Introduction to Artificial Intelligence
Assignment 3

Adrian Opheim

October 16, 2017

Contents

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

```python
class Node(object):
    def __init__(self, row, col, value):
        self.row = row
        self.col = col
        self.value = value
        self.g = 1000
        self.h = 1000
        self.f = 2000
        self.parent = self
        self.start = False
        self.end = False
        self.free = False
        self.wall = False

    def gFunc(self, start, node): #Calculates the distance from the root to the node (Manhattan distance)
        return manhattanDist(start, node)

    def hFunc(self, node, end):
        return manhattanDist(node, end)

    def fFunc(self):
        self.f = self.g + self.h

    def printNode(self):
        print(
            "\n\nPosition: \t[", self.row, " ", self.col, "]",
            "\nStart: \t", self.start,
            "\nEnd: \t", self.end,
            "\ng: \t", self.g,
            "\nh: \t", self.h,
            "\nf: \t", self.f,
            "\nFree: \t", self.free,
            "\nWall: \t", self.wall,
            "\nValue: \t", self.value,
            "\nParent: \t[", (self.parent).row, " ", (self.parent).col, "]", end=""
        )
```

1.2 A_star

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

    # Initializing the start node:
    start.g = 0
    start.h = start.hFunc(start, end)
    start.f = start.g + start.h
    start.parent = start

    print("Start: ")
    start.printNode()
    print("End: ")
    end.printNode()


    # Appending the start node to the set of opened nodes
    open.append(start)

    success = False
    while((len(open) > 0) and (success == False)):        # while the open list is not empty
        print("\n**************************************************\nOpen contains: \n")
        for el in open:
            el.printNode()
        print("\n**************************************************\n")

        q = open.pop(0)          # popping the first element of the open array, the one with the lowest f value.
        print("\n------------------------CURRENT NODE--------------------------------------\n")
        q.printNode()

        img_object = colorPixel(False, img_object, q, (255, 255, 102))
        succ = generateAllSucc(q, nodes)     # Generating all valid neighbouring elements of q

        print("\n*********************************************")
        print("\nValid neighbours of [", q.row, ", ", q.col, "]: ")
        for S in succ:
            S.printNode()
        print("\n*********************************************\n")

        for S in succ:
            #print("In succ")
            #S.printNode()
            if (S.end == True):          # If the neighbouring element is the goal, end the while loop
                print("\n\nEnd node is found!")
                success = True
                lastNode = S
```

```
                S.parent = q
                break
            tmp_S_g = q.g + manhattanDist(q, S) # Updating the neighbour's g value
            tmp_S_h = S.hFunc(S, end)
            tmp_S_f = tmp_S_g + tmp_S_h

            # If the node is already in the closed or open list, but with lower f value, skip adding it that neighbour
            if ((S in open) and (S.f <= tmp_S_f)):
                print("\nS in open with <= f")
                continue
            if ((S in closed) and (S.f <= tmp_S_f)):
                print("\nS in closed with <= f")
                continue

            else:  #Otherwise, add the neighbour to the open list, and set its f, g and h values
                print("\n\nAdding node")
                S.g = tmp_S_g
                S.h = tmp_S_h
                S.f = tmp_S_f
                S.parent = q
                S.printNode()
                open.append(S)      # Adding S to the open list.
                open.sort(key=lambda Node: Node.f)      # TODO: check if correct. sorting the opened list after f value.
        closed.append(q) # adding q to the closed list

    # Outside while loop
    solution = []
    solution = getSolution(lastNode, solution)
    # Adding the start and end node, as the getSolution does not add them
    solution.append(start)
    solution.insert(0, end)

    return [img_object, solution]
```

1.2.1 manhattanDist()

```
def manhattanDist(start, end):
    xDist = abs(end.col - start.col)
    yDist = abs(end.row - start.row)
    #print("xDist: ", xDist, ", yDist: ", yDist)
    return xDist + yDist
```

1.2.2 readFromTxt()

```
def readFromTxt(filePath):
    file = open(filePath, "r")
    lines = file.readlines()
```

```
    #print(lines)
    return lines
```

1.2.3 generateBoard()

```
def generateBoard(board, fileName):
    img = Image.new('RGB', (len(board[0]), len(board)), "white")  # Creates image object in the size of the board.
    pixels = img.load()            # creating a pixel map

    for line in range(0, len(board)):
        for char in range(0, len(board[0])):
            if (board[line][char] == "."):
                pixels[char, line] = (192,192,192)       # open pixels appear grey

            if (board[line][char] == "#"):
                pixels[char, line] = (0,0,0)       # border pixels appear black

            if (board[line][char] == "A"):
                pixels[char, line] = (0,0,204)       # start pixel appear blue

            if (board[line][char] == "B"):
                pixels[char, line] = (255,0,0)       # end pixel appear red

    img.save(fileName, "PNG")
    return img
```

1.2.4 colorPixel()

```
def colorPixel(fileName, img, node, color):



    pixels = img.load()
    pixels[node.col, node.row] = color          # Marking the path yellow
    if (fileName != False):      # if image file is to be created
        img.save(fileName, "PNG")

    return img
```

1.2.5 convertToNodes()

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

```
        for j in range(0, len(board[0])):
            new.append(Node(i,j, board[i][j]))
        nodeList.append(new)

    # Filling in necessery information for the node class
    for i in range(0, len(board)):
        for j in range(0, len(board[0])):
            if (nodeList[i][j].value == "A"):
                nodeList[i][j].start = True
                nodeList[i][j].free = True
                start = nodeList[i][j]

            if (nodeList[i][j].value == "B"):
                nodeList[i][j].end = True
                nodeList[i][j].free = True
                end = nodeList[i][j]

            if (nodeList[i][j].value == "."):
                nodeList[i][j].free = True

            if (nodeList[i][j].value == "#"):
                nodeList[i][j].wall = True

    return [nodeList, start, end]
```

1.2.6 generateAllSucc()

```
def generateAllSucc(node, nodeList):
    colStart = 0
    colEnd = len(nodeList[0])
    rowStart = 0
    rowEnd = len(nodeList)
    #print("colEnd: ", colEnd, "rowEnd: ", rowEnd)

    iNorth = node.row - 1
    iSouth = node.row + 1
    iEast = node.col + 1
    iWest = node.col - 1


    neighbourList = []
    # If indexes are outside the board, set them as "invalid"
    if (iNorth < rowStart or (nodeList[iNorth][node.col].free == False)): # if over the array or not free
        iNorth = None
    else:
        neighbourList.append(nodeList[iNorth][node.col])

    if (iEast > colEnd - 1 or (nodeList[node.row][iEast].free == False)):
        iEast = None
    else:
        neighbourList.append(nodeList[node.row][iEast])
```

```python

    if ((iSouth > rowEnd - 1) or (nodeList[iSouth][node.col].free ==
        False)):
        iSouth = None
    else:
        neighbourList.append(nodeList[iSouth][node.col])

    if ((iWest < colStart) or (nodeList[node.row][iWest].free == False))
        :
        iWest = None
    else:
        neighbourList.append(nodeList[node.row][iWest])

    #print("neighbourList: ", neighbourList)
    return neighbourList
```

1.2.7 getSolution()

```python
def getSolution(lastNode, solution):
    if ((lastNode.parent).start != True):
        par = lastNode.parent
        solution.append(par)
        getSolution(lastNode.parent, solution)

    #solution.append(lastNode.parent)
    return solution
```

1.2.8 colorSolution()

```python
def colorSolution(solution, img, fileName):
    for el in solution:
        colorPixel(fileName, img, el, (0, 204, 0))

    # Coloring the start pixel:
    colorPixel(fileName, img, solution[0], (255, 0, 0) )
    #Coloring the end pixel:
    colorPixel(fileName, img, solution[len(solution) - 1], (0, 0, 204))
```

1.3 The main function

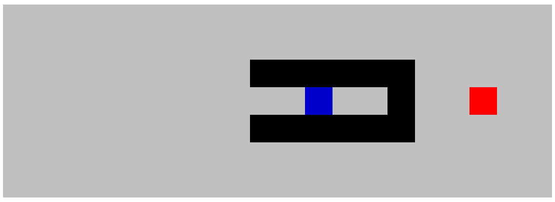
```python
def main():

    board_1_1 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-1-1.txt")
    board_1_2 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-1-2.txt")
    board_1_3 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-1-3.txt")
```

```
    board_1_4 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-1-4.txt")

    img_1_1 = generateBoard(board_1_1, "board_1_1.png")
    img_1_2 = generateBoard(board_1_2, "board_1_2.png")
    img_1_3 = generateBoard(board_1_3, "board_1_3.png")
    img_1_4 = generateBoard(board_1_4, "board_1_4.png")



    [nodes, start, end] = convertToNodes(board_1_1)
    [img_1_1, sol_1_1] = A_star(nodes, start, end, img_1_1)
    colorSolution(sol_1_1, img_1_1, "board_1_1_path.png")

    [nodes, start, end] = convertToNodes(board_1_2)
    [img_1_2, sol_1_2] = A_star(nodes, start, end, img_1_2)
    colorSolution(sol_1_2, img_1_2, "board_1_2_path.png")

    [nodes, start, end] = convertToNodes(board_1_3)
    [img_1_3, sol_1_3] = A_star(nodes, start, end, img_1_3)
    colorSolution(sol_1_3, img_1_3, "board_1_3_path.png")

    [nodes, start, end] = convertToNodes(board_1_4)
    [img_1_4, sol_1_4] = A_star(nodes, start, end, img_1_4)
    colorSolution(sol_1_4, img_1_4, "board_1_4_path.png")
```

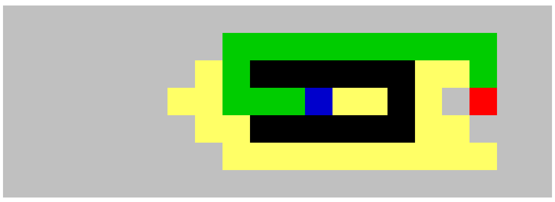
1.4 Visualization of solutions

The code above renders the following solutions

1.4.1 Board-1-1



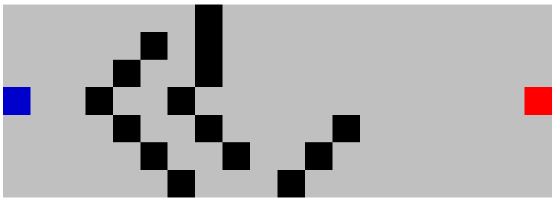
(a) Board 1-1



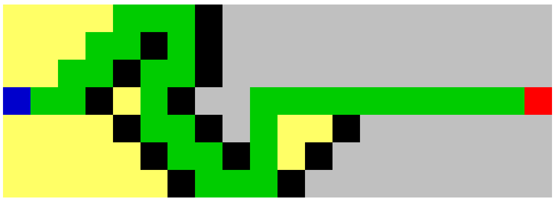
(b) Solution to board 1-1

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



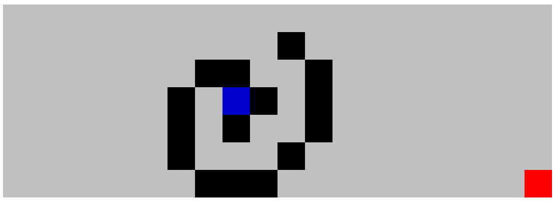
(a) Board 1-2



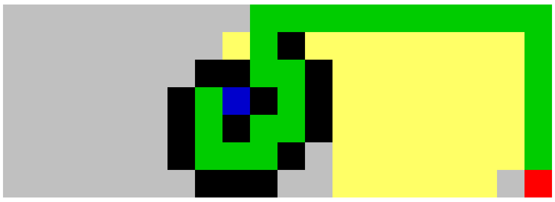
(b) Solution to board 1-2

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



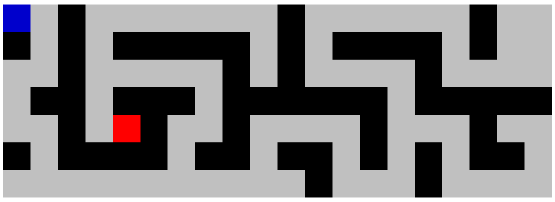
(a) Board 1-3



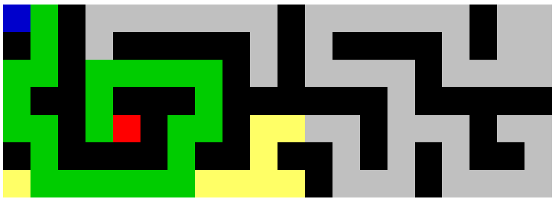
(b) Solution to board 1-3

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



(a) Board 1-4



(b) Solution to 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

```
class Node(object):
    def __init__(self, row, col, value):
        self.row = row
        self.col = col
        self.value = value
        self.g = 1000
```

```
        self.h = 1000
        self.f = 2000
        self.parent = self
        self.start = False
        self.end = False
        self.free = False
        self.wall = False
        self.cost = 2000

    def gFunc(self, start, node): #Calculates the distance from the root to the node (Manhattan distance)
        return manhattanDist(start, node)

    def hFunc(self, node, end):
        return manhattanDist(node, end)

    def fFunc(self):
        self.f = self.g + self.h

    def printNode(self):
        print(
            "\n\nPosition: \t[", self.row, " ", self.col, "]",
            "\nStart: \t", self.start,
            "\nEnd: \t", self.end,
            "\ng: \t", self.g,
            "\nh: \t", self.h,
            "\nCost: \t", self.cost,
            "\nf: \t", self.f,
            "\nFree: \t", self.free,
            "\nWall: \t", self.wall,
            "\nValue: \t", self.value,
            "\nParent: \t[", (self.parent).row, " ", (self.parent).col, "]", end=""
        )
```

2.2 A_star

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

    # Initializing the start node:
    start.g = 0
    start.h = start.hFunc(start, end)
    start.f = start.g + start.h
    start.parent = start

    #print("Start: ")
    #start.printNode()
    #print("End: ")
    #end.printNode()
```

```


    # Appending the start node to the set of opened nodes
    open.append(start)

    success = False
    while((len(open) > 0) and (success == False)):        # while the open list is not empty
        #print("\n************************************************\nOpen contains: \n")
        #for el in open:
            #el.printNode()
        #print("\n************************************************\n")

        q = open.pop(0)          # popping the first element of the open array, the one with the lowest f value.
        #print("\n--------------CURRENT NODE--------------------\n")
        #q.printNode()

        #img_object = colorPixel(False, img_object, q, (255, 255, 102))
        succ = generateAllSucc(q, nodes)      # Generating all valid neighbouring elements of q

        #print("\n******************************************")
        #print("\nValid neighbours of [", q.row, ",", ", q.col, "]: ")
        #for S in succ:
            #S.printNode()
        #print("\n******************************************\n")

        for S in succ:
            #print("In succ")
            #S.printNode()
            if (S.end == True):          # If the neighbouring element is the goal, end the while loop
                print("\n\nEnd node is found!")
                success = True
                lastNode = S
                S.parent = q
                break
            tmp_S_g = q.g + manhattanDist(q, S) # Updating the neighbour's g value
            tmp_S_h = S.hFunc(S, end)
            tmp_cost = q.cost + S.cost
            tmp_S_f = tmp_S_g + tmp_S_h + S.cost

            # If the node is already in the closed or open list, but with lower f value, skip adding it that neighbour
            if ((S in open) and (S.f <= tmp_S_f)):
                #print("\nS in open with <= f")
                continue
            if ((S in closed) and (S.f <= tmp_S_f)):
                #print("\nS in closed with <= f")
                continue
```

```

            else:  #Otherwise, add the neighbour to the open list, and set its f, g and h values
                #print("\n\nAdding node")
                S.g = tmp_S_g
                S.h = tmp_S_h
                S.cost = tmp_cost
                S.f = tmp_S_f
                S.parent = q
                #S.printNode()
                open.append(S)      # Adding S to the open list.
                open.sort(key=lambda Node: Node.f)      # TODO: check if correct. sorting the opened list after f value.
        closed.append(q) # adding q to the closed list

    # Outside while loop
    solution = []
    solution = getSolution(lastNode, solution)
    # Adding the start and end node, as the getSolution does not add them
    solution.append(start)
    solution.insert(0, end)

    return [img_object, solution]
```

2.2.1 manhattanDist()

```
def manhattanDist(start, end):
    xDist = abs(end.col - start.col)
    yDist = abs(end.row - start.row)
    #print("xDist: ", xDist, ", yDist: ", yDist)
    return xDist + yDist
```

2.2.2 readFromTxt()

```
def readFromTxt(filePath):
    file = open(filePath, "r")
    lines = file.readlines()
    #print(lines)
    return lines
```

2.2.3 generateBoard()

```
def generateBoard(board, fileName):
    img = Image.new('RGB', (len(board[0]), len(board)), "white")  # Creates image object in the size of the board.
    pixels = img.load()              # creating a pixel map

    for line in range(0, len(board)):
        for char in range(0, len(board[0])):
            if (board[line][char] == "."):
```

```
                pixels[char, line] = (192,192,192)      # open pixels appear grey

            if (board[line][char] == "#"):
                pixels[char, line] = (0,0,0)      # border pixels appear black

            if (board[line][char] == "A"):
                pixels[char, line] = (255,255,0)      # start pixel appear blue

            if (board[line][char] == "B"):
                pixels[char, line] = (255,0,0)      # end pixel appear red

            if (board[line][char] == "w"):
                pixels[char, line] = (0,0,255)      # Water

            if (board[line][char] == "m"):
                pixels[char, line] = (128,128,128)      # mountains

            if (board[line][char] == "f"):
                pixels[char, line] = (0,102,0)      # forest

            if (board[line][char] == "g"):
                pixels[char, line] = (102,204,0)      # grass

            if (board[line][char] == "r"):
                pixels[char, line] = (153,76,0)      # roads

    img.save(fileName, "PNG")
    return img
```

2.2.4 colorPixel()

```
def colorPixel(fileName, img, node, color):



    pixels = img.load()
    pixels[node.col, node.row] = color        # Marking the path yellow
    if (fileName != False):     # if image file is to be created
        img.save(fileName, "PNG")

    return img
```

2.2.5 convertToNodes()

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

```
        new = []
        for j in range(0, len(board[0])):
            new.append(Node(i,j, board[i][j]))
        nodeList.append(new)

    # Filling in necessery information for the node class
    for i in range(0, len(board)):
        for j in range(0, len(board[0])):
            if (nodeList[i][j].value == "A"):
                nodeList[i][j].start = True
                nodeList[i][j].free = True
                start = nodeList[i][j]

            if (nodeList[i][j].value == "B"):
                nodeList[i][j].end = True
                nodeList[i][j].free = True
                end = nodeList[i][j]

            if (nodeList[i][j].value == "."):
                nodeList[i][j].free = True

            if (nodeList[i][j].value == "#"):
                nodeList[i][j].wall = True

            if (nodeList[i][j].value == "w"):
                nodeList[i][j].cost = 100
                nodeList[i][j].free = True

            if (nodeList[i][j].value == "m"):
                nodeList[i][j].cost = 50
                nodeList[i][j].free = True

            if (nodeList[i][j].value == "f"):
                nodeList[i][j].cost = 10
                nodeList[i][j].free = True

            if (nodeList[i][j].value == "g"):
                nodeList[i][j].cost = 5
                nodeList[i][j].free = True

            if (nodeList[i][j].value == "r"):
                nodeList[i][j].cost = 1
                nodeList[i][j].free = True

    return [nodeList, start, end]
```

2.2.6 generateAllSucc()

```
def generateAllSucc(node, nodeList):
    colStart = 0
    colEnd = len(nodeList[0])
    rowStart = 0
    rowEnd = len(nodeList)
```

```
    #print("colEnd: ", colEnd, "rowEnd: ", rowEnd)

    iNorth = node.row - 1
    iSouth = node.row + 1
    iEast = node.col + 1
    iWest = node.col - 1


    neighbourList = []
    # If indexes are outside the board, set them as "invalid"
    if (iNorth < rowStart or (nodeList[iNorth][node.col].free == False)): # if over the array or not free
        iNorth = None
    else:
        neighbourList.append(nodeList[iNorth][node.col])

    if (iEast > colEnd - 1 or (nodeList[node.row][iEast].free == False)):
        iEast = None
    else:
        neighbourList.append(nodeList[node.row][iEast])

    if ((iSouth > rowEnd - 1) or (nodeList[iSouth][node.col].free == False)):
        iSouth = None
    else:
        neighbourList.append(nodeList[iSouth][node.col])

    if ((iWest < colStart) or (nodeList[node.row][iWest].free == False)):
        iWest = None
    else:
        neighbourList.append(nodeList[node.row][iWest])

    #print("neighbourList: ", neighbourList)
    return neighbourList
```

2.2.7 getSolution()

```
def getSolution(lastNode, solution):
    if ((lastNode.parent).start != True):
        par = lastNode.parent
        solution.append(par)
        getSolution(lastNode.parent, solution)

    #solution.append(lastNode.parent)
    return solution
```

2.2.8 colorSolution()

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

```
        colorPixel(fileName, img, el, (204, 0, 204))      # colouring
            the path

    # Coloring the start pixel:
    colorPixel(fileName, img, solution[len(solution) - 1], (255, 255, 0)
        )

    #Coloring the end pixel:
    colorPixel(fileName, img, solution[0], (255, 0, 0) )
```

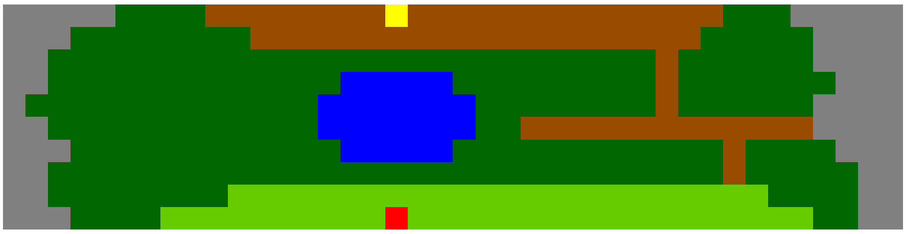
2.3 The main function

```
def main():

    #For Part 1:
    board_1_1 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-1-1.txt")
    board_1_2 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-1-2.txt")
    board_1_3 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-1-3.txt")
    board_1_4 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-1-4.txt")

    # For part 2:
    board_2_1 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-2-1.txt")
    board_2_2 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-2-2.txt")
    board_2_3 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-2-3.txt")
    board_2_4 = readFromTxt("C:\\Users\\adria\\Documents\\Dokumenter\\
        NTNU\\Introduksjon til kunstig intelligens\\Assignments\\
        A3_A_star\\boards\\boards\\board-2-4.txt")

    img_1_1 = generateBoard(board_1_1, "board_1_1.png")
    img_1_2 = generateBoard(board_1_2, "board_1_2.png")
    img_1_3 = generateBoard(board_1_3, "board_1_3.png")
    img_1_4 = generateBoard(board_1_4, "board_1_4.png")

    img_2_1 = generateBoard(board_2_1, "board_2_1.png")
    img_2_2 = generateBoard(board_2_2, "board_2_2.png")
    img_2_3 = generateBoard(board_2_3, "board_2_3.png")
    img_2_4 = generateBoard(board_2_4, "board_2_4.png")


```

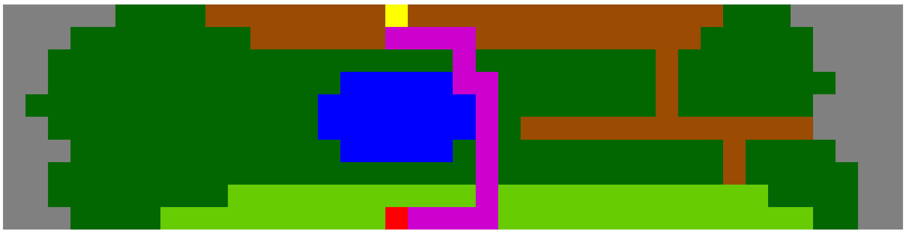
```
    [nodes, start, end] = convertToNodes(board_1_1)
    [img_1_1, sol_1_1] = A_star(nodes, start, end, img_1_1)
    colorSolution(sol_1_1, img_1_1, "board_1_1_path.png")

    [nodes, start, end] = convertToNodes(board_1_2)
    [img_1_2, sol_1_2] = A_star(nodes, start, end, img_1_2)
    colorSolution(sol_1_2, img_1_2, "board_1_2_path.png")

    [nodes, start, end] = convertToNodes(board_1_3)
    [img_1_3, sol_1_3] = A_star(nodes, start, end, img_1_3)
    colorSolution(sol_1_3, img_1_3, "board_1_3_path.png")

    [nodes, start, end] = convertToNodes(board_1_4)
    [img_1_4, sol_1_4] = A_star(nodes, start, end, img_1_4)
    colorSolution(sol_1_4, img_1_4, "board_1_4_path.png")



    [nodes, start, end] = convertToNodes(board_2_1)
    [img_2_1, sol_2_1] = A_star(nodes, start, end, img_2_1)
    colorSolution(sol_2_1, img_2_1, "board_2_1_path.png")

    [nodes, start, end] = convertToNodes(board_2_2)
    [img_2_2, sol_2_2] = A_star(nodes, start, end, img_2_2)
    colorSolution(sol_2_2, img_2_2, "board_2_2_path.png")

    [nodes, start, end] = convertToNodes(board_2_3)
    [img_2_3, sol_2_3] = A_star(nodes, start, end, img_2_3)
    colorSolution(sol_2_3, img_2_3, "board_2_3_path.png")

    [nodes, start, end] = convertToNodes(board_2_4)
    [img_2_4, sol_2_4] = A_star(nodes, start, end, img_2_4)
    colorSolution(sol_2_4, img_2_4, "board_2_4_path.png")
```

2.4 Visualization of solutions

2.4.1 Board-2-1



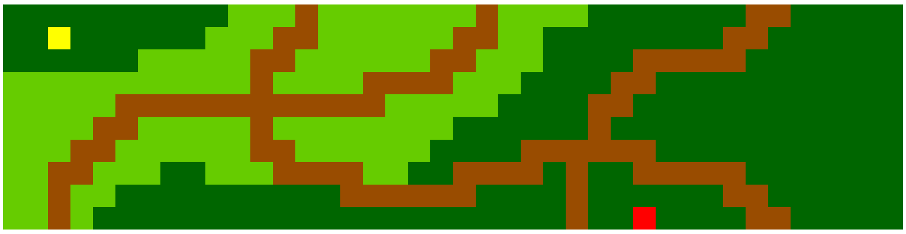
(a) Board 2-1



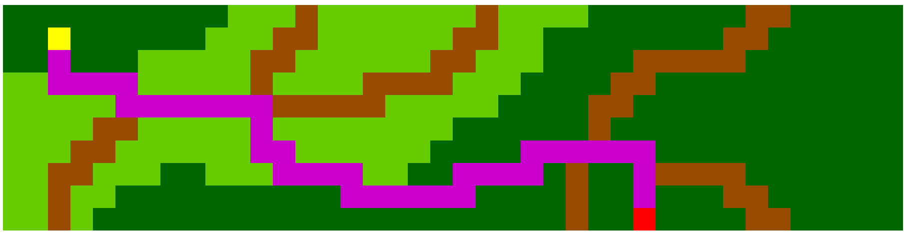
(b) Solution to board 2-1

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.

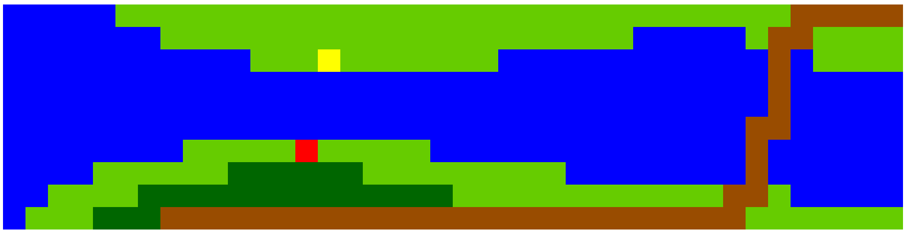
2.4.3 Board-2-3

(a) Board 2-3

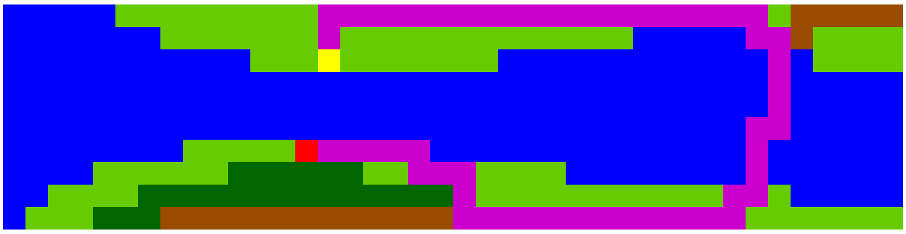
(b) Solution to board 2-3

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()

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

```
    closed = []

    # Initializing the start node:
    start.g = start.cost
    #start.h = start.hFunc(start, end)
    start.f = start.g
    start.parent = start

    #print("Start: ")
    #start.printNode()
    #print("End: ")
    #end.printNode()


    # Appending the start node to the set of opened nodes
    open.append(start)

    success = False
    while((len(open) > 0) and (success == False)):        # while the open list is not empty
        #print("\n************************************************\nOpen contains: \n")
        #for el in open:
            #el.printNode()
        #print("\n************************************************\n")

        q = open.pop(0)         # popping the first element of the open array, the one with the lowest g value.
        #print("\n------------CURRENT NODE--------------------\n")
        #q.printNode()

        #img_object = colorPixel(False, img_object, q, (255, 255, 102))
        succ = generateAllSucc(q, nodes)      # Generating all valid neighbouring elements of q

        #print("\n*******************************************")
        #print("\nValid neighbours of [", q.row, ", ", q.col, "]: ")
        #for S in succ:
            #S.printNode()
        #print("\n*******************************************\n")

        for S in succ:
            #print("In succ")
            #S.printNode()
            if (S.end == True):          # If the neighbouring element is the goal, end the while loop
                print("\n\nEnd node is found!")
                success = True
                lastNode = S
                S.parent = q
                break

```

```
            tmp_S_g = q.g + S.cost          # Updating the neighbour's g value
            #tmp_S_f = q.f + S.cost

            # If the node is already in the closed or open list, but with lower f value, skip adding it that neighbour
            if ((S in open) and (S.f <= tmp_S_g)):
                #print("\nS in open with <= f")
                continue
            if ((S in closed) and (S.f <= tmp_S_g)):
                #print("\nS in closed with <= f")
                continue

            else:  #Otherwise, add the neighbour to the open list, and set its f, g and h values
                print("\n\nAdding node")
                S.g = tmp_S_g
                #S.h = tmp_S_h
                #S.cost = tmp_cost
                #S.f = tmp_S_f
                S.parent = q
                S.printNode()
                open.append(S)      # Adding S to the open list.
                open.sort(key=lambda Node: Node.g)      # Dijkstra: Sort the opened list by the g value
                colorPixel(False, img_object, S, openedColor)

        closed.append(q) # adding q to the closed list
        colorPixel(False, img_object, S, closedColor)

    # Outside while loop
    solution = []
    solution = getSolution(lastNode, solution)
    # Adding the start and end node, as the getSolution does not add them
    solution.append(start)
    solution.insert(0, end)

    print("Dijkstra solution is, from end to start: \n")
    for el in solution:
        print("[", el.row, " ", el.col, "] ")

    return [img_object, solution]
```

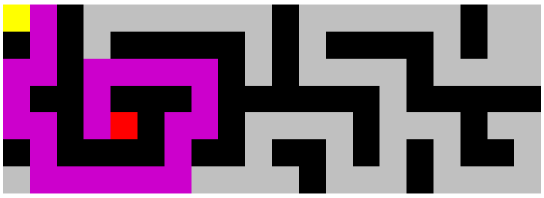
3.2 BFS()

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

    # Initializing the start node:
```

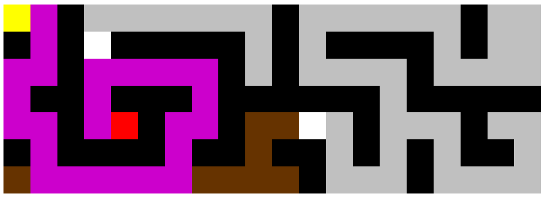
```
    #start.g = 0
    #start.h = start.hFunc(start, end)
    start.f = 0
    start.parent = start

    #print("Start: ")
    #start.printNode()
    #print("End: ")
    #end.printNode()


    # Appending the start node to the set of opened nodes
    open.append(start)

    success = False
    while((len(open) > 0) and (success == False)):        # while the open list is not empty
        #print("\n**********************************************\nOpen contains: \n")
        #for el in open:
            #el.printNode()
        #print("\n**********************************************\n")

        q = open.pop(0)          # popping the first element of the open array, the one with the lowest f value.
        #print("\n--------------CURRENT NODE--------------------\n")
        #q.printNode()

        #img_object = colorPixel(False, img_object, q, (255, 255, 102))
        succ = generateAllSucc(q, nodes)      # Generating all valid neighbouring elements of q

        #print("\n*****************************************")
        #print("\nValid neighbours of [", q.row, ", ", q.col, "]: ")
        #for S in succ:
            #S.printNode()
        #print("\n*****************************************\n")

        for S in succ:
            #print("In succ")
            #S.printNode()
            if (S.end == True):          # If the neighbouring element is the goal, end the while loop
                print("\n\nEnd node is found!")
                success = True
                lastNode = S
                S.parent = q
                break
            #tmp_S_g = q.g + manhattanDist(q, S) # Updating the neighbour's g value
            #tmp_S_h = S.hFunc(S, end)
            #tmp_cost = q.cost + S.cost
            tmp_S_f = q.f + S.cost
```

```
            # If the node is already in the closed or open list, but with lower f value, skip adding it that neighbour
            if ((S in open) and (S.f <= tmp_S_f)):
                #print("\nS in open with <= f")
                continue
            if ((S in closed) and (S.f <= tmp_S_f)):
                #print("\nS in closed with <= f")
                continue

            else:  #Otherwise, add the neighbour to the open list, and set its f, g and h values
                #print("\n\nAdding node")
                #S.g = tmp_S_g
                #S.h = tmp_S_h
                #S.cost = tmp_cost
                S.f = tmp_S_f
                S.parent = q
                #S.printNode()
                open.insert(0, S)       # Adding S to the open list at first position. FIFO, used in BFS
                colorPixel(False, img_object, S, openedColor)

        closed.append(q) # adding q to the closed list
        colorPixel(False, img_object, q, closedColor)

    # Outside while loop
    solution = []
    solution = getSolution(lastNode, solution)
    # Adding the start and end node, as the getSolution does not add them
    solution.append(start)
    solution.insert(0, end)

    print("Breadth-first solution is, from end to start: \n")
    for el in solution:
        print("[", el.row, " ", el.col, "] ")

    return [img_object, solution]
```

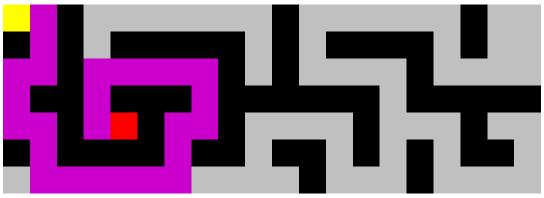
3.3 Comparisons of results for Board 1.4



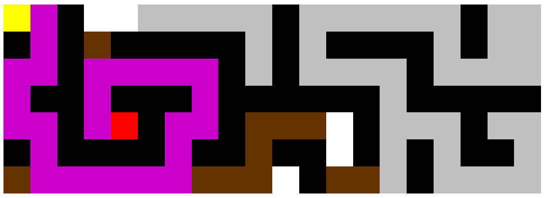
(a) A* solution



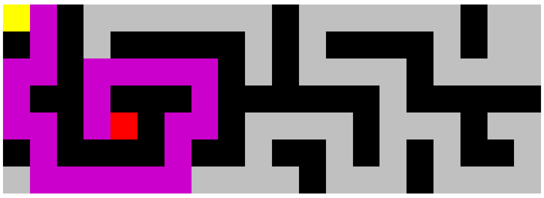
(b) The opened (white) and closed (brown) nodes in the A* solution



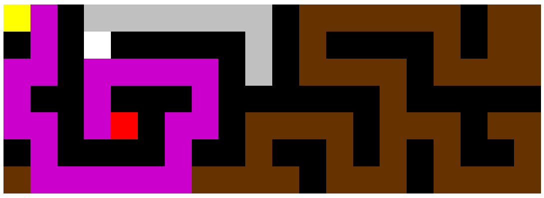
(c) A* solution



(d) The opened (white) and closed (brown) nodes in the Dijkstra solution



(e) A* solution

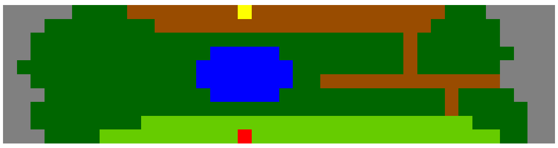


(f) The opened (white) and closed (brown) nodes in the BFS solution

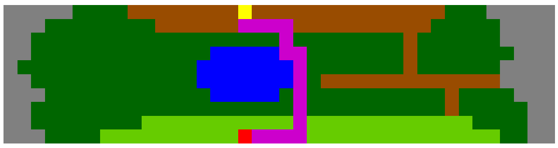
Figure 10: The found solutions compared with the opened (white) and closed (brown) nodes for Board 1-4

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.

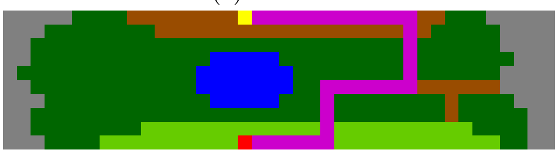
3.4 Comparison of results for Board 2.1



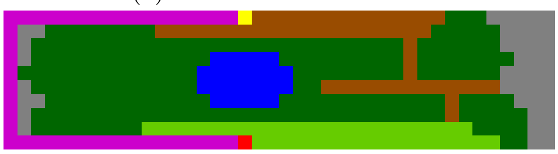
(a) Board 2-1



(b) A* solution to board 2-1

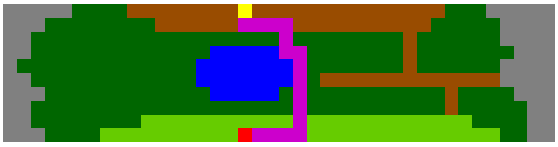


(c) Dijkstra solution to board 2-1

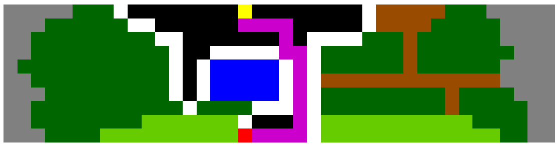


(d) BFS solution to board 2-1

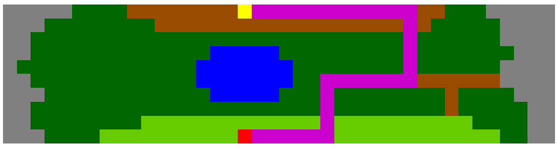
Figure 11: The original board and found solutions using A* algorithm, Dijkstra and BFS. The purple pixels mark the found path.



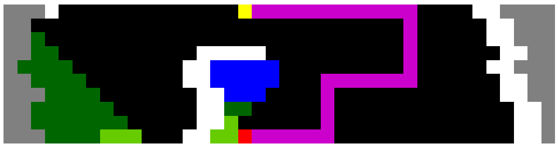
(a) A* solution



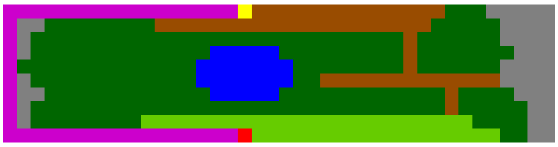
(b) The opened and closed nodes in the A* solution



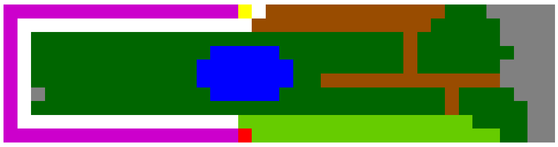
(c) Dijkstra solution to board 2-1



(d) The opened and closed nodes in Dijkstra



(e) BFS solution to board 2-1



(f) The opened and closed nodes in BFS

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://coecsl.ece.illinois.edu/ge423/lecturenotes/AstarHandOut.pdf`