Département de génie informatique et de génie logiciel

INF4215

Introduction à l'intelligence artificielle

Laboratoire #1

Éric Morissette, #1631103

Sacha Licatese-Roussel, #1635849

14 février 2016

École Polytechnique de Montréal

Introduction

Pour le premier laboratoire, il nous fallait écrire deux petits programmes Python. Ces deux programmes ont le même but: prendre plusieurs points dans un plan cartésien et deux valeurs reliées au coût, soient le prix d'une nouvelle antenne (K) et le prix par rayon de distribution au carré (C).

$$Coût = K + Cr^2$$

L'algorithme en question est de positionner un certain nombre d'antennes afin de couvrir tous les points tout en minimisant le coût total de l'infrastructure.

Pour ce faire, nous devons implémenter deux méthodes de recherches différentes. La première solution est une recherche en arborescence et la deuxième est une recherche locale.

Choix d'implémentation

Solution #1 : Recherche arborescente

Pour cette première partie du travail nous avons défini notre état initial comme étant le positionnement d'une première antenne sur n'importe lequel des points à couvrir. Ensuite, pour ce qui est des transitions entre les différents états, nous les avons définis comme étant, soit l'ajout d'une nouvelle antenne à la topologie ou bien le déplacement d'une antenne existante, sous certaines conditions.

Pour ce faire, nous avons défini notre concept d'antenne de manière à les représenter à l'aide d'un rayon commun. En effet, lors de leur initialisation, nous déterminons, par-rapport aux données de la simulation, la longueur maximale du rayon qui s'avère la plus avantageuse comparé au coût d'ajout d'une antenne supplémentaire.

$$ceil(\sqrt{(K/C)})$$

Comme vous pouvez voir, cette équation nous permet de maximiser l'utilisation de nos antennes lors de leur initialisation et, aussi, prend en compte le fait que l'on doit uniquement représenter le rayon sous forme d'entier (Ceil).

Par la suite, on recherche parmis les points non desservis celui le plus proche de notre antenne actuellement traitée. Ensuite, si la distance entre notre antenne et ce point est plus plus petite ou égale à deux fois le rayon maximal de l'antenne actuelle, alors il nous est possible de déplacer cette antenne de manière à desservir ce dernier point en plus de ceux qui étaient précédemment desservis par cette antenne. Si jamais la comparaison n'est pas vérifiée ou que le déplacement de l'antenne serait détrimental pour les points déja recouverts, on crée alors une nouvelle antenne à la position du point inaccessible et on fait de cette nouvelle antenne notre focus pour l'itération suivante.

On boucle donc sur cet algorithme tant et aussi longtemps que tous les points n'aient pas été recouverts.

Finalement, après l'étape de recouvrement, on optimise nos antennes en vérifiant, pour chacune d'entre-elles, si il est possible, tout dépendamment de son contenu, de rétrécir son rayon, puisque celui-ci avait été initialisé à sa valeur maximale tel qu'expliqué plus haut.

On peut donc voir que cet algorithme représente une recherche arborescente en profondeur et donc, que l'algorithme assure une solution au problème étant donné l'absence de boucle dans notre graphe d'états. Par-contre, celle-ci ne sera pas necéssairement la solution optimale au problème.

Solution #2 : Recherche locale

Le deuxième algorithme, celui de recherche locale, est un peu similaire à celui de recherche arborescente mais est plus axé sur une maximisation de l'efficacité de chaque étape de transformation de l'état courant.
Pour commencer, on prend tous les points à couvrir et nous plaçons une antenne directement sur chacun d'entres eux. Aussi, nous créeons chaque antenne avec un rayon rMax, qui est calculé en fonction de K et de C:

$$rMax = ceil(\sqrt{(K/C)})$$

Cela est l'état initial à partir duquel nous allons itérer dans l'algorithme afin d'obtenir une solution potentiellement optimale.

Une fois que l'étape initiale est effectuée, nous allons prendre chaque antenne et la comparer avec chaque autre antenne afin de calculer:

- Est-ce que les zones des deux antennes se recouvrent?
- Est-ce que l'ensemble de points couverts par chacune des antennes peut être recouverte par une seule antenne de même rayon rMax?
- Est-ce que la distance entre les deux antennes est la plus petite distance entre deux antennes compatibles? Par "antennes compatibles", nous parlons de deux antennes qui répondent aux questions 1 et 2.

Une fois que nous avons la paire d'antennes compatibles qui sont les plus près l'une de l'autre, nous les joignons ensemble afin d'obtenir une seule antenne qui recouvre l'ensemble des points couverts par chacune des antennes précédentes. Cette étape est la transformation qui nous rapproche le plus rapidement possible de l'état final. Si nous ne trouvons plus aucunes antennes qui se recouvrent ou s'il est impossible de joindre deux antennes recouvrantes, alors il ne nous reste qu'à réduire au maximum notre rayon tout en gardant tous les points à servir.

Cet algorithme est basé sur des "itérations à meilleur amélioration" ce qui veut dire que chaque itération regroupe deux antennes avec le joint le plus efficace et profitable à la solution. Il peut aussi être qualifié de recherche locale car on commence avec un état initial où chaque point est recouvert par une antenne et on exécute plusieurs itérations de l'algorithme afin de maximiser l'efficacité de la solution finale.

Questions

Question 1 :

Soit le code suivant :

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

Expliquez ce que fait cette fonction et fournissez un exemple utilisant cette fonction.

Pour commencer l'analyse de ce code, nous allons le séparer couche par couche afin de comprendre ce qui ce passe.

Tout d'abord, on voit très bien qu'on défini une fonction `fct` qui prend deux arguments, `predList` et `inputList`, nous allons voir plus loin qu'est-ce qu'ils représentent. On peut aussi voir que la fonction fait un return sur le résultat d'une fonction `filter`:

```
filter(lambda x: all([f(x) for f in predList]), inputList)
filter(function, iterable)
```

La fonction `filter` prend une fonction `function` et l'applique sur chaque élément dans la liste `iterable`. Si le résultat de la fonction est True, alors l'objet sur lequel on a effectué le test sera ajouté dans une nouvelle liste. Lorsqu'on a passé la liste `iterable` au complet, on retourne seulement les éléments qui ont satisfaient la fonction `function`.

Maintenant, prenons seulement le `lambda`, qui crée une fonction sans nom qui est référencée par l'application du filtre sur la liste `inputList`:

```
lambda x: all([f(x) for f in predList])
```

Ensuite, séparons le lambda en deux parties, la déclaration et la valeur retournée ainsi que le corps de fonction du `lambda`:

```
lambda x:
all([f(x) for f in predList])
```

Le `lambda` effectue la fonction `all` qui prend tous les éléments dans la liste `predList` reçue en argument et regarde si leur valeur est True et/ou existante. S'il y a au moins un élément nul ou inexistant, alors la fonction retourne False, sinon elle retourne True. Si on regarde l'intérieur de la fonction `all`, on voit qu'on crée une liste avec une boucle "for in":

```
[f(x) for f in predList]
```

Cette boucle `for in` va prendre la liste `predList`, qui contient des fonctions et on crée un itérateur `f` à partir duquel on va appeler la fonction `f(x)` avec l'argument du `lambda` plus haut. Les résultats des fonctions `f(x)` sont mis dans une liste.

Donc, si on remonte:
1- On applique toutes les fonctions "f(x)" se trouvant dans la liste `predList`.
2- On regarde si tous les résulats existent et sont non nuls.
3- On encapsule les étapes 1 et 2 ensemble dans une fonction `lambda`.
4- On crée une liste comportant tous les éléments de la liste `inputList` qui ont passé le test de la fonction obtenue à l'étape 3

Exemple de code utilisant la fonction `fct`:

```
predList = []
predList.append(lambda x: x >= 1)
predList.append(lambda x: x <= 5)

inputList = [5, 20, 30, 1, 32, 3, 4, 62, 2, 6]

print(list(fct(predList, inputList)))
```

Ce code crée une liste de fonctions lambdas qui demandent à ce qu'un chiffre soit entre 1 et 5 inclusivement. Ensuite on crée une liste de nombres sur lesquels on veut appliquer la liste de fonctions de test. Le résultat est:

```
[5, 1, 3, 4, 2]
```

Ce qui représente tous les nombres qui passent l'ensemble des tests dans `predList`.

Question 2 :

Quelles sont les points forts et les faiblesses de vos implémentations ?

Pour ce qui est de la première approche, le recherche arborescente, on remarque que l'utilisation d'une fouille en profondeur limitera l'utilisation mémoire. En effet, puisque notre condition d'arrêt se situe sur la couverture complète des points, que le graphe représentant la situation n'est pas cyclique et est fini, on ne verra pas d'explosion de l'utilisation de la mémoire comme on verrait dans une recherche de type arborescente en largeur. Cependant, on remarque que l'utilisation d'une recherche en profondeur, bien que visiblement efficace dans ce type de problème et dans cet ordre de grandeur, ne produira pas nécessairement la solution optimale au problème. Pour parer à cette observation, on pourrait se diriger vers une implémentation d'un algorithme de type A* afin de pouvoir améliorer la solution au fur et à mesure de l'algorithme à l'aide d'une heuristique en plus du coût, plutôt que de se limiter à la recherche en profondeur toute simple.

Passons maintenant à notre seconde implémentation, c'est-à-dire la recherche locale. Celle-ci offre, théoriquement, une solution optimale par-rapport à l'implémentation précédente. Par-contre, on a remarqué en pratique que le temps d'exécution est plus élevé que notre solution en recherche arborescente. Une amélioration potentielle à ce second algorithme pourrait être faite au niveau de l'ajout d'une heuristique de construction afin d'essayer de déterminer un état de départ plus proche de la solution optimale, plutôt que de toujours démarrer l'algorithme avec une antenne sur chacun des pôles à couvrir. De cette manière, on pourrait espérer réduire le temps d'exécution, à condition d'avoir une heuristique intéressante.

Compétition

Pour la compétition, nous désirons utiliser la solution 1, soit la recherche arborescente.

Conclusion

En conclusion, dans ce laboratoire nous avons débuté nos expérimentations quant aux notions acquises dans ce cours d'introduction à l'intelligence artificielle. Ceci nous à permis de nous familiariser avec l'implémentation de certain types d'algorithmes de recherche dans un espace d'états. De plus, cet exercice à aussi permis de souligner l'importance de définir clairement l'état initial, les transitions et les conditions de succès lors de la modélisation et de la représentation d'un problème dans un espace d'états.