CS26410 Report 3
Hide and Seek

Chris Savill
chs17@aber.ac.uk

April 16, 2013

Contents

1 Description of algorithms used

1.1 Depth-First-Search (DFS) algorithm

The DFS algorithm is used to aid in exploring a whole area. The algorithm requires that a stack is used to store the path along which the robot has travelled to get to its current cell on the occupancy grid. The reason it acts as a DFS is because the algorithm keeps the robot searching further rather than restricting it to searching a close area to a start point and then working out.

```
if (oG->getNeighboursUnexplored() != 0) {
    cout << "Picking a neighbour to explore..." << endl;
    targetDirection = oG->chooseNextCell(currentDirection); //(
} else if (oG->getNeighboursUnexplored() == 0) {
    cout << "All neighbours of current cell explored." << endl

    if (oG->checkFinished()) {
        cout << "Finished Mapping.";
    } else if (!oG->getPathStack().empty()) {
        cout << "Path not empty." << endl;
        targetDirection = oG->getDirectionOfLastCell(); //Gets
        cout << "New Direction: " << targetDirection << endl;
    }
}
```

Figure 1: Snapshot of DFS code.

The algorithm has a way of determining if the mapping of an area is complete. The check is a function that evaluates if every cell that has been explored and has no obstacle present, has no unexplored neighbours. If this return true then the search and mapping is complete so the path stack is emptied thus finishing the mapping task.

```
bool Occupancy_Grid::checkFinished() {
    cout << "Checking if finished..." << endl;

    for (int counter = 0; counter < pathLength; counter++) {
        if (grid[pathStack.at(counter).xPos][pathStack.at(counter).yPos].isExplored == true &&
                grid[pathStack.at(counter).xPos][pathStack.at(counter).yPos].obstacleValue == 0
                && grid[pathStack.at(counter).xPos][pathStack.at(counter).yPos].neighboursUnexplored != 0) return false;
    }

    pathLength = 0;
    pathStack.erase(pathStack.begin(), pathStack.end());
    return true;
}
```

Figure 2: Snapshot of check finished function code.

The way in which the algorithm chooses which direction to continue in at each cell is quite simple. If the cell directly in front of the robot is explorable, the robot will continue in that direction. This was implemented as it speeds up the search because the robot takes a long time to turn. If the cell in front has an obstacle present, a random number is generated that correlates to each of the 4 directions it can travel and checks if that cell is unexplored and has no obstacle present. This random number generator is constantly generated until a cell that is unexplored and is explorable is chosen.

1.2 Localisation algorithm

The localisation algorithm is use to determine whether the robot is within the occupancy grid and if so, where. The algorithm itself works as follows. A new temporary occupancy grid is created for the robot to use for its mapping of its immediate area. Once the immediate area is scanned the temporary grid is compared to all areas of the original occupancy grid. The comparison algorithm uses a 2 for loops to ensure that the temporary grid area is compared to all possible areas of the original grid. The temporary grid is compared to areas of the same size and each individual cell is compared with the corresponding cell of the area being compared with in the original grid.

```
bool Occupancy_Grid::compareArea(Occupancy_Grid *temp, int xCounter, int yCounter) {
    int xEnd = xCounter + temp->getXLength() - 1;
    int yEnd = yCounter + temp->getYLength() - 1;
    int tempXCounter = 0;
    int tempYCounter = 0;

    while (xCounter <= xEnd && yCounter <= yEnd) {
        if (!compareCells(grid[xCounter][yCounter], temp->getGrid()[tempXCounter][tempYCounter])) {
            possibleRobotX = 0;
            possibleRobotY = 0;
            return false;
        }

        if (tempXCounter == temp->getRobotX() && tempYCounter == temp->getRobotY()) {
            possibleRobotX = xCounter;
            possibleRobotY = yCounter;
        }

        if (xCounter >= temp->getXLength() - 1) {
            yCounter++;
            tempYCounter++;
        } else {
            xCounter++;
            tempXCounter++;
        }
    }

    return true;
}
```

Figure 3: Snapshot of compare area function code.

If the localisation of that area is unsuccessful then the algorithm exits telling the user that the robot is not within the original occupancy grid area. If there is more than 1 place the robot could possible be, the robot uses the DFS algorithm and moves to the next unexplored cell and tries to localise with the additional information gathered. If the comparison algorithm determines that there is only one possible place for the robot to be, the robot switches to using the original occupancy grid and is localised to the area it was found in. As localisation requires that an original occupancy grid exists to compare with, there is a boolean value that tell the robot that it mapped successfully so localisation can be attempted.

```
int Occupancy_Grid::attemptLocalisation(Occupancy_Grid * temp) {
    int possibleAreas = 0;

    for (int xCounter = 0; xCounter < (xLength - temp->getXLength()); xCounter++) {
        for (int yCounter = 0; yCounter < (yLength - temp->getYLength()); yCounter++) {
            if (compareArea(temp, xCounter, yCounter)) possibleAreas++;
            if (possibleAreas > 1) return 2; //More than one possible area.
        }
    }

    if (possibleAreas == 1) return 3; //True.
    else return 1; //False.
}
```

Figure 4: Snapshot of attempt localisation function code.

1.3 Hiding algorithm

The hiding algorithm consists of 3 parts; the first part involves finding the best place to hide, the second part involves finding the shortest path to that position and the third part involves getting the robot to travel along that path to its hiding location. The hiding function can only be accessed if the localisation of the robot is successful. In order to find the best hiding spot for the robot the algorithm first searches for empty, explored cells and checks if the have 3 walls around them, if none can be found does the same but looks for 2 walls and then 1.

```
void Occupancy_Grid::getHideLocation() {
    if (!findLocation(3)) {
        cout << "Could not find a location with 3 walls surrounding. Trying 2 wall locations..." << endl;

        if (!findLocation(2)) {
            cout << "Could not find a location with 2 walls surrounding. Trying 1 wall locations..." << endl;
            findLocation(1);
        }
    }
}
```

Figure 5: Snapshot of code that calls function to search for a hiding place.

```
bool Occupancy_Grid::findLocation(int numberOfWalls) {
    for (int xCounter = 0; xCounter < xLength; xCounter++) {
        for (int yCounter = 0; yCounter < yLength; yCounter++) {
            if (grid[xCounter][yCounter].obstacleValue == 0 && grid[xCounter][yCounter].isExplored == true) {
                if (getNumberOfWalls(xCounter, yCounter) == numberOfWalls) {
                    frontier.resize(++frontierLength);
                    frontier.back().xPos = robotX;
                    frontier.back().yPos = robotY;
                    frontier.back().pathValue = 0;
                    if (plotPath(robotX, robotY, xCounter, yCounter, 0)) {
                        cout << "Found a location surrounded by " << numberOfWalls << " walls and plotted path." << endl;
                        return true;
                    }
                }
            }
        }
    }

    return false;
}
```

Figure 6: Snapshot of function that tries to find a hiding location and plot a path using other functions.

The path finding algorithm is a simple A* search that keeps track of the distance/cost so far and the distance/cost left. The search tries to find the shortest path possible as the search is guided in the direction of the target cell and the total goal value is underestimated not overestimated so it is admissible. Once the path as been plotted from the robot's current location to its hiding location, the robot moves cell by cell along its path until the path stack is empty, at which point the robot should be at its hiding location.

```
bool Occupancy_Grid::plotPath(int currentX, int currentY, int targetX, int targetY, int cost) {
    int distanceRemaining = (targetX - currentX) + (targetY - currentY);
    addCellToPath(frontier.back().directionCameFrom, frontier.back().xPos, frontier.back().yPos);

    if (currentX == targetX && currentY == targetY) { //Checks if at target cell.
        neighbours.erase(neighbours.begin(), neighbours.end());
        frontier.erase(frontier.begin(), frontier.end());
        return true;
    } else {
        frontier.pop_back();
        getExplorableNeighbours(currentX, currentY, cost, distanceRemaining); //Gets list of neighbours that

        for (int counter = neighboursLength - 1; counter >= 0; counter--) {
            addNodesToFrontier();
            if (plotPath(frontier.back().xPos, frontier.back().yPos, targetX, targetY, cost + 1) == true) {
                return true;
            } else {
                pathStack.pop_back();
                pathLength--;
            }
        }
    }
    pathStack.pop_back();
    pathLength--;
    neighbours.erase(neighbours.begin(), neighbours.end());
    neighboursLength = 0;
    frontier.erase(frontier.begin(), frontier.end());
    frontierLength = 0;
    return false;
}
```

Figure 7: Snapshot of function that uses performs an A* search with the aid of other functions.

1.4 Seeking algorithm

The seeking algorithm is very simple, if the robot localised successfully, the robot will perform a DFS of the whole pre-explored area, if however any cells that did not contain an obstacle on the original occupancy grid are detected to have an obstacle, then the DFS finishes and the anomaly is stated to the user, thus finishing the seeking function.

```
bool Occupancy_Grid::detectAnomaly(int xPos, int yPos) {
    if (grid[xPos][yPos].obstacleValue == 0) {
        cout << "Anomaly found at grid co-ordinates (" << xPos << ", " << yPos << ")" << endl;
        anomalyFound = true;
        return true;
    } else return false;
}
```

Figure 8: Snapshot of function that determines if an anomaly has been found.

2 Trial runs

2.1 Simulated trial runs

In order to perform real world trials, the robot program first had to be tested through the use of Player/Stage. Before testing proceeding onto real world trials, the robot had to successfully map the whole area of the simulated world provided. Many of the runs were unsuccessful resulting in the robot crashing at some point. Below is a series of screen shots of one of the successful trials for mapping:

Key:

Icon	What it represents
#	Obstacle present
~	Unexplored cell
R	Robot's current location
	An empty cell represents an explored cell with no obstacle present.

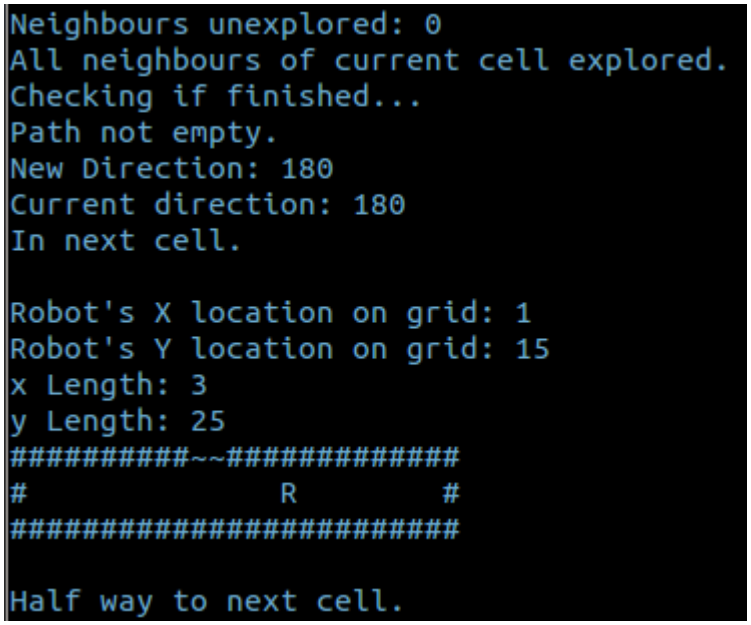

Figure 9: Screen shot of the terminal window corresponding to the screen shot below. Having mapped the the bottom area and hitting a dead end, the robot return along its path to the next available unexplored cell.

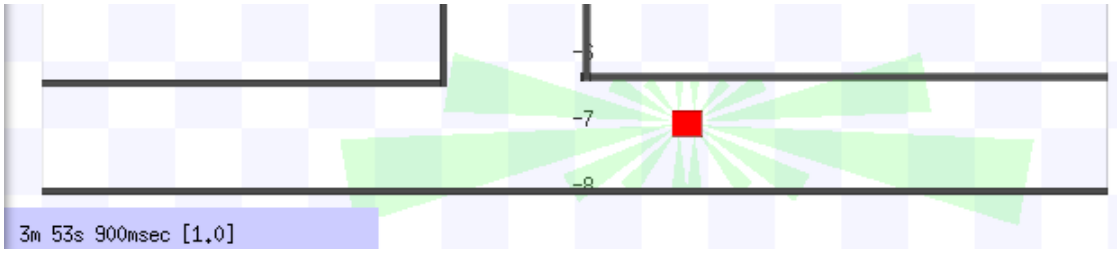

Figure 10: Screen shot of the Stage window corresponding to the screen shot above.

```
Neighbours unexplored: 2
Picking a neighbour to explore...
Current direction: 90
In next cell.

Robot's X location on grid: 1
Robot's Y location on grid: 11
x Length: 5
y Length: 25
~~~~~~~~~~~~#~~~~~~~~~~~~
~~~~~~~~~~~~R#~~~~~~~~~~~~
##########~ ##############
#                         #
###########################
```

Figure 11: Screen shot of the terminal window corresponding to the screen shot below. Having found the next unexplored cell the robot starts to travel and map upwards.

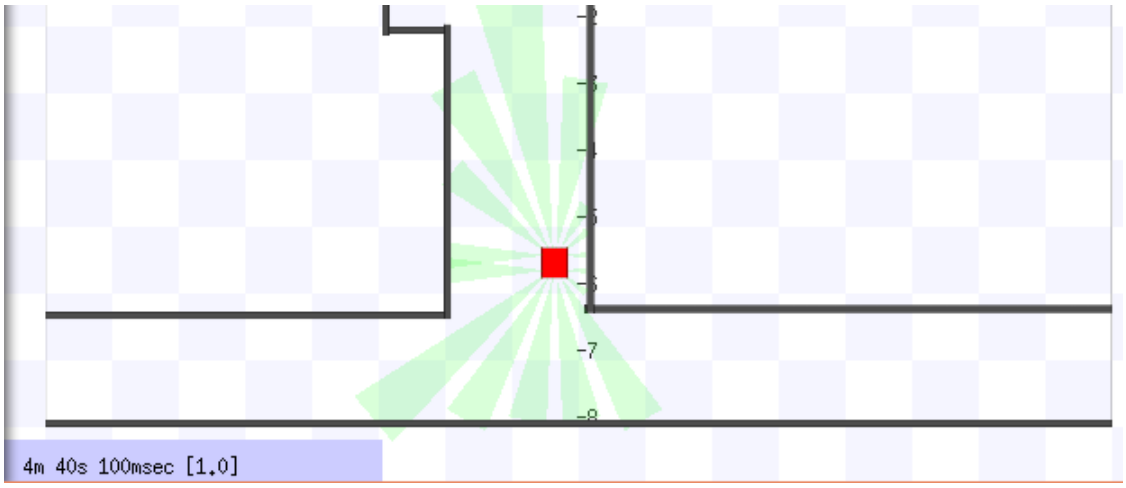

Figure 12: Screen shot of the Stage window corresponding to the screen shot above.

Figure 13: Screen shot of the terminal window corresponding to the screen shot below. Having reached the top the robot turns left as it is the only neighbour cell unexplored.

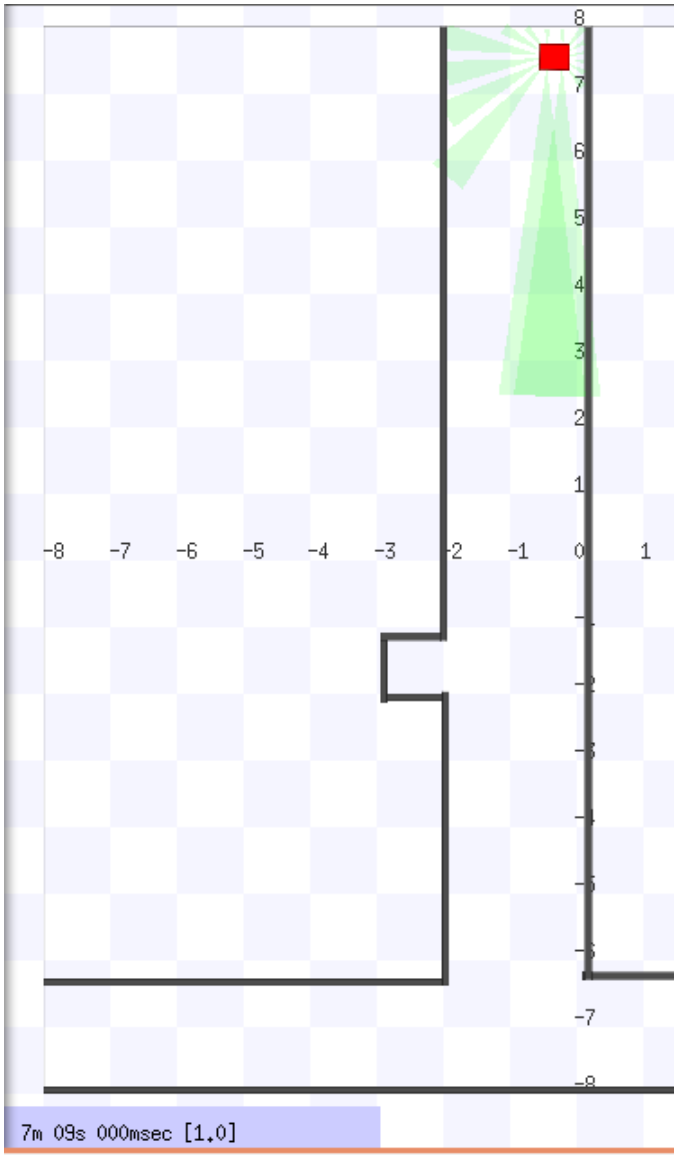

Figure 14: Screen shot of the Stage window corresponding to the screen shot above.

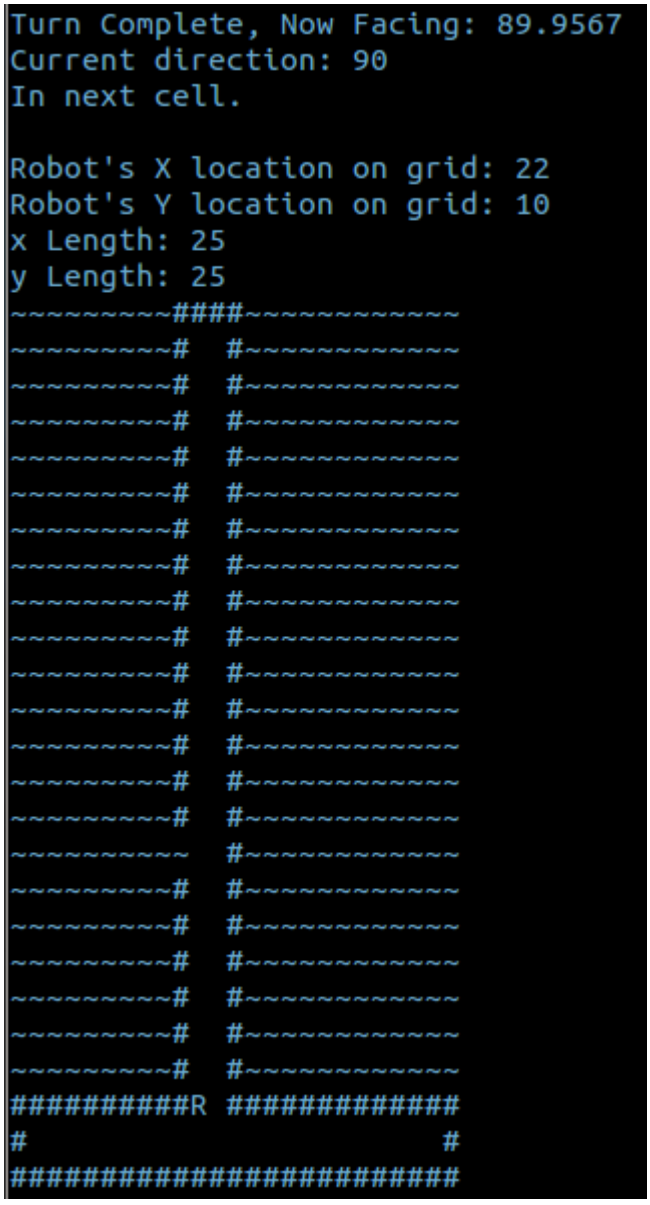

Figure 15: Screen shot of the terminal window corresponding to the screen shot below. Having scanned the left side all the way back to the pre-explored area, the robot turns to travel back along its path to the next unexplored cell.

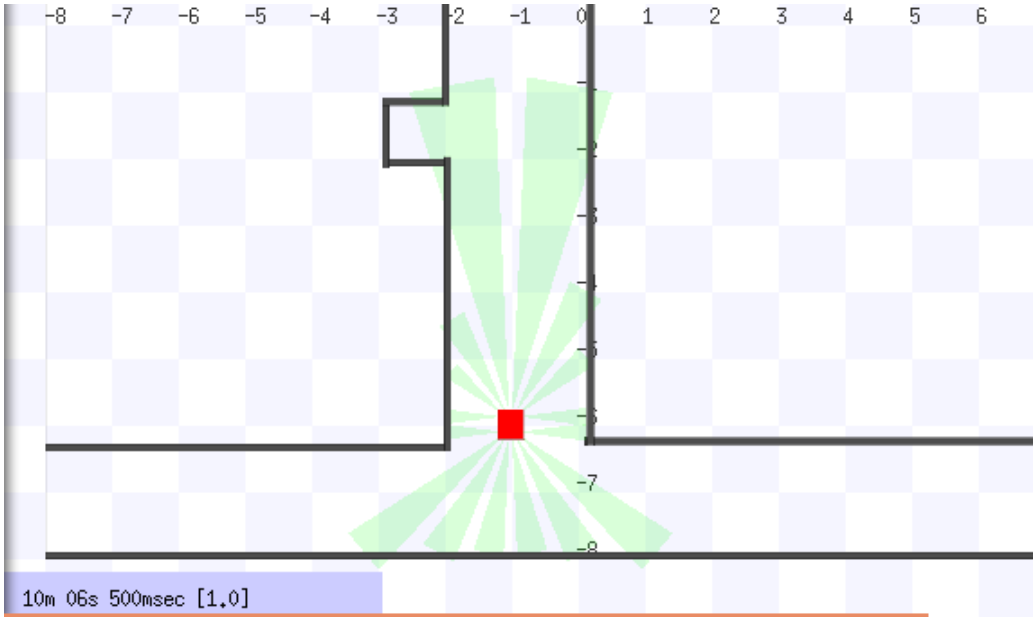

Figure 16: Screen shot of the Stage window corresponding to the screen shot above.

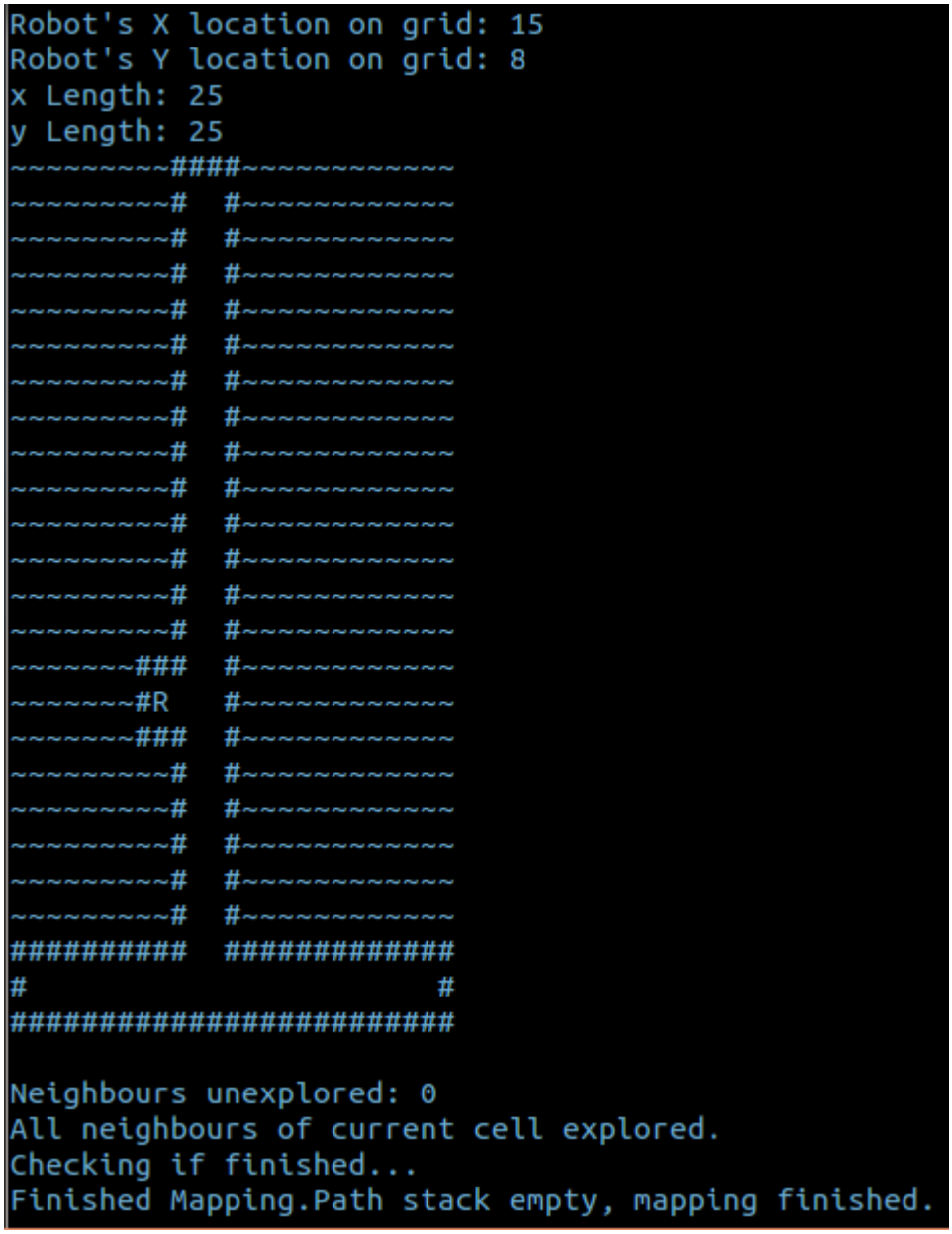

Figure 17: Screen shot of the terminal window corresponding to the screen shot below. Having reached a dead end, the robot evaluates if it has any cells left to explore. As it evaluates there are none, the mapping is declared finished.

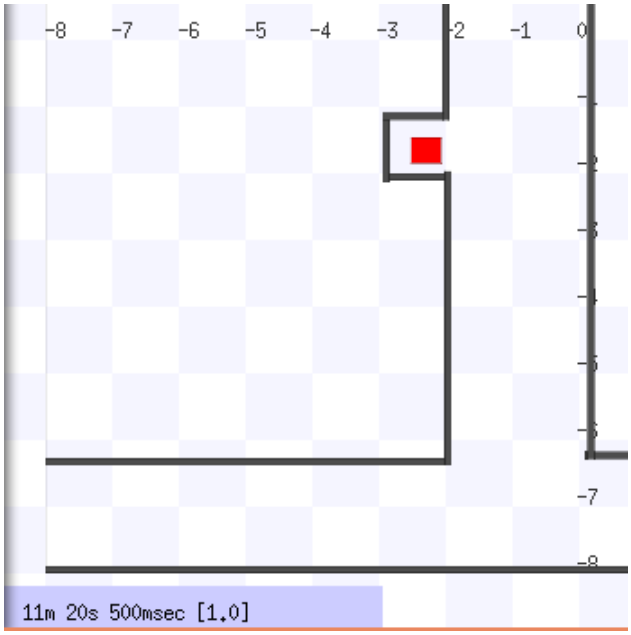

Figure 18: Screen shot of the Stage window corresponding to the screen shot above.

2.2 Real world trial runs

3 Discussion of trial run results

At the start it was expected that a lot of the simulated trial runs would finish prematurely due to a robot crash. This was in fact the case. Most of the time the robot would crash when turning and usually it was at the bottom of the simulated world where the robot tries to explore the upper section. Once it occurred that a 60cm by 60cm cell space was not sufficient to give the robot space to turn (explained in the evaluation section below) and this was amended to 70cm by 70cm cells, the robot managed to avoid getting too close to the walls so did not have problems when turning and was then able to complete a full successful map of the simulated area. This same result was replicated again when the function implemented to check that the robot had finished mapping needed testing.

In terms of real world results,

4 Evaluation of work done

4.1 Positive parts

The mapping algorithm implemented ensures that all explorable cells are explored due to the nature of a depth-first-search. The accuracy of the mapping is also very good as shown in the simulated trial run. As the mapping is very accurate due to only scanning a maximum of 8 cells around the robot at any given time (to reduce errors with sonar spread at longer ranges), there is no need to apply a threshold to the values. This helps improves efficiency as there is one less task to run.

The robot program implemented the way it is would work very well in a square world where all shapes are square-like. This is because the robot turns to the angles 0, 90, 180 and 270 degrees and no others. The path plotting algorithm is quite simple and as it uses an A* search, the path required to get the robot to its hiding location should be found quite fast without too many extra nodes being expanded. Also because the robot can keep track of a path along which it has travelled and return along, the path found by the A* algorithm is simply loaded into the robot for it to follow to get to its destination.

The robots localisation algorithm is also simple but enables the robot to localise as soon as possible and also notify the user that localisation is unsuccessful as soon as possible. This is great as a robot that has limited battery power wants to avoid draining power where possible moving around.

4.2 Negative parts

The main negative part about the robot program implemented is that the robot is very deliberative and is constrained to working in a square environment. As the robot only turns to the 0, 90, 180, 270, the robot is very limited in moving around environments. The robot is also constrained in moving forwards not backwards and at set distances, this again limits the robots capabilities with adapting to different environments. All of these problems exist mainly because of the deliberative nature of of the program implemented.

4.3 Hardest parts

The hardest part to implement was the mapping algorithm as it is a complex task to accomplish. Deciding how many and which sensors to use was a problem as well as figuring/estimating the angles and error bounds for each sensor. Also, regarding the sensors, the range at which the values returned would be considered accurate was an aspect of the project that took some thought. It was decided to just use ranges that would enable one cell in each direction to be scanned reliably including the diagonal/corner cells (8 cells around the robot in total). Detecting corners was a big problem that needed a lot of trial and error to get reasonably working.

With regards to problems encountered with using a real robot, the main problem was the robots believing they had turned to the specified angle wanted when in reality they had not. This needed a lot of trial of error with changing the PGAIN value using when calculating the turn rate for the robot. Even at that point each turn would not produce the same result so the robot would easily be able to produce different results with each run, sometimes resulting in a crash.

4.4 What would be done differently

An attempt to implement a reactive robot would be made to enable the robot to adaptive to a changing environment more effectively and avoid crashing better. If a reactive system was implemented, it may be a lot more complex (reason for not using this approach) but would allow the robot to have more freedom and deal with obstacles appearing right next

to it (such as an object moving into its path etc). It would also be easier for the robot to avoid being too close to walls (or at least correct to get a minimum distance from the walls) and thus VOID crashing when it turns. However, giving the robot more freedom with movement would make it more difficult to map as it would be harder to determine where the robot would be located on the grid so this may still not be a real viable option.

Another place where things would be done differently would be in the testing area. Testing the robot in different simulated environments would help to produce a robot that could function better in various kinds of environments (be more versatile). Also the same goes for real world testing/trials.