Antoine Balieu
Quentin Colras
Valérian Bal

Master 2 TNSID

Compte Rendu
Travaux pratiques

Heuristiques et Métaheuristiques

12/01/2022

Sommaire

I. Introduction

1.1 Introduction générale

Dans le cadre de ce TP d'heuristiques et de métaheuristiques, l'objectif est de développer des algorithmes pour maximiser l'occupation des sièges d'une salle tout en respectant les distances minimales de sécurité afin de minimiser les risques d'infections pour les étudiants et les enseignants. Nous développerons ainsi un algorithme naïf avec la méthode gloutonne, une amélioration de l'algorithme précédent en utilisant la recherche locale et un algorithme utilisant les heuristiques afin de nous donner une solution presque optimale de l'occupation des sièges de la salle.

Le tout sera développé en utilisant le langage de programmation Java permettant de concevoir le problème de manière orientée objet.

1.2 Présentation du problème

Pour ce problème d'allocation maximale des sièges, nous avons besoin d'une salle, celle-ci est décomposée en trois secteurs dans l'énoncé du sujet, chaque secteur possède un nombre de sièges, un nombre de sièges par ligne et un nombre de colonne, ses propriétés peuvent varier selon les secteurs.

La distance entre les sièges voisins d'une même rangée de 50 cm, et la distance entre les séries de rangées voisines dans le même secteur est de 75 cm tandis que la distance entre secteur est de 120 cm. Dans notre cas, la distance minimale peut valoir les distances suivantes : 100 cm, 120 cm, 140 cm, 160 cm, 180 cm et 200 cm.

Pour finir, nous calculerons la distance entre les sièges en utilisant la distance euclidienne qui vaut : $\sqrt{(xi - xj)}\ 2 + (yi - yj)^2$ pour une distance entre le siège i et le siège j.

II. Analyse du programme

2.1 Structures de données

Afin de modéliser ce problème sous la forme orientée objet, nous avons besoin de créer des classes.

Nous avons choisis de créer les classes **Seat** et **Room** pour représenter les sièges ainsi que la pièce où sont positionnés les sièges afin de premièrement représenter un siège avec une identifiant, et sa position x et y. Concernant la pièce, c'est pour la flexibilité du programme au cas où l'on aurait besoin de traiter plusieurs pièces et ainsi bien distinguer et traiter les sièges dans leurs pièces respectives et par conséquent mieux structurer le programme et le rendre plus flexible.

Toujours dans un souci de flexibilité, nous avons créé une classe abstraite **Solver**, dont les solveurs seront hérités pour définir chacun leurs propres fonctions *solve()* qui va permettre d'optimiser l'allocation des sièges. Ainsi, nous pouvons définir autant de solveurs que l'on veut et pouvoir en rajouter si besoin dans le futur et obtenir les premiers résultats en changeant le type à l'endroit où l'on fait appelle à la fonction *solve()*.

Pour entrer dans le détail, commençons par la classe Seat, cette dernière possède trois attributs qui sont id (identifiant), x (position en x) et y (position en y) qui sont tous entiers. Cet objet en plus des accesseurs et mutateurs des attributs, nous avons une fonction qui *toString()* qui permet de retourner la chaîne de caractère d'un siège et la fonction *distanceTo(Seat other)* qui va retourner la distance euclidienne du siège par rapport au siège passé en paramètre.

Ensuite, nous avons la classe Room, qui n'a d'autre rôle que de contenir un ensemble de sièges, pour cela nous avons choisis l'objet **TreeSet** qui va permettre de contenir les sièges en prévenant le fait d'avoir un doublon, et permet aussi de trier les objets contenus, ici nous trions les sièges par leurs identifiants dans l'ordre croissant, nous avons fait ce choix par rapport au fichier texte *instances.txt* qui contient les sièges, leurs identifiants et leurs position en x et en y.
La classe possède une fonction *displayRoom()* qui va afficher les sièges de la pièce, cela est utile pour visualiser l'affectation des sièges de la pièce.

Pour continuer sur ce sujet, nous avons une classe LoaderUtils dédiée à la lecture du fichier, qui va retourner la pièce obtenue après l'appel de la fonction *fromFile(String nomFichier)* qui va lire le fichier passé en paramètre. La lecture des fichiers se fait via une classe dédiée pour structurer une fois de plus le projet.

En continuant sur les utilitaires nous avons la classe BenchMark qui possède deux attributs qui sont le nombre d'itérations et le tableau des distances minimales à tester pour essayer d'obtenir une solution optimale. La classe dispose de deux fonctions, une fonction *benchmark(Room, Solver)* qui prend en paramètre une pièce et un solveur, et qui va retourner le temps CPU qu'à mis le solveur à optimiser l'allocation des sièges de la pièce passé elle aussi en paramètre.
Ensuite la deuxième fonction est la fonction *displayBenchmark(timings)* qui prend en paramètre le résultat de la première fonction et va les afficher.

Pour finir, la classe Solver, celle-ci à un attribut entier distance, qui représente la distance minimale à avoir entre deux sièges occupés. Nous avons choisi de ne pas mettre un attribut pièce de type Room afin de bien séparer les parties du programme, une partie utilitaire, une partie donnée et une partie algorithme. Pour chaque solveur, la fonction *solve(Room room)* prend en paramètre la pièce où l'on doit optimiser l'allocation des sièges et va retourner l'ensemble des sièges occupés.

2.2 Analyse du code

Maintenant que nous avons vu comment les données sont structurées, nous pouvons passer au principal du projet, c'est-à-dire les algorithmes d'optimisation.

Premièrement l'algorithme naïf utilisant la méthode gloutonne, il est implémenté dans la classe **GreedySolver** et dont voici le pseudo-code ci-dessous :

Algorithme naif :
Entrée : P, la pièce que l'on veut optimiser l'allocation des sièges
Sortie : S, l'ensemble des sièges occupés

Fonction Solve :
début
déclaration de E (ensemble des sièges occupés)
pour chaque Siege s de la pièce faire
    siegeVoisinOccupé← faux
    pour chaque Siege s2 dans E faire
        si la distance entre s et s2 est inférieur à distanceMinimale alors
            siegeVoisinOccupé ← vrai
            sortir de la boucle pour
        fin si
    fin pour
    si siegeVoisinOccupé = faux alors
        ajouter s dans E
    fin si
fin pour
retourner E
fin fonction

Cet algorithme est le plus simple de tous, on parcourt l'ensemble des sièges de la pièce, pour chaque siège s'il n'y a aucun siège voisin, c'est-à-dire dans la zone de la distance minimale qui n'est occupée alors on l'ajoute dans l'ensemble des sièges occupés.

Deuxièmement, une amélioration de l'algorithme précédent en utilisant la recherche locale, il est implémenté dans la classe LocalSearchSolver, la classe possède deux attributs supplémentaires qui sont le nombre maximum d'itérations et le temps maximale d'exécutions et elle hérite de la class GreedySolver afin de pouvoir obtenir une première solution initiale.

Voici le pseudo-code de l'algorithme de recherche locale ci-dessous :

Algorithme recherche locale:

Entrée : P, la pièce que l'on veut optimiser l'allocation des sièges
Sortie : S, l'ensemble des sièges occupés

Fonction Solve :
début
déclaration de E (ensemble des sièges occupés) ← GreedySolver.solve(P)
iteration ← 0 //Compteur d'itération
tempsMax ← temps actuel + maxTemps //Afin de vérifier que la fonction ne dépasse pas le temps //maximale
tantQue (iteration < maxIteration et tempsActuel < tempsMax) faire
    s1 ← choisir un siege occupé au hasard
    s2 ← choisir un siege libre au hasard
    déclaration de NewE (nouvel ensemble de sièges occupés) ← swap(E, s1, s2)
    si newE.size() > E.size() alors
        E = newE
    fin si
    iteration ← iteration + 1
fin tantQue
retourner E
fin fonction

Entrée : E, l'ensemble des sièges occupés, s1le siège occupé, s2 le siège libre
Sortie : S, le nouvel ensemble des sièges occupés
Fonction Solve :
début
déclaration de NewE (ensemble des sièges occupés) ← E
NewE.ajouter(s2) //On ajoute le siège libre
NewE.retirer(s1) //On retirer le siège occupé
pour chaque Siege s de NewE
    si la distance entre s et s2 est inférieur à distanceMinimale alors
        NewE.retirer(s)
    fin si
fin pour
pour chaque Siege s de la pièce faire
    si s = s1 alors passer
    sinon
        siegeVoisinOccupé← faux
        pour chaque Siege s2 dans NewE faire
            si la distance entre s et s2 est inférieur à distanceMinimale alors
                siegeVoisinOccupé ← vrai
                sortir de la boucle pour
            fin si
        fin pour
        si siegeVoisinOccupé = faux alors
            ajouter s dans NewE
        fin si
    fin si
fin pour
retourner NewE
fin fonction

Cet algorithme est plus complexe que le premier, on utilise le premier algorithme pour obtenir une première solution, ensuite tant qu'on aura pas dépassé le nombre maximum d'itération ou le temps maximale, on choisit au hasard un siège occupé et libre que l'on va inverser.

Le fait d'inverser va parfois ne plus respecter la distance entre les autres sièges et donc on va ne rendre occupés les autres sièges autour du siège libre que l'on vient d'ajouter.

Par ailleurs le fait d'avoir libérer un siège, va peut être pouvoir permettre à d'autres sièges de devenir occupé, ainsi on parcours une nouvelle fois la liste des sièges de la salle pour déterminer les nouveaux sièges que l'on va rendre occupés et on retourne ce nouvel ensemble de sièges occupés que l'on va récupérer. Si le nouvel ensemble améliore la solution alors on remplace la solution sinon on ne fait rien, on répète ce processus jusqu'à atteindre une des deux limites fixées puis on retourne l'ensemble final des sièges occupés.

Pour finir, le plus complexe d'entre tous, l'algorithme génétique :

- Trois paramètres à connaître : taillePopulation, kMeilleurs et generationMax :
    - *taillePopulation* est le nombre maximum d'individus que nous allons créer, cela permet d'éviter une recherche de solution trop longue.
    - *kMeilleurs* est le nombre de solutions à retenir, parmi les meilleures.
    - generationMax est le nombre de solutions de départ à générer où on viendra récupérer les kMeilleurs Solutions.

Dans un premier temps, nous allons générer nos premières solutions qui risquent de ne pas d'être très impressionnantes.
En effet, cela est généré aléatoirement mais avec une particularité c'est que en fonction de la Distance_Min, le random sera plus ou moins élevé, logiquement plus la Distance_Min est élevé moins il aura de places occupés donc dans notre générateur, nous aurions aussi moins de chance d'occuper un siège. Notre équation est : Si (random < MIN_DISTANCE/50) alors la place est occupée. Toujours dans la génération, nous vérifierons que les solutions créées ne l'ont pas déjà été. Sinon nous regénérons jusqu'à obtenir generationMax.

Ensuite, nous devons trier nos solutions du plus grand score au plus petit. Le score est calculé de sorte qu'une pièce non valide prend pour valeur obligatoirement zéro. Et que pour les autres, nous allons compter le nombre de sièges occupés qui fera office de score.

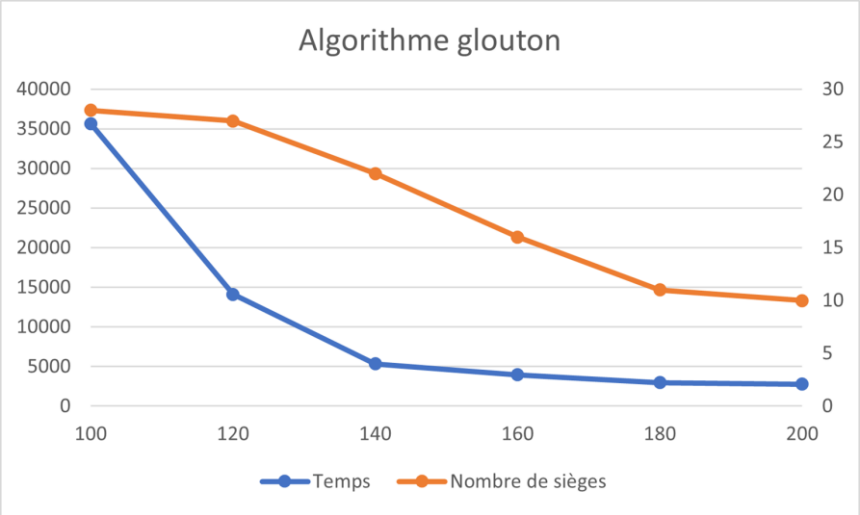
Après le tri, nous allons récupérer les kMeilleurs valeurs.

Passons à présent à la présentation à la deuxième étape, qui est la plus compliquée.
Donc avec nos k solutions nous alons modifier l'état d'un siège, dans les deux sens.
Nous allons donc doubler nos kSolutions. Et nous allons recommencer le processus vu plus haut, le tri, puis la sélection de ces kSolutions.
On s'arrête quand nous avons obtenu un nombre de solution totale égale à la taille de la population.

III. Résultats numériques

3.1 Algorithme naïf

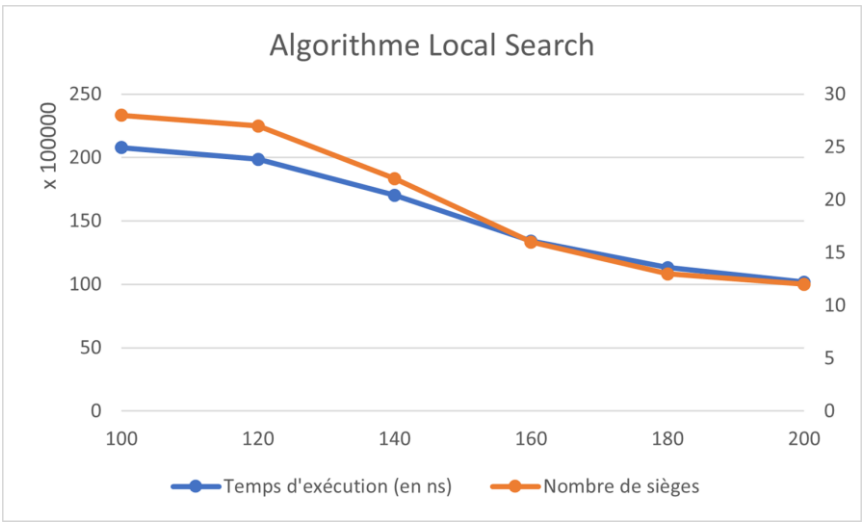
Les résultats numériques pour l'algorithme naïf sont les suivants :



Le temps affiché est en nanosecondes, on remarque donc que l'algorithme glouton est très rapide, au maximum, il met entre 35000 et 5000 nanosecondes à obtenir une solution, cependant il n'est pas très efficace en terme d'optimalité des solutions retournées avec seulement 28 sièges pour une distance d'un mètre et seulement 10 sièges pour la distance de deux mètres.

3.2 Algorithme de recherche locale

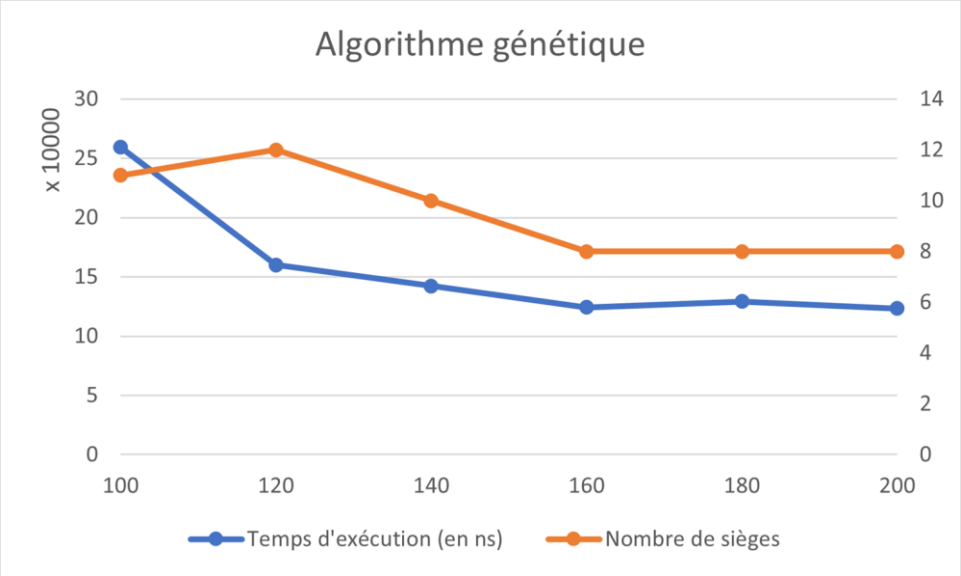
Les résultats numériques pour l'algorithme de recherche locale sont les suivants :



Le temps affiché est en nanosecondes, et le graphique possède une échelle logarithmique pour on remarque donc que l'algorithme est plus lent que l'algorithme glouton avec près de 20 millions de nanosecondes pour le plus haut et 10 millions de nanosecondes pour le plus bas, cependant il augmente le nombre de sièges de deux sièges supplémentaires pour une distance minimale de deux mètres.
12 28 8 12

3.3 Algorithme avec heuristiques

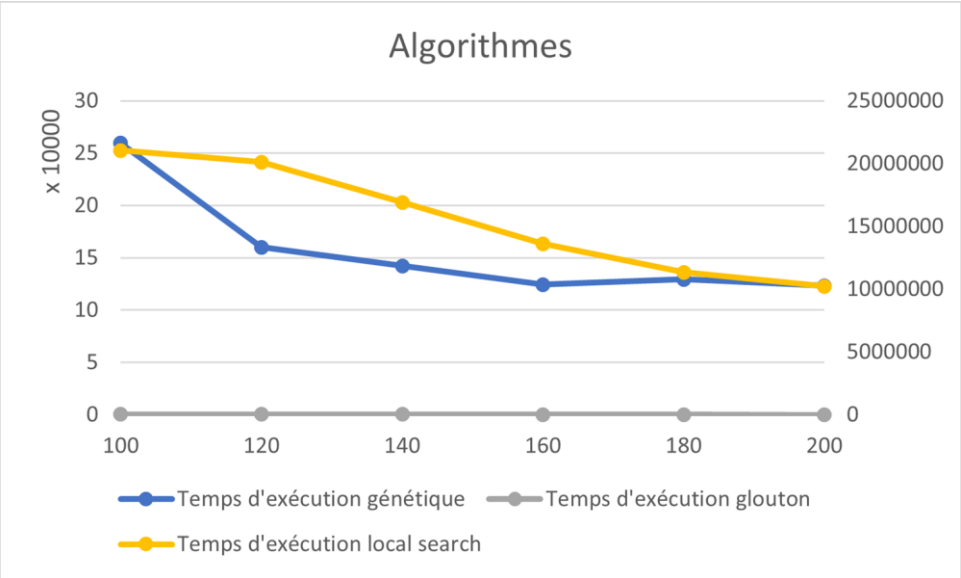
Les résultats numériques pour l'algorithme avec heuristiques sont les suivants :



Cette fois-ci l'algorithme est plus rapide que le deuxième algorithme avec seulement 250 000 nanosecondes, cependant le nombre de sièges maximale est compris entre 8 et 12 sièges seulement. Cela est dû au fait que les paramètres de l'algorithme était mal configuré.

3.4 Comparaison des algorithmes

On va maintenant comparer les algorithmes ensemble.



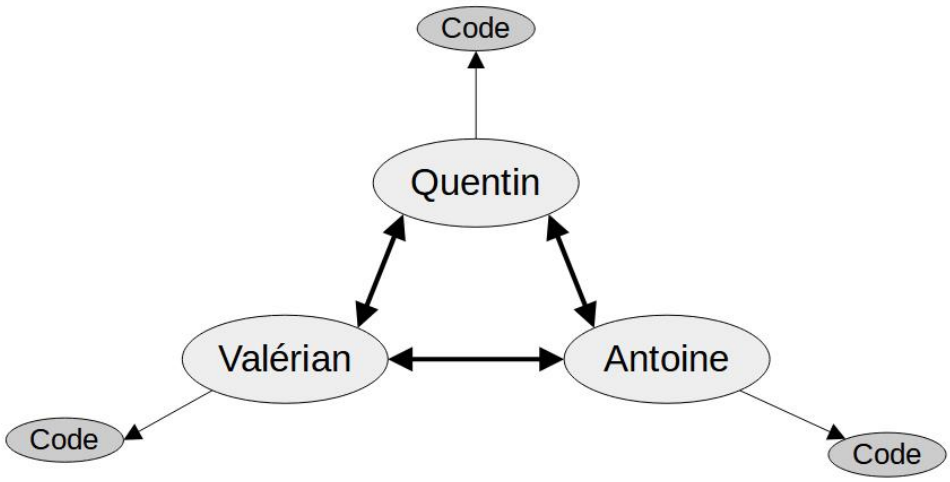
Tout d'abord, nous pouvons remarquer que l'algorithme local semble le plus rapide mais ce qu'il faut savoir c'est que la configuration n'était pas les même, en effet, l'algorithme génétique à pour particularité de fonctionner avec plusieurs recherches à la fin. Cependant, il est certains que l'algorithme glouton est le moins performant.

IV. Organisation

4.1 Méthode d'organisation

Concernant notre organisation, au départ nous avons créer un GitHub afin de mettre en commun notre code. Cependant nous nous en sommes que très peu servi, du fait que nous avons beaucoup échanger sur Discord, et qu'il était plus facile de nous envoyer les bouts de codes afin que l'on puisse travailler à trois rapidement sur le code, plutôt que d'à chaque fois envoyer sur GitHub, récupérer les projets, etc.

Au final, on peut résumer notre méthode de travail au graphique suivant :



4.2 Rôle de chacun

En ce qui concerne le code, Quentin a fait la classe Seat, la fonction displayRoom de la classe Room et la fonction isFeasible. En terme de quantités au final, il ne reste plus grand-chose mais au départ, on était parti sur l'idée d'avoir la solution que l'on vient de décrire dans ce rapport mais aussi une autre solution similaire basée sur une structure de donnée différente, à partir de matrices avec des 0 et des 1 pour représenter les sièges dont Quentin était chargé de la réaliser mais au final l'idée n'a pas était garder jusqu'au bout puisque l'on s'est rendu compte que la solution n'était pas si bonne que ça et on a préféré se concentrer sur la première.

Valérian quant à lui à fait les classes HeuristicSolver, Room et Comparateur et Antoine à fait le reste des classes c'est à dire Solver, GreedySolver, LocalSearchSolver, LoaderUtils et BenchMark.

Concernant, l'investissement de chacun dans ce projet, nous nous sommes tous investis et avons beaucoup communiqué ensemble, on s'entraidait beaucoup lorsque que quelqu'un ne comprenait pas quelque chose, on lui expliquait, etc.

Au final, nous avons échangé un peu plus de 1500 messages sur Discord confirmant nos échanges nombreux en dehors des séances.

V. Conclusion

5.1 Conclusion générale

En conclusion, nous pouvons dire que les différents algorithmes ont tous des avantages et des inconvénients.
Tantôt la facilité de compréhension de l'algorithme est intéressante mais sa rapidité diminue et inversement. Tout ceci, nous montre qu'il faut choisir avec soin son algorithme en fonction du travail que l'on souhaite accomplir.

Nous pouvons encore sité le Recuit Simulé ou le Backtracking jusqu'au Forward-Checking mais ce qui importe c'est l'usage que l'on en fait tout simplement. Notre problème était minimaliste car il concernait une seule pièce mais déjà nous pouvons voir les limites de certains algorithmes. Nous pouvons déjà nous tourner vers des nouvelles possibilités comme l'extension vers un Bâtiment qui comporterait des salles et qu'entre chaque classe il doit y avoir une certaine distance et prendre en considération les étages.

Au final, se projet fut rude car très algorithmiques mais il nous a permit d'apprendre de nouveaux algorithmes, de nous perfectionner dans l'algorithmie donc et de créer un projet en groupe afin de se surpasser en donnant le meilleur de nous même.

Lien vers le GitHub : GitHub - Vallaser/COVID