EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

MATH. - NATURWISS. FAKULTÄT
FACHBEREICH INFORMATIK
KOGNITIVE SYSTEME · PROF. A. ZELL

Artificial Intelligence

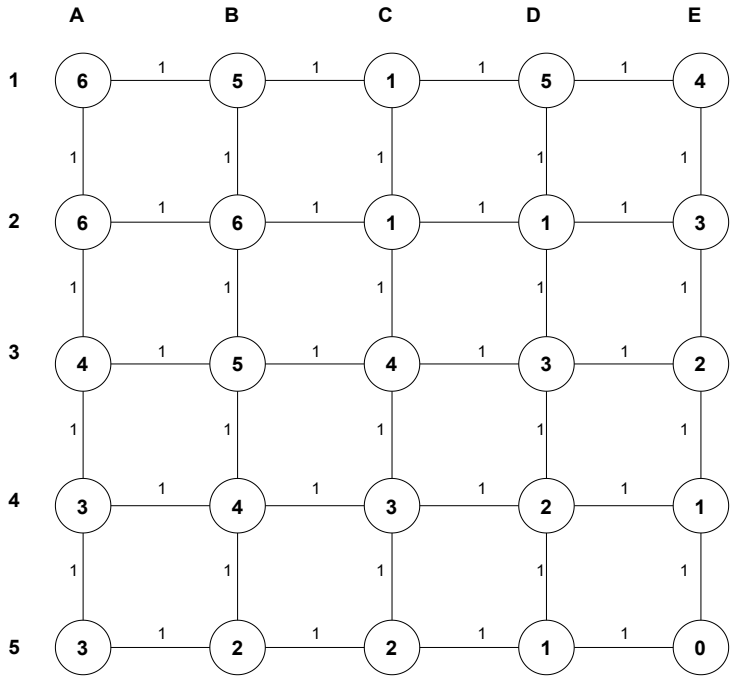
Assignment 6

Assignment due by: 10.12.2021

Notes: Code files that do not compile will be awarded 0 points. Do not rename the code files i.e. submit them named as $lrta.py$ and $annealing.py$.

Question 1 Learning Real-Time A* (7 points)

The graph below shows a problem you have to solve using Learning Real-Time A* (LRTA*). The starting state is $A1$ (top left) and the goal state is $E5$ (right bottom). The values inside the nodes are the heuristics $h(s)$ of those nodes to the goal state $E5$. The costs between all the neighboring nodes $c(s, a, s')$ are 1 as shown in the graph



By following the algorithm in $slide\ 77$ of the $lecture\ script\ 04$, apply LRTA* on the graph above and fill in the table below. s is the previous state, s' is the current state and $H(s)$ is the cost LRTA* updates for the previous node in each iteration. Rather than writing the cost of all neighbouring nodes in the table, simply write the one which has the minimum cost. For example, from $A1$ we could choose $A2$ (cost=7) or choose $B1$ (cost=6), but we only write $B1$ in the table below because it has the minimum cost.

Note: In case of equal cost to move Right or Down, always choose to move Down the graph.

s	s'	$H(s)$
-	A1	-
A1	B1	5+1=6
B1	?	?

Question 2 Implementing Learning Real-Time A* (1+3+2 = 6 points)

The goal of this exercise is to implement Learning Real-Time A* on the graph defined in Question 1. You should follow the algorithm in *slide* 77 of the *lecture script* 04. Download the template *lrta.py* which contains the classes and the graph.

A note on terminology: in this graph, the actions possible at a node are the target nodes that can be reached from there. Therefore, actions and their resulting states can be treated equivalently. Also, `H[s]` is implemented as a variable inside each node object, rather than a value inside a list *H table*.

(a) Implement a function `lrta_cost(previous_state, state)`, which returns the cost estimate of a node using the distance between the nodes (`previous_state` and `state`) and the H table as done in the algorithm. To support the implementation of this function, we have already:

(i) added `h(s)` for the goal node E5 and `get_distance()` to calculate the distance between `node` and its neighbour (referred as `c(s,a,s')` in slide 77).

(ii) initialized `H[s]` to `h[s]`.

(b) Implement a function `lrta_agent(state, previous_state, goal)`, which returns the next node to be expanded. It should update the cost of the previous node `H[s]` as shown in the algorithm. You should use the function in part (a) to calculate the costs.

(c) Implement a function `lrta_graphsearch(start_node, goal_node)` which uses the function in part (b) until it reaches the goal. This function returns the path and the number of steps taken by the agent. For example, for the graph defined in Question 1 we have the following output:

```
The path from A1 to E5 is:
'A1', 'B1', 'C1', 'C2', 'D2', 'C2', 'C1', 'B1', 'C1', 'C2', 'D2', 'D3', 'D4',
'D5', 'E5'
Total number of steps taken:  14
```

Question 3 Simulated Annealing with Python (5 + 2 = 7 points)

In this question, we want to solve the 8-puzzle problem with simulated annealing algorithm. The goal state is represented as `board=[[0,1,2],[3,4,5],[6,7,8]]`, where `board[0]` is the first row of the board, `board[1]` the second, and so on. The 0 represents the empty tile. For your solution use the template file *annealing.py*.

(a) To solve this problem, we need three main parts: the simulated annealing algorithm, the function that selects the successors, and the temperature scheduler. We have already implemented the function `successors()`. For the algorithm, see the pseudocode in the lecture slides, but treat it

as an energy minimization problem as opposed to how it is formulated there. The energy function E_1 that we use here is the number of inversions[1], which is already implemented in the template.

(i) Implement the function `scheduler()` that takes an integer step as parameter and returns the computed scheduler value using: $T \longmapsto 1/\sqrt{t}+1$. Note that this scheduler will never reach a temperature of zero, and thus, we only want to terminate when a solution is found.

(ii) Implement the simulated annealing algorithm (from *slide 13*) in the function `simulated_annealing()`. This function should return the number of steps it took to find the solution. For termination condition check at each iteration whether we have found a solution (i.e. `current_state==[[0,1,2],[3,4,5],[6,7,8]]`) or not. *Remember here we are targeting an energy minimization problem.*

Try your algorithm with initial state as `[[8,1,7],[5,4,2],[0,6,3]]`. If implemented correctly, a solution should typically be found within about 30000 steps, though this could vary.

(b) We now want to examine the effect of different energy functions on the problem. To compensate for the randomness in the running time, you should implement the function `compare_energies()` that runs the algorithm at least n times (default 20) with a given scheduler and averages the number of steps. Investigate the algorithm with the following energy functions, which are implemented in the template.

(i) E_1: The number of inversions

(ii) E_2: The number of misplaced tiles

(iii) E_3: Sum of Manhattan distances of tiles to their correct places

Energy function	Goal reached at average step
E_1	
E_2	
E_3	

[1] https://en.wikipedia.org/wiki/Inversion_(discrete_mathematics)