Artificial Intelligence – TP

Finding Simple Strategies

2018 – 2019

The objective is to put into action some of the algorithms we have seen for finding strategies, using basic and optimal path-finding.

You can choose the programming language you prefer, though the instructor can best understand C, C++, Python, Java, and Haskell. Functions for the basic creation of the environment, its display and the computation of the successors of a given tile are available on the Hippocampus server for C++ and Python. For other languages you will have to recode them.

We model the environment as a tiled rectangular area of length L and height H (use constants or variables). A robot tries to move from coordinates $(1, 1)$ (top-left of the screen) to coordinates $(L-2, H-2)$ (bottom-right of the screen). Each tile either is empty or contains a wall. The first and last lines and columns contain only walls.

1 Basic Path-finding

Q1. Write a function implementing depth-first search to find a path to the goal.

Display each iteration of the algorithm by materialising the waiting and passed lists in some way (e.g. with some special character). At the end compute and display the path, its length, the number of visited tiles, and the maximum size of the waiting list. Is the exploration satisfying ?

Some hints :

— The easiest way to represent the passed list is probably to use an array (say of booleans) with the same size as the environment array. This array can also be used to record that an element is in the waiting list ;

— The waiting list itself could be a doubly-linked list, or a dequeue, or any similar data structure that permits efficient insertion and deletion both at the front and at the back of the list.

Q2. Do the same for breadth-first search. Compare with DFS.

2 Optimal Path-finding

Now we want to add a positive cost to visit each tile. We assume all tiles have cost 1 except the ones in the rectangle defined by points $(L/3, 0)$ and $(2L/3, H-1)$ which have some constant cost $c > 1$.

Q3. Update the world random-generation function to also generate costs.

Q4. Update the world display function to highlight tiles with costs > 1 (when there is no wall)

Q5. Write a function to implement Dijkstra's algorithm. Display the successive iterations (materialise the visited states : those that have a finite "cost to get there" value). Display the length and cost of the path found (also update breadth- and depth-first search to compute and display the costs of the paths found)

Hint : the classic data structure for the waiting list here is called a *priority queue*. Most languages (e.g. Python or C++) have some standard implementation of priority queues, but not C for instance.

Q6. Devise an admissible heuristic for this problem.

Q7. Write a function implementing the A^* algorithm. Display the successive iterations, path length and cost as before.