Ruby Case Study

Gavriilidis Iason

***Abstract*—Uninformed search strategy for maze solving scenarios is a problem discussed in Artificial Intelligence, also known as blind search. In this case study, we explore possible approaches for the following problem: a robot is placed in an arbitrary position of a two-dimensional map (viewed as a rectangular grid) and its mission is to find a goal point (also arbitrarily placed). The difficulty of this problem arises from the fact that the robot has no knowledge either of the map or the goal position. We derive to a plausible solution which take advantage of the robot's view, memory and intelligence.**

I. Introduction

GRAPH problems pervade computer science, and algorithms for working with them are fundamental to the field. Many interesting computational problems are couched in term of graphs. This problem can be transformed into a graph searching problem. Most commonly discussed algorithms are informed, in the sense that the graph is considered known a priori. The use of heuristics in those cases is a big advantage in finding optimal and low complexity algorithms. Unfortunately, in the problem presented, these methods cannot be applied and we seek a strategy that perform better than a random walk. We propose strategies pointing that direction, and we try to gradually build up an algorithm.

II. Search Algorithms

The most common algorithms for searching a graph, are *Breadth First Search* (BFS) and *Depth First Search* (DFS). These algorithms are the archetype for many other important graph algorithms.

```
BFS(G, s)
 1 for each vertex u ∈ G.V
 2     u.color = WHITE
 3     u.d = ∞
 4     u.π = NIL
 5 s.color = GRAY
 6 s.d = 0
 7 s.π = NI:
 8 Q = ∅
 9 ENQUEUE(Q, s)
10 while Q ≠ ∅
11     u = DEQUEUE(Q)
12     for each v ∈ G.Adj[u]
13         if v.color == WHITE
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUE(Q, v)
18         u.color = BLACK
```

The pseudocode from *Introduction to Algorithms* [1]*(p. 595-612)* of the BFS algorithm uses a first-in, first-out queue Q. A recursive algorithm is also presented for DFS , but we can work on an iterative algorithm just by changing the position the elements are queued: DFS enqueues on the front where BFS enqueues on the back.

Another more abstract formulation which fits our problem better is from *Artificial Intelligence: A Modern Approach* [2]*(p. 55-90)*. In this approach the graph isn't considered known a priori. Here we disclose the high level pseudocode which consists the foundation of our approach.

```
GENERAL−SEARCH(PROBLEM, STRATEGY)
− returns a solution or failure
 1 initialize the search tree using the initial state of problem
 2 loop do
 3     if there are no candidates for expansion
 .     − then return failure
 4     choose a leaf node for expansion according to strategy
 5     if the node contains goal state
 .     − then return solution
 6     else expand the node
 .     − add the resulting nodes to the search tree
 7 end
```

There are two key points to analyse further:

- How to choose a leaf node to expand? How is this translated to the robot's movement?
- How to choose which nodes and in which order are inserted in the search tree?

Let us consider a different problem and analyse the behaviour for the two algorithms. The robot is placed on the root of a complete binary tree and tries to explore it (exhaustive search).

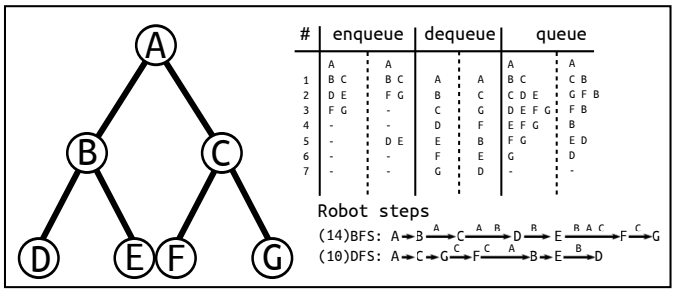

Fig. 1. Robot's moves for exhausting search of a binary tree

As we can see even on a binary tree with $depth = 2$, using the BFS strategy is not optimal, in respect of robot's total moves (the robot will visit all the nodes of a given depth before moving deeper). Therefore it is reasonable to choose a DFS strategy for this problem. We would also meet the same conclusion in the case of a blind search, as the exhaustive search is the worst-case scenario for a blind search.

The problem shown, is closely related to our problem. Moving on a two-dimensional map, where *UP*, *DOWN*, *LEFT*, *RIGHT* moves are permitted, can be translated to a $max - degree = 4$ graph.

A more detailed high level description of DFS appears below. At each step, the robot moves towards an undiscovered position. If it meets a dead-end (every possible movement has already made), it returns to the last found alternative.

```
DFS−SEARCH
1 Insert the starting position to the Queue
2 While Queue is not empty (there are candidate moves)
3     position = dequeue(Queue)
4     for each neighbour (up−down−right−left) if not a wall
5         if neighbour hasn't already visited
6             enqueue(Queue, neighbour)
```

Clearly, **the robot needs to store in memory a list of all the map positions already visited.**

Before moving to the next section, it is worth mentioning that other algorithms have been taken into consideration but turned down: *Uniform Cost Search* (UCS), *Depth Limited Search* (DLS), *Iterative Deepening Search* (IDS).

III. Using acquired knowledge - Jumping through positions

As far as the robot's behaviour is concerned, there are few unclear steps. The positions dequeued, in the case of a dead-end, are not neighbouring. Thus, the issue is how we could move to the dequeued position. The obvious approach is to reverse the path, however this could lead to unnecessary moves. This will become clear in the following example.

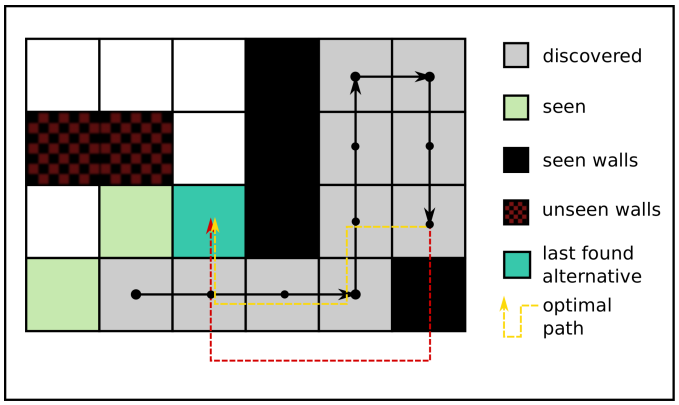

Fig. 2. Strategy for moving to a position using already acquired knowledge

In this example the robot is found in a dead-end, and has to move to the last known alternative point (and continue from there). In the naive approach (of reversing the path) 9 steps are needed. It's obvious that a shorter path exists (5 steps). We can take advantage of the robot's memory and **store the acquired knowledge** in order for a **path-finding algorithm to be used and find the shortest path.**

The most popular (because of its flexibility, performance and accuracy) algorithm used in path-finding is A^*

The properties of A^* [2]*(p. 97-104)* arise from the fact that it's a combination of $Dijkstra$'s algorithm (which favours nodes close to starting point) and *Greedy Best First Search* (which favours nodes close to goal point). The above is done by using a *knowledge - heuristic* cost function to determine the order in which the positions are visited (using a *Priority Queue*). This cost function, denoted $f(x)$, is the sum of two separate functions:

- *past* cost function: distance from the starting position to the current position x, denoted $g(x)$
- *future* cost function: heuristic estimate of the distance between current position x and goal, denoted $h(x)$

```
A∗−SEARCH
1 insert the startingPoint to the Priority Queue with priority = 0
2 set came from for starting Point to NULL
3 set cost so far for starting Point to 0
4 While Priority Queue is not empty
5     currentPoint = dequeue(Priority Queue)
6     if currentPoint = goalPoint
7         break
8     for each neighbour
9         new cost = cost so far + cost(currentPoint,neighbour)
10        if neighbour have cost so far and new cost < cost so far
11        set cost so far for neighbour to new cost
12        priority = new cost + heuristic(goal, neighbour)
13        insert neighbour to Priority Queue with priority = priority
14        set came from for neighbour to currentPoint
```

The previous pseudocode describes $A*$ algorithm, and in the following we give the pseudocode to reconstruct the shortest path.

```
RECONSTRUCT PATH
1 currentPoint = goalPoint
2 create a list and insert currentPoint
3 While currentPoint not goalPoint
4     current = came from( current
5     insert current to the list
```

In our implementation, the heuristic cost function is the commonly known $Manhattan\ distance$. It was chosen due to the fact that it offers accurate estimate of distance in rectangular grid maps.

IV. Using acquired knowledge - Decide on positions without visiting

In order to avoid visiting one position, it is required this position to be seen at least once and all its neighbours to be seen and/or discovered. Let us consider the two possibilities for that position:

- The position is labelled as the target:
  As the position is already seen, the algorithm would have already terminated in that case.
- The position is normal and accessible:
  The position is a candidate for expanding the robot's path. In the general case, the next move from this point will be on one of its neighbours. As all the neighbours are already be seen and/or discovered, the robot has considered those four possible paths. Therefore, we can avoid visiting this

position as neither new knowledge would be acquired nor any possible path would be ignored.

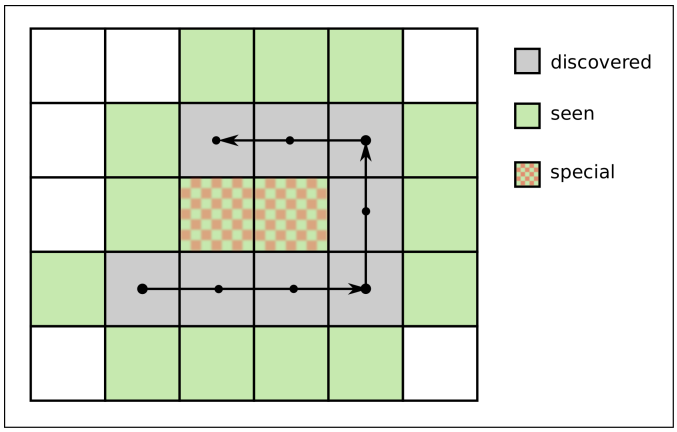

Fig. 3. Strategy for deciding on positions without visiting

A desired strategy would be to move through the map trying to increase robot's *visibility*. By increasing its visibility, positions are consequently cycled. The following example will make our intention clear and lead to a strategy for achieving increased visibility.

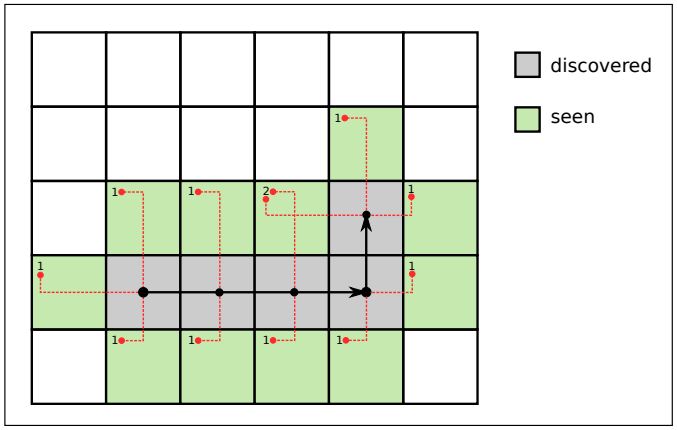

Fig. 4. Strategy for circling positions

The general strategy for achieving this, is to **sort all the neighbours in respect of how many times each has been seen.** Hence, the least seen neighbours come first and in this way the movement tries to increase visibility.

We have achieved better results by using the following reasoning: The robot **prioritise the position which keeps his direction unchanged**, if this position belongs to the least seen positions. When two candidate positions are seen equal times, the robot will randomly prioritise them.

The next example shows the results of this strategy:

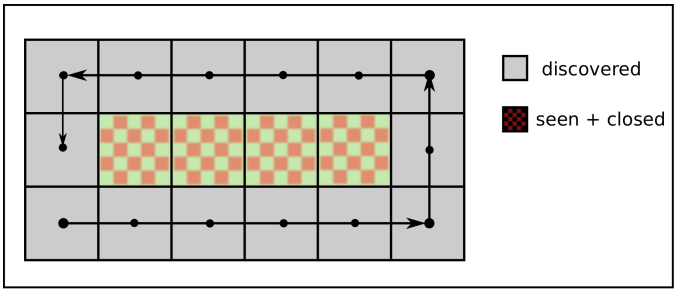

Fig. 5. Result of prioritising positions

V. Possible Improvements

It's worth mentioning some possible improvements to be implemented. These improvements follow the same ideas, but can reduce the computation complexity of the algorithm.

- Optimize the A* algorithm taking advantage the uniform-cost grid presented (Jump Point Search)
- Use a min Heap as the backbone for Priority Queue (Binary Heap)

VI. Conclusion

In a blind search problem the robot has no previous knowledge both of the map and the target. In this case study an effort has been made to take advantage of the limited visibility of the robot and to acquire knowledge of the map. Moreover, robot's intelligence has been used so as the knowledge to be built in the most efficient way. As presented in the attached implementation, the followed strategy as well as the deductions appear to be robust.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd ed., 2009.
[2] S. Russell, P. Norvig, and A. Intelligence, *A modern approach*, vol. 25. Citeseer, 1995.