Assignment #3 TDT4136 - Introduction to Artificial Intelligence

Part 1: Grids with obstacles

1.1. All coding is done in Python. References are listed below. To start off we define the necessary classes and functions for the main function (the A* search algorithm) to work. First, we need a class containing each cell of the board:

```python
import heapq
from PIL import Image, ImageDraw

class Cell():
    #Class for each cell (1x1) of the board with parameters:
    # Reachable - is it possible to get to the cell
    # x - the x coordinate of the cell
    # y - the y coordinate of the cell
    # parent - allows relations/graph structure between the cells on a path/graph
    # g - cost from start cell to current cell
    # h - heuristic cost from current cell to goal cell
    # f = g + h - the expected cost function for any given cell
    def __init__(self, x, y, reachable):
        self.reachable = reachable
        self.x = x
        self.y = y
        self.parent = None
        self.g = 0
        self.h = 0
        self.f = 0

    #To make ordering of heap work properly when values might be identical
    def __lt__(self, other):
        return self.f < other.f
```

Continuing from this we will implement the algorithm as a function of a algorithm-specific class, Astar:

```python
class AStar():
    #The main class of the algorithm with parameters:
    #opened - a heap containing the cells that are queued at any given time
    #closed - a set containing visited cells (set avoids duplicates)
    #cells - a list of all the cells in the grid
    #grid_height - The height of the grid
    #grid_width - The width of the grid
    #start - The starting cell
    #goal - The ending cell
    def __init__(self):
        self.opened = []
        heapq.heapify(self.opened)
        self.closed = set()
        self.cells = []
        self.grid_height = 0
        self.grid_width = 0
        self.start = None
        self.goal = None
```

Inside of this class we have the following methods:

```
def init_grid(self):
    #Initialize the grid from the txt-file
    #Updates cells with all the cells, sets width,
    #height, start and goal and if a cell is reachable
    with open(filename) as f:
        grid = f.read().splitlines()
    self.grid_width = len(grid)
    self.grid_height = len(grid[0])
    start = []
    goal = []
    for x in range(self.grid_width):
        for y in range(self.grid_height):
            if(grid[x][y] == "#"):
                reachable = False
            elif (grid[x][y] == "A"):
                start = [x,y]
                reachable = True
            elif (grid[x][y] == "B"):
                goal = [x,y]
                reachable = True
            else:
                reachable = True
            self.cells.append(Cell(x, y, reachable))
    self.start = self.get_cell(start[0],start[1])
    self.goal = self.get_cell(goal[0],goal[1])

def get_cell(self, x, y):
    #return the cell object corresponding to a given coordinate.
    return self.cells[x * self.grid_height + y]

def get_heuristic(self, cell):
    #return the manhattan distance from the goal
    #cell to the current cell, i.e. h(n)
    return (abs(cell.x - self.goal.x) + abs(cell.y - self.goal.y))

def get_neighbours(self, cell):
    #return list of neighbouring cells clockwise starting from
    #the rightmost one
```

```
        cells = []
        if cell.x < self.grid_width-1:
            cells.append(self.get_cell(cell.x+1, cell.y))
        if cell.y > 0:
            cells.append(self.get_cell(cell.x, cell.y-1))
        if cell.x > 0:
            cells.append(self.get_cell(cell.x-1, cell.y))
        if cell.y < self.grid_height-1:
            cells.append(self.get_cell(cell.x, cell.y+1))
        return cells

    def get_path(self):
        #get (print) the solution path as list of coordinates
        pathList = []
        cell = self.goal
        while cell.parent is not self.start:
            cell = cell.parent
            #print('path: cell: %d,%d' % (cell.x, cell.y))
            pathList.append((cell.x, cell.y))
        return pathList

    def update_cell(self, adj, cell):
        #Given a cell and a neighbour/adjacent cell, update the neighbours
        #expected cost and set neigh. cell to child of reference cell
        adj.g = cell.g + 1
        adj.h = self.get_heuristic(adj)
        adj.parent = cell
        adj.f = adj.g + adj.h
```

As well as the A* search algorithm itself:

```
    def process(self):
        #The main algorithm: A* search.
        #initialize
        self.init_grid()
        #Start node is added to the "opened" heap queue
        heapq.heappush(self.opened, (self.start.f, self.start))

        #Main loop invariant - continues as long as there are opened cells
        while (len(self.opened)):
            #pop the first element from the heap, i.e. the cell with lowest f value.
            f, cell = heapq.heappop(self.opened)
            #add the cell to the set of closed/visited cells.
            self.closed.add(cell)
            #If the current cell is the goal cell we are done.
            if cell is self.goal:
                break
            #loop through list of adjecent cells
            adj_cells = self.get_neighbours(cell)
            for adj_cell in adj_cells:
                if adj_cell.reachable and adj_cell not in self.closed:
                    #if neighbouring cell is reachable and not
                    #closed/visited, proceed:
                    if (adj_cell.f, adj_cell) in self.opened:
                        #if neighbouring cell is opened we need to
                        #see if this path is better:
                        if adj_cell.g > cell.g + 1:
                            #if the new path is better,
                            #then relax the adjacent cell.
                            self.update_cell(adj_cell, cell)
                    else:
                        #adjacent cell not opened,
                        #therefore update and push
                        #to heap to explore path further
                        self.update_cell(adj_cell, cell)
                        heapq.heappush(self.opened, (adj_cell.f, adj_cell))
```

To produce the visualization we use PIL:

```
choice = input("Choose map: 1,2,3 or 4 \n")
while(choice not in [1,2,3,4]):
    choice = input("Not a valid value, please try again \n")

filename = "board-1-" + str(choice) + ".txt"
a = AStar()
path = a.process()
pathList = a.get_path()

with open(filename) as f:
     grid = f.read().splitlines()

im = Image.new("RGB", (20*a.grid_height, 20*a.grid_width), (255, 255, 255))
draw = ImageDraw.Draw(im)
draw.line(((0,0), (0, 20*a.grid_width)), fill=(0,0,0), width = 1)
draw.line(((0,0), (20*a.grid_height, 0)), fill=(0,0,0), width = 1)
draw.line(((20*a.grid_height, 20*a.grid_width-1), (0, 20*a.grid_width-1)),
fill=(0,0,0), width = 1)
draw.line(((20*a.grid_height - 1, 0), (20*a.grid_height - 1, 20*a.grid_width)),
fill=(0,0,0), width = 1)

for x in range(a.grid_width):
    for y in range(a.grid_height-1, -1, -1):
        if (grid[x][y]=="#"):
            #draw black square dimension 20x20 if unreachable
            draw.rectangle(((20*y,20*x), (20*(y+1), 20*(x+1))) ,
            fill=(0,0,0), outline=(0,0,0))
        elif ((x,y) in pathList):
            #draw small blue rectangle for path
            draw.rectangle(((20*y+7,20*x+7), (20*y+7+4, 20*x+7+4)),
            fill=(30,144,255), outline=(30,144,255))
        elif (grid[x][y]=="B"):
            #goal is green rectangle
            draw.rectangle(((20*y+7,20*x+7), (20*y+7+4, 20*x+7+4)),
            fill=(124,252,0), outline=(124,252,0))
        elif (grid[x][y]=="A"):
            #start is red rectangle
            draw.rectangle(((20*y+7,20*x+7), (20*y+7+4, 20*x+7+4)),
            fill=(255,0,0), outline=(255,0,0))

del draw
im.save("test1-" + str(choice) + ".png", "PNG")
```

1.2. The result of this code is seen below:

1)

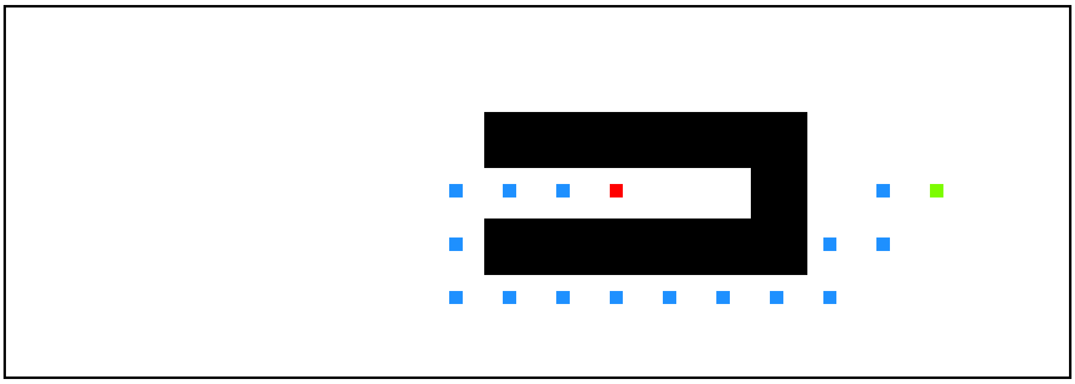

FIGURE 1. A* search on board 1-1

2)

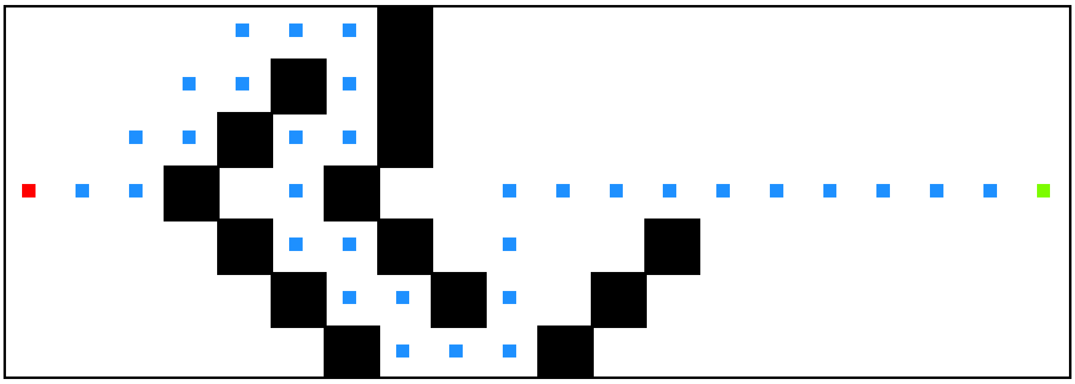

FIGURE 2. A* search on board 1-2

3)

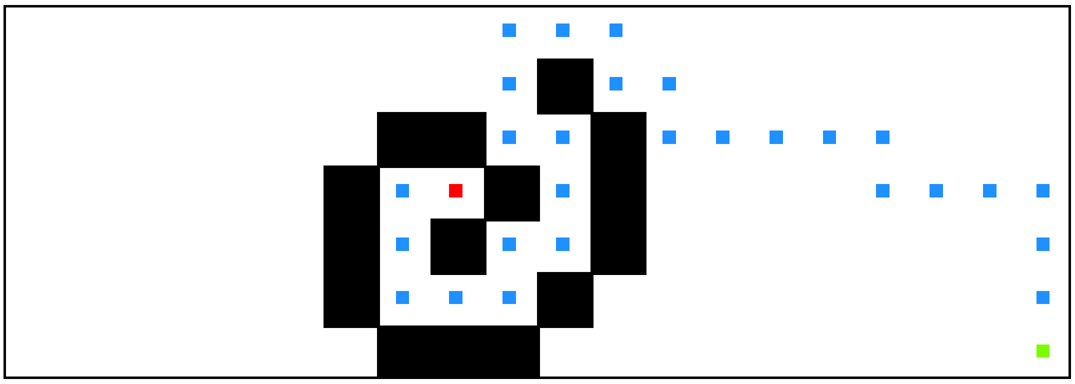

FIGURE 3. A* search on board 1-3

4)

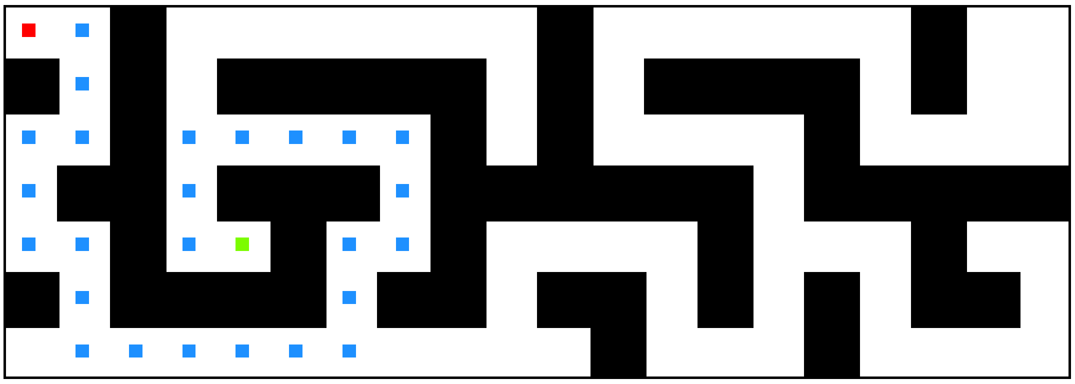

FIGURE 4. A* search on board 1-4

Part 2: Grids with different cell costs

2.1. For this problem, much of the same code is recycled from a) with minor changes. The cell class is changed as we no longer need the "reachable" attribute:

```
class Cell():
    #Class for each cell (1x1) of the board with parameters:
    # value - the cost of moving through the cell
    # x - the x coordinate of the cell
    # y - the y coordinate of the cell
    # parent - allows relations/graph structure between the cells on a path/graph
    # g - cost from start cell to current cell
    # h - heuristic cost from current cell to goal cell
    # f = g + h - the expected cost function for any given cell
    def __init__(self, x, y, value):
        self.value = value
        self.x = x
        self.y = y
        self.parent = None
        self.g = 0
        self.h = 0
        self.f = 0
```

Furthermore the function that initializes the board is also changed so that each cell has an associated value of cost:

```
def init_grid(self):
    #Initialize the grid from the txt-file
    #Updates cells with all the cells, sets width, height,
    #start and goal and the associated value of the cell
    with open(filename) as f:
        grid = f.read().splitlines()
    self.grid_width = len(grid)
    self.grid_height = len(grid[0])
    start = []
    goal = []
    for x in range(self.grid_width):
        for y in range(self.grid_height):
            ch = grid[x][y]
            if(ch=='w'):
                value = 100
            elif(ch=='m'):
                value = 50
            elif(ch=='f'):
                value = 10
            elif(ch=='g'):
                value = 5
            elif(ch=='r'):
                value = 1
            elif (ch == "A"):
                start = [x,y]
            elif (grid[x][y] == "B"):
                goal = [x,y]
            self.cells.append(Cell(x, y, value))
    self.start = self.get_cell(start[0],start[1])
    self.goal = self.get_cell(goal[0],goal[1])
```

The visualizations are produces by the following code:

```
choice = input("Choose map: 1,2,3 or 4 \n")
while(choice not in [1,2,3,4]):
    choice = input("Not a valid value, please try again \n")

filename = "board-2-" + str(choice) + ".txt"
a = AStar()
path = a.process()
pathList = a.get_path()

with open(filename) as f:
     grid = f.read().splitlines()

im = Image.new("RGB", (20*a.grid_height, 20*a.grid_width), (255, 255, 255))
draw = ImageDraw.Draw(im)
draw.line(((0,0), (0, 20*a.grid_width)), fill=(0,0,0), width = 1)
draw.line(((0,0), (20*a.grid_height, 0)), fill=(0,0,0), width = 1)
draw.line(((20*a.grid_height, 20*a.grid_width-1), (0, 20*a.grid_width-1)),
 fill=(0,0,0), width = 1)
draw.line(((20*a.grid_height - 1, 0), (20*a.grid_height - 1, 20*a.grid_width)),
 fill=(0,0,0), width = 1)

for x in range(a.grid_width):
    for y in range(a.grid_height-1, -1, -1):
        #colouring of the map
         if (a.get_cell(x,y).value == 100):
             draw.rectangle(((20*y,20*x), (20*(y+1), 20*(x+1))) ,
              fill=(30,144,255), outline=(30,144,255))
         elif (a.get_cell(x,y).value == 50):
             draw.rectangle(((20*y,20*x), (20*(y+1), 20*(x+1))),
              fill=(169,169,169), outline=(169,169,169))
         elif (a.get_cell(x,y).value == 10):
             draw.rectangle(((20*y,20*x), (20*(y+1), 20*(x+1))),
              fill=(0,100,0), outline=(0,100,0))
         elif (a.get_cell(x,y).value == 5):
             draw.rectangle(((20*y,20*x), (20*(y+1), 20*(x+1))),
              fill=(152,251,152), outline=(152,251,152))
         elif (a.get_cell(x,y).value == 1):
             draw.rectangle(((20*y,20*x), (20*(y+1), 20*(x+1))),
              fill=(205,133,63), outline=(205,133,63))
         if ((x,y) in pathList):
             #draw black circle on path
             draw.ellipse(((20*y+7,20*x+7), (20*y+7+4, 20*x+7+4)),
             fill=(0,0,0), outline=(0,0,0))
         elif (grid[x][y]=="B"):
             draw.ellipse(((20*y+7,20*x+7), (20*y+7+4, 20*x+7+4)),
             fill=(124,252,0), outline=(124,252,0))
         elif (grid[x][y]=="A"):
             draw.ellipse(((20*y+7,20*x+7), (20*y+7+4, 20*x+7+4)),
             fill=(255,0,0), outline=(255,0,0))
del draw
im.save("test2-" + str(choice) + ".png", "PNG")
```

2.2. The code then produces the following images:

1)

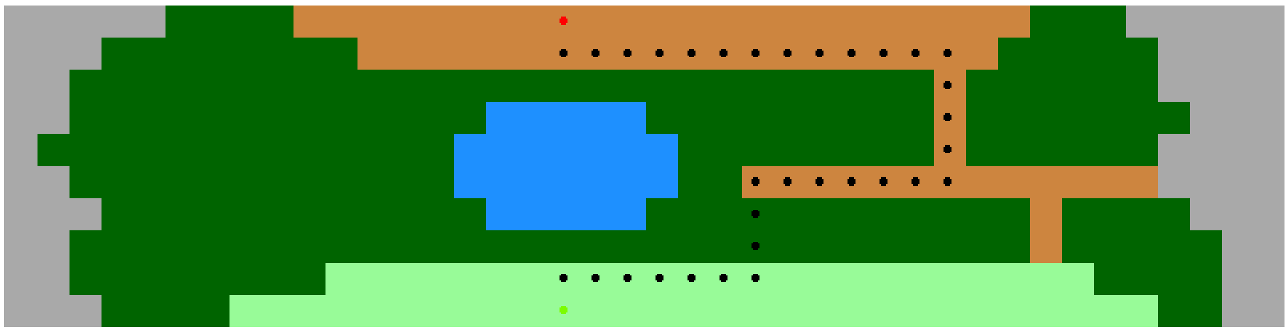

FIGURE 5. A* search on board 2-1

2)

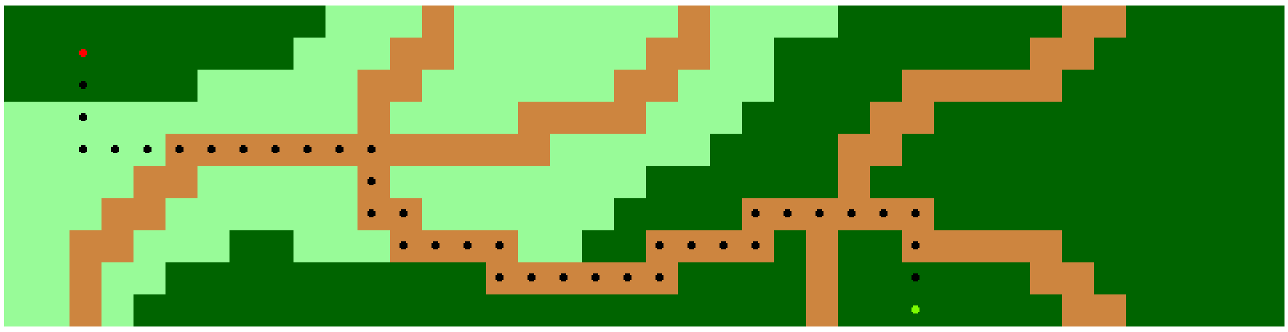

FIGURE 6. A* search on board 2-2

3)

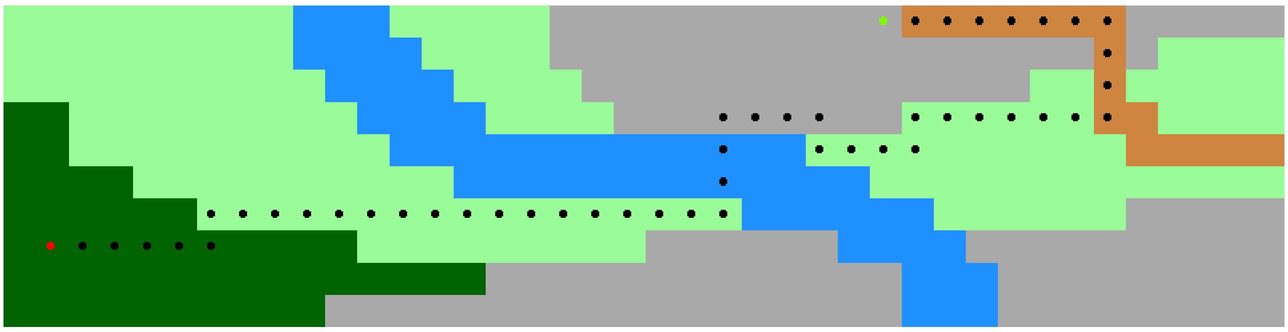

Figure 7. A* search on board 2-3

4)

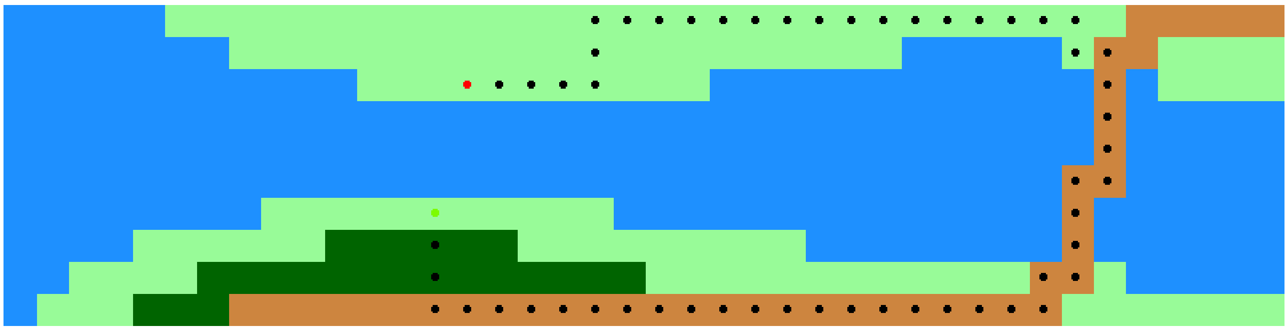

Figure 8. A* search on board 2-4

Part 3: Comparison with BFS and Dijkstra's algorithm

3.1. Some changes in the code is needed here. The way we have solved this is by asking the user for input on which algorithm to choose:

```
alg = input("Choose algorithm: \n 1: A* search \n 2: BFS \n 3: Dijkstra \n")
while(alg not in [1,2,3]):
    alg = input()
```

From there it is noted that whenever the user chooses option 2 we need a queue for the list of opened nodes in contrast to a priority queue/heap. If the user chooses option 3 we have no heuristic function. The changes are summarized here:

```
class AStar():
    #The main class of the algorithm with parameters:
    #opened - a heap containing the cells that are queued at any given time
    #closed - a set containing visited cells (set avoids duplicates)
    #cells - a list of all the cells in the grid
    #grid_height - The height of the grid
    #grid_width - The width of the grid
    #start - The starting cell
    #goal - The ending cell
    def __init__(self):
        self.opened = []
        if (alg == 2):
            self.opened = deque(self.opened)
        else:
            heapq.heapify(self.opened)
        self.closed = set()
        self.cells = []
        self.grid_height = 0
        self.grid_width = 0
        self.start = None
        self.goal = None

    def get_closed(self):
        #return a list of the closed cells
        return self.closed

    def get_opened(self):
        tmp = []
        for cell in self.opened:
            tmp.append(cell[1])
        return tmp
(...)

    def get_heuristic(self, cell):
        if (alg == 3):
            #Dijkstra
            return 0
        else:
            return (abs(cell.x - self.goal.x) + abs(cell.y - self.goal.y))

(...)

    def process(self):
        #The main algorithm: A* search.
        #initialize
        self.init_grid()
        #Start node is added to the "opened" heap or queue
        if (alg == 2):
            self.opened.append((self.start.f, self.start))
        else:
            heapq.heappush(self.opened, (self.start.f, self.start))
        #main loop invariant - continues as long as there are opened cells
        while (len(self.opened)):
            #pop the first element from the heap/queue,
            #i.e. the cell with lowest f value.
            if (alg == 2):
                f, cell = self.opened.popleft()
            else:
                f, cell = heapq.heappop(self.opened)
            #add the cell to the set of closed cells.
            self.closed.add(cell)
            #if the current cell is the goal cell we are done.
            if cell is self.goal:
                break
            #loop through list of adjecent cells
            adj_cells = self.get_neighbours(cell)
            for adj_cell in adj_cells:
                if adj_cell not in self.closed:
                    if (adj_cell.f, adj_cell) in self.opened:
                        #if neighbouring cell is opened
                        #we need to see if the path is better:
                        if adj_cell.g > cell.g + adj_cell.value:
                            #if the new path is better,
                            #update current adjacent cell.
                            self.update_cell(adj_cell, cell)
                    else:
                        #adjacent cell not opened, therefore
                        #update and push to heap to explore path further
                        self.update_cell(adj_cell, cell)
                        if (alg == 2):
                            self.opened.append((adj_cell.f, adj_cell))
                        else:
                            heapq.heappush(self.opened, (adj_cell.f, adj_cell))
```

To produce the visualization we have the following snippet:

```
choice = input("Choose map: 1,2,3 or 4 \n")
while(choice not in [1,2,3,4]):
    choice = input("Not a valid value, please try again \n")

filename = "board-2-" + str(choice) + ".txt"
a = AStar()
path = a.process()
pathList = a.get_path()

with open(filename) as f:
     grid = f.read().splitlines()

im = Image.new("RGB", (20*a.grid_height, 20*a.grid_width), (255, 255, 255))
draw = ImageDraw.Draw(im)
closed = a.get_closed()
opened = a.get_opened()
for x in range(a.grid_width):
    for y in range(a.grid_height-1, -1, -1):
        #colouring of the map
         if (a.get_cell(x,y).value == 100):
             draw.rectangle(((20*y,20*x), (20*(y+1), 20*(x+1))),
              fill=(30,144,255), outline=(30,144,255))
         elif (a.get_cell(x,y).value == 50):
             draw.rectangle(((20*y,20*x), (20*(y+1), 20*(x+1))),
             fill=(169,169,169), outline=(169,169,169))
         elif (a.get_cell(x,y).value == 10):
             draw.rectangle(((20*y,20*x), (20*(y+1), 20*(x+1))),
             fill=(0,100,0), outline=(0,100,0))
         elif (a.get_cell(x,y).value == 5):
             draw.rectangle(((20*y,20*x), (20*(y+1), 20*(x+1))),
             fill=(152,251,152), outline=(152,251,152))
         elif (a.get_cell(x,y).value == 1):
             draw.rectangle(((20*y,20*x), (20*(y+1), 20*(x+1))),
             fill=(205,133,63), outline=(205,133,63))
         if ((x,y) in pathList):
             #draw blue circle on path
             draw.ellipse(((20*y+7,20*x+7), (20*y+7+4, 20*x+7+4)),
             fill=(0,0,0), outline=(0,0,0))
         elif (grid[x][y]=="B"):
             draw.ellipse(((20*y+7,20*x+7), (20*y+7+4, 20*x+7+4)),
             fill=(124,252,0), outline=(124,252,0))
         elif (grid[x][y]=="A"):
             draw.ellipse(((20*y+7,20*x+7), (20*y+7+4, 20*x+7+4)),
             fill=(255,0,0), outline=(255,0,0))
         elif (a.get_cell(x,y) in closed):
             #draw x in closed cells
             draw.line(((20*y+6,20*x+6), (20*y+14, 20*x+14)),
             fill=(40,40,40), width = 1)
             draw.line(((20*y+6,20*x+14), (20*y+14, 20*x+6)),
             fill=(40,40,40), width = 1)
         elif (a.get_cell(x,y) in opened):
             #draw star in opened cells
             draw.polygon(((20*y + 5.5, 20*x + 8), (20*y + 9.5, 20*x +13),
             (20*y + 13.5, 20*x + 8)), fill=(55,55,55))
            draw.polygon(((20*y + 5.5, 20*x + 11), (20*y + 9.5, 20*x + 6),
            (20*y + 13.5, 20*x + 11)), fill=(55,55,55))
del draw
im.save("test3-" + str(choice) + str(alg) + ".png", "PNG")
```

3.2. For this task we have chosen board 4 from the second part and the first map from the first part.

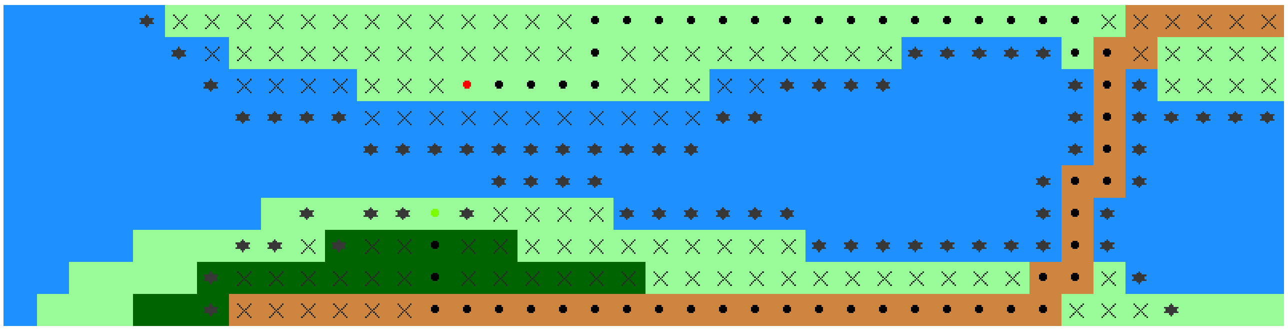

FIGURE 9. A* search on board 2-4

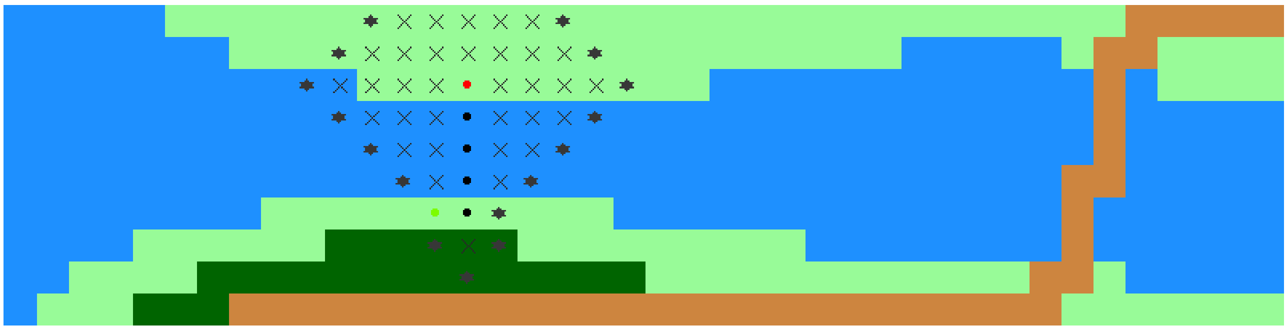

FIGURE 10. BFS on board 2-4

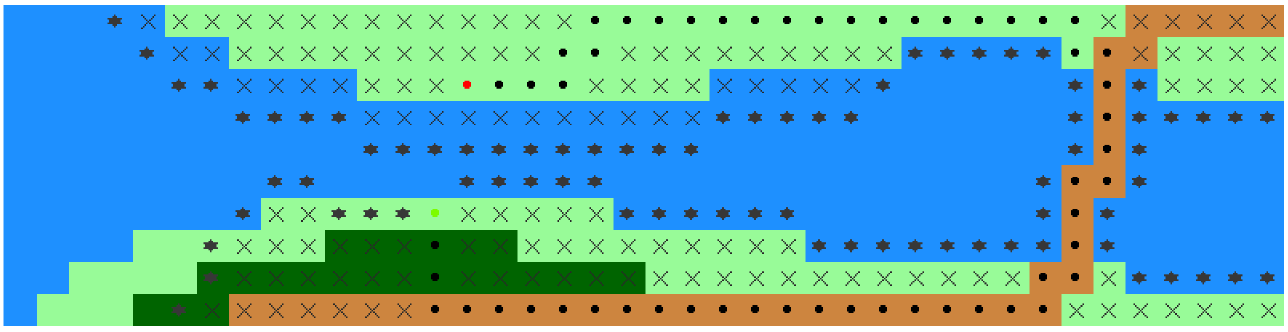

FIGURE 11. Dijkstra on board 2-4

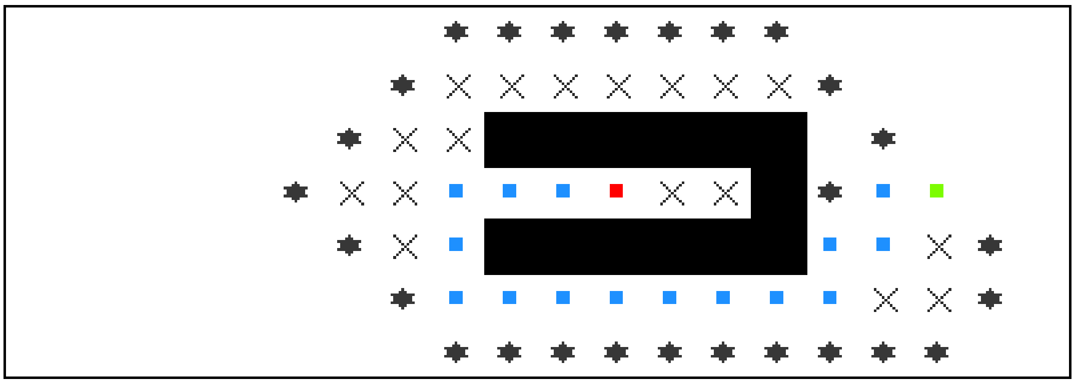
FIGURE 12. A* search on board 1-4

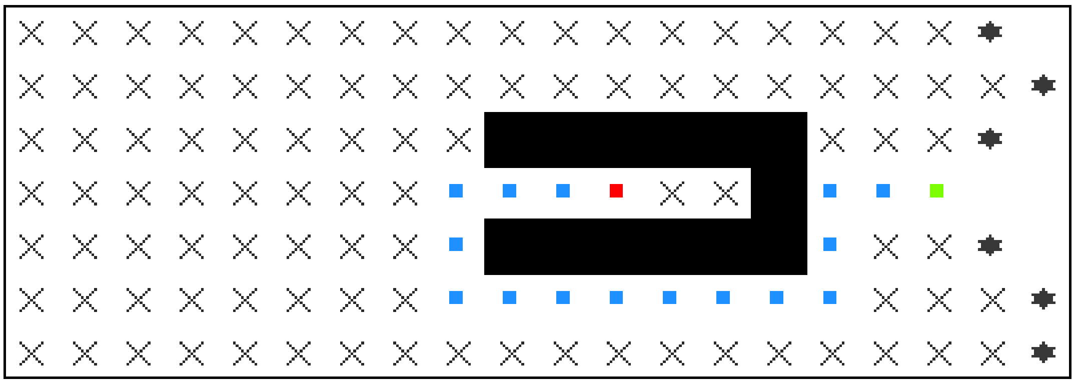
FIGURE 13. BFS on board 1-4

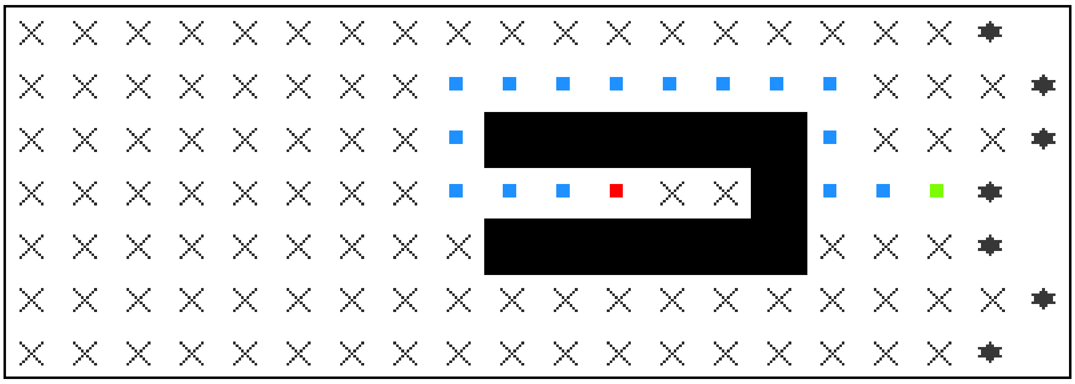
FIGURE 14. Dijkstra on board 1-4

3.3.

a) The path choice in board 2-4 is not surprising: Both A* and Dijkstra find more or less the same path and BFS finds the shortest path which is to cross the ”river”. This is because there are no considerations of cost in the implementation of BFS as it used a queue, not a heap. In board 1-1 all algorithms have found more or less the same path. This too is not surprising due to how the board is formed.

b) The number of closed cells are much smaller in both boards in the A* search. This is to be expected due to the heuristic function which gives a sensible direction to continue searching in. This is not the case in BFS or Dijkstra. In BFS one could image that the search is done by expanding a larger and larger circle around the starting point until the goal cell is found. This implies a large amount of closed cells and that the open cells are the ones on the perimeter of the circles. Dijsktra and A* have more or less the same amount of closed and open cells with Dijkstra having a small amount of more closed cells. This is because Dijkstra is considering the locally optimal solution and therefore needs to expand slightly further before it concludes that a path is not worth continuing in contrast to a cheaper path. This is for instance why we see that Dijkstra in board 2-4 continues to the bottom left corner since this is a cheap path. But the direction is wrong, as the A* search notes which gives a penalty in the heuristic function.

When there are no cost associated with the cells, BFS and Dijkstra are more similar since there are nothing to order the priority queue by when every cell have the same weight. Here we see that A* search is totally different due to the heuristic function making the priority queue "worthwhile".

References

https://www.laurentluce.com/posts/solving-mazes-using-python-simple-recursivity-and-a-search/