RAPPORT DE PROJET - METAHEURISTIQUES

Planification d'évacuations

Julien Ferry

Quentin Genoud

4IR-A

Lien vers le repository Git : https://github.com/jujudu121212/Metaheuristique

Introduction

Rappel du problème

Le problème proposé consiste en la mise en place d'un plan d'évacuation dans le cas d'un incendie. Les données consistent en :

- un point "sécurisé".
- une liste de points à évacuer, avec pour chacun de ces points le nombre de personnes s'y trouvant initialement, et le chemin qu'elles devront emprunter pour rejoindre le point sécurisé.
- la liste des "arcs", ressources reliant deux points de la carte, avec pour chaque arc une date limite d'évacuation, un temps de parcours et une capacité maximale.

Une solution à ce problème consiste alors, pour chaque point à évacuer, à donner :

- sa date de début d'évacuation.
- son débit d'évacuation (c'est à dire le nombre de personnes qui vont quitter ce noeud, par unité de temps, au cours de son évacuation).

Objectif du projet

Le problème proposé ici est en fait un problème d'optimisation, dont la fonction objectif est, pour une solution donnée, la date de fin d'évacuation pour cette solution (c'est à dire la durée totale de l'évacuation). Il s'agit alors de minimiser cette durée, tout en respectant (si c'est possible) les différentes contraintes imposées.

Après avoir mis en place les fonctions nécessaires, nous utiliserons une méthode de recherche locale pour tenter de générer des solutions valides minimisant autant que possible (il ne s'agit en effet pas d'une méthode exacte) la valeur de la fonction objectif.

Vérification des solutions

Une fois les méthodes permettant de lire un jeu de données, et de lire/écrire une solution implémentées, nous avons écrit un programme vérifiant la validité d'une solution par rapport à un jeu de données : le **Checker**.

Algorithme du Checker

L'algorithme fonctionne en 2 temps :

- On va d'abord simuler l'évacuation en calculant, pour la solution proposée, la capacité restante sur chaque arc et à chaque instant. En ce faisant, on vérifie le respect des contraintes de capacité de chaque arc (ou ressource) et la réalisation de la fonction objectif.
- Dans un second temps, en utilisant la matrice remplie précédemment, on vérifie le respect des due date de chaque arc, en vérifiant que chaque arc ne soit plus utilisé après sa due date.

Le pseudo-code de l'algorithme de notre checker est présenté ci-après.
On notera les éléments suivants :

- `s.objectiveValue` : valeur de la fonction objectif (ie. temps total d'évacuation) pour la solution s
- `d.capacity(e)` est une fonction renvoyant la capacité de l'arc e dans l'instance de données d
- `d.length(e)` est une fonction renvoyant le temps de parcours de l'arc e dans l'instance de données d
- `d.PeopleToBeEvacuated(n)` est une fonction renvoyant le nombre de personnes à évacuer depuis le sommet n dans l'instance de données d
- `d.evacuationPath(n)` est une fonction renvoyant la liste (dans l'ordre de parcours) des arcs du chemin d'évacuation du noeud n dans l'instance de données d
- `s.startEvacuationDate(n)` est une fonction renvoyant la date de début d'évacuation du noeud n dans la solution s
- `s.evacuationRate(n)` est une fonction renvoyant le débit d'évacuation du noeud n dans la solution s
- `edgesData[t][e]` représente la capacité restante pour l'arc (la ressource) e au temps t.

L'accès à chacun de ces éléments se fait en temps constant grâce à l'utilisation, dans notre implémentation, de structures de données adaptées (tableaux ou HashMap).

Algorithm 1 Checker

Global Variable: edgesData: matrix[s.objectiveValue][E]
Input: d : Data, s : Solution
Output: boolean representing the correctness of s for dataset d

```
0:  function CHECK(d,s)
1:  for int t in 0..s.objectiveValue do
2:    for edge e in 0..E do
3:      edgesData[t][e] ⇐ d.capacity(e)
4:    end for
5:  end for
6:  for all node aNode to be evacuated do
7:    tInit ⇐ s.startEvacuationDate(aNode)
8:    remainPeople ⇐ d.PeopleToBeEvacuated(aNode)
9:    while remainPeople > 0 do
10:     t ⇐ tInit
11:     for all edge e ∈ d.evacuationPath(aNode) do
12:       if t > s.objectiveValue then
13:         return false
14:       end if
15:       edgesData[t][e] ⇐ edgesData[t][e] − s.evacuationRate(aNode)
16:       if edgesData[t][e] < 0 then
17:         return false
18:       end if
19:       t ⇐ t + d.length(e)
20:     end for
21:     remainPeople ⇐ remainPeople − s.evacuationRate(aNode)
22:     tInit ⇐ tInit + 1
23:   end while
24: end for
25: for edge e in 0..E do
26:   for t in dueDate(e)..s.objectiveValue do
27:     if edgesData[t][e] ≠ d.capacity(e) then
28:       return false
29:     end if
30:   end for
31: end for
32: return false
```

Analyse de complexité

Voici les **notations** utilisées :

- **T** = nombre d'unités de temps de la fonction objectif
- **E** = nombre d'arcs dans le graphe des routes d'évacuations
- **L** = longueur maximale d'un chemin d'évacuation (dans le pire cas, c'est à dire celui où chaque chemin d'évacuation utilise (presque) chaque arc, **L** se rapproche de **E**)
- **N** = nombre de sommets à évacuer
- **P** = nombre maximal de paquets de personnes devant quitter un sommet

Complexité des différentes étapes de notre algorithme :

- Initialisation (lignes 1 à 5) : `O(T*E)`
  On remplit la matrice des capacités disponibles pour chaque unité de temps, pour chaque arc à sa valeur de capacité.

- Vérification de la contrainte de capacité (lignes 6 à 24) : `O(N*P*L)`
  On simule l'évacuation. Pour chaque noeud à évacuer, on simule le trajet de chaque paquet le long du chemin d'évacuation correspondant et on met à jour les capacités disponibles pour les arcs traversés aux unités de temps adéquates.
  Remarque : Si on voulait prendre en compte le fait que le dernier paquet de personnes à évacuer un noeud peut comprendre un nombre de personne inférieur au débit initial, on remplacerait la ligne 15 `edgesData[t][e] <= edgesData[t][e] - s.evacuationRate(aNode)` *par* `edgesData[t][e] <= edgesData[t][e] - remainPeople` *dans le cas où* `remainPeople < s.evacuationRate(aNode)` *. Ceci pourrait permettre d'obtenir de meilleures valeurs de la fonction objectif dans certains cas. Néanmoins, le sujet spécifie que le débit d'évacuation reste constant quoi qu'il arrive, par conséquent notre algorithme respecte cette contrainte.*

- Vérification de la contrainte des Due Dates (lignes 25 à 31) : `O(T*E)`
  On vérifie qu'aucun arc n'est utilisé après sa due date. Bien que la complexité soit la même que pour la phase d'initialisation, on ne parcourt en réalité pas l'intégralité de la matrice, puisque pour chaque arc, on ne s'intéresse qu'aux unités de temps situées après sa due date.

Complexité totale: `O((N*P*L) + (T*E))`

Détection d'erreurs

- ligne 13 : Cas où la solution ne permet pas d'évacuer tout le monde en respectant la valeur voulue de la fonction objectif.
- ligne 18 : Cas où la solution ne respecte pas les contraintes de capacité de chaque arc (ou ressource).
- ligne 28 : Cas où la solution ne respecte pas les due date : des gens entrent encore sur un arc après sa date d'expiration.

Note sur le format de retour du Checker : En réalité, notre implémentation du Checker retourne une structure contenant des informations utiles pour le procédé de recherche locale (par exemple si la contrainte de capacité est violée sur un arc, il indique quel est cet arc et quels noeuds d'évacuation l'utilisent à l'instant critique). Un booléen associé à chaque instance du Checker permet d'indiquer à ce dernier s'il doit remplir cette structure ou se contenter d'indiquer si la solution est correcte ou non.

Calculs de borne inférieure et supérieure

1) Borne inférieure

Le problème à résoudre est un problème de minimisation, donc la borne inférieure est une valeur de la fonction objectif telle que la "vraie" valeur ne lui sera jamais inférieure (i.e. une solution réalisable ne sera jamais meilleure).
La borne inférieure que nous avons retenue est la suivante : c'est le maximum des temps d'évacuation des points à évacuer. On considère donc que les secteurs peuvent tous être évacués simultanément à leur taux d'évacuation maximum, le temps total est donc la durée de l'évacuation la plus longue, c'est le meilleur des cas.

NB : On pourrait générer la solution correspondante et faire tourner notre Checker dessus (ce dernier pouvant, si on l'appelle d'une certaine manière, déterminer lui-même la valeur de la fonction objectif), mais la complexité algorithmique serait plus importante. Ce sont uniquement les valeurs qui nous intéressent ici, et c'est donc uniquement elles que nous allons calculer.

On notera les éléments suivants :

- `d.capacity(e)` est une fonction renvoyant la capacité de l'arc e dans l'instance de données d
- `d.length(e)` est une fonction renvoyant le temps de parcours de l'arc e dans l'instance de données d
- `d.PeopleToBeEvacuated(n)` est une fonction renvoyant le nombre de personnes à évacuer depuis le sommet n dans l'instance de données d
- `d.evacuationPath(n)` est une fonction renvoyant la liste (dans l'ordre de parcours) des arcs du chemin d'évacuation du noeud n dans l'instance de données d
- `min(L)` retourne le plus petit des éléments de la liste d'entiers non triée L.
- `add(L,k)` ajoute l'élément k à la liste L

Algorithm 2 Lower Bound calculation (function computeInfValue)

Input: d : Data
Output: Integer representing an upper bound value for the evacuation time of the instance d

```
0:  function computeEvacTimes(d)
1:  evacTimesList : List < Integer >
2:  for all node aNode to be evacuated do
3:     Integer minCapa ⇐ +∞
4:     Integer travelTime ⇐ 0
5:     for all edge e ∈ d.evacuationPath(aNode) do
6:        if d.capacity(e) < minCapa then
7:           minCapa ⇐ d.capacity(e)
8:        end if
9:        travelTime ⇐ travelTime + d.length(e)
10:    end for
11:    Integer nbPackets ⇐ ⌈d.PeopleToBeEvacuated(aNode) ÷ minCapa⌉
12:    add(evacTimesList,(nbPackets + travelTime))
13: end for
14: return evacTimesList
14:    function computeInfValue(d)
15: return min(computeEvacTimes(d))
```

Voici les **notations** utilisées :

- **N** = nombre de sommets à évacuer
- **L** = longueur maximale d'un chemin d'évacuation (dans le pire cas, c'est à dire celui où chaque chemin d'évacuation utilise (presque) chaque arc, **L** se rapproche de **E**)

Analyse de complexité :

- Calcul des durées d'évacuation pour chaque sommet à évacuer : *fonction computeEvacTimes(Data d) (lignes 0 à 14)*
  Pour chaque sommet à évacuer, on parcourt le chemin d'évacuation qui lui est associé. La complexité est donc `O(N*L)`
- Recherche du minimum des éléments d'une liste de `N` entiers non triés : `O(N)`

Complexité totale: `O(N*L)`

Les valeurs calculées pour certaines instances sont présentées dans notre tableau récapitulatif en fin de rapport.

2) Borne supérieure

Une borne supérieure est une valeur de la fonction objectif telle que la valeur optimale de la fonction objectif ne lui sera pas supérieure (i.e. la solution idéale sera forcément meilleure) et la solution générée doit être réalisable.
Notre borne supérieure est la somme des durées d'évacuation de chaque noeud à évacuer individuellement. On considère que les secteurs sont évacués chacun à leur tour et qu'un secteur ne peut pas commencer à évacuer tant que le secteur en cours d'évacuation n'a pas entièrement fini d'évacuer tous ses occupants jusqu'au sommet sécurisé.
Cette méthode génère donc bien une borne supérieure, dont il est facile de prouver la validité, mais dont la valeur de la fonction objectif est assez grossière. Une méthode plus fine pourrait par exemple permettre de paralléliser les évacuations dont les chemins n'ont aucun arc en commun. Il est important de noter que cette solution ne tient pas compte des contraintes de due date (qu'elle ne respecte par ailleurs pas dans le cas général).

NB : On pourrait générer la solution correspondante et faire tourner notre Checker dessus (ce dernier pouvant, si on l'appelle d'une certaine manière, déterminer lui-même la valeur de la fonction objectif), mais la complexité algorithmique serait plus importante. Ce sont uniquement les valeurs qui nous intéressent ici, et c'est donc uniquement elles que nous allons calculer.

On notera les éléments suivants :

- `d.capacity(e)` est une fonction renvoyant la capacité de l'arc e dans l'instance de données d
- `d.length(e)` est une fonction renvoyant le temps de parcours de l'arc e dans l'instance de données d
- `d.PeopleToBeEvacuated(n)` est une fonction renvoyant le nombre de personnes à évacuer depuis le sommet n dans l'instance de données d
- `d.evacuationPath(n)` est une fonction renvoyant la liste (dans l'ordre de parcours) des arcs du chemin d'évacuation du noeud n dans l'instance de données d
- `sum(L)` retourne la somme des éléments de la liste d'entiers L.
- `add(L,k)` ajoute l'élément k à la liste L

Algorithm 3 Upper Bound calculation (function computeSupValue)

Input: d : Data
Output: Integer representing an upper bound value for the evacuation time of the instance d

```
0:  function computeEvacTimes(d)
1:  evacTimesList : List < Integer >
2:  for all node aNode to be evacuated do
3:     Integer minCapa ⇐ +∞
4:     Integer travelTime ⇐ 0
5:     for all edge e ∈ d.evacuationPath(aNode) do
6:        if d.capacity(e) < minCapa then
7:           minCapa ⇐ d.capacity(e)
8:        end if
9:        travelTime ⇐ travelTime + d.length(e)
10:    end for
11:    Integer nbPackets ⇐ ⌈d.PeopleToBeEvacuated(aNode) ÷ minCapa⌉
12:    add(evacTimesList,(nbPackets + travelTime))
13: end for
14: return evacTimesList
14:    function computeSupValue(d)
15: return sum(computeEvacTimes(d))
```

Voici les **notations** utilisées :

- **N** = nombre de sommets à évacuer
- **L** = longueur maximale d'un chemin d'évacuation (dans le pire cas, c'est à dire celui où chaque chemin d'évacuation utilise (presque) chaque arc, **L** se rapproche de **E**)

Analyse de complexité :

- Calcul des durées d'évacuation pour chaque sommet à évacuer : *fonction computeEvacTimes(Data d) (lignes 0 à 14)*
  La fonction est la même que pour la calcul de la borne inférieure, la complexité aussi : `O(N*L)`
- Somme des éléments d'une liste de `N` entiers : `O(N)`

Complexité totale: `O(N*L)`

Les valeurs calculées pour certaines instances sont présentées dans notre tableau récapitulatif en fin de rapport.

Intensification

Notre cycle d'intensification se déroule comme ci-dessous :
On part d'une solution de départ valide, puis on effectue les étapes suivantes :
`Compactage -> Réduction des débits -> Compactage -> Augmentation des débits -> Compactage`
A chaque étape, on vérifie la valeur de la fonction objectif des solutions générées grâce à notre Checker, qui nous retourne une structure particulière contenant des informations utiles. Pour tenir compte des due date dans la recherche locale, il faut assigner une valeur particulière à une variable globale (voir section "Configuration de nos programmes").

Présentation des voisinages

Une solution est totalement spécifiée par :
- la date de début d'évacuation de chaque noeud à évacuer
- le débit d'évacuation de chaque noeud à évacuer

Les différentes étapes de notre cycle d'intensification explorent différents voisinages :

Etape du cycle d'intensification	**Solutions "voisines" de la solution courante (valeurs des débits et dates de début d'évacuation comparées à celles de la solution courante)**
Compactage	Solutions ayant des débits d'évacuation identiques, mais une ou plusieurs dates de début d'évacuation inférieures
Augmentation des débits	Solutions ayant des dates de début d'évacuation identiques, mais des débits d'évacuation potentiellement plus élevés
Réduction des débits	Solutions ayant un ou plusieurs débits d'évacuation diminués, et une ou plusieurs dates d'évacuation augmentées

Méthodes d'exploration des voisinages

Cycle de **compactage** : On essaye de faire partir chaque groupe associé à un site de départ le plus tôt possible. Quand il n'est pas possible de diminuer la date de départ de l'un des sites (car on obtient pas une solution valide pour le checker), on essaye de diminuer la date de départ d'un autre site. On répète cela jusqu'à ce que l'on ne puisse plus du tout diminuer la date de départ de tous les sites sans obtenir de solution valide ou quand les dates de départ de chaque site sont t=0.
Note sur la performance : diminuer les dates d'évacuation unité de temps par unité de temps, et lancer une analyse du checker à chaque étape est bien sûr inconcevable d'un point de vue des performances. Aussi, nous calculons, au début du cycle, un facteur de diminution (égal initialement à la moitié de la valeur de la fonction objectif de la solution courante). Nous tentons de diminuer les dates d'évacuation de cette valeur. Lorsque ce n'est plus possible, nous divisons cette valeur par 2 et répétons le procédé jusqu'à ce qu'aucune solution ne soit possible et que le facteur de diminution soit égal à 1.

Cycle de **réduction des débits** : On diminue la taille des débits (paquets de personnes) de certains sites de départ. Pour ce faire, on détermine les tronçons limitants puis on détermine les secteurs qui entrent en conflit sur ces tronçons. On diminue ensuite tour à tour le débit de chaque secteur en conflit individuellement (on obtient plusieurs solutions différentes). On fait aussi une solution où l'on diminue le débit de manière équitable entre tous les secteurs en conflit. Si l'une des solutions obtenues est invalide, on l'ignore. On a donc au maximun autant de solutions qu'il y avait de secteurs en conflits plus la solution où l'on répartit la diminution du débit sur tous les secteurs équitablement. On effectue un nouveau cycle de compression et on garde la meilleure de toutes les solutions (si il y en a plusieurs, on garde la première générée).

Remaque : réduire le débit d'évacuation entraîne une augmentation de la durée d'évacuation pour le noeud concerné. C'est pourquoi nous calculons la valeur exacte, en unités de temps, de cette augmentation et nous retardons les évacuations suivantes d'autant.

Cycle **d'augmentation des débits** : On part d'une solution et on essaye d'augmenter les débits sur tous les arcs jusqu'à que les solutions deviennent invalides (capacité des arcs dépassées par exemple). On refait ensuite un cycle de compression car il est possible qu'en ayant augmenté les débits on ait rendu réalisable une diminution des dates de départ pour certains noeuds.

Les valeurs calculées pour certaines instances sont présentées dans notre tableau récapitulatif en fin de rapport.

Analyse des voisinages : taille et complexité de l'exploration

Etape du cycle d'intensification	**Taille du voisinage**	**Complexité de l'exploration**
Compactage	`O(N)` si on ne s'intérresse qu'aux compactages "maximaux" (où `N` est le nombre de sommets à évacuer)	`O(N*log(s)*complexité(Checker))` car pour atteindre les voisins on explore des solutions intermédaires, dans le pire des cas `log(s)` où `s` est la valeur de la fonction objectif initiale NB : Le logarithme vient du fait qu'on divise notre facteur par deux à chaque itération
Augmentation des débits	`O(N)` si on ne s'intérresse qu'aux augmentations "maximales" (où `N` est le nombre de sommets à évacuer)	`O(N*log(k)*complexité(Checker))` car pour atteindre les voisins on explore des solutions intermédaires, dans le pire des cas `log(k)` où `k` est la valeur initiale du facteur d'augmentation NB (1) : Le logarithme vient du fait qu'on divise notre facteur par deux à chaque itération NB (2) : Ici `log(k)` est donc une constante donc on pourrait donner une complexité théorique de `O(N*complexité(Checker))`

Réduction des débits	`O(N)` où `N` est le nombre de sommets à évacuer	`O(N*complexité(Checker))` car : - on lance le Checker une fois pour connaitre l'arc limitant et les `L` noeuds qui l'utilisent au moment où sa capacité est violée - on génère ensuite `L+1` solutions en diminuant tour à tour les débits de chaque noeud d'évacuation empruntant l'arc limitant au moment critique (`L` solutions pour lesquelles on ne diminue le débit que pour un seul noeud d'évacuation et une autre pour laquelle on partage la diminution entre tous les `L` noeuds concernés) et on lance le Checker sur chacune d'elles. NB : Dans le pire cas `L = N`, d'où la complexité retenue ici

En conclusion, lors de notre cycle d'intensification, la seule opération impossible sur la solution de départ est l'augmentation directe des dates de départ. En particulier, si une évacuation débute à t=0, alors ce sera toujours le cas dans les solutions explorées par notre cycle d'intensification. C'est donc notamment sur cet aspect que nous avons axé notre processus de diversification.

Diversification

Notre cycle d'intensification s'arrête normalement lorsqu'à l'issue d'une itération, aucune meilleure solution n'a pu être générée. Nous avons toutefois intégré un mécanisme de diversification à cette condition : on autorise le programme à itérer un certains nombre de fois supplémentaires (paramétrable) même si la solution n'a pas été améliorée, en utilisant la meilleure solution de l'étape qui n'a pas permis d'amélioration. Ce paramètre permet donc de "ressortir", dans certains cas, d'un minimum local.

Ensuite, le processus de diversification que nous avons choisi d'appliquer est le **multi-start**. Cela consiste tout simplement à choisir plusieurs ordres d'évacuation différents, à tous les explorer et à choisir le meilleur des résultats à la fin. On choisit en fait un ordre d'évacuation des secteurs de départ totalement aléatoire (par exemple, ordre 1 d'évacuation des secteurs : 1->2->3->4, ordre 2 : 3->1->4->2 ...).
On peut choisir le nombre de séquences d'évacuation différentes que l'on veut explorer. Comme les séquences générées sont aléatoires, on peut donc tomber sur 2 ordres identiques mais si le nombre de noeuds de départs différents est important ce scénario devient très rare.
Le problème du multi-start, c'est que l'on applique notre cycle d'intensification sur chaque séquence d'évacuation générée aléatoirement. La durée totale d'exécution peut donc être conséquente pour les instances les plus volumineuses, quand le nombre de points de départ multi-start est élevé. Pour améliorer la vitesse d'exécution de notre algorithme, nous avons décidé de pouvoir utiliser différents threads pour pouvoir paralléliser les calculs des différentes solutions lorsqu'on utilise le multi-start. On peut désactiver cette fonctionnalité simplement avec un paramètre et on peut même choisir le nombre de threads qui tournent en simultané. Lorsqu'un thread a fini son travail et retourné la solution trouvée, le programme principal en lance un nouveau s'il reste des séquences d'évacuation à développer.

Performances

Cette section présente les résultats obtenus pour 3 instances du problème : l'exemple simple du TD, une instance "sparse", et une instance "dense". Pour chacune de ces instances, nous présentons le temps total de calcul, la valeur de la fonction objectif pour la solution obtenue, pour :
- un cycle d'intensification sans diversification : pas de multi-start et on s'arrête dès qu'on atteint un minimum local
- notre recherche locale (diversification et intensification) pour différents nombres de points de départ multi-start

- Toutes les mesures présentées dans le tableau ci-dessous ont été effectuées sur un ordinateur équipé d'un processeur Intel Core i7 3612-QM et de 8 Go de RAM. Notre programme utilise ici, au maximum, quatre threads en simultané pour s'exécuter.
- Les valeurs données dans ce tableau peuvent évidemment varier d'une exécution à l'autre, puisque les séquences d'évacuation à explorer sont générées aléatoirement.

Instance	**Diversification (Oui/Non)**	**Nombre de points de départ (multi-start)**	**Nombre d'arcs du graphe**	**Nombre de noeuds du graphe**	**Borne sup**	**Borne inf (heuristique)**	**Instant de fin d'évacuation**	**Durée totale d'exécution de la recherche locale (en secondes)**
Exemple du TD	Non	N/A	8	7	94	34	37	< 1
Exemple du TD	Oui	5	8	7	94	34	37	< 1
Exemple du TD	Oui	20	8	7	94	34	37	< 1
Exemple du TD	Oui	100	8	7	94	34	37	< 1
dense_10_30_3_1	Non	N/A	930	549	617	91	184	23
dense_10_30_3_1	Oui	5	930	549	617	91	180	118
dense_10_30_3_1	Oui	20	930	549	617	91	180	421
dense_10_30_3_1	Oui	100	930	549	617	91	180	2291
sparse_10_30_3_1	Non	N/A	472	280	602	111	161	12
sparse_10_30_3_1	Oui	5	472	280	602	111	145	26
sparse_10_30_3_1	Oui	20	472	280	602	111	134	88
sparse_10_30_3_1	Oui	100	472	280	602	111	132	540

Analyse des résultats obtenus :

On note logiquement que la durée d'exécution augmente environ linéairement avec la nombre de points de multi-start.
On peut voir que de manière générale, l'augmentation du nombre de points de multi-start conduit à une amélioration de la fonction objectif. Néanmoins, ce n'est pas toujours le cas, ceci étant dû à deux facteurs :
- les ordres d'évacuation sont générés aléatoirement
- lorsque le nombre de points de multi-start devient très grand, il est probable qu'on ne teste plus de nouvelles permutations (cas de l'exemple du TD pour de grands nombres de points de départ)

Configuration de nos programmes

Afin de rendre nos programmes facilement configurables, nous avons, pour plusieurs fichiers, mis en place plusieurs variables globales dont la modification permet de décider le comportement de nos algorithmes. Les variables les plus utiles sont présentées ici.

- `int debug` : présente dans plusieurs de nos fichiers, cette variable peut prendre les valeurs suivantes :
  0 : pas d'affichage dans la console
  1 : affichage minimal dans la console
  2 : affichage de toutes les informations dans la console
  3 : affichage détaillé des informations dans la console
- `Boolean checkDueDate` (fichier `Checker.java`) : indique à l'instance correspondante du Checker s'il doit vérifier la contrainte de due date des arcs ou non.
- `Boolean respDueDates` (fichier `localSearch.java`) : S'il vaut vrai alors les solutions vont être générées en tenant compte de la contrainte de due date. Il est à noter que cette contrainte n'est vérifiée qu'après un cycle de compression, car les solutions que nous utilisons comme solutions initiales ne respectent en général pas ces contraintes. S'il vaut faux alors les solutions générées tiendront uniquement compte des contraintes de capacité.
- `int multiStartNbPoints` (fichier `localSearch.java`) : indique le nombre de séquences aléatoires que le programme va générer. Pour la valeur 0, seule une solution sera générée et intensifiée : il s'agira donc d'intensification "pure".
- `Boolean useMultiThreading` (fichier `localSearch.java`) : indique si les calculs doivent se faire ou non sur plusieurs threads.
- `int nbThreads` (fichier `localSearch.java`): dans le cas où l'utilisateur choisit d'utiliser le multithreading, indique le nombre maximum de threads qui peuvent s'exécuter en parallèle.

Pour utiliser nos programmes simplement :
Le fichier `Test.java` permet de lancer simplement la recherche locale sur une instance. Le calcul des valeurs d'une borne inférieure et d'une borne supérieure est également lancé par ce fichier. Si `Boolean debug` vaut `true`, la recherche locale s'exécute pour l'exemple simple du TD. S'il faut `false`, alors l'instance résolue par la recherche locale est celle dont le nom est contenue dans la variable `String inst`. A noter que les instances doivent être contenues dans le répertoire `InstancesInt/` de notre repository, et que les fichiers générés le seront au format exigé, dans le dossier `Generated_best_solutions` sous le nom `nomInstanceAAAAMMJJHH:mm:ss` (où `AAAAMMJJHH:mm:ss` est la date de fin de l'exécution de notre recherche locale).

Pour compiler :

```
javac *.java
```

Pour lancer la recherche locale :

```
java Test
```

Pour plus de détails concernant l'utilisation de nos programmes, voir le README.

Conclusion

Récapitulatif du travail effectué

La première partie de ce projet a donc consisté à la mise en place des fonctions supports nécessaires (lecture d'un jeu de données, lecture/écriture et vérification de solutions).
Au cours de la seconde partie, nous nous sommes intéressés au calcul de bornes inférieure et supérieure de la valeur de la fonction objectif pour un jeu de données. Nous disposions alors d'un encadrement de cette valeur pour chaque instance souhaitée. Cet encadrement nous permet ensuite de vérifier la cohérence des résultats trouvés par notre mécanisme de recherche locale, qui a constitué la majeure partie du travail du projet.

Nous avons imaginé un cycle d'intensification cohérent, permettant d'atteindre un minimum local, pour une solution initiale donnée (la seule opération impossible au cours de ce cycle étant le retardement direct du démarrage d'une évacuation). L'ajout à ce cycle de la possibilité d'itérations n'améliorant pas la solution a constitué un premier pas dans la diversification, permettant en fait de sortir d'un minimum local. Mais le mécanisme principal de diversification de notre recherche locale est le multi-start, qui nous a permis, dès les premiers tests, d'observer des améliorations significatives des valeurs obtenues pour la fonction objectif, avec l'augmentation du nombre de solutions initiales.

Possibles améliorations

Nous pourrions améliorer plusieurs étapes de notre algorithme. La principale amélioration possible concerne la fonction modifyRates. Cette

fonction, qui prend en argument une solution invalide du point de vue des capacités, trouve la ressource limitante (c'est à dire l'arc sur lequel la contrainte de capacité n'est pas respectée), et génère plusieurs solutions en diminuant le débit d'évacuation des noeuds utilisant la ressource au moment du conflit. Bien que les évacuations suivant celle dont le débit est réduit soient décalées afin de compenser l'allongement de sa durée, cette méthode génère parfois des solutions invalides. En effet, l'allongement de la durée d'évacuation d'un noeud entraîne une utilisation prolongée de toutes les ressources du chemin d'évacuation (et pas seulement de la ressource identifiée plus tôt). Il serait bien sûr possible de ne générer que des solutions faisables, soit par une analyse plus profonde des modifications à apporter, soit en décalant plus fortement les évacuations concourantes. Néanmoins, la première idée entrainerait un allongement conséquent de la durée d'exécution de modifyRates, tandis que la deuxième ralentirait le cycle de compactage suivant chacune des solutions générées. Pour des raisons de performances, nous avons donc choisi de conserver notre méthode, bien qu'elle génère parfois des solutions inexploitables.