Maze World

Siyang Li

February 9, 2015

1 Introduction

In this assignment, "robot in a maze"-type simulations are designed. Robots, faced with obstacles in a maze, use different search algorithms to find efficient paths to a set goal. The purpose of this assignment is to implement informed search (more specifically A* search) and to practice setting up algorithms to solve diverse problem conditions by identifying appropriate state space and collision space.

2 A* Search

A* Search is implemented in the file `InformedSearchProblem.java`. Here's my code for `aStarSearch`:

```
public class InformedSearchProblem extends SearchProblem {

public List<SearchNode> aStarSearch() {
  resetStats();

  SearchNode currentNode = startNode;
  ArrayList<SearchNode> successorsArray;
  PriorityQueue<SearchNode> fringe = new PriorityQueue<SearchNode>();
  HashMap<SearchNode, SearchNode> visitedMap = new HashMap<SearchNode, SearchNode>();

  visitedMap.put(startNode, startNode);
  incrementNodeCount();

  while (!currentNode.goalTest()) {
    successorsArray = currentNode.getSuccessors();

    for (SearchNode successor: successorsArray) {
      if (!visitedMap.containsKey(successor)) {
        successor.addParent(currentNode);
        fringe.add(successor);
        visitedMap.put(successor, successor);
        updateMemory(visitedMap.size());
      }

      if ((visitedMap.containsKey(successor)) &&
      ((successor.compareTo(visitedMap.get(successor))) == -1)) {
        visitedMap.get(successor).markRemoved();
        fringe.add(successor);
        visitedMap.put(successor, successor);
```

```
            }
          }

        currentNode = fringe.poll();

        while (currentNode.isRemoved()) {
          currentNode = fringe.poll();
        }

        incrementNodeCount();
      }

        return aStarBackchain(currentNode);
};
      }
```

A* search is implemented in a way very similar to regular BFS. Instead of a queue, the fringe is set up as a PriorityQueue (which is implemented in java as a heap). Priority of a state is based on the sum of the cost to get to the node and a heuristic that estimates how close to node is to the goal. This is heuristic-based priority is the essence of A* search. Both cost function and heuristic function are included in the `SearchNode` class. In the A* search algorithm, the most difficult part was the replacement of nodes in the fringe if the node was accessed through a shorter path. The difficulty arises from the removal of a node in a heap. Because of the way a heap is designed, removing a node requires destroying and rebuilding the heap entirely - this is very inefficient. An alternative is to simply mark nodes as "removed" if they are replaced. This is what we did; by adding a isRemoved field into the `SearchNode` class, we can mark nodes themselves as removed. If a "removed" node is taken out of the fringe, it is simply discarded. In order to access previous nodes in the fringe easily however, visitedMap must now store itself as the object. This allows easy access to the fringe (heap) via direct mapping. The visitedMap contains as keys all nodes that were visited. As objects, it contains a reference to the lowest priority instance of the object (thus the only one not marked "removed"). This allows easy comparison of the priority of the node from the fringe and the current node if the current node was already visited. In addition, because visitedMap no longer contains reference to parents of each node, that information must also be stored in a separate field contained in the SearchNode itself. Aside from these alterations and adjustments, A* search's algorithm remains fairly similar to BFS.

In addition, we did use a different backchaining method `aStarBackchain` to supplement our main code:

```
  protected List<SearchNode> aStarBackchain(SearchNode currentNode) {

    LinkedList<SearchNode> backchainList = new LinkedList<SearchNode>();

    while (currentNode.returnParent() != null) {
      backchainList.addFirst(currentNode);
      currentNode = currentNode.returnParent();
    }

    return backchainList;
  }
```

This backchaining method finds the parent by calling the `returnParent()` method which returns the parent field in the node (as opposed to using the visitedMap like previous backchaining method used in BFS).

3 A* Search Discussion and Results

To test the A* Search algorithm, we made the `SimpleMazeProblemDriver.java` call upon A* Search in addition to previously implemented BFS and DFS searches. The resulting path of the BFS, DFS, A*Search are shown in Figure 1. It is noted that both BFS and A* Search find the shortest path (even if they diverged because of search order). Where A* is different from BFS is in its run-time and memory usage. The statistics are shown below:

```
BFS:
  Nodes explored during search:  21
  Maximum space usage during search 21
A*:
  Nodes explored during search:  19
  Maximum space usage during search 24
```

Because A* selectively expands upon nodes that are more likely to be closer to goal (through our heuristic and cost functions), it saves upon the number of nodes search. Nodes that are not likely to lead to goal (for example, those that lead away from goal) are not expanded. This however does increase the maximum space used during search since many nodes will be stored and not popped until later on. However, for a larger maze problem, A* search also saves on memory since it does not needlessly expand upon unnecessary nodes and add excessive successors to the queue.

4 Multi-Robot Coordination

In the next section of the assignment, a more complicated version of the original robot maze problem is presented. Instead of one robot, there are three robots in the maze. Each may not collide with one another (in addition to the walls and obstacles in the maze). Each may move one at a time and can opt to stay where they are. The goal state is met when all three robots are in their exact respective goal positions. All of the code for the multi-robot problem can be found in the `MultiRobotMaze.java` file. However, important parts of the code will be analyzed here.

First, in order to break down the problem effectively, it must be decided what is needed to represent the state in the problem. This is found in the declarator for the `MultiRobotMazeNode`:

```
public MultiRobotMazeNode(int a_x, int a_y, int b_x, int b_y, int c_x,
int c_y, int currentRobot, double c) {
      state = new int[7];
      this.state[0] = a_x;
      this.state[1] = a_y;
      this.state[2] = b_x;
      this.state[3] = b_y;
      this.state[4] = c_x;
      this.state[5] = c_y;
      this.state[6] = currentRobot;

      cost = c;
    }
```

As is seen, the state can be represented in 7 variables. These variables include the x, y coordinates of each of the three robots. In addition, a 7th variable currentRobot is included to represent which robot is current robot to move. With these 7 variables in the state, the entire situation of the problem can be represented.

What is left is to change the main functions of the basic SearchProblem so that it can deal with the new

state and conditions of the problem.

First is the `getSuccessors` method:

```
public ArrayList<SearchNode> getSuccessors() {

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

    for (int[] action: actions) {
      int xNew = state[(state[6] * 2)] + action[0];
      int yNew = state[((state[6] * 2) + 1)] + action[1];

      if (maze.isLegal(xNew, yNew)) {
        SearchNode succ = null;

        if (state[6] == 0) {
          succ = new MultiRobotMazeNode(xNew, yNew, state[2], state[3], state[4],
          state[5], 1, getCost() + 1.0);
        }

        if (state[6] == 1) {
          succ = new MultiRobotMazeNode(state[0], state[1], xNew, yNew, state[4],
          state[5], 2, getCost() + 1.0);
        }

        if (state[6] == 2) {
          succ = new MultiRobotMazeNode(state[0], state[1], state[2], state[3],
          xNew, yNew, 0, getCost() + 1.0);
        }

        if (! succ.robotOverlap()) {
          successors.add(succ);
        }
      }

    }
    return successors;

  }
```

The premise of the `getSuccessors` method is very similar to previous implementations. The method needs to (1) find all potential moves from a certain state and (2) test if they are legal (if they are, add them to a successors array to be returned). However, in this version of the method, we need to take into consideration the complication multiple robots may present. First of all, we must look at state[6], the currentRobot field to decide which robot is to move; this changes successor states as moving different robots creates different end states. The currentRobot field is a number 0 to 2 which represents the 1st to 3rd robot respectively. Using math, we can use the numerical value of state[6] to index directly into state to find which x, y fields to alter (state[6] * 2 = x value wanted; state[6] * 2 + 1 = y value wanted). Once we know which robot to move, we move the robot either north, south, west, east, or not at all. This is done by adding either 0 or 1 to the x or y of the current robot (north is 1 to x, 0 to y for example). This gives us a "potential successor". We must then test if that successor is legal by using the `Maze.java` class's `isLegal()` function. This makes sure that the current robot's move does not hit a wall or obstacle. However, in addition, we must also test that the current robot does not move onto another robot in the maze. This can be done in a helper method,

`robotOverlap()`:

```
public boolean robotOverlap() {
  return (((state[0] == state[2]) && (state[1] == state[3])) ||
      ((state[0] == state[4]) && (state[1] == state[5])) ||
      ((state[2] == state[4]) && (state[3] == state[5])));
}
```

All this method does is compare the x,y coordinates of each of the three robots in the potential state to see if any are in the exact same location. If any robots do overlap, the method returns true and the potential successor is not added to the list of legal successors. In the end, our `getSuccessors` method takes into consideration the more complex state space and conditions to return a list of all possible successors from a given state.

In addition to the `getSuccessors` method, another important method changed was the `goalTest()` method:

```
public boolean goalTest() {
  return state[0] == a_xGoal && state[1] == a_yGoal &&
      state[2] == b_xGoal && state[3] == b_yGoal &&
      state[4] == c_xGoal && state[5] == c_yGoal;
}
```

This method is pretty straightforward though. Instead of checking if the current x, y of a single robot equals the goalX, goalY, this method checks to make sure all robots are in their correct goal positions. If any of them aren't, the method returns false - the problem is not yet solved.

Lastly, we must discuss the heuristic function for this type of problem. The heuristic function is important to determine priority used in A* search in addition to the `getCost()` function. The cost function is still just the number of moves to get to the current node. However, the heuristic function is more complex. Here is the `heuristic()` method:

```
public double heuristic() {
    double a_dx = a_xGoal - state[0];
    double a_dy = a_yGoal - state[1];

    double b_dx = b_xGoal - state[2];
    double b_dy = b_yGoal - state[3];

    double c_dx = c_xGoal - state[4];
    double c_dy = c_yGoal - state[5];

    return Math.abs(a_dx) + Math.abs(a_dy) + Math.abs(b_dx) + Math.abs(b_dy)
    + Math.abs(c_dx) + Math.abs(c_dy);
}
```

It is not complicated. Essentially, the heuristic is simply the sum of the Manhattan distance for all of the robots. This makes more sense than taking the average since only one robot can move any how each turn (and cost++). This heuristic is good because it is both admissible; in actuality, there will be obstacles making a Manhattan path not possible. Therefore, it can only underestimate the distance to the goal. It should also be fairly consistent for each location in the maze.

5 Multi-Robot Discussion and Results

To understand our multi-robot problem in more detail, we can analyze the problem here. First, we discuss the upper bound of the state space in such a problem. If n represents the width/height of the square maze

and k represents the number of robots, the upper bound of the size of the state space is (n**2)**k. Each robot has the option of being any where in the n**2 grid. To get total combination of positions, number of positions for each robot are multiplied. However, this is the upper bound since collisions with other robots and walls are not considered. If there are w number of walls, (w/n**2)**k of the states represent collision states. This is because w/n**2 represents the chance of a wall collision. If any of the robots collide with the wall, the whole state is a collision state. Therefore, the chances of each robot to collide with wall are multiplied to get the percent of all states that are collision states. If there w is small, n is large, and k is small, breadth-first search can not be expected to be computationally feasible for all start and goal pairs. This is because the size of the fringe grows too exponentially especially with three robots.

Our results are displayed in Figure 2. This is a test trial on the original maze with three robots. As you can see, the path is very efficient and short (as created by A*). In addition, the robots move with respect to each other. This can be clearly seen towards the end of the path. The robot cluster reaches the end corridor but the second and first robots must switch place. The first robot therefore goes into a gap on side while the second robot moves ahead of it. In addition to this basic test, multiple other trials (at least 5 from the maze dump on Piazza) on different mazes were done. These mazes were found from as public domain material from fellow student Ryan Amos. The search and robot problem was solved in all other cases efficiently. In each case, the robots were able to move, pause and essentially coordinate with each other to avoid collisions (while at the same time getting closer to the goal). When meet with corridors, robots moved into crevices in order to switch places (like seen in end of our displayed example). This is due to a strong admissible, consistent heuristic.

The 8 puzzle described in the book is a special case of our problem since not all robots on the maze can even get the option of moving in a single turn. In fact, only 2-4 robots even have the chance to move every turn. The heuristic function chosen for the multi-robot problem would not be good in the case of the 8-puzzle. This is because distance in that problem is less important than configuration. It may be harder to get a piece to its goal if doing so destroys the configuration of other pieces (even if it was really close to its goal). The state space of the 8-puzzle would consist of two disjoint sets (one that contains the positions of each of the robots and another containing the robots bordering the empty slot that can move). This can be modified from our code by changing the currentRobot field to a HashSet containing the robots which are able to move.

6 Blind Robot w/ Pacman Physics

In this section of the assignment, another version of the original robot maze problem is presented. There is again one robot. However, the robot does not know its original location on the maze. In addition, it has no sensors to tell that it hit a wall or obstacle (although it will still be blocked in such moves). It does know the maze layout. Using these information, an algorithm must be designed to take the robot to a goal coordinate. Again, although this problem is very different from previous scenarios, it can still be broken down and solved using strategies of AI used in previous problems. All code for the blind robot problem can be found in the `BlindRobotMaze.java` file. Here we will analyze important sections of the code.

First, we must again look at how to represent the state in the problem. This is found in the declarator for the `BlindRobotMazeNode`:

```
public BlindRobotMazeNode(HashSet<int[]> possibleStates, double c) {
    state = possibleStates;
    cost = c;
}
```

At first, this declarator might seem very simple and somewhat confusing. Instead of any defined state fields, we have a HashSet created to contain a tuple of integers representing a possible x,y location for our

robot. It becomes clear why we need an expandable HashSet with undefined size when we look out how this problem can be solved. Because we do not have any idea of initial location, our state initially is any and all possible legal states we could be in a given map. As we make actions, our successor's state will be any and all possible legal states made with that action from any of our previous legal states. Because moves will be illegal at parts of the map, our state, the HashSet of possible x,y location, gets smaller. When the size of the HashSet becomes 1, we know our location and the problem becomes like the `SimpleMazeProblem.java`. However, the fact that our state has such variable size is the reason why we need to store it in a data structure who's size can easily change. A HashSet is also good in its ability to access/remove data in a quick fashion.

To see in detail how the problem is initialized without a start location, we look at the declarator for the `BlindRobotMaze.java` class:

```
public BlindRobotMaze(Maze m, int goalX, int goalY) {
    HashSet<int[]> startState = new HashSet<int[]>();
    goalPosition[0] = goalX;
    goalPosition[1] = goalY;

    for (int i = 0; i < maze.height; i++) {
      for (int j = 0; j < maze.width; j++) {
        if (maze.isLegal(j, i)) {
          int[] possiblePosition = {j, i};
          startState.add(possiblePosition);
        }
      }
    }

    startNode = new BlindRobotMazeNode(startState, 0);

    maze = m;
  }
```

The declarator basically takes in (1) the maze and (2) our goal location. These are, after all, the only pieces of information the robot has to work with. First, the goalPosition (an array of size 2 with the goal coordinates) is created using the inputted goalX and goalY values. Then, the state of the startNode is created by taking the width and height of the maze and using a nested for-loop to go through every possible coordinate location in the maze. Using the `isLegal` method of the `Maze.java` class, each location is tested for legality in the boundary of the current maze. All legal possible states the robot could be in the start are added to the startState (which is then added to create the startNode). This sets up the search problem as a startNode and goal conditions are created from which to run `getSuccessors` and ultimately search algorithms on.

`getSuccessors`, the next important method, is here:

```
    public ArrayList<SearchNode> getSuccessors() {

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

      for (int[] action: actions) {
        Iterator<int[]> stateIterator = state.iterator();
        HashSet<int[]> newState = new HashSet<int[]>();

        while (stateIterator.hasNext()) {
```

```
    int[] possibleState = stateIterator.next();

    int newX = possibleState[0] + action[0];
    int newY = possibleState[1] + action[1];

    int[] newPosition = {newX, newY};

    if (maze.isLegal(newX, newY)) {
      Iterator<int[]> newStateIterator = newState.iterator();
      int[] newPotentialPosition;
      boolean inNewState = false;

      while (newStateIterator.hasNext()) {
        newPotentialPosition = newStateIterator.next();
        if ((newPotentialPosition[0] == newPosition[0])
        && (newPotentialPosition[1] == newPosition[1])) {
          inNewState = true;
        }
      }

      if (! inNewState) {
        newState.add(newPosition);
      }

    }

    if (! maze.isLegal(newX, newY)) {
      Iterator<int[]> newStateIterator = newState.iterator();
      int[] newPotentialPosition;
      boolean inNewState = false;

      while (newStateIterator.hasNext()) {
        newPotentialPosition = newStateIterator.next();
        if ((newPotentialPosition[0] == possibleState[0])
        && (newPotentialPosition[1] == possibleState[1])) {
          inNewState = true;
        }
      }

      if (! inNewState) {
        newState.add(possibleState);
      }
    }
  }


  if (! newState.isEmpty()) {
    SearchNode succ = new BlindRobotMazeNode(newState, getCost() + 1.0);
    successors.add(succ);
  }
```

```
    }

    return successors;

  }
```

This version of `getSuccessors` here works a little different from previous versions. Instead of checking each move for legality, we assume each move is legal. Instead, we determine all legal end states if we made that move (which become the state for the new node). Only if NO legal end states are found for a move do we not add it to the successors list. Essentially, we are working backwards from previous scenarios. First, the method uses a for-loop to go through each of the 4 possible actions for the robot in any situation (north, south, east, and west). The method then works to iterate through the HashSet of possible coordinates for the robot (remember, at the beginning, this is essentially every legal location) and apply the move for each location. Using the `isLegal` method of the `Maze.java` class, each new location is checked for legality before being added into an Arraylist representing the new belief state (possible locations) for the successor node. If the move is not legal, the current possible position is re-added to the new state; this is because the robot could still be in the original location if it didn't move. In addition, to get rid of duplicate positions in the new state, iterators go through the newState to make sure the position wanted to be added is not already in the new state. Because some moves are bound to be illegal or not, we are bound to get non-increasing belief state size. This can continue until the belief state size is 1, in which case we know our location and the search problem proceeds like simple maze problem.

In addition to the `getSuccessors` method, another important method changed was the `goalTest()` method:

```
  public boolean goalTest() {
    return ((state.contains(goalPosition)) && (state.size() == 1));
  }
```

This method checks first that the state has size of 1 (which means the robot narrowed down its location to one position; the robot is sure where it is). Then, it checks if the state has/is the goal position (defined earlier). Only if both conditions are true do we have the goal.

Lastly, we must discuss the heuristic function again for this type of problem. This heuristic function is more complex. Here is the `heuristic()` method:

```
public double heuristic() {
      if (state.size() > 1) {
        Iterator<int[]> stateIterator = state.iterator();
        int minX = 999999, minY = 999999;
        int maxX = 0, maxY = 0;
        int[] possibleState;

        while (stateIterator.hasNext()) {
          possibleState = stateIterator.next();

          if (possibleState[0] < minX) {
            minX = possibleState[0];
          }
          if (possibleState[0] > maxX) {
            maxX = possibleState[0];
          }
          if (possibleState[1] < minY) {
            minY = possibleState[1];
          }
```

```
        if (possibleState[1] > maxY) {
          maxY = possibleState[1];
        }
      }

      return ((double) (maxX - minX) + (maxY + minY));

      else {
      Iterator<int[]> stateIterator = state.iterator();
      int[] position;

      while (stateIterator.hasNext()) {
        position = stateIterator.next();

        double dx = goalPosition[0] - position[0];
        double dy = goalPosition[1] - position[1];
        return ((double) (Math.abs(dx) + Math.abs(dy)));
      }
    }

    return 999999.9;
  }

      }
```

The method goes through all possible locations in the belief state using an iterator. It keeps track of the maximum and minimum x and y values (4 values total). Using these values, it determines the range of the belief state by finding the range of the x values (maxX - minX) and y values (maxY - minX) and adding them. The idea is that the lower the range, the closer we are to the solving the problem (since the goal is size 1 and having a known location makes a search much more direct). Once the size of the belief state is 1 however, the heuristic becomes simply Manhattan distance (like previous implementations). I've found this heuristic to be good though it is not perfect. First of all, though the function is admissible, it is likely not consistent (simply because the function changes entirely at certain points). However, for the purpose of our search, it should work fine.

7 Blind Robot Discussion and Results

In the end, our tests on the blind robot problem was successful at finding a path to the goal even when the initially starting location was unknown. The basic test on the a simpler 4x4 maze is shown in Figure 3. As you can see, the algorithm is able to find a path which decreases the size of the belief state so that the position of the robot can be made certain. Then, the robot can go to the goal like a simple maze problem search. The path of the robot in the displayed example is as follows: south, east, south, east, south, south, east, east, north, north, north. Every move, the belief state shrunk. When the belief state size was 1 (at step 10), we were certain of location. Because of the way our heuristic works, the priority is first based on the range of the belief state. Only when the size of the belief state was 1 do we care about distance. This forces the algorithm to prioritizes figuring out its location before trying to go to the goal. I think this heuristic is effective at giving relatively short paths (though other algorithms that focus on distance as well as range might be shorter).

Other tests were done with larger examples (for example, simple.maz and other files found in the maze dump provided by Ryan Amos). However, run time grows exponentially as the size of the maze increases. Even on our 4x4 maze, A* search had to go through 5534 nodes (and used 22133 memory size). Therefore,

for larger mazes (6x6 or greater), the tests took an significant long amount of time. For even larger mazes, this algorithm is not feasible. We would need to design an algorithm to be polynomial or linear time.

8 Polynomial-time Blind Robot Planning

A sensorless plan can also be found for the blind robot problem if the maze is finite size and the goal and start are in the same connected region of the maze. This can be proven simply by looking at the algorithm. The algorithm continues to move in all directions, hitting walls/obstacles to decrease the size of the belief state. As long as there are walls/obstacles, the belief state can be decreased to size 1. If the location of the belief state at that point is in the same connected region of the maze as the goal (and it will be since that is where it started), the goal can be reached through simple A* Search which will expand the search outward until the goal is reached. When the size of the belief system becomes 1, the algorithm turns into a simple maze problem anyways.

How would the blind robot problem be modified so that it runs in linear or polynomial time with respect to the size of the maze? This can be implemented simply by making sure that the size of the belief state decreases each move. This can be done by altering the `getSuccessors` method so that it first checks that the newState.size() ¡ state.size(). By doing this, linear/polynomial time is ensured since the robot will be sure of its location in at most n**2 number of moves. A regular A* search can be done afterwards to move it towards the goal.