Intelligence artificielle
Plus court chemin, avec un algorithme A^* (A-étoile)

N. Durand, D. Gianazza

Intelligence artificielle:

- Implantation d'un algorithme A^* générique
- Application à un problème de plus court chemin

Notion de programmation Ocaml (pas l'objectif principal):

- Modules et foncteurs
- Génericité, polymorphisme
- Interfaces (`.mli`), types abstraits
- Librairies, Makefile
- Générateur de documentation (`ocamldoc`)

Comment démarrer ?

- Télécharger le fichier `A_star_lab.zip` (or `.tgz` or `.gz`) sur `http://e-campus.enac.fr` (cours IP-403 – Intelligence Artificielle) et le décompresser (`unzip`, ou `tar -zxf`, ou `gunzip`). Ceci créera un sous-répertoire `A_star_lab` sous votre répertoire courant.
- Compilez la librairie A^* : allez dans le répertoire `A_star_lab` et entrez la commande `make`. Ceci crée les fichiers de librairies `a_star.cma` and `a_star.cmxa` et une documentation HTML (dans le sous-répertoire `doc`).
- Compilez l'exemple d'application de l'A^* au problème de plus court chemin : allez dans le sous-répertoire `examples/PathFinder/` et compiler avec `make`. Ceci crée un exécutable `findpath` et une autre documentation HTML (dans le sous-répertoire `doc`) de `examples/PathFinder/`.
- Vous pouvez lire la doc HTML avec votre browser web préféré (`file:///__YourPath__/doc/index.html`).
- Vous pouvez exécuter le programme exemple avec la commande `./findpath` (voir la doc pour les options). Vous devriez obtenir une fenêtre graphique comme celle de la Figure 1.

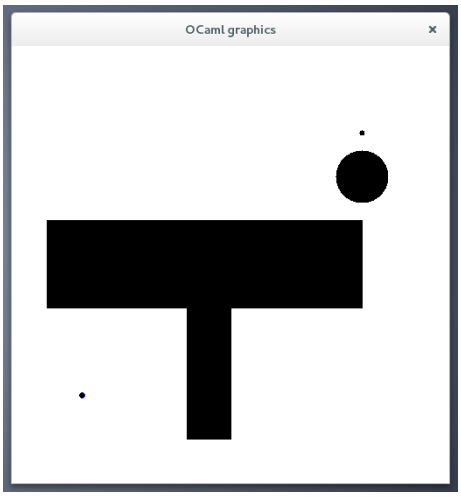

Figure 1: Calculez le plus court chemin entre origine et destination, en évitant les obstacles.

- L'objectif est double :
  - Écrire le code manquant dans `a_star.ml` qui doit implanter un algorithme A^* générique,
  - Érire le code manquant dans `pathfinder.ml` dans le sous-répertoire `examples/PathFinder` de façon à appliquer l'algorithme A^* au problème de plus court chemin. L'objectif est de trouvers le plus court chemin sur une grille, entre une origine O et une destination D, en évitant les obstacles.

Travail à réaliser

1. Implanter l'algorithme 1 dans la fonction `search` du fichier `a_star.ml`. Utilisez pour cela les fonctions déjà implantées dans les modules `Pqueue` and `Memory`.

2. Modifier le module `MyModel` du fichier `pathfinder.ml` (sous-répertoire `examples/PathFinder`). Il vous faut coder les fonctions de coût et d'heuristique, la fonction retournant les nœuds successeurs d'un nœud parent, ainsi que la fonction indiquant si un nœud est un état terminal. Vous pouvez ajouter vos propres fonctions si besoin est.

3. Tester votre programme. Essayez l'heuristique suivante : `(fun v -> 0.)`. Que se passe-t'il ? Quel type de recherche arborescente fait l'algorithme dans ce cas ? Proposez une heuristique plus efficace.

4. Comment pouvez-vous modifier votre code pour faire une recherche de type "profondeur en premier" ? Est-ce plus efficace ?

Implantation de l'algorithme

L'algorithme A^* pourrait être codé en s'inspirant directement de l'algorithme vu en cours, en utilisant les listes G et D, et un tableau pour mémoriser les coûts $g(u)$ et les prédécesseurs $father(u)$. Cependant, plusieurs remarques vont nous conduire à choisir une implantation légèrement différente :

- Il est plus efficace d'utiliser des arbres binaires équilibrés, avec une complexité en $O(log(n))$ pour les opérations d'insertion ou d'extraction, plutôt que des listes avec une complexité en $O(n)$.
- Il est coûteux, et pas indispensable, d'allouer de la mémoire pour un tableau contenant les coûts et prédecesseurs de tous les états possible. Tous les états ne sont pas visités durant la recherche de l'A^*.

En tenant compte de ces remarque, nous remplaçons la liste "ouverte" G par une file de priorité contenant tous les états déjà visités, mais pas encore développés (*i.e.* on n'en a pas encore calculé les successeurs). On choisit également de coder $D + G$ par une table de hash, plutôt que de représenter D comme une liste. Une table de hash est une table contenant des associations $cl \rightarrow donne$. Cette table d'association permet un stockage plus économe des coûts et prédecesseurs de n'importe quel nœud de D or G.

Avec cette implantation, la liste "fermée" D n'est plus nécessaire : il suffit de stocker un booléen dans les données de la table de hash pour indiquer si un nœud a été développé ou non.

Dans la suite, Q dénote la file de priorité qui remplace la liste G, et M est la "mémoire" (codée par une table de hash) qui remplace $D + G$.

Proposed A^* implementation

Algorithm 1 Proposed implementation for A^* algorithm.

1: $cost_0 \leftarrow 0$
2: $f_0 \leftarrow cost_0 + h(u_0)$
3: Initialize memory M with u_0 and associated data $(cost_0, f_0)$
4: Initialize priority queue Q by inserting u_0 with priority f_0
5: **while** priority queue Q not empty **do**
6: Extract u from Q
7: $Q \leftarrow Q - u$
8: **if** `is_goal`(u) (terminal state) **then**
9: **Exit and return** path from u_0 to final state u
10: **end if**
11: **if** u has never been expanded before **then**
12: Memorize u as an expanded node
13: $ls \leftarrow$ `next`(u)
14: **for** all v in ls **do**
15: **if** $v \notin M$ or $cost(v) > cost(u) + k(u, v)$ **then**
16: $cost_v \leftarrow cost(u) + k(u, v)$
17: $f_v \leftarrow cost_v + h(v)$
18: $father_v \leftarrow u$
19: Store v in memory M with data $(cost_v, father_v)$
20: Insert v in Q with priority f_v
21: **end if**
22: **end for**
23: **end if**
24: **end while**
25: Raise exception (no solution)

Notez que les lignes 11, 12 et 23 ne sont pas nécessaires quand l'heuristique est consistante. Si h est consistante (ou monotone), le chemin de u_0 à u construit par l'A^* est

nécessairement de coût minimum. En conséquence, ce chemin à nécessairement la plus faible valeur de $f(u) = cost(u) + h(u)$, et il n'est pas possible que l'on revienne vers le nœud u plus tard (via un de ces successeurs), par un chemin de coût inférieur, et qu'on le re-développe.

Suggestions

Du code déjà écrit vous est fourni afin que vous puissiez compléter cet exercice en environ 2 heures. Regardez les documentations HTML (ou les fichiers `.mli`) pour choisir les fonctions qui vous seront utiles, et pour voir comment les utiliser.

Pour la question 1, vous aurez besoin des modules suivants :

- `Pqueue`, qui contient des fonctions pour créer et manipuler la file de priorité Q,
- `Memory`, avec des fonction our gérer la "memoire" M (i.e. la table de hash) qui remplace $D + G$ dans l'algorithme initial.

Pour la question 2, vous vous focaliserez essentiellement sur le fichier `pathfinder.ml` (i.e. le module `Pathfinder`). Vous pouvez également jeter un œil sur la fonction principale du fichier `main.ml` et sur les autres modules `Geo`, `Problem`, `Draw`, et `Options` présents dans le sous-répertoire `examples/PathFinder/`.

Suggestions pour la question 1 : ouvrez le fichier `a_star.ml` avec `emacs` et écrire le code de la fonction `search` implémentant l'algorithme 1:

```
let search user_fun u0 is_goal next k h =

...
```

La fonction `search` doit avoir plusieurs arguments compatibles avec la signature de la fonction, donnée dans `astar.mli`. Ces arguments sont listés ci-dessous :

- `user_fun`, une structure contenant deux fonctions `do_at_extraction` et `do_at_insertion`. Pour utiliser ces fonctions :
  - insérez `user_fun.do_at_extraction !q m u` juste après l'extraction de l'état courant u de la file de priorités `!q` (*i.e.* entre les lignes 7 et 8 de l'algorithme 1),
  - insérez `user_fun.do_at_insertion u v` juste avant l'insertion d'un nouvel état v dans la file de priorités.
- `u0` est l'état initial (ou le nœud initial dans une représentation sous forme de graphe),
- `is_goal` est une fonction telle que `is_goal u` est vrai quand `u` est un état terminal, et faux sinon,
- `next` est une fonction retournant la liste des successeurs d'un état donné,
- k est la fonction telle que `k u v` est le coût du chemin entre deux nœuds successifs `u` et `v`.
- h est l'heuristique.

Pour coder `search`, vous pouvez simplement utiliser ces arguments, sans vous préoccuper de les coder pour l'instant. Vous n'avez besoin que de leur types, décrit dans la signature de `search` dans `a_star.mli`.

Notez que la fonction `search` est utilisée dans le foncteur `Make` vers la fin du fichier. Ce foncteur prends en entrée un module du type `Model`. Il fournit en sortie un module du type `Astar` qui est une instance de l'algorithme A^* pour votre `Model`.

Suggestions pour la question 2 : Implantez le module `MyModel` du fichier `pathfinder.ml`. Ce module est du type `Model`. C'est votre représentation du problème que vous allez résoudre avec l'algorithme A^*. Il vous faut coder plusieurs fonctions dans ce module :

- `is_goal` doit retourner vrai quand la destination est atteinte,
- `next` doit retourner toutes les positions suivantes possibles à partir de la position courante sur la grille. Vous avez le choix entre deux possibilités pour les mouvements :
  - déplacement horizontal ou vertical seulement,
  - déplacement horizontal, vertical, ou en diagonale.

  N'oubliez pas d'enlever les positions se situant à l'intérieur d'un obstacle ou à l'extérieur de la grille. Vous pouvez coder une fonction `check_constraints` qui retourne vrai quand une position est libre d'obstacle et à l'intérieur du domaine, et qui retourne faux sinon.
- `k u v` doit renvoyer le coût d'un déplacement entre deux positions successives u et v. Pour votre problème, ce doit être une distance (euclidienne, ou distance de Manhattan, selon votre représentation du problème).
- `h v` doit renvoyer l'heuristique, c'est-à-dire l'estimation du coût du trajet restant à parcourir jusqu'à un état final (ici la destination). Une heuristique typique pour ce genre de problème est la distance à vol d'oiseau.