TECHNIQUES ALGORITHMIQUES ET PROGRAMMATION

Supplement on the A* Algorithm

Recall the A* algorithm seen in the course :

Algorithme A* (from course)

Entrée : A graph G, potentially asymmetric, edge-weighted by a non-negative weight function ω, $s, t \in V(G)$, and a heuristic $h(x,t)$ estimating the distance between vertices x and t in G.

Sortie : A path between s and t in G, an error if it was not found.

1. Set $P := \varnothing$, $Q := \{s\}$, $\text{cost}[s] := 0$, $\text{parent}[s] := \perp$, $\boxed{\text{score}[s] := \text{cost}[s] + h(s,t)}$
2. While $Q \neq \varnothing$:
   (a) Choose $u \in Q$ such that $\boxed{\text{score}[u]}$ is minimum and remove it from Q
   (b) If $u = t$, then return the path from s to t using the relation $\text{parent}[u] : t \to \text{parent}[t] \to \text{parent}[\text{parent}[t]] \to \cdots \to s$
   (c) Add u to P
   (d) For every neighbor $v \notin P$ of u :
      i. Set $c := \text{cost}[u] + \omega(u,v)$
      ii. If $v \notin Q$, add v to Q
      iii. Otherwise, if $c \geqslant \text{cost}[v]$ continue the loop
      iv. $\text{cost}[v] := c$, $\text{parent}[v] := u$, $\boxed{\text{score}[v] := c + h(v,t)}$
3. Return the error : "the path was not found"

We assume the following property (seen in the course) : if h is monotonic, then the A* algorithm calculates a shortest path between s and t (if it exists).

Question 1. *Recall the definitions of monotonicity and distance underestimation for h.*

Question 2. *Show that if h is monotonic and $h(t,t) \leqslant 0$, then it underestimates the distance to t.*

Question 3. *Show that if h satisfies the triangle inequality and underestimates the distance, then the A* algorithm correctly calculates a shortest path between s and t (if it exists).*

Question 4. *Show that for a cycle, for a certain valuation ω and heuristic h, A* does not find the shortest path between s and t.*

The algorithm presented in the course is actually a variant of the original A* algorithm. In its original version, A* can modify $\text{cost}[u]$ for a vertex u already in P and put it back into Q. The analysis of its complexity is then more complex. It can be shown that the original A* algorithm always calculates a shortest path between s and t as long as h underestimates the distance.

Question 5. *Construct an example where the A* algorithm seen in the course does not calculate the shortest path between s and t (which exists otherwise) while h is positive and underestimates the distance.*

Lab Session

Download the files corresponding to the lab session from the course page available below :

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

Complete the file `a_star.c` and modify `mygrid.txt` as you wish. You will need `heap.c`, as well as `tools.h` and `tools.c`. Compile with `make a_star`. In principle, you only need to modify these two files (`a_star.c` and `mygrid.txt`). Correctly test your algorithm by taking relevant examples that differentiate DIJKSTRA from A* (with the as-the-crow-flies [=vol d'oiseau] heuristic), possibly taking examples that make A* fail (for example with cells of type `TX_TUNNEL` with weight < 1). Compare performance between the cost of the paths found and the number of visited vertices. Program the proposed improvements.

Switch to 3D (key `[d]`) and develop a heuristic, say `level(s,t,&G)`, allowing to avoid mountains and/or valleys. For that, use the field `G.height[x][y]` returning the height for the cell (x, y), a value in [`G.hmin`,`G.hmax`]. It can also be combined with the as-the-crow-flies [=vol d'oiseau] heuristic to avoid mountains and valleys while heading toward the destination. Test your heuristic on a virgin terrain to observe the change in behavior. Is it monotonic ?

In a second phase, we wish to implement a version of A* that constructs in parallel a path from s to t and a path from t to s, `A_star2(G,h)`, very similar to `A_star(G,h)`. Since the difference in code is small, you can even use a single function and add a global variable `version` to specify if `A_star(G,h)` uses the simple version (`version=1`) or the double version (`version=2`).

The idea is to execute A* from s and from t. Let P_s denote the classic set P when A* runs from s, and P_t denote the set P when A* runs from t. The heap Q, on the other hand, is common to both executions. When a node is extracted from the heap, it will have a minimum score either with respect to s or with respect to t, and it will then be added either to P_s or to P_t.

More precisely :

1. Add a field `int source;` to the `node` structure indicating if a node has been explored from s (`=1` for example) or from t (`=0`).
2. Use the `MK_USED2` mark to indicate that the node has been added to P_t. You will use the old `MK_USED` mark to indicate that it is in P_s.
3. When you add a node to Q, be careful to mark the corresponding vertex as `MK_FRONT` only if it is not already marked, which is now possible because of the two sets P_s and P_t, and their marking.

The algorithm stops when a node extracted from the heap Q corresponds to a vertex already marked but by another origin. This happens, for example, if the origin of node u is s (`u->source=1`) while the mark of the vertex it represents is `MK_USED2` (thus it is in P_t). To reconstruct the full path, you can look for a node in the heap whose father v is a node corresponding to the same vertex as u but coming from the other origin.