Lab 1 TDDC17

Oskar Håstad, Martin Hinnerson

Version 0.1

2018-11-07

Status

Reviewed		
Approved		

Project identity

2018/Autumn term
Tekniska högskolan vid Linköpings universitet, IDA

Name	Telephone	Email
Oskar Håstad		oskha905@student.liu.se
Martin Hinnerson	072 210 43 72	marhi386@student.liu.se

Course responsible: Patrick Doherty, patric.doherty@liu.se
Lab Assistant: Axel Strandberg, axest863@student.liu.se

1 Task 1

First we evaluated the possibility of starting the dirt-sucking right away, regardless of where the agent starts, and working our way to the outer walls in a spiral-like motion. However, we realized this would result in too many unique cases of tiles left to cover when it hits a wall. Therefore we decided to use a very systematic approach instead.

Our systematic approach begins with the agent finding a corner. Since the ending position is the north-west corner we wanted to use another one, and picked the south-east. We do this by using cases. Our first case consists of the agent turning south and moving forward until it hits a wall. The second case includes turning east and moving forward until it hits a wall. Following this is the cleaning-case which turns north and then moves forward until it hits a wall. If it hits a wall it will turn west and walk one step forward and then turn the opposite direction than it was facing when hitting the wall. This will result in a systematic north-to-south and south-to-north cleaning motion. It will stop when it finally hits a wall while facing west and will then move as far north as possible to ensure being back at the home position.

We decided a systematic approach would be best since all tiles had to be covered eventually, and the only wasted actions are the ones finding the corner. We concluded that avoiding these wasted actions would be extremely difficult and not all that time-saving. Therefore we decided that there were no viable alternatives. Our agent is a lot better than the reactive one; the reactive one will only turn when it hits a wall and will therefore only sweep adjacent to the walls and miss all other tiles.

2 Task 2

In task two we first tried to think of ways to improve the solution from task one by for example always redirection around objects we bump into in different ways. We realised that this would not give us the required performance. We decided to use a search algorithm to solve the problem. After studying chapter 3 in the course book we decided to implement a broad-first search algorithm (BFS). This decision was made because the BFS was said to be complete and optimal if the step cost was constant. On top of that the algorithm didn't seem to complicated to implement.

To implement this we added a few more things to store in the `MyAgentState` class. Firstly we added an new class called `Square` which holds the information for a square in the grid such as coordinates, parent and path cost. In the agent we then create a new grid called `BFSgrid` (similar to `world`) where each item is a `Square`. The squares in this grid will be updated by the BFS when searching for a new goal node. We also added a FIFO-queue called `BFSqueue`. This is the queue used by the BFS to put the next available nodes during search. These are the most significant additions, a few more variables are added which can be found in the code.

The main loop now works like this:

1. If we hit Dirt -> Suck
2. If we bump into something, rerun the BFS and get a new goal square and path.
3. If we are at the goal:
   (a) If goHome is true we are done
   (b) Else we run the BFS again to find the next goal.
4. Loop though the parents from the goal square back to the current square. This is the path we want to go. When we have looped back to the current square we know which adjacent square to move to.
5. We now call our movement function called `move` which will perform the correct movement according to our rotation.

The BFS function is also important in our solution and it works like this:

1. Reset the `BFSgrid` and clear the `BFSqueue`.
2. Add the current square to the queue
3. Loop through the queue until its empty and perform the following actions. Note, if it is empty it means we have no more squares to explore and we are done and can go home.
   (a) Pop the first square of the queue
   (b) If an adjacent square is not already searched (cost = -1), if it's not outside of the maximum grid size and if its not a wall, then:
      - increment its cost by 1
      - set its parent to the current position
      - If it is an unknown square set it as the new goal and return. Else, push the square to the `BFSqueue`

In most cases the BFS will find a directly adjacent square that is unknown and thus only go to depth one of the search tree. In the end when most squares are explored we will have to search deeper to find new goals, thus pushing more nodes to the queue and then searching them before returning.

The solution we chose looks like a well performing search algorithm for this case. What we realised was that the path cost was not really uniform since the cost of turning makes square in front of the agent cost less. However after discussing and trying out a solution that would prioritise squares in front of the agent we saw that this made the agent move all over the grid, missing single squares unknown in many places. The agent then has to move many moves just to explore single squares which reduces the total performance of the agent.