Adam Rohdin - adaro815
David Tran - davtr766

Lab1 Report - Intelligent Agents

The definition of an intelligent agent is an autonomous entity which observes the environment through sensors and then receives a perception and then takes an action based on the actuators to achieve its goal. The “agent” paradigm has rapidly become one of the major conceptual frameworks for understanding and structuring activities within Artificial Intelligence.

In this lab, an agent is introduced and the goal is to implement an agent’s behavior to provide cleaning capabilities in the vacuum cleaning world. This can be done through a variety of search algorithms. The algorithm proposed in this report is the A* search algorithm.

The A* search algorithm is one of the most popular algorithms to solve problems that involves pathfinding and graph traversals. In our case, the goal is to find the most appropriate path.

We chose A* for both the expansion (exploration) of the world as well as for the navigation home. To enable the use of A* as an exploration algorithm we create a stack of nodes corresponding to unknown tiles surrounding the player at any instance. We then take the last node from that stack and perform A* to generate a path and navigate to that tile. By doing this we can explore the whole field using A*.

When generating the path using A* we use a heuristic based on the manhattan distance. This works, but it’s not optimal. A better solution would be to also base the heuristic on the number of turns it would take to go to a specific tile. Since only one action (turn left, turn right, move or suck) can be completed each game-loop it is in the interest of the algorithm to minimize the number of turns in order to complete the task as quickly as possible.

```
private LinkedList<int[]> AStar(int[] start, int[] goal) {

    LinkedList<int[]> path = new LinkedList<int[]>();
    int pathCost = 0;
    ArrayList<int[]> visited = new ArrayList<int[]>();

    path.add(start);
    visited.add(start);


    while(!path.isEmpty() && !Arrays.equals(path.peekLast(), goal)) {
        if( !Arrays.equals(path.peekLast(), start))
            break;

        int[] bestNextStep = {};
        int lowestCost = (int)Double.POSITIVE_INFINITY;
        int[] current = path.peekLast();

        ArrayList<int[]> directions = new ArrayList<int[]>();
            directions.add(new int[] {0, -1});
            directions.add(new int[] {1, 0});
            directions.add(new int[] {0, 1});
            directions.add(new int[] {-1, 0});

        for(int[] nextDirection : directions) {
            int fn, gn, hn = 0;
            int[] next = {current[0] + nextDirection[0], current[1] + nextDirection[1]};

            if(next[0] < 1 || next[0] > 29 || next[1] < 1 || next[1] > 29)
                continue;

            if(!compareTo(visited, next) && state.world[next[0]][next[1]] != state.WALL) {
                gn = pathCost + 1;
                hn = calcManhattan(next, goal);
                fn = gn + hn;
                if(fn < lowestCost) {
                    lowestCost = fn;
                    bestNextStep = next;
                }
            }
        }

        if(bestNextStep.length == 0)
            path.pop();
        else {
            path.add(bestNextStep);
            pathCost++;
            visited.add(bestNextStep);
        }
    }

    path.removeFirst();
    return path;
}
```

Fig. 1 This is the code used in A*, contains some small bugs....

When we've found a path using A* we then start the procedure of moving along the given path one tile at the time. Moving a tile is not as simple as to tell the program to move to the next tile. This is due to the fact that the player has to actually turn the character around before it can move in a new direction. On order to do this we implemented something called an *actionQueue*. This actionQueue is is a queue which holds the required actions that must be performed in order to navigate to the next tile in the generated path. This means that if the character is looking EAST and we want to go WEST the actionQueue contains {TURN_LEFT, TURN_LEFT, MOVE_FORWARD}.

```
private LinkedList<Action> generateActionQueue(int[] lookVector, int[] walkVector){
    LinkedList<Action> actionQueue = new LinkedList<Action>();

    int[] from = lookVector;
    int[] to = walkVector;

    int angle = getAngle(from, to);

    int v21 = from[0] * to[1];
    int v12 = from[1] * to[0];

    if(v21 < 0)
        angle*=-1;
    if(v12 > 0)
        angle*=-1;

    if(angle == 0) {
        actionQueue.add(LIUVacuumEnvironment.ACTION_MOVE_FORWARD);
    }
    else if(angle == 180) {
        actionQueue.add(LIUVacuumEnvironment.ACTION_TURN_LEFT);
        actionQueue.add(LIUVacuumEnvironment.ACTION_TURN_LEFT);
        actionQueue.add(LIUVacuumEnvironment.ACTION_MOVE_FORWARD);
    }
    else if(angle > 0) {
        actionQueue.add(LIUVacuumEnvironment.ACTION_TURN_RIGHT);
        actionQueue.add(LIUVacuumEnvironment.ACTION_MOVE_FORWARD);
    }
    else {
        actionQueue.add(LIUVacuumEnvironment.ACTION_TURN_LEFT);
        actionQueue.add(LIUVacuumEnvironment.ACTION_MOVE_FORWARD);
    }


    return actionQueue;
}
```

Fig 2. Code used to generate the actions required to move to the next tile.

In order to generate this queue we calculate the angle between where the character is looking and where it wants to go. From this angle we know what queue of actions needs to be performed. It ***is*** more optimal to hard-code this instead of calculate angles.

This pattern of adding unknown tiles to a stack, generating a path using A* and then generating a queue based on the actions required to get to that sequence of tiles is done until the stack of unknown tiles are empty. When this state is reached we know that all available tiles are explored and that we now can switch goals from exploration to navigation home.
To navigate home we simply set the goal-node to {1,1} and then pass that into the A* algorithm which generates a path home.