Rapport de TP Métaheuristique

Maxence Biers - Loïca Marotte

Juin 2019

1 Introduction

Dans le cadre du cours de Métaheuristique de 4e année IR, nous avons travaillé sur un sujet de TP concernant la planification d'évacuations de personnes dans une situation d'urgence, comme un incendie par exemple. L'objectif était de résoudre un problème d'optimisation combinatoire à l'aide d'heuristiques et de métaheuristiques. Pour cela, nous avons mis en oeuvre les méthodes vues en cours, comme le calcul des bornes inférieure et supérieure, un processus de recherche local et d'intensification et de diversification.

Nous avons choisi de travailler en Java car l'approche orientée objet nous semblait pertinente afin de pouvoir créer des structures de données claires à nos graphes.

Après une présentation de notre structure de données et du vérifieur de solution, nous détaillerons le calcul des bornes inférieure et supérieure, pour enfin envisager l'intensification et la diversification de notre recherche locale.

2 Structure des données et des solutions du problème

Afin de pouvoir vérifier des solutions, nous avons créé un système de données classique pour un graphe, composé d'une classe Graph contenant des Nodes reliés par des Edges. Les nodes desquels doivent partir des personnes à évacuer sont des Nodes_evac, une classe qui étend Node. Ces Node_evac contiennent en plus de leur id : leur population de départ, leur population actuelle, le flux maximal de personnes qui peut en sortir, la capacité maximale que peut accueillir sa route d'évacuation, et la répartition des personnes qui lui sont liées sur toute sa route d'évacuation (c'est-à-dire que le flux de personnes sur un Edge est égal à la somme des flux qui lui sont attribués contenus dans tous les Nodes_evac), en plus de sa route d'évacuation sous la forme d'une liste d'Edges.

Les solutions sont représentées avec une classe Solution contenant les mêmes éléments que les solutions sous forme de texte, c'est à dire un nom, un nombre de Nodes, la validité de la solution, son objectif (le temps de complétion de l'évacuation), son temps de calcul et sa méthode de calcul. De plus, les solutions contiennent une liste de la classe Solution_Node, qui représentent les informations sur l'évacuation de chacun des nodes. Solution_Node contient le node auquel il fait référence, sa date de début d'évacuation et son taux d'évacuation.

Une méthode de Solution, writeToFile() permet d'écrire directement cette solution sous forme de fichier selon le modèle qui nous avait été donné.

3 Vérifieur de solution

Le vérifieur est présenté sous la forme d'une classe Checker, qui contient une unique fonction static "check", demandant comme argument une Solution et un Graphe. Il fonctionne en simulant l'évacuation complète.

Pour cela il parcourt les Node_evac, et fait avancer toutes les personnes qui proviennent de ce node sur leur chemin, qui se présente comme une liste de tableaux. Par exemple, un chemin associé à un Node_evac composé de 2 edges ayant une durée de traversée de 2 puis de 7 sera une liste de 2 tableaux d'int, le premier de 2 cases et le second de 7 cases. Le checker va parcourir ce chemin puis avancer toutes les personnes qui s'y trouvent d'une case.

Au passage, il va également utiliser ces valeurs pour mettre à jour les valeurs de flot qui se trouvent dans les Edge du graphe. Par exemple, si 2 personnes se trouvent dans la 3ème case du 1er tableau, l'algorithme va rajouter ces 2 personnes dans la 3ème case du tableau de flots du 1er Edge de la route d'évacuation du Node_evac. De cette manière, les Edge contiennent toujours la somme des flots qui le traversent.

Il suffit ainsi de s'assurer qu'aucun flot ne dépasse la capacité de son Edge. Pour cela, on ne va vérifier à chaque étape que la première case de chaque Edge, car si le flot total passant à l'entrée de l'Edge est valide, il le sera tout au long de l'Edge, vérifier tout son long serait donc inutile.

Une fois tous les Node_Evac et leur chemins vides, on considère la vérification terminée, et si l'algorithme a atteint ce point, la Solution est valide.

Algorithm 1 check(solution,graphe)

```
Ensure: Vérifie qu'une solution est valide pour un graphe donné
  fini ← false
  while !fini do
    fini ← true
    resetFlow() {permet de s'assurer que les flots des Edges soient vides}
    temps ← tempsCourant
    for Node_solution nodeSol : solution do
      dateNode ← dateSolution
      if dateNode < 0 then
        return false
      end if
      if dateNode < temps and !nodeSol.routeFinie() then
        routeCourante ← nodeSol.getRoute()
        popu ← nodeSol.getPopulation()
        flotMax ← nodeSol.getTauxEvac()
        if popu > 0 then
          if popu > flotMax then
            flotPrec ← flotMax
            solNode.setPopulation(popu − flotMax)
          else
            flotPrec ← popu
            solNode.setPopulation(0)
          end if
        else
          flotPrec ← 0
        end if
        for Arc arc : routeCourante do
          arc.mise_a_jour(flotPrec)
        end for
      end if
      fini ← fini and nodeSol.routeFinie()
    end for
    for Arc arc : graphe do
      if arc.getFlot()[0] > arc.getCapacity() then
        return false
      end if
    end for
    tempsCourant ++
  end while
  resetGraphe() {remet à 0 le graphe pour une future verification de solution}
  return true
```

En considérant que la taille de notre problème est le nombre de Node_evac multiplié par la taille moyenne de leurs routes d'évacuation (car il s'agit des seules données que nous utilisons/parcourons), cet algorithme parcourt les nodes et leur route autant de fois que nécessaire pour terminer. Comme le temps d'évacuation va probablement être lié linéairement à la taille de nos données (car multiplier par 2 la taille des routes par exemple multipliera probablement le nombre de boucles a réaliser par 2). Nous aurions donc un vérificateur en complexité $O(n^2)$, qui fonctionnera bien pour des jeux de données de taille raisonnable. Comme nous verrons plus tard, cela suffit pour notre projet. Aucun parcours de recherche inutile qui aurait pu être remplacé par une 'map'par exemple n'est effectué dans notre algorithme, mais peut-être aurions nous pu trouver une autre méthode de vérification moins gourmande ?

4 Bornes de la solution optimale

4.1 Borne inférieure

4.1.1 Principe

Afin de calculer une borne minimale, nous avons décidé de tout simplement faire partir tous les Node_Evac au temps 0 et les faire évacuer à leur capacité de route (qui est la valeur la plus petite entre leur taux maximal d'évacuation et la plus petite capacité présente sur leur route). Nous sommes ainsi certains qu'il est impossible de faire plus rapide que cette solution, même si celle-ci n'est pratiquement jamais admissible, comme nous le vérifions à l'aide de notre Checker.

Algorithm 2 find_inf(graphe)

Ensure: Retourne la borne inférieure pour un graphe donné
```
temps ← 0
listeSolNode = new ArrayList(Solution_node)
for Node_Evac nodeEvac : graphe do
  listeSolNode.add(new Solution(depart = 0))
  t ← nodeEvac.getTempsParcours()
  if t > temps then
    temps ← t
  end if
end for
solution = new Solution(listeSolNode)
return solution
```

La création de cette borne parcourt les Node_Evac, et est donc de complexité O(n), ce qui garantit sa rapidité. Il est aussi à noter que nous aurions normalement dû parcourir les edges associés à chaque Node_Evac afin de calculer la durée minimale de chaque route, mais nous avons déjà effectué ce calcul à la création du graphe et stocké dans chaque Node_Evac la valeur afin de ne pas avoir à refaire ce calcul durant la création de nos Solutions bornes. Nous ajouterons que dans notre code, nous vérifions systématiquement la validité de ces solutions minimales, ce qui fait que la création totale est en réalité en complexité $O(n^2)$. Le temps d'exécution est si négligeable comparé aux processus d'intensification ou diversification que cela ne pose aucun problème pour la suite.

4.1.2 Résultats

Nous avons lancé cet algorithme de recherche de borne inférieure sur les instances de jeux de données qui nous étaient mis à disposition. Leurs résultats sont résumés dans le tableau suivant. Les tests ont été réalisés pour des graphes ayant 10 noeuds à évacuer, mais des densités de noeuds différentes.

Instance	Sommets	Fonction objectif	Validité	Temps de calcul (en ms)
medium_10_0_3_2.full	428	2369	non	2
medium_10_0_3_3.full	428	2215	non	3
medium_10_0_3_4.full	428	2077	non	4
dense_10_0_3_2.full	549	2323	non	11
dense_10_0_3_3.full	549	1961	non	7
dense_10_0_3_4.full	549	2085	non	3
sparse_10_0_3_2.full	280	2049	oui	110
sparse_10_0_3_3.full	280	2104	non	3
sparse_10_0_3_4.full	280	2291	oui	215

FIGURE 1 – Tableaux récapitulatif des resultats obtenus avec l'algorithme de recherche de borne inférieure

Cette heuristique est très rapide, de l'ordre de quelques millisecondes, et donne les résultats attendus. On constate que les solutions qui sont valides sont beaucoup plus lentes en terme de calcul puisque la fonction Checker prends plus de temps de calcul que la fonction de recherche de borne inférieure, et que le fait qu'elles soient valides implique que la vérification se fait jusqu'à l'évacuation complète.

Si il est techniquement possible qu'une borne minimale soit valide, il est étrange d'obtenir autant de résultats valides. Nous émettons des doutes sur notre vérificateur. Malheureusement, durant sa conception et phase de tests, nous avons eu la malchance de seulement tomber sur des jeux de données qui ont généré des bornes minimales non valides, donc nous ne nous sommes pas rendus compte de problèmes, et c'est uniquement en lançant les algorithmes sur l'intégralité des jeux de données une fois le projet quasiment terminé que nous avons observé ces résultats... Nous avons cherché la cause de ce problème mais n'avons pas pu la trouver. Nous aurions pu enlever les cas positifs de nos tableaux mais cela n'apportait rien.

Il est possible de critiquer la pertinence d'une telle borne, qui est peut-être trop basse et évidente pour être exploitable dans nos futures expériences ? Il nous est en revanche compliqué d'imaginer une heuristique différente garantissant de donner une borne inférieure.

4.2 Borne supérieure

4.2.1 Principe

La borne maximale est calculée de manière très simple : on considère que les Nod_evac commencent leur évacuation une fois que le Node_evac précédent a fini la sienne. Pour cela, il suffit de parcourir la liste de Node_evac, et incrémenter à chaque fois une variable "temps" en utilisant leur attribut représentant le temps d'évacuation si cette évacuation se déroule à la capacité maximale dans leur route (que nous avons déjà utilisé pour la borne minimale). Cet algorithme nous garantit la création d'une solution viable (que nous vérifions avec notre Checker), mais bien supérieure à la solution optimale.

Algorithm 3 find_sup(graphe)

Ensure: Retourne la borne supérieure pour un graphe donné
  $temps \leftarrow 0$
  $listeSolNode$ = new ArrayList(Solution_node)
  for $Node\_EvacnodeEvac : graphe$ **do**
    listeSolNode.add(new Solution($depart = temps$))
    $t \leftarrow nodeEvac.getTempsParcourt()$
    $temps \leftarrow temps + t$
  end for
  $solution$ = new Solution(listeSolNode)
  return solution

4.2.2 Résultats

Les tests ont été réalisés pour des graphes ayant 10 noeuds à évacuer, mais des densités de noeuds différentes.

Instance	Sommets	Fonction objectif	Validité	Temps de calcul (en ms)
medium_10_0_3_2.full	428	12844	oui	237
medium_10_0_3_3.full	428	16096	oui	339
medium_10_0_3_4.full	428	18540	oui	462
dense_10_0_3_2.full	549	13680	oui	286
dense_10_0_3_3.full	549	15189	oui	331
dense_10_0_3_4.full	549	16739	oui	396
sparse_10_0_3_2.full	280	12705	oui	220
sparse_10_0_3_3.full	280	16454	oui	359
sparse_10_0_3_4.full	280	17875	oui	449

FIGURE 2 – Tableaux récapitulatif des resultats obtenus avec l'algorithme de recherche de borne supérieure

Tout comme l'heuristique de la borne minimale, nous pouvons contester l'intérêt d'une telle heuristique, qui est peut-être un peu évidente et trop élevée par rapport aux solutions optimales. Son temps d'éxécution est plus élevé que celui de la borne inférieure, mais les solutions sont toutes valides, le temps d'execution du Checker augmente donc ce chiffre.

5 Recherche locale

5.1 Intensification

5.1.1 Principe

Afin d'améliorer les résultats obtenu par la solution obtenue avec borne supérieure, nous avons mis en place un système de recherche. La première étape consiste à appliquer un processus d'intensification s'appuyant sur la méthode de descente.

Notre méthode part de la solution retournée par l'algorithme de borne supérieure, qui évacue chaque node entièrement, les uns à la suite des autres. Cet algorithme d'intensification va conserver l'ordre d'évacuation donné, mais tenter de faire se chevaucher les évacuations.

Pour cela, il va tenter d'avancer le départ de chaque node au maximum, tout en conservant une solution valide. Il va donc parcourir tous les Node_Evac de la solution de borne supérieure. Pour chaque Node_evac, il avance son départ selon un pas fixé, et avance en conséquence les Node_Evac suivant. Il va tester si cette solution est valide à l'aide du vérifieur, et répéter ce processus jusqu'à ce que la solution ne soit plus valide. Il va proceder ainsi pour chaque Node_Evac et décaler autant que possible chaque départ. Il faut en revanche faire attention à ce que la date de départ du premier Node_Evac ne soit pas avancée, étant donnée que celle-ci est déjà à 0.

Algorithm 4 intensification(graphe,solution,pas)

Ensure: Retourne la solution de recherche locale avec intensification

```
listeSolution ← solution.getListeSolution()[1..solution.getListeSolution().length]
for Solution_NodesolNode : listeSolution do
    cond ← true
    solPrec ← solNode
    while cond do
        solTest ← solPrec.decalage(pas)
        if !solTest.isValide() then
            cond ← false
        else
            solPrec ← solTest
        end if
    end while
    solution ← solPrec
end for
return solution
```

La complexité de cet algorithme devrait être en O(n^2)*'complexité de notre vérificateur' puisque pour chaque node, il doit parcourir toute la donnée pour effectuer son décalage (O(n)), vérifier la solution (complexité de notre vérificateur), puis faire cela autant de fois autant que nécessaire, ce qui sera linéaire avec la taille de la donnée (O(n)). Mais nous avons choisi un pas de décalage dépendant de la taille de la donnée (pas = tempsTraverseeMoyenEdge * tailleMoyenneRoute /10), donnant une complexité totale en O(n^3), le nombre de nodes à évacuer étant constant dans nos jeux de données.

Il est a noter que cette algorithme d'intensification ne gère pas les taux d'évacuation. En effet, nous avions mal compris le problème et pensions que le taux d'évacuation était fixe, égal à la plus petite valeur entre le taux d'évacuation maximal d'un node et la capacité de sa route d'évacuation. Nous nous sommes rendus compte de cette erreur tard dans le projet, et avons essayé de la corriger. L'implémentation de notre système de graphe devrait être capable de supporter cette version plus complète du problème après correction, mais nous avons été pris par le temps et n'avons pas pu implémenter une méthode différente d'intensification. Une idée que nous avons eu était de combiner notre algorithme avec un algorithme réduisant le taux d'évacuation d'un node au hasard, puis essayant d'avancer la date de départ de tous les nodes suivants avant d'essayer de réaugmenter le taux d'évacuation de ce node autant que possible. C'est dommage, mais cette méthode d'intensification reste valable pour le problème et nous permettra d'obtenir des résultats.

5.1.2 Résultats

Les tests ont été réalisés pour des graphes ayant 10 noeuds à évacuer, mais des densités de noeuds différentes.

Instance	Sommets	Fonction objectif	Validité	Temps de calcul (en s)
medium_10_0_3_2.full	428	7416	oui	10,210
medium_10_0_3_3.full	428	6688	oui	19,576
medium_10_0_3_4.full	428	13490	oui	11,305
dense_10_0_3_2.full	549	12736	oui	2,412
dense_10_0_3_3.full	549	8377	oui	15,817
dense_10_0_3_4.full	549	10220	oui	15,192
sparse_10_0_3_2.full	280	6525	oui	12,107
sparse_10_0_3_3.full	280	7070	oui	22,344
sparse_10_0_3_4.full	280	4855	oui	22,860

FIGURE 3 – Tableaux récapitulatif des resultats obtenus avec l'algorithme de recherche de borne inférieure

On constate que cette méthode donne de meilleurs résultats en terme de fonction objectifs, mais le temps de calcul a considérablement augmenté. En effet, il est passé de quelques millisecondes à une dizaine ou vingtaine de secondes. Cela peut se justifier par sa complexité beaucoup plus importante.

5.2 Diversification

5.2.1 Principe

Afin de diversifier notre recherche, nous avons choisi une approche multi-start. Pour cela nous allons réutiliser notre algorithme de recherche de borne supérieure, mais cette fois ci en changeant l'ordre des Node_Evac qu'il va utiliser. Cela a pour effet de toujours faire les évacuations les unes après les autres, mais dans un ordre différent. Le temps total d'évacuation est donc égal pour toutes ces bornes supérieures générées, mais ces solutions de base données à notre algorithme d'intensification changent drastiquement les résultats.

L'ordre des Node_Evac dans la borne superieure est généré aléatoirement grâce à la méthode find_sup_random. Nous générons plusieurs de ces bornes, les donnons à notre algorithme d'intensification, puis conservons uniquement la solution de sortie possédant l'objectif le plus bas.

Algorithm 5 diversification(graphe, nbIterations,pas)

Ensure: Retourne la solution de recherche locale avec intensification et diversification
  $solution \leftarrow null$
  $iteration \leftarrow nbIteration$
  while $iteration > 0$ **do**
    $solutionSup \leftarrow find\_sup\_random(graphe)$
    $solutionIntens \leftarrow intensification(graphe, solutionSup, pas)$
    if $solution = null$ **or** $solutionIntens.getObjectif() < solution.getObjectif()$ **then**
      $solution \leftarrow solutionIntens$
    end if
    $iteration$ - -
  end while
  return solution

5.2.2 Résultats

Les tests ont été réalisés pour des graphes ayant 10 noeuds à évacuer, mais des densités de noeuds différentes. On a choisit de réaliser 10 itérations de multi-start.

Instance	Sommets	Fonction objectif	Validité	Temps de calcul (en s)
medium_10_0_3_2.full	428	2460	oui	90,898
medium_10_0_3_3.full	428	8305	oui	96,032
medium_10_0_3_4.full	428	9248	oui	143,937
dense_10_0_3_2.full	549	6954	oui	69,991
dense_10_0_3_3.full	549	7853	oui	80,257
dense_10_0_3_4.full	549	5609	oui	122,077
sparse_10_0_3_2.full	280	2405	oui	105,464
sparse_10_0_3_3.full	280	9278	oui	112,637
sparse_10_0_3_4.full	280	2468	oui	175,207

On constate que les resultats de la fonction objectif sont considérablement améliorés, on se rapproche de la borne inférieure. Cependant le temps de calcul a encore augmenté, il est passé à l'ordre de quelques minutes pour chaque graphe.

6 Discussion

6.1 Qualité des résultats

Nous avons rassemblé les valeurs objectif des differentes solution dans un graphique pour visualiser l'amélioration au travers des étapes. Les jeux de données n'ont pas été triés de manière particulière

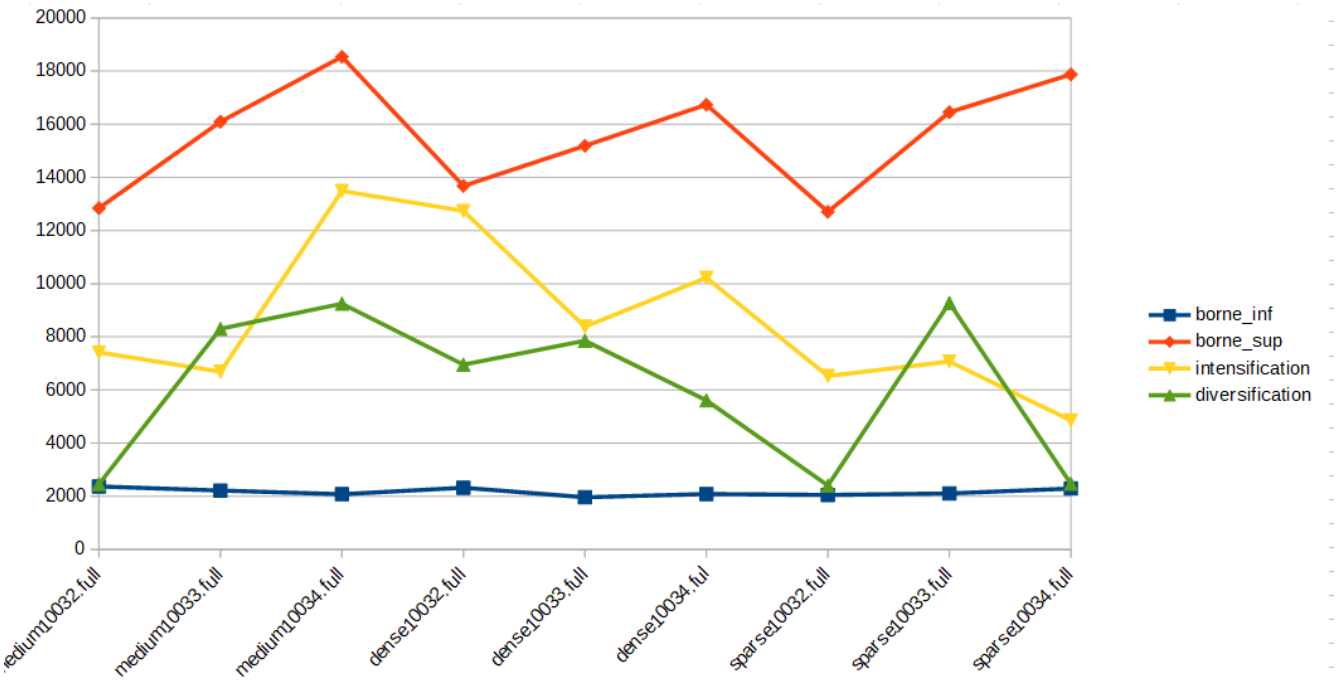

FIGURE 4 – Graphique présentant les valeurs objectif des solutions obtenues par les differentes techniques sur different jeux de données

On note que, comme attendu, nos résultats d'intensification et de diversification se situent entre la borne superieure et la borne inferieure. Des fois, les résultats de la diversification sont très proches de la borne minimale, ce qui signifie que nous avons trouvé une très bonne solution.

De plus, certaines fois l'intensification seule obtient de meilleurs résultats que la diversification. Ceci est dû au fait que notre multi-start ne se fait pas sur assez de solutions de départ et par malchance nous ne trouvons pas de meilleure solution que celle produite avec la solution de base servant à notre processus d'intensification.

7 Conclusion

Ainsi, nous avons réalisé une structure de données correspondant au problème combinatoire que représente la planifiacation d'évacuation . Nous avons ensuite pu obtenir une solution grâce à une heuristique, que nous avons ensuite optimisé par des principes d'intensification et de diversification. Ces TP nous ont permis de mettre en pratique les notions abordées en cours sur un cas concret, et de nous rendre compte de leurs résultats ainsi que de leur efficacité.