Mazeworld

Sujay Busam

January 26, 2014

1 Introduction

The following is a maze drawn in "ASCII art" with the periods representing open floor tiles (places that can be traveled over) and hash signs representing walls or obstacles where a robot cannot go:[1]

```
.......
.##....
..##...
.......
..##...
#.###..
....##.
```

The maze is represented by coordinates on an x/y plane with origin at the bottom left corner of the maze. In the simple implementation of this problem, there is one robot that can

in four directions: North, South, East, and West. In this problem, the robot must go through the maze, starting at a given start node and reaching a goal node. The robot traverses the maze according to the specific search algorithm used and must take care not to run into any obstacles.

In the provided code, `SimpleMazeDriver` shows this simple implementation using BFS and DFS. Not surprisingly, the BFS algorithm allows one of the robots to find the shortest path to the goal, while DFS yields a longer path with more nodes explored.

The code is factored in an MVC (Model, View, Controller) framework in which different models (representations of mazes and robots) are solved by the controller (search algorithms) and displayed through the graphical view (animation of robots movement). This allows us to explore more complicated maze scenarios.

2 A-Star Search

A star (A*) search is an example of an informed search strategy - a search that utilizes extra knowledge of the problem to find a more efficient solution that what uninformed search yields.

In A* search, every successor node is evaluated by by combining the cost to reach the node, $g(n)$, with a heuristic value, $h(n)$, which estimates the cost to get to the goal node from the current node. Note that unlike $g(n)$, the heuristic function $h(n)$ is dependent only on the state at the given node. In other words, given a state, the heuristic value should be able to be computed. This is an important detail of A* search since the algorithm needs to determine, at every state, whether the path is making progress towards the goal.

[1]From the assignment: https://dartmouth.instructure.com/courses/744/assignments/6503

My A* search algorithm is implemented in the `InformedSearchProblem` class. The class is a specialized version of its parent - it inherits from `SearchProblem` and has access to variables such as `startNode`.

Here is my implementation of A* search:

```java
  public List<SearchNode> astarSearch() {

    resetStats();

    // Initialize fringe / frontier
    PriorityQueue<SearchNode> fringe = new PriorityQueue<SearchNode>();

    // Initialize explored hashmap. Maps node to its priority
    HashMap<SearchNode, Double> visited = new HashMap<SearchNode, Double>();

    // Add start node to fringe
    startNode.setParent(null);
    fringe.add(startNode);

    SearchNode currentNode;

    // Implementation of algorithm
    while (!fringe.isEmpty()) {
      incrementNodeCount();
      updateMemory(fringe.size() + visited.size());


      // Pop node from queue and continue, but only if its priority is lower than what
      // this node's priority is if it was visited before or if it hasn't been
      // visited at all before
      currentNode = fringe.peek();

      if ((visited.containsKey(currentNode) && currentNode.priority() <
          visited.get(currentNode)) || !visited.containsKey(currentNode)) {

        currentNode = fringe.poll();

        // Test if goal found
        if (currentNode.goalTest()) {
          // Backchain
          return this.astarBackchain(currentNode);
        }

        // Add node to explored map
        visited.put(currentNode, currentNode.priority());


        // For each successor
        for (SearchNode successor: currentNode.getSuccessors()) {

          // If successor not explored, add to fringe
          if (!visited.containsKey(successor)) {
            successor.setParent(currentNode); // Set parent
            fringe.add(successor);
          }

          // If node was visited but now has less expensive path cost, add node
          else if (visited.containsKey(successor) && successor.priority() <
              visited.get(successor)) {

            successor.setParent(currentNode); // Set parent
            fringe.add(successor);
          }

          // Otherwise, disregard node
        }
      }

      // Otherwise, disregard current node
      else {
        fringe.remove();
      }
    }

    // Path to goal not found
    return null;
  }
```

The implementation is similar to that of BFS that we saw in the Missionaries and Cannibals problem. However, with A* there are some key differences. A* search evaluates nodes' priorities, which are simply the sum of the cost to reach that state and the heuristic value. Thus, A* does not pull nodes from its fringe in a queue-like (FIFO) fashion; a regular queue is not even used in the implementation. Instead, a priority queue is used to determine which node to pull off the fringe next. The priority queue is arranged as a minheap - the node with the lowest priority (lowest cost + heuristic value) is the head and gets pulled first when dequeueing from the fringe.

It is also important to note that to determine a successor to add to the fringe, we cannot simply test whether the visited hashmap does not contain that successor. We must also test whether the visited hashmap contains that successor but it was explored previously at a greater cost. In that case, the node can still be explored because a path incurring less cost has been found. This is very similar to the test required for a Memoizing Depth-First search algorithm.

However, as mentioned in the assignment, this presents us with a slight problem. In the test case mentioned above, the current node's state with a higher path-cost exists in the fringe and must be replaced. This is a not an efficient process when using a priority queue implemented with a heap. The heap's structure (a cleverly laid out array) is such that removing and adding a node into the middle incurs a large computational cost. The heap must rearrange itself using its `heapify` method. Note that this could be avoided by using a sorted arraylist to implement the priority queue, but this only serves to create a greater problem. Adding a node with a certain priority to an arraylist that is sorted by priority is very costly in terms of runtime. Elements in the array have to be shifted to make room for in a certain index for the node.

To get around this, we add the node to the fringe even if it already contains a node with that state at a higher path-cost. We then discard the node with the higher cost if it is found when popping from the fringe. This is an effective way to deal with the impliciations of inserting and removing from the middle of a priority queue.

As mentioned, the priority of a node is comprised of the cost of the state in addition to the heuristic value. Therefore, the specific node class must implement a heuristics function. In order for A* search to be optimal, the heuristic function must be an admissible, or optimistic, heuristic. That is, the function can never overestimate the cost to reach the goal. Examples of this would be straight-line distance or Manhattan distance. The implementation of the heuristic function for the simple maze problem is as follows:

```
    @Override
    public double heuristic() {
      // manhattan distance metric for simple maze with one agent:
      double dx = xGoal - state[0];
      double dy = yGoal - state[1];
      return Math.abs(dx) + Math.abs(dy);
    }
```

This implementation uses the Manhattan distance metric, which represents how a taxi cab would move through the streets of Manhattan. It is an optimistic heuristic for this problem because it will always underestimate the distance to the goal. A robot cannot achieve a shorter path to the goal than the Manhattan distance.

My A* search, similarly to Breadth-First Search, implements a backchain method to return the optimal path, starting with the start node and ending with the goal node. Here is my code for the `astarBackchain` method:

```
  public List<SearchNode> astarBackchain(SearchNode node) {
    SearchNode currentNode = node;
    List<SearchNode> path = new LinkedList<SearchNode>();

    // While the current node isn't null, add it to the list and get its parent
    while (currentNode != null) {
      ((LinkedList<SearchNode>) path).addFirst(currentNode);
      currentNode = currentNode.getParent();
    }

    return path;
  }
```

The backchain method raises a question in regards to implementing `parent` nodes. In BFS and other search algorithms, the parent node could be stored in the visited hashmap as the value in the key, value pairing. However, with the A* algorithm, the hashmap serves to map nodes with their respective priorities. Do we then create another hashmap for a node - parent mapping?

I decided for a more practical solution with less overhead. I incorporated `parent` as a private instance variable of the `SearchNode` class and wrote appropriate `get` and `set` methods. I also did this for the `MultiRobotNode` used in the next section.

3 Multi-Robot Coordination

In this problem, k robots live in an nxn rectangular maze. The objective is to have them start at some given state and reach a given end state. Each robot can perform the same actions as in the simple problem, in addition to having the option of not moving. The same rules apply in regards to walls and obstacles, but now there is the added constraint that robots cannot collide with each other; only one robot may occupy a square. The map, which is given at the start of the problem, is a static map and will not change its characteristics during the problem.

With this problem, it was vital to think about the state space and how it is to be represented. The best way to design the problem is to have each node, a `MultiRobotNode` in my case, entirely represent the current state. By doing this, the search algorithms we have already written, namely A*, can perform a search and return a path of nodes. Thus, a path of states, starting with the start state and terminating with the end state, will be found.

But how exactly is the state represented? All that is necessary to reconstruct the state are the x,y coordiates of each robot and a variable representing which robot's turn it is to move. So if the maze contains three robots, seven numbers (three x and y pairs and turn) representing the state are all that are required to know everything about the configuration of the robots and what actions are available from that state.

To calculate an upper bound for the number of states we must consider the size of the maze and number of robots. In an nxn maze with k robots, the first robot can be in one of n^2 spots, the second robot can be in one of $n^2 - 1$ spots, the third in $n^2 - 2$ spots, and so on. Therefore, we are essentially choosing k spots out of a possible n^2 for these robots. This is equivalent to the binomial expression n^2 choose k or $\binom{n^2}{k}$. The expanded version of this expression is $\frac{n!}{k!(n-k)!}$

However, we haven't taken into account which robot's turn it is, which is a vital part of the state. For each of these states, there are k possibilities for whose turn it is. Thus, the final upper bound value is $\binom{n^2}{k}^k$ or $\left(\frac{n!}{k!(n-k)!}\right)^k$

We can also estimate how many of these possible states represent collisions given n is much larger than k and the number of wall squares is w. In this case the number of possible legal states is not n^2 choose k since there are walls present. It is now $n^2 - w$ choose k. Therefore the number of states representing collisions is $\binom{n^2}{k}^k$ - $\binom{n^2 - w}{k}^k$

Let us now consider a multirobot maze problem in which there are not many walls, n is large, and there are several robots. With a maze of this nature, a straightforwards breadth-first search would not be computationally feasible for all start and goal pairs. In order to complete one turn, the algorithm has to process an astounding number of nodes. Depending on the position of the robots, each could potentially have 5 valid actions to choose from. This means, in the worst cast, the first node has 5 successors. Each of those 5 nodes has 5 successors of themselves, and so on and so on. BFS visits all of these nodes because it does not use a heuristic, and there are simply too many for this kind of search to be feasible.

A useful, monotonic heuristic would be the straight line distance computed using the distance formula. With staight line distance, it is easy to see that it is a monotonic heuristic because it necessarily obeys triangle inequality. To not do so would be to disobey basic geometrical laws. The straight line distance from node N to the goal is necessarily shorter than the cost to a successor node from N + the straight line distance from that successor to the goal node.

My code for this problem is in the `MultiRobotProblem` class with an inner `MultiRobotNode` class. Just as with the simpler problem, the state is contained in the node class, but this time it is a 2D array that holds the x,y coordinates of every robot.

The `getSuccessors` method is at the core of the node classes for these maze problems. The method has to correctly return all possible, legal successor nodes (states) from "this" node. Here is my code for the method:

```
// Get the valid successors of the current node
public ArrayList<SearchNode> getSuccessors() {

  ArrayList<SearchNode> successorsList = new ArrayList<SearchNode>();


  // Run through all possible actions for bot whose turn it is to move
  for (int[] action: actions) {
    int xNew = currentState[turn][0] + action[0];
    int yNew = currentState[turn][1] + action[1];

    //System.out.println("testing successor " + xNew + " " + yNew);

    if(maze.isLegal(xNew, yNew) && !isOccupied(xNew, yNew)) {
      //System.out.println("legal successor found " + " " + xNew + " " + yNew);

      // Make a copy of the current state
      int[][] newState = copyState(this.currentState);

      // Modify new state appropriately
      newState[turn][0] = xNew;
      newState[turn][1] = yNew;

      // Cost incurred regardless of movement
      double newCost = getCost() + 1.0;

      // Create successor node with this new state
      MultiRobotNode successor = new MultiRobotNode(newState,
          newCost, (turn + 1) % numBots);

      // Add successor to list
      successorsList.add(successor);
    }
  }
  return successorsList;
}
```

The method runs through each robot and applies all possible actions for each. The action results in a new potential location for the robot. However, this location must be checked for legality. This is done by `isLegal` of the `Maze` class. We must also test to see if this potential location would result in a collision with another robot. This is done with my helper method `isOccupied`:

```
// Helper method to see if x,y space is already occupied by a bot
private boolean isOccupied(int x, int y) {

  // Run through x,y of all bots (excluding current one) to check for collisions
```

```
    for (int i = (turn + 1) % numBots; i != (turn % numBots); i = ((i + 1) % numBots)) {

      // Check if this bot's position matches with the checked bot's position
      int currentBotX = currentState[i][0];
      int currentBotY = currentState[i][1];

      if (x == currentBotX && y == currentBotY) {
        return true;
      }
    }

    // No collisions with other bots if for loop exited
    return false;
  }
```

If the potential location passes these legality tests, a successor can be created with this new state. It should be noted that the current state had to be copied and modified in order for the method to work correctly. It is not enough to simply assign the state (the 2D array) to a new variable. This will cause clobbering - modifications to this supposedly new state will in fact be modifying the old 2D array referencing the old state. Here is my helper method to copy the state:

```
  // Helper method to copy predecessor node's state array
  private int[][] copyState(int[][] oldState) {
    // Copy to be returned
    int[][] newState = new int[numBots][2];

    for (int i = 0; i < oldState.length; i++) {
      for (int j = 0; j < 2; j++) {
        newState[i][j] = oldState[i][j];
      }
    }

    return newState;
  }
```

Here is my code for the heuristic function implementation:

```
  public double heuristic() {

    // Straight line distance heuristic
    double heuristic = 0;

    for (int i = 0; i < numBots; i++) {
      heuristic += (Math.sqrt(Math.pow((goalState[i][0] - currentState[i][0]),
          2.0) + Math.pow((goalState[i][1] - currentState[i][1]), 2.0)));
    }
```

In order to test, I had to modify the animation / graphics code to deal with multiple robots and moves in one turn. I did tests with various size mazes to ensure A* search was working properly on multiple robots. Here are some of them:

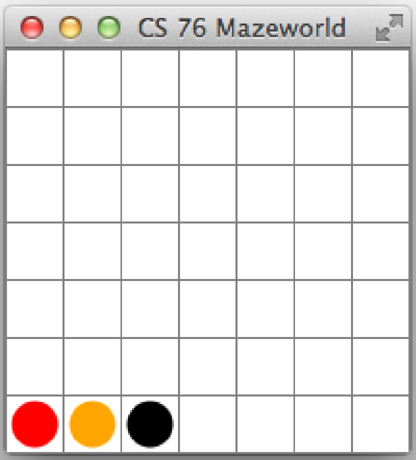

Figure 1: Simple Maze - no obstacles. Start position

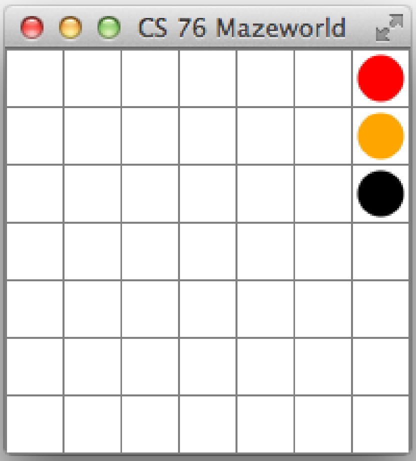

Figure 2: End position.

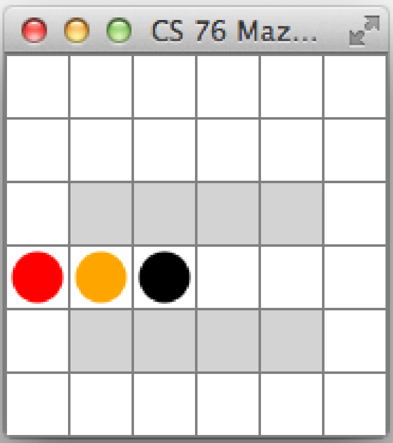

Figure 3: Maze with corridor of obstacles. Start position

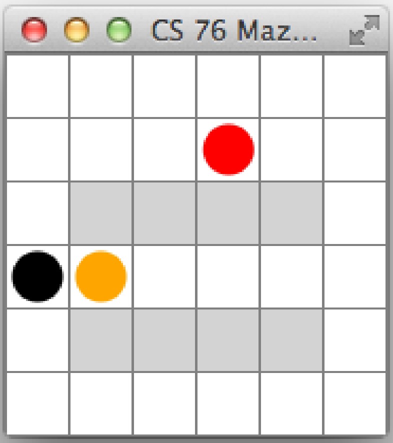

Figure 4: Rearrangement of robots.

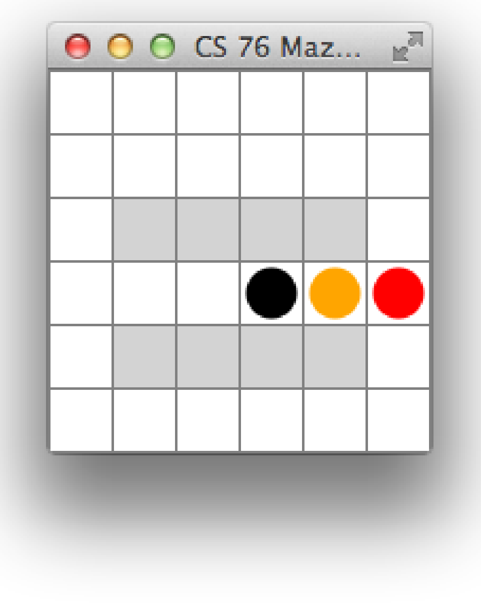

Figure 5: End positon

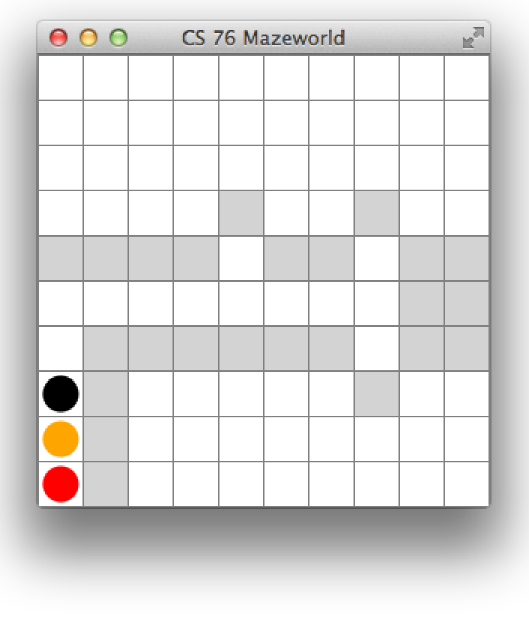

Figure 6: Another corridor of obstacles. Start position

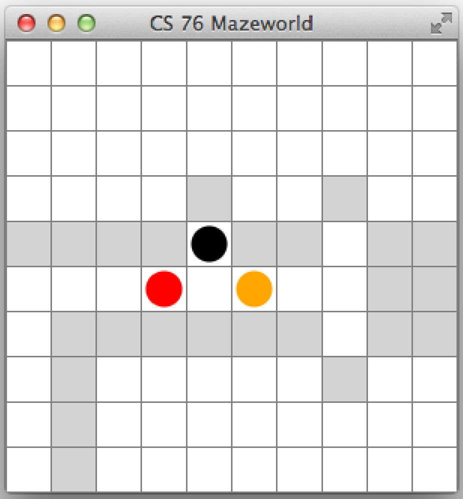

Figure 7: Rearrangement of robots.

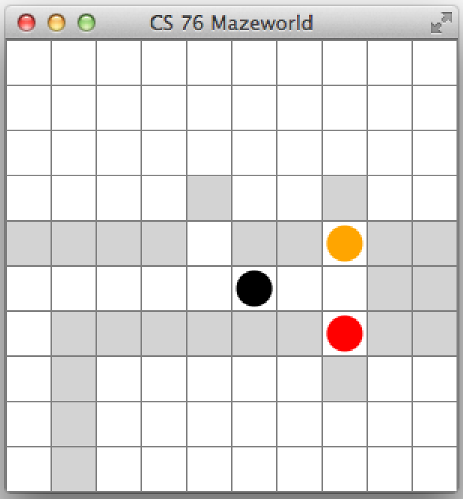

Figure 8: Extra rearrangement

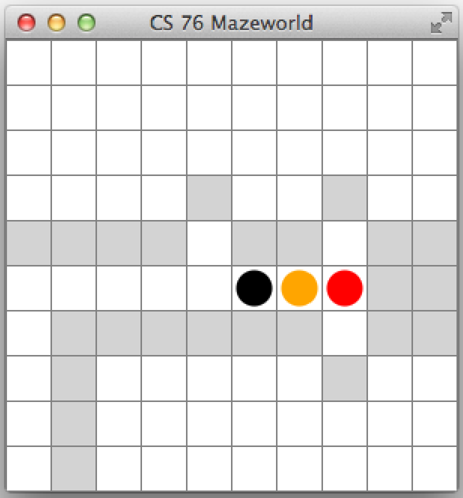

Figure 9: End positon