Algorithme A*

Thibaut Antoine (thibaut.antoine@ens-rennes.fr)

Phrase d'introduction

On présente l'algorithme A^*, qui permet de calculer la plus courte distance entre deux sommets d'un graphe pondéré en présence d'informations supplémentaires.

1 Principe

Algo de recherche de plus court chemin dans un graphe

Entrée : $G = (S, A, d), s, t, \boldsymbol{h}$
Sortie : Longueur d'un plus court chemin de s à t dans G.

L'algorithme A^* est un algorithme de recherche de plus court chemin dans un graphe pondéré. Il prend en entrée un graphe pondéré $G = (S, A, d)$ et deux sommets s et t *[pour plus d'effet, on commence par ne pas parler de l'heuristique]* et renvoie la distance d'un plus court chemin dans G de s à t.

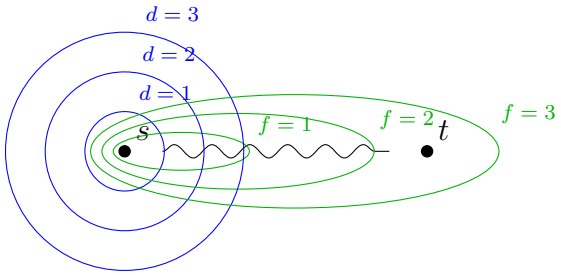
Dijkstra : Choix non éclairé

$\rightsquigarrow$ *Idée d'A^* :* utiliser des infos en plus pour optimiser la recherche (la dest. est au nord, distance à vol d'oiseau dans un réseau routier...) $\rightarrow$ **heuristique**

Mais contrairement à l'algorithme de Dijkstra qui répond au même besoin, A^* se sert d'informations supplémentaires sur la position de t pour accélérer la recherche, comme la distance à vol d'oiseau si notre graphe représente un réseau routier, qu'on formalise par la notion d'heuristique. *[On peut ajouter l'heuristique en entrée de l'algo à ce moment là]*

2 Illustration [AIM]

Les deux algorithmes explorent les sommets du graphe par cercles concentriques autour de s de « diamètre » croissant. Le but d'A^* est d'aplatir ces cercles afin de les transformer en ellipses allongées vers t.



On minimise f = distance estimée + heuristique

3 Algorithme [TOR]

Plus précisément, on va appliquer l'algorithme de Dijkstra, mais en sortant les sommets de la file de priorité par $g + h$ (distance estimée + heuristique) minimale.

A$^*(G, s, t, h)$

```
1  Q := {s}                                              # File de priorité
2  g := [v : ∞ pour chaque sommet v] ; g[s] ← 0
3  f := [v : ∞ pour chaque sommet v] ; f[s] ← h(s)
4  tant que Q ≠ ∅ faire
5  |  v := Sommet de Q tel que f[n] minimal
6  |  si v = t alors
7  |  |  renvoyer g[v]
8  |  sinon  pour chaque u successeur de v faire
9  |  |  si g[v] + d(v, u) < g[u] alors                  # Relaxation de v → u
10 |  |  |  g[u] ← g[v] + d(v, u)
11 |  |  |  f[u] ← g[u] + h(u)
12 |  |  |  Ajouter u dans Q
13 renvoyer Pas trouvé
```

On peut noter que, contrairement à l'algorithme de Dijkstra, on peut ajouter plusieurs fois un même sommet dans la file de priorité si on l'a redécouvert par plusieurs chemins différents. Cela vient du fait qu'a priori, la valeur de f peut diminuer le long d'un chemin ; ainsi, lorsqu'on rencontre un sommet n pour la première fois, on ne connaît pas forcément sa distance minimale à

s, bien qu'il soit le sommet ayant la plus petite valeur de f. C'est contraire à ce qu'il se passe pour Dijkstra, où grâce à l'hypothèse des poids positifs, la distance sur un chemin ne peut qu'augmenter.

4 Correction [TOR]

On pose $\delta(u, v) =$ distance minimale de u à v dans G.

Théorème. *Si pour tout u, $h(u) \leq \delta(u, t)$ [on dit que h est admissible], alors A^* renvoie la longueur d'un plus court chemin de s à t.*

Preuve. Supposons que ça ne soit pas le cas, i.e. A^* renvoie la valeur $d > \delta(s, t) = \delta^*$. Alors t n'est jamais ajouté à Q avec une priorité plus faible que c'est avec une priorité $f[t] = d + h(t) = d$ [car $h(t) \leq \delta(t, t) = 0$, et si on l'ajoutait avec une priorité $d' < d$ il serait sorti avant et on aurait renvoyé d'].

Soit $s \to v_1 \to \cdots \to v_n \to t$ un chemin de longueur δ^*. Alors au moment où on explore s, on ajoute v_1 à Q avec priorité :

$$\begin{aligned} f[v_1] &= g[v_1] + h(v_1) \\ &= d(s, v_1) + h(v_1) \\ &\leq d(s, v_1) + \delta(v_1, t) \\ &\leq \delta(s, t) = \delta^* \qquad \text{[car chemin le plus court]} \end{aligned}$$

Ainsi v_1 est exploré avant t. À ce moment-là, on ajoute v_2 dans la file avec priorité inférieure à δ^* [l'argument est similaire, à ceci près que $g[v_1]$ a pu changer mais n'a pu que diminuer, donc l'inégalité est toujours vérifiée].

De proche en proche, v_n est exploré avant t, et t est ajouté à Q avec priorité inférieure à $\delta^* < d$: contradiction. □

On n'a pas parlé du cas où l'algorithme ne trouve pas de chemin entre s et t, mais il n'est pas très dur de voir que A^* va finir par explorer tous les sommets de la composante (simplement) connexe de s dont la valeur de f est strictement inférieure à δ^* (sauf s'il y a un cycle de poids négatif, mais on a implicitement éliminé ce cas dès le début).

Si l'on n'est pas convaincu(e), on peut toujours prendre un tel sommet v et supposer qu'il n'est jamais exploré malgré l'existence d'un chemin $s \rightsquigarrow v$. Alors si w est le sommet de Q le plus proche de v sur le chemin (qui existe puisque $s \in Q$ initialement), l'absence de cycle de poids négatif implique que w sortira nécessairement de Q son successeur sur $s \rightsquigarrow v$ sera ajouté à Q : contradiction.

5 Complexité [TOR]

— Complexité au pire cas : potentiellement exponentielle
— Si $h(u) \leq d(u,v) + h(v)$ pour tous u, v : idem que Dijkstra. A^* devient équivalent à Dijkstra avec $\hat{d}(u,v) := d(u,v) + h(v) - h(u) \geq 0$.

Comme on l'a fait remarquer avant, les sommets déjà explorés peuvent l'être à nouveau, ce qui donne une complexité exponentielle au pire cas car il y a un nombre au plus exponentiel de chemins sans cycle entre s et t. On peut empêcher cette situation d'arriver en choisissant l'heuristique de sorte à ce que la valeur de f sur un chemin ne puisse qu'augmenter. Cela revient à avoir $h(u) \leq d(u,v) + h(v)$ pour tous sommets u, v ; on dit alors que h est monotone.

En fait, lorsqu'on choisit h d'une telle manière, l'algorithme A^* est exactement le même que l'algorithme de Dijkstra appliqué à G dont la pondération est modifiée ainsi :

$$\hat{d}(u,v) := d(u,v) + h(v) - h(u).$$

On peut noter qu'on obtient bien des poids positifs en faisant cette modification.

Organisation du tableau

Titre 1. Principe 2. Illustration	3. Algorithme	4. Correction	 5. Complexité

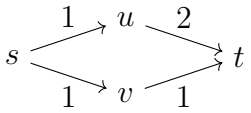
Remarques

— Il est à noter que A* est aussi l'algorithme visitant le moins de nœuds possible pour trouver un chemin optimal. La preuve de ce fait se trouve dans [**BAR**].

Ce n'est peut-être pas très pertinent de la faire au cours du développement, mais ça peut faire l'objet d'une question.

— On peut noter aussi qu'il y a deux cas extrêmes d'heuristique. Le premier est celui d'une heuristique non informative $h = 0$; dans ce cas on retombe sur l'algorithme de Dijkstra. Le second est celui d'une heuristique « oracle » : $h(u) = \delta(u, t)$; dans ce cas tout sommet sur un plus court chemin de s à t sera ajouté avec une priorité δ^* et tous les autres avec une priorité strictement plus grande. En particulier cela veut dire qu'A^* n'explore aucun sommet qui n'est pas sur un plus court chemin de s à t.

— Si h n'est pas admissible, A^* ne renvoie pas forcément la distance la plus courte (il renvoie tout de même la longueur d'un chemin du graphe donc il surestime la distance). Un contre-exemple est le graphe suivant :



avec l'heuristique donnée par $h(u) = 1, h(v) = 10$ et $h(t) = 0$. Le déroulé de l'algorithme sur ce graphe donne la distance 3 avec le chemin $s \to u \to t$.

— On peut modifier facilement l'algorithme pour qu'il renvoie le chemin le plus court qu'il a obtenu en maintenant un tableau des parents qu'on met à jour à chaque relaxation d'arc.

— En tant que tel ce développement n'est peut être pas très adapté à la leçon sur les jeux, cependant il est facilement adaptable en changeant l'illustration pour un exemple d'un graphe d'un jeu.

— Le développement est peut-être légèrement trop long, mais il devrait se faire sans problème si l'on garde un bon rythme.

— J'ai repris ici la preuve de [**TOR**] qui est assez concise, mais il existe une autre preuve dans [**AIM**] que je trouve plus élégante ; elle est cependant un peu trop longue il me semble (et beaucoup moins lisible), et il est nécessaire d'avoir le temps pour dire quelques mots sur la partie 5.

— L'illustration n'est pas dans [**TOR**] mais dans [**AIM**].

— [**TOR**] donne un bon exemple de graphe pour lequel il est nécessaire de remettre un sommet dans la file de priorité pour avoir le bon résultat dans le cas d'une heuristique admissible mais non monotone.

Références

[TOR] *"La Tortue MPI" Informatique - MP2I/MPI - CPGE 1re et 2e années - Cours et exercices corrigés*, Balabonski, Conchon, Filliâtre, Nguyen, Sartre

[AIM] *Artificial Intelligence - A modern approach*, Stuart J. Russell, Peter Norvig

[BAR] *Informatique MP2I et MPI - CPGE 1re et 2e années*, Vincent Barra