Recherche heuristique dans les graphes d'états

Références Bibliographiques

Dans ce cours je suis largement les chapitres de référence suivants:

> - *Recherche heuristique dans les graphes d'états*, Henry Farreny, In Panorama de l'IA, vol. 1

- *Solving problems by Searching* (Chapter 3) et *Informed (Heuristic) Search Strategies* (Chapter 4), Stuart Russel and Peter Norvig. In Artificial Intelligence, A Modern Approach.
- Le code est également inspiré de la structure de http://aima.cs.berkeley.edu/python/search.html

Je recommande aussi très fortement les deux sites:

- Red Blob Games (https://www.redblobgames.com/pathfinding/a-star/introduction.html) de Amit Patel
- la page de Nathan Sturtevant (https://www.movingai.com/index.html) qui contient également de nombreuses animations et vidéos illustrant les algorithmes de recherche.

1. Graphes d'états

On considère de manière générale le problème consistant à trouver un chemin de coût minimal pour aller d'un **état initial** à un **état but** dans un **graphe d'états**.

- Passer d'un état à un autre à un **coût**.
- Une solution est **optimale** si elle permet de passer de l'état initial à l'état but avec un coût minimal.
- Un paramètre important permettant d'apprécier la difficulté d'un problème est le **facteur de branchement** du graphe d'état (le nombre de successeurs possibles à chaque état).

La taille des problèmes rend parfois impossible la création exhaustive du graphe d'état en mémoire.

> Par exemple, le problème du taquin de taille *n x n* donne lieu à *(n^2)!* états possible. Pour vous convaincre de la difficulté du problème, vous pouvez générer quelques taquins au hasard et essayer de trouver la solution. Dans le code ci-dessous, la case 0 représente la case vide. Les coups possibles sont le déplacement d'une tuile sur la case vide. Note: en générant deux taquins au hasard, vous avez une chance sur deux de tomber sur un problème réalisable. Voyez-vous pourquoi?

In […
```
%pylab inline
import numpy as np
```

In […
```
tiles1 = np.random.permutation(range(0,16)) # 0 is the empty slot
tiles2 = np.random.permutation(range(0,16)) # 0 is the empty slot
puzzle1 = np.array(tiles1)
puzzle2 = np.array(tiles2)
puzzle1 = np.reshape(puzzle1,(4,4))
puzzle2 = np.reshape(puzzle2,(4,4))
print("Taquin initial:\n", puzzle1)
print("Taquin but:\n", puzzle2)
```

> Quel est le facteur de branchement pour un problème de taquin?

Critères de comparaison

Suivant (Russel & Norvig, 2003), ces différentes approches peuvent se comparer en terme de :

- **complétude**: l'algorithme trouve-t-il une solution (si une existe)?
- leur **qualité**: la solution trouvée est-elle optimale?
- **complexité de calcul**: le temps de calcul nécessaire pour obtenir la solution.
- **complexité en espace**: l'espace de stockage nécessaire pendant le calcul.

2. Rappels: Recherche non informée

Le graphe est parcouru en étendant les noeuds. Etendre un noeud consiste à générer ses successeurs dans le graphe d'état. On peut donc distinguer la **frontière**, i.e. les noeuds qui ont été générés et qui sont candidats à être étendus à l'itération suivante, et la **réserve**, i.e. les noeuds qui ont déjà été étendus. On note ici:

- *b* le facteur de branchement du problème,
- *d* la valeur de la solution la moins profonde

Il existe plusieurs types possibles d'exploration, qui se distinguent selon la stratégie d'expansion des noeuds.

- **depth-first** (DFS, profondeur d'abord): on étend le noeud le plus profond de le frontière. Noeuds à stocker: $\mathrm{O}(\mathrm{bd})$. Problèmes de terminaison.

- **breadth-first** (BFS, largeur d'abord): tous les successeurs d'un noued sont étendus. Noeuds à stocker: $O(b^{d+1})$
- **uniform-cost**: on étend le noeud avec le coût de chemin le plus faible jusqu'à présent. Si tous les coûts sont de les mêmes, on se retrouve avec un BFS.

3. Recherche heuristique informée

Idée: utiliser des connaissances du domaine pour guider la recherche. Supposons que l'on puisse estimer le coût depuis le noeud n jusqu'à un état but, et notons $h(n)$ ce coût.

La première idée est de guider la recherche en étendant le noeud n qui possède la valeur de $h(n)$ la plus faible. C'est l'algorithme **greedy best-first**. L'algorithme A* améliore cette idée.

3.1. Algorithme *A* *

Il s'agit d'un algorithme de type best-first, qui étend en priorité le noeud de la frontière avec le meilleur score *f(n)*

```
f(n) = g(n) + h(n)
```

avec

- *g(n)* est le coût du chemin jusqu'au noeud *n*
- *h(n)* est le coût (estimé) de *n* jusqu'à un état but.

Propriétés des heuristiques

Admissibilité

- une heuristique est **admissible** lorsqu'elle ne surestime jamais la distance à l'état but (on parle aussi d'heuristique minorante)

Sous l'hypothèse d'admissibiilité de l'heuristique, *A* * retourne une solution optimale.

Consistence

- une heuristique est consistante ssi pour tout état m et tout fils de n de m, on a:

$$h(m) \leq cost(m, n) + h(n)$$

Autrement dit, les valeurs de $f(n)$ sont non-décroissantes sur les chemins depuis la racine.

Implémentation

On va utiliser deux structures de données:

- la **frontière**, qui stocke les noeuds candidats à être étendus; et

- la **réserve**, qui garde en mémoire les noeuds déjà étendus. C'est important pour éviter de ré-étendre des noeuds déjà visités.

Gestion de la frontière grâce à un tas (permettant l'insertion rapide et garantissant que l'élement racine est toujours celui de meilleure priorité). En Python on peut utiliser le module heapq.

In […

```
import heapq

q=[]
heapq.heappush(q,2)
heapq.heappush(q,5)
heapq.heappush(q,3)
heapq.heappush(q,1)
h = heapq.heappop(q)
h = heapq.heappop(q)
print (q)
print(h)
```

Pour la gestion de la réserve, il est important de pouvoir accéder rapidement aux éléments. Une table de hachage (dictionnaire en Python) est un choix judicieux ici. Au final, la boucle principale de l'algorithme est la suivante:

In […

```
def astar(p):
    """ application de l'algorithme a-star sur un probleme donné
        """
    nodeInit = Noeud(p.init,0,None)
    frontiere = [(nodeInit.g+p.h_value(nodeInit.etat,p.but),nodeInit)]
    reserve = {}
    bestNoeud = nodeInit

    while frontiere != [] and not p.estBut(bestNoeud.etat):
        (min_f,bestNoeud) = heapq.heappop(frontiere)
    # Suppose qu'un noeud en réserve n'est jamais ré-étendu
    # Hypothèse de consistence de l'heuristique
    # ne gère pas les duplicatas dans la frontière

        if p.immatriculation(bestNoeud.etat) not in reserve:
            reserve[p.immatriculation(bestNoeud.etat)] = bestNoeud.g #maj d
            nouveauxNoeuds = bestNoeud.expand(p)
            for n in nouveauxNoeuds:
                f = n.g+p.h_value(n.etat,p.but)
                heapq.heappush(frontiere, (f,n))
    # Afficher le résultat
    return
```

Problème de ré-expansion de noeuds

Important: si l'heuristique utilisée est consistante, alors un noeud un sommet développé par *A* * ne peut l'être qu'une fois. Autrement dit, les noeuds qui passent en réserve ne repassent jamais dans la frontière. (Cette hypothèse est faite dans le code ci-dessus)

Lorque l'heuristique n'est pas consistante, ce n'est pas le cas et on peut avoir à ré-étendre des noeuds...

> Dans quelle situation faut-il replacer un noeud en frontière?

3.2. Iterative Deepening A*

Il faut noter que A, lorsqu'il étend un noeud, génère **tous ses fils**. C'est donc potentiellement très gourmand en espace mémoire.

Pour éviter ce problème, *IDA* * n'étend qu'un seul de ses fils.

Idée: au lieu de stocker la frontière, on va ré-effectuer les calculs. Le **seuil** à partir duquel on ré-entend les noeuds depuis la racine sera augmenté incrémentalement. Ce seuil est initialisé à la valeur *h* de la racine.

4. Trouver les bonnes heuristiques

Notons tout d'abord que l'**heuristique nulle** est bien une heuristique admissible.

> A quoi correspond alors l'algortihme *A* *?

Des heuristiques classiques pour les problèmes de navigation/recherche de chemin (*path-finding*) sont:

- distances de Manhattan (nombre de cases à parcourir sur un monde de type grille)
- distance euclidienne (distance à vol d'oiseau)

> Quelles heuristiques sont envisageables pour le taquin?

Une première idée peut être de compter le nombre de pièces mal placées entre le taquin initial et le taquin but. Appelons cette heuristique `pieces`. Peut-on trouver une meilleure heuristique? (Mais comment peut-on exactement définir ce qu'est une heuristique meilleure qu'une autre?)

In […
```
import probleme
import taquin
```

In […
```
puzzle1 = np.array(([2,1,6,4,0,8,7,5,3]))
puzzle2 = np.array(([1,2,3,8,0,4,7,6,5]))

puzzle1 = np.reshape(puzzle1,(3,3))
puzzle2 = np.reshape(puzzle2,(3,3))
```

In […
```
p1 = taquin.ProblemeTaquin(puzzle1,puzzle2,'pieces')
print ("=-----------------------------=")
print ("Heuristique:",  p1.heuristique)
print ("Etat initial:\n", puzzle1)
print ("Etat but:\n", puzzle2)
print ("=-----------------------------=")
print ("Solution:\n")
probleme.astar(p1,True,False) # premier booleen pour mode verbose, deuxieme
```

In […
```
p1 = taquin.ProblemeTaquin(puzzle1,puzzle2,'manhattan')
print ("=-----------------------------=")
print ("Heuristique:",  p1.heuristique)
print ("Etat initial:\n", puzzle1)
print ("Etat but:\n", puzzle2)
print ("=-----------------------------=")
print ("Solution:\n")
probleme.astar(p1,True,False)
```

In […
```
p1 = taquin.ProblemeTaquin(puzzle1,puzzle2,'manhattan')
print ("=-----------------------------=")
print ("Heuristique:",  p1.heuristique)
print ("Etat initial:\n", puzzle1)
print ("Etat but:\n", puzzle2)
print ("=-----------------------------=")
print ("Solution:\n")
probleme.idastar(p1,True,True)
```

Pattern Databases

Idée: on s'intéresse au coût exact de sous-problèmes de notre problème général, qui consistuent des bornes inf du coût du problème général.

On créé donc une base de donnée avec le coût exact correspondant à toutes les configurations possibles du sous-problème. Pour cela, on part du but et on calcule le coût de chaque configuration rencontrée.

5. Recherche sous-optimale

Dans de nombreuses situations, on peut souhaiter trouver une solution rapidement même si elle est sous-optimale. Dans ce cas, il est souhaitable d'apporter des garanties sur la perte d'optimalité. Un algorithme classique se basant sur cette idée est le Weighted-A* (Post, 1970).

L'idée consiste à utiliser l'heuristique $\mathrm{f(n) = g(n) + w \times h(n)}$.

La conséquence est de pénaliser avec le facteur w l'évalutaion h, l'algorithme se comporte donc de manière plus gloutonne, et trouve en général une solution plus rapidement. A l'extrême (quand le pois devient très grand), l'algorithme ne considère en fait que l'heuristique et devient donc un greedy Best-First.

L'intérêt de Weighted A* est de garantir que la solution se trouve au pire à un facteur w de la solution optimale.

Version du 2022-02-16

In […