

# Introduction to Artificial Intelligence

## Assignment 3

Adrian Opheim

October 16, 2017

### Contents

<b>1</b>	<b>Part 1: Grids with obstacles</b>	<b>3</b>
1.1	The Node class . . . . .	3
1.2	A_star . . . . .	3
1.2.1	manhattanDist() . . . . .	5
1.2.2	readFromTxt() . . . . .	5
1.2.3	generateBoard() . . . . .	6
1.2.4	colorPixel() . . . . .	6
1.2.5	convertToNodes() . . . . .	6
1.2.6	generateAllSucc() . . . . .	7
1.2.7	getSolution() . . . . .	8
1.2.8	colorSolution() . . . . .	8
1.3	The main function . . . . .	8
1.4	Visualization of solutions . . . . .	9
1.4.1	Board-1-1 . . . . .	9
1.4.2	Board-1-2 . . . . .	10
1.4.3	Board-1-3 . . . . .	10
1.4.4	Board-1-4 . . . . .	11
<b>2</b>	<b>Part 2: Grids with different cell costs</b>	<b>11</b>
2.1	The Node class . . . . .	11
2.2	A_star . . . . .	12
2.2.1	manhattanDist() . . . . .	14
2.2.2	readFromTxt() . . . . .	14
2.2.3	generateBoard() . . . . .	14
2.2.4	colorPixel() . . . . .	15
2.2.5	convertToNodes() . . . . .	15
2.2.6	generateAllSucc() . . . . .	16
2.2.7	getSolution() . . . . .	17
2.2.8	colorSolution() . . . . .	17
2.3	The main function . . . . .	18
2.4	Visualization of solutions . . . . .	20
2.4.1	Board-2-1 . . . . .	20
2.4.2	Board-2-2 . . . . .	20
2.4.3	Board-2-3 . . . . .	21
2.4.4	Board-2-4 . . . . .	21

<b>3</b>	<b>Part 3: Comparison with BFS and Dijkstra's Algorithm</b>	<b>21</b>
3.1	Dijkstra() . . . . .	21
3.2	BFS() . . . . .	23
3.3	Comparisons of results for Board 1.4 . . . . .	26
3.4	Comparison of results for Board 2.1 . . . . .	26
<b>4</b>	<b>References</b>	<b>27</b>

# 1 Part 1: Grids with obstacles

The A star algorithm follows the pseudocode taken from the PDF shown in References, with changes to fit our problem

The implementation uses classes in Python to define a **Node** class with the desired node variables. Visualizations of the algorithm are made using the Python Imaging Library (Pillow)  
The implementation is given below, with print statements making it easier to follow the algorithm's flow.

## 1.1 The Node class

```
1 class Node(object):
2     def __init__(self, row, col, value):
3         self.row = row
4         self.col = col
5         self.value = value
6         self.g = 1000
7         self.h = 1000
8         self.f = 2000
9         self.parent = self
10        self.start = False
11        self.end = False
12        self.free = False
13        self.wall = False
14
15        def gFunc(self, start, node): #Calculates the distance from the root
16            to the node (Manhattan distance)
17            return manhattanDist(start, node)
18
19        def hFunc(self, node, end):
20            return manhattanDist(node, end)
21
22        def fFunc(self):
23            self.f = self.g + self.h
24
25        def printNode(self):
26            print(
27                "\n\nPosition: \t[", self.row, " ", self.col, "]",
28                "\nStart: \t", self.start,
29                "\nEnd: \t", self.end,
30                "\ng: \t", self.g,
31                "\nh: \t", self.h,
32                "\nf: \t", self.f,
33                "\nFree: \t", self.free,
34                "\nWall: \t", self.wall,
35                "\nValue: \t", self.value,
36                "\nParent: \t[", (self.parent).row, " ", (self.parent).col,
37                "]", end=""
38            )
```

## 1.2 A\_star

```

1 def A_star(nodes, start, end, img_object):
2     # Initializing the closed and open lists, containing elements
      already evaluated.
3     open = []
4     closed = []
5
6     # Initializing the start node:
7     start.g = 0
8     start.h = start.hFunc(start, end)
9     start.f = start.g + start.h
10    start.parent = start
11
12    print("Start: ")
13    start.printNode()
14    print("End: ")
15    end.printNode()
16
17
18    # Appending the start node to the set of opened nodes
19    open.append(start)
20
21    success = False
22    while((len(open) > 0) and (success == False)):          # while the
      open list is not empty
23        print("\n*****\nOpen
          contains: \n")
24        for el in open:
25            el.printNode()
26        print("\n*****\n")
27
28        q = open.pop(0)          # popping the first element of the open
          array, the one with the lowest f value.
29        print("\n-----CURRENT NODE-----\n")
30        q.printNode()
31
32        img_object = colorPixel(False, img_object, q, (255, 255, 102))
33        succ = generateAllSucc(q, nodes)          # Generating all valid
          neighbouring elements of q
34
35        print("\n*****")
36        print("\nValid neighbours of [", q.row, ", ", q.col, "]: ")
37        for S in succ:
38            S.printNode()
39        print("\n*****\n")
40
41        for S in succ:
42            #print("In succ")
43            #S.printNode()
44            if (S.end == True):          # If the neighbouring element is
          the goal, end the while loop
45                print("\n\nEnd node is found!")
46                success = True
47                lastNode = S

```

```

48         S.parent = q
49         break
50     tmp_S_g = q.g + manhattanDist(q, S) # Updating the neighbour
        's g value
51     tmp_S_h = S.hFunc(S, end)
52     tmp_S_f = tmp_S_g + tmp_S_h
53
54     # If the node is already in the closed or open list , but
        with lower f value , skip adding it that neighbour
55     if ((S in open) and (S.f <= tmp_S_f)):
56         print("\nS in open with <= f")
57         continue
58     if ((S in closed) and (S.f <= tmp_S_f)):
59         print("\nS in closed with <= f")
60         continue
61
62     else: #Otherwise, add the neighbour to the open list , and
        set its f, g and h values
63         print("\n\nAdding node")
64         S.g = tmp_S_g
65         S.h = tmp_S_h
66         S.f = tmp_S_f
67         S.parent = q
68         S.printNode()
69         open.append(S) # Adding S to the open list.
70         open.sort(key=lambda Node: Node.f) # TODO: check if
            correct. sorting the opened list after f value.
71     closed.append(q) # adding q to the closed list
72
73     # Outside while loop
74     solution = []
75     solution = getSolution(lastNode, solution)
76     # Adding the start and end node, as the getSolution does not add
        them
77     solution.append(start)
78     solution.insert(0, end)
79
80     return [img_object, solution]

```

### 1.2.1 manhattanDist()

```

1 def manhattanDist(start, end):
2     xDist = abs(end.col - start.col)
3     yDist = abs(end.row - start.row)
4     #print(" xDist: ", xDist, ", yDist: ", yDist)
5     return xDist + yDist

```

### 1.2.2 readFromTxt()

```

1 def readFromTxt(filePath):
2     file = open(filePath, "r")
3     lines = file.readlines()

```

```

4     #print( lines )
5     return lines

```

### 1.2.3 generateBoard()

```

1 def generateBoard( board , fileName ):
2     img = Image.new( 'RGB' , ( len( board[0] ) , len( board ) ) , "white" )
3         # Creates image object in the size of the board.
4     pixels = img.load()           # creating a pixel map
5
6     for line in range(0, len( board )):
7         for char in range(0, len( board[0] )):
8             if ( board[ line ][ char ] == "." ):
9                 pixels[ char , line ] = (192,192,192)      # open pixels
10                    appear grey
11
12             if ( board[ line ][ char ] == "#" ):
13                 pixels[ char , line ] = (0,0,0)           # border pixels appear
14                    black
15
16             if ( board[ line ][ char ] == "A" ):
17                 pixels[ char , line ] = (0,0,204)         # start pixel appear
18                    blue
19
20             if ( board[ line ][ char ] == "B" ):
21                 pixels[ char , line ] = (255,0,0)         # end pixel appear
22                    red
23
24     img.save( fileName , "PNG" )
25     return img

```

### 1.2.4 colorPixel()

```

1 def colorPixel( fileName , img , node , color ):
2
3
4     pixels = img.load()
5     pixels[ node.col , node.row ] = color           # Marking the path yellow
6     if ( fileName != False ):           # if image file is to be created
7         img.save( fileName , "PNG" )
8
9     return img

```

### 1.2.5 convertToNodes()

```

1 def convertToNodes( board ):
2     # Converting the board with characters to nodes
3     nodeList = []           #nodeList: List in list indexed nodeList[row][
4                             col]
5     for i in range(0, len( board )):
6         new = []

```

```

6         for j in range(0, len(board[0])):
7             new.append(Node(i,j, board[i][j]))
8         nodeList.append(new)
9
10    # Filling in necessary information for the node class
11    for i in range(0, len(board)):
12        for j in range(0, len(board[0])):
13            if (nodeList[i][j].value == "A"):
14                nodeList[i][j].start = True
15                nodeList[i][j].free = True
16                start = nodeList[i][j]
17
18            if (nodeList[i][j].value == "B"):
19                nodeList[i][j].end = True
20                nodeList[i][j].free = True
21                end = nodeList[i][j]
22
23            if (nodeList[i][j].value == "."):
24                nodeList[i][j].free = True
25
26            if (nodeList[i][j].value == "#"):
27                nodeList[i][j].wall = True
28
29    return [nodeList, start, end]

```

### 1.2.6 generateAllSucc()

```

1 def generateAllSucc(node, nodeList):
2     colStart = 0
3     colEnd = len(nodeList[0])
4     rowStart = 0
5     rowEnd = len(nodeList)
6     #print("colEnd: ", colEnd, "rowEnd: ", rowEnd)
7
8     iNorth = node.row - 1
9     iSouth = node.row + 1
10    iEast = node.col + 1
11    iWest = node.col - 1
12
13
14    neighbourList = []
15    # If indexes are outside the board, set them as "invalid"
16    if (iNorth < rowStart or (nodeList[iNorth][node.col].free == False))
17        : # if over the array or not free
18        iNorth = None
19    else:
20        neighbourList.append(nodeList[iNorth][node.col])
21
22    if (iEast > colEnd - 1 or (nodeList[node.row][iEast].free == False))
23        :
24        iEast = None
25    else:
26        neighbourList.append(nodeList[node.row][iEast])

```

```

25
26     if ((iSouth > rowEnd - 1) or (nodeList[iSouth][node.col].free ==
27         False)):
28         iSouth = None
29     else:
30         neighbourList.append(nodeList[iSouth][node.col])
31
32     if ((iWest < colStart) or (nodeList[node.row][iWest].free == False)):
33         :
34         iWest = None
35     else:
36         neighbourList.append(nodeList[node.row][iWest])
37
38     #print("neighbourList: ", neighbourList)
39     return neighbourList

```

### 1.2.7 getSolution()

```

1 def getSolution(lastNode, solution):
2     if ((lastNode.parent).start != True):
3         par = lastNode.parent
4         solution.append(par)
5         getSolution(lastNode.parent, solution)
6
7     #solution.append(lastNode.parent)
8     return solution

```

### 1.2.8 colorSolution()

```

1 def colorSolution(solution, img, fileName):
2     for el in solution:
3         colorPixel(fileName, img, el, (0, 204, 0))
4
5     # Coloring the start pixel:
6     colorPixel(fileName, img, solution[0], (255, 0, 0))
7     #Coloring the end pixel:
8     colorPixel(fileName, img, solution[len(solution) - 1], (0, 0, 204))

```

## 1.3 The main function

```

1 def main():
2
3     board_1.1 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
4         NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
5         A3_A_star\\boards\\boards\\board-1-1.txt")
6     board_1.2 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
7         NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
8         A3_A_star\\boards\\boards\\board-1-2.txt")
9     board_1.3 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
10        NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
11        A3_A_star\\boards\\boards\\board-1-3.txt")

```



```

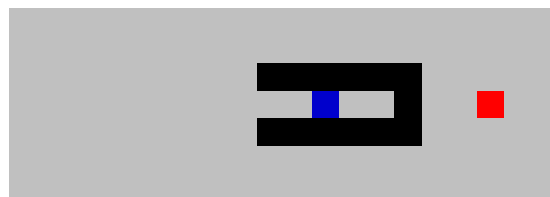
6      board_1_4 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
      NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
      A3_A_star\\boards\\boards\\board-1-4.txt")
7
8      img_1_1 = generateBoard(board_1_1, "board_1_1.png")
9      img_1_2 = generateBoard(board_1_2, "board_1_2.png")
10     img_1_3 = generateBoard(board_1_3, "board_1_3.png")
11     img_1_4 = generateBoard(board_1_4, "board_1_4.png")
12
13
14
15     [nodes, start, end] = convertToNodes(board_1_1)
16     [img_1_1, sol_1_1] = A_star(nodes, start, end, img_1_1)
17     colorSolution(sol_1_1, img_1_1, "board_1_1_path.png")
18
19     [nodes, start, end] = convertToNodes(board_1_2)
20     [img_1_2, sol_1_2] = A_star(nodes, start, end, img_1_2)
21     colorSolution(sol_1_2, img_1_2, "board_1_2_path.png")
22
23     [nodes, start, end] = convertToNodes(board_1_3)
24     [img_1_3, sol_1_3] = A_star(nodes, start, end, img_1_3)
25     colorSolution(sol_1_3, img_1_3, "board_1_3_path.png")
26
27     [nodes, start, end] = convertToNodes(board_1_4)
28     [img_1_4, sol_1_4] = A_star(nodes, start, end, img_1_4)
29     colorSolution(sol_1_4, img_1_4, "board_1_4_path.png")

```

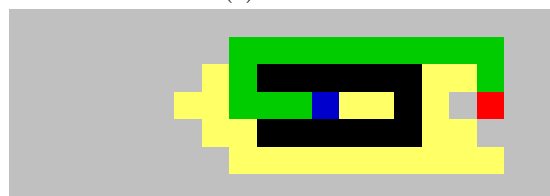
## 1.4 Visualization of solutions

The code above renders the following solutions

### 1.4.1 Board-1-1



(a) Board 1-1



(b) Solution to board 1-1

Figure 1: The original board and found solution using the implemented A\* algorithm.

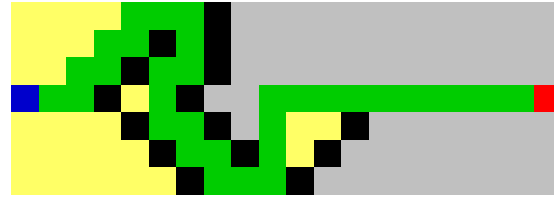
#### Color codes:

Blue: Start node Red: End node Green: Shortest path found using A\* Yellow: Nodes that have been explored Black: Walls (impassable)

#### 1.4.2 Board-1-2



(a) Board 1-2



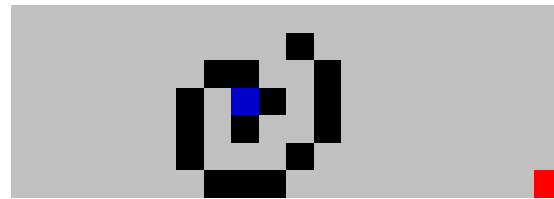
(b) Solution to board 1-2

Figure 2: The original board and found solution using the implemented A\* algorithm.

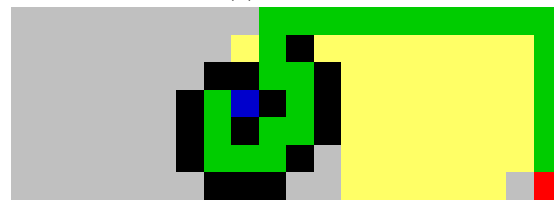
**Color codes:**

Blue: Start node Red: End node Green: Shortest path found using A\* Yellow: Nodes that have been explored Black: Walls (impassable)

#### 1.4.3 Board-1-3



(a) Board 1-3



(b) Solution to board 1-3

Figure 3: The original board and found solution using the implemented A\* algorithm.

**Color codes:**

Blue: Start node Red: End node Green: Shortest path found using A\* Yellow: Nodes that have been explored Black: Walls (impassable)

#### 1.4.4 Board-1-4

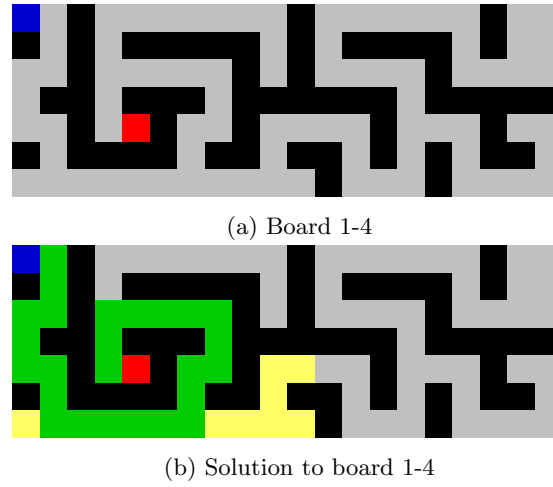


Figure 4: The original board and found solution using the implemented A\* algorithm.

##### Color codes:

Blue: Start node Red: End node Green: Shortest path found using A\* Yellow: Nodes that have been explored Black: Walls (impassable)

## 2 Part 2: Grids with different cell costs

The grids will now have a cost for each pixel, representing the cost of moving through it, corresponding to Figure 5






	CHAR.	DESCRIPTION	COST
	w	Water	100
	m	Mountains	50
	f	Forests	10
	g	Grasslands	5
	r	Roads	1

Figure 5: The cost for moving through pixels

Only small changes are done on the Node class and the A\* algorithm. The Node class has a new attribute `cost`, which holds the cost of each cell. The A\* algorithm is modified to take advantage of this: The  $f$  value of each node is now  $f = g + h + cost$ , where the cost of neighbour nodes is updated by adding its parents cost, making the cost "cumulative".

### 2.1 The Node class

```

1 class Node(object):
2     def __init__(self, row, col, value):
3         self.row = row
4         self.col = col
5         self.value = value
6         self.g = 1000

```

```

7         self.h = 1000
8         self.f = 2000
9         self.parent = self
10        self.start = False
11        self.end = False
12        self.free = False
13        self.wall = False
14        self.cost = 2000
15
16    def gFunc(self, start, node): #Calculates the distance from the root
17        to the node (Manhattan distance)
18        return manhattanDist(start, node)
19
20    def hFunc(self, node, end):
21        return manhattanDist(node, end)
22
23    def fFunc(self):
24        self.f = self.g + self.h
25
26    def printNode(self):
27        print(
28            "\n\nPosition: \t[", self.row, " ", self.col, "]",
29            "\nStart: \t", self.start,
30            "\nEnd: \t", self.end,
31            "\ng: \t", self.g,
32            "\nh: \t", self.h,
33            "\nCost: \t", self.cost,
34            "\nf: \t", self.f,
35            "\nFree: \t", self.free,
36            "\nWall: \t", self.wall,
37            "\nValue: \t", self.value,
38            "\nParent: \t[", (self.parent).row, " ", (self.parent).col,
39            "]", end=""
40        )

```

## 2.2 A\_star

```

1 def A_star(nodes, start, end, img_object):
2     # Initializing the closed and open lists, containing elements
3     # already evaluated.
4     open = []
5     closed = []
6
7     # Initializing the start node:
8     start.g = 0
9     start.h = start.hFunc(start, end)
10    start.f = start.g + start.h
11    start.parent = start
12
13    #print(" Start: ")
14    #start.printNode()
15    #print(" End: ")
16    #end.printNode()

```

```

16
17
18 # Appending the start node to the set of opened nodes
19 open.append(start)
20
21 success = False
22 while((len(open) > 0) and (success == False)):          # while the
    open list is not empty
23     #print("\n*****\n")
        nOpen contains: \n")
24     #for el in open:
        #el.printNode()
25     #print("\n*****\n")
26
27     q = open.pop(0)          # popping the first element of the open
        array, the one with the lowest f value.
28     #print("\n-----CURRENT NODE-----\n")
        #q.printNode()
29
30     #img_object = colorPixel(False, img_object, q, (255, 255, 102))
31     succ = generateAllSucc(q, nodes)          # Generating all valid
        neighbouring elements of q
32
33     #print("\n*****\n")
34     #print("\nValid neighbours of [", q.row, ", ", q.col, "]: ")
35     #for S in succ:
        #S.printNode()
36     #print("\n*****\n")
37
38     for S in succ:
39         #print("In succ")
40         #S.printNode()
41         if (S.end == True):          # If the neighbouring element is
            the goal, end the while loop
42             print("\n\nEnd node is found!")
43             success = True
44             lastNode = S
45             S.parent = q
46             break
47
48         tmp_S_g = q.g + manhattanDist(q, S) # Updating the neighbour
            's g value
49         tmp_S_h = S.hFunc(S, end)
50         tmp_cost = q.cost + S.cost
51         tmp_S_f = tmp_S_g + tmp_S_h + S.cost
52
53         # If the node is already in the closed or open list, but
            with lower f value, skip adding it that neighbour
54         if ((S in open) and (S.f <= tmp_S_f)):
55             #print("\nS in open with <= f")
56             continue
57         if ((S in closed) and (S.f <= tmp_S_f)):
58             #print("\nS in closed with <= f")
59             continue
60
61

```

```

62
63         else: #Otherwise, add the neighbour to the open list, and
              set its f, g and h values
64             #print("\n\nAdding node")
65             S.g = tmp_S_g
66             S.h = tmp_S_h
67             S.cost = tmp_cost
68             S.f = tmp_S_f
69             S.parent = q
70             #S.printNode()
71             open.append(S)          # Adding S to the open list.
72             open.sort(key=lambda Node: Node.f)      # TODO: check if
              correct. sorting the opened list after f value.
73         closed.append(q) # adding q to the closed list
74
75     # Outside while loop
76     solution = []
77     solution = getSolution(lastNode, solution)
78     # Adding the start and end node, as the getSolution does not add
       them
79     solution.append(start)
80     solution.insert(0, end)
81
82     return [img-object, solution]

```

### 2.2.1 manhattanDist()

```

1 def manhattanDist(start, end):
2     xDist = abs(end.col - start.col)
3     yDist = abs(end.row - start.row)
4     #print(" xDist: ", xDist, ", yDist: ", yDist)
5     return xDist + yDist

```

### 2.2.2 readFromTxt()

```

1 def readFromTxt(filePath):
2     file = open(filePath, "r")
3     lines = file.readlines()
4     #print(lines)
5     return lines

```

### 2.2.3 generateBoard()

```

1 def generateBoard(board, fileName):
2     img = Image.new('RGB', (len(board[0]), len(board)), "white")
       # Creates image object in the size of the board.
3     pixels = img.load()          # creating a pixel map
4
5     for line in range(0, len(board)):
6         for char in range(0, len(board[0])):
7             if (board[line][char] == "."):

```

```

8         pixels[char, line] = (192,192,192)      # open pixels
          appear grey
9
10        if (board[line][char] == "#"):
11            pixels[char, line] = (0,0,0)        # border pixels appear
              black
12
13        if (board[line][char] == "A"):
14            pixels[char, line] = (255,255,0)     # start pixel
              appear blue
15
16        if (board[line][char] == "B"):
17            pixels[char, line] = (255,0,0)       # end pixel appear
              red
18
19        if (board[line][char] == "w"):
20            pixels[char, line] = (0,0,255)      # Water
21
22        if (board[line][char] == "m"):
23            pixels[char, line] = (128,128,128)   # mountains
24
25        if (board[line][char] == "f"):
26            pixels[char, line] = (0,102,0)      # forest
27
28        if (board[line][char] == "g"):
29            pixels[char, line] = (102,204,0)     # grass
30
31        if (board[line][char] == "r"):
32            pixels[char, line] = (153,76,0)      # roads
33
34    img.save(fileName, "PNG")
35    return img

```

#### 2.2.4 colorPixel()

```

1 def colorPixel(fileName, img, node, color):
2
3
4     pixels = img.load()
5     pixels[node.col, node.row] = color          # Marking the path yellow
6     if (fileName != False):                    # if image file is to be created
7         img.save(fileName, "PNG")
8
9     return img

```

#### 2.2.5 convertToNodes()

```

1 def convertToNodes(board):
2     # Converting the board with characters to nodes
3     nodeList = []                             #nodeList: List in list indexed nodeList[row][
        col]
4     for i in range(0, len(board)):

```

```

5         new = []
6         for j in range(0, len(board[0])):
7             new.append(Node(i,j, board[i][j]))
8         nodeList.append(new)
9
10    # Filling in necessary information for the node class
11    for i in range(0, len(board)):
12        for j in range(0, len(board[0])):
13            if (nodeList[i][j].value == "A"):
14                nodeList[i][j].start = True
15                nodeList[i][j].free = True
16                start = nodeList[i][j]
17
18            if (nodeList[i][j].value == "B"):
19                nodeList[i][j].end = True
20                nodeList[i][j].free = True
21                end = nodeList[i][j]
22
23            if (nodeList[i][j].value == "."):
24                nodeList[i][j].free = True
25
26            if (nodeList[i][j].value == "#"):
27                nodeList[i][j].wall = True
28
29            if (nodeList[i][j].value == "w"):
30                nodeList[i][j].cost = 100
31                nodeList[i][j].free = True
32
33            if (nodeList[i][j].value == "m"):
34                nodeList[i][j].cost = 50
35                nodeList[i][j].free = True
36
37            if (nodeList[i][j].value == "f"):
38                nodeList[i][j].cost = 10
39                nodeList[i][j].free = True
40
41            if (nodeList[i][j].value == "g"):
42                nodeList[i][j].cost = 5
43                nodeList[i][j].free = True
44
45            if (nodeList[i][j].value == "r"):
46                nodeList[i][j].cost = 1
47                nodeList[i][j].free = True
48
49    return [nodeList, start, end]

```

### 2.2.6 generateAllSucc()

```

1 def generateAllSucc(node, nodeList):
2     colStart = 0
3     colEnd = len(nodeList[0])
4     rowStart = 0
5     rowEnd = len(nodeList)

```



```

6      #print(" colEnd: ", colEnd, "rowEnd: ", rowEnd)
7
8      iNorth = node.row - 1
9      iSouth = node.row + 1
10     iEast = node.col + 1
11     iWest = node.col - 1
12
13
14     neighbourList = []
15     # If indexes are outside the board, set them as "invalid"
16     if (iNorth < rowStart or (nodeList[iNorth][node.col].free == False))
17         : # if over the array or not free
18         iNorth = None
19     else:
20         neighbourList.append(nodeList[iNorth][node.col])
21
22     if (iEast > colEnd - 1 or (nodeList[node.row][iEast].free == False))
23         :
24         iEast = None
25     else:
26         neighbourList.append(nodeList[node.row][iEast])
27
28     if ((iSouth > rowEnd - 1) or (nodeList[iSouth][node.col].free ==
29         False)):
30         iSouth = None
31     else:
32         neighbourList.append(nodeList[iSouth][node.col])
33
34     if ((iWest < colStart) or (nodeList[node.row][iWest].free == False))
35         :
36         iWest = None
37     else:
38         neighbourList.append(nodeList[node.row][iWest])
39
40     #print(" neighbourList: ", neighbourList)
41     return neighbourList

```

### 2.2.7 getSolution()

```

1  def getSolution(lastNode, solution):
2      if ((lastNode.parent).start != True):
3          par = lastNode.parent
4          solution.append(par)
5          getSolution(lastNode.parent, solution)
6
7      #solution.append(lastNode.parent)
8      return solution

```

### 2.2.8 colorSolution()

```

1  def colorSolution(solution, img, fileName):
2      for el in solution:

```

```

3         colorPixel(fileName, img, el, (204, 0, 204))      # colouring
           the path
4
5     # Coloring the start pixel:
6     colorPixel(fileName, img, solution[len(solution) - 1], (255, 255, 0)
7         )
8
9     #Coloring the end pixel:
10    colorPixel(fileName, img, solution[0], (255, 0, 0) )

```

## 2.3 The main function

```

1 def main():
2
3     #For Part 1:
4     board_1_1 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NINU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-1-1.txt")
5     board_1_2 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NINU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-1-2.txt")
6     board_1_3 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NINU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-1-3.txt")
7     board_1_4 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NINU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-1-4.txt")
8
9     # For part 2:
10    board_2_1 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NINU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-2-1.txt")
11    board_2_2 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NINU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-2-2.txt")
12    board_2_3 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NINU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-2-3.txt")
13    board_2_4 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NINU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-2-4.txt")
14
15    img_1_1 = generateBoard(board_1_1, "board_1_1.png")
16    img_1_2 = generateBoard(board_1_2, "board_1_2.png")
17    img_1_3 = generateBoard(board_1_3, "board_1_3.png")
18    img_1_4 = generateBoard(board_1_4, "board_1_4.png")
19
20    img_2_1 = generateBoard(board_2_1, "board_2_1.png")
21    img_2_2 = generateBoard(board_2_2, "board_2_2.png")
22    img_2_3 = generateBoard(board_2_3, "board_2_3.png")
23    img_2_4 = generateBoard(board_2_4, "board_2_4.png")
24
25

```

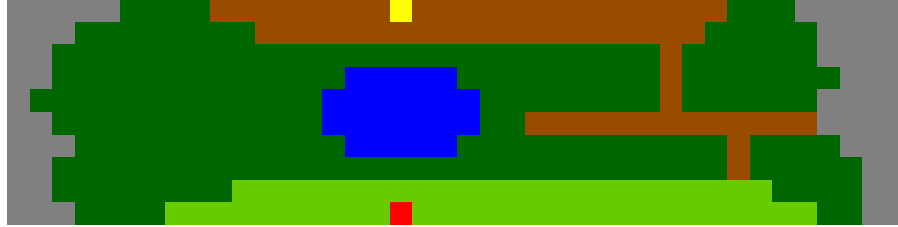
```

26 [nodes, start, end] = convertToNodes(board_1_1)
27 [img_1_1, sol_1_1] = A_star(nodes, start, end, img_1_1)
28 colorSolution(sol_1_1, img_1_1, "board_1_1_path.png")
29
30 [nodes, start, end] = convertToNodes(board_1_2)
31 [img_1_2, sol_1_2] = A_star(nodes, start, end, img_1_2)
32 colorSolution(sol_1_2, img_1_2, "board_1_2_path.png")
33
34 [nodes, start, end] = convertToNodes(board_1_3)
35 [img_1_3, sol_1_3] = A_star(nodes, start, end, img_1_3)
36 colorSolution(sol_1_3, img_1_3, "board_1_3_path.png")
37
38 [nodes, start, end] = convertToNodes(board_1_4)
39 [img_1_4, sol_1_4] = A_star(nodes, start, end, img_1_4)
40 colorSolution(sol_1_4, img_1_4, "board_1_4_path.png")
41
42
43
44 [nodes, start, end] = convertToNodes(board_2_1)
45 [img_2_1, sol_2_1] = A_star(nodes, start, end, img_2_1)
46 colorSolution(sol_2_1, img_2_1, "board_2_1_path.png")
47
48 [nodes, start, end] = convertToNodes(board_2_2)
49 [img_2_2, sol_2_2] = A_star(nodes, start, end, img_2_2)
50 colorSolution(sol_2_2, img_2_2, "board_2_2_path.png")
51
52 [nodes, start, end] = convertToNodes(board_2_3)
53 [img_2_3, sol_2_3] = A_star(nodes, start, end, img_2_3)
54 colorSolution(sol_2_3, img_2_3, "board_2_3_path.png")
55
56 [nodes, start, end] = convertToNodes(board_2_4)
57 [img_2_4, sol_2_4] = A_star(nodes, start, end, img_2_4)
58 colorSolution(sol_2_4, img_2_4, "board_2_4_path.png")

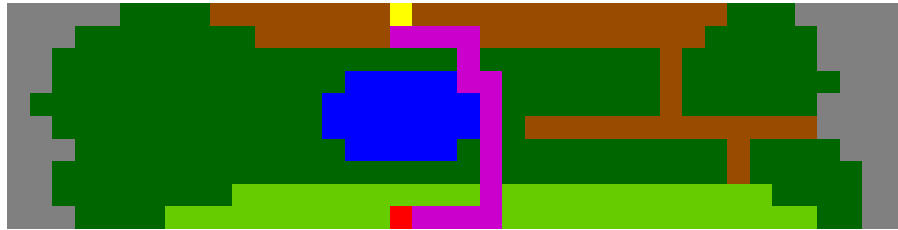
```

## 2.4 Visualization of solutions

### 2.4.1 Board-2-1



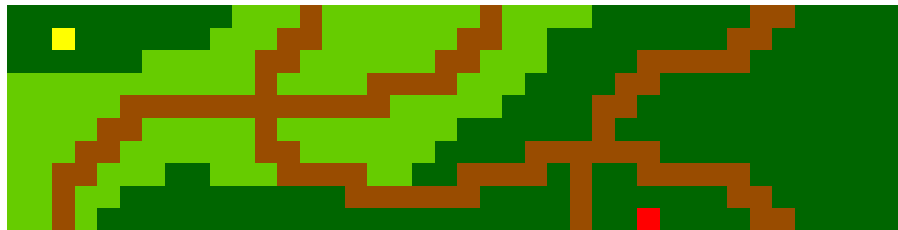
(a) Board 2-1



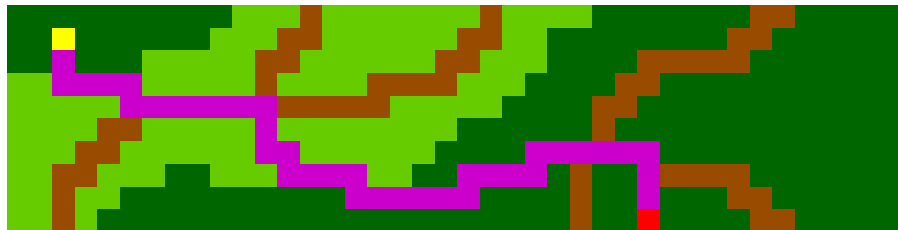
(b) Solution to board 2-1

Figure 6: The original board and found solution (purple) using the implemented A\* algorithm.

### 2.4.2 Board-2-2



(a) Board 2-2



(b) Solution to board 2-2

Figure 7: The original board and found solution (purple) using the implemented A\* algorithm.

### 2.4.3 Board-2-3



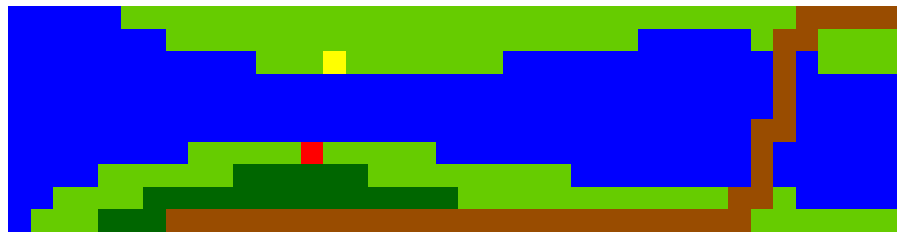
(a) Board 2-3



(b) Solution to board 2-3

Figure 8: The original board and found solution (purple) using the implemented A\* algorithm.

### 2.4.4 Board-2-4



(a) Board 2-4



(b) Solution to board 2-4

Figure 9: The original board and found solution (purple) using the implemented A\* algorithm.

## 3 Part 3: Comparison with BFS and Dijkstra's Algorithm

The implemented Dijkstra and Breadth-First Search (BFS) algorithms are shown below.

### 3.1 Dijkstra()

```
1 def dijkstra(nodes, start, end, img_object):  
2     # Initializing the closed and open lists, containing elements  
   already evaluated.  
3     open = []
```

```

4     closed = []
5
6     # Initializing the start node:
7     start.g = start.cost
8     #start.h = start.hFunc(start, end)
9     start.f = start.g
10    start.parent = start
11
12    #print(" Start: ")
13    #start.printNode()
14    #print(" End: ")
15    #end.printNode()
16
17
18    # Appending the start node to the set of opened nodes
19    open.append(start)
20
21    success = False
22    while((len(open) > 0) and (success == False)):      # while the
        open list is not empty
23        #print("\n*****\n")
24        #for el in open:
25            #el.printNode()
26        #print("\n*****\n")
27
28        q = open.pop(0)      # popping the first element of the open
            array, the one with the lowest g value.
29        #print("\n-----CURRENT NODE-----\n")
30        #q.printNode()
31
32        #img_object = colorPixel(False, img_object, q, (255, 255, 102))
33        succ = generateAllSucc(q, nodes)      # Generating all valid
            neighbouring elements of q
34
35        #print("\n*****\n")
36        #print("\nValid neighbours of [", q.row, ", ", q.col, "]: ")
37        #for S in succ:
38            #S.printNode()
39        #print("\n*****\n")
40
41        for S in succ:
42            #print("In succ")
43            #S.printNode()
44            if (S.end == True):      # If the neighbouring element is
                the goal, end the while loop
45                print("\n\nEnd node is found!")
46                success = True
47                lastNode = S
48                S.parent = q
49                break
50

```

```

51         tmp_S_g = q.g + S.cost          # Updating the neighbour's g
           value
52         #tmp_S_f = q.f + S.cost
53
54         # If the node is already in the closed or open list , but
           with lower f value , skip adding it that neighbour
55         if ((S in open) and (S.f <= tmp_S_g)):
56             #print("\nS in open with <= f")
57             continue
58         if ((S in closed) and (S.f <= tmp_S_g)):
59             #print("\nS in closed with <= f")
60             continue
61
62         else: #Otherwise, add the neighbour to the open list , and
           set its f, g and h values
63             print("\n\nAdding node")
64             S.g = tmp_S_g
65             #S.h = tmp_S_h
66             #S.cost = tmp_cost
67             #S.f = tmp_S_f
68             S.parent = q
69             S.printNode()
70             open.append(S)          # Adding S to the open list.
71             open.sort(key=lambda Node: Node.g)          # Dijkstra: Sort
           the opened list by the g value
72             colorPixel(False, img_object, S, openedColor)
73
74         closed.append(q) # adding q to the closed list
75         colorPixel(False, img_object, S, closedColor)
76
77     # Outside while loop
78     solution = []
79     solution = getSolution(lastNode, solution)
80     # Adding the start and end node, as the getSolution does not add
       them
81     solution.append(start)
82     solution.insert(0, end)
83
84     print("Dijkstra solution is, from end to start: \n")
85     for el in solution:
86         print("[", el.row, " ", el.col, "] ")
87
88     return [img_object, solution]

```

### 3.2 BFS()

```

1 def BFS(nodes, start, end, img_object):
2     # Initializing the closed and open lists , containing elements
       already evaluated.
3     open = []
4     closed = []
5
6     # Initializing the start node:

```

```

7      #start.g = 0
8      #start.h = start.hFunc(start , end)
9      start.f = 0
10     start.parent = start
11
12     #print(" Start: ")
13     #start.printNode()
14     #print(" End: ")
15     #end.printNode()
16
17
18     # Appending the start node to the set of opened nodes
19     open.append(start)
20
21     success = False
22     while((len(open) > 0) and (success == False)):          # while the
        open list is not empty
23         #print("\n*****\n")
        nOpen contains: \n")
24         #for el in open:
25             #el.printNode()
26         #print("\n*****\n")
27
28         q = open.pop(0)          # popping the first element of the open
        array, the one with the lowest f value.
29         #print("\n-----CURRENT NODE-----\n")
30         #q.printNode()
31
32         #img_object = colorPixel(False, img_object, q, (255, 255, 102))
33         succ = generateAllSucc(q, nodes)          # Generating all valid
        neighbouring elements of q
34
35         #print("\n*****\n")
36         #print("\nValid neighbours of [", q.row, ", ", q.col, "]: ")
37         #for S in succ:
38             #S.printNode()
39         #print("\n*****\n")
40
41         for S in succ:
42             #print("In succ")
43             #S.printNode()
44             if (S.end == True):          # If the neighbouring element is
        the goal, end the while loop
45                 print("\n\nEnd node is found!")
46                 success = True
47                 lastNode = S
48                 S.parent = q
49                 break
50             #tmp_S_g = q.g + manhattanDist(q, S) # Updating the
        neighbour's g value
51             #tmp_S_h = S.hFunc(S, end)
52             #tmp_cost = q.cost + S.cost
53             tmp_S_f = q.f + S.cost

```



```

54
55         # If the node is already in the closed or open list , but
56         # with lower f value , skip adding it that neighbour
57         if ((S in open) and (S.f <= tmp_S.f)):
58             #print("\nS in open with <= f")
59             continue
60         if ((S in closed) and (S.f <= tmp_S.f)):
61             #print("\nS in closed with <= f")
62             continue
63
64         else: #Otherwise, add the neighbour to the open list , and
65             # set its f, g and h values
66             #print("\n\nAdding node")
67             #S.g = tmp_S.g
68             #S.h = tmp_S.h
69             #S.cost = tmp_cost
70             S.f = tmp_S.f
71             S.parent = q
72             #S.printNode()
73             open.insert(0, S)          # Adding S to the open list at
74             # first position. FIFO, used in BFS
75             colorPixel(False, img_object, S, openedColor)
76
77         closed.append(q) # adding q to the closed list
78         colorPixel(False, img_object, q, closedColor)
79
80     # Outside while loop
81     solution = []
82     solution = getSolution(lastNode, solution)
83     # Adding the start and end node, as the getSolution does not add
84     # them
85     solution.append(start)
86     solution.insert(0, end)
87
88     print("Breadth-first solution is, from end to start: \n")
89     for el in solution:
90         print("[" , el.row, " " , el.col, "]" )
91
92     return [img_object, solution]

```

### 3.3 Comparisons of results for Board 1.4

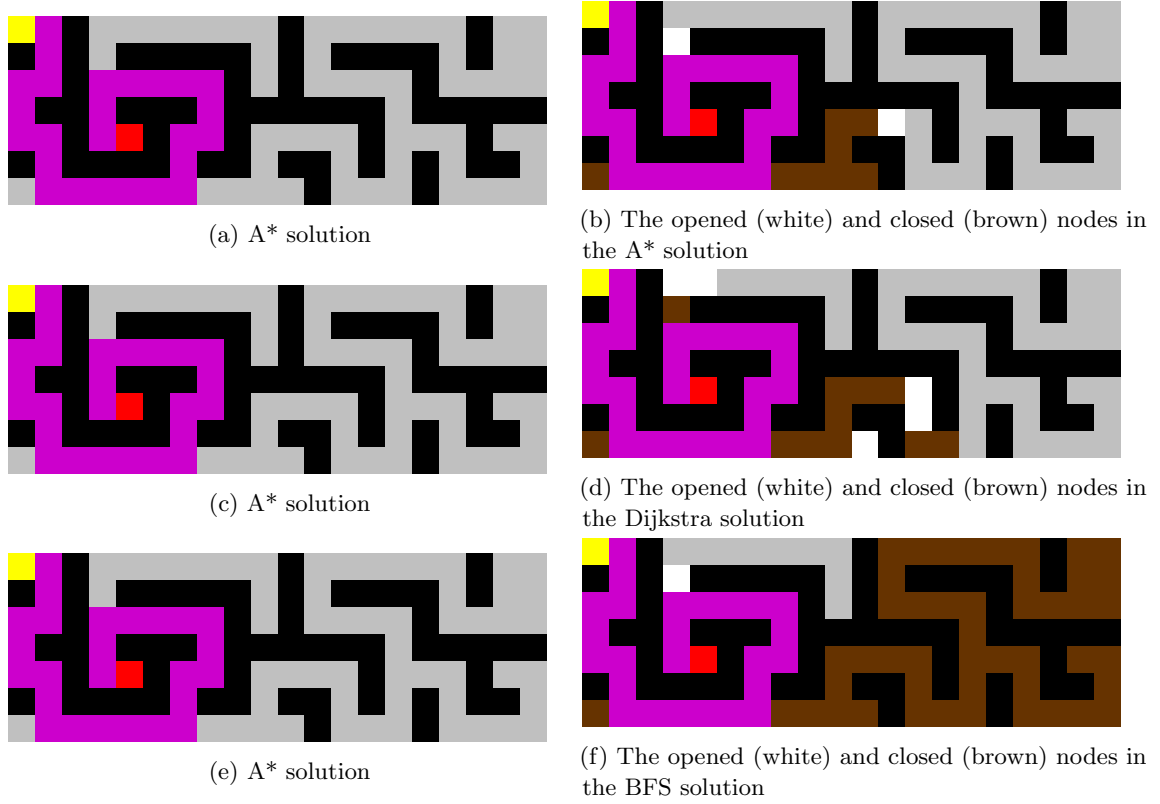


Figure 10: The found solutions compared with the opened (white) and closed (brown) nodes for Board 1-4

As Board 1-4 only has one possible solution, the different algorithms do not differ in the solutions found. However, there are differences in the number of opened or closed nodes. A\* and Dijkstra have more or less the same number of opened/closed nodes, while BFS has a huge amount, due to the implementation of how it visits neighbours, described below.

### 3.4 Comparison of results for Board 2.1

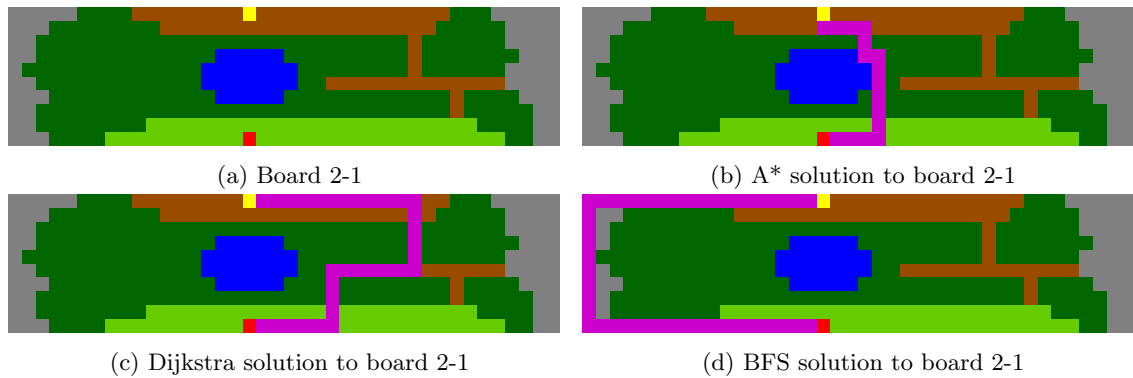


Figure 11: The original board and found solutions using A\* algorithm, Dijkstra and BFS. The purple pixels mark the found path.

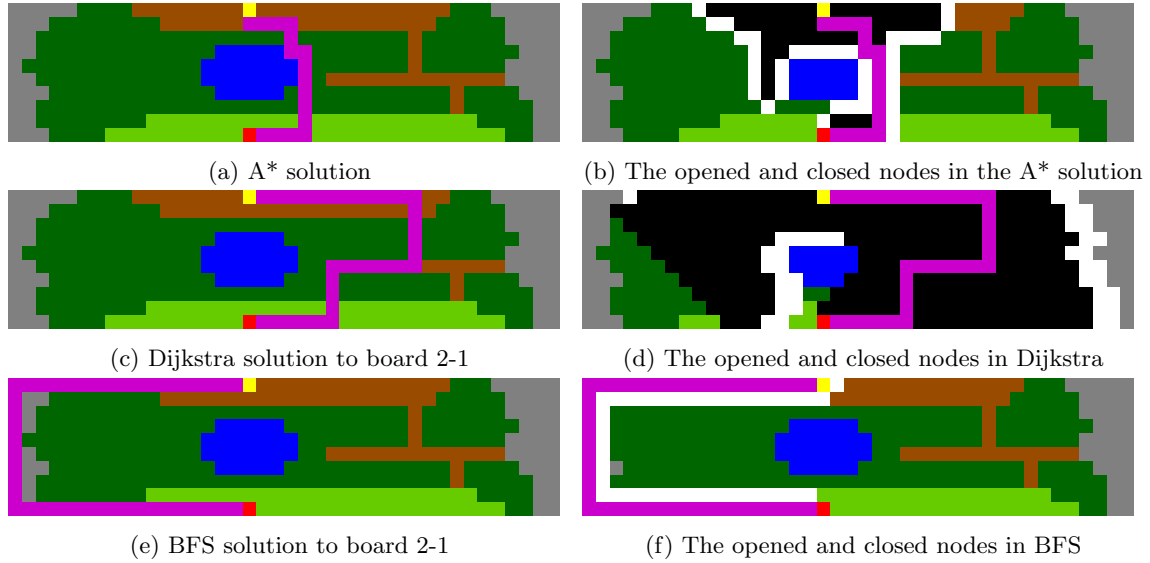


Figure 12: The found solutions compared with the opened (white) and closed (black) nodes for Board 2-1

As seen in Figure 11, the solutions found differ dramatically from algorithm to algorithm. For the A\* case, what seems like the optimal solution is found. One can see from the opened and closed nodes that the algorithm has propagated in what seems like a reasonable way: Excluding nodes in water because of their high value, but at the same time trying to find the shortest path among the nodes.

The Dijkstra solution has explored significantly more nodes than A\*, but also utilized the road network with lower cost. As Dijkstra explores all neighbouring nodes before selecting one, it has a significantly higher cost in terms of memory than A\*.

The BFS solution is obviously not the best in terms of total cost from start to end, but is rather "one of many solutions". The left turn of the solution is due to how the algorithm finds neighbours. It explores neighbours in order north, east, south, west, and then adds the nodes to the opened list, at the first position. When a new node is picked from the opened list, it is picked out of the First-In First-Out principle, meaning that the western node is picked first, because it was the last node to be placed at the top of the opened list. The BFS solution is the one that has opened/closed the least amount of nodes in order to get a solution, but as the solution is not the optimal one, it is not a desired algorithm to use in this case.

## 4 References

Pseudocode for the A-star algorithm:

<http://coecl1.ece.illinois.edu/ge423/lecturenotes/AstarHandOut.pdf>