TECHNIQUES ALGORITHMIQUES ET PROGRAMMATION

Complément sur l'algorithme A*

On rappelle l'algorithme A* vu en cours :

Algorithme A* (du cours)

Entrée : Un graphe G, potentiellement asymétrique, arête-valué par une fonction de poids ω positive ou nulle, $s, t \in V(G)$, et une heuristique $h(x,t)$ estimant la distance entre les sommets x et t dans G.
Sortie : Un chemin entre s et t dans G, une erreur s'il n'a pas été trouvé.

1. Poser $P := \varnothing$, $Q := \{s\}$, $\text{coût}[s] := 0$, $\text{parent}[s] := \perp$, $\boxed{\text{score}[s] := \text{coût}[s] + h(s,t)}$
2. Tant que $Q \neq \varnothing$:
   (a) Choisir $u \in Q$ tel que $\boxed{\text{score}[u]}$ est minimum et le supprimer de Q
   (b) Si $u = t$, alors renvoyer le chemin de s à t grâce à la relation $\text{parent}[u]$: $t \to \text{parent}[t] \to \text{parent}[\text{parent}[t]] \to \cdots \to s$
   (c) Ajouter u à P
   (d) Pour tout voisin $v \notin P$ de u :
      i. Poser $c := \text{coût}[u] + \omega(u,v)$
      ii. Si $v \notin Q$, ajouter v à Q
      iii. Sinon, si $c \geqslant \text{coût}[v]$ continuer la boucle
      iv. $\text{coût}[v] := c$, $\text{parent}[v] := u$, $\boxed{\text{score}[v] := c + h(v,t)}$
3. Renvoyer l'erreur : « le chemin n'a pas été trouvé »

On admettra la propriété suivante (vue en cours) : si h est monotone, alors l'algorithme A* calcule un plus court chemin entre s et t (s'il existe).

Question 1. *Rappelez les définitions de monotonie et de sous-estimation de distance pour h.*

Question 2. *Montrez que si h est monotone et $h(t,t) \leqslant 0$, alors elle sous-estime la distance à t.*

Question 3. *Montrez que si h vérifie l'inégalité triangulaire et sous-estime la distance, alors l'algorithme A* calcule correctement un plus court chemin entre s et t (s'il existe).*

Question 4. *Montrez que pour un cycle, pour une certaine valuation ω et heuristique h, A* ne trouve pas le plus court chemin entre s et t.*

L'algorithme présenté dans le cours est en fait une variante de l'algorithme A* original. Dans sa version originale, A* peut modifier $\text{coût}[u]$ pour un sommet de u déjà dans P et le remettre dans Q. L'analyse de sa complexité est alors plus complexe. On peut montrer que l'algorithme original A* calcule toujours un court chemin entre s et t dès que h sous-estime la distance.

Question 5. *Construisez un exemple où l'algorithme A* vu en cours ne calcule pas le plus court chemin entre s et t (qui existe par ailleurs) alors que h est positive et sous-estime la distance.*

1 En TP

Télécharger les fichiers correspondant au TP à partir de la page de l'UE disponible ci-après :

http://dept-info.labri.fr/~gavoille/UE-TAP/

Complétez le fichier `a_star.c` et modifiez à votre gré `mygrid.txt`. Il vous faudra `heap.c`, ainsi que `tools.h` et `tools.c`. Compilez avec `make a_star`. A priori vous n'avez à modifier que ces deux fichiers (`a_star.c` et `mygrid.txt`). Testez correctement votre algorithme en prenant des exemples pertinents qui différencient DIJKSTRA d'A* (avec l'heuristique vol d'oiseau), en prennant éventuellement des exemples qui font échouer A* (par exemple avec les cellules de type `TX_TUNNEL` de poids < 1). Comparez les performances entre le coût des chemins trouvés et du nombre de sommets visités. Programmez les améliorations proposées.

Passer à la 3D (touche `[d]`) et developez une heuristique, disons `level(s,t,&G)`, permettant d'éviter les montagnes et/ou les valées. Elle peut aussi être combinée avec l'heuristique vol d'oiseau. Testez votre heuristique sur un terrain vierge pour constater le changement de comportement. Est-elle monotone ?

Dans un deuxième temps, on désire implémenter une version d'A* qui construit en parallèle un chemin de s à t et un chemin de t à s, `A_star2(G,h)`, très similaire à `A_star(G,h)`. La différence de code étant faible, vous pouvez même utiliser une seule fonction et ajouter une variable globale `version` pour préciser si `A_star(G,h)` utilise la version simple (`verson=1`) ou double (`version=2`) .

L'idée est d'exécuter A* depuis s et depuis t. On notera P_s l'ensemble P classique lorsqu'A* s'exécute depuis s, et P_t l'ensemble P lorsqu'A* s'exécute depuis t. Le tas Q quant à lui est commun aux deux exécutions. Lorsqu'on extrait un nœud du tas il sera de score minimum soit vis-à-vis de s soit vis-à-vis de t, et on l'ajoutera alors soit à P_s soit à P_t.

Plus précisément :

1. Ajouter un champ `int source;` à la structure `node` indiquant si un nœud a été exploré à partir de s (`=1` par exemple) ou de t (`=0`).
2. Utilisez la marque `MK_USED2` pour indiquer que le nœud a été ajouté dans P_t. Vous utiliserez l'ancienne marque `MK_USED` pour indiquer qu'il est dans P_s.
3. Lorsque vous ajoutez un nœud à Q faite attention de marquer le sommet correspondant à `MK_FRONT` seulement s'il n'est pas déjà marqué, ce qui est rendu possible maintenant à cause des deux ensembles P_s et P_t, et de leur marquage.

L'algorithme s'arrête lorsqu'un nœud extrait du tas Q correspond à un sommet déjà marqué mais par une autre origine. Cela arrive par exemple, si l'origine du nœud u est s (`u->source=1`) alors que la marque du sommet qu'il représente est `MK_USED2` (donc est dans P_t). Pour reconstruire le chemin complet vous pourrez chercher un nœud du tas dont le père v est un nœud correspondant au même sommet que u mais issu de l'autre origine.