Norwegian University of Science and Technology

Assignment Title

Assignment 2
-
Using the A* Algorithm

Course

TDT4136 Introduction to Artificial Intelligence

Semester

FALL 2018

Date Submitted

27/09/2018

Submitted by

Dusan Jakovic

NTNU

Kunnskap for en bedre verden

Part 1: Grids with Obstacles

1.1

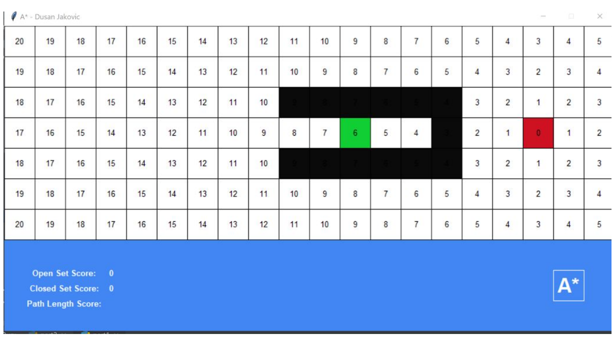
I wrote the algorithm and rest of the application by myself. I used the A*-wikipedia-page (pseudo code) as inspiration for the algorithm: https://en.wikipedia.org/wiki/A*_search_algorithm.
I used the graphics.py library to make the program visual.

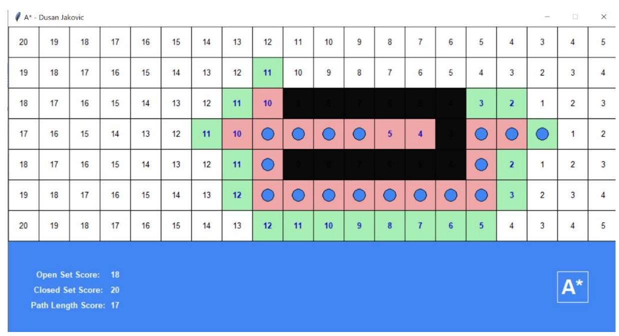
The source code can be found under “Source Code: Part 1” at the last pages

1.2

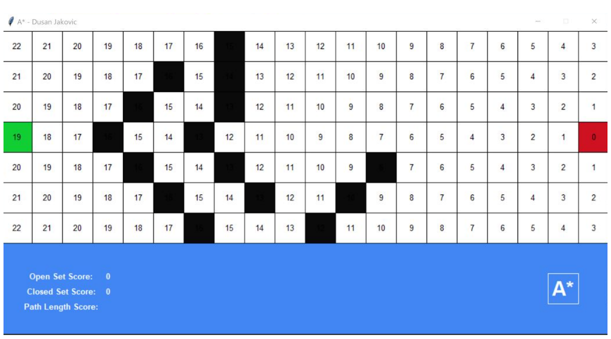
All the cells have a value in them, which represent the h-value of the cell. The green cell is the start and the red one is the end/goal. When the program runs, the cells which are in the open set will be represented with a lighter shade of green, while the cells in the closed set will be represented with a lighter red. A blue dot without fill will represent the cell that is currently visited/evaluated. When the algorithm executes, the path will be drawn by a dots. If the path isn't found, the program will return a message that the path couldn’t be reached.

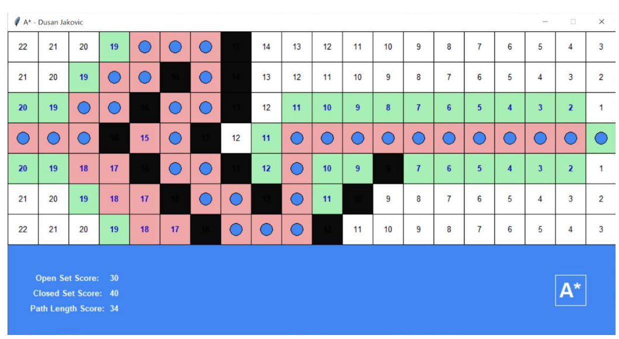
board-1-1



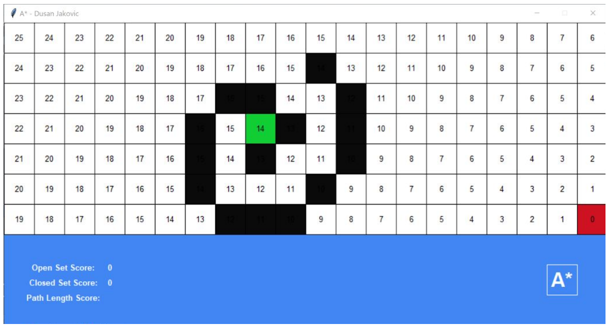


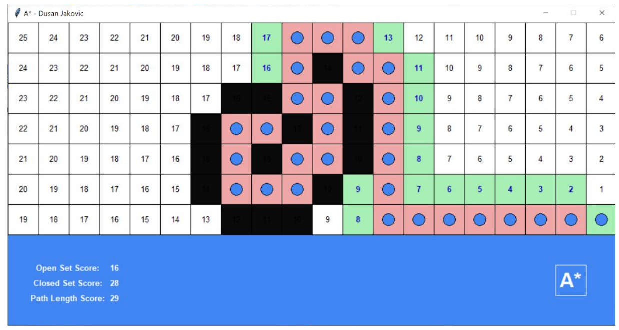
board-1-2



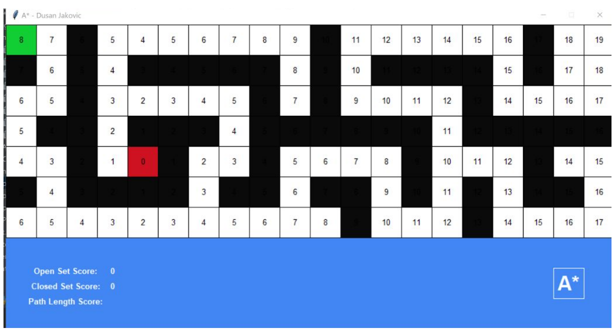


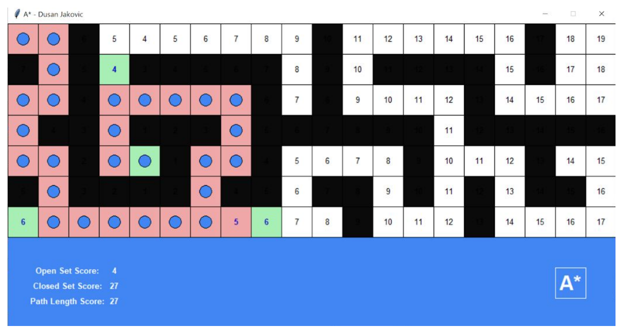
board-1-3





board-1-4





Part 2: Grids with Different cell Costs

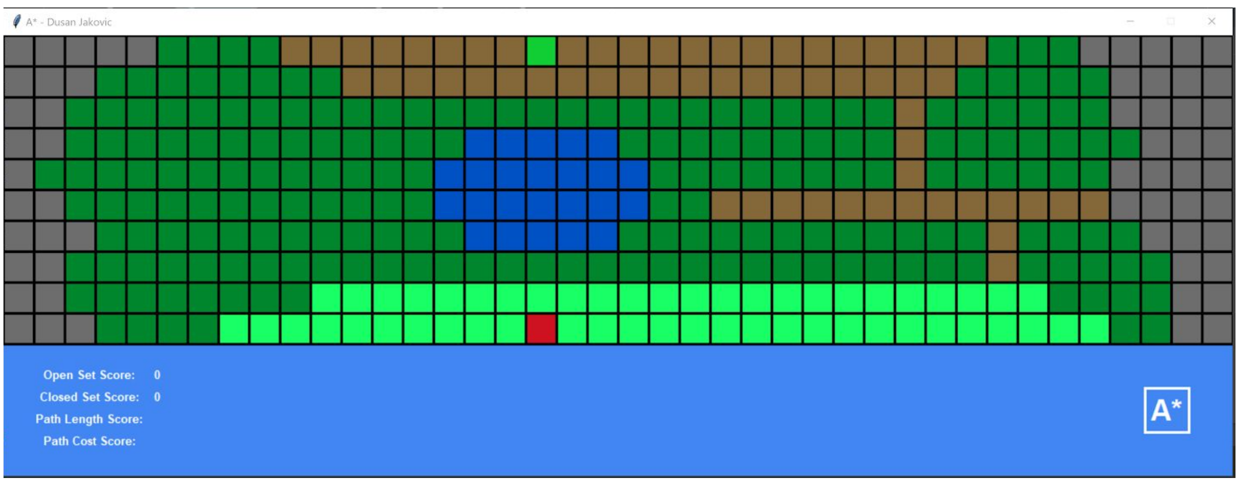
2.1

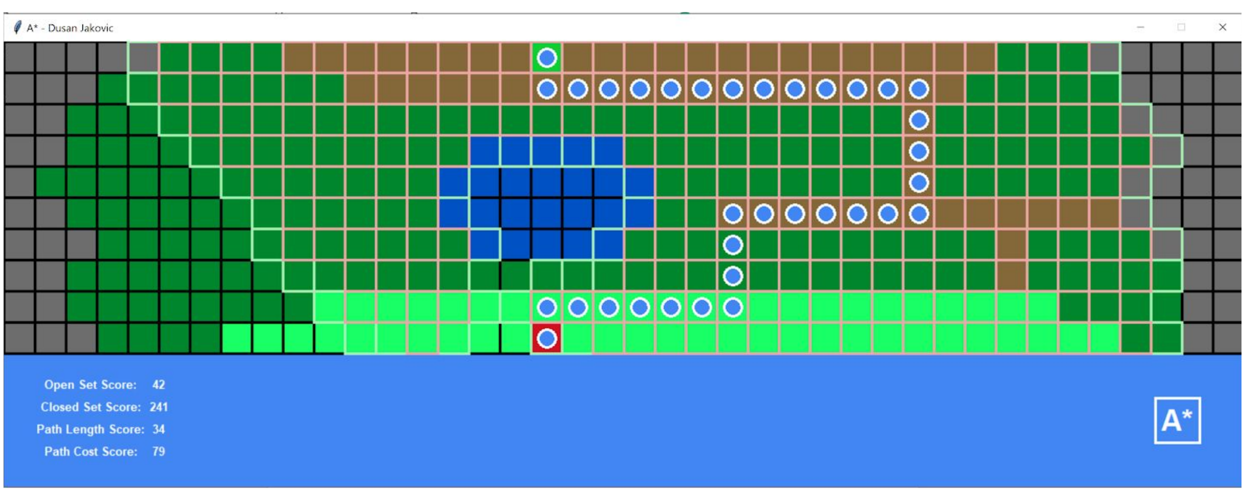
The visual part is slightly changed to make it a bit more visible: the open set and closed set are represented with red and green outline, instead of fill..I also added a “Path Cost Score”, which is the total cost of the path - considering the cost of each cell.

The source code can be found under “Source Code: Part 2” at the last pages

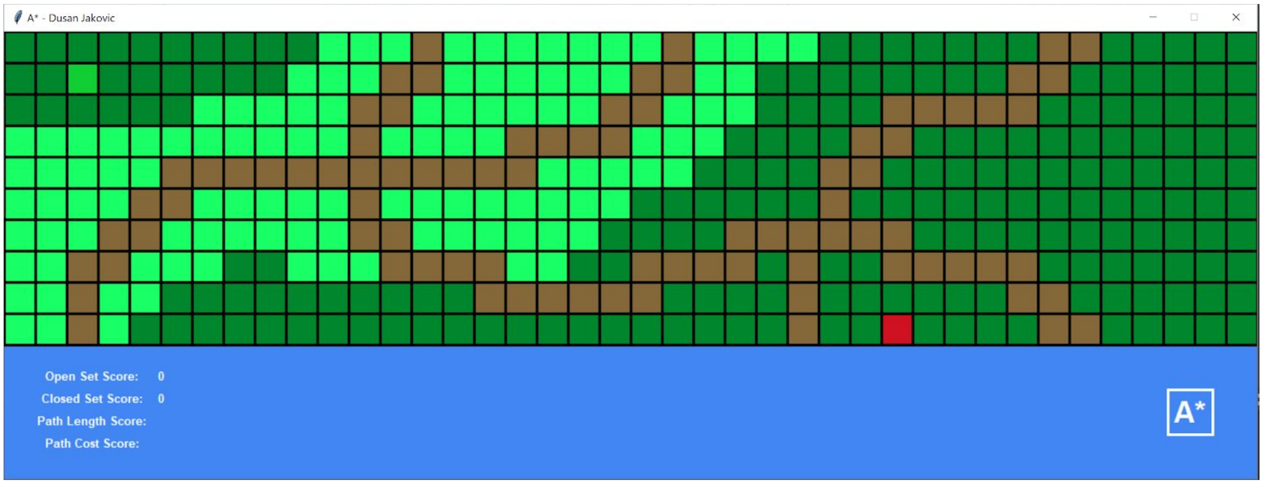
2.2

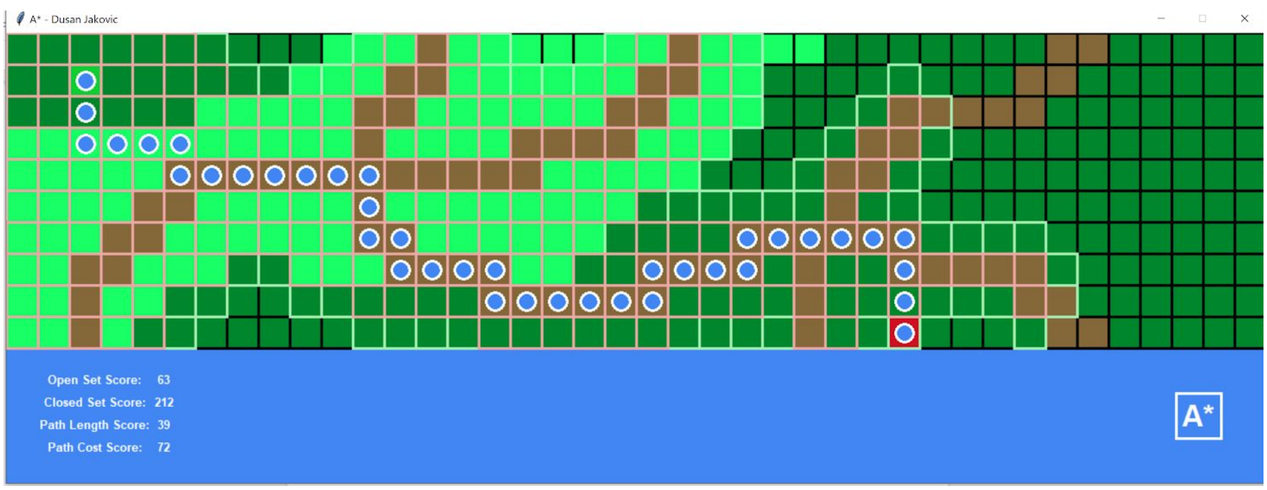
board-2-1



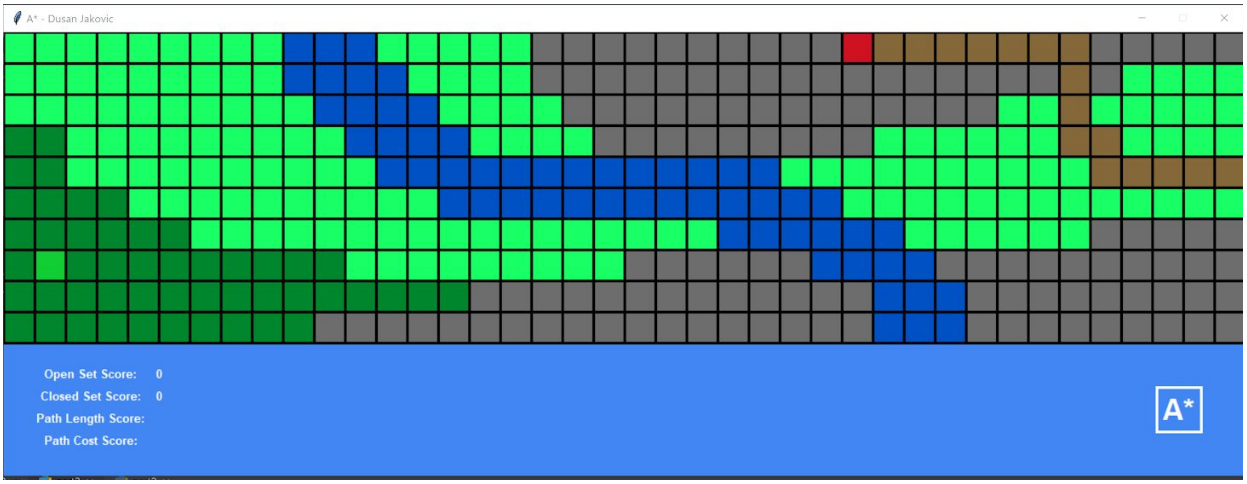


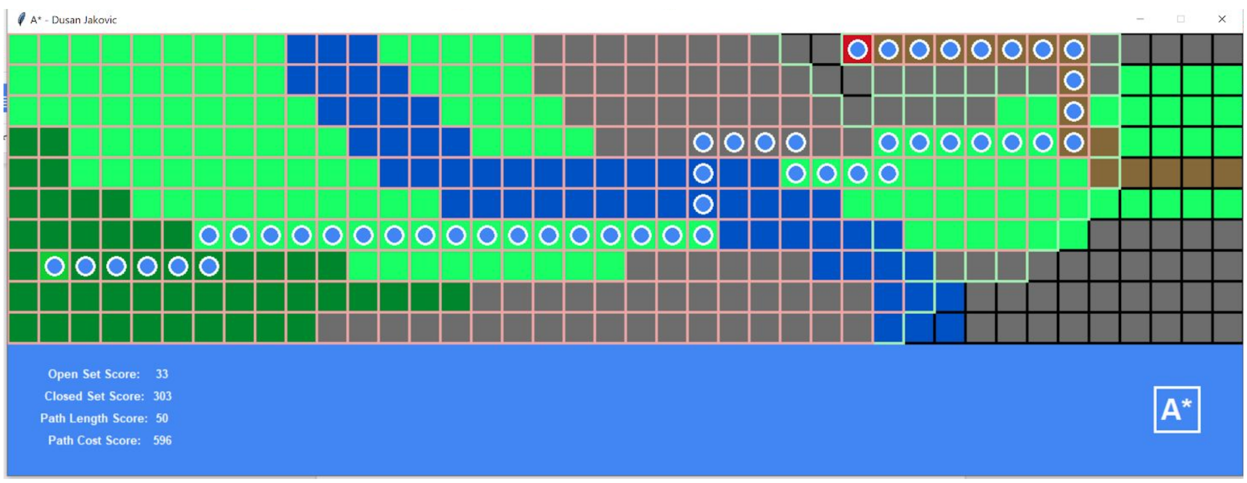
board-2-2


A* - Dusan Jakovic
Open Set Score: 0
Closed Set Score: 0
Path Length Score:
Path Cost Score:
A*

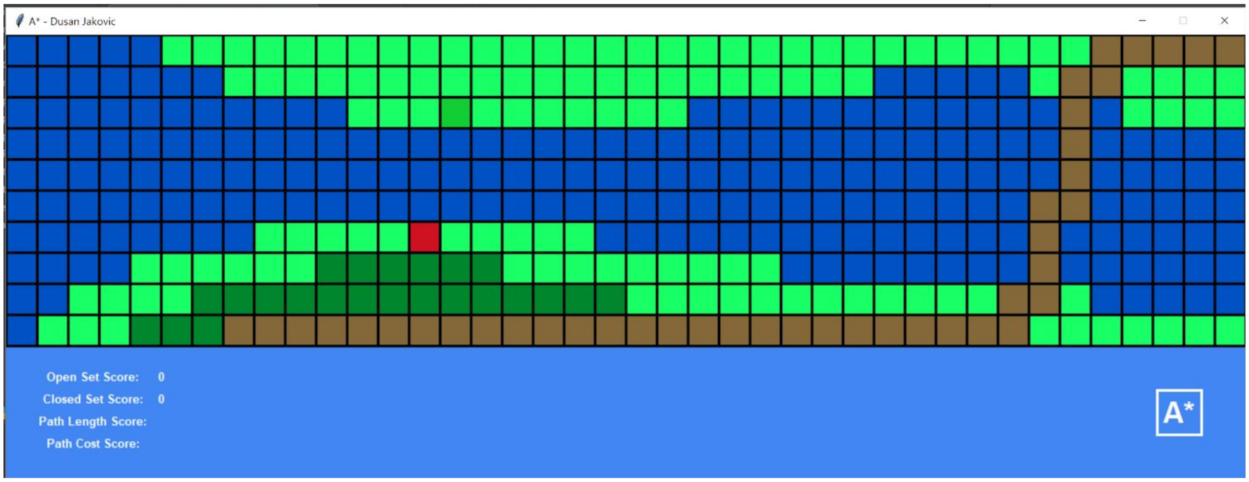

A* - Dusan Jakovic
Open Set Score: 63
Closed Set Score: 212
Path Length Score: 39
Path Cost Score: 72
A*

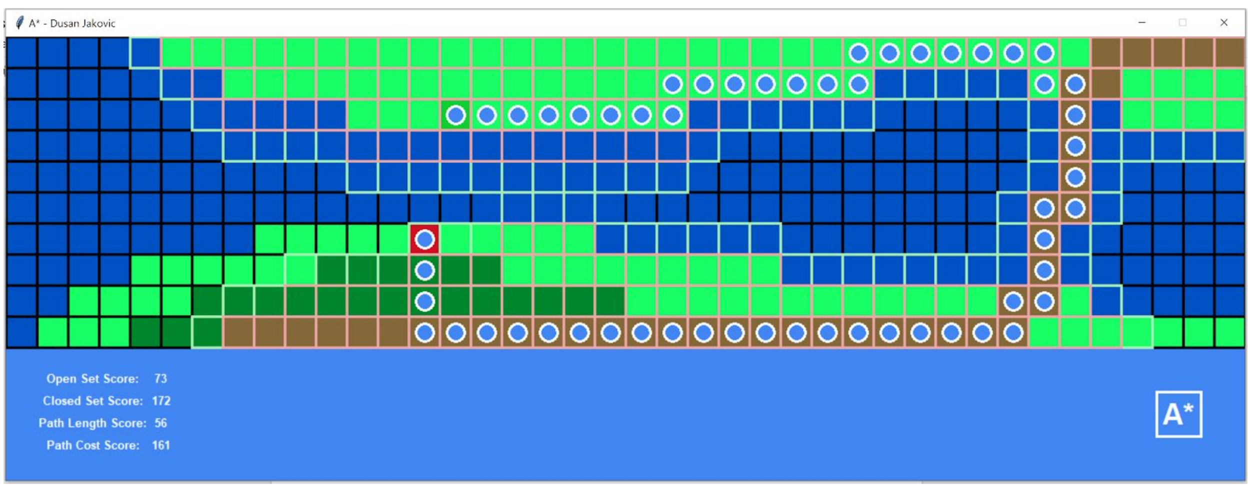
board-2-3


A* - Dusan Jakovic
Open Set Score: 0
Closed Set Score: 0
Path Length Score:
Path Cost Score:
A*


A* - Dusan Jakovic
Open Set Score: 33
Closed Set Score: 303
Path Length Score: 50
Path Cost Score: 596
A*

board-2-4


A* - Dusan Jakovic
Open Set Score: 0
Closed Set Score: 0
Path Length Score:
Path Cost Score:
A*


A* - Dusan Jakovic
Open Set Score: 73
Closed Set Score: 172
Path Length Score: 56
Path Cost Score: 161
A*

Part 3: Comparison with BFS and Dijkstra’s Algorithm

3.1

The visual part of the program is the same as in the previous part (part 2). I consider this sufficient enough for an answer to 3.1.

The source code can be found in the the zipped folder “code”

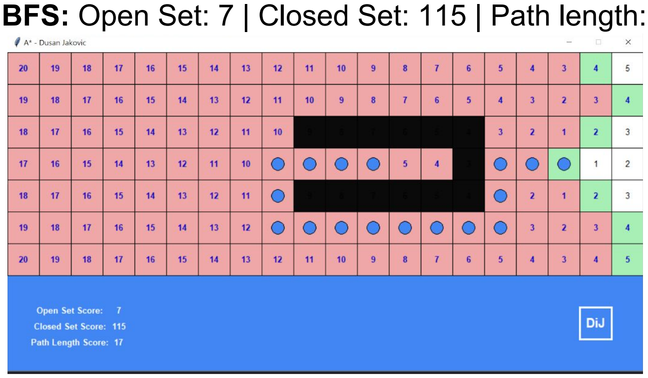
File: part2 (The code for BFS and Dijkstra is commented out)

3.2

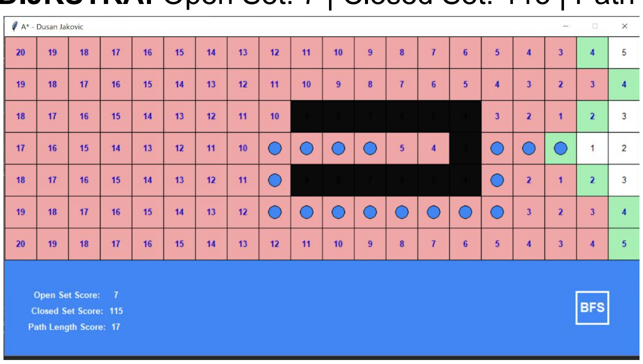
board-1-1

A*: Open Set: 18 | Closed Set: 28 | Path length: 17 | Path Cost = Path Length

BFS: Open Set: 7 | Closed Set: 115 | Path length: 17 | Path Cost = Path Length

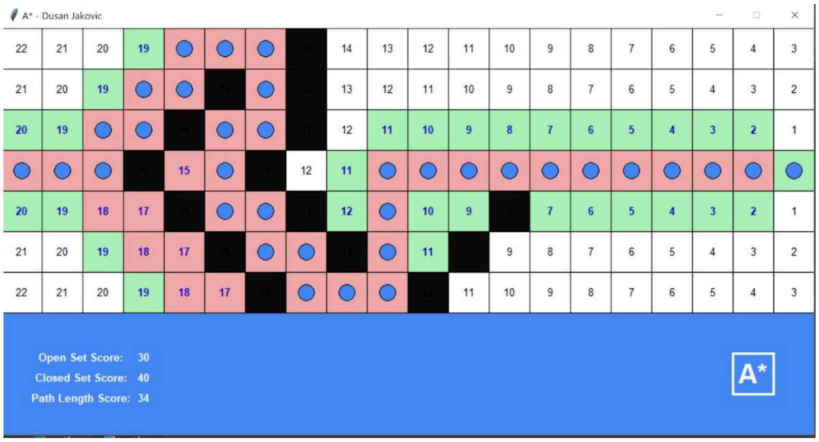


DIJKSTRA: Open Set: 7 | Closed Set: 115 | Path length: 17 | Path Cost = Path Length

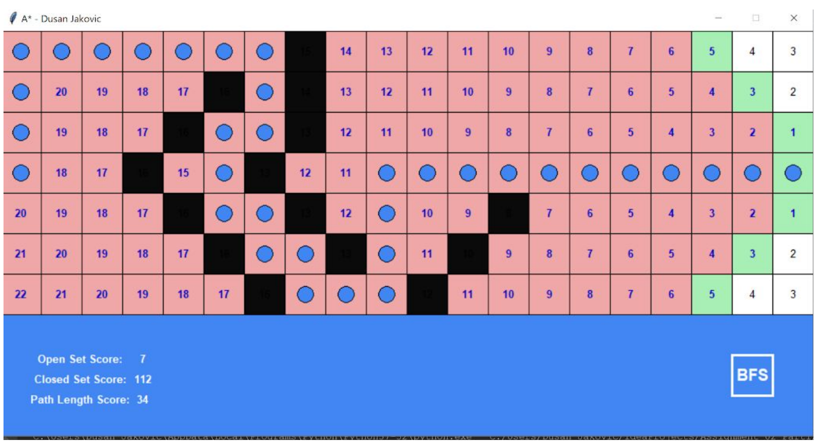


Board-1-2

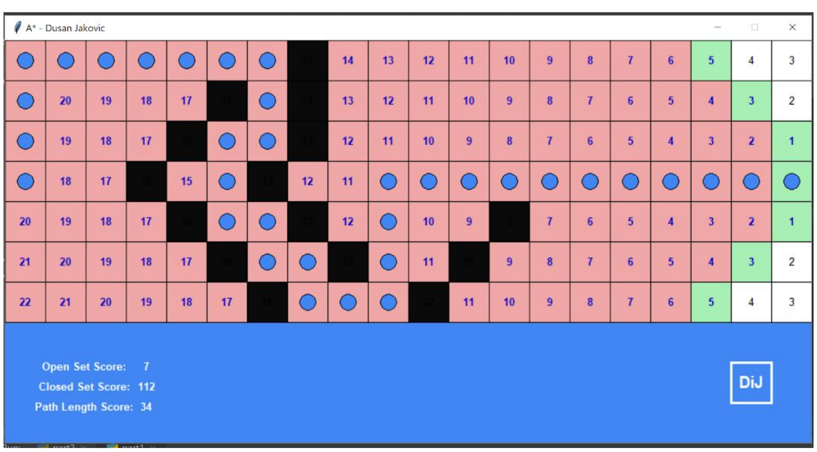
A*: Open Set: 30 | Closed Set: 40 | Path length: 34 | Path Cost = Path Length



BFS: Open Set: 7 | Closed Set: 112 | Path length: 34 | Path Cost = Path Length

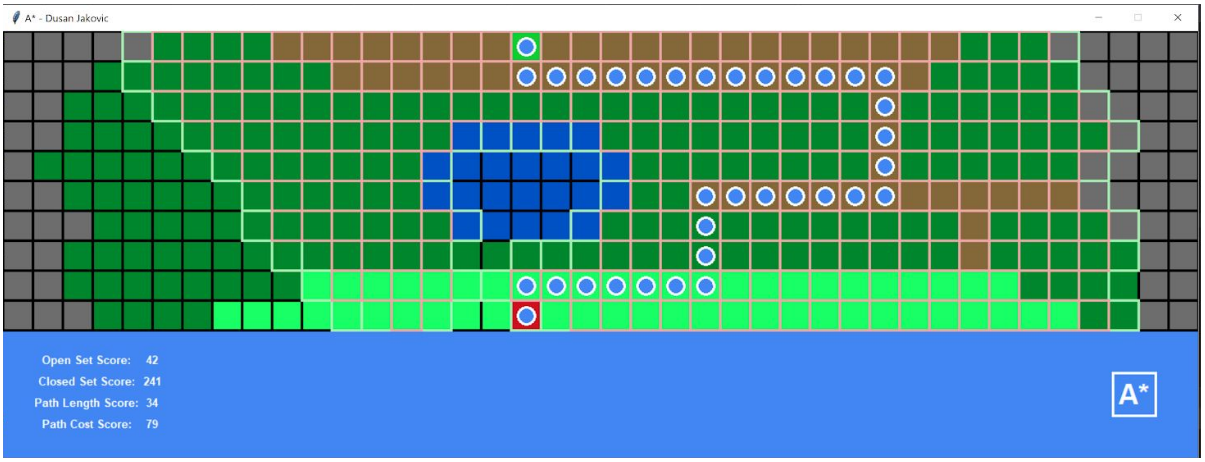


DIJKSTRA: Open Set: 7 | Closed Set: 112 | Path length: 34 | Path Cost = Path Length

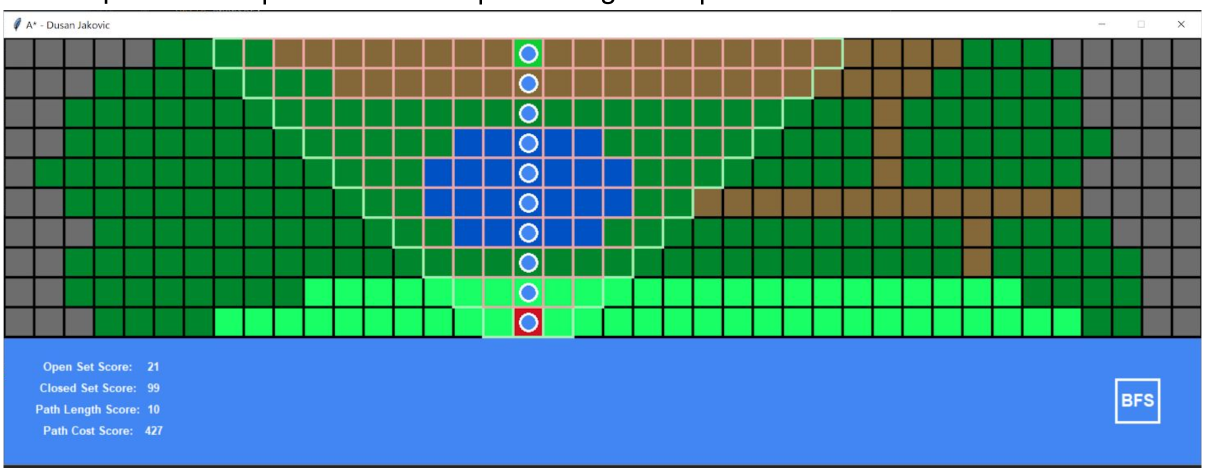


Board-2-1

A*: Open Set: 42 | Closed Set: 241 | Path length: 34 | Path Cost: 79



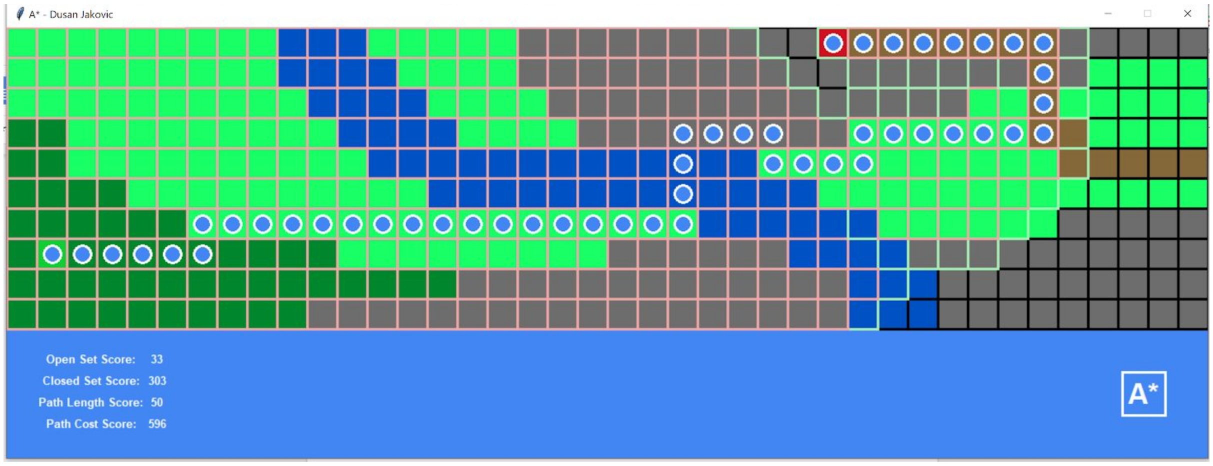
BFS: Open Set: 21 | Closed Set: 99 | Path length: 10 | Path Cost: 427



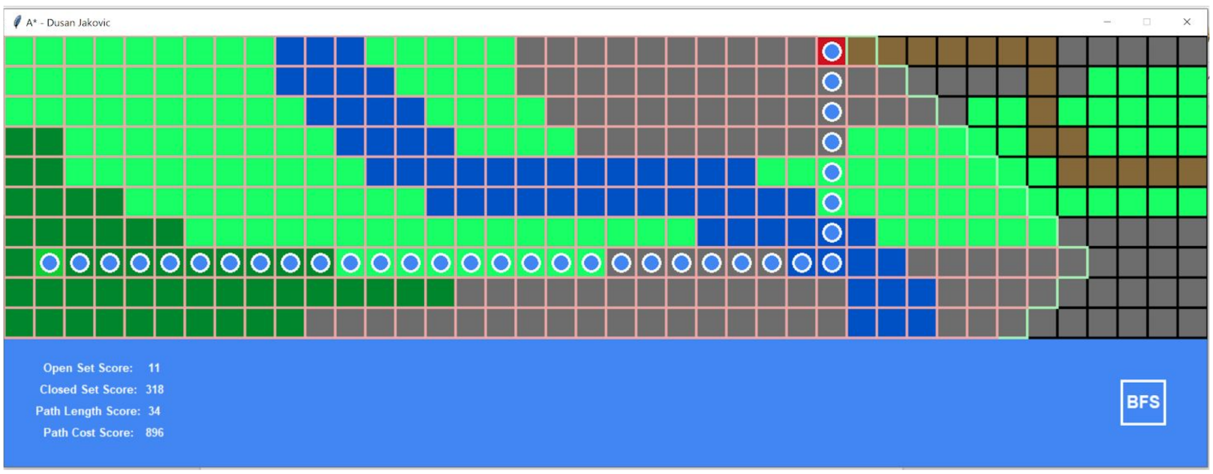
Dijkstra: Open Set: 74 | Closed Set: 95 | Path length: 40 | Path Cost: 103

Board-2-3

A*: Open Set: 33 | Closed Set: 303 | Path length: 50 | Path Cost: 596



BFS: Open Set: 11 | Closed Set: 318 | Path length: 38 | Path Cost: 896



DIJKSTRA: Open Set: 63 | Closed Set: 281 | Path length: 54 | Path Cost: 741

3.3 Board 1-1 and 1-2

Looking at the results, it’s clear that A* is the most efficient, as expected. The final paths are all the same, but the amount of open and closed set differ quite a lot between A* and the two other.

Looking at the first board; A* has 4 times less cells in the closed set compared to BFS and Dijkstra. A* has double the amount of cells in the open set than the two other algorithms. The differences between these three are similar in the other boards as well. A* star has more cells in the open set and less in the closed set, whilst the other two are the opposite. BFS and Dijkstra are identical in these cases.

3.3 Board 2-1 and 2-3

As in the previous comparison, A* star is the winner by looking at the path cost. The A* star algorithm may have bigger open set or closed set, or a longer path (length) than BFS/Dijkstra, but it will eventually find a optimal path.

BFS will always find the shortest path, but that has an extreme effect on the cost. Dijkstra may evaluate less cells than A* but the cost is slightly higher.

Source Code: Part 1

The source code may be a bit messy because everything is inside one file. A* is the default method that is used. BFS and Dijkstra are commented out in the code

```python
from graphics import *
import time

square_size = 50

walking_space = '.'
obstacle_type = '#'
start_point = 'A'
goal_point = 'B'

openSet = []
closeSet = []
start = []
goal = []
cellGrid = []
path = []

# The Map File
filepath = 'maps/board-1-2.txt'

#GUI
win = GraphWin("A* - Dusan Jakovic", 1000, 500)
win.setBackground(color_rgb(66, 134, 244))

openSet_text = Text(Point(100,405), "Open Set Score: ")
openSet_text.setTextColor(color="white")
openSet_text.setSize(10)
openSet_text.setStyle("bold")
openSet_text.draw(win)

openSet_score = Text(Point(175,405), openSet.__len__())
openSet_score.setTextColor(color="white")
openSet_score.setSize(10)
openSet_score.setStyle("bold")
openSet_score.draw(win)

closedSet_text = Text(Point(100,430), "Closed Set Score: ")
closedSet_text.setTextColor(color="white")
closedSet_text.setSize(10)
closedSet_text.setStyle("bold")
closedSet_text.draw(win)

closedSet_score = Text(Point(175,430), closeSet.__len__())
closedSet_score.setTextColor(color="white")
closedSet_score.setSize(10)
closedSet_score.setStyle("bold")
closedSet_score.draw(win)

pathLength_text = Text(Point(100,455), "Path Length Score: ")
pathLength_text.setTextColor(color="white")
pathLength_text.setSize(10)
pathLength_text.setStyle("bold")
pathLength_text.draw(win)

pathLength_text = Text(Point(175,455),'')
pathLength_text.setTextColor(color="white")
pathLength_text.setSize(10)
pathLength_text.setStyle("bold")
pathLength_text.draw(win)

a_rect = Rectangle(Point(900, 400), Point(950, 450))
a_rect.setOutline(color="white")
a_rect.setWidth(3)
a_rect.draw(win)

Astar_text = Text(Point(925,425), "A*")
Astar_text.setSize(25)
# Astar_text = Text(Point(925,425), "BFS")
# Astar_text.setSize(15)
# Astar_text = Text(Point(925,425), "DiJ")
# Astar_text.setSize(15)
Astar_text.setTextColor(color="white")
#Astar_text.setTextColor(color_rgb(66, 134, 244))
Astar_text.setStyle("bold")
Astar_text.draw(win)

# for updating cell-color. Keeps track of already updated cells. Updates only if not already updated
open_update = []
closed_update = []

# Cell class that includes coordinates, type, g-value, h-value, f-value, neighbors and the parent
# It includes methods:
#                       - addNeighbors() to add 0-4 neighbors to each cell
#                       - getNeighbors() return 0-4 neighbors
#                       - rectangle() represents the cell in the grid
#                       - hScoreText() is the cells h-value
class Cell():
   def __init__(self, i, j, type):
       self.i = i
       self.j = j
       self.g = 0
       self.h = 0
       self.f = 0
       self.type = type
       self.parent = None
       self.neighbors = []

   def addNeighbors(self, i, j):
       if i < cellGrid.__len__() - 1:
           if cellGrid[i + 1][j].type != obstacle_type:
               self.neighbors.append(cellGrid[i + 1][j])
       if i > 0:
           if cellGrid[i - 1][j].type != obstacle_type:
               self.neighbors.append(cellGrid[i - 1][j])
       if j < cellGrid[i].__len__() - 1:
           if cellGrid[i][j + 1].type != obstacle_type:
               self.neighbors.append(cellGrid[i][j + 1])
       if j > 0:
           if cellGrid[i][j - 1].type != obstacle_type:
               self.neighbors.append(cellGrid[i][j - 1])

   def getNeighbors(self):
       return self.neighbors

   def rectangle(self, i, j):
       rect = Rectangle(Point(j * square_size, i * square_size),
                        Point(j * square_size + square_size, i * square_size + square_size))
       if self.type is walking_space:
           rect.setFill(color='white')
       if self.type is obstacle_type:
           rect.setFill(color_rgb(10, 10, 10))
       if self.type is start_point:
           rect.setFill(color_rgb(18, 206, 53))
       if self.type is goal_point:
           rect.setFill(color_rgb(206, 18, 34))
       return rect

   def hScoreText(self, i, j):
```

```
        displayScore = manhattan(self, goal)
        text = Text(Point(cellGrid[i][j].j * square_size + square_size / 2,
                          cellGrid[i][j].i * square_size + square_size / 2), displayScore)
        text.setSize(10)
        return text

# Manhattan - Heuristic function
def manhattan(point, point2):
    return abs(point.i - point2.i) + abs(point.j - point2.j)


# Parse map into grid array specified by the text-file
def parseMap():
    with open(filepath) as mapFile:
        mapArray = mapFile.readlines()
    return mapArray

# TODO: Clean up. Put this in parseMap function
# Populate elements with Cells specified in the
map = parseMap()
for i in range(map.__len__()):
    subGrid = []
    cellGrid.append(subGrid)
    for j in range(map[i].__len__() - 1):
        if map[i][j] == obstacle_type:
            subGrid.append(Cell(i, j, obstacle_type))
        elif map[i][j] is start_point:
            c = Cell(i, j, start_point)
            start = c
            subGrid.append(c)
        elif map[i][j] == goal_point:
            c = Cell(i, j, goal_point)
            goal = c
            subGrid.append(c)
        else:
            c = Cell(i, j, walking_space)
            subGrid.append(c)
openSet.append(start)

# give the cekks neighbors
def create_neighbors():
    for i in range(cellGrid.__len__()):
        for j in range(cellGrid[i].__len__()):
            cellGrid[i][j].addNeighbors(i, j)

# Draws the board with the cells
def draw_board():
    for y in range(cellGrid.__len__()):
        for x in range(cellGrid[y].__len__()):
            cell = cellGrid[y][x]
            rect = cell.rectangle(y, x)
            rect.draw(win)
            text = cell.hScoreText(y, x)
            text.draw(win)
    win.getMouse()

# Updates the board by the number of seconds specified by the argument
def update_board(second_per_update):
    # A*
    current = min(openSet, key=lambda o: o.f)
    # BFS
    # current = openSet[0]
    # DIJKSTRA
    # current = min(openSet, key=lambda o: o.g)
    for i in openSet:
        if i not in open_update:
            rect = i.rectangle(i.i, i.j)
            rect.setFill(color_rgb(167, 239, 180))
            rect.draw(win)
            open_update.append(i)
            text = i.hScoreText(i.i, i.j)
            text.setOutline(color_rgb(19, 10, 200))
            text.setStyle("bold")
            text.draw(win)
        dot = Circle(Point(current.j * square_size + square_size / 2, current.i * square_size + square_size / 2), 10)
        dot.setOutline(color_rgb(19, 10, 200))
        dot.draw(win)
    for i in closeSet:
        if i not in closed_update:
            rect = i.rectangle(i.i, i.j)
            rect.setFill(color_rgb(239, 167, 167))
            rect.draw(win)
            closed_update.append(i)
            text = i.hScoreText(i.i, i.j)
            text.setOutline(color_rgb(19, 10, 200))
            text.setStyle("bold")
            text.draw(win)
    openSet_score.setText(openSet.__len__())
    closedSet_score.setText(closeSet.__len__())
    time.sleep(second_per_update)


# Draws the path after the final path is found
def draw_path(final_path):
    for i in final_path:
        dot = Circle(Point((i.j * square_size) + square_size / 2,
                           (i.i * square_size) + square_size / 2), 10)
        dot.setFill(color_rgb(66, 134, 244))
        dot.draw(win)


# A* algorithm that finds the shortest path. It includes the update_board method
# which updates the board as long as the algorithm runs
def aStar():
    while openSet:
        update_board(0)
        # A*
        current = min(openSet, key=lambda o: o.f)
        # BFS
        # current = openSet[0]
        # DIJKSTRA
        # current = min(openSet, key=lambda o: o.g)
        if current == goal:
            while current.parent:
                path.append(current)
                current = current.parent
            path.append(current)
            pathLength_text.setText(path.__len__())
            print("GOAL REACHED")
            return path[::-1]

        openSet.remove(current)
        closeSet.append(current)

        neighbors = current.neighbors
        for neighbor in neighbors:
            if neighbor not in closeSet:
                tempG = current.g + 1
                if neighbor in openSet:
                    if tempG < neighbor.g:
                        neighbor.g = tempG
                        neighbor.parent = current
                else:
                    neighbor.h = manhattan(neighbor, goal)
                    neighbor.f = neighbor.g + neighbor.h
                    neighbor.parent = current
                    openSet.append(neighbor)

    raise ValueError('--- NO PATH FOUND! ----\n --- '
                     'CHECK IF THE GOAL/END-POINT IS ACCESSIBLE')


# Runs the application
def run():
```

```
    draw_board()
    create_neighbors()
    aStar()
    draw_path(path)
    win.getMouse()

run()
```

Source Code: Part 2

The source code may be a bit messy because everything is inside one file. A* is the default method that is used. BFS and Dijkstra are commented out in the code. Comment out win.update* to skip the animation of the algorithm. It may take some seconds to complete with the animation “turned on”, because the drawing solution for updating isn’t the most optimal one.

```
from graphics import *
import time

square_size = 35

road_type = 'r'
grass_type = 'g'
forest_type = 'f'
mountain_type = 'm'
water_type = 'w'
start_point = 'A'
goal_point = 'B'

openSet = []
closeSet = []
start = []
goal = []
cellGrid = []
path = []


filepath = 'maps/board-2-1.txt'

#GUI
win = GraphWin("A* - Dusan Jakovic", 1400, 500, autoflush=False)
win.setBackground(color_rgb(66, 134, 244))

openSet_text = Text(Point(100,385), "Open Set Score: ")
openSet_text.setTextColor(color="white")
openSet_text.setSize(10)
openSet_text.setStyle("bold")
openSet_text.draw(win)

openSet_score = Text(Point(175,385), openSet.__len__())
openSet_score.setTextColor(color="white")
openSet_score.setSize(10)
openSet_score.setStyle("bold")
openSet_score.draw(win)

closedSet_text = Text(Point(100,410), "Closed Set Score: ")
closedSet_text.setTextColor(color="white")
closedSet_text.setSize(10)
closedSet_text.setStyle("bold")
closedSet_text.draw(win)

closedSet_score = Text(Point(175,410), closeSet.__len__())
closedSet_score.setTextColor(color="white")
closedSet_score.setSize(10)
closedSet_score.setStyle("bold")
closedSet_score.draw(win)

pathLength_text = Text(Point(100,435), "Path Length Score: ")
pathLength_text.setTextColor(color="white")
pathLength_text.setSize(10)
pathLength_text.setStyle("bold")
pathLength_text.draw(win)

pathLength_text = Text(Point(175,435),'')
pathLength_text.setTextColor(color="white")
pathLength_text.setSize(10)
pathLength_text.setStyle("bold")
pathLength_text.draw(win)

pathCost_text = Text(Point(100,460), "Path Cost Score: ")
pathCost_text.setTextColor(color="white")
pathCost_text.setSize(10)
pathCost_text.setStyle("bold")
pathCost_text.draw(win)

pathCost_text = Text(Point(175,460),'')
pathCost_text.setTextColor(color="white")
pathCost_text.setSize(10)
pathCost_text.setStyle("bold")
pathCost_text.draw(win)

a_rect = Rectangle(Point(1300,400),Point(1350,450))
a_rect.setOutline(color="white")
a_rect.setWidth(3)
a_rect.draw(win)

#white_rect = Rectangle(Point(950,400),Point(1000,450))
#white_rect.setOutline(color="white")
#white_rect.setFill(color="white")
#white_rect.draw(win)

Astar_text = Text(Point(1325,425), "A*")
Astar_text.setSize(25)
# Astar_text = Text(Point(1325,425), "BFS")
# Astar_text.setSize(15)
# Astar_text = Text(Point(1325,425), "DiJ")
# Astar_text.setSize(15)
Astar_text.setTextColor(color="white")
#Astar_text.setTextColor(color_rgb(66, 134, 244))
Astar_text.setStyle("bold")
Astar_text.draw(win)

#for updating cell-color
##keeps track of already updated cells. Updates only if not already updated
open_update = []
closed_update = []

class Cell():
    def __init__(self, i, j, type, cost):
        self.i = i
        self.j = j
        self.g = cost
        self.h = 0
        self.f = 0
        self.cost = cost
```

```python
        self.costSoFar = cost
        self.type = type
        self.parent = None
        self.neighbors = []

    def addNeighbors(self, i, j):
        if j < cellGrid[i].__len__() - 1:
            self.neighbors.append(cellGrid[i][j + 1])
        if j > 0:
            self.neighbors.append(cellGrid[i][j - 1])
        if i < cellGrid.__len__() - 1:
            self.neighbors.append(cellGrid[i + 1][j])
        if i > 0:
            self.neighbors.append(cellGrid[i - 1][j])


    def getNeighbors(self):
        return self.neighbors

    def rectangle(self, i, j):
        rect = Rectangle(Point(j * square_size, i * square_size),
                         Point(j * square_size + square_size, i * square_size + square_size))
        rect.setWidth(3)
        if self.type is grass_type:
            rect.setFill(color_rgb(25, 255, 102))
        if self.type is forest_type:
            rect.setFill(color_rgb(0, 135, 45))
        if self.type is water_type:
            rect.setFill(color_rgb(0, 81, 196))
        if self.type is mountain_type:
            rect.setFill(color_rgb(109, 109, 109))
        if self.type is road_type:
            rect.setFill(color_rgb(132, 104, 58))
        if self.type is start_point:
            rect.setFill(color_rgb(18, 206, 53))
        if self.type is goal_point:
            rect.setFill(color_rgb(206, 18, 34))
        return rect

    def hScoreText(self, i, j):
        displayScore = manhattan_with_cost(self,goal)
        text = Text(Point(cellGrid[i][j].j * square_size + square_size / 2,
                          cellGrid[i][j].i * square_size + square_size / 2), displayScore)
        text.setSize(10)
        return text

    def fScoreText(self, i, j):
        text = Text(Point(cellGrid[i][j].j * square_size + square_size / 2,
                          cellGrid[i][j].i * square_size + square_size / 2), self.f)
        text.setSize(10)
        return text


#Manhattan method for finding the heuristic cost (f(h)
def manhattan(point, point2):
    return abs(point.i - point2.i) + abs(point.j - point2.j)

#This euqlidian Manhattan method is used to show the h from
#current cell to the Goal, including the cost for the given cell
def manhattan_with_cost(point, point2):
    return abs(point.i - point2.i) + abs(point.j - point2.j) + point.cost

# Parse map into grid array
def parseMap():
    with open(filepath) as mapFile:
        mapArray = mapFile.readlines()
    return mapArray

map = parseMap()

#TODO: Clean up. Put this in parseMap function
# Populate elements with Cells
for i in range(map.__len__()):
    subGrid = []
    cellGrid.append(subGrid)
    for j in range(map[i].__len__() - 1):
        if map[i][j] == grass_type:
            subGrid.append(Cell(i, j, grass_type, 5))
        elif map[i][j] == forest_type:
            subGrid.append(Cell(i, j, forest_type, 10))
        elif map[i][j] == water_type:
            subGrid.append(Cell(i, j, water_type, 100))
        elif map[i][j] == mountain_type:
            subGrid.append(Cell(i, j, mountain_type, 50))
        elif map[i][j] == road_type:
            subGrid.append(Cell(i, j, road_type, 1))
        elif map[i][j] is start_point:
            c = Cell(i, j, start_point, 0)
            start = c
            subGrid.append(c)
        elif map[i][j] == goal_point:
            c = Cell(i, j, goal_point, 0)
            goal = c
            subGrid.append(c)

openSet.append(start)

# give neighbors
def create_neighbors():
    for i in range(cellGrid.__len__()):
        for j in range(cellGrid[i].__len__()):
            cellGrid[i][j].addNeighbors(i, j)


def draw_board():
    for y in range(cellGrid.__len__()):
        for x in range(cellGrid[y].__len__()):
            cell = cellGrid[y][x]
            rect = cell.rectangle(y, x)
            rect.draw(win)
    win.getMouse()


def update_board(second_per_update):
    # A*
    current = min(openSet, key=lambda o: o.f)
    # BFS
    # current = openSet[0]
    # DIJKSTRA
    # current = min(openSet, key=lambda o: o.g)
    for i in openSet:
        if i not in open_update:
            rect = i.rectangle(i.i, i.j)
            rect.setOutline(color_rgb(167, 239, 180))
            rect.setWidth(3)
            rect.draw(win)
            open_update.append(i)
            #text = i.fScoreText(i.i, i.j)
            #text.setOutline(color_rgb(19, 10, 200))
            #text.setStyle("bold")
            #text.draw(win)
        dot = Circle(Point(current.j * square_size + square_size / 2, current.i * square_size + square_size / 2), 10)
        dot.setOutline(color_rgb(66, 134, 244))
        dot.setWidth(3)
        dot.draw(win)
    for i in closeSet:
        if i not in closed_update:
            rect = i.rectangle(i.i, i.j)
            rect.setOutline(color_rgb(239, 167, 167))
            rect.setWidth(3)
            rect.draw(win)
            closed_update.append(i)
            #text = i.fScoreText(i.i, i.j)
            #text.setOutline(color_rgb(19, 10, 200))
```

```
                #text.setStyle("bold")
                #text.draw(win)
    openSet_score.setText(openSet.__len__())
    closedSet_score.setText(closeSet.__len__())
    # remove update method-call to skip animation
    win.update()
    time.sleep(second_per_update)


def draw_path(final_path):
    for i in final_path:
        dot = Circle(Point((i.j * square_size) + square_size / 2,
                           (i.i * square_size) + square_size / 2), 10)
        dot.setFill(color_rgb(66, 134, 244))
        dot.setOutline(color="white")
        dot.setWidth(3)
        dot.draw(win)

def aStar():
    while openSet:
        update_board(0)
        # A* STAR
        current = min(openSet, key=lambda o: o.f)
        # BFS
        # current = openSet[0]
        # DIJKSTRA
        # current = min(openSet, key=lambda o: o.g)
        if current == goal:
            while current.parent:
                path.append(current)
                current = current.parent
            path.append(current)
            pathLength_text.setText(path.__len__())
            #Initilized as 1 becaus goal is in the path
            path_cost = 1
            for cell in path:
                path_cost += cell.costSoFar
            pathCost_text.setText(path_cost)
            print("GOAL REACHED")
            return path[::-1]

        openSet.remove(current)
        closeSet.append(current)

        if current.parent:
            current.cost = current.parent.cost + current.cost

        neighbors = current.neighbors
        for neighbor in neighbors:
            if neighbor not in closeSet:
                tempG = current.g + current.cost
                if neighbor in openSet:
                    if tempG < neighbor.g:
                        neighbor.g = tempG
                        neighbor.parent = current
                else:
                    neighbor.h = manhattan(neighbor, goal)
                    neighbor.f = neighbor.g + neighbor.h + current.cost
                    neighbor.parent = current
                    openSet.append(neighbor)
    raise ValueError('--- NO PATH FOUND! ----\n --- '
                     'CHECK IF THE GOAL/END-POINT IS ACCESSIBLE')


def run():
    draw_board()
    create_neighbors()
    aStar()
    draw_path(path)
    win.getMouse()

run()
```