université de BORDEAUX

Collège Sciences et Technologies
UF Mathématiques et Interactions

Placement optimal de pompiers pour éteindre des incendies

Déborah Nash
Lin Hirwa Shema
Lucas Villenave
Martin Debouté

Projet d'algorithmique appliquée

M2 CMI OPTIM / GL
2022–2023

Table des matières

Engagement de non plagiat

Nous, Déborah Nash, Lin Hirwa Shema, Lucas Villenave, Martin Debouté, déclarons être pleinement conscients que le plagiat de documents ou d'une partie d'un document publiés sur toutes formes de support, y compris l'internet, constitue une violation des droits d'auteur ainsi qu'une fraude caractérisée.

En conséquence, nous nous engageons à citer toutes les sources que nous avons utilisées pour produire et écrire ce rapport.

Fait à Talence le 19 janvier 2023.

Signatures

Déborah Nash
Lin Hirwa Shema
Lucas Villenave
Martin Debouté

1 Introduction

Dans ce projet l'objectif est de protéger un ensemble de ville de plusieurs départs de feux. Ces feux se propagent en lignes droites dans toutes les directions et pour les arrêter nous pouvons déployer des pompiers. L'objectif est de contenir les feux avec un nombre de pompier minimal de manière à ce qu'aucune des villes ne soit mise en danger. La propagation d'un feu est simulée par un nombre fini de rayons partant du foyer dans toutes les directions.

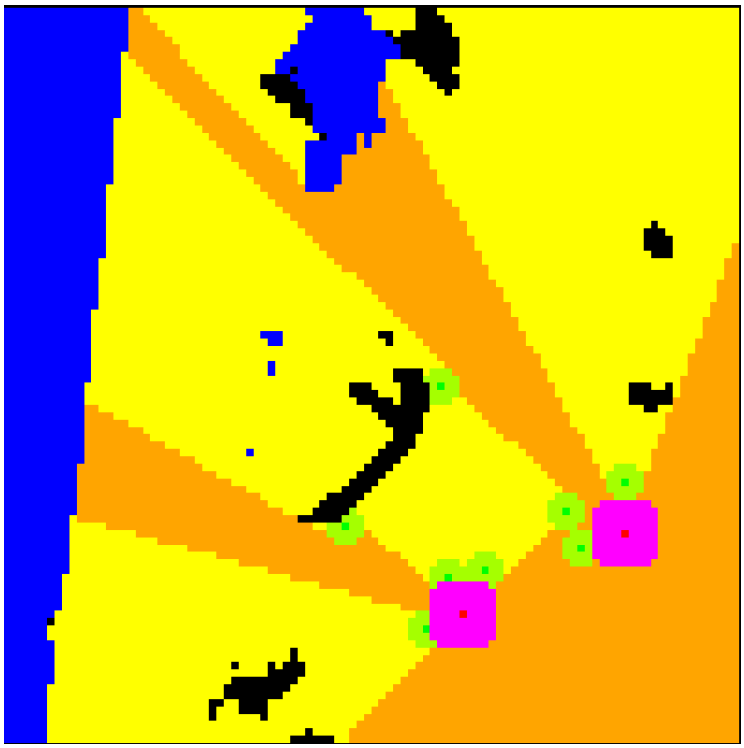

FIGURE 1.1 – Exemple graphique d'une solution au problème

Dans cet exemple, les pixels noirs représentent des villes, les pixels jaunes les terrains vagues et les bleues des terrains inaccessibles aux feux et aux pompiers. Les pompiers sont affichés en vert et leur champ d'action en vert clair. Les feux sont représentés en rouge et en violet leur surface de fournaise, zone inaccessible aux pompiers. On représente en orange la propagation du feu.

2 Formalisation du problème

2.1 Choix de représentation du problème

Dans les données du problème, les centres des feux correspondent à des pixels rouges dans l'image. Mais pour pouvoir représenter les surfaces de fournaise de ces feux ainsi que leurs lignes de propagation, on a choisi de transmettre ces données sur une grille à valeurs réelles. On calcule le cercle de fournaise de chaque feu ainsi que la demi-droite issue de son centre pour chacune de ses lignes de propagation, et enfin on les discrétise de nouveau en pixels. Pour cela, on a dû faire les choix, suppositions et concessions suivantes :

— Si un feu ou un pompiers a comme centre le pixel (x, y), alors son centre réel utilisé dans nos calculs sera $(x + 0.5, y + 0.5)$. Ceci permet que le feu ou pompier reste bien centré dans le pixel et est équidistant à tous ses pixels voisins, quelque soit l'orientation de la carte.

— Tout pixel se trouvant à l'intérieur du cercle de fournaise d'un feu fait partie de la surface de fournaise (ensemble de pixels) du feu. De plus, on dit que tout pixel traversé par ce cercle fait aussi partie de la surface de fournaise du feu. Donc, même si le cercle ne touche que très peu un pixel, un pompier ne pourra pas être déployé sur ce pixel. La surface d'action d'un pompier est aussi décidé de la même manière.

— Tout pixel traversé par une demi-droite issue d'un centre de feu fait partie du chemin de la ligne de feux correspondante. Ce chemin de pixels (liste de pixels) est construit itérativement en suivant la droite depuis le centre de feu, jusqu'à ce que la droite touche un pixel de ville, ou de terrain inaccessible. Ainsi, on ne considère une ligne de propagation de feu comme « dirigé vers une ville » que si la droite correspondante touche la ville (*i.e.* le dernier pixel sur le chemin de cette ligne est un pixel ville). Donc même si un pixel du chemin de la ligne de feux est adjacent à un pixel « ville », cela ne veut pas nécessairement dire que ce dernier est à risque de cette ligne de feux (la figure 2.1 montre un exemple du cas où le pixel $(2, 3)$ est adjacent au pixel « ville » $(2, 2)$ sans pour autant que la ville soit à risque de feu).

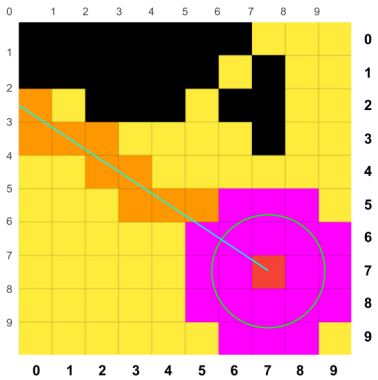

FIGURE 2.1 – Choix de représentation des surfaces de fournaises et des rayons de feu en pixels (coordonnées réels en petit et coordonnées pixels en grand et gras).

Sur la figure ci-dessus (2.1), on a un feu centré sur le pixel $(7, 7)$ et de coordonnées réels $(7.5, 7.5)$. On a son cercle de fournaise en vert, sa surface de fournaise (ensemble de pixels) en magenta, la demi-droite d'une de ses lignes de propagation en cyan et son chemin de propagation induit (liste de pixels) en orange.

— On considère qu'un placement de pompier arrête une ligne de feux uniquement si un des pixels sur le chemin de cette ligne de feux se retrouve dans l'ensemble de pixels constituant la surface d'action du pompier. Il faut noter ici que même si le cercle d'action réel du pompier n'intercepte pas la demi-droite de la ligne de feux, il suffit que les pixels traversés par ceux-si s'intersectent pour qu'on considère quant-même que le pompier arrête la ligne de feux.

Une fois ces calculs et choix fait, on ne travaille plus qu'avec des pixels et des listes de pixels et toutes nos données sont discrètes.

2.2 Réduction de la taille du problème

Si on devait évaluer le placement d'un pompier sur chaque pixel de la carte, le temps de calcul de notre solution exploserait. En effet, pour notre cinquième instance 4.5 dont l'image est de dimension 100×100, on aurait 10000 placements de pompiers possibles, ce qui serait extrêmement coûteux si on essayait d'évaluer toutes les combinaisons possibles des placements de pompiers. Il nous faut donc réduire notre liste de placements possibles de pompiers pour ne garder que les plus pertinents sans perte de solution(s) optimale(s).

Tout d'abord, il faut éliminer les placements non réalisables. On ne peut pas placer un pompier dans un milieu aquatique ou dans la surface de fournaise d'un feu. De plus, on fait l'hypothèse qu'on ne peut pas déployer des pompiers dans un milieu urbain (puisque ça laisserait sous-entendre que le feu a déjà atteint la ville, ce que l'on ne veut pas). Avec ceci, on peut éliminer de nos placements potentiels de pompiers les pixels correspondant à ces milieux.

Ensuite, on voit bien qu'il serait inutile de placer un pompier dans un endroit où il n'arrêterait aucune ligne de feux. C'est pour cela qu'après avoir calculé la trajectoire de chaque ligne de feux, on cherche les pixels autour de cette trajectoire de telle manière que la surface d'action d'un pompier placé sur ces pixels arrêterait la propagation de ce rayon de feu. En limitant les emplacement possibles à ceux qui arrêtent des lignes de feux, on réduit encore plus la taille du problème.

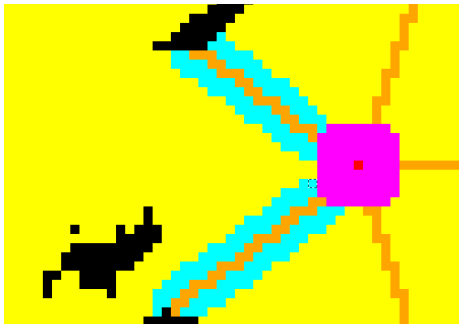

FIGURE 2.2 – Placements possibles de pompiers (zones en cyan) pour arrêter les lignes de feux (en orange).

Comme énoncé précédemment, on remarque qu'il n'y a pas d'intérêt à placer un pompier pour arrêter une ligne de feux qui ne se dirige pas vers une ville. Donc on se réduit à ne considérer que les lignes de feux qui se dirigent vers des villes et les pompiers qui peuvent les arrêter.

Enfin, notre plus grande réduction du problème résulte d'une élimination des symétries. En effet si on a un placement de pompier (x_1, y_1) qui n'arrête que la ligne de feux $r \in R$ (R étant l'ensemble des lignes de feux dirigés vers des villes) et un autre placement (x_2, y_2) qui n'arrête que cette même ligne de feux r, alors ces deux placements sont équivalents et on peut en éliminer l'un des deux sans risquer de perdre une meilleure solution.

Si on a un placement de pompier (x_1, y_1) qui stoppe un ensemble de lignes de feux $R_1 \subseteq R$ et un autre placement de pompier (x_2, y_2) qui arrête un ensemble de lignes de feux $R_2 \subseteq R$ tel que $R_2 \subseteq R_1$, alors le placement (x_1, y_1) est meilleur que le placement (x_2, y_2) et on peut éliminer ce dernier de notre liste de placements potentiels sans perdre une meilleure solution. En effet, s'il existe une solution optimale contenant (x_2, y_2), alors on peut remplacer cette position par celle en (x_1, y_1) et couvrir les mêmes lignes de feux (voir même plus) tout en gardant le même nombre de pompiers.

Petit exemple explicatif

Voici un exemple simple pour illustrer nos propos :

$(a - f)$ des positions de pompiers, $(1 - 4)$ des feux ;
a couvre (1, 2)
b couvre (2, 3)
c couvre (1, 4)
d couvre (2)
e couvre (3)
f couvre (1)

Ici, a couvre strictement plus que d et f. Le sommet que couvre e est également couvert par b. On conservera donc (a, b, c) comme noeuds de « pompiers potentiels ». Cette méthode permet de réduire drastiquement le nombre de positions potentielles de pompiers, nous permettant ainsi de pouvoir appliquer nos algorithmes sur des instances de tailles assez grosses.

Cette élimination de symétrie a un coût de $O(r^2)$ où r est le nombre total de lignes de feux dirigés vers des villes. Elle réduit considérablement la taille du problème à résoudre pour garder une liste de « pompiers potentiels » la plus petite possible.

2.3 Création du graphe

Une fois la réduction terminée, on créer un sommet « pompier potentiel » pour chaque placement potentiel de pompier non-éliminé et on crée un sommet « feu » pour chacune des lignes de feux dirigées vers des villes. Enfin, on ajoute des arêtes entre les sommets « pompiers potentiels » et les sommets « feux » qu'ils couvrent. Avec l'exemple précédent (2.2), le graphe serait :

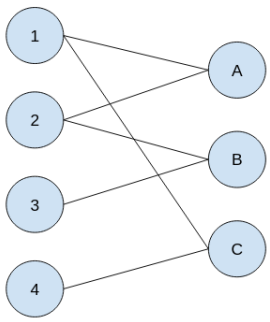

FIGURE 2.3 – Graphe de l'exemple 2.2.

On notera ici que le graphe est biparti et donc que le problème original équivaut à un problème de domination de la partie gauche (les sommets « feux ») par un sous-ensemble de taille minimale de la partie de droite (les sommets « pompiers potentiels »).

2.4 Structure de données

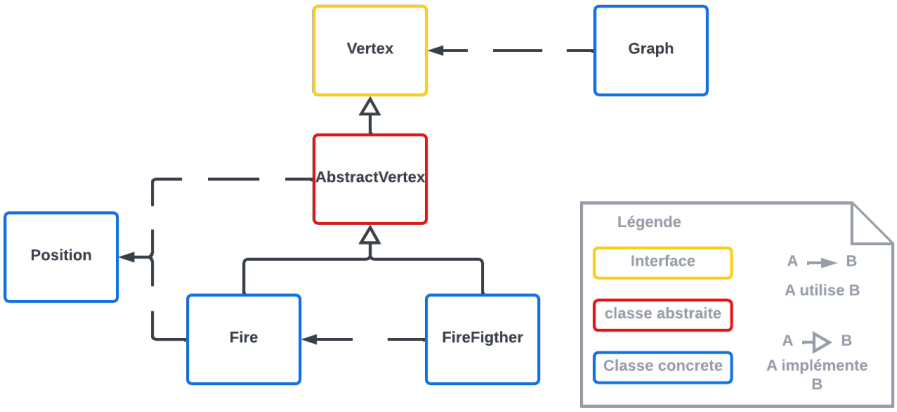

FIGURE 2.4 – Diagramme de classes UML de la structure de données du graphe

Comme il est possible de le voir dans la figure 2.4, les données du graphe sont stockées dans trois classes principales : `Fire`, `FireFigther` et `Graph`.

Nous avons fait le choix, ici, de ne pas représenter les arêtes du graphe par une classe. En effet nous avons pris parti de stocker la liste d'adjacence du graphe directement dans la classe `Graph` afin d'optimiser son parcours. De plus, nous n'avons aucune donnée relative aux arêtes à stocker qui ne puissent pas être récupérées via cette liste.

2.4.1 Fire

La classe `Fire` nous permet de stocker les sommets du graphe qui représente les lignes de feux qui touchent une zone à protéger. Elle contient exclusivement la position du centre de feu et la position de l'intersection de la ligne de feux avec un élément de la carte.

Ces deux positions sont déterminées *a priori*, lors de la lecture de l'image et de la génération des données. Une collision est décrite comme l'intersection entre la ligne de feux et un élément de la carte : le bord de l'image, une ville ou un point d'eau. Néanmoins, seules les lignes de feux qui entrent en collision avec les villes nous intéressent ici, ce sont donc seulement ces lignes de feux qui sont enregistrées dans les données.

2.4.2 FireFighter

La classe `FireFighter` représente un sommet du graphe qui correspond à une position de pompier. Elle stocke la position du pompier et la liste des lignes de feux qu'il intercepte.

2.4.3 Graph

La classe `Graph` stocke toutes les données du graphe, elle contient la liste des pompiers et la liste des lignes de feux. À la création du graphe, un premier élagage parmi les pompiers est réalisé comme expliqué dans la partie 2.2.

3 Méthodes de résolution du problème

3.1 Modèle mathématique linéaire en nombres entiers

Dimensions

$I = \{1, ..., n \times m\}$: l'ensemble des pixels de l'image.
$F \subset I$: l'ensemble des pixels qui correspondent à des foyers de feux.
$R = \bigcup_{f \in F} R(f)$: l'ensemble de lignes de feux dirigé vers des villes.
$P = \bigcup_{r \in R} P(r)$: l'ensemble de pixels réalisables pour le placement d'un pompier.

Données

$\forall f \in F, R(f)$: l'ensemble des lignes de feux émis par le feu f qui se dirigent vers une ville.
$\forall r \in R, P(r)$: l'ensemble des pixels réalisables pour le placement d'un pompier qui se trouvent dans le voisinage de la ligne de feux r (placer un pompier sur un de ces pixels arrête la propagation de cette ligne de feux vers la ville).

Variables de décision

$x_p \in \{0, 1\}$: 1 si on place un pompier au pixel $p \in P$, 0 sinon.

Fonction objectif

$$\min \ : \sum_{p \in P} x_p \tag{3.1}$$

L'objectif (3.1) est de minimiser le nombre total de pompiers à placer.

Contraintes

$$\sum_{p \in P(r)} x_p \geqslant 1 \qquad \forall r \in R \tag{3.2}$$

La contrainte (3.2) assure que chaque ligne de feux dirigée vers une ville se trouve dans le champ d'action d'un pompier (*i.e.* est stoppé).

Implémentation du modèle mathématique

En pratique, il existe des outils informatiques appelés solveurs (on peut citer Gurobi qui est le solveur MIP [1] que nous utilisons ici). Ces outils requièrent que nous leurs donnions notre définition mathématique du problème, ainsi qu'une instance de données. Ensuite, le solveur utilise des méthodes avancé d'énumération (algorithmes de *branch-and-cut*) de manière à trouver la solution optimale.

Sans rentrer dans des explications complexes, le solveur peut être vu plus simplement comme une boîte noire qui à partir d'une définition mathématique rigoureuse et un jeu de données, retourne une solution réalisable au problème avec un certain degré de qualité (*gap* d'optimalité) ou au mieux la solution optimale. Les performances d'un solveur sont remarquables sur la plupart des problèmes qui sont « simples » à définir mathématiquement (tel que notre problème). De plus, les solveurs ont des méthodes avancées d'élimination de symétries qui sont meilleures que celle qu'on décrit à la fin de la partie 2.2. C'est pour cela que contrairement à nos autres méthodes, lorsqu'on résoudra le problème avec le solveur, on lui passera les données sans d'abord éliminer les symétries.

Cette méthode de résolution trouve la solution optimale pour chacune de nos instances. Néanmoins il ne faut pas oublier que la complexité d'un solveur est dans le pire des cas exponentielle, et bien qu'efficace sur la plupart des instances de problèmes classiques, il faudra se tourner vers des méthodes plus avancées (*e.g.* des méthodes de décompositions mathématiques avancées) pour résoudre des problèmes plus complexes et de plus grandes tailles.

3.2 Bruteforce

L'algorithme *bruteforce* est une énumération explicite intelligente. Il prend en entrée les listes des pompiers potentiels et des feux (préalablement réduites) et va rechercher l'équipe de pompiers de plus petite taille couvrant tous les feux en énumérant toutes les possibilités. Cette méthode a l'avantage de retourner la solution optimale mais elle prend énormément de temps.

En effet une énumération explicite complète d'une liste de taille n reviendrait à chercher la meilleure solution réalisable parmi les 2^n partitions possibles. On voit vite qu'avec une liste de taille ne serait-ce que 50 on aura $2^{50} = 1125899906842624$ possibilités à énumérer. C'est pour cette raison que les tests de cet algorithme seront vite contraints par la taille des instances (*c.f.* 4.3).

Pour éviter cette exploration complète, nous faisons une énumération explicite intelligente en éliminant, avant la recherche, de nombreuses partitions grâce à des bornes. Nous pouvons déjà fixer une borne supérieur sur le nombre maximum de pompiers nécessaire pour contenir tous les feux. Il suffit de placer un pompier par ligne de feux. Cette borne supérieure est donc égale à la taille de la liste de lignes de feux. Pour ce qui est d'obtenir une borne inférieur sur le nombre minimal de pompiers nécessaires pour stopper tous les feux, nous pouvons tout simplement prendre la partie entière supérieure du nombre de feux divisé par le nombre maximum de feux couverts par le meilleur pompier de la liste.

On va ensuite trier la liste des pompiers potentiels par le nombre de feux qu'ils peuvent éteindre de manière décroissante. Puis on génère uniquement les partitions (combinaisons de pompiers) de taille au moins borne inférieure et au plus borne supérieure. On génère d'abord toutes les partitions (*i.e.* solutions) de taille égale à la borne inférieure. Si une de ces partitions est réalisable alors c'est une solution optimale et on arrête l'algorithme. Sinon, on augmente la borne inférieure de 1 et on régénère toutes les partitions de taille égale à la nouvelle borne inférieure. On répète ceci jusqu'à ce qu'on obtienne une solution réalisable qui sera une solution optimale pour le problème, de valeur égale à la borne inférieure courante.

1. Mixed Integer Programming

3.3 Glouton

L'algorithme glouton, *greedy* en anglais, est tout ce qu'il y a de plus simple. Le principe est de construire une équipe de pompiers en prenant un à un les pompiers les plus efficaces.

On part de deux listes, la liste des sommets feux et des pompiers potentiels (listes ayant déjà été réduites). On parcourt la liste des pompiers potentiels et on cherche le pompier couvrant le plus de feux de notre liste de sommets feux. Ensuite, on retire de notre liste les feux qui sont couverts par le pompier placé pour ne plus garder que les feux qui ne sont toujours pas couverts. On répète finalement ce procédé jusqu'à ce que tous les feux soit pris en charges par les pompiers.

En résumé, à chaque itération on sélectionne un pompier couvrant le plus de feux non couvert possible. Notre solution sera défini par l'ensemble des pompiers sélectionnés. Cette méthode à l'avantage de nous donner une « bonne » (*c.f.* 4.3) solution réalisable très rapidement.

3.4 Recuit simulé

L'algorithme du recuit simulé, *simulated annealing* en anglais, appartient à une catégorie de métaheuristiques [2]. L'idée de base est d'imiter le processus de recuit en métallurgie, où un métal est chauffé à haute température puis refroidi lentement pour augmenter son intégrité structurelle. Dans notre cas une solution peut se représenter comme un vecteur de 0 et de 1, où le 1 à la ième position signifie qu'on sélectionnera le ième pompier dans l'équipe, le 0 non.

Dans le contexte de l'optimisation, l'algorithme commence par une solution initiale aléatoire (dans notre cas généré à partir de notre algorithme glouton présenté précédemment, 3.3), puis y apporte de manière itérative de petites modifications (dans notre cas grâce à une méthode dite de *pick and drop* [3]), la probabilité d'accepter une nouvelle solution diminuant à mesure que la température (un paramètre de contrôle) diminue.

L'algorithme procède en une série d'étapes, où à chaque étape une nouvelle solution est générée en apportant de petites modifications à la solution actuelle. La nouvelle solution est ensuite évaluée et comparée à la solution actuelle. Si la nouvelle solution est meilleure que la solution actuelle, elle est acceptée comme nouvelle solution actuelle. Si la nouvelle solution est moins bonne que la solution actuelle, elle est acceptée avec une probabilité qui diminue avec la température.

La température est progressivement diminuée au cours du processus d'optimisation. Au début, lorsque la température est élevée, l'algorithme est plus susceptible d'accepter de mauvaises solutions, ce qui lui permet d'explorer plus largement l'espace de recherche. À mesure que la température diminue, l'algorithme devient plus sélectif et moins susceptible d'accepter les pires solutions, ce qui lui permet de converger vers une meilleure solution.

Dans l'ensemble, le recuit simulé est un algorithme d'optimisation puissant qui peut trouver de bonnes solutions à des problèmes d'optimisation difficiles. Cependant, cela nécessite une bonne compréhension du problème et du réglage des paramètres. Il est également coûteux en temps de calcul et la complexité temporelle augmente avec la taille du problème.

2. Une métaheuristique est un algorithme d'optimisation visant à résoudre des problèmes d'optimisation difficile de manière approchée (souvent issus des domaines de la recherche opérationnelle, de l'ingénierie ou de l'intelligence artificielle) pour lesquels on ne connaît pas de méthode classique plus efficace.

3. Cette méthode sélectionne aléatoirement un élément du vecteur de solution pour changer sa valeur, si c'est un 0, cela devient un 1 et inversement si c'est un 1, cela devient un 0. Cependant, vous remarquerez que dans notre problème la nouvelle solution qui résulte de ce mouvement n'est pas nécessairement réalisable. Mais ce n'est pas grave puisque nous lui attribuerons un coût très élevé.

4 Expérimentations et résultats

4.1 Présentation des instances

Une instance est constituée de deux fichiers. Une image au format `PPM` ainsi qu'un fichier de configuration au format `TXT`.

Nous avons donc une matrice $n \times m$ de pixels colorés. La couleur d'un pixel (un triplet de nombres compris entre 0 et 255) représente le type de structure se trouvant à la position associée. Les différentes structures possibles sont : départ de feu (en rouge), ville (en noir), terrain infranchissable (en bleu, *e.g.* de l'eau) et terrain vague (en jaune).

Le fichier de configuration quant à lui nous indique : le nombre d'angles (*i.e.* le cercle des directions possibles du feu est discrétisé en un nombre fini de rayons), le rayon de fournaise (*i.e.* le rayon du cercle dans lequel un pompier ne peut pas intervenir pour éteindre le feu) et le rayon d'action d'un pompier.

Dans la suite, nous considérons des instances avec les mêmes valeurs pour ce qui est du rayon de fournaise et du rayon d'action d'un pompier que l'on fixera à 2 et 1.5 pour les petites instances ($<$ 100x100), et 4.5 et 1.6 respectivement pour les moyennes et grosses instances ($\geqslant$ 100x100).

Pour chaque instance, nous ferons varier le nombre d'angle pour évaluer nos algorithmes. Idéalement, quand le nombre d'angles est suffisamment grand, le feu se répand dans toutes les directions sans angles morts tel qu'on peut le voir *ici*. L'une des questions que nous nous sommes posé, était de déterminer la limite à partir de laquelle ces angles morts n'apparaissaient plus.

Pour calculer ce nombre d'angle maximal nous avons déterminé la distance la plus longue entre un départ de feu et un coin de l'image et nous avons ensuite déterminé l'angle minimal à partir duquel ce pixel « le plus éloigné possible » serait nécessairement couvert. Pour ce faire, nous avons imaginé un triangle passant par le départ de feu en question et les deux angles opposés au pixel du coin (les angles (0,1) et (1,0)). Ensuite, nous avons calculé l'angle au départ du feu grâce au théorème d'Al-Kashi. Le nombre d'angle maximal est donc, au plus, 360 divisé par cet angle.

Voici ci-dessous les images des instances que nous considérons dans nos tests par la suite.

Instance 1 (15x15 et 1 départs de feux) :

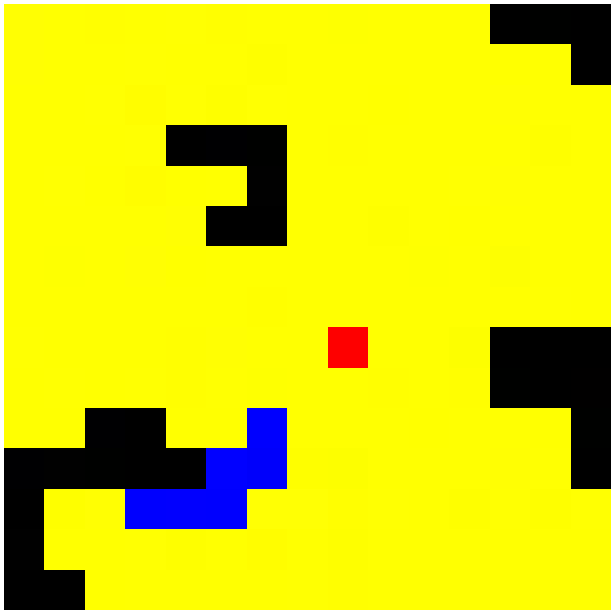

FIGURE 4.1 – Image de l'instance 1.

Instance 2 (20x20 et 2 départs de feux) :

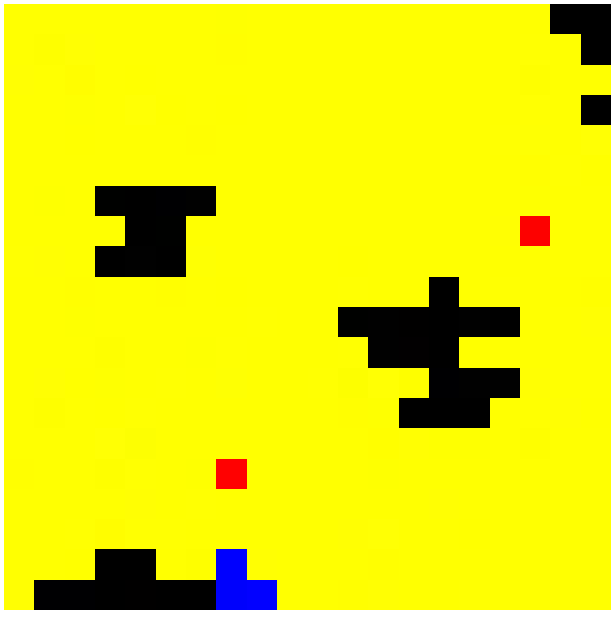

FIGURE 4.2 – Image de l'instance 2.

Instance 3 (50x50 et 1 départs de feux) :

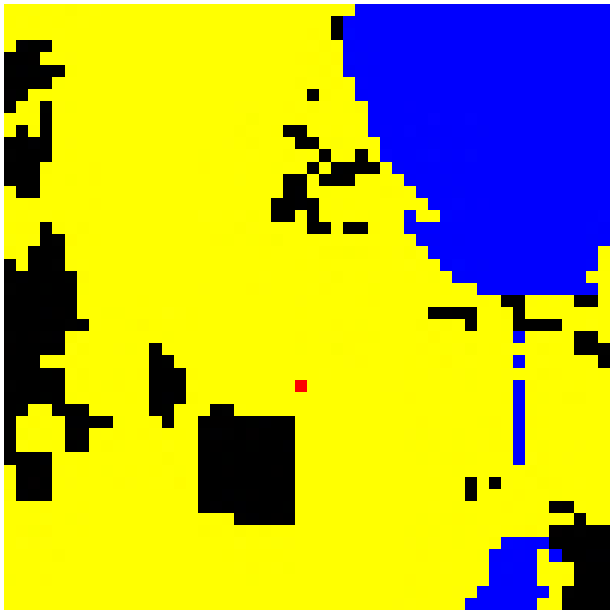

FIGURE 4.3 – Image de l'instance 3.

Instance 4 (50x50 et 3 départs de feux) :

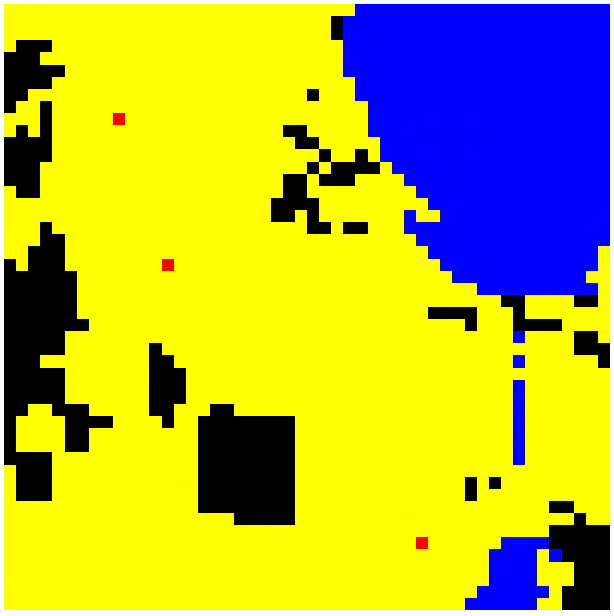

FIGURE 4.4 – Image de l'instance 4.

Instance 5 (100x100 et 2 départs de feux) :

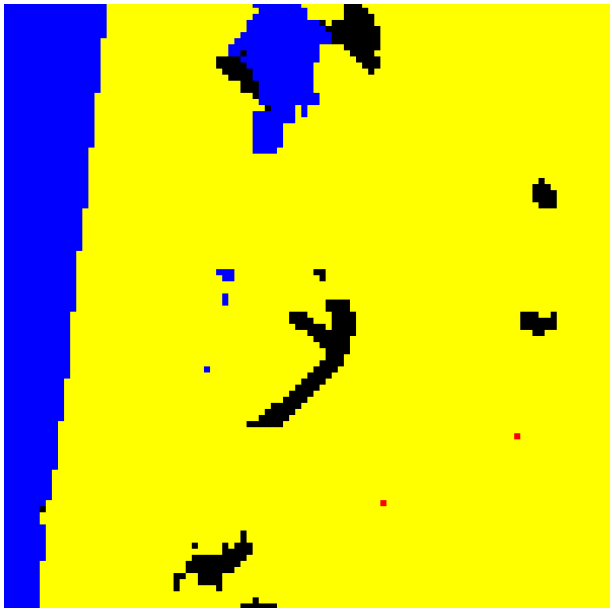

FIGURE 4.5 – Image de l'instance 5.

Instance 6 (486x421 et 3 départs de feux) :

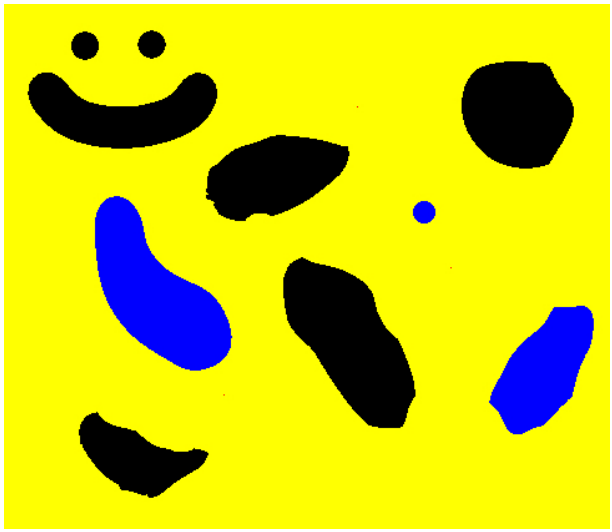

FIGURE 4.6 – Image de l'instance 6.

Instance 7 (789x1388 et 4 départs de feux) :

FIGURE 4.7 – Image de l'instance 7.

4.2 Sorties de nos programmes

Tous nos algorithmes ont la même sortie sur le terminal et enregistrent la solution directement sur une image du nom `result_<nom_de_l'instance>.ppm` dans un dossier nommé `solution`.

Voici un exemple de sortie pour la résolution de l'instance 6 (4.6) avec notre algorithme de recuit simulé (3.4) :

```
❯ ./test.out data/instance_6 -sa
Number of angles: 150
Furnace radius: 4.5
Radius of action of a firefighter: 1.6
Map size: 421x486

Necessary number of angles: 1904

Start gathering data...
Finished gathering firecenters and partial feasible placement data!
Finished gathering fire furnace areas and removed them from feasible placements!
Finished gathering ray paths and feasible fighter placements!
Finished gathering data!

Start cutting useless fighters... (starting with 35153 fighters)
Finished cutting useless fighters in 7.94208 sec! (268 remainings)

Initial temperature: 9.62377e+06
Result: runtime = 1.68269 sec; objective value = 12

We place a fighter at position (288, 82)
We place a fighter at position (361, 205)
We place a fighter at position (353, 209)
We place a fighter at position (179, 308)
We place a fighter at position (171, 316)
We place a fighter at position (277, 83)
We place a fighter at position (355, 216)
We place a fighter at position (173, 318)
We place a fighter at position (353, 214)
We place a fighter at position (282, 87)
We place a fighter at position (181, 312)
We place a fighter at position (281, 87)

Writing result to solution/result_instance_6.ppm
```

FIGURE 4.8 – Sortie de l'algorithme recuit simulé pour l'instance 6 en limitant chaque feu à 150 rayons.

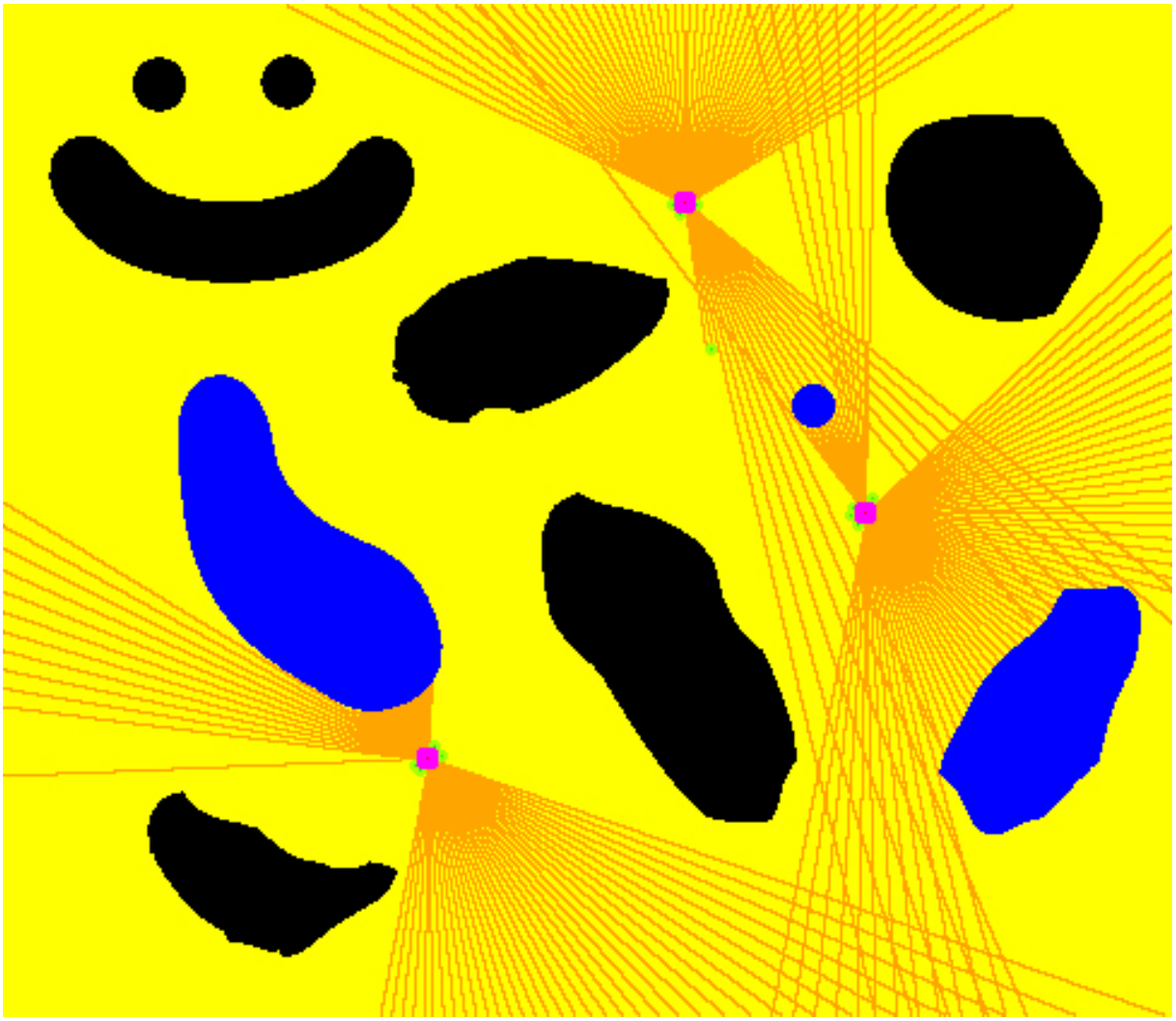

FIGURE 4.9 – Solution graphique de l'instance 6 avec 150 rayons par feu.

Afin de tester nos algorithmes sur toutes nos instances de manière automatique, nous avons développé un script shell dont voici la sortie type :

```
› ./benchmark.sh data --mip
Experimental Campaign:
Data directory: data
Output directory: log/mip
Algorithm: --mip
Time limit: 600

-- GUROBI_LIBRARIES : /Library/gurobi903/mac64/lib/libgurobi_c++.a;/usr/local/lib/libgurobi90.dylib
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/mdeboute/Desktop/Etudes/Master/M2/S9/Applied Algorithmics/AAP/build

[100%] Built target unit_test.out

Unit tests passed, running the benchmark now...

[100%] Built target mip.out

Solving instance_1
Solving instance_2
Solving instance_3
Solving instance_4
Solving instance_5
Solving instance_6
Solving instance_7

Benchmark finished, results are in log/mip/results.csv
```

FIGURE 4.10 – Sortie du *benchmark* sur le MIP.

4.3 Performances des algorithmes développés

4.3.1 Tableaux de résulats

FIGURE 4.11 – Résultats de l'instance 1 (15x15 et 1 départ de feu).

Nombre d'angles	10	48
Durée d'élimination de symétries (sec)	0.00	0.00
Taille de la solution *glouton* | Temps de résolution (sec)	1 | 0.00	4 | 0.00
Taille de la solution *recuit simulé* | Temps de résolution (sec)	1 | 0.00	4 | 0.00
Taille de la solution *bruteforce* | Temps de résolution (sec)	1 | 0.00	4 | 0.00
Taille de la solution **optimale** MIP | Temps de résolution (sec)	1 | 0.00	4 | 0.00

FIGURE 4.12 – Résultats de l'instance 2 (20x20 et 2 départs de feu).

Nombre d'angles	10
Durée d'élimination de symétries (sec)	0.00
Taille de la solution *glouton* | Temps de résolution (sec)	6 | 0.00
Taille de la solution *recuit simulé* | Temps de résolution (sec)	6 | 0.00
Taille de la solution *bruteforce* | Temps de résolution (sec)	6 | 0.00
Taille de la solution **optimale** MIP | Temps de résolution (sec)	6 | 0.00

FIGURE 4.13 – Résultats de l'instance 3 (50x50 et 1 départ de feu).

Nombre d'angles	10
Durée d'élimination de symétries (sec)	0.00
Taille de la solution *glouton* | Temps de résolution (sec)	4 | 0.00
Taille de la solution *recuit simulé* | Temps de résolution (sec)	4 | 0.00
Taille de la solution *bruteforce* | Temps de résolution (sec)	4 | 0.00
Taille de la solution **optimale** MIP | Temps de résolution (sec)	4 | 0.00

FIGURE 4.14 – Résultats de l'instance 4 (50x50 et 3 départs de feu).

Nombre d'angles	10	13	14	33
Durée d'élimination de symétries (sec)	0.01	0.02	0.02	0.11
Taille de la solution *glouton* | Temps de résolution (sec)	8 | 0.00	9 | 0.00	9 | 0.00	11 | 0.00
Taille de la solution *recuit simulé* | Temps de résolution (sec)	8 | 0.00	9 | 0.00	9 | 0.00	11 | 0.40
Taille de la solution *bruteforce* | Temps de résolution (sec)	8 | 0.24	9 | 20.9	9 | 57.3	–
Taille de la solution **optimale** MIP | Temps de résolution (sec)	8 | 0.00	9 | 0.00	9 | 0.00	**10** | 0.01

FIGURE 4.15 – Résultats de l'instance 5 (100x100 et 2 départ de feu).

Nombre d'angles	50	700
Durée d'élimination de symétries (sec)	0.09	3.34
Taille de la solution *glouton* | Temps de résolution (sec)	6 | 0.00	8 | 0.01
Taille de la solution *recuit simulé* | Temps de résolution (sec)	6 | 0.11	7 | 7.54
Taille de la solution *bruteforce* | Temps de résolution (sec)	5 | 8.67	–
Taille de la solution **optimale** MIP | Temps de résolution (sec)	**5** | 0.01	**6** | 0.06

FIGURE 4.16 – Résultats de l'instance 6 (486x421 et 3 départs de feu).

Nombre d'angles	150	200
Durée d'élimination de symétries (sec)	13.5	25.1
Taille de la solution *glouton* | Temps de résolution (sec)	14 | 0.00	14 | 0.01
Taille de la solution *recuit simulé* | Temps de résolution (sec)	12 | 1.76	12 | 7.47
Taille de la solution *bruteforce* | Temps de résolution (sec)	–	–
Taille de la solution **optimale** MIP | Temps de résolution (sec)	**12** | 0.06	**12** | 0.06

FIGURE 4.17 – Résultats de l'instance 7 (789x1388 et 4 départ de feu).

Nombre d'angles	30	150
Durée d'élimination de symétries (sec)	2.35	48.8
Taille de la solution *glouton* | Temps de résolution (sec)	18 | 0.00	22 | 0.00
Taille de la solution *recuit simulé* | Temps de résolution (sec)	17 | 0.00	20 | 0.02
Taille de la solution *bruteforce* | Temps de résolution (sec)	–	–
Taille de la solution **optimale** MIP | Temps de résolution (sec)	**16** | 0.05	**19** | 0.20

4.3.2 Analyse des résultats

Les tableaux ci-dessus présentent les résultats de nos algorithmes sur nos 7 instances selon le nombre d'angles considéré. On peut relever plusieurs choses de ces résultats. Tout d'abord, on peut noter que l'augmentation du nombre d'angles augmente bien la difficulté de l'instance mais ce n'est pas une fonction convexe. En effet, même si ce n'est pas illustré ici, il est possible dans de rares cas qu'une augmentation du nombre d'angles donne lieu à une instance plus simple. Par exemple, si un départ de feu touche 4 villes avec 5 lignes de feux, rien ne garantit qu'il touche ne serait-ce qu'une seule ville avec 6 lignes de feux. À l'extrême par contre quand le nombre d'angles tend vers l'infini, toutes les villes qui sont touchées avec 5 ou 6 nombres d'angles seront également touchées. Nous avons quand même décidé de faire varier le nombre d'angles mais ceci est à prendre avec des pincettes quand le nombre d'angles est petit.

Bien qu'on aimerais avoir des résultats avec plus d'angles sur chacune des instances pour bien observer toutes les différences en temps, on est en fait limité par plusieurs éléments :

— On a notamment la durée d'élimination des symétries qui augmente avec la taille des instances et le nombre d'angles. Cette étape de réduction est utilisée par tous nos algorithmes sauf le MIP, et puisque que sa durée devient considérablement plus grande que les temps de résolution des algorithmes lorsque la taille du problème augmente (comme on peut le voir sur les tableaux 4.16 et 4.17), on n'arrive pas à bien observer les différences de performances en temps de nos algorithmes car elles se voient éclipsées par les durées d'élimination des symétries.

— Les instances 2 et 3, sont construites de façon à ce que les départs de feu se trouvent à proximité des villes. Bien que ces instances sont réalisables pour des nombres d'angles petits (10 ici), lorsqu'on augmente le nombre d'angles il devient impossible de placer des pompiers pour protéger certaines villes sans que ces pompiers tombent dans la surface de fournaise du feu. Du coup, on ne peut pas tester les instances 2 et 3 sur des plus grands nombres d'angles sans qu'elles deviennent irréalisables.

— Il est inutile d'augmenter le nombre d'angles d'une instance passé son nombre d'angles maximal car les positions possibles des pompiers restent les mêmes ainsi que les temps de résolution. C'est notamment le cas pour l'instance 1 sur le tableau 4.11 où le problème reste le même pour tout nombre d'angles après son nombre maximal d'angles qui est 48.

Pour ce qui est des performances de nos algorithmes, on se servira du modèle mathématique comme point de référence. Le modèle mathématique donne la solution optimale tout comme le *bruteforce*. Ceci nous a permi de vérifier que le *bruteforce* donne bel et bien des solutions optimales. Par contre, l'algorithme *bruteforce* atteint rapidement ses limites. Comme on peut le voir sur le tableau 4.14 de l'instance 4, en passant de 10 à 13 angles le temps que prend le *bruteforce* est presque multiplié par 100. Et ce temps passe de 20 secondes à 57 secondes juste en augmentant le nombre d'angles d'un. Il est clair que cet algorithme explose très rapidement en temps.

Quant au glouton, on peut faire une comparaison avec le modèle mathématique en ce qui concerne la qualité des solutions trouvées. Les solutions que trouve le glouton sont souvent sous optimales, néanmoins elles ne divergent jamais trop et restent cohérentes. Visuellement, il serait impossible de dire si une solution produite par notre algorithme glouton est sous optimale ou non. De plus, l'algorithme glouton est la plus rapide de nos algorithmes et résout le problème presque instantanément sur chacune de nos instances.

Finalement, le recuit simulé trouve un bon compromis entre la rapidité de résolution et l'exactitude de la solution obtenue. Cet algorithme trouve souvent une solution plus proche de l'optimal que le glouton et cela dans un temps raisonnable.

5 Conclusion

Pour conclure, le problème du placement optimal de pompiers se résout rapidement avec le modèle mathématique en utilisant un solveur MIP. Notre seule autre méthode de résolution exacte étant le *bruteforce* explose rapidement en temps à mesure que la taille de l'instance augmente, et se voit donc inefficace. Nos deux méthodes approchées nous donnent des solutions de bonne qualité, l'algorithme glouton favorisant la rapidité comparé au recuit simulé qui se voit plus proche de l'optimal mais avec un temps de calcul légèrement plus long. Néanmoins, ces résultats ne peuvent être admis sans prendre en compte les choix de représentation et d'implémentation du problème qui influent sans doute sur les résultats obtenus. Il serait alors intéressant d'étudier combien ces choix affectent ces résultats et s'ils peuvent eux-mêmes se prêter à l'optimisation.