BOUZEREAU Laura
GAY Constance

Rapport Métaheuristique

06/06/2019

https://www.sciencesetavenir.fr

Introduction

La planification des évacuations en cas de situation d'urgence est une question importante de la sécurité. Cependant, la gestion des flux de personnes n'est pas un problème solvable facilement. L'utilisation de connaissances provenant du domaine de la métaheuristique devient alors nécessaire.

C'est dans ce cadre que nous avons étudié ce domaine de l'optimisation. Au travers de notre projet d'optimisation de l'évacuation de points d'intérêts, nous démontrons toute la complexité et la réflexion nécessaire pour concevoir des algorithmes satisfaisant au mieux le problème posé.

Afin d'approfondir notre démarche, nous commencerons par présenter notre méthode pour vérifier que les solutions proposées sont valides, tout d'abord en décrivant nos hypothèses de travail, le format de données choisi et enfin l'algorithme de vérification. Nous présenterons ensuite comment nous avons cerné le problème par la création de bornes inférieure et supérieure. Nous décrirons ensuite la phase d'intensification de la recherche locale de solution, que nous avons abordé de deux manières différentes, à savoir une recherche aléatoire et une recherche successive de validité puis d'optimalité. Enfin, nous discuterons de l'approche de la phase de diversification pour améliorer nos algorithmes de recherche.

I. Vérification de la validité des solutions envisagées

1) Hypothèses de travail

Lors de l'étude du problème et de l'analyse des fichiers fournis, nous avons listé les hypothèses de travail qui nous ont paru importantes pour simplifier notre travail, à savoir:

- On doit évacuer tout le monde : les nombre de personnes sauvées ne doit cond pas être pris en compte dans nos heuristiques.
- Il n'y a qu'un arc qui sort de chaque noeud, c'est-à-dire qu'un arc peut être complètement retrouvé avec comme seul identifiant celui du noeud père.
- Il n'y a qu'un chemin d'évacuation par noeud à évacuer

A noter qu'avec la contrainte de péremption des arcs, il est possible que certaines instances du problème n'aient pas de solution valide. Il aurait pu être intéressant de considérer la maximisation du nombre de personnes sauvées mais ceci est hors-sujet.

2) Format des données

Dans l'implémentation du problème en java, nous avons décidé de représenter les données en deux classes principales.

La première classe est la classe *graph*, qui représente le graphe tel qu'exprimé dans le fichier.

```
public class Graph {

    private int nb_node;                        //number of nodes in total
    private int nb_evac_nodes;                  //number of nodes to evacuate
    private int safe_node;                      //the node that needs to be reached
    private ArrayList<Node> evac_nodes;         //list of nodes to evacuate
    private ArrayList<Node> Nodes;              //list of nodes
```

La classe Node permettant de représenter les noeuds du fichier avec toutes les informations requises.

```java
public class Node {

    private int id;
    private Arc arc_evac;
    private int population;
    private int max_rate;
    private ArrayList<Integer> evac_path;
```

La deuxième classe est la classe *solution*, elle représente les taux d'évacuation et les dates de début d'évacuation de chaque noeud dans une solution donnée.

```java
public class Solution {
    private String filename;
    private int nb_evac_node;
    private ArrayList<EvacNode> list_evac_node;
    private boolean is_valid;
    private int date_end_evac;
    private long calcul_time=0;
    private String method;
    private String free_space;
```

Avec la classe EvacNode qui permet de contenir toutes les choix relatifs à un noeud (taux d'évacuation et date de début d'évacuation)

```java
public class EvacNode {
    private int id_node;
    private int rate;
    private int start_evac;
```

3) Principe et algorithme du checker

Notre approche de la vérification des solutions se base sur la présomption de validité. Une solution est considérée comme valide jusqu'à ce qu'on ait prouvé le contraire.

Les éléments discriminants la validité d'une solution sont:

- Dépassement de la date de péremption de l'entrée d'un arc
- Dépassement de la date de péremption d'un noeud à évacuer
- Dépassement de la capacité totale d'un arc
- Dépassement du taux d'évacuation maximal d'un noeud à évacuer
- Discordance entre le temps d'évacuation annoncé et celui constaté

Notre algorithme se base sur la représentation du nombre de personnes entrants dans un arc à un temps donné. Tout d'abord, nous créons une matrice de taille nombre_d_arc x temps_d'évacuation_total.

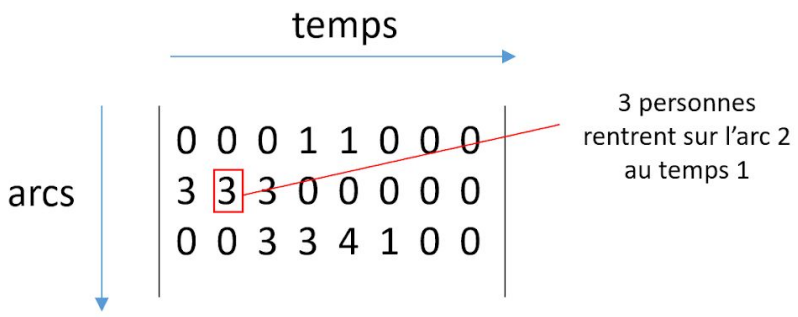


Schéma de la représentation matricielle d'une solution

Notre algorithme est alors un simple parcours de tous les noeuds d'évacuation d'une solution, pour lequel on parcourt tout le chemin d'évacuation en remplissant chaque case avec le bon nombre de personne.

On effectue régulièrement des tests de validité sur toutes les fonctionnalités prises en compte afin d'arrêter le checker dès qu'une caractéristique invalidante est détectée.

Soit n le nombre de noeud à évacuer, m le nombre d'arcs et t la date de fin d'évacuation de la solution. La complexité de notre algorithme est donc o(n.m.t).

4) Amélioration du checker

Afin de pouvoir améliorer notre algorithme de descente, nous avons modifié notre checker. Au lieu de renvoyer un booléen, nous avons créé une classe *Checker_message* contenant de plus amples informations, comme par exemple la raison pour laquelle la solution est invalide. Les raisons peuvent être *"duedate"* ou *"overflow"*. Nous renvoyons également la liste des noeuds qui provoquent ce problème.

```
public class Checker_message {
    private boolean validity;
    private String reason;
    private ArrayList<Integer> list_nodes_to_change;
```

Une autre piste d'amélioration était de ne pas arrêter l'algorithme à la première invalidité détectée et de quantifier l'invalidité d'une solution. Cela pourrait servir comme heuristique dans la recherche d'une solution valide (notre maximum étant rarement valide à

cause des dates de péremption). Une des fonctions objectif envisagées était de faire la somme du nombre d'arc en surcharge et du nombre d'arc traversés après péremption.

II. Calcul des bornes inférieure et supérieure

Tout notre travail se base sur l'emploi de la fonction objectif qui nous renvoie la durée d'évacuation totale. Cela a une influence sur le calcul de nos bornes ainsi que sur la génération de nos voisinages.

1) Calcul de la borne inférieure

La borne inférieure est la solution qui possède la date d'évacuation la plus courte (en dépit de sa validité). Cette durée peut être approximée par l'évacuation de tous les noeuds à leur taux maximal et démarrant le plus tôt possible.

La durée d'évacuation sera donc le plus long temps d'évacuation d'un noeud d'évacuation en particulier. C'est-à-dire le chemin d'évacuation critique.

Le calcul de ce chemin passe par une itération sur tous les noeuds à évacuer pour lesquels on itère sur tous les arcs composant leur chemin d'évacuation. Soit n le nombre de noeuds à évacuer et m le nombre d'arêtes, la complexité de notre calcul de borne inférieure est de complexité o(n.m).

2) Calcul de la borne supérieure

La borne supérieure est la solution qui possède la date d'évacuation la plus longue (en dépit de sa validité). Cette durée peut être approximée par l'évacuation séquentielle de chaque noeud, c'est-à-dire qu'un noeud ne commence son évacuation que si le précédent a fini d'évacuer.

La durée d'évacuation sera donc la somme des durées d'évacuation de chaque noeud à évacuer.

Ce calcul passe par une itération sur tous les noeuds à évacuer pour lesquels on itère sur tous les arcs composant leur chemin d'évacuation. Soit n le nombre de noeuds à évacuer et m le nombre d'arêtes, la complexité de notre calcul de borne supérieure est de complexité o(n.m).

III. Recherche locale et intensification

1) Première approche par une génération de voisinages aléatoires

Notre première approche à l'implémentation d'une fonction de recherche locale fût d'utiliser la recherche aléatoire.

Tout d'abord nous avons créé une méthode *generateNeighborhoodRandom* qui prend en entrée la taille du voisinage à générer, le delta maximal sur le taux de sortie et la date de départ. Il parcourt la liste de noeud d'évacuation et pour chacun il choisit aléatoirement si il va augmenter ou diminuer le taux (et la date) et ensuite il choisit aléatoirement de combien il va les changer (en prenant en compte le delta maximum).

Nous avons ensuite écrit une méthode *recherche_local_sans_diversification* qui va tourner sur un nombre d'itération donnés. Sur chaque boucle il va générer un voisinage avec la fonction *generateNeighborhoodRandom*. Ensuite il va parcourir ce voisinage en cherchant le meilleure solution. Notre heuristique est la date de fin d'évacuation. Nous mettons cependant une priorité sur les solutions valides, donc si nous comparons une solution fausse avec une heuristique meilleure qu'une solution valide nous garderons la solution valide.

2) Deuxième approche validité-optimalité

Dû à la contrainte sur la date de péremption des arcs, la recherche de solution valide est devenue une part importante de notre conception. Après avoir envisagé d'utiliser un système de handicap pour les solutions invalides, nous avons fini par opter pour rendre prioritaire toute solution valide sur une solution invalide. Notre algorithme fut bien plus efficace à partir de cette étape.

Néanmoins, dans notre première version de l'algorithme de descente, le seul critère était de minimiser la durée d'évacuation en espérant trouver une solution valide au cours de l'exécution. Nous avons donc décidé de segmenter notre algorithme de descente en deux parties : la première partie concerne la recherche d'une solution valide et la deuxième partie l'optimisation de cette solution.

Pour ce faire, nous avons modifié le checker. Ce dernier renvoie un objet de classe *Checker_message* permettant de savoir la raison de l'invalidité d'une solution. Dans le cas d'une péremption d'arc, il faut générer un voisinage en diminuant les dates de départs des noeuds à évacuer prédécesseurs de l'arc concerné en utilisant la fonction *generate_Neighborhood_DATE*. Dans le cas d'une surcharge d'un arc, le voisinage peut être généré en diminuant les taux d'évacuation des noeuds concernés *generate_Neighborhood_RATE*. Cependant, il peut arriver qu'une surcharge ne se règle

qu’avec une bonne gestion des départs des évacuations, sans savoir s’il faut les augmenter ou les diminuer. De même, il n’est pas dit , même en modifiant ce paramètres, qu’une solution valide soit dans le voisinage.

Afin de pallier ce problème, une solution pourrait être de créer un checker modifié qui quantifierait l’invalidité d’une solution (par exemple avec la somme du nombre d’arc en surcharge et du nombre d’arc traversés après péremption). Ainsi, un premier algorithme de descente pourrait s’exécuter avec comme fonction objectif à minimiser la quantité d’invalidité, puis passer à l’optimisation de la solution valide.

IV. Diversification

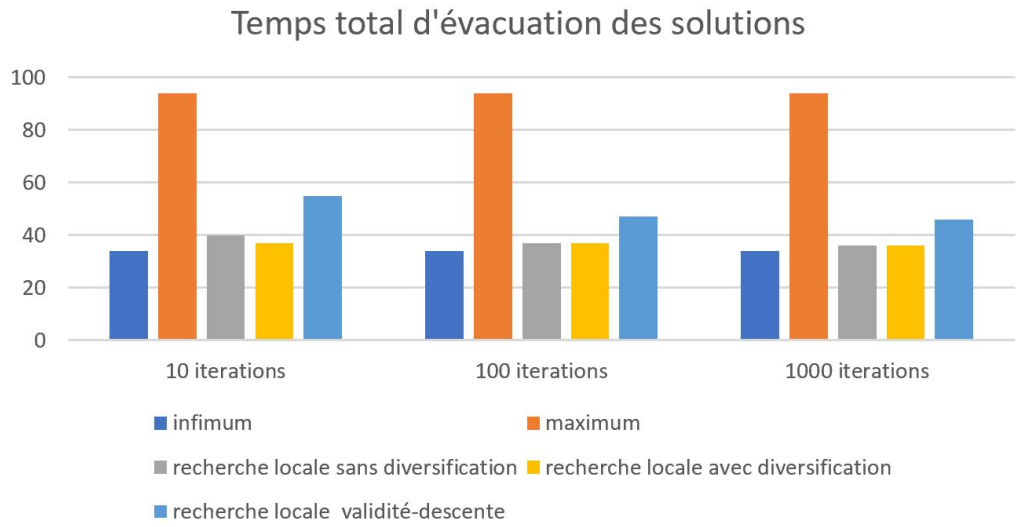
L'implémentation de notre premier algorithme de descente avec des changements aléatoires se rapprochait d'une forme limitée de diversification. De fait, nous avons pu observer par plusieurs fois que notre algorithme de descente aléatoire était plus efficace que les descentes orientés.

Afin de diversifier un peu plus nos voisinages, nous avons fait une copie de *recherche_local_sans_diversification* appelée *recherche_local_avec_diversification* où nous avons rendu aléatoire les limites de modifications des caractéristiques des solutions. C'est-à-dire que plutôt que de donner des *delta_rate* et *delta_date* fixé à notre algorithme de génération de voisinage, nous avons rendu ce paramètre aléatoire. Cet algorithme s'est montré plus efficace encore que le premier.

V. Discussion sur les résultats des différentes méthodes

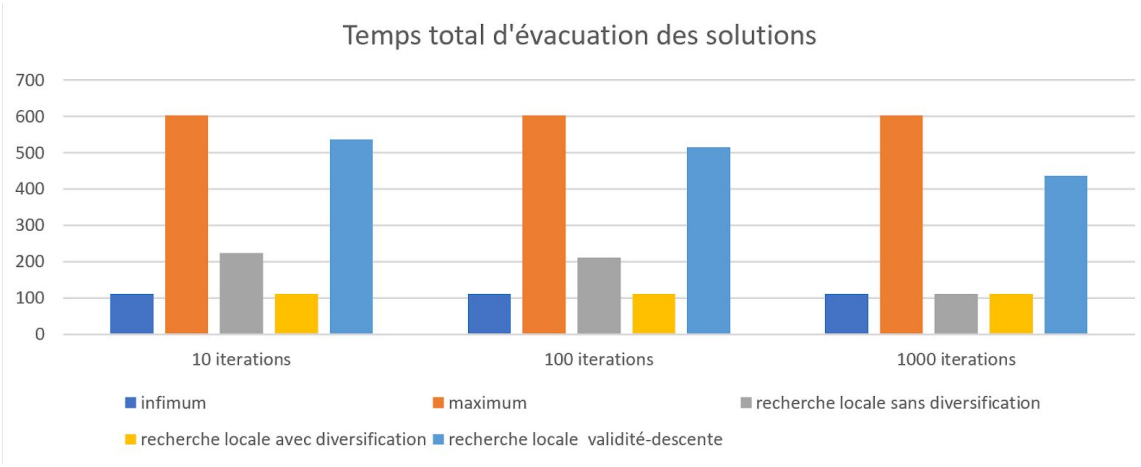
Nous avons testé notre checker à l'aide de 10 solutions fournies par nos camarades sur les différentes instances du projet. Il renvoyait systématiquement les mêmes résultats que le checker en ligne, que ça soit pour la validité ou la raison d'invalidité.

Nous avons testé nos différentes méthodes tout d'abord sur l'exemple simple du sujet. Le graphe suivant correspond à la moyenne de 100 résultats, avec 10, 100 et puis 1000 itérations dans les différentes fonctions de recherche locale. Nous avons également inclu la borne inférieure et supérieure afin d'avoir une idée de l'échelle. Nous avons fixé 50 comme taille de voisinage. Dans cet exemple toutes les méthodes ont trouvés une solution valide.



Nous voyons donc, dans cet exemple,que l'algorithme le plus efficace pour trouver la solution optimale est celui de recherche locale aléatoire avec diversification. Celui sans diversification le précède de peu. Cependant, même si l'algorithme de validité-optimalité trouve une solution valide, il n'en trouve pas une aussi performante que les autres.

Nous avons ensuite testé nos algorithmes sur l'exemple sparse_10_30_3_1_I avec les mêmes paramètres que précédemment. Cependant, aucune méthode n'a trouvé de solution valide.



On constate ici qu'en l'absence de solution valide trouvée, les deux recherches aléatoires converge vers l'infimum. En partant de la borne supérieure la recherche de validité-descente baisse très lentement.

Conclusion

Nous avons mobilisé, au cours de ce projet, beaucoup de ressources fournies par le domaine de la métaheuristique. En effet, en traduisant le problème initial en un problème d'optimisation, nous avons pu tenter de le résoudre par les algorithmes de recherches locales bien connus de ce domaine.

Malgré les difficultés rencontrées lors de sa réalisation, nous avons implémenté les calculs de bornes ainsi que les algorithmes d'intensification et de diversification, tout en comparant l'efficacité d'approches bien distinctes.

Les compétences acquises lors de ce module d'apprentissage nous permettront d'aborder les problèmes semblant irrésolubles avec un nouvel oeil et de nouveau outils de compréhension.