TDT4136 Introduction to Artificial Intelligence
Assignment 3

Marius Lervik

October 2017

Part 1

1-1

The source code for this part is found under the part1 folder. I implemented the A* algorithm in go. I used a struct for every node that is shown below. I created a 2d array containing all nodes in the board. I used the manhatten distance for estimating the cost to the goal node.

```
type node struct {
        y int
        x int
        g int
        f int
        weight int
        open     bool
        closed   bool
        solution bool // Is this part of best path?
        block    bool

        parent *node
}
```

The main loop in my A* implementation is shown on the next page. I used two sets open and closed that contains pointers to nodes in the 2d array that contains all nodes. As can be seen from my implementation i take out the node with the lowest f value at every iteration from open. I check if it is the solution, if not i add it to the closed set. I find all the neighbours and add the ones not in open to open. Then i update the estimate f for every neighbor and set the parent pointer to the current node. At the end of the loop i sort the nodes in open in ascencing values of f. The weights of all nodes was set to 1 in this part. My program takes the board as command-line argument. So for it to solve board-1-1 i have to type: go run part1.go board-1-1.txt . I created the function drawImage() in order to visualize the result of the algorithm.

```
for len(open) != 0 {
                current := open[0]
                fmt.Println("Current", *current, "Estimate of dist: ", current.f)
                if current.x == stopnode.x && current.y == stopnode.y { // done
                        return true
                }
                open = open[1:] // pops open
                current.open = false
                current.closed = true
                closed = append(closed, current)

                neighbors := findNeighbors(board, height, width, current)
                for _, node := range neighbors {
                        if nodeInSet(node, closed) {
                                tempg := current.g + 1
                                if tempg < node.g {
                                        fmt.Println("Found a shorter path to a closed node")
                                }
                                continue
                        }
                        if !nodeInSet(node, open) {
                                open = append(open, node)
                                node.open = true
                        }
                        tempg := current.g + node.weight
                        if tempg >= node.g { // did not find a better path
                                continue
                        }
                        // This path is best
                        node.parent = current
                        node.g = tempg
                        node.f = calculateF(node, stopnode)
                }
                sort.Sort(nodes(open))
}
```

1-2

The result is presented below. The squares with a pink square in the upper left corner are squares that are in the closed set when the algorithm terminates.

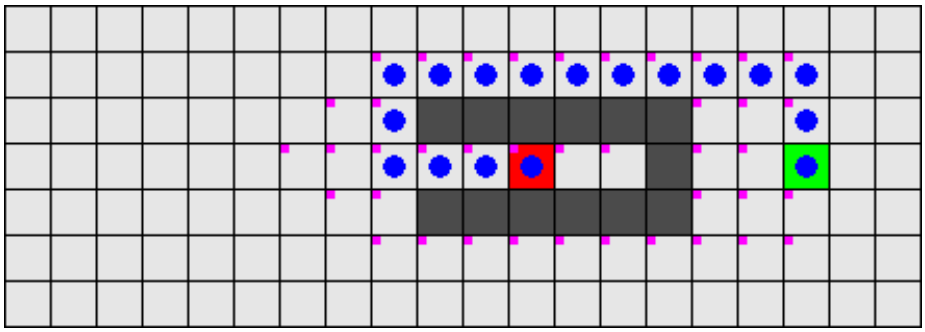

Figure 1: Board 1-1 solution

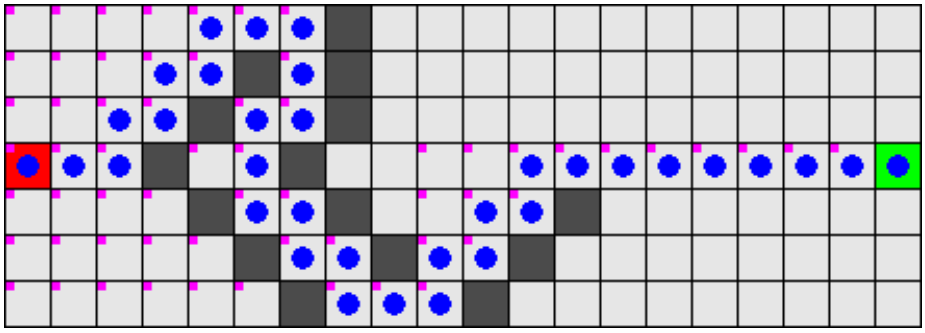

Figure 2: Board 1-2 solution

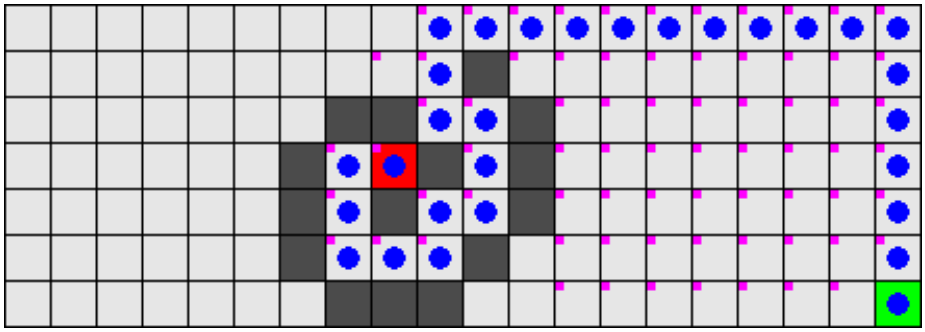

Figure 3: Board 1-3 solution

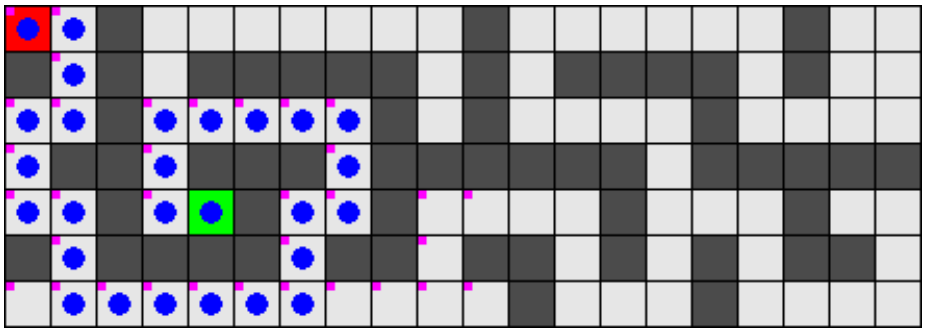

Figure 4: Board 1-4 solution

Part 2

2-1

Only small modifications was needed for this part. I just had to check for different cell types when creating the 2d array of nodes. I also had to make small changes to the drawImage() function in order to draw the different colors. The code is in the part2.go file in the part2 folder.

2-2

The results are shown below. The cells with a yellow square in the top left corner are nodes that are in the open set. The cells with a pink square are the nodes in the closed set.

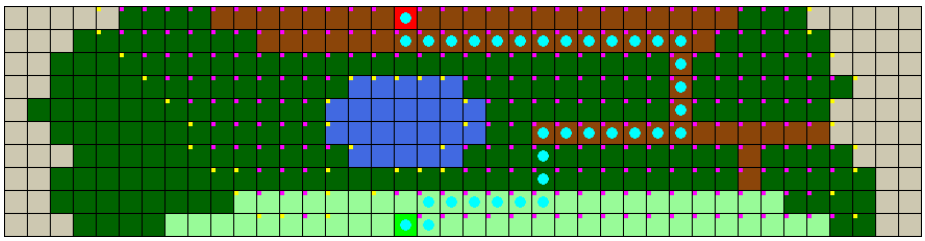

Figure 5: Board 2-1 solution

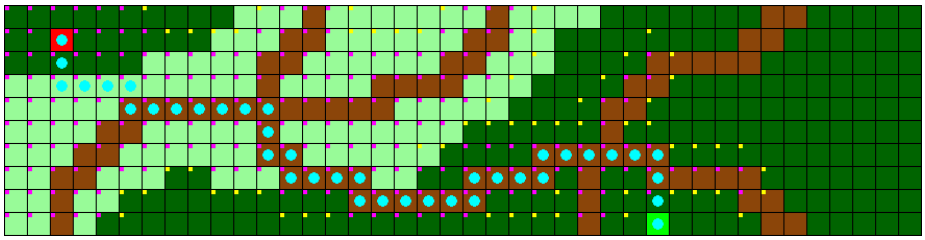

Figure 6: Board 2-2 solution

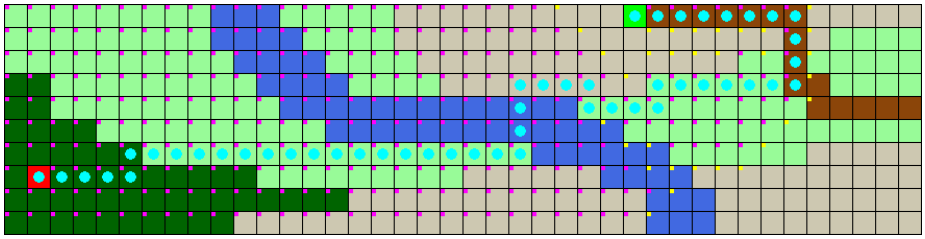

Figure 7: Board 2-3 solution

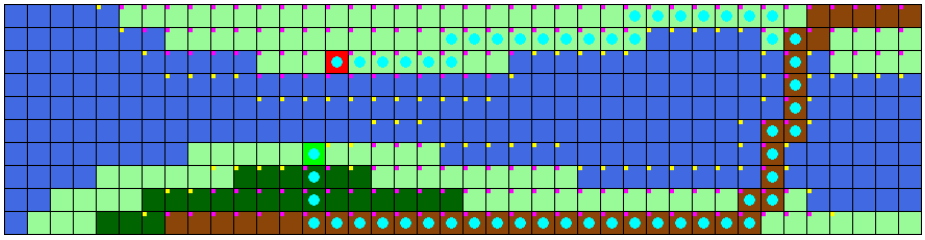

Figure 8: Board 2-4 solution

Part 3

3-1

Implementing dijkstra and bfs was very simple after implementing A*. Dijkstra is just A* without the heuristics. My implementation of dijkstra and bfs also stops when we have reached the goal node. The implementations can be seen in the file part3.go in the part3 folder. For the bfs algorithm i just appended

and popped the open set to get a FIFO queue. The algorithm loops for both bfs and dijkstra are shown below.

Dijkstra:

```
for len(open) != 0 {
                current := open[0]
                fmt.Println("Current", *current)
                if current.x == stopnode.x && current.y == stopnode.y { // done
                        return true
                }
                open = open[1:] // pops open
                current.open = false
                current.closed = true
                closed = append(closed, current)

                neighbors := findNeighbors(board, height, width, current)
                for _, node := range neighbors {
                        if nodeInSet(node, closed) {
                                continue // done with this node
                        }
                        if !nodeInSet(node, open) {
                                open = append(open, node)
                                node.open = true
                        }
                        tempg := current.g + node.weight
                        if tempg >= node.g { // did not find a better path
                                continue
                        }
                        // This path is best
                        node.parent = current
                        node.g = tempg
                        node.f = tempg
                }
                sort.Sort(nodes(open))
}
```

BFS:

```
for len(open) != 0 {
                current := open[0]
                open = open[1:] // pops open
                fmt.Println("Current", *current)
                if current.x == stopnode.x && current.y == stopnode.y { // done
                        return true
                }
                current.open = false
                current.closed = true
                closed = append(closed, current)

                neighbors := findNeighbors(board, height, width, current)
                for _, node := range neighbors {
                        if !nodeInSet(node, open) && !nodeInSet(node, closed) {
                                open = append(open, node)
                                node.open = true
                                node.parent = current
                        }

                }
}
```

The visualizations are again produced from the drawImage() function. This version of my program takes in two command-line arguments. One for the board, and one for the algorithm to use. So if i want to use dijkstra on board2-2 i have to type: go run part3.go board-2-2.txt dijkstra

3-2

The solutions for board 1-2 are shown below

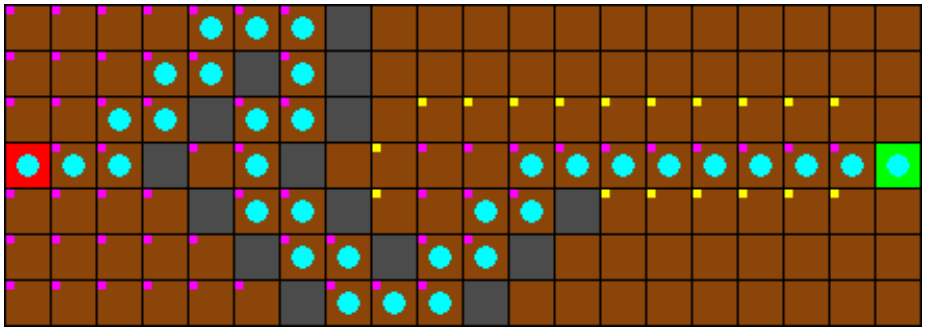

Figure 9: Board 1-2 solution A*

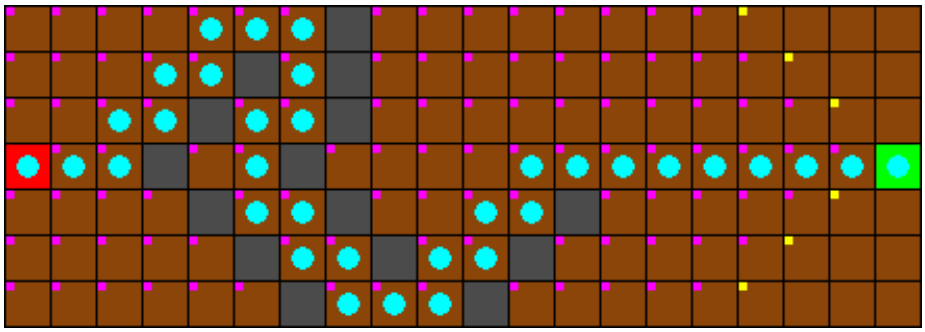

Figure 10: Board 1-2 solution Dijkstra

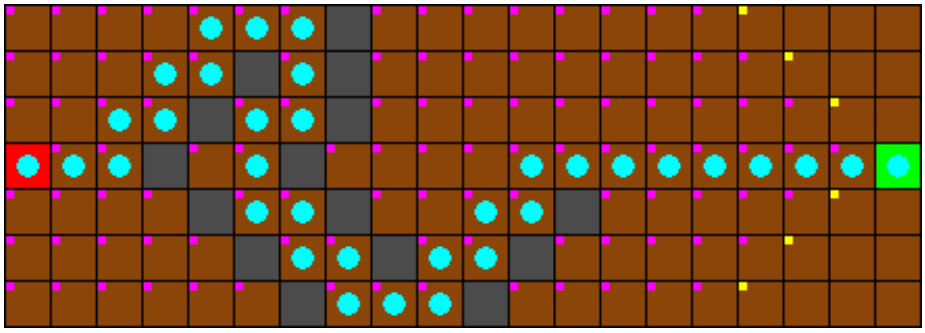

Figure 11: Board 1-2 solution BFS

The solutions for board 2-2 are shown below

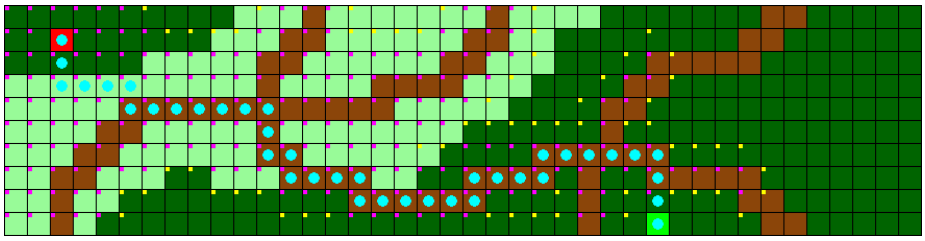

Figure 12: Board 2-2 solution A*

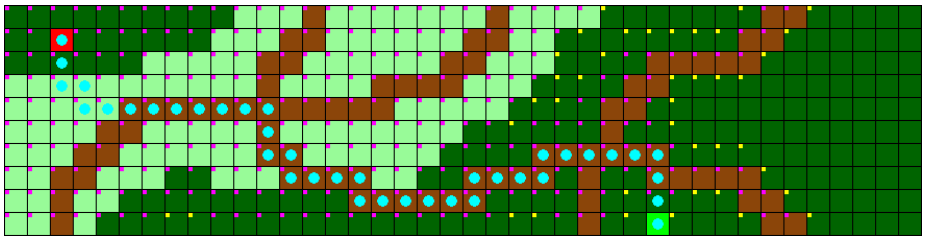

Figure 13: Board 2-2 solution Dijkstra

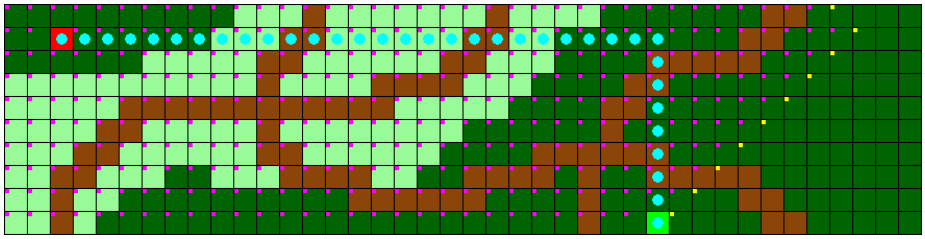

Figure 14: Board 2-2 solution BFS

3-3

For board 1-2 we see that Dijkstra and bfs are identical. That is to be expected when the weighting on all nodes are the same. Tha A* algorithm performs better and does not explore as many nodes as BFS and dijkstra. However they all find the same path. When we introduce weights to the nodes BFS fails to find the shortest path as can bee seen in the visualization from the previous section. Both A* and Difjkstra manages to find the shortest path as was expected. The difference here is the number of nodes in the open and closed set. We see that dijkstra will explore more nodes compared to A*.