Noah Buchanan
Problem Set 6
AI at 5:25 PM

May 11, 2021

Critical Thinking

1. A priority queue is "semi-sorted" because the smallest(or largest) value depending on how you are implementing the priority queue will always be the first value so that when you pop() the head off of the priority queue it guarantees the smallest or largest. HOWEVER it makes no guarantees past that point it only worries about the head, for example the other end if say you were sorting by largest, would not always be the smallest as you would expect for a sorted data structure. A binary search tree in comparison, the leftmost or rightmost nodes will be the smallest or largest that much we can be sure of.

2. UCS time complexity: $O(b^{C/e})$ where b is the branching factor(how many edges are present for a node, C is the cost of the optimal solution and e is the minimum cost of one step. Whereas DFS time complexity: $O(V+E)$ in the case where the entire tree is traversed.

3. iterative deepening search is repeatedly running a depth-limited version of depth-first search with increasing depths until the goal state is found. Iterative Deepening Depth-first search is complete when the number of nodes is finite and optimal if path cost is a function non-decreasing with depth.

4. A heuristic function in AI is a function that ranks steps in algorithms at each branching step based on any available information to decide which path would be optimal to follow. It is important for a heuristic to never overestimate based on available information for the solution to be optimal. A heuristic CAN overestimate but if it is allowed it may not always result

in an optimal solution.

5. (a) An algorithm is complete if for an arbitrary input it guarantees a correct result.

   (b) An algorithm is optimal if at worst performs only a constant factor worse than the best possible performance.

   (c) Complete Algorithms: Best First Search, BFS, UCS, Iterative Deepening Search, Bidirectional search, A*, Recursive Best First Search, Iterative deepening A* search, bidirectional A* search.

   Optimal Algorithms: UCS, Iterative Deepening Search, bidirectional search, A*, Bidirectional A* search(considered quasi-optimal), Iterative deepening A* search

Code Modified in GameAgent.java

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.PriorityQueue;

import javax.swing.Timer;

/******************************
Name: Noah Buchanan
Username: ua100
Problem Set: PS6
Due Date: May 11,2021
******************************/


class GameAgent {

  static Node[][] terrain = new Node[1200][600];
  static int[][] possibleMoves = new int[8][2];
  final static Node nil = new Node();
  ArrayList<Node> path = new ArrayList<>();

  static class Node {

    boolean visited = false;
    double pathcost = 0;
    Node predecessor = nil;
    int x;
    int y;
}

void reset(GameModel m) {
    for (int i = 0; i < terrain.length; i++) {
        for (int j = 0; j < terrain[0].length; j++) {
            terrain[i][j].visited = false;
            terrain[i][j].predecessor = nil;
            terrain[i][j].pathcost = m.getSpeedOfTravel(i, j);
        }
    }
}

void UCS(int x, int y, int xdest, int ydest) {
    path = new ArrayList<>();
    Comparator<Node> comp = new Comparator<Node>() {
        public int compare(Node i, Node j) {
            if (i.pathcost < j.pathcost) {
                return 1;
            } else if (i.pathcost > j.pathcost) {
                return -1;
            }
            return 0;
        }
    };

    PriorityQueue<Node> Q = new PriorityQueue<Node>(comp);
    terrain[x][y].pathcost = 0;
    terrain[x][y].visited = true;
    terrain[x][y].predecessor = nil;
   Q.add(terrain[x][y]);
    Node u = null;
    while (!Q.isEmpty()) {
       u = Q.remove();
        if (Math.sqrt(Math.pow(u.x - xdest, 2) + Math.pow(u.y - ydest, 2)) <= 10)
            break;// node is in euclidian distance of 10 from solution
        }
        for (int i = 0; i < possibleMoves.length; i++) {
            if (u.x > 10 && u.y > 10 && u.x < 1190 && u.y < 590) {
                Node v = terrain[u.x + possibleMoves[i][0]][u.y +
                                                          possibleMoves[i][1]];
                if (!v.visited) {
                    v.visited = true;
```

```
          v.pathcost += Math.sqrt(Math.pow(u.x - v.x, 2) +
              Math.pow(u.x - v.x, 2));
          v.predecessor = u;
          Q.add(v);
        }
      }
    }
    u.visited = true;
  }
  while (u.predecessor != nil) {
    path.add(u);
    u = u.predecessor;
  }

}

void A_star(int x, int y, int xdest, int ydest) {
  path = new ArrayList<>();
  Comparator<Node> comp = new Comparator<Node>() {
    public int compare(Node i, Node j) {
      if (i.pathcost < j.pathcost) {
        return 1;
      } else if (i.pathcost > j.pathcost) {
        return -1;
      }
      return 0;
    }
  };

  PriorityQueue<Node> Q = new PriorityQueue<Node>(comp);
  terrain[x][y].pathcost = 0;
  terrain[x][y].visited = true;
  terrain[x][y].predecessor = nil;
  Q.add(terrain[x][y]);
  Node u = null;
  while (!Q.isEmpty()) {
    u = Q.remove();
    if (Math.sqrt(Math.pow(u.x - xdest, 2) + Math.pow(u.y - ydest, 2)) <= 10)
      break;// node is in euclidian distance of 10 from solution
    }
    for (int i = 0; i < possibleMoves.length; i++) {
      if (u.x > 10 && u.y > 10 && u.x < 1190 && u.y < 590) {
        Node v = terrain[u.x + possibleMoves[i][0]][u.y + possibleMoves[i][1]]
        if (!v.visited) {
          v.visited = true;
          v.pathcost += Math.sqrt(Math.pow(u.x - v.x, 2) +
              Math.pow(u.x - v.x, 2)) + heuristic(v.x,v.y,xdest,ydest);
          v.predecessor = u;
          Q.add(v);
        }
      }
    }
    u.visited = true;
  }
  while (u.predecessor != nil) {
    path.add(u);
    u = u.predecessor;
  }

}

public double heuristic(int x, int y, int xdest, int ydest) {

  double max = 0;
  int maxx = 0;
  int maxy = 0;
  for(int i = 0; i < (int)GameModel.XMAXIMUM; i++) {
    for(int j = 0; j < (int)GameModel.YMAXIMUM; j++) {
      if(terrain[i][j].pathcost > max) {
        max = terrain[i][j].pathcost;
        maxx = i;
        maxy = j;
      }
    }
  }
  double distance = Math.sqrt(Math.pow(terrain[maxx][maxy].x-xdest, 2) +
      Math.pow(terrain[maxx][maxy].y-ydest, 2));

  return terrain[maxx][maxy].pathcost*-distance;
}

static void assignCosts(GameModel m) {
  for (int i = 0; i < terrain.length; i++) {
    for (int j = 0; j < terrain[0].length; j++) {
      terrain[i][j] = new Node();
    }
  }
  if (m != null) {
    for (int i = 0; i < terrain.length; i++) {
      for (int j = 0; j < terrain[0].length; j++) {
        terrain[i][j].pathcost = m.getSpeedOfTravel(i, j);
        terrain[i][j].x = i;
```

```
          terrain[i][j].y = j;
        }
      }
    }
//up left
    possibleMoves[0][0] = -10;
    possibleMoves[0][1] = +10;
//up middle
    possibleMoves[1][0] = 0;
    possibleMoves[1][1] = +10;
//up right
    possibleMoves[2][0] = +10;
    possibleMoves[2][1] = +10;
//middle right
    possibleMoves[3][0] = +10;
    possibleMoves[3][1] = 0;
//down right
    possibleMoves[4][0] = +10;
    possibleMoves[4][1] = -10;
//down middle
    possibleMoves[5][0] = 0;
    possibleMoves[5][1] = -10;
//down left
    possibleMoves[6][0] = -10;
    possibleMoves[6][1] = -10;
//middle left
    possibleMoves[7][0] = -10;
    possibleMoves[7][1] = 0;
  }

  void drawPlan(Graphics g, GameModel m) {
    g.setColor(Color.red);

    int prevx = (int) m.getDestXValue();
    int prevy = (int) m.getDestYValue();
    for (int i = path.size()-1; i >= 0; i--) {
      g.drawOval(prevx-3, prevy-3, 6, 6);
      g.drawLine( prevx, prevy, (int) path.get(i).x, (int) path.get(i).y);
      prevx = path.get(i).x;
      prevy = path.get(i).y;
    }

  }

  void update(GameModel m) {
    GameController c = m.getController();
    if(path.size() > 0) {
      if(Math.abs(m.getX()-m.getDestXValue()) < 0.001 &&
          Math.abs(m.getY()-m.getDestYValue()) < 0.001) {
        Node dest = path.remove(path.size()-1);
        m.setDest(dest.x, dest.y);
      }
    } else {
      reset(m);
    }
    while (true) {

      MouseEvent e = c.nextMouseEvent();
      if (e == null)
        break;

      if(e.getButton() == MouseEvent.BUTTON3) {
        A_star((int) m.getX(), (int) m.getY(), e.getX(), e.getY());
      } else if(e.getButton() == MouseEvent.BUTTON1) {
        UCS((int) m.getX(), (int) m.getY(), e.getX(), e.getY());
      }


    }
  }

  public static void main(String[] args) throws Exception {
    GameController c = new GameController();
    c.initialize();

// This will instantiate a new instance of JFrame. Each will spawn in another
// thread to generate events
// and keep the entire program running until the JFrame is terminated.
    c.view = new GameView(c, c.model);
    assignCosts(c.model);

// this will create an ActionEvent at fairly regular intervals. Each of the
// events are handled by
// GameView.actionPerformed()
    new Timer(20, c.view).start();
  }
}
```