Fundamentals of Artificial Intelligence

Practice 1: Search Algorithms.
Master in Artificial Intelligence 2022-23

Group 2

Guijas Bravo, Pedro
Miguélez Millos, Ángel
Vila de la Cruz, Daniel

January 17, 2023

Contents

1 Methods

In this first section, the problem to be solved will be formally defined, followed by an analysis of the suitability of one of the main blind search methods: breadth-first search. Finally, the two heuristics used to solve the problem with the A* algorithm will also be formalized.

1.1 Formal characterization of the problem

The problem to be solved consists of finding the easiest way for Abdel to reach his girlfriend's, Gertrudis, house. First of all, the available information of the problem is the following:

- **$\mathbf{POS_{AX}}$**, **$\mathbf{POS_{AY}}$** and orientation: Initial position of the agent (Abdel's house). As for orientation. Abdel is supposed to start looking North.
- **$\mathbf{POS_{GX}}$**, **$\mathbf{POS_{GY}}$** and orientation: Position to be achieved by the agent (Gertrudis' house). In this case, the orientation of the target is not relevant, only the position will matter.
- **Map size**: Size of the terrain in terms of rows and columns $HEIGHT \times WIDTH$.
- **TERRAIN_COST**: Mapping from a position (x, y) to an integer value $c \in [1, 5]$, that models the cost of moving to a specific position on the map, where the (0,0) corresponds to the top-left corner.

With this information, we can formally define the problem like:

- **Initial state**: Initial position and orientation of the agent, given by Abdel's house position:

$$(POS_{AX}, POS_{AY}, North)$$

- **Target state**: Agent destination, given by Gertrudis' house position:

$$(POS_{GX}, POS_{GY}, North/East/South/West)$$

- **Available operators and transition model**: The agent will be able to move around the terrain map executing 3 possible actions. The agent can always move forward (**MOVE**) unless he is already in the border and looking outside of the map. It can change its orientation as well through 90 and -90 degree rotations (**ROTATE_R** and **ROTATE_L**, respectively).

  Considering the x axis from south to north and the y axis from west to east, all the possible actions that could be executed are represented in the Table 1, where it is shown that only transitions in which forward motion is applied have a precondition, since the agent cannot leave the map.

State	Operator	Precondition	Result
$(\text{pos}_x, pos_y, North)$	ROTATE_R		$(\text{pos}_x, pos_y, East)$
$(\text{pos}_x, pos_y, North)$	ROTATE_L		$(\text{pos}_x, pos_y, West)$
$(\text{pos}_x, pos_y, East)$	ROTATE_R		$(\text{pos}_x, pos_y, South)$
$(\text{pos}_x, pos_y, East)$	ROTATE_L		$(\text{pos}_x, pos_y, North)$
$(\text{pos}_x, pos_y, South)$	ROTATE_R		$(\text{pos}_x, pos_y, West)$
$(\text{pos}_x, pos_y, South)$	ROTATE_L		$(\text{pos}_x, pos_y, East)$
$(\text{pos}_x, pos_y, West)$	ROTATE_R		$(\text{pos}_x, pos_y, North)$
$(\text{pos}_x, pos_y, West)$	ROTATE_L		$(\text{pos}_x, pos_y, South)$
$(\text{pos}_x, pos_y, North)$	MOVE	$\text{pos}_x > 0$	$(\text{pos}_x - 1, pos_y, North)$
$(\text{pos}_x, pos_y, East)$	MOVE	$\text{pos}_y < WIDTH - 1$	$(\text{pos}_x, pos_y + 1, East)$
$(\text{pos}_x, pos_y, South)$	MOVE	$\text{pos}_x < HEIGHT - 1$	$(\text{pos}_x + 1, pos_y, South)$
$(\text{pos}_x, pos_y, West)$	MOVE	$\text{pos}_y > 0$	$(\text{pos}_x, pos_y - 1, West)$

Table 1: Transition model for every possible state and action

- **Goal test**: The goal is reached when the coordinates of the agent are equal to the coordinates of the goal. In other words, when Abdel arrives to Gertrudis' house.

$$pos_x == POS_{GX} \ \ \& \ \ pos_y == POS_{GY}$$

- **Cost function**: The transitions carried by a rotation require a cost of 1, while the cost of the forward movement depends on the terrain cost. The cost associated to each action is shown in the Table 2.

State	**Operator**	**Cost**
$(\text{pos}_x, pos_y, North/East/South/West)$	ROTATE_R	1
$(\text{pos}_x, pos_y, North/East/South/West)$	ROTATE_L	1
$(\text{pos}_x, pos_y, North)$	MOVE	$6 >$ TERRAIN_COST$(\text{pos}_x - 1, pos_y) > 0$
$(\text{pos}_x, pos_y, East)$	MOVE	$6 >$ TERRAIN_COST$(\text{pos}_x, pos_y + 1) > 0$
$(\text{pos}_x, pos_y, South)$	MOVE	$6 >$ TERRAIN_COST$(\text{pos}_x + 1, pos_y) > 0$
$(\text{pos}_x, pos_y, West)$	MOVE	$6 >$ TERRAIN_COST$(\text{pos}_x, pos_y - 1) > 0$

Table 2: Cost function for every possible state and action

1.2 Analysis of suitability of a blind search method

In this section, an analysis that justifies the suitability of the **breadth-first** search algorithm before the implementation will be performed. This search method has been chosen since, despite the fact that it can scale worse in time and memory, it will find simpler routes than the depth-first search.

This algorithm goes through all the possible actions on a state before focusing on the new nodes generated. This is translated as generating in the first level b nodes, in the second level b^2 and in the d-level b^d nodes, which have to be expanded before passing to the next level. This way, we can assure that the algorithm is **complete** because the map is finite (so the depth is finite) and the number of actions are limited (only 3 actions)[1]. In fact, it will always find the shallowest solution. Assuming a relatively uniform terrain cost, it will also find a less expensive path compared to the depth-first. It should be noted that if an expensive solution is found at a shallower level than other cheaper solutions, this algorithm will **not provide the optimal result**. However, if the cost of the nodes follows a non-decreasing function in terms of the depth, we can assure that that solution will always be optimal, but it is not the case of the problem proposed.

In mathematical terms, the time complexity can be formulated as:

$$1 + b^1 + b^2 + ... + b^d = O(b^d) \rightarrow O(3^d) \tag{1}$$

where:

d = shallowest depth of a goal node.
b = branching factor (3 posible actions).

Effectively, it is exponentially complex, because it requires to generate all the nodes in the shallower depths from the goal node. This brute force approach will process a large number of nodes, making it a bad idea to use it for large terrain problems.

Regarding the space complexity, and assuming that a graph-based search is performed, we find that it is also exponential:

$$O(b^{d-1}) + O(b^d) = O(b^d) \rightarrow O(3^d) \tag{2}$$

where:

[1]For this kind of analysis, we suppose that the problem is well formed and the coordinates of Gertrudis are inside the map. If not, no algorithm can reach a solution.

b^{d-1} = number of nodes explored.
b^d = number of nodes in the frontier.

Therefore, the algorithmic **complexity** in both terms of time and memory of this method **increase exponentially** with the size of the problem, so it is only feasible in small instances (e.g. the cities are very close).

1.3 Heuristic functions for A*

In this section, two heuristics will be formalized for informed search strategies. The first will be the Manhattan distance, and the second the Manhattan distance taking into account the minimum number of rotations to reach the goal position.

A heuristic function is a function that, given some specific knowledge about the domain (beyond the definition of the problem itself), provides with an approximation of the cost to reach the goal node from a certain node, so it serves as a criteria for selecting the next node to expand, guiding the search through, a priori, a good path.

The heuristic function will only be used in the A* algorithm, as it is the only informed search strategy developed to solve this problem. Choosing a proper heuristic is a major decision, as the cost-optimality depends on the admissibility and consistency of the heuristic.

Manhattan distance

The first heuristic taken into account is the Manhattan distance, a distance metric between two points in the space. It is usually used in problems where the movement can be done over a grid only in the vertical and horizontal axis (no diagonal movements), and it is defined as the sum of the absolute difference of each dimension between two points $a = (a_1, a_2), b = (b_1, b_2)$:

$$ManhattanDistance(a, b) = |a_1 - b_1| + |a_2 - b_2| \tag{3}$$

For this problem, we want to minimze the distance to Gertrudis, so $b = POS_G = (POS_{GX}, POS_{GY})$ and a will be the position of a node n to be calculated its distance:

$$h_1(n) = ManhattanDistance(POS_n, POS_G) \tag{4}$$

In this problem, the Manhattan distance leads to an admissible and consistent heuristic. In the first case, because for admissibility it is just required that the heuristic never overestimates the real cost of reaching the goal from the node. This is satisfied since Abdel can only move horizontally and vertically and the best case where Abdel is looking straight to the goal and every position has a cost of 1 (the minimum), then $h_1(n)$ is exactly the cost of the path. In other case, the real cost will be greater than the Manhattan distance because a rotation may be needed or a position with cost > 1 may be found.

On another hand, for being consistent or monotonical, the heuristic estimate must be always less than or equal to the estimated distance from any neighbouring node n' to the goal, plus the cost of reaching that neighbour:

$$h(n) <= cost(n, n') + h(n') \tag{5}$$

To prove that this condition is satisfied, the three possible cases have been taken into account:

- For those reachable nodes whose $h_1(n') > h_1(n)$, it means that the action separated us from the target (in terms of Manhattan distance). In this case, $h_1(n) < c(n, n') + h_1(n')$ because $h_1(n) < h_1(n')$ and $c(n, n') > 0$ because there are not negative costs.
- For those reachable nodes whose $h_1(n') < h_1(n)$, it means that the action got us closer to the target. In this case, $h_1(n) <= c(n, n') + h_1(n')$ because the action can, at most, bring us one step closer, but it cost is, at least, 1. So in that scenario $h_1(n) = c(n, n') + h_1(n')$ because $h_1(n) = h_1(n') - 1$ and $c(n, n') = 1$. For other scenarios, the cost of the action will be greater than 1 and $h_1(n)$ will be strictly lower than the right term.

- For those reachable nodes whose $h_1(n') = h_1(n)$, it means that the action applied was a rotation, so the position remains the same. In this case, $h_1(n) < c(n, n') + h_1(n')$ because $h_1(n) = h_1(n')$ and $c(n, n') = 1$.

It is recommended in the case of graph search to satisfy this property, in addition to admissibility, because it ensures that when a node is expanded, the path to reach it is the cheapest one, so no node will be explored twice because a new cheaper path to it has been found later. As a result, when the goal node is found we are sure that it was by the optimal path. If $h_1(n)$ is admissible but non-monotonical, then it can be the case where a node already explored is expanded again with lower cost.

Applying the heuristic to the problem, it would be suitable if there are no big regions very difficult to walk through in a straight line. The Manhattan distance tries to get us closer to the goal as possible, and it is said that the shortest path between two points is a straight line. We cannot move diagonal, but this distance works with the same philosophy, so if the target is in a straight line, then $h_1(n)$ will try to just follow the line. However, if that terrain is all 5, while the surroundings are 1-4, then it would be much more appropriate to deviate a little from the straight path, instead of heading directly to the destination. In this case, as the map is finite, a solution will always be found (**complete**), and by the characteristics of the algorithm and the heuristic, it will be the optimal.

Manhattan distance and rotations

The second heuristic is an extension of the previous one, adding the minimum number of orientation changes or rotations needed to reach the goal position to the Manhattan distance. Due to the fact that in the Section 3.3 the mathematical explanation has been given for a little variation of the problem, we will avoid deepening in the admissibility and consistency demonstration of this new heuristic. In summary, it is admissible because the cheapest path will contain the minimum number of rotations to align with the destination, and the Manhattan distance, again, models the cheapest path of all 1's. It is also consistent because when executing MOVE, the number of rotations does not decrease so the same cases applies as for $h_1(n)$, and for ROTATE_L or ROTATE_R their cost is 1 and they can reduce at most 1 the heuristic on the new node (one less rotation), so it is impossible that $h_2(n) > 1 + h_2(n')$ because $h_2(n) = h_2(n')$ or $h_2(n) \pm 1 = h_2(n')$.

About its suitability, it is clear that it gives a better estimation of the real path cost, since it takes into account the rotations instead of just the movements. Therefore, it should guide much well the search process, increasing the efficiency with respect to the simple Manhattan distance by discarding nodes with a state that is not beneficial.

The heuristic, specifically, is defined as follows:

$$h_2(n) = h_1(n) + TotalRotations(n) \tag{6}$$

where:

$TotalRotations(n)$ = least number of rotations (0, 1 or 2) in a path that reaches to the goal from n.

A pseudocode has been provided in the Algorithm 1 in order to understand how these term can be calculated.

Algorithm 1 Computation of the TotalRotations term

$North \leftarrow 0$
$East \leftarrow 1$
$South \leftarrow 2$
$West \leftarrow 3$

procedure R(x, y) ▷ $x, y \in \{North, East, South, West\}$
  return $max(|x - y| \mod 2, |x - y| \mod 3)$
end procedure

$n \leftarrow$ *node to calculate the rotations from*

$north\_south \leftarrow (n_x < POS_{GX}) * 2$
$ver\_rotations \leftarrow r(n_{or}, north\_south) * (n_x \neq POS_{GX})$ ▷ n_{or} is the orientation of the node n

$west\_east \leftarrow (n_y < POS_{GY}) * 2 + 1$
$hor\_rotations \leftarrow r(n_{or}, west\_east) * (n_y \neq POS_{GY})$

$TotalRotations \leftarrow min(ver\_rotations + hor\_rotations, 2)$

2 Results

This second section will present the results obtained when solving the stated problem using the different approaches formalized in the previous section.

2.1 Execution examples

In this subsection, an example of the complete execution trace of each one of the implemented search algorithms will be shown for a chosen example of a 5×5 map. Specifically, the file `5x5_1.txt` has been selected for testing. The trace can be visualized in the Table 3 for the informed search methods and in the Table 4 for the blind ones.

A* (h1)
(0, 0, None, 3, (2, 2, North))
ROTATE_R
(1, 1, ROTATE_R, 3, (2, 2, East))
MOVE
(2, 4, MOVE, 2, (2, 3, East))
MOVE
(3, 7, MOVE, 1, (2, 4, East)
ROTATE_R
(4, 8, ROTATE_R, 1, (2, 4, South))
MOVE
(5, 9, MOVE, 0, (3, 4, South))

A* (h2)
(0, 0, None, 5, (2, 2, North))
ROTATE_R
(1, 1, ROTATE_R, 4, (2, 2, East))
MOVE
(2, 4, MOVE, 3, (2, 3, East))
MOVE
(3, 7, MOVE, 2, (2, 4, East))
ROTATE_R
(4, 8, ROTATE_R, 1, (2, 4, South)
MOVE
(5, 9, MOVE, 0, (3, 4, South))

Table 3: Execution trace of informed search methods in the first map of dimension 5×5

breadth-first
(0, 0, None, (2, 2, North))
ROTATE_R
(1, 1, ROTATE_R, (2, 2, East))
MOVE
(2, 4, MOVE, (2, 3, East))
MOVE
(3, 7, MOVE, (2, 4, East))
ROTATE_R
(4, 8, ROTATE_R, (2, 4, South))
MOVE
(5, 9, MOVE, (3, 4, South))

depth-first
(0, 0, None, (2, 2, North))
MOVE
(1, 2, MOVE, (1, 2, North))
MOVE
(2, 3, MOVE, (0, 2, North))
ROTATE_R
(3, 4, ROTATE_R, (0, 2, East))
MOVE
(4, 7, MOVE, (0, 3, East))
MOVE
(5, 8, MOVE, (0, 4, East))
ROTATE_R
(6, 9, ROTATE_R, (0, 4, South))
MOVE
(7, 11, MOVE, (1, 4, South))
MOVE
(8, 14, MOVE, (2, 4, South))
MOVE
(9, 15, MOVE, (3, 4, South)

Table 4: Execution trace of blind search methods in the first map of dimension 5×5

2.2 Performance comparison

In this subsection, the performance measurements of several simulations using each algorithm are shown. To evaluate each algorithm, four different random maps were generated with sizes 3×3, 5×5, 7×7 and 9×9, respectively. For each of them, five different random starting positions were set for Abdel and Gertru, so a total of $4 \cdot 5 = 20$ simulations were carried out for each algorithm. In the Table 5, the average results on every dimension can be visualized, where each column represents:

- **d**: depth of the exploration tree-graph in which the solution has been found.
- **g**: cost of the solution path obtained.
- **#E**: final number of nodes stored in the explored list.
- **#F**: final number of nodes stored in the frontier.

	d	**g**	**#E**	**#F**
Width	**2.40**	6.20	**4.20**	**2.40**
Depth	8.00	17.00	11.60	9.20
A* (h1)	2.60	5.80	6.80	3.80
A* (h2)	2.60	5.80	4.40	3.80

(a) Dimension 3×3

	d	**g**	**#E**	**#F**
Width	**3.60**	7.60	17.60	12.00
Depth	14.40	34.60	14.40	21.00
A* (h1)	3.80	7.60	17.60	10.20
A* (h2)	3.80	7.60	**11.40**	**8.80**

(b) Dimension 5×5

	d	**g**	**#E**	**#F**
Width	**7.80**	20.60	91.00	23.20
Depth	30.60	83.20	**30.60**	48.00
A* (h1)	9.20	16.40	74.00	23.75
A* (h2)	9.20	16.40	59.80	**21.40**

(c) Dimension 7×7

	d	**g**	**#E**	**#F**
Width	**9.00**	25.80	132.60	22.40
Depth	36.00	96.00	**49.00**	60.00
A* (h1)	9.80	22.40	139.00	**16.80**
A* (h2)	9.80	22.40	125.40	17.40

(d) Dimension 9×9

Table 5: Comparative tables of performance of the search methods for different map dimensions

3 Discussion

After collecting the results in the previous section, now a brief discussion about the suitability of each search method and possible variations of the problem are presented.

3.1 Advantages and disadvantages of the search methods

First of all, we will analyse how the different search methods behaved according to the varying size of the map in the problem proposed. Specifically, the blind search methods will be discussed before moving to the results of the A* algorithm, that will be compared with respect to the heuristic functions $h_1(n)$ and $h_2(n)$. Finally, we will present the main conclusions about the suitability of each method regarding this problem.

From the Table 5, we can easily check that **depth-first** tends to be the algorithm that explores less nodes as the size of the map grows up, although for the dimension 3×3 is exactly the opposite and in the dimension 5×5 is beat by A* with $h_2(n)$. For the other two dimensions, it explores just half or even the third part of the nodes compared with the other algorithms. However, the path found is really deep, $\sim 3-4$ bigger than the other paths, as well as the path cost, that is ~ 4 times bigger. This is the effect of expanding one full path before checking another one, since it will always expand as many nodes as needed until no more new nodes can be created. The size of the frontier, in contrast with the explored list, is the biggest among the 4 alternatives, as the algorithm only pops a new element when the current path cannot be extended (no new successors).

With respect to the **breadth-first** algorithm, it is the one that always find the shortest path (less deep), as the full depth level is explored at a certain moment before expanding a node of the next depth level. Even so, it is not cost optimal and in big maps it requires to explore a lot of nodes (e.g. dimension 7×7), so it should be used only for small instances of the problem.

For the **informed search**, the optimal solution is always found, so both A* search methods return the best path cost compared to the rest. Of course, the depth is not as good as in the case of breadth-first (although there is not a huge difference), but it is much better than in the case of depth-first. The second heuristic $h_2(n)$ improves the number of nodes explored with respect to $h_1(n)$, while the nodes in the frontier are approximately equal, so we conclude that it is always a better choice. This makes sense because, in the end, $h_2(n)$ is just an expansion of $h_1(n)$ with more information that can provide with more accurate approximations, so the number of bad node choices to explore are reduced.

For the problem proposed, it would be advisable to choose an informed search method whenever the information is available, because the cost is always optimal and the depth of the path found is almost optimal. Despite that, if we want to really optimize the depth, the breadth-first is the best option. In both cases, the number of explored nodes is large, so if some kind of cost would be involved in the expansion of a node, then the depth-first algorithm is better, at cost of finding a very deep and expensive path. However, the breadth-first would be a very good option if the cost of the terrain follows a non-decreasing function in terms of depth, that could be the case if, for example, the initial state is at the center of the map and the difficulty of going to a more external position is given by the distance to that center due to some kind of defense built around the city.

3.2 Best heuristic function

As it has already been commented previously, the A* algorithm with $h_2(n)$ always returns better results in terms of efficiency compared to $h_1(n)$. This happens because the $h_2(n)$ is just a expansion of $h_1(n)$ (see Equation 6), adding the minimum number of rotations to reach the goal position to the Manhattan distance, so the real path cost is better estimated and some routes can be discarded from expansion if they require more changes in the orientation. In addition, $h_2(n)$ will be equal to $h_1(n)$ only in the goal node, since they must have an heuristic cost of 0, and when Abdul is already looking straight to Jertru and no rotations are needed. In the rest of cases, it will be strictly greater than $h_1(n)$ because one rotation will be needed, at least.

3.3 Map containing positions with zero cost

Let us consider a slightly different problem where there is the possibility that the map could contain positions with zero cost. Using a graph search, the implementation chosen to approach the initial problem, all the algorithms would find a solution because the map is finite. In the worst case, where all the nodes but the target one have been found or expanded, the last possible node will correspond to the goal.

For the blind search, as in the initial problem, the algorithms are not expected to find the optimal solution (but they could) because they just expand the nodes in the order that the algorithm itself specifies, regardless of the cost of the nodes.

In the case of A*, our heuristic functions will not work well, so in this case the solution may not be optimal. This is because the Manhattan distance could be overestimating the real path cost. For example, in a path where all the nodes have zero cost and where Abdul is already oriented to the destination, the path cost will be zero, while the Manhattan distance would be returning some value > 0 because the difference in the coordinates is always > 0 (except in the target). However, notice that the term that models the number of rotations is not affected by this variation, since the rotations still have a cost of 1.

One adjustment that can be performed to still have an admissible and consistent heuristic function is to replace the $ManhattanDistance$ by the minimum cost of the adjacent cells on the terrain. However, that could return a different value from 0 in the target if it is surrounded by non-zero cost neighbours, so a little check is performed to know if it is the goal node with the $ManhattanDistance$:

$$h_3(n) = TotalRotations + min(\{adj(n)\}) \cdot (ManhattanDistance \neq 0) \tag{7}$$

where $adj(n)$ corresponds to the cost associated to move to any of the adjacent cells of n in the map. This new term is cancelled when the $ManhattanDistance = 0$, this is, when the n is the goal node. We can assure that from a certain position, the minimum distance to reach the goal is the minimum cost of its adjacent cells plus the rotations. No distance metric can be used in this variation because it can happen that after exploring a new node, a zero cost path may be found right after it, so the only cost associated to the path from the original cell would be the cost of reaching the non-zero cost node (plus rotations).

The new heuristic proposed in the Equation 7 is **admissible** because the minimum number of rotations and the minimum cost of the neighbours cells are only taken into account, so it is impossible to overestimate the real path cost (when there are zero cost cells surrounding the heuristic will return only the rotations). It also satisfies the **consistency** property. It must be pointed out that the real neighbour nodes are the one reachable after applying a rotation or a forward movement, and not the adjacent cells in the map.

Now, we are going to prove that the inequation $h_3(n) \leq c(n, n') + h_3(n')$ is satisfied, so the heuristic function is consistent. Suppose that the action done is MOVE. This action does not reduce the number of rotations to reach the goal. Moreover, it may add 1 more rotation if the node was in the same row or column as the goal node and it is left. Then:

$$TotalRotations(n) + min(\{adj(n)\}) \leq c(n, n') + min\{adj(n')\} + TotalRotations(n') \tag{8}$$

because $min(\{adj(n)\}) \leq c(n, n')$ since $c(n, n') \in \{adj(n)\}$, $TotalRotations(n) \leq TotalRotations(n')$ since the number of rotations cannot decrease, and there are no negative costs for $\{adj(n')\}$.

For the rotation actions, the term $min(\{adj(n)\})$ remains the same for the heuristic of both nodes, and only the number of possible rotations can vary, while $c(n, n') = 1$ is the cost associated to the rotate action:

$$TotalRotations(n) \leq 1 + TotalRotations(n')$$

The rotation action can provoke 3 different situations:

- If the rotation applied gets us closer to the goal node, then the number of rotations in the new node is reduced by 1:

$$TotalRotations(n) \leq 1 + (TotalRotations(n) - 1) \rightarrow 0 \leq 0 \tag{9}$$

  that is always true.

- If the rotation applied does not improve the situation:

$$TotalRotations(n) \leq 1 + TotalRotations(n) \rightarrow 0 \leq 1 \tag{10}$$

  that is always true.

- Finally, if the rotation applied is a bad option and aparts us from the goal node:

$$TotalRotations(n) \leq 1 + (TotalRotations(n) + 1) \rightarrow 0 \leq 2 \tag{11}$$

  that is always true.

Notice that in this demonstration it has not been taken into account if the neighbour node is the goal node, so $ManhattanDistance = 0$. The reasoning for the rotation actions is still valid because the goal node cannot be reached with a rotation, only with a move and when we are oriented to it, so $TotalRotations(n) = 0$. As $h_3(n') = 0$ if n' is the goal node, the inequation would look like:

$$min(\{adj(n)\}) \leq c(n, n') \tag{12}$$

that is always true because $c(n, n') \in \{adj(n)\}$. If n is the goal node, the left part of the inequation is 0, that is always $\leq$ than any other value in the right part of the inequation.

Basically, this new heuristic will give priority to the nodes that can explore a zero cost path. This makes sense because, intuitively, all non-zero cost nodes are actually neighbours of others if they are connected with zeros (at most one rotation may be needed in addition to travel from one to another), so this null paths can be considered as a kind of shortcuts and should be explored because they could already be connecting with the goal node. However, notice that with this approach we could be strongly underestimating the real path cost since only the cost of the adjacent cells and the rotations is taken into account, no matter how far from the goal node it is n, so we are probably loosing a lot of efficiency as it will be required to expand more nodes, compared to the good fit of the heuristics for the initial problem.

Appendices

A Software content

Attached to this report, four Python files are also included for the submission:

- `map.py`: it contains a `Map` class to generate random maps, load a map from a file or write it on a file.

  ```
  # Example generating a new 3x3 map and saving it on a file
  python3 map.py --dims 3 3 --initial 0 0 --goal 2 2 --min-cost 1 --max-cost 5
      --output file.txt
  ```

- `search.py`: it implements all the graph search algorithms, the interface that every problem must declare and the `Node` class to hold all the relevant information and actions that can be executed over a specific node.

- `go_to_jisrael_problem.py`: implements the problem interface with the specific characteristics proposed for this task, as well as the `State` class that holds the information of a specific state for this problem.

  ```
  # Example running the A* algorithm with h1 from the map in file.txt
  python3 go_to_jisrael_problem.py file.txt --algorithm Astar --h h1
  ```

  The name of the command line algorithms are `BreadthFirst`, `DepthFirst` and `Astar`, while the possible heuristic functions are `h1` and `h2`. Notice that you can specify an heuristic function with a blind search method as well, but it will not be taken into account. If you want to debug all the search process (iteration number, state of the frontier, explored list, number of nodes created) you can add the flag `--debug`.

- `utils.py`: some utilities for the implementation process.

There are also some map examples under the folder `maps/`, where the files used for the performance measurements are located.

B Software architecture

Our software can be divided into three parts:

- Problem specific data: implements the specific `Problem` class with all the problem logic (actions available, heuristic functions, goal test...) and the specific `State` class with the information needed for our states.
- Search algorithms: contains the implementation of the different searching methods, as well as the definition of a node.
- Utils: includes the `Map` class and the `utils.py` file.

Some class diagrams are drawn in the Figure 1 for the first part and in the Figure 2 for the second part, as they are the most interesting ones.

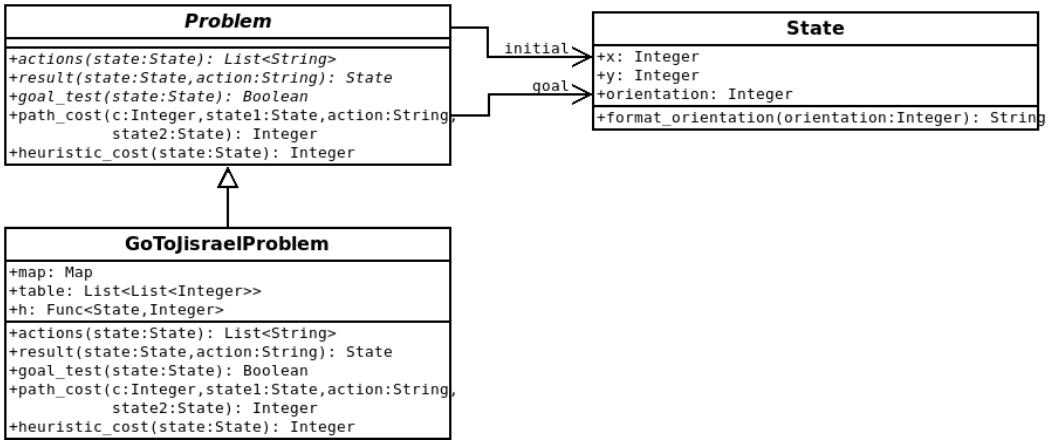

Figure 1: Diagram of classes of the problem proposed.

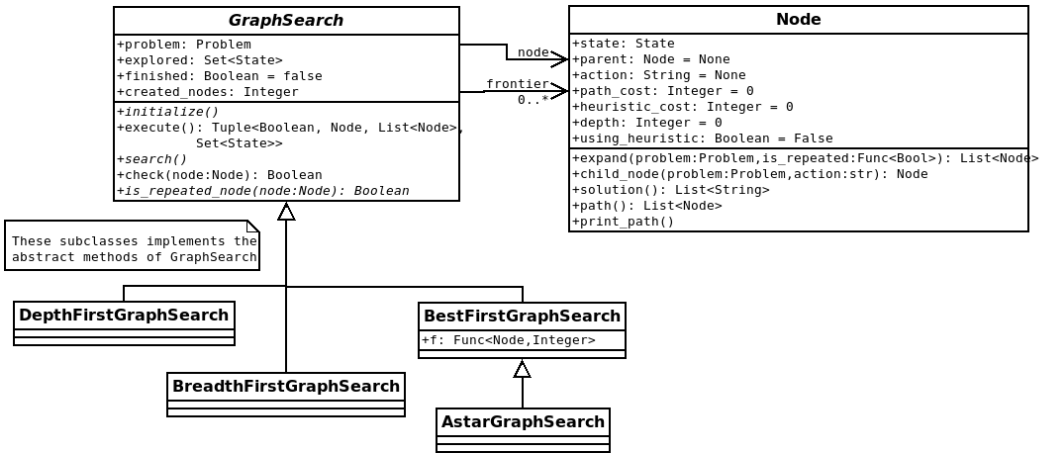

Figure 2: Diagram of classes of the search methods.

C Details of the search methods

Our code has been built upon the already existing `aima-python`[2] project, which is based on [1]. The most interesting details about the search methods are the following:

- A **graph search** is implemented, so a frontier and a explored list is used by all the algorithms. The `GraphSearch` class implements the main loop of the search process in the method `execute(self)`, and logs all the relevant debug information. This method is the responsible for calling the specific behaviour of the algorithm, that must be implemented under the `search(self)` method. It returns if it has found a solution, the last examined node, the frontier and the explored list.

- The implementation of the frontier is the one that varies among them. In breadth-first, a FIFO list (queue) is used, so the first added **nodes** are the ones that are first expanded. In depth-first, a LIFO list (stack) is used, so the last added nodes are the ones that are first expanded. Lastly, in A* a priority queue is used, so the nodes with the minimum $f(n) = g(n) + h(n)$ are expanded first. The explored list is just a set where no repeated **states** are allowed.

- Notice that if the possible actions are returned in the order ROTATE_R $\rightarrow$ ROTATE_L $\rightarrow$ MOVE, the LIFO list will pop MOVE before the other ones, while the FIFO will pop ROTATE_R first. To keep the consistency of the expansion order (left to right or right to left) between both blind search methods, the list of successors added to the frontier of the breadth-first algorithm is reversed, so it will prioritize MOVE over ROTATE_L and ROTATE_L over ROTATE_R, just as the pop function works in the depth-first algorithm.

- Following the common behaviour of the algorithms, breadth-first stops when a goal node is **found**, while the others stop when a goal node is **expanded**:

```
# breadth-first solution check

... expansion of current node ...
for child in successors:
    if not self.is_repeated_node(child):
        if self.check(child):
            self.node = child
            return
        self.frontier.append(child)
```

```
# depth-first and A* solution check

self.node = self.frontier.pop()
if self.check(self.node):
    return
... expansion of current node ...
```

- The A* algorithm is just the best-first algorithm whose $f(n)$ is the sum of the path cost to n plus its heuristic value $h(n)$.

```
class AstarGraphSearch(BestFirstGraphSearch):
    def __init__(self, problem):
        super().__init__(problem)
        self.f = lambda n: n.path_cost + problem.heuristic_cost(n.state)
```

- For A*, one node is considered as repeated (it is not a new possible successor) if its state is already in the explored list or if there is another node with the same state but lower path cost in the frontier. For the blind search methods, it is repeated if some node has the same state in the frontier or if the state has been explored:

```
# Blind search repeated nodes checking
def is_repeated_node(self, node):
    return node.state in self.explored
        or node in self.frontier
```

```
# A* repeated nodes checking
def is_repeated_node(self, node):
    if node in self.frontier:
        return self.f(node) >= self.frontier[node]
    return node.state in self.explored
```

[2] https://github.com/aimacode/aima-python

- New successors are created after a node is retrieved from the frontier and its method `expand(self,problem,is_repeated)` is called. This method gets the possible actions from the state of the node and calculates the reachable nodes from each action with `child_node(self,problem,action)`. The parameter `is_repeated` is only used in order to not print repeated successors, but they are actually discarded later by the specific algorithm later so it can count the total number of nodes created.

```
def expand(self, problem, is_repeated):
    logging.debug(f"Expanding {self}")
    children = [self.child_node(problem, action)
                for action in problem.actions(self.state)]
    logging.debug(f"New successors: {', '.join(str(child) for child in
        children if not is_repeated(child))}")
    return children

def child_node(self, problem, action):
    next_state = problem.result(self.state, action)
    next_node = Node(next_state, self, action, problem.path_cost(self.path_cost, self.state,
        action, next_state), problem.heuristic_cost(next_state))
    return next_node
```

Bibliography

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall, 2010.