Rapport TP1 INF4215

Adrien Logut (1815142), Elliot Sisteron (1807165)

Introduction

Le but de ce TP est à partir d'un ensemble de points, trouver un ensemble d'antennes qui couvrent tous les points, et cela en minimisant le coût d'installation des antennes qui est : $\eta(r) = K + Cr^2$ avec K et C constants et r le rayon de l'antenne A. Il a été demandé d'implémenter deux méthodes : une recherche arborescente et une recherche locale.

Le rapport se présente donc en 3 grandes parties : l'approche du problème et les explications nécessaires à la compréhension de nos algorithmes, la recherche arborescente, la recherche locale et enfin les questions.

Table des matières

1 Approche du problème

1.1 Too much to handle

À première vue, le nombre d'états est énorme. En effet, considérons une map de taille $n \times m$.

Alors, il y a nm positions possibles pour une antenne. De plus, cette antenne a aussi un rayon. Considérons le pire cas, i.e. l'antenne est en $(0, 0)$ et le point est en (n, m). Le rayon à utiliser pour englober ce point est donc de $\sqrt{n^2 + m^2}$ (pythagore).

Ainsi, on a déjà nm positions possibles, puis $\lceil\sqrt{n^2 + m^2}\rceil$ rayons possibles. Et cela, uniquement pour une antenne !

Considérons que l'on a p points, soit un nombre maximal de p antennes. Il y a donc $\left(nm\lceil\sqrt{n^2 + m^2}\rceil\right)^p$ états possibles !

Pour une map de taille 10×10 et 3 points, c'est déjà près de 3 milliards d'états à parcourir. Il nous faut clairement réduire ce nombre !

1.2 Problème du cercle minimum

Une bonne façon de simplifier le problème est de considérer un cas simple : supposons que nous avons à notre disposition une seule antenne. Quelle est la meilleure façon de placer cette antenne de sorte à minimiser le coût et à englober tous nos p points ?

Soit $p_1, p_2, \ldots, p_p$ nos p points. Soit r le rayon de l'antenne que l'on va poser. Le coût est $\eta(r) = K + Cr^2$. K et C sont des constantes positives, donc il suffit simplement de minimiser r !

On cherche donc un cercle de centre x et de rayon $r : \mathcal{C}(x, r)$ t.q. r soit minimal et que tous les points soient inclus dans le cercle. Comment s'assurer que tous les points sont inclus dans le cercle ? Il suffit de considérer le point le plus loin du centre de prendre ça comme rayon, i.e $r = \max_{i \in [\![1,p]\!]} d(x, p_i)$, avec d la distance euclidienne.

Le problème devient donc :

$$\inf_{x \in \mathbb{R}^2} \max_{i \in [\![1,p]\!]} d(x, p_i)$$

Pour résoudre ce problème, il suffit de se rendre compte qu'il est facile de trouver le cercle minimum pour 1, 2 ou 3 points :

- Pour un point, il suffit de prendre ce point pour centre et prendre un rayon de 0.
- Pour deux points, on prend le milieu des deux points et le rayon est la taille du segment formé par ces deux points divisé par deux.
- Pour trois points, ils forment un triangle. Dans le cas où le triangle est acutangle (tous les angles de ce triangle sont aigus) on peut prendre le cercle circonscrit. Si ce n'est pas le cas, on peut simplement prendre le plus grand segment et considérer qu'il n'y a que deux points (le troisième tombera forcément dans le cercle).

Résoudre ce problème par force brute consiste à essayer toutes les paires et les triplets de points possibles. Cela donne une complexité en $O(p^4)$, ce qui n'est pas très efficace.

On se rend assez facilement compte que pour que le cercle soit minimum, il faut au moins deux points sur la frontière. Ainsi, si l'on ajoute un point au cercle et qu'il tombe en dehors de la frontière, il faut recalculer le cercle minimum en prenant en compte que ce point sera forcément sur la frontière du nouveau cercle construit.

On peut donc tenter une implémentation récursive (algorithme de Welzl), mais c'est assez difficile à programmer. On va préférer un algorithme itératif à construction incrémentale aléatoire.

On procède de la manière suivante :

Calcul dans le cas général

1. Mélanger les points dont on cherche à calculer le cercle minimum (pour ajouter une notion aléatoire qui nous permettra d'obtenir une meilleure complexité amortie).
2. Prendre les deux premiers points de cet ensemble et considérer que le cercle minimum est celui formé par ces deux points.
3. Pour chaque point, si ce point n'est pas dans le cercle minimum actuellement calculé, recalculer le cercle minimum pour les points déjà parcouru et en considérant que le point courant sera sur la frontière du nouveau cercle (décrit ci-dessous).
4. Retourner le cercle ainsi calculé.

Calcul dans le cas où l'on sait qu'un point est sur la frontière

1. Mélange des points.
2. Prendre le point sur la frontière et le premier point de l'ensemble, considérer le cercle minimum formé par ces deux points.
3. Pour chaque point, si ce point n'est pas dans le cercle minimum actuellement calculé, recalculer le cercle minimum pour les points déjà parcouru et en considérant que le point courant et le point de la frontière seront sur la frontière du nouveau cercle (décrit ci-dessous).
4. Retourner le cercle ainsi calculé.

Calcul dans le cas où l'on sait que deux points sont sur la frontière

1. Mélange des points.
2. Prendre les deux points sur la frontières et le premier point de l'ensemble, considérer le cercle minimum formé par ces trois points.
3. Pour chaque point, si ce point n'est pas dans le cercle minimum actuellement calculé, recalculer le cercle minimum en considérant les deux points de la frontière et le point courant.
4. Retourner le cercle ainsi calculé.

Cet algorithme a une complexité *amortie* linéaire car il est rare que les points ainsi choisi se retrouvent en dehors du cercle. $O(p^3)$ dans le pire des cas, mais il a très peu de chances de survenir.

1.3 Nombre de Bell

On vient de découvrir une propriété du problème qui va nous aider énormément : pour tout ensemble de points, il est possible de calculer directement l'antenne qui minimise le coût. Ainsi, il ne nous suffit plus qu'à savoir comment séparer nos points en ensembles de points et le tour est joué !

Prenons un exemple, on considère que l'on a trois points p_1, p_2 et p_3. On peut donc poser jusqu'à trois antennes. Avec notre propriété, on peut directement mettre une antenne qui englobe p_1, p_2 et p_3. Mais, on pourrait aussi mettre une antenne qui englobe p_1, puis une autre qui s'occupe p_2 et p_3 ensemble. Ou encore p_2 tout seul et p_1 et p_3 ensemble ; p_3 tout seul et p_1 et p_2 ensemble. Et même juste une antenne pour p_1, une pour p_2 et une pour p_3.

On a donc transformé le problème et nos états ne sont finalement que des partitions de notre ensemble de points initial. Combien y'a-t-il de partitions possibles pour un ensemble de taille p ? Il s'agit là du nombre de Bell $B(p)$.

C'est un nombre qui explose à mesure que p grandit, mais beaucoup moins vite que le nombre d'états que l'on avait auparavant. Encore mieux, pour p points et une carte infiniment grande, le nombre d'états reste $B(p)$ et ne dépend pas de la taille de la carte.

Pour reprendre l'exemple où l'on a une carte de taille 10×10 et 3 points, il ne reste plus que $B(3) = 5$ états à parcourir (au lieu de 3 milliards hein...) !

1.4 Valeurs entières

On a vu qu'il existe des algorithmes en $O(p)$ pour calculer le cercle minimum. Ce cercle a des coordonnées réelles, or nous devons travailler en coordonnées entières. Finalement, cette simplification nous complexifie beaucoup la tâche car nous devrons convertir le cercle trouvé en coordonnées entières tout en gardant la propriété de cercle minimum. En effet, on pourrait tout à fait tout faire en valeurs réelles sans changer la difficulté du problème vu que la difficulté de notre problème se mesure directement en fonction du nombre de points (et uniquement le nombre de points).

Pour ce faire, étant donné le centre en valeurs réelles (x, y), il existe plusieurs valeurs possibles pour le centre en valeur entières, nous en testerons 9 encadrant le centre réel (i.e. toutes celles dans un carré de 3×3 encadrant le centre original). Il suffit de prendre le centre qui minimise le rayon du cercle.

On notera aussi qu'une antenne entière est de rayon 1 *au minimum.*

1.5 Représentons l'état

Il nous suffit simplement de représenter l'état comme une liste de points à couvrir ainsi qu'une liste des ensembles déjà couverts. Pour ce faire, on va créer une classe *Antenne* qui est constitué d'un centre, d'un rayon et d'une liste de points couverts. On peut soit ajouter des points à l'antenne et donc recalculer le cercle minimum ou bien recréer une nouvelle antenne.

Notre état est donc une liste de points à couvrir et une liste d'antennes déjà posées.

2 Recherche arborescente

Ok bon c'est bien nice tout ça mais on fait comment faire une recherche arborescente sur les partitions... ?

Considérons un ensemble de points $P = \{p_1, p_2, \dots, p_p\}$. On voit bien qu'il existe deux types de partitions, les partitions du genre :

$$X = \{\{p_1\}, \dots\}$$

Et celle qui ressemblent à :

$$X = \{\{p_1, \dots\}, \dots\}$$

Autrement dit, pour un point donné, on a deux choix qui s'offrent à nous :

1. Lui mettre une antenne pour lui tout seul
2. Partager son antenne avec des copains à lui

On démarre donc en posant aucune antenne. À chaque étape de notre descente arborescente, on choisit un point qui n'est pas englobé par une antenne. Soit on pose une antenne pour lui tout seul, soit on vient fusionner ce point là avec une antenne déjà posée. On a une feuille quand tous les points sont couverts.

Voilà un exemple pour $p = 3$:

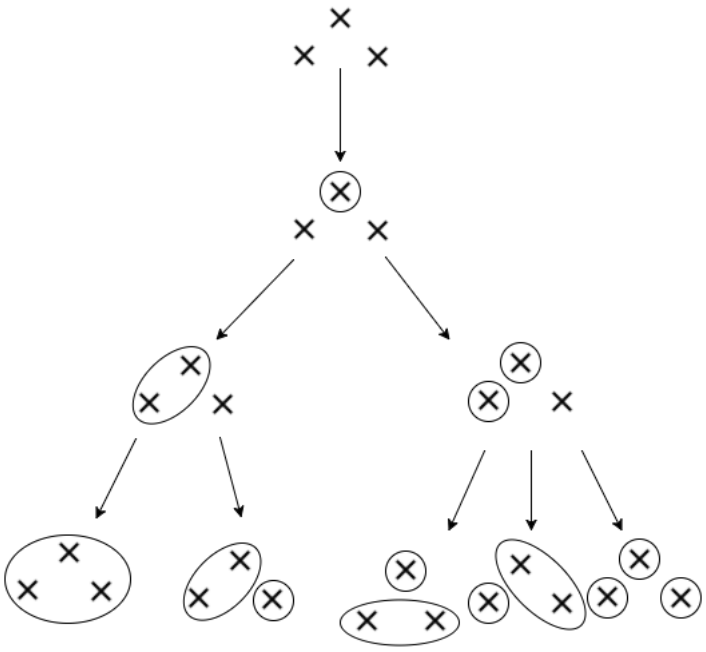

FIGURE 1 – Exemple pour $p = 3$

3 Recherche locale

3.1 Algorithme de recuit simulé

Et la recherche locale alors ?

La recherche locale est une technique qui permet de ne pas visiter tous les états mais de faire passer le système à un état voisin à partir d'une configuration donnée. Cependant, contrairement à la recherche arborescente, la recherche locale ne garantie pas de trouver une solution optimale. Il faut donc être vigilant sur les choix de paramètres et faire de nombreux jeux de tests pour trouver la solution optimale dans la plus grande majorité des cas.

Pour la recherche locale de ce TP, la technique mise en oeuvre est celle du *recuit simulé* (ou *Simulated Annealing* dans la langue de Shakespeare). Cette méthode est en fait inspirée de la métallurgie, faire doucement baisser la température pour faire refroidir un métal, et le faire atteindre un état avec une énergie globale plus faible. L'analogie est facile à faire ici avec la physique. La température est une mesure de l'agitation des molécules. Une température élevée permet plus de chocs, une température faible réduit le nombre de chocs. Après cet intermède physique, comment est-ce qu'on peut appliquer cette technique dans notre problème ?

Prenons comme énergie du système, le coût global de nos antennes dans l'état actuel du système. On cherche bien à minimiser cette valeur, on va donc essayer de faire "baisser" la température pour obtenir un niveau d'énergie le plus bas possible.

Le principe est donc simple. On part d'une subdivision de notre ensemble de points, chaque subdivision représentant une antenne (définie de la même manière qu'avec la recherche arborescente : *Problème du cercle minimum*). On a ensuite deux choix qui s'offrent à nous : Soit on fusionne deux ensembles pour donner un ensemble plus gros. Soit on divise un ensemble en deux pour obtenir deux ensembles plus petits. Après cette action, on regarde la nouvelle énergie du système et on la compare à l'ancienne énergie, celle juste avant l'action. Si elle est plus basse on la prend, sinon on la prend avec une probabilité qui dépend de la température actuelle. Plus la température est élevée, plus il est probable qu'on choisisse un état qui est "moins bon" que le précédent. L'agitation est plus élevée. (On définira les formules un peu plus loin). Puis au bout d'un nombre fixe d'itérations, on fait baisser la température.

A l'aide de cette technique, on peut passer par des états intermédiaires qui ne sont pas considérés comme bons mais qui permettent de passer à un état meilleur qu'au début. Avoir une température élevée au début permet de faire sortir le système d'un minimum local mais pas global, alors qu'une température faible permet d'être plus précis pour affiner sur la fin.

Mettons donc en équation, et récapitulons les variables :

— T_0 : La température initiale du système

— λ_T : Le coefficient de baisse de température

— k_{step} : Le nombre d'itérations avant de baisser la température

— k_{max} : Le nombre d'itérations au total que "tourne" notre algorithme

La fonction de changement de température :

$$f(T,k) = \begin{cases} \lambda_T * T & \text{Si } k \bmod n = 0 \\ T & \text{Sinon} \end{cases}$$

La fonction de probabilité :

$$P(T, \Delta_E) = \begin{cases} 1 & \text{Si } \Delta_E < 0 \\ e^{\frac{-\Delta_E}{T}} & \text{Sinon} \end{cases}$$

On a donc tout défini, reste à savoir comment on choisit quelle action faire et comment. Et bien c'est simple, c'est totalement aléatoire (et c'est ça la beauté de la chose). On a une probabilité de $\frac{1}{2}$ pour fusionner deux ensembles et $\frac{1}{2}$ pour fissionner un ensemble en deux. La taille des deux ensembles fissionnés est aussi aléatoire. Bien sûr on vérifie qu'on peut fusionner (que la subdivision actuelle ne contient pas tous les points) ou qu'on peut fissionner un ensemble (que la subdivision ne soit pas déjà un singleton).

3.2 Résultats intermédiaires

Des tests automatisés ont été effectués pour déterminer des bons paramètres. Si l'on veut de bon résultats, il faut laisser l'algorithme tourner plus longtemps, et faire baisser la température pas trop vite. On obtient de très bon résultats en peu d'itérations avec 3 points. On a plus de mal pour 5 points, et pour 10 points la meilleure solution approximée (car on est jamais sûr que c'est la meilleure solution) est trouvée que dans 10% des cas. C'est trop peu. Il faut donc une amélioration

Vous pouvez trouver un tableur Excel avec toutes les données et résultats : TabWithoutRandomRestart.xlsx. C'est indigeste par contre...

3.3 Random restart

Le fait de ne pas trouver toujours la bonne solution montre que notre algorithme n'arrive pas à se sortir des minimums locaux. Il reste coincé et ne peut aller explorer une solution beaucoup plus éloignée. Pour palier ce problème, une solution est possible. Si on remarque qu'on ne bouge pas de notre minimum actuel sur k_{random} itérations, on enregistre la solution actuelle si elle est meilleure que la solution précédemment enregistrée et on fait un *random restart*. On recommence tout simplement l'algorithme avec une subdivision aléatoire nouvelle. Si on se base sur les schémas vu en cours, ça permet de se sortir de "petits puits" que sont les minimum locaux.

Bien sûr on rajoute un paramètre supplémentaire à notre problème :

— k_{random} : Nombre d'itérations à énergie constante avant random restart

3.4 Résultats finaux

Pour l'instant les tests ne sont pas aussi poussés en termes de paramètres, mais on remarque une amélioration nette des résultats. En faisant tourner l'algorithme avec ce jeu de paramètres :

— $T_0 = 10$
— $\lambda_T = 0.99$
— $k_{step} = 10$
— $k_{max} = 10000$
— $k_{random} = 15$

On obtient proche de 100% de réussite, pour un temps d'exécution d'une seconde, sur 10 points. Plus on augmente le nombre de points, plus le nombre d'itération doit être élevé (pour explorer assez de solutions).

Bon ben on est tout bon, c'est partit(ion) pour les questions !

4 Questions

4.1 Question 1

```
1 def fct(predList, inputList):
2     return filter(lambda x: all([f(x) for f in predList]), inputList)
```

Cette fonction prend deux paramètres :

1. Une liste de prédicats *predList*, il s'agit une liste de fonctions booléennes.
2. Une liste d'éléments *inputList*.

Ici, la fonction *filter* va retourner une liste qui sera une partie de notre *inputList*. La fonction *filter* prend en argument une fonction (qui est en fait un prédicat) et une liste à filtrer. On lui donne une fonction lambda en paramètre qui consiste à faire un *and* sur tous nos prédicats (*all* fait une opération *and* sur tous les prédicats).

Soit x un élément de notre *inputList*. Ainsi, pour faire partie de la liste retournée, x doit vérifier tous les prédicats de *predList*.

Prenons un exemple :

```
1 import math
2
3 def is_prime(x):
4     for i in range(2, int(math.ceil(math.sqrt(x))) + 1):
5         if (x%i == 0):
6             return False
7     return True
8
9 fct([lambda x: x >= 8, lambda x : x <= 17, is_prime], [2, 7, 8, 10, 9, 14, 1, 3, 11, 12, ↩
      17])
```

Cet appel retourne bien :

```
[11, 17]
```

À noter que la fonction *fct* n'est pas optimale. Ici, elle fait un appel à *all* en lui passant une liste en paramètre, c'est mieux d'utiliser un générateur car cela évite d'utiliser de la mémoire pour rien. Si *predList* est de taille n et *inputList* est de taille m, l'appel à *fct* demande $O(n+m)$ en mémoire.

Le générateur est paresseux et donc ne générera pas toute la liste d'un coup, il fera l'équivalent d'une boucle et économisera donc la mémoire. Avec un générateur, on passe ainsi à $O(m)$ en complexité spatiale (pour stocker le résultat de *filter*).

Voici une version optimisée :

```
1 def fct(predList, inputList):
2     return filter(lambda x: all(f(x) for f in predList), inputList)
```

4.2 Question 2

Discutons nos solutions. Dans le cas de la recherche arborescente, on a la garantie de trouver le minimum. Mais, cela se paye au niveau de la complexité : elle est exponentielle en fonction de p (nombre de Bell). Autrement dit, elle est à privilégier pour jusqu'à une vingtaine de points. Elle marche aussi très bien dans le cas où les points sont regroupés tous ensemble dans la solution optimale (et aussi dans le cas inverse : où il y a p antennes pour p points). On se la met de côté donc pour des petits exemplaires et des cas triviaux.

Pour la recherche locale, la complexité est presque constante : on fait un nombre fini d'itérations (on peut même la faire s'arrêter après un certain temps à la place, rendant la complexité constante). Mais, elle ne trouve pas forcément la solution optimale. Force est de constater qu'elle est quand même rudement efficace et que le recuit simulé est un algorithme qui fonctionne bien. Il n'en est pas moins que nous n'avons aucun garantie quant à la solution retournée. On va donc privilégier la recherche locale pour de gros exemplaires ($p > 20$) non triviaux.

Quels sont les avantages de notre représentation de l'état ?

— On réduit considérablement le nombre d'états parcourus comparé au problème initial.
— La solution optimale est rapidement trouvée pour de petits exemplaires. Pour de gros exemplaires, la recherche locale semble fonctionner correctement (il nous aurait fallu des gros exemplaires résolus pour en être certain).
— L'état est facilement modulable et permet aussi de résoudre le même type de problème avec des contraintes (par exemple : nombre d'antennes limité, points non-accessibles...). C'est une fonctionnalité importante car ce type de problème apparaît avec des contraintes dans la vraie vie.

Quels en sont les inconvénients ?

— Difficile à implémenter (un algorithme linéaire pour le problème du cercle minimum n'est pas chose facile à mettre en oeuvre). De plus, nous n'avons pas implémenté le meilleur algorithme (Welzl).
— Assez volumineux à stocker si on garde en mémoire tous les états.
— Probablement pas la solution la plus efficace, il existe sûrement une ou plusieurs manières de faire qui sont plus intéressantes.
— Nous n'avons pas eu le temps de faire une heuristique A* et c'est bien dommage, nous avons essayé d'y réfléchir mais soit nous trouvions des choses inutiles, soit nous trouvions une heuristique non-admissible.
— Le fait d'avoir utilisé l'algorithme générique de Michel pour la recherche arborescente nous fait perdre en efficacité car c'est une solution qui n'est pas assez ciblée au problème (notion de frontière, noeud, état, ...). Avec une recherche arborescente directement implémentée (sans passer par toutes ces classes), on aurait probablement pu faire mieux (mais c'est quand même rudement pratique d'utiliser les fonctions déjà faites).
— Made in python, donc relativement lent comparé à du C/C++ ou de l'assembleur (enfin bon, en assembleur faut le vouloir quand même).