Mazeworld–A* Search & Path-Planning Problems

Xinqi Li

January 23, 2014

Abstract

The report examines the implementations of the maze-robot planning problem using the A* search. The first part shows the implementation of A* search by animating the single robot path planning problem. The second part is about the multiple robots cooperative path finding problem. The third part discusses the blind robot problem which means the robot cannot sense its location but can find its way to goal by applying a proper heuristic function to A* search. In the last part, a paper about the previous work on applying A* search on the multi-robot path-finding problem will be discussed.

1 Part One A-star search: Introduction

In my perspective, A-star search is a combination of Uniform Cost Search and Greedy Best First Search. It first searches the routes that appear to be most likely to lead towards the goal. To do this it adds the cost from start to the current state and the estimated cost from current state to the goal. As the instruction suggests, I implement the Uniform Cost Search at first. Then I change the node cost by adding the heuristic value of each state.

2 Implementation and How it Works

2.1 Code Design

There are 6 .java file in the provided skeleton code. Like the previous assignment, the skeleton code's design uses the abstract class `SearchProblem` to generalize all search space problems. `SearchProblem` itself implements various types of search algorithms and defines an inner interface called `SearchNode` that subclasses of `SearchProblem` must implement. In this problem's case, `InformedSearchProblem` subclasses `SearchProblem` and then `SimpleMazeProblem` subclasses `InformedSearchProblem`. The `SimpleMazeProblem` thus implements `SearchNode` via `SimpleMazeNode`. The Maze class defines the properties related to the maze like width, height, actions and the functions like `toString` and `isLegal`. The `MazeView` class defines the attributes used in animating the motion of points.

2.2 Code and explanation

The model is implemented in `InformedSearchProblem.java`. Here's my code for `astarsearch`:

```
/**
 * @param frontier a priority queue to store the next explore states
 * @param explored a map to store backchaining information
 * @param nodecost a hashmap to store the state and its cost
 */
public List<SearchNode> astarSearch() {
  resetStats();
```

```
    PriorityQueue<SearchNode> frontier = new PriorityQueue<SearchNode>();
    HashMap<SearchNode, SearchNode> explored = new HashMap<SearchNode,
        SearchNode>();
    HashMap<SearchNode,Double> nodecost=new HashMap<SearchNode,Double>();
    explored.put(startNode, null); // startNode was not reached from any other
         node
    nodecost.put(startNode,startNode.getCost()+startNode.heuristic());//
        nodecost store the explored nodes
    frontier.add(startNode);

    while (!frontier.isEmpty()) {
      incrementNodeCount();

      updateMemory(frontier.size() + explored.size());

      SearchNode currentNode = frontier.remove();

      if (currentNode.goalTest()) {
        return backchain(currentNode, explored);
      }

      ArrayList<SearchNode> successors = currentNode.getSuccessors();

      for (SearchNode node : successors) {
        // if not visited or cost less
        if (!explored.containsKey(node)
            ||(node.getCost()+node.heuristic()<nodecost.get(node))){
          explored.put(node, currentNode);
          nodecost.put(node, node.getCost()+node.heuristic());
          frontier.add(node);
        }
      }
    }
    return null;
  }
```

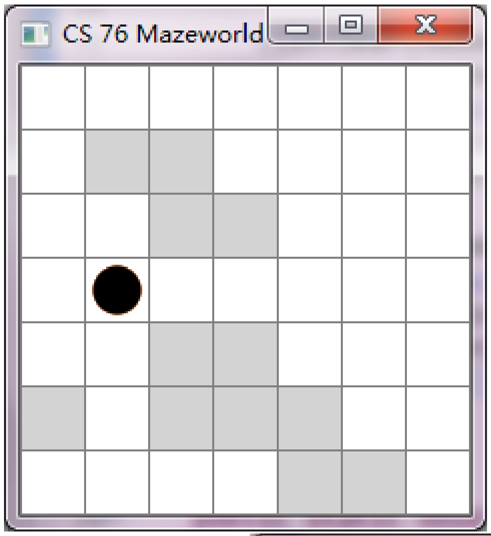
The key part of it is the Priority Queue. As Professor Balkcom mentioned, heap is the standard choice to implement priority queue. I find out that Java provides the `PriorityQueue` class which implements the min-heap of the elements in queue. Hence in this case the node with smallest cost will dequeue firstly every time. Like Breath First Search, it has an explored hashmap to store the explored state nodes and its father state. It ensures that (1) the state has not been visited or (2)although it has been explored, it has lower cost than the one in the explored map every time before it add current state to explored and the priority queue. Intuitively, I use a `nodecost` hashmap to store the cost of every node.

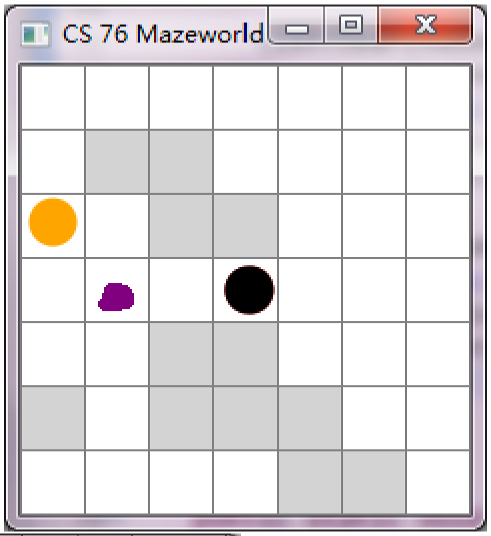
2.3 Testing the code:getSuccessors

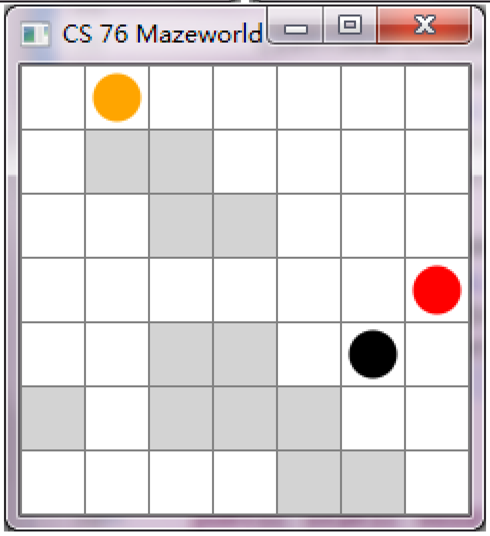
The result shows that three robots (in different colors) move in a maze by different path searching methods. The costs are

The screen shots are. The screen shots show that at first 3 robots walks together. At the purple dot, the orange one keep walk north since it applies DFS and the other two walk east. At last the robots apply BFS and A* search get to goal together with the same path length. The robot with DFS method will only change direction when it hit the wall or edge.

```
DFS:
  Nodes explored during search:  37
  Maximum space usage during search 41
BFS:
  Nodes explored during search:  21
  Maximum space usage during search 21
A*:
  Nodes explored during search:  19
  Maximum space usage during search 30
```







2.4 Discussion

The results show that A* search explore less nodes than the uninformed search methods DFS and BFS. This means that with some information the search can perform better with means can find a better path to goal. Here the information is the current cost and estimated cost.

3 Part two Multi-robot coordination: Introduction

This problem is about multi-robot path planning in an N*N square maze. Each robot takes turns to move and can move in 4 directions or stop at its turn. Only one robot can occupy on square at a time.
To plan paths in the complete state space, I consider the coordinates of all robots together as a state. One noteworthy point is that in `getsuccessor` function, I use a queue to store the coordinates after the robot who takes its turn move and finally it returns the state after all the robots move. There are many detail changes of the data structures due to the change of state space. These details will be discussed in the following implementations.

3.1 Implementation and How it Works

Q: If there are k robots, how would you represent the state of the system?
A: I represent the state with x and y coordinates of k robots in it. The state will have 2k numbers. To determine which action is legal to the state, the information of maze is needed. In addition, to determine which action is legal to the sub-step of each robot, the positions of other robot are needed. The former provided code uses int parameters to store the start and goal coordinates, so here I change the int into int[] and int[] into `ArrayList<int[]>` to store. For example xStart stores the x coordinates of all robots; in `getSuccessors()` `currentstate` in an `ArrayList<int[]>` which contains two int array that stores x and y coordinates of all robots in state.

The model is implemented in `MultiRobotProblem.java`. Here's my code:

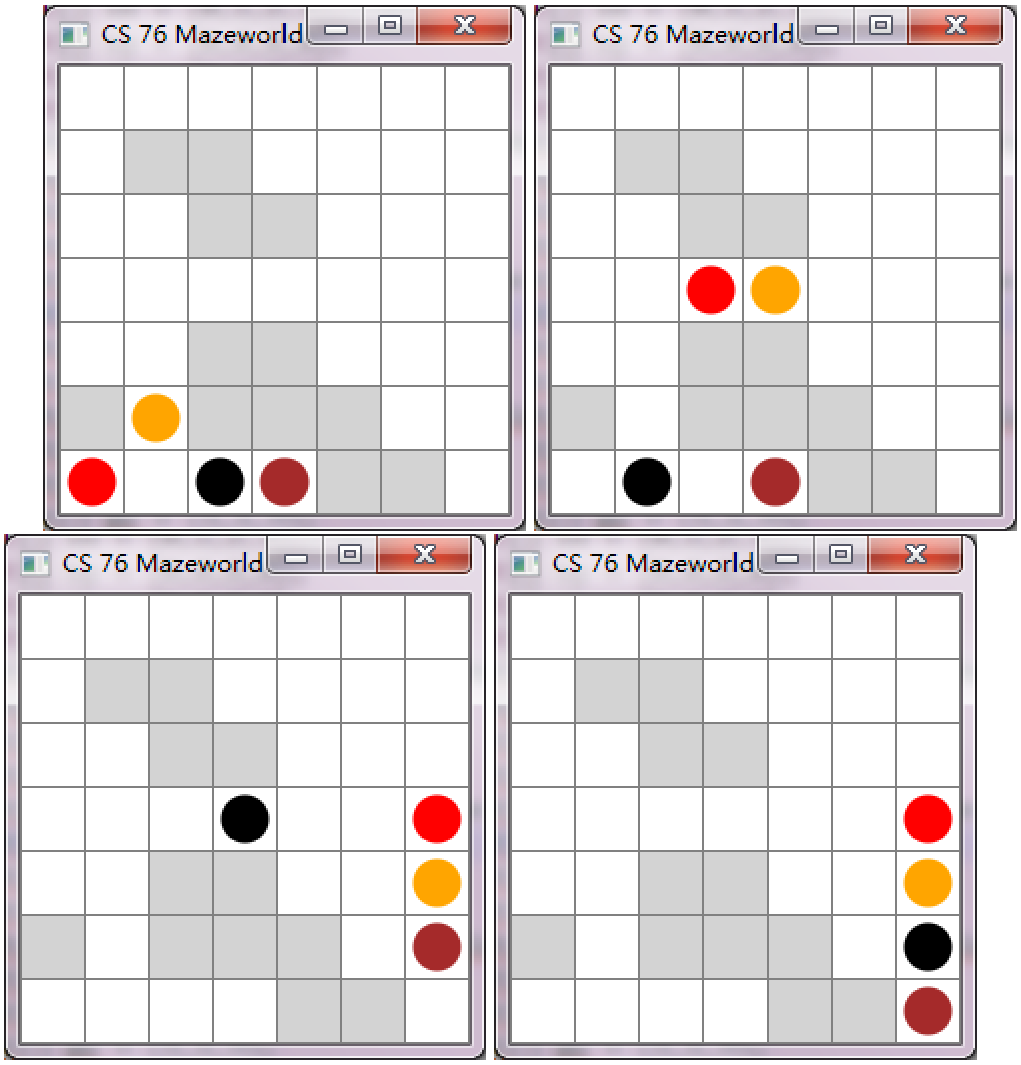
```
  public class MultiRobotProblem extends InformedSearchProblem {

private static int actions[][] = {Maze.NORTH, Maze.EAST, Maze.SOUTH, Maze.
    WEST, Maze.STILL};

private int[] xStart, yStart, xGoal, yGoal;

private Maze maze;

public MultiRobotProblem(Maze m, int[] sx, int[] sy, int[] gx, int[] gy) {
  startNode = new SimpleMazeNode(sx, sy, 0);
  xStart = sx;
  yStart = sy;
  xGoal = gx;
  yGoal = gy;

  maze = m;
}

// node class used by searches.  Searches themselves are implemented
//  in SearchProblem.
public class SimpleMazeNode implements SearchNode {
```

```
// location of the agent in the maze
protected ArrayList<int[]> state;

// how far the current node is from the start.  Not strictly required
//  for uninformed search, but useful information for debugging,
//  and for comparing paths
private double cost;
private int totalrobot;
public SimpleMazeNode(int[] x, int[] y, double c) {
  state = new ArrayList<int[]>(2);
  this.state.add(0, x);
  this.state.add(1, y);
  cost = c;
  totalrobot=state.get(0).length;
}

public int[] getX() {
  return state.get(0);
}

public int[] getY() {
  return state.get(1);
}

public ArrayList<SearchNode> getSuccessors() {

  ArrayList<SearchNode> successors = new ArrayList<SearchNode>();
  Queue<ArrayList<int[]>> sucqueue=new LinkedList<ArrayList<int[]>>();
  ArrayList<int[]> currentstate=new ArrayList<int[]>();//only state
  ArrayList<int[]> firststate=new ArrayList<int[]>();//contains the turn
  int totalturn=totalrobot;
  int[] turn={0};
  int currentturn=0;
  firststate.addAll(state);
  firststate.add(turn);
  sucqueue.add(firststate);
  while(!sucqueue.isEmpty()){
    currentstate=sucqueue.remove();//sucqueue contains the states after
        partial robots move
    currentturn=currentstate.get(2)[0];//it means the "currentturn"th
        robot has moved
    for (int[] action: actions) {//get success actions
      int[] xNew = currentstate.get(0).clone();
      int[] yNew = currentstate.get(1).clone();
      xNew[currentturn] = currentstate.get(0)[currentturn] + action[0];
      yNew[currentturn] = currentstate.get(1)[currentturn] + action[1];

      if(maze.isLegal(xNew, yNew,currentstate.get(0),currentstate.get(1)))
          {

        if(currentturn+1==totalturn){
          SearchNode succ = new SimpleMazeNode(xNew, yNew, getCost() +
              1.0);
          successors.add(succ);
        }
        else{
          ArrayList<int[]>newstate=new ArrayList<int[]>();
          newstate.add(0, xNew);
          newstate.add(1, yNew);
          turn[0]=currentturn+1;
          newstate.add(2, turn);
          sucqueue.add(newstate);
        }
      }
    }
  }
  return successors;
}


@Override
public boolean goalTest() {
  return Arrays.equals(state.get(0), xGoal) && Arrays.equals(state.get(1),
      yGoal);
}


// an equality test is required so that visited sets in searches
// can check for containment of states
@Override
public boolean equals(Object other) {
  int[] x=state.get(0);
  int[] y=state.get(1);
  for(int i=0; i<x.length;i++){
    if(x[i]!=((SimpleMazeNode) other).state.get(0)[i]||
        y[i]!=((SimpleMazeNode) other).state.get(1)[i])
      return false;
  }
  return true;
}

@Override
public int hashCode() {
  int hashcode=0;
  for(int hc=0;hc<totalrobot; hc++)
    hashcode+=(state.get(0)[hc]* 10 + state.get(1)[hc]*1)*Math.pow(10, hc)
        ;
  return hashcode;
}

@Override
public String toString() {
```

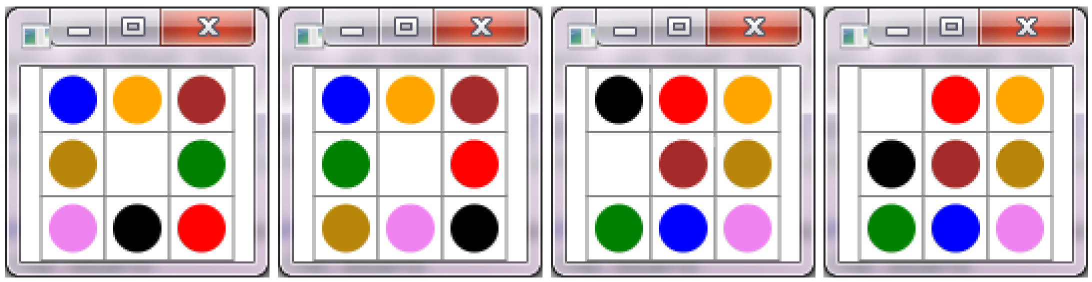
```
            String s=new String();
            s="Maze␣state␣" ;
            for(int i=0;i<totalrobot; i++) s+="["+state.get(0)[i]+","+state.get(1)[
                i]+"]";
            s+= "␣depth␣" + getCost()+"\n";
            return s;
        }

        @Override
        public double getCost() {
          return cost;
        }


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

          return dx+dy;
        }

        @Override
        public int compareTo(SearchNode o) {
          return (int) Math.signum(priority() - o.priority());
        }

        @Override
        public double priority() {
          return heuristic() + getCost();
        }
      }
    }
```

3.2 Testing and Result of the code:MultiRobotProblem

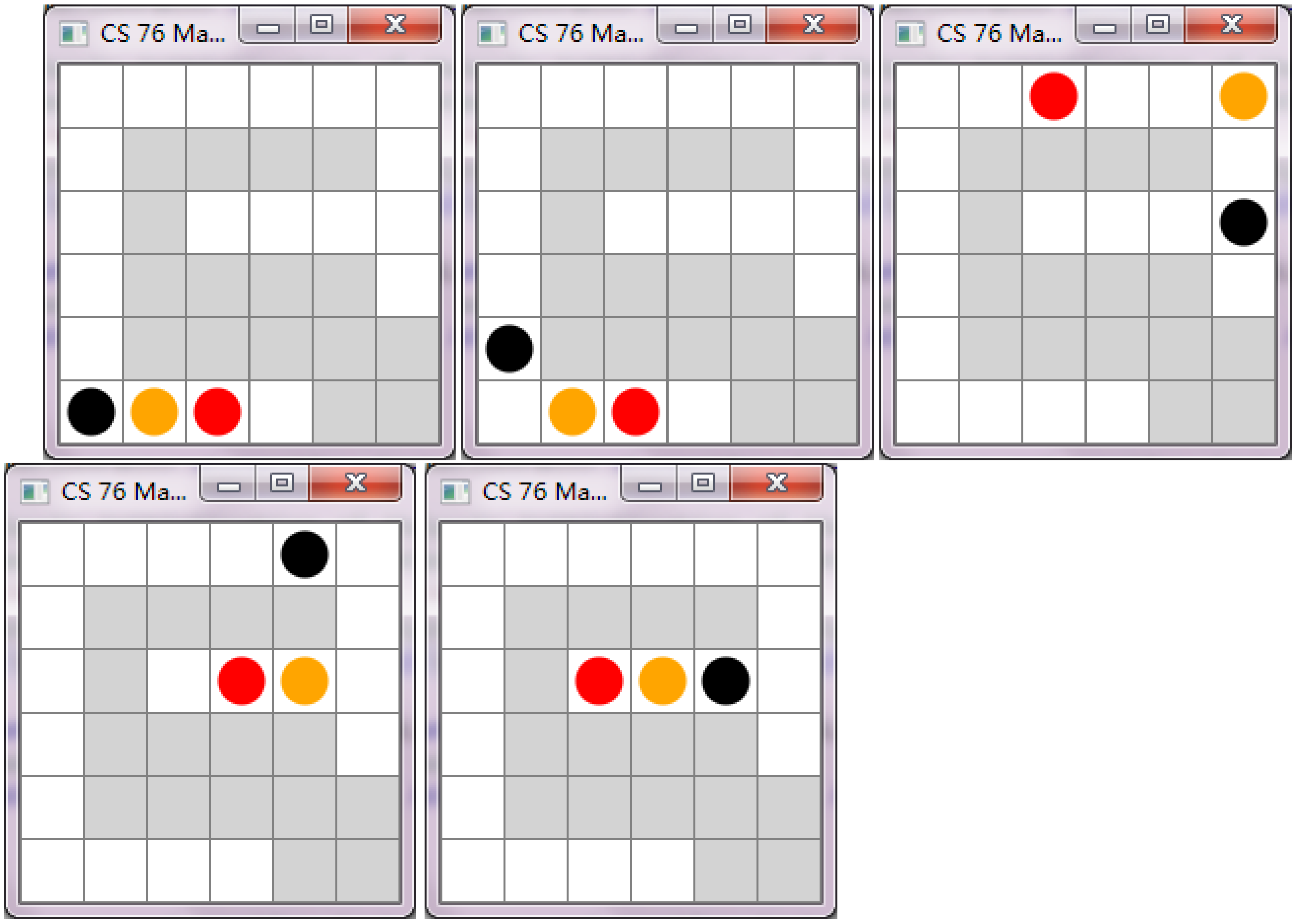
I have test it in simple problem like the example problem, 8 puzzles problem, the large 40 by 40 maze problem(the picture is too big that cannot be shown) and tricky problem with a tunnel in it. Following are the result(the firs is the starting state, the last is the end state).
sample maze with four robots



8 puzzles



tricky problem and the running time



```
A*:
  Nodes explored during search:  2858
  Maximum space usage during search 2686
```

3.3 Discussion

Q: Give an upper bound on the number of states in the system, in terms of n and k. A: Despite the validity of states and neglect the existence of walls, there are n*n positions in maze, which means (n2̂)*(n2̂-1)**(n2̂-k+1) is the upper bound of the state space. Q: Give a rough estimate on how many of these states represent collisions if the number of wall squares is w, and n is much larger than k. A: If the first one collides to wall, it has w possible positions. Then the second one will not collide with the first one so if it collides to wall it has w-1 possible positions to do that. If there are i robots collide into wall, the number of possible states is the combination of (i,k)*w*(w-1)**(w-i+1). Among the former states, there are Sum[(k!/i!*(k-i)!)*(w!/(w-k)!)],i [1,k] Q:If there are not many walls, n is large (say 100x100), and several robots (say 10), do you expect a straightforward breadth-first search on the state space to be computationally feasible for all start and goal pairs? Why or why not? A: I dont think BFS is feasible for this problem. According to the former estimation, the number of states will be (1002̂)*(1002̂-1)**(1002̂-9) which is close to 104̂0. Since the state space is very large and BFS requires searching almost all states to find the goal, it is impossible for a structure and common machine to store these states for BFS.

Q:Describe a useful, monotonic heuristic function for this search space. Show that your heuristic is monotonic. See the textbook for a formal definition of monotonic. A: I add up the Manhattan distance of all robots as the heuristic. The animation shows that some robots will walk back and forth when they wait for others. The reason for walking back and forth is that the robots take turns to walk, when one robot waits for others it can move one step and then come back or stop two turns. Its related to which successors the code gets at first.

Implement a model of the system and use A* search to find some paths. Test your program on mazes with between one and three robots, of sizes varying from 5x5 to 40x40. (You might want to randomly generate the 40x40 mazes.) I'll leave it up to you to devise some cool examples – but give me at least five and describe why they are interesting. (For example, what if the robots were in some sort of corridor, in the "wrong" order, and had to do something tricky to reverse their order?) 8-puzzle Q: Describe why the 8-puzzle in the book is a special case of this problem. Is the heuristic function you chose a good one for the 8-puzzle? A: They are similar problems because 8-puzzle can be seen as the 8 robot in a 9-square maze. I implement it and I think the Manhattan distance works well here. Q: The state space of the 8-puzzle is made of two disjoint sets. Describe how you would modify your program to prove this. (You do not have to implement this.) A: According to http://www.hh.se/download/18.519b93e12ad446eb4e80001211/aima, let N denote the number of lower numbers following a number. It shows that N is either always even or odd. In program, I will add a function to calculate the N for each state and store the Ns to see whether it is conserved to be even or odd.

4 Part three Blind Robot Problem: Introduction

In this problem the state space changes again. At the beginning, the robot doesnt know where it is and all valid squares in maze are its possible positions which are in its sate. When the robot moves, its possible positions will decrease since it collides with walls. Finally it will get to goal which only contains one position. I show the actions the robot has to move to goal, according to actions I can locate the robots position which I implement the animation of how it walks to goal.

4.1 Implementation and How it Works

```
public class BlindRobotProblem extends InformedSearchProblem {

  private static int actions[][] = {Maze.NORTH, Maze.EAST, Maze.SOUTH, Maze.
     WEST};

  //private int[] goal=new int[2];
  private ArrayList<int[]> GOAL=new ArrayList<int[]>(1);
  private ArrayList<int[]> START;
  private int totalrows;
  private int totalcols;
  private Maze maze;

  public BlindRobotProblem(Maze m, int[] g) {
    maze = m;
    GOAL.add(g);

    totalrows=maze.height;
    totalcols=maze.width;
    START=new ArrayList<int[]>(totalrows*totalcols);
    int[] s=new int[2];
    int i=0;
    for(int x=0;x<totalcols;x++)
      for(int y=0;y<totalrows;y++){
        if(maze.getChar(x, y)=='.'){
          s[0]=x;
          s[1]=y;
          START.add(i,s.clone());
          i++;
        }
      }
    int[] pastaction=new int[2];
    startNode = new SimpleMazeNode(START, 0,pastaction);
  }



  // node class used by searches.  Searches themselves are implemented
  //  in SearchProblem.
  public class SimpleMazeNode implements SearchNode {

    // location of the agent in the maze
    protected ArrayList<int[]> state;

    // how far the current node is from the start.  Not strictly required
    //  for uninformed search, but useful information for debugging,
    //  and for comparing paths
    private double cost;
    private int[] pastaction=new int[2];

    public SimpleMazeNode(ArrayList<int[]> s, double c,int[] a) {
      state = new ArrayList<int[]>();
      state=(ArrayList<int[]>) s.clone();
      cost = c;
      pastaction=a.clone();
    }

//    public int getX() {
//      return state[0];
//    }
//
//    public int getY() {
//      return state[1];
//    }

    public ArrayList<SearchNode> getSuccessors() {

      ArrayList<SearchNode> successors = new ArrayList<SearchNode>();
      ArrayList<int[]>sucstate=new ArrayList<int[]>();
      int[] s=new int[2];
      for (int[] action: actions) {
        sucstate.clear();
        for(int t=0; t<state.size();t++){
          s[0]=state.get(t)[0]+action[0];
          s[1]=state.get(t)[1]+action[1];
          if(maze.isLegal(s[0], s[1])) {
            //System.out.println("legal successor found " + " " + xNew + " " +
                 yNew);
            sucstate.add( s.clone());
          }
        }
        if(sucstate.size()==0)
          sucstate=(ArrayList<int[]>) state.clone();
        SearchNode succ = new SimpleMazeNode(sucstate, getCost() + 1.0,action)
            ;
        successors.add(succ);
      }
      return successors;

    }

    @Override
    public boolean goalTest() {
      return Arrays.equals(state.get(0), GOAL.get(0))&&state.size()==GOAL.size
          ();
    }


    // an equality test is required so that visited sets in searches
    // can check for containment of states
    @Override
    public boolean equals(Object other) {
      for(int t=0; t<state.size();t++)
        if(!Arrays.equals(state.get(t), ((SimpleMazeNode) other).state.get(t))
            ) return false;
      return true;
    }

    @Override
    public int hashCode() {
      int hc=0;
      for(int t=0; t<state.size();t++)
        hc+=state.get(t)[0]*100+state.get(t)[1];
      return hc;
    }

    @Override
    public String toString() {
      String s=new String();
      s="Maze␣state␣" ;
       int hash=pastaction[0]*1+pastaction[1]*10;
        switch(hash){
        case 10:
          s+="NORTH";
          break;
        case 1:
          s+="EAST";
          break;
        case -10:
          s+="SOUTH";
          break;
        case -1:
          s+="WEST";
          break;
        }
      for(int t=0; t<state.size();t++) s+="("+state.get(t)[0]+","+state.get(t
          )[1]+"),";
      s+= " depth " + getCost()+"\n";
      return s;
    }

    @Override
    public double getCost() {
      return cost;
    }
    @Override
    public int[] getAction(){
      return pastaction;
    }

    @Override
    public double heuristic() {
      // manhattan distance metric for simple maze with one agent:
      double dx=0;
      double dy=0;
      int d=0;
      for(int t=0; t<state.size();t++){
//          dx += Math.abs(GOAL.get(0)[0] - state.get(t)[0]);
//          dy += Math.abs(GOAL.get(0)[1] - state.get(t)[1]);
        dx = Math.abs(GOAL.get(0)[0] - state.get(t)[0]);
        dy = Math.abs(GOAL.get(0)[1] - state.get(t)[1]);
        d+=Math.sqrt(Math.pow(dx, 2) +Math.pow(dy, 2));
      }

//        return (dx+dy)/state.size();//average
      return (d)/state.size();//average
    }

    @Override
    public int compareTo(SearchNode o) {
      return (int) Math.signum(priority() - o.priority());
    }

    @Override
    public double priority() {
      return heuristic() + getCost();
    }
```

}

4.2 Discussing

Q: Describe what heuristic you used for the A* search. Is the heuristic optimistic? Are there other heuristics you might use?
A: I use the average Manhattan distance for positions in the state. I also try to use the average straight line distance between positions and goal. The result for both methods to solve the 4 by 3 maze shows below. We can see that the Manhattan distance is better and the search cost less time and explored fewer states than the one uses straight line distance. I think this is because the straight line distance cannot differentiate different states heuristic well. Many states have the same heuristic value.

4.3 Polynomial-time blind robot planning

(1) Prove that a sensorless plan of the type you found in problem 3 always exists, as long as the size of the maze is finite and the goal is in the same connected component of the maze as the start.
A: As the robot move, the positions in state will not increase but decrease or maintain the same. In the worst case, the robot can take turns to move in four directions until the state contain only one position. Then it can know the path from current position to goal since the goal is in the connected component of the maze as the start.
(2) Describe (but do not implement) a motion planner that runs in time that is linear or polynomial in the number of cells in the maze.
A: I am not sure I understand the question well, but I think one way to do this is like the method I mentioned above: to move in four directions until it find its position. This method costs polynomial time. Then it will cost linear time to find the goal using A* search.

5 previous work(I am a graduate student)

I read the Complete algorithms for cooperative pathfinding problems. The paper is about introducing and comparing few kinds of path searching methods for the multi-robot maze. These methods are based on the A* search. And the agents should move cooperatively and in more complicate way like diagonal move and move in circles. To solve the problem, the paper mentions two general methods: global search approaches which treat all agents as a single state and decoupled approaches which plan path for each agent separately. The former one is complete which means they can always find a path. While the latter one is fast.
The paper shows an example of decoupled approach called HCA*. It plan path for each agent in turn to avoid confliction. But it also causes the failure to find path for some agents because agents take turns in a fixed order. Another example is WHCA* which prohibit the confliction by creating laws to the move. But it will affect the quality of search since some optimal paths are pruned if they dont obey the law.
On the other hand, the paper shows some complete methods like Independence Detection (ID) and Operator Decomposition (OD). ID is an algorithm that finds path for each agent independently. OD is a global search A* algorithm that groups the conflict agents and find a new path for the new group. ID can avoid unnecessary merge by finding another path that will not conflict to other agents and OD as a cost limit to control the new path should have the same cost and the initial one. The whole process begins with ID assigning each agent to its own group and finding paths independently. Then OD will merge conflict paths until there are no conflicts. Moreover, the paper talks about the two constraints that makes OD+ID algorithm optimal. Then it shows a higher-quality algorithm called MGSx which dynamically drop the constraints. This means x is the maximum group size of ID+OD method.
Then it discusses the optimal anytime (OA) algorithms which can stop at any time which is different from OD+ID and MGS algorithm. But MGS can be adapted to OA by doing iterative deepening.
Finally, it shows the performance of each algorithm. The data demonstrates that the HCA* has the lowest average running time but it costs too much extra moves. MGS2 seems to have the best performance since

it has both low value of moves and time. In addition, the OA algorithms can generate high-quality paths which are suitable in the digital entertainment.