux séances complètes à conceptualiser le checker, après quoi nous avons dû prendre la décision de dédier l'un de nous au développement exclusif du checker, pendant que l'autre implémentait les bornes et fonctions de recherche locale. Le checker a donc pour nous été de très loin la plus grande source de difficultés, ce qui nous a semblé très dommage compte tenu du faible intérêt pédagogique de celui-ci par rapport au reste du TP.

2.2.2 Explications

Par manque de temps, nous ne prenons pas en compte les *duedates* dans notre résolution du problème. Le vérificateur de solution effectue donc deux vérifications : le respect de la capacité de chaque noeud par l'*evac_rate* choisi pour le chemin d'évacuation, ainsi que la vérification dynamique de respect des capacités à chaque instant. Pour cette dernière, nous avons représenté le problème avec une structure de diagramme de Gantt.

L'algorithme de vérification du *max_rate* est assez simple, il va parcourir tous les arcs de chaque chemin d'évacuation, et pour chaque arc, vérifier si sa capacité ne dépasse pas l'*evac_rate* choisie par une solution.

L'algorithme de vérification des capacités à chaque instant est bien plus complexe. Il fonctionne en deux parties :

- Il faut d'abord représenter notre problème sous forme de diagramme de Gantt. Pour cela, nous créons un dictionnaire python de la forme :
`{'noeud de départ', ('noeud1', 'noeud2') : 'evac rate', 'start time'}`
- Ensuite, pour vérifier les capacités, on va sélectionner tous les arcs impliqués dans au moins un gantt, puis itérer dessus. Pour chaque arc, on calcule les dates de début et de fin de passage sur celui-ci, puis on itère sur chaque unité de temps entre les deux dates, et on vérifie à chaque instant que la capacité n'est pas dépassée.

L'algorithme prend en compte le fait que le ratio :

$$\frac{Population}{ArcCapacity} \quad (1)$$

Puisse donner un reste, et donc une unité de temps où le flot présent sur un arc ne vaut pas forcément sa capacité maximale, mais la valeur suivante :

$$1 \leq Actualrate \leq Capacity \quad (2)$$

Note : L'implémentation de cette fonctionnalité est peut-être source de légers faux-positifs.

Enfin, la fonction de calcul d'objectif se fait assez simplement, en déroulant l'exécution représentée par un gantt de la solution, et notant le temps maximal de fin de gantt.

2.2.3 Complexité

La complexité du checker est probablement la plus importante, car celui-ci va être appelé à chaque modification de paramètre par l'heuristique. Nous avons choisi comme structure de données pour tout notre projet des dictionnaires python, car ceux-ci permettent un accès aux données par clé en $O(1)$, ce qui nous intéresse beaucoup. Malgré cela, notre implémentation du vérificateur demeure très complexe.

En effet, la fonction `check_max_rate` est en $O(M)$, avec M la taille moyenne des chemins d'évacuation. Il en va de même pour la fonction de création de gantt, et celle de calcul de fonction objectif. Enfin, la fonction `check_capacity` est la plus complexe, avec une complexité en $O(M + N*M)$, avec N le nombre total d'arcs.

Au final, nous avons une complexité que l'on peut arrondir en $O(N*M)$, ce qui est assez élevé et explique bien le temps d'exécution de notre algorithme.

2.2.4 Readme

Le checker est contenu dans le module `checker.py`. Il s'utilise de la manière suivante :

- **run** est une fonction qui prend pour argument un chemin vers un fichier instance ET un fichier solution et qui renvoie un affichage textuel de vérification de la validité de la solution.

- **run_dico** est une fonction qui prend pour argument un chemin vers un fichier instance et un dictionnaire solution et qui renvoie un affichage textuel de vérification de la validité de la solution.

- **run_with_objective** est une fonction qui prend pour argument un chemin vers un fichier instance et un dictionnaire solution et qui renvoie un tuple booléen/entier : (valide, fonction_objectif).

- **run_with_objective_** est une fonction qui prend pour argument un dictionnaire d'instance déjà parsée et un dictionnaire solution et qui renvoie un tuple booléen/entier : (valide, fonction_objectif).

3 Bornes du problème

3.0.1 Commentaires

Entre deux séances de Tp l'un de nous a réfléchi à différentes manières de borner le problème. Il a fallu environ une séance pour implémenter cette partie. Pour les deux bornes, les idées théoriques que nous avons retenues ont été validées par un professeur. Comme nous nous sommes rendu compte que parfois la valeur `max_rate` dans les fichiers instances était fausse nous la calculons nous-mêmes.

3.1 Borne inférieure

3.1.1 Explications

Pour la borne inférieure, nous avons choisi de faire partir chaque convoi dès le temps 0 avec des tailles de convoi égales au débit maximal du trajet d'évacuation (plus petite capacité de transit des arcs sur le trajet). On ne prend pas en compte les possibles chevauchement d'évacuation (deux convois utilisant un même arc pourrait dépasser sa capacité.) Ainsi nous obtenons une valeur de temps d'évacuation qui sera au mieux égale ou sinon inférieur au temps d'évacuation optimal.

3.1.2 Readme

- **borne_inf** cette fonction prend pour arguments un dictionnaire `evac_path` et un arcs, elle retourne la valeur objectif de temps d'évacuation. L'algorithme parcourt les chemins d'évacuations afin de faire la somme de la taille de chaque arcs pour obtenir la taille totale de chaque chemin d'évacuation. Ensuite pour chacun des chemins on additionne ce trajet total et le résultat de la division du nombre de personnes à évacuer par le débit max d'évacuation. On obtient ainsi la durée d'évacuation d'un chemin d'évacuation. La fonction retourne le plus long de ces durées, c'est-à-dire le temps d'évacuation totale de tout les noeuds avec une évacuation simultanée.

3.2 Borne supérieure

3.2.1 Explications

Pour la borne supérieure, nous avons choisi de faire partir chaque convoi les uns après les autres. L'évacuation `n+1` prend place une fois que l'évacuation `n` est terminée. Encore une fois, chaque évacuation s'effectue avec une taille de convoi égale au débit maximal du trajet. Ainsi on obtient l'heuristique d'une évacuation qui fonctionne (si on ne prend pas en compte la propagation d'incendie) à coup sûr car il n'y a pas de conflits entre les différentes évacuations.

3.2.2 Readme

- **borne_sup** cette fonction prend pour arguments un dictionnaire `evac_path` et un arcs, elle retourne une heuristique du temps d'évacuation. L'algorithme parcourt les chemins d'évacuations afin de faire la somme de la taille de chaque arcs pour obtenir la taille totale de chaque chemin d'évacuation. Ensuite pour chacun des chemins on additionne ce trajet total et le résultat de la division du nombre de personnes à évacuer par le débit max d'évacuation. On obtient ainsi la durée d'évacuation d'un chemin d'évacuation. La fonction retourne la somme

de ces durées, c'est-à-dire le temps d'évacuation totale de tout les noeuds avec des évacuations consécutives.

3.3 Solution borne supérieure

3.3.1 Explications

Afin de pouvoir faire la recherche locale nous avons besoin de partir d'une solution fonctionnelle du problème. Nous avons donc fait une fonction qui permet de créer un dictionnaire solution à partir d'un A. Ainsi on obtient une heuristique d'une évacuation qui fonctionne (si on ne prend pas en compte la propagation d'incendie) à coup sûr car il n'y a pas de conflits entre les différentes évacuations.

3.3.2 Readme

- **borne_sup** cette fonction prend pour arguments un dictionnaire `evac_path` et un arcs, elle retourne un dictionnaire `solution`. L'algorithme fonctionne comme celui de `borne_sup` sauf que nous mémorisons certaines valeurs au cours de sa réalisation afin de remplir ensuite le dictionnaire à retourner.

3.4 Exemple

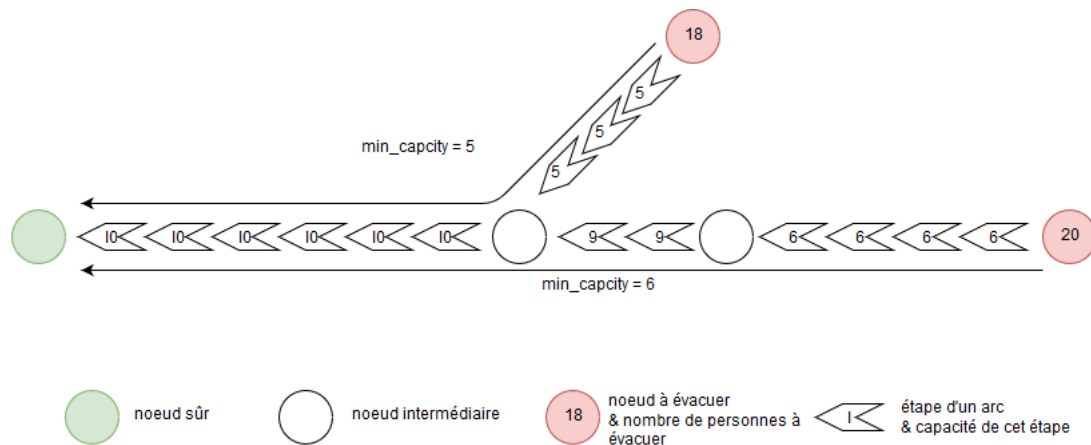


Fig. 3: Diagramme d'une évacuation simple

3.4.1 Borne inférieure

Pour l'évacuation du noeud avec 18 personnes à évacuer :

- en additionnant les étapes des arcs sur le trajet d'évacuation on obtient une taille totale de chemin de $3+6 = 9$
- la capacité minimale des arcs sur le trajet d'évacuation est égale à 5
- 18 divisé par 5 est égale à 3 avec un reste de 3.
- Le convoi sera donc composé de 3 voitures de 5 personnes et une voiture de 3 personnes.

- La dernière voiture va donc mettre 9 unités de temps (trajet) + 4 unités de temps (convoi de 4 voitures) = 13 unités de temps, c'est le temps d'évacuation de ce noeud

Pour l'évacuation du noeud avec 20 personnes à évacuer :

- en additionnant les étapes des arcs sur le trajet d'évacuation on obtient une taille totale de chemin de $4 + 2 + 6 = 12$
- la capacité minimale des arcs sur le trajet d'évacuation est égale à 6
- 20 divisé par 6 est égale à 3 avec un reste de 2.
- Le convoi sera donc composé de 3 voitures de 6 personnes et une voiture de 2 personnes.
- La dernière voiture va donc mettre 12 unités de temps (trajet) + 4 unités de temps (convoi de 4 voitures) = 16 unités de temps, c'est le temps d'évacuation de ce noeud

$16 > 13$: la fonction `borne_inf` renvoie donc 16 .C'est le temps objectif d'évacuation de cette solution. On remarque que sur les arcs de capacité 10 il y aurait pu avoir un chevauchement des 2 convois ($5 + 6 = 11 > 10$) mais on ne le prend pas en compte dans cette fonction.

3.4.2 Borne supérieure

Pour l'évacuation du noeud avec 18 personnes à évacuer :

- en additionnant les étapes des arcs sur le trajet d'évacuation on obtient une taille totale de chemin de $3+6 = 9$
- la capacité minimale des arcs sur le trajet d'évacuation est égale à 5
- 18 divisé par 5 est égale à 3 avec un reste de 3.
- Le convoi sera donc composé de 3 voitures de 5 personnes et une voiture de 3 personnes.
- La dernière voiture va donc mettre 9 unités de temps (trajet) + 4 unités de temps (convoi de 4 voitures) = 13 unités de temps, c'est le temps d'évacuation de ce noeud

Pour l'évacuation du noeud avec 20 personnes à évacuer :

- en additionnant les étapes des arcs sur le trajet d'évacuation on obtient une taille totale de chemin de $4 + 2 + 6 = 12$
- la capacité minimale des arcs sur le trajet d'évacuation est égale à 6
- 20 divisé par 6 est égale à 3 avec un reste de 2.
- Le convoi sera donc composé de 3 voitures de 6 personnes et une voiture de 2 personnes.
- La dernière voiture va donc mettre 12 unités de temps (trajet) + 4 unités de temps (convoi de 4 voitures) = 16 unités de temps, c'est le temps d'évacuation de ce noeud

Comme on considère les évacuations comme successives on a le temps d'évacuation qui est égal à la somme du temps d'évacuation des noeuds ici $16 + 13 = 29$. C'est la valeur de notre heuristique pour cette évacuation.

4 Recherche locale - Intensification

4.1 Commentaires

La partie de recherche locale en intensification a été relativement rapide à implémenter. La théorie de la méthode a été confirmée par un professeur de TD.

4.2 Description

Pour descendre la valeur de la borne supérieure vers celle de la borne inférieure on tire au hasard un trajet à modifier puis on tire au hasard une modification de ce trajet. Les solutions ainsi modifiées sont le voisinage de notre solution de départ, sur 20 on retient la meilleure et on ré-itére l'algorithme.

4.3 Readme

- **local_search** cette fonction prend en entrée un chemin d'instance, un dictionnaire `evac_path`, un dictionnaire arc et un dictionnaire solution et retourne un autre dictionnaire solution.

La partie intensification se fait par le tirage au sort de deux nombres aléatoires : un pour le chemin à modifier et un pour la modification à effectuer. On appelle ensuite la fonction `modify_solution` qui prend en entrée ces nombres aléatoires et nous renvoie la solution modifiée.

On teste si la nouvelle solution est mieux que la meilleure solution à ce jour, si c'est le cas on la remplace. On ré-itére avec cette meilleure solution (qui a changée ou est restée la même).

La recherche s'arrête si on égalise la fonction objectif (même si on arrive à la meilleure solution et qu'elle est différente de la fonction objectif on ne s'arrête pas) ou si on atteint le nombre d'itérations maximum.

- **modify_solution** cette fonction prend en entrée un dictionnaire solution, un entier correspondant à un chemin à modifier et un entier correspondant à une modification.

Au départ de la fonction on fait une copie profonde du dictionnaire, ce qui est essentiel vu que notre dictionnaire contient un autre dictionnaire. En effet, si on faisait une simple copie, les deux copies du dictionnaires solution contiendraient le même dictionnaire `node_data`.

L'entier du chemin à modifier correspond au numéro de départ du chemin qui est la clé du dictionnaire `node_data` qui est lui même une clé du dictionnaire solution.

L'entier pour les modifications entraîne 4 modifications possible : +1 ou -1 sur le temps de départ / +1 ou -1 sur la taille du convoi. On renvoie la nouvelle solution

4.4 Expérimentation

Nous avons testé la recherche locale seulement sur l'exemple vu en TD, notre heuristique haute avait un résultat de 94 et la recherche en intensification permettait de descendre jusqu'à 65.

On ne peut descendre plus car on est bloqué dans l'espace de solution, il faut passer par des solutions fausses pour trouver de meilleures solutions justes. C'est la diversification.

5 Recherche locale - Diversification

5.1 Commentaires

Cette partie a été plus longue à implémenter que la précédente. Notamment la partie test qui a été plus poussée.

5.2 Description

Afin de diversifier notre descente nous acceptons d'explorer des solutions où la solution est fausse. On met cependant un compteur afin qu'après un certain nombre de solutions fausses explorées on retourne à une solution vraie.

5.3 Readme

L'implémentation de la diversification c'est faite par la mise en place d'un compteur et le rajout de deux conditions **if** qui permettent de traiter le cas où la solution est fausse et le cas où la solution est fausse et le compteur à atteint son maximum.

On enregistre dans deux tableaux la meilleure solution vraie et la meilleure solution fausse afin de pouvoir retourner à la dernière meilleure solution vraie en cas de dépassement du compteur lors de l'exploration de solutions fausses.

5.4 Experimentation

Au départ nous testions seulement sur l'instance la plus simple à notre disposition, c'est-à-dire le graphe exemple vu en TD. Au bout d'un moment nous sommes passés sur les fichiers sparse, medium puis dense.

Nos temps d'exécution ont vite explosés (plus de 10 minutes pour un fichier sparse) nous avons donc étudié en détails la répartition du temps entre les différentes fonctions appelées.

Il se trouve que nous parsions le fichier à chaque itération et que cela prenait plus de 50% du temps d'exécution. En faisant une nouvelle fonction `run_with_objective` qui ne nécessite pas de parser le fichier mais prend directement les dictionnaires `evac_path` et `arcs` nous avons réduit notre temps de calcul de plus de moitié!

En enlevant nos prints de debug nous avons encore amélioré notre temps de calcul.

	exempleTD	sparse_2	medium_2	dense_2
Temps de calcul (s)	0.535	99.81	171.329	218.71
Différence avec la valeur objectif	0	0	0	0

Fig. 4: Tableau des temps d'exécutions pour différentes instances

On peut voir sur ce tableau que la densité impacte le temps de calcul. C'est le nombre d'itérations qui joue sur cette durée (l'heuristique est bien plus haute et donc plus longue à diminuer) et non la taille de nos structures (un grand dictionnaire est aussi rapide qu'un petit).

	reverse_arcs	check_capacity	create_gantt	check_max_rate	calculate_objective
Temps (%)	31.3	19.6	15	9.6	5.4

Fig. 5: Tableau de la répartition du temps d'exécutions des fonctions

Ce tableau montre les temps d'exécution respectifs des fonctions appelés par `local_search`. Malgré le fait qu'on utilise des structures optimisées pour leur usage, on voit que les fonctions qui nécessitent un parcours de dictionnaire restent les plus gourmandes en terme de temps.

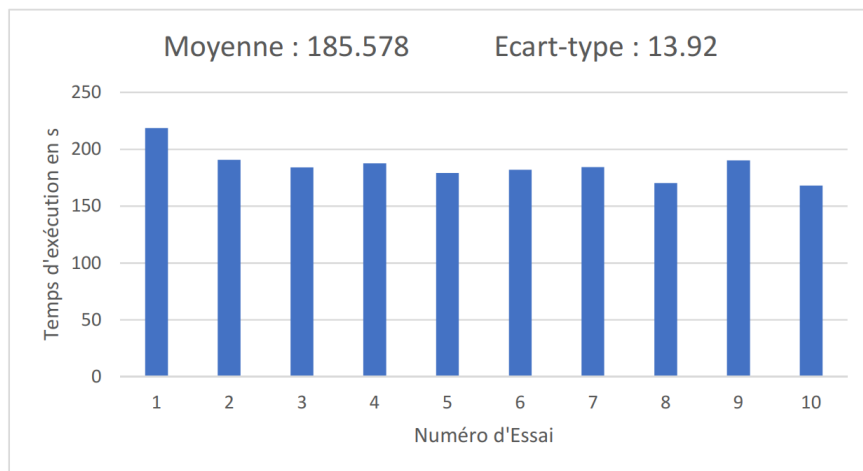


Fig. 6: Tableau d'évaluation de l'algorithme sur la même instance

Ce tableau permet de démontrer le caractère aléatoire de notre algorithme. On voit en effet que le temps d'exécution varie sur l'exécution de la même instance. Un écart-type de 14 sur des solutions tournant autour de 185 reste une variation aléatoire assez importante.

	sparse_1	sparse_2	sparse_3	sparse_4	sparse_5	sparse_6	sparse_7	sparse_8	sparse_9	sparse_10
Résultat	111	108	110	128	131	108	121	107	111	100
	medium_1	medium_2	medium_3	medium_4	medium_5	medium_6	medium_7	medium_8	medium_9	medium_10
Résultat	106	96	114	129	124	138	103	96	111	98
	dense_1	dense_2	dense_3	dense_4	dense_5	dense_6	dense_7	dense_8	dense_9	dense_10
Résultat	91	120	106	124	230	126	104	105	98	101

Fig. 7: Tableau des temps d'évacuation pour chaque instance

Ce tableau révèle la limite de notre checker. Ici nous avons fait un script afin de tester tout les fichiers instances. En vert sont noté les résultats d'exécutions égaux à la valeur objectif. En rouge le seul résultat où ce n'est pas le cas.

Dans la plupart des cas le checker nous renvoie que certaines solutions sont juste alors qu'elles ne le sont pas et la recherche local continue de s'approcher de la valeur objective jusqu'à ce qu'une condition la fasse sortir de ces itérations. Sans cette condition la recherche locale continue la plupart du temps à améliorer la solution en dessous de la borne minimale.

6 Conclusion

Le checker n'étant pas entièrement fonctionnel (cf. partie checker) la recherche locale ne s'arrêtait pas après avoir dépassée la fonction objectif. Nous avons partiellement résolu le problème en faisant sortir le programme lorsqu'on égalise la fonction objectif. Mais on ne peut être certain que cette solution soit juste. Quasiment à chaque fois la fonction de recherche s'arrête et son résultat est égal à la borne inférieure, on peut se dire que le checker ne renvoie pas le résultat faux suffisamment tôt au cours de la descente.

Cela ne nous a pas empêché de tester nos solutions et surtout d'avoir des informations sur les caractéristiques de l'exécution de la recherche locale.