TP: A*

Intelligence Artificielle

Sol Rosca

26 octobre 2019

Table des matières

1. Architecture

Le projet est issu des mes expérimentations personnelles sur les algorithmes de recherche et est une version amputée d'un projet plus large qui a été construit avec une certaine généricité en tête.

Le projet contient 3 packages:

- **containers**: Simples structures de données utiles.
- **graph**: Contient les classes issues de la hierarchie `Graph` ainsi que la classe `Vertex` Ce package contient lui même deux sous-packages:
  - **heuristics**: contient les stratégies heuristiques
  - **search**: contient les stratégies de recherche
- **data**: Données du problème ainsi qu'un module d'extraction.

le module graph possède deux modules

2. Conventions

- Conventions usuelles PEP3
- Type anotations
- Nommage et commentaires en anglais

Le code n'est pas documenté.

3. Quick start

À la racine du projet: `python3 main.py`

3.1. Requirement

- *python 3.7*[1]

1. N'a pas été testé sous Windows, uniquement Linux

4. Heuristiques

L'heuristique est ce qui permet de "tuner" le comportement de A* en lui donnant une estimation du coût minimum entre tout vertex n et le but. Dans le packages `graph/heuristics` se trouvent les 5 heuristiques à implémenter:

4.1. Heuristique nulle

- $h_0(n) = 0$
- **Admissible**: 0 ne suréstime jamais le coût réel.

Python

```python
# graph.heuristics.NullHeuristic

class NullHeuristic(HeuristicStrategy):
    def compute(self, x1: int, y1: int, x2: int, y2: int) -> int:
        return 0
```

4.2. Heuristique sur l'axe des abscisses

- $h_1(n) = abs(y_1 - y_2)$
- **Admissible**: Pareil que la précédente mais sur l'autre axe.

Python

```python
# graph.heuristics.XHeuristic

class YHeuristic(HeuristicStrategy):
    def compute(self, x1: int, y1: int, x2: int, y2: int) -> int:
        return abs(y1 - y2)
```

4.3. Heuristique sur l'axe des ordonnées

- $h_2(n) = abs(x_1 - x_2)$
- **Admissible**: La distance en x est dans "le meilleur des cas" équivalente à la distance. Cette heuristique ne suréstime donc jamais le coût réel.

Python

```python
# graph.heuristics.YHeuristic

class XHeuristic(HeuristicStrategy):
    def compute(self, x1: int, y1: int, x2: int, y2: int) -> int:
        return abs(x1 - x2)
```

4.4. Heuristique à vol d'oiseau

- $h_3(n) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
- **Admissible**: Par définition la distance la plus courte. Cette heuristique ne suréstime donc jamais le coût réel.

Python

```python
# graph.heuristics.AsCrowFliesHeuristic

class AsCrowFliesHeuristic(HeuristicStrategy):
    def compute(self, x1: int, y1: int, x2: int, y2: int) -> float:
        return math.sqrt((x2 - x1)**2 + (y2 - y1) ** 2)
```

4.5. Heuristique distance de Manhatan

- $h_4(n) = abs(x_1 - x_2) + abs(y_1 - y_2)$
- **Pas admissible**: suréstime toujours la distance entre deux points.

Python

```python
# graph.heuristics.ManhatanHeuristic

class NullHeuristic(HeuristicStrategy):
    def compute(self, x1: int, y1: int, x2: int, y2: int) -> int:
        abs(x1 - x2) + abs(y1 - y2)
```

5. Implémentation d'A*

A* combine:

- L'algorithme de **Uniform-cost-search (variante de Dijkstra)**: Favorise les vertices proches du point de départ
- **Greedy-best-first-search**: Utilisation d'heuristique favorise les vertices proches du but.

Le choix a été fait d'utiliser une PriorityQueue pour la frontière. De cette façon il n'est pas nécessaire de se soucier du tri de cette dernière au sein même de l'algo. De plus un arbre binaire est une structure de données efficiente pour ce cas de figure de part sa capacité à permettre des insertions et retraits rapides. Voici son implémentation, il s'aggit d'un simple wrapper autour de `heapq`:

Python

```python
class PriorityQueue:

    def __init__(self) -> None:
        self.elements: List[Tuple[float, Any]] = []

    def __contains__(self, item) -> bool:
        return item in self.elements

    def __iter__(self) -> Iterator:
        return iter(self.elements)

    def empty(self) -> bool:
        return not self.elements

    def add(self, item: Any, priority: int) -> None:
        heapq.heappush(self.elements, (priority, item))

    def get(self) -> Any:
        return heapq.heappop(self.elements)[1]
```

Lors de l'appel de la methode `get`, `heapq` se charge de retourner l'élément ayant la priorité la plus importante dans la liste `elements` qui contient une liste de `tuples` composés de la priorité ainsi que d'un `Vertex`.

L'algorithme de recherche A* est comme expliqué précédament un mélange de **Uniform-cost-search** et de **Greedy-best-first-search**. Cette implémentation utilise des `Vertex` qui conntiennent des `City` et oppère au sein d'une classe `CitiesNetwork` qui est une spécialisation de `SearchableGraph` lui même spécialisatoin de `Graph`. Ces classes se trouvent dans le dossier `graph/`.

Python

```python
class AStarStrategy(SearchStrategy):

    def search(self,
               graph: SearchableGraph,
               start: Vertex,
               goal: Vertex) -> Vertex or None:

        frontier = PriorityQueue()
        frontier.add(start, 0)
        cost_so_far = {start: 0}
        counter = 0

        while not frontier.empty():
            current = frontier.get()
            counter += 1

            if current == goal:
                return (current, counter)

            for next in current.connections:
                new_cost = cost_so_far[current] + current.weight_to(next)

                if new_cost < cost_so_far.get(next, 99999):
                    cost_so_far[next] = new_cost

                    x1, y1 = next.id.x, next.id.y
                    x2, y2 = goal.id.x, goal.id.y

                    heuristic = graph.heuristic.compute(x2, y2, x1, y1)
                    priority = new_cost + heuristic

                    next.parent = current
                    frontier.add(next, priority)

        return None
```

6. Questions

6.1. Question 1

> *L'utilisation des différentes heuristiques a-t-elle une influence sur l'efficacité de la recherche ? (en termes du nombres de noeuds visités)*

Oui, le nombre de villes visitées change en fonction à l'heuristique.

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

- Si $h(n) = 0$ alors il n'y a que $g(n)$ qui influence le comportement. Alors, A* aura le même comportement que l'algorithme de Uniform-cost-search (assure de trouver le chemin le plus court).
- Si $h(n)$ est toujours **plus petit ou égale** au coût du déplacement entre n et le but, alors A* assure de toujours trouver le chemin le plus court. Plus la valeur $h(n)$ est basse, plus A* explorera de vertices et le rendra plus lent.
- Si $h(n)$ **est égale** au coût du déplacement entre n et le but, alors l'exploration de A* sera équivalente au meilleur chemin. (*Si A* est parfaitement informé, son comportement est parfait*)
- Si $h(n)$ est **parfois plus grand** que le coût du déplacement entre n et le but, A* n'assure pas de trouver le chemin le plus court (mais en contrepartie sera plus rapide).
- Si $h(n)$ est très grand vis à vis de $g(n)$ alors, c'est uniquement $h(n)$ qui influence le comportement et A* aura le **même comportement que Greedy-best-first-search** (trouve un chemin plus rapidement que Dijkstra mais pas forcément le plus court).

6.2. Question 2

> *Pouvez-vous trouver des exemples où l'utilisation de différentes heuristiques donne des résul- tats différents en termes de chemin trouvé ?*

Les trois cas de figure suivants (entre autres) donnent des résultats plus longs avec la distance de Manhatan:

- Bruxelles à Prague
- Paris à Lisbonne
- Paris à Prague

6.3. Question 3

> *Dans un cas réel, quelle heuristique utiliseriez-vous?*

La distance à vol d'oiseau donnera toujours le trajet le plus court en plus d'être relativement éfficiente en terme de "hops".

7. Exemple d'output

Pour la configuration "De **Bruxelles** à **Prague** l'output est le suivant:

Bash

```
Starting search with AStar + Null
from Brussels (983; 992) to Prague (574; 975)

Hops: 10; Best path: 4

Brussels (983; 992)
Amsterdam (957; 1096)
Munich (679; 835)
Prague (574; 975)
-------------------------------------------------
Starting search with AStar + X
from Brussels (983; 992) to Prague (574; 975)

Hops: 9; Best path: 4

Brussels (983; 992)
Amsterdam (957; 1096)
Munich (679; 835)
Prague (574; 975)
-------------------------------------------------
Starting search with AStar + Y
from Brussels (983; 992) to Prague (574; 975)

Hops: 8; Best path: 4

Brussels (983; 992)
Amsterdam (957; 1096)
Munich (679; 835)
Prague (574; 975)
-------------------------------------------------
Starting search with AStar + AsCrowFlies
from Brussels (983; 992) to Prague (574; 975)

Hops: 8; Best path: 4

Brussels (983; 992)
Amsterdam (957; 1096)
Munich (679; 835)
Prague (574; 975)
-------------------------------------------------
Starting search with AStar + Manhatan
from Brussels (983; 992) to Prague (574; 975)

Hops: 6; Best path: 5

Brussels (983; 992)
Amsterdam (957; 1096)
Hamburg (774; 1175)
Berlin (626; 1131)
Prague (574; 975)
-------------------------------------------------
```