Report: Artificial Intelligence 1 - Jeu De Labyrinthe

Nguyen Viet Sang, Le Tan Nhat Linh
{viet-sang.nguyen, tan-nhat-linh.le}@etu.unilim.fr

Master 1 Cryptis, Faculty of Science and Technique, University of Limoges

1 Introduction

In this report, we present our implementation to solve the Labyrinth Problem. In section 2.1, we introduce our approach with the definition of problem and structure of a node used in search tree. This approach is referenced to the textbook [1]. The details of applying Best-First Search and A* algorithms are shown in section 3. We also discuss and compare the results of these two algorithms in section 5. Not only that, we also implement Depth-First Search and Breadth-First Search algorithms to search in case that we do know the position of the goal state.

For the results, we successfully applied Best-First Search and A* algorithms to solve the problem of labyrinth. We also build an attractive Graphic User Interface to demonstrate our solutions as shown in figure 2a.

2 Labyrinthe Problem

For solving this problem, we define two main class. The first one is to specify the problem which includes necessary characteristics of an agent in the field of solving problems by searching. The second one is the node holding some useful information to build a search tree [1].

After reading the information from a given file, we have an initial state, a goal state and a labyrinth for the problem. In our convention, a state is a tuple `(y,x)` where `y, x` (starting from 0) denote the line and the column of the current position respectively.

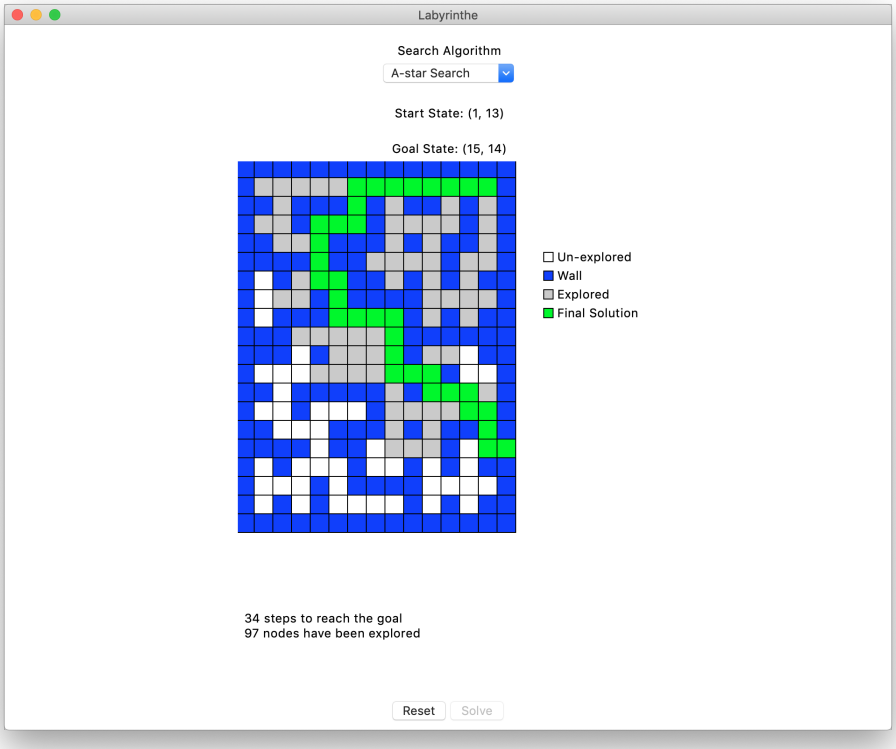
2.1 General Problem

We define a general class for this problem named `class Labyrinthe`. There is a constructor with three parameters to initialize the problem which are an `initial state` (position at the beginning), a `matrix labyrinthe` representing the labyrinth read from input file, and the `goal state` (goal position).

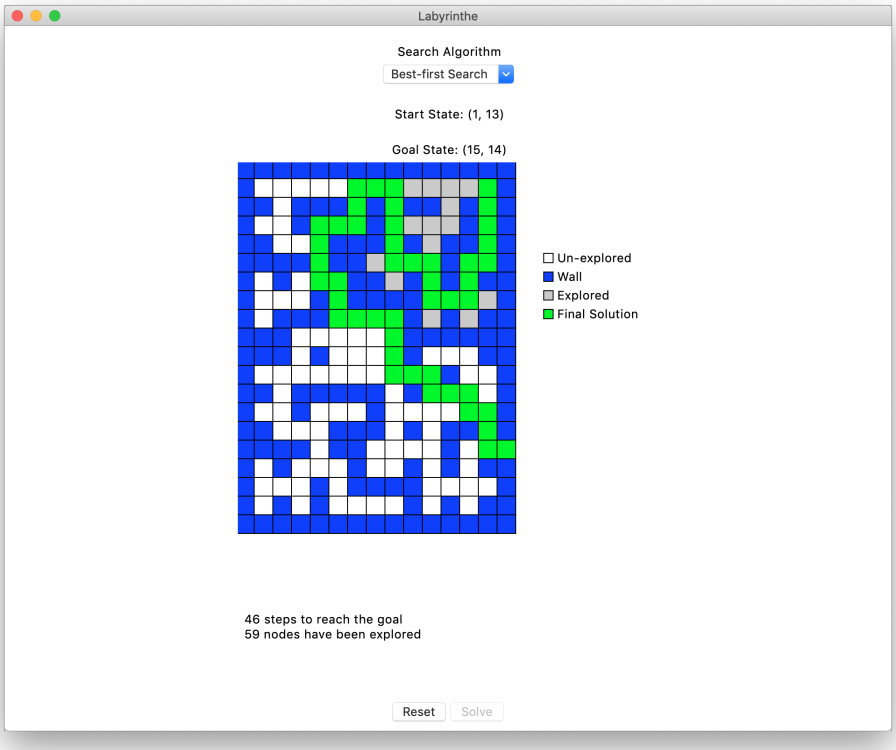
In this class, we define five methods as below:

- `actions(state)`: returns a list of possible next actions that it can move on in from the current `state`. For example, if from the current position, there is a wall on the left and it can move up or down or to the right, then this function will return the list `['UP', 'DOWN', 'RIGHT']`. This helps to expand the current state to neighbor states.
- `result(state, action)`: returns a new state when taking an `action` for a given `state`. For instance, if the current state is `(3,2)` and we take the action `'UP'`, then the new state is `(2,2)`.
- `is_goal(state)`: returns `true` if the considered `state` is equivalent to the goal state.
- `g(cost, state1, action, state2)`: parameter `cost` represents the number of steps taken from the original position to the current state (`state1`). This function returns return the new cost if from `state1`, we take the `action` to move to `state2`. By this way, we could compare the costs of all neighbors of the current state to choose the best neighbor to move on in the next step. In this project, we return `cost + 1` for this function.
- `h(state)`: returns the distance from the current state to the goal state. Suppose (y, x) and (g_y, g_x) correspond to the current state and the goal state. We calculate the distance by the formula $\sqrt{(y - g_y)^2 + (x - g_x)^2}$

The above problem is specified for this project. In general, we could define a search problem in a similar way.



(a) Solving with A* algorithm



(b) Solving with Best-First Search algorithm

Fig. 1: GUI of Searching in Labyrinth Problem

2.2 Node

A node contains the following attributes:

- `state`: is the current state (position).
- `parent`: is the parent node of this node.
- `action`: is the action which the parent node has taken to move to this node. We utilize this attribute to trace back and get the path of final solution once we have done the searching work.
- `path_cost`: is the cost to reach to the state of this node from the beginning.
- `depth`: is the depth of the tree search from the root to this node.

3 Search Algorithms

We approach this problem by using tree search. Hence, it is necessary to construct a priority queue for the work of expansion with a certain node. We define a class called `PriorityQueue`. This permits us to append a new node when expanding the tree and pop the node with the highest priority to explore. We define the priority of a node is the value of `f(node)` where `f(node) = g(node) + h(node)` in A* algorithm, for instance.

Our aims is to make a general search for both Best-First Search and A*. The difference is the parameter of cost function `f(node)` to pass to the search. First, we have the node of initial state in our priority queue. Then, we pop it from the queue and check whether it is the goal state or not. If not, we expand it to consider the next movement from there. The expansion works thanks to the functions defined in general problem which is discussed in the previous section. After expanding the current node, we get a set of children and push them to the priority queue one by one. Then, we continue popping the next node in the queue which has the highest priority at the moment and do the similar work until we reach the goal.

Listing 1.1 shows the implementation of our search algorithm. Line 2 is a function to memorize the value of `f(node)` in the node itself. This aims to use the value for the next times without calculating it again. At the end, if a solution is found (line 10-12), the last node of the path and the explored set will be returned. From the last node, we could trace back the path since each node contains the attribute `parent` which holding the parent node.

```
1  def search(problem, f):
2      f = memorize(f, f_attr='f') # keep value of f in node for next usage
3      node = Node(problem.initial)
4      queue = PriorityQueue('min', f)
5      queue.append(node)
6      explored = set()
7
8      while queue:
9          node = queue.pop()
10         if problem.is_goal(node.state):
11             print(len(explored), "nodes have been expanded and", len(queue),
       "nodes remain in the queue")
12             return node, explored
13
14         explored.add(node.state)
15         for child in node.expand(problem):
16             if child.state not in explored and child not in queue:
17                 queue.append(child)
18             elif child in queue:
19                 if f(child) < queue[child]:
20                     del queue[child]
21                     queue.append(child)
```

Listing 1.1: Search algorithm

3.1 Best-First Search (Meilleur D'abord)

We define that the function f is the distance from the current state to the goal state. By this heuristic, at a certain state (y, x), we choose the next state which has the shortest distance to the goal state (g_y, g_x). In other words, $f = \sqrt{(y - g_y)^2 + (x - g_x)^2}$. The smaller f is, the better movement to choose.

3.2 A*

The function f now contains two parts of g and h. We define g is the number of step that we have taken to move to the current state from the initial state. h is the estimated distance from the current stat to the goal state. It is also calculated by $h = \sqrt{(y - g_y)^2 + (x - g_x)^2}$. A now, $f = g + h$ is the function to pass to the parameter of search function.

3.3 Extra Work

In a similar way, we implement Depth-First Search and Breadth-First Search algorithms. These ones are implemented by applying the data structures of `stack` and `queue`. We could use the both algorithms in a scenario of lacking knowledge about the goal state, hence we could not apply any heuristic. We try to implement these two algorithm in a manner that they could be easily adapted to the problem we define in section 2.1.

4 Graphic User Interface (GUI)

We use the library `tkinter` to build the GUI. As we can see in the figure 2a, the information of initial state, goal state, number of nodes explored, number of steps to reach the goal are shown in the window. This program allows users to choose an algorithm to execute.

5 Results

We compare the results returned by A* algorithm and Best-First Search algorithm as shown in figure 2a. We configure the same conditions including initial state $(1, 13)$ and goal state $(15, 14)$. As we can see, the solution of A* algorithm costs 34 steps to reach the goal while Best-First Search algorithm takes 46 steps. Despite the number of steps when solving with Best-First Search algorithm is more than the one of A* algorithm, it explores fewer nodes than A* algorithm (59 nodes comparing to 97 nodes).

We notice that there is a trick in the initial state which we configure above. We could easily infer that both algorithms will choose to go `'DOWN'` first and it is clearly not the best solution. If there is no trick in the way we configure the initial state an goal state as shown in figure 2, our experiments point out that both algorithms give the same path to reach the goal (18 steps). But there is a big difference here. The A* algorithm explores 44 nodes while the Best-First Search algorithm only explore 17 nodes (not including the goal) to find the best solution.

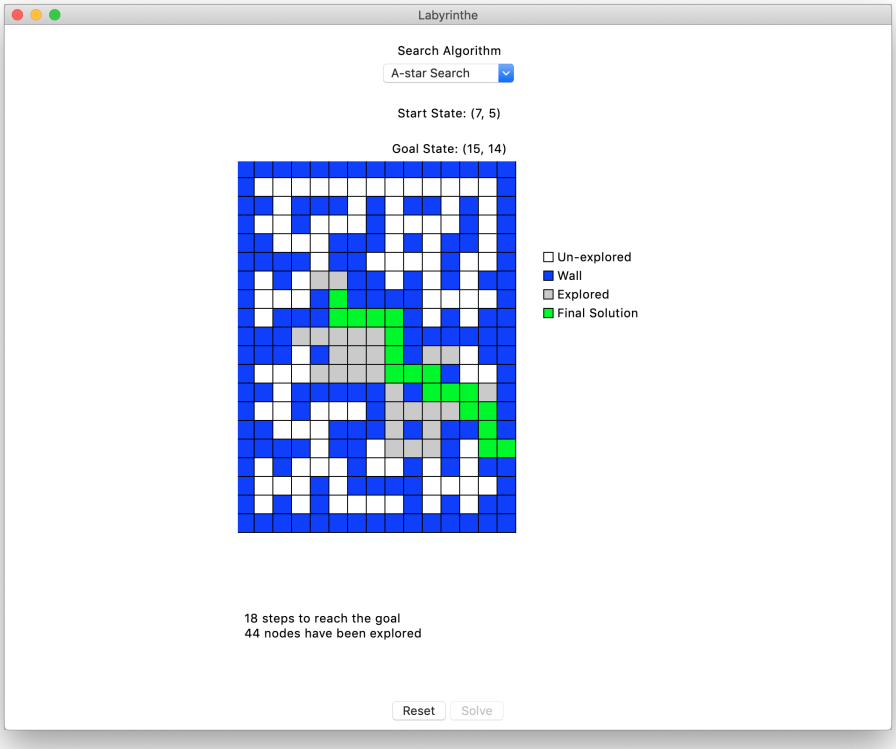
From the comparison above, we conclude that in many difficult cases, the A* algorithm returns the better path to reach the goal (with fewer steps). But in almost configuration, the number of nodes that A* algorithm explore is larger than the one of Best-First Search. This also leads to the time taken by A* algorithm to search the best path is more than the other one's.

6 Conclusion

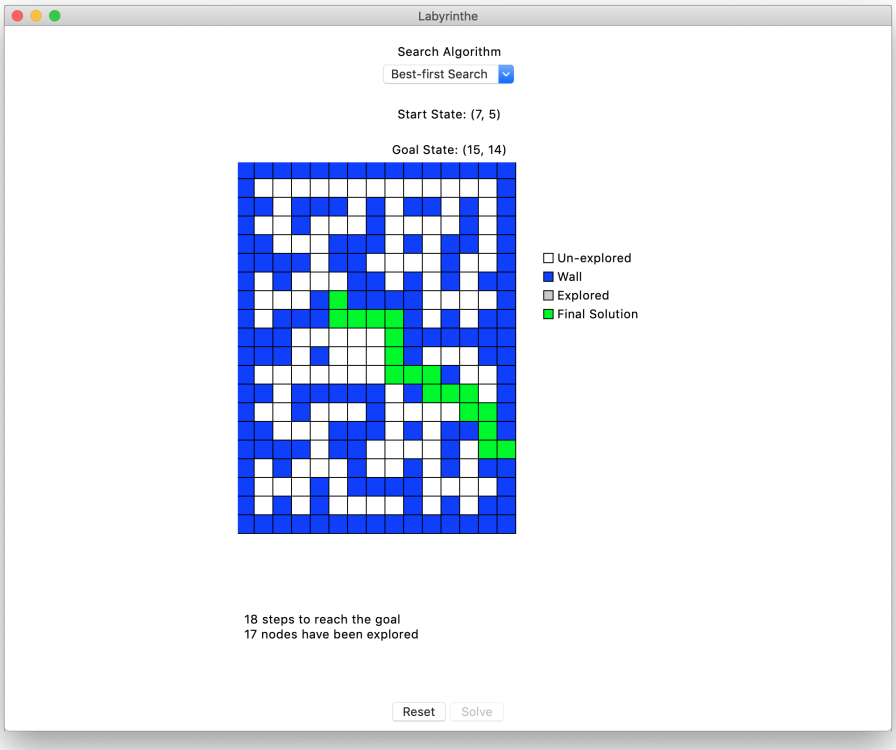
In this project, we successfully applied A* algorithm and Best-First Search algorithm to solve the labyrinth problem. We also build an attractive GUI to show the solutions given by these algorithms. Finally, we compare the results to prove that A* algorithm often provides a better solution but explores more nodes than the Best-First Search algorithm.

References

1. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Series in Artificial Intelligence, Prentice Hall, Upper Saddle River, NJ, third edn. (2010), http://aima.cs.berkeley.edu/



(a) Solving with A* algorithm



(b) Solving with Best-First Search algorithm

Fig. 2: Searching in no trick condition