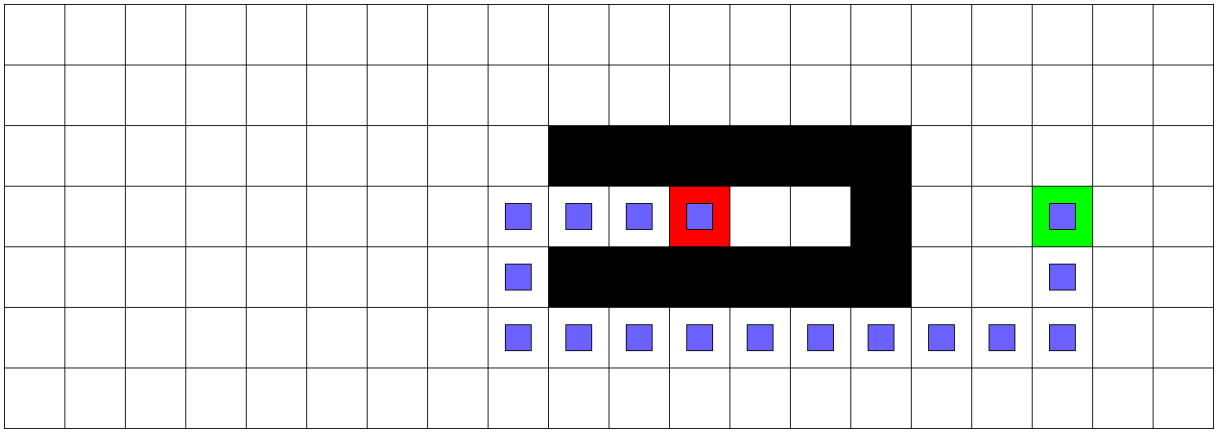
Øving 3 TDT4136

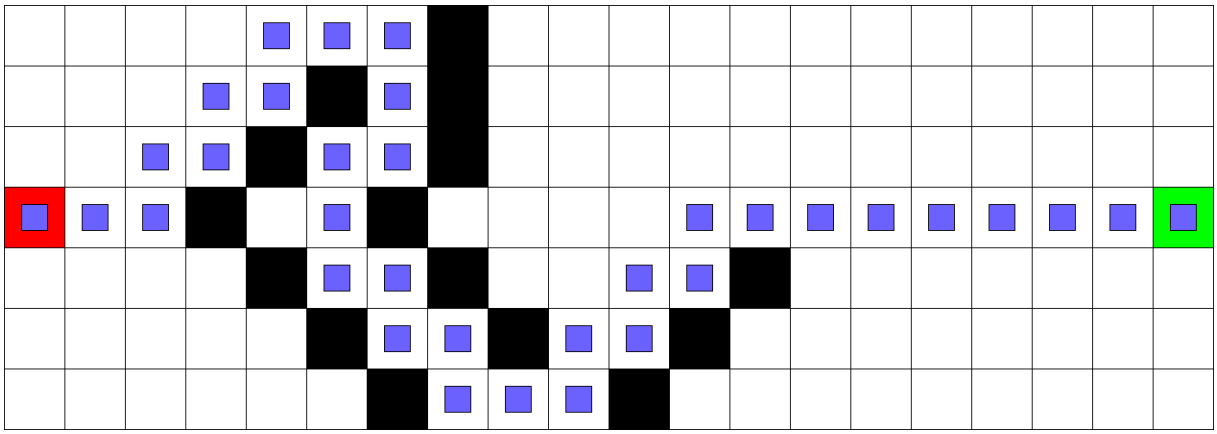
Sondre Foslien, NTNU October 6, 2017

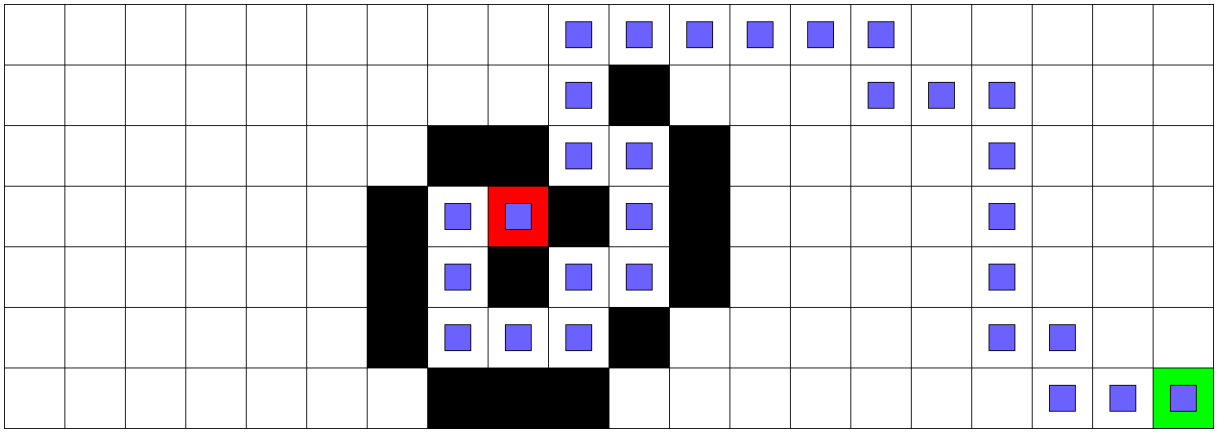
Task 1:

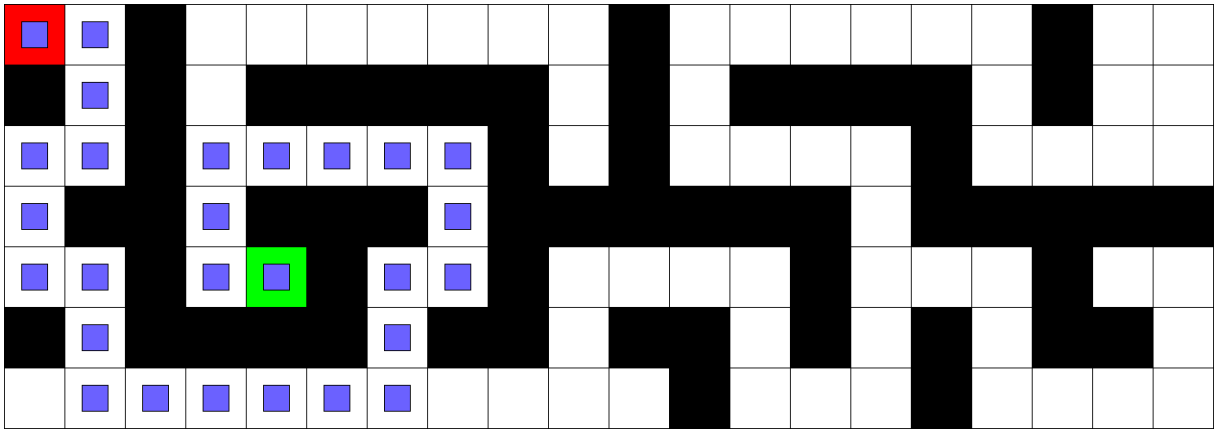
```python
from PIL import Image, ImageDraw
from heapq import heappush, heappop


class Tile:     #tile class for every tile generated
    xPos = 0
    yPos = 0
    g = 0
    h = 0
    f = 0
    parent = None

    def __init__(self, x, y, parent):       #initialize parent and coordinates
        self.xPos = x
        self.yPos = y
        self.parent = parent

    def __cmp__(self, rhs):     #overload cmp operator, compares only f value
        if rhs == None: return False
        if self.f > rhs.f: return 1
        if self.f == rhs.f: return 0
        if self.f < rhs.f: return -1

    def __eq__(self, rhs):      ##overload == operator to compare x and y coordinates
        if rhs == None: return False
        if self.xPos == rhs.xPos and self.yPos == rhs.yPos: return True
        return False

    def manhattanDist(self, end):       #calculates manhattan dist from self to given tile
        #calculates |x_2 - x_1| + |y_2 - y_1|
        return abs(self.xPos - end.xPos) + abs(self.yPos - end.yPos)



class Board:
    x, y = 0,0                          #initialize variables to save board size
    board = []                          #initialize board for saving input file

    def __init__(self, filename):

        file = open(filename, 'r')          #open boardfile
        for line in file:                   #iterate trought file
            l = list(line[:-1])             #make string into list & removing the last \n
            if self.x == 0: self.x = len(l) #if x is not set it is set to the length of input list
            if 'A' in l:                    #if there is an element in input list with value A save as startTile
                self.startTile =Tile(l.index('A'),(len(self.board)), None)
            if 'B' in l:                    #if there is element in input list with value B save as endTile
                self.endTile = Tile(l.index('B'),(len(self.board)), None)
            self.board.append(l)            #append the input list to board when finished investigating it
            self.y+=1                       #increment y for every input list

    def aStar(self):
        c, o = [], []                       #initialize closed and open lists

        #give startTile right h and f value, and append to open list
        self.startTile.h = self.startTile.manhattanDist(self.endTile)
        self.startTile.f = self.startTile.h + self.startTile.g
        o.append(self.startTile)

        while o:                            #while there is elements in open-list
            x = heappop(o)                  #pop the best element(smallest f) from
            if x in c: continue             #if element is already closed continue
            heappush(c, x)                  #push element into closed list

            #if we are at endTile we are finished, return good values
```

```
            if x == self.endTile: return x, True
            succ = self.genSucc(x)          #find successors to x
            for s in succ:                  #for all successors
                if s in c: continue         #if already closed continue
                g = x.g + 1                 #increment g value from parent

                #if s not opened or has better value than already existing opened s
                if s not in o or g < o[o.index(s)].g:
                    s.g = g                 #update g and calculate new h
                    s.h = s.manhattanDist(self.endTile)
                    s.f = s.g + s.h         #update h
                    if s not in o:          #if not opened push into heap at right place
                        heappush(o, s)
        return None, False                  #if we didn't reach endTile return false


    def genSucc(self, current):
        succ = []                           #initialize successor list
        x, y = current.xPos, current.yPos   #get current coordinates

        #array for iterate over successors
        array = [(x-1, y), (x+1,y), (x, y-1), (x, y+1)]
        for x_,y_ in array:                 #iterate through all sucessor coordinates

            #if inside board and not a wall append to successor list
            if 0 <= x_ < self.x and 0 <= y_ < self.y:
                if self.board[y_][x_] == '#': continue
                succ.append(Tile(x_, y_, current))
        return succ

    def reconstructPath(self, current):
        #reconstruct path by starting at a current tile
        path = [(current.xPos, current.yPos)]

        while current.parent:  #and working your way backwards trought parents until there is none
            path.append((current.parent.xPos,current.parent.yPos))
            current = current.parent
        return path

    def getColor(self, symbol):
        #returns color value based on which symbol
        if symbol == "A":
            return (255, 0, 0)
        elif symbol == "B":
            return (0, 255, 0)
        elif symbol == ".":
            return (255, 255, 255)
        elif symbol == "#":
            return (0, 0, 0)

    def printBoardGraphics(self, path):
        img = Image.new("RGB", (70*self.x+1, 70*self.y+1), "white")         #draw background image with 70*x by 70*y
      resolution
        idraw = ImageDraw.Draw(img)                                     #make possible to draw
        for x in range(0, len(self.board)):                             #iterate trough all elements in board
            for y in range(0, len(self.board[0])):
                c = self.getColor(self.board[x][y])                     #get color of current element
                #draw 70x70 rectangle with right color and a black outline
                idraw.rectangle([(y*70,x*70),(y*70+70,x*70+70)], fill=c, outline=(0,0,0))

                #if coordinate is in path draw smaller square on top to show path
                if (y,x) in path:
                    c = (107,97,255)
                    idraw.rectangle([(y*70+20,x*70+20),(y*70+50,x*70+50)], fill=c, outline=(0,0,0))
        img.save("pictures/board-1-3.png")                                      #save board


def main():
    #make board, run aStar and save graphics
    a = Board("boards/board-1-3.txt")
    current, success = a.aStar()
    if success:
        print "success"
        path = a.reconstructPath(current)
        print path
        a.printBoardGraphics(path)

if __name__ == "__main__":
    main()
```





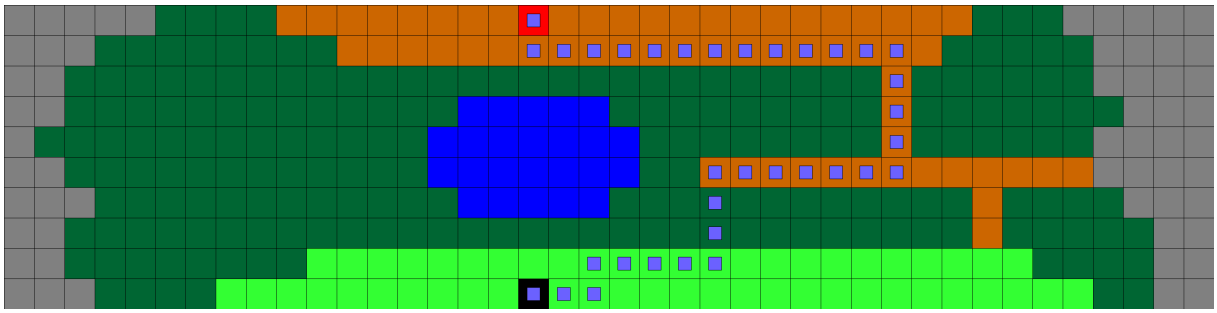


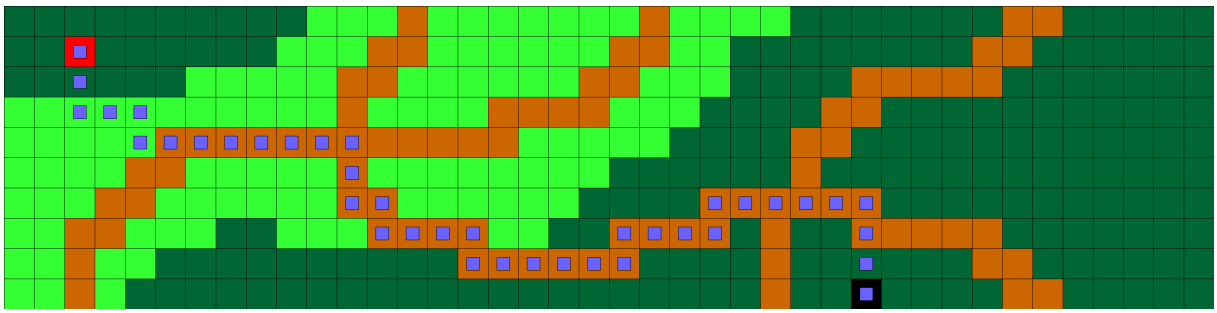


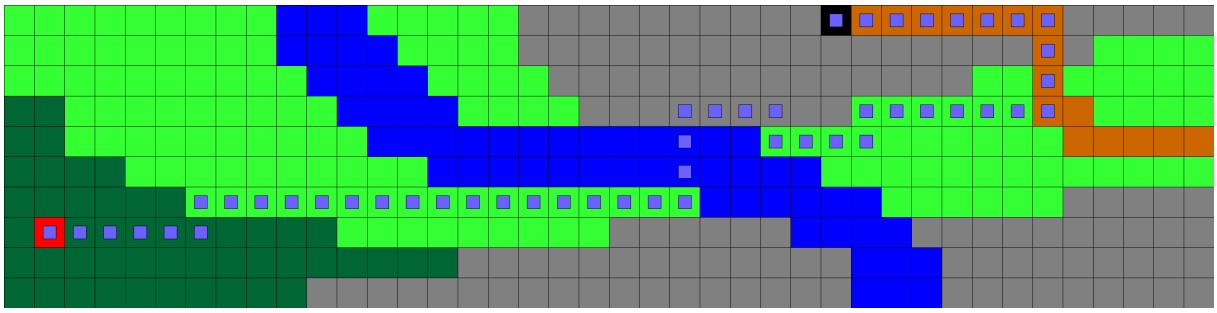
Task 2:

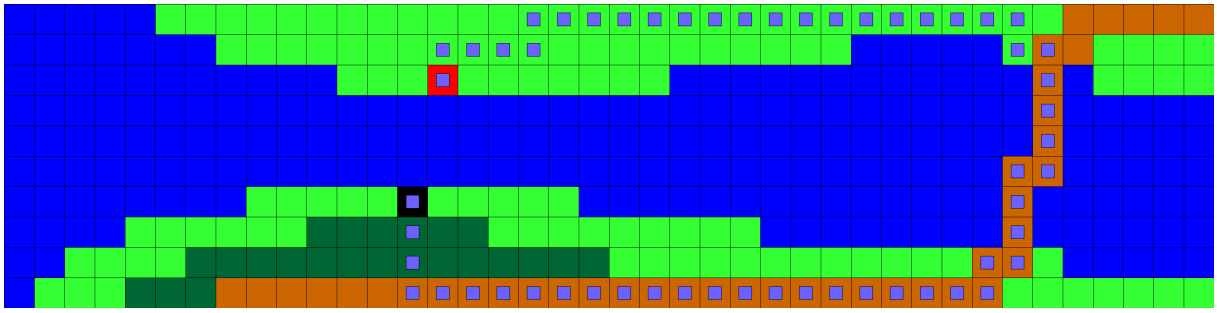
```
from PIL import Image, ImageDraw
from heapq import heappush, heappop


class Tile:     #tile class for every tile generated
    xPos = 0
    yPos = 0
    g = 0
    h = 0
    f = 0
    cost = 0
    parent = None

    def __init__(self, x, y, parent, cost):     #initialize parent and coordinates
        self.xPos = x
        self.yPos = y
        self.parent = parent
        self.cost = cost

    def __cmp__(self, rhs):     #overload cmp operator, compares only f value
        if rhs == None: return False
        if self.f > rhs.f: return 1
        if self.f == rhs.f: return 0
        if self.f < rhs.f: return -1

    def __eq__(self, rhs):      ##overload == operator to compare x and y coordinates
        if rhs == None: return False
        if self.xPos == rhs.xPos and self.yPos == rhs.yPos: return True
        return False

    def manhattanDist(self, end):       #calculates manhattan dist from self to given tile
        #calculates |x_2 - x_1| + |y_2 - y_1|
        return abs(self.xPos - end.xPos) + abs(self.yPos - end.yPos)


class Board:
    x, y = 0,0                          #initialize variables to save board size
    board = []                          #initialize board for saving input file

    def __init__(self, filename):
        file = open(filename, 'r')          #open boardfile
        for line in file:                   #iterate trought file
            l = list(line[:-1])             #make string into list & removing the last \n
            if self.x == 0: self.x = len(l) #if x is not set it is set to the length of input list
            if 'A' in l:                    #if there is an element in input list with value A save as startTile
                self.startTile =Tile(l.index('A'),(len(self.board)), None, 0)
            if 'B' in l:                    #if there is element in input list with value B save as endTile
                self.endTile = Tile(l.index('B'),(len(self.board)), None, 0)
            self.board.append(l)            #append the input list to board when finished investigating it
            self.y+=1                       #increment y for every input list

    def aStar(self):
        c, o = [], []                       #initialize closed and open lists

        #give startTile right h and f value, and append to open list
        self.startTile.h = self.startTile.manhattanDist(self.endTile)
        self.startTile.f = self.startTile.h + self.startTile.g
        o.append(self.startTile)

        while o:                            #while there is elements in open-list
            x = heappop(o)                  #pop the best element(smallest f) from
            if x in c: continue             #if element is already closed continue
            heappush(c, x)                  #push element into closed list

            #if we are at endTile we are finished, return good values
            if x == self.endTile: return x, True
            succ = self.genSucc(x)          #find successors to x
            for s in succ:                  #for all successors
                if s in c: continue         #if already closed continue
                g = x.g + s.cost            #calculate value of g for successor

                #if s not opened or has better value than already existing opened s
                if s not in o or g < o[o.index(s)].g:
                    s.g = g                 #update g and calculate new h
                    s.h = s.manhattanDist(self.endTile)
                    s.f = s.g + s.h         #update h
                    if s not in o:          #if not opened push into heap at right place
```

```
                        heappush(o, s)
        return None, False                  #if we didn't reach endTile return false

    def genSucc(self, current):
        succ = []                           #initialize successor list
        x, y = current.xPos, current.yPos   #get current coordinates

        #array for iterate over successors
        array = [(x-1, y), (x+1,y), (x, y-1), (x, y+1)]
        for x_,y_ in array:                 #iterate through all sucessor coordinates

            #if inside board get cost associated with tile and append to successor list
            if 0 <= x_ < self.x and 0 <= y_ < self.y:
                cost = self.getCost(self.board[y_][x_])
                succ.append(Tile(x_, y_, current, cost))
        return succ

    def reconstructPath(self, current):
        #reconstruct path by starting at a current tile
        path = [(current.xPos, current.yPos)]

        while current.parent:  #and working your way backwards trought parents until there is none
            path.append((current.parent.xPos,current.parent.yPos))
            current = current.parent
        return path

    def getColor(self, symbol):
        #returns color value based on which symbol
        if symbol == "A": return (255,0,0)
        elif symbol == "B": return (0,0,0)
        elif symbol == "w": return (0,0,255)
        elif symbol == "m": return (128,128,128)
        elif symbol == "f": return (0,102,51)
        elif symbol == "g": return (51,255,51)
        elif symbol == "r": return (204,102,0)
        elif symbol == "#": return (0,0,0)

    def getCost(self, symbol):
        #returns cost based on input symbol
        if symbol == "w": return 100
        elif symbol == "m": return 50
        elif symbol == "f": return 10
        elif symbol == "g": return 5
        elif symbol == "r": return 1
        elif symbol == "B" or symbol == "A" or symbol ==".." or symbol == "#": return 1

    def printBoardGraphics(self, path):
        img = Image.new("RGB", (70*self.x, 70*self.y), "white")         #draw background image with 70*x by 70*y resolution
        idraw = ImageDraw.Draw(img)                                      #make possible to draw
        for x in range(0, len(self.board)):                              #iterate trough all elements in board
            for y in range(0, len(self.board[0])):
                c = self.getColor(self.board[x][y])                      #get color of current element
                #draw 70x70 rectangle with right color and a black outline
                idraw.rectangle([(y*70,x*70),(y*70+70,x*70+70)], fill=c, outline=(0,0,0))

                #if coordinate is in path draw smaller square on top to show path
                if (y,x) in path:
                    c = (107,97,255)
                    idraw.rectangle([(y*70+20,x*70+20),(y*70+50,x*70+50)], fill=c, outline=(0,0,0))
        img.save("pictures/board-2-4.png")                                       #save board

def main():
    #make board, run aStar and save graphics
    a = Board("boards/board-2-4.txt")
    current, success = a.aStar()
    if success:
        path = a.reconstructPath(current)
        print path
        a.printBoardGraphics(path)

if __name__ == "__main__":
    main()
```









Task 3:

Task 3:.1 Astar

```
from PIL import Image, ImageDraw, ImageFont
from heapq import heappush, heappop


class Tile:     #tile class for every tile generated
    xPos = 0
    yPos = 0
    g = 0
    h = 0
    f = 0
    cost = 0
    parent = None

    def __init__(self, x, y, parent, cost):     #initialize parent and coordinates
        self.xPos = x
        self.yPos = y
        self.parent = parent
        self.cost = cost

    def __cmp__(self, rhs):     #overload cmp operator, compares only f value
        if rhs == None: return False
        if self.f > rhs.f: return 1
        if self.f == rhs.f: return 0
        if self.f < rhs.f: return -1

    def __eq__(self, rhs):      ##overload == operator to compare x and y coordinates
        if rhs == None: return False
        if self.xPos == rhs.xPos and self.yPos == rhs.yPos: return True
        return False

    def manhattanDist(self, end):       #calculates manhattan dist from self to given tile
        #calculates |x_2 - x_1| + |y_2 - y_1|
        return abs(self.xPos - end.xPos) + abs(self.yPos - end.yPos)


class Board:
    x, y = 0,0                          #initialize variables to save board size
    board = []                          #initialize board for saving input file

    def __init__(self, filename):
        file = open(filename, 'r')      #open boardfile
        for line in file:               #iterate trought file
            l = list(line[:-1])         #make string into list & removing the last \n
            if self.x == 0: self.x = len(l) #if x is not set it is set to the length of input list
            if 'A' in l:                #if there is an element in input list with value A save as startTile
                self.startTile =Tile(l.index('A'),(len(self.board)), None, 0)
            if 'B' in l:                #if there is element in input list with value B save as endTile
                self.endTile = Tile(l.index('B'),(len(self.board)), None, 0)
            self.board.append(l)        #append the input list to board when finished investigating it
            self.y+=1                   #increment y for every input list

    def aStar(self):
        c, o = [], []                   #initialize closed and open lists

        #give startTile right h and f value, and append to open list
        self.startTile.h = self.startTile.manhattanDist(self.endTile)
        self.startTile.f = self.startTile.h + self.startTile.g
        o.append(self.startTile)

        while o:                        #while there is elements in open-list
            x = heappop(o)              #pop the best element(smallest f) from
            if x in c: continue         #if element is already closed continue
            heappush(c, x)              #push element into closed list

            #if we are at endTile we are finished, return good values
            if x == self.endTile: return x, True, o, c
            succ = self.genSucc(x)      #find successors to x
            for s in succ:              #for all successors
                if s in c: continue     #if already closed continue
                g = x.g + s.cost        #calculate value of g for successor

                #if s not opened or has better value than already existing opened s
                if s not in o or g < o[o.index(s)].g:
                    s.g = g             #update g and calculate new h
```

```
                    s.h = s.manhattanDist(self.endTile)
                    s.f = s.g + s.h         #update h
                    if s not in o:          #if not opened push into heap at right place
                        heappush(o, s)
        return None, False                  #if we didn't reach endTile return false

    def genSucc(self, current):
        succ = []                           #initialize successor list
        x, y = current.xPos, current.yPos   #get current coordinates

        #array for iterate over successors
        array = [(x-1, y), (x+1,y), (x, y-1), (x, y+1)]
        for x_,y_ in array:                 #iterate through all sucessor coordinates

            #if inside board get cost associated with tile and append to successor list
            if 0 <= x_ < self.x and 0 <= y_ < self.y:
                if self.board[y_][x_] == '#': continue
                cost = self.getCost(self.board[y_][x_])
                succ.append(Tile(x_, y_, current, cost))
        return succ

    def reconstructPath(self, current):
        #reconstruct path by starting at a current tile
        path = [(current.xPos, current.yPos)]

        while current.parent:  #and working your way backwards trought parents until there is none
            path.append((current.parent.xPos,current.parent.yPos))
            current = current.parent
        return path

    def getColor(self, symbol):
        #returns color value based on which symbol
        if symbol == "A": return (255,0,0)
        elif symbol == "B": return (0,0,0)
        elif symbol == "w": return (0,0,255)
        elif symbol == "m": return (128,128,128)
        elif symbol == "f": return (0,102,51)
        elif symbol == "g": return (51,255,51)
        elif symbol == "r": return (204,102,0)
        elif symbol == "#": return (0,0,0)

    def getCost(self, symbol):
        #returns cost based on input symbol
        if symbol == "w": return 100
        elif symbol == "m": return 50
        elif symbol == "f": return 10
        elif symbol == "g": return 5
        elif symbol == "r": return 1
        elif symbol == "B" or symbol == "A" or symbol ==". " or symbol == "#": return 1

    def printBoardGraphics(self, path, opened, closed):
        img = Image.new("RGB", (70*self.x, 70*self.y), "white")         #draw background image with 70*x by 70*y resolution
        idraw = ImageDraw.Draw(img)                                      #make possible to draw
        for x in range(0, len(self.board)):                              #iterate trough all elements in board
            for y in range(0, len(self.board[0])):
                c = self.getColor(self.board[x][y])                      #get color of current element
                #draw 70x70 rectangle with right color and a black outline
                idraw.rectangle([(y*70,x*70),(y*70+70,x*70+70)], fill=c, outline=(0,0,0))

                #if coordinate is in path draw smaller square on top to show path
                if (y,x) in path:
                    c = (107,97,255)
                    idraw.rectangle([(y*70+20,x*70+20),(y*70+50,x*70+50)], fill=c, outline=(0,0,0))
                elif (y,x) in closed:
                    c = (0,0,0)
                    font = ImageFont.truetype("arial.ttf", size=40)
                    idraw.text([y*70+20,x*70+20],"X", c, font)
                elif (y,x) in opened:
                    c = (0,0,0)
                    font = ImageFont.truetype("arial.ttf", size=100)
                    idraw.text([y*70+15,x*70+5],"*", c, font)
        img.save("pictures/Astar board-1-4.png")                                        #save board

def main():
    #make board, run aStar and save graphics
    a = Board("boards/board-1-4.txt")
    current, success, opened, closed = a.aStar()
    o, c = [], []
    if success:
        path = a.reconstructPath(current)
```

```
        for element in opened:
            o.append((element.xPos, element.yPos))
        for element in closed:
            c.append((element.xPos, element.yPos))
        a.printBoardGraphics(path, o, c)

if __name__ == "__main__":
    main()
```

Task 3:.2 BFS

```
from PIL import Image, ImageDraw, ImageFont
from heapq import heappush, heappop


class Tile:     #tile class for every tile generated
    xPos = 0
    yPos = 0
    g = 0
    h = 0
    f = 0
    cost = 0
    parent = None

    def __init__(self, x, y, parent, cost):     #initialize parent and coordinates
        self.xPos = x
        self.yPos = y
        self.parent = parent
        self.cost = cost

    def __cmp__(self, rhs):     #overload cmp operator, compares only f value
        if rhs == None: return False
        if self.f > rhs.f: return 1
        if self.f == rhs.f: return 0
        if self.f < rhs.f: return -1

    def __eq__(self, rhs):      ##overload == operator to compare x and y coordinates
        if rhs == None: return False
        if self.xPos == rhs.xPos and self.yPos == rhs.yPos: return True
        return False

    def manhattanDist(self, end):       #calculates manhattan dist from self to given tile
        #calculates |x_2 - x_1| + |y_2 - y_1|
        return abs(self.xPos - end.xPos) + abs(self.yPos - end.yPos)


class Board:
    x, y = 0,0                          #initialize variables to save board size
    board = []                          #initialize board for saving input file

    def __init__(self, filename):
        file = open(filename, 'r')          #open boardfile
        for line in file:                   #iterate trought file
            l = list(line[:-1])             #make string into list & removing the last \n
            if self.x == 0: self.x = len(l) #if x is not set it is set to the length of input list
            if 'A' in l:                    #if there is an element in input list with value A save as startTile
                self.startTile =Tile(l.index('A'),(len(self.board)), None, 0)
            if 'B' in l:                    #if there is element in input list with value B save as endTile
                self.endTile = Tile(l.index('B'),(len(self.board)), None, 0)
            self.board.append(l)            #append the input list to board when finished investigating it
            self.y+=1                       #increment y for every input list



    def bfs(self):
        c, o = [], []                       #initialize closed and open lists

        #give startTile right h and f value, and append to open list
        self.startTile.h = self.startTile.manhattanDist(self.endTile)
        self.startTile.f = self.startTile.h + self.startTile.g
        o.append(self.startTile)

        while o:                            #while there is elements in open-list
            x = o.pop(0)                        #pop the best element(smallest f) from
            if x in c: continue             #if element is already closed continue
            c.append(x)                     #push element into closed list

            #if we are at endTile we are finished, return good values
            if x == self.endTile: return x, True, o, c
```

```
            succ = self.genSucc(x)          #find successors to x
            for s in succ:                  #for all successors
                if s in c: continue         #if already closed continue
                g = x.g + s.cost            #calculate value of g for successor

                #if s not opened or has better value than already existing opened s
                if s not in o or g < o[o.index(s)].g:
                    s.g = g                 #update g and calculate new h
                    s.h = s.manhattanDist(self.endTile)
                    s.f = s.g + s.h         #update h
                    if s not in o:          #if not opened push into heap at right place
                        o.append(s)
        return None, False                  #if we didn't reach endTile return false

    def genSucc(self, current):
        succ = []                           #initialize successor list
        x, y = current.xPos, current.yPos   #get current coordinates

        #array for iterate over successors
        array = [(x-1, y), (x+1,y), (x, y-1), (x, y+1)]
        for x_,y_ in array:                 #iterate through all sucessor coordinates

            #if inside board get cost associated with tile and append to successor list
            if 0 <= x_ < self.x and 0 <= y_ < self.y:
                if self.board[y_][x_] == '#': continue
                cost = self.getCost(self.board[y_][x_])
                succ.append(Tile(x_, y_, current, cost))
        return succ

    def reconstructPath(self, current):
        #reconstruct path by starting at a current tile
        path = [(current.xPos, current.yPos)]

        while current.parent:  #and working your way backwards trought parents until there is none
            path.append((current.parent.xPos,current.parent.yPos))
            current = current.parent
        return path

    def getColor(self, symbol):
        #returns color value based on which symbol
        if symbol == "A": return (255,0,0)
        elif symbol == "B": return (0,0,0)
        elif symbol == "w": return (0,0,255)
        elif symbol == "m": return (128,128,128)
        elif symbol == "f": return (0,102,51)
        elif symbol == "g": return (51,255,51)
        elif symbol == "r": return (204,102,0)
        elif symbol == "#": return (0,0,0)

    def getCost(self, symbol):
        #returns cost based on input symbol
        if symbol == "w": return 100
        elif symbol == "m": return 50
        elif symbol == "f": return 10
        elif symbol == "g": return 5
        elif symbol == "r": return 1
        elif symbol == "B" or symbol == "A" or symbol ==". " or symbol == "#": return 1
            return 1

    def printBoardGraphics(self, path, opened, closed):
        img = Image.new("RGB", (70*self.x, 70*self.y), "white")         #draw background image with 70*x by 70*y resolution
        idraw = ImageDraw.Draw(img)                                      #make possible to draw
        for x in range(0, len(self.board)):                              #iterate trough all elements in board
            for y in range(0, len(self.board[0])):
                c = self.getColor(self.board[x][y])                      #get color of current element
                #draw 70x70 rectangle with right color and a black outline
                idraw.rectangle([(y*70,x*70),(y*70+70,x*70+70)], fill=c, outline=(0,0,0))

                #if coordinate is in path draw smaller square on top to show path
                if (y,x) in path:
                    c = (107,97,255)
                    idraw.rectangle([(y*70+20,x*70+20),(y*70+50,x*70+50)], fill=c, outline=(0,0,0))
                elif (y,x) in closed:
                    c = (0,0,0)
                    font = ImageFont.truetype("arial.ttf", size=40)
                    idraw.text([y*70+20,x*70+20],"X", c, font)
                elif (y,x) in opened:
                    c = (0,0,0)
                    font = ImageFont.truetype("arial.ttf", size=100)
                    idraw.text([y*70+15,x*70+5],"*", c, font)
```

```
149
150         img.save("pictures/bfs board-1-4.png")                                  #save board
151
152 def main():
153     #make board, run bfs and save graphics
154     a = Board("boards/board-1-4.txt")
155     current, success, opened, closed = a.bfs()
156     o, c = [], []
157     if success:
158         path = a.reconstructPath(current)
159         for element in opened:
160             o.append((element.xPos, element.yPos))
161         for element in closed:
162             c.append((element.xPos, element.yPos))
163         a.printBoardGraphics(path, o, c)
164
165 if __name__ == "__main__":
166     main()
```

Task 3:.3 Djikstra

```
1  from PIL import Image, ImageDraw, ImageFont
2  from heapq import heappush, heappop
3
4
5  class Tile:     #tile class for every tile generated
6      xPos = 0
7      yPos = 0
8      g = 0
9      h = 0
10     f = 0
11     cost = 0
12     parent = None
13
14     def __init__(self, x, y, parent, cost):     #initialize parent and coordinates
15         self.xPos = x
16         self.yPos = y
17         self.parent = parent
18         self.cost = cost
19
20     def __cmp__(self, rhs):     #overload cmp operator, compares only f value
21         if rhs == None: return False
22         if self.g > rhs.g: return 1
23         if self.g == rhs.g: return 0
24         if self.g < rhs.g: return -1
25
26     def __eq__(self, rhs):      ##overload == operator to compare x and y coordinates
27         if rhs == None: return False
28         if self.xPos == rhs.xPos and self.yPos == rhs.yPos: return True
29         return False
30
31     def manhattanDist(self, end):       #calculates manhattan dist from self to given tile
32         #calculates |x_2 - x_1| + |y_2 - y_1|
33         return abs(self.xPos - end.xPos) + abs(self.yPos - end.yPos)
34
35
36 class Board:
37     x, y = 0,0                          #initialize variables to save board size
38     board = []                          #initialize board for saving input file
39
40     def __init__(self, filename):
41         file = open(filename, 'r')      #open boardfile
42         for line in file:               #iterate trought file
43             l = list(line[:-1])         #make string into list & removing the last \n
44             if self.x == 0: self.x = len(l) #if x is not set it is set to the length of input list
45             if 'A' in l:                #if there is an element in input list with value A save as startTile
46                 self.startTile =Tile(l.index('A'),(len(self.board)), None, 0)
47             if 'B' in l:                #if there is element in input list with value B save as endTile
48                 self.endTile = Tile(l.index('B'),(len(self.board)), None, 0)
49             self.board.append(l)        #append the input list to board when finished investigating it
50             self.y+=1                   #increment y for every input list
51
52
53
54     def djikstra(self):
55         c, o = [], []                   #initialize closed and open lists
56
57         #give startTile right h and f value, and append to open list
58         self.startTile.h = self.startTile.manhattanDist(self.endTile)
```

```
        self.startTile.f = self.startTile.h + self.startTile.g
        o.append(self.startTile)

        while o:                            #while there is elements in open-list
            x = heappop(o)                  #pop the best element(smallest f) from
            if x in c: continue             #if element is already closed continue
            heappush(c, x)                  #push element into closed list

            #if we are at endTile we are finished, return good values
            if x == self.endTile: return x, True, o, c
            succ = self.genSucc(x)          #find successors to x
            for s in succ:                  #for all successors
                if s in c: continue         #if already closed continue
                g = x.g + s.cost            #calculate value of g for successor

                #if s not opened or has better value than already existing opened s
                if s not in o or g < o[o.index(s)].g:
                    s.g = g                 #update g and calculate new h
                    s.h = s.manhattanDist(self.endTile)
                    s.f = s.g + s.h         #update h
                    if s not in o:          #if not opened push into heap at right place
                        heappush(o, s)
        return None, False                  #if we didn't reach endTile return false

    def genSucc(self, current):
        succ = []                           #initialize successor list
        x, y = current.xPos, current.yPos   #get current coordinates

        #array for iterate over successors
        array = [(x-1, y), (x+1,y), (x, y-1), (x, y+1)]
        for x_,y_ in array:                 #iterate through all sucessor coordinates

            #if inside board get cost associated with tile and append to successor list
            if 0 <= x_ < self.x and 0 <= y_ < self.y:
                if self.board[y_][x_] == '#': continue
                cost = self.getCost(self.board[y_][x_])
                succ.append(Tile(x_, y_, current, cost))
        return succ

    def reconstructPath(self, current):
        #reconstruct path by starting at a current tile
        path = [(current.xPos, current.yPos)]

        while current.parent:  #and working your way backwards trought parents until there is none
            path.append((current.parent.xPos,current.parent.yPos))
            current = current.parent
        return path

    def getColor(self, symbol):
        #returns color value based on which symbol
        if symbol == "A": return (255,0,0)
        elif symbol == "B": return (0,0,0)
        elif symbol == "w": return (0,0,255)
        elif symbol == "m": return (128,128,128)
        elif symbol == "f": return (0,102,51)
        elif symbol == "g": return (51,255,51)
        elif symbol == "r": return (204,102,0)
        elif symbol == "#": return (0,0,0)

    def getCost(self, symbol):
        #returns cost based on input symbol
        if symbol == "w": return 100
        elif symbol == "m": return 50
        elif symbol == "f": return 10
        elif symbol == "g": return 5
        elif symbol == "r": return 1
        elif symbol == "B" or symbol == "A" or symbol ==". " or symbol == "#": return 1

    def printBoardGraphics(self, path, opened, closed):
        img = Image.new("RGB", (70*self.x, 70*self.y), "white")         #draw background image with 70*x by 70*y resolution
        idraw = ImageDraw.Draw(img)                                      #make possible to draw
        for x in range(0, len(self.board)):                              #iterate trough all elements in board
            for y in range(0, len(self.board[0])):
                c = self.getColor(self.board[x][y])                      #get color of current element
                #draw 70x70 rectangle with right color and a black outline
                idraw.rectangle([(y*70,x*70),(y*70+70,x*70+70)], fill=c, outline=(0,0,0))

                #if coordinate is in path draw smaller square on top to show path
                if (y,x) in path:
                    c = (107,97,255)
```

```
                    idraw.rectangle([(y*70+20,x*70+20),(y*70+50,x*70+50)], fill=c, outline=(0,0,0))
                elif (y,x) in closed:
                    c = (0,0,0)
                    font = ImageFont.truetype("arial.ttf", size=40)
                    idraw.text([y*70+20,x*70+20],"X", c, font)
                elif (y,x) in opened:
                    c = (0,0,0)
                    font = ImageFont.truetype("arial.ttf", size=100)
                    idraw.text([y*70+15,x*70+5],"*", c, font)

        img.save("pictures/djikstra board-1-1.png")                                    #save board

def main():
    #make board, run bfs and save graphics
    a = Board("boards/board-1-1.txt")
    current, success, opened, closed = a.djikstra()
    o, c = [], []
    if success:
        path = a.reconstructPath(current)
        for element in opened:
            o.append((element.xPos, element.yPos))
        for element in closed:
            c.append((element.xPos, element.yPos))
        a.printBoardGraphics(path, o, c)

if __name__ == "__main__":
    main()
```

Board 1-3

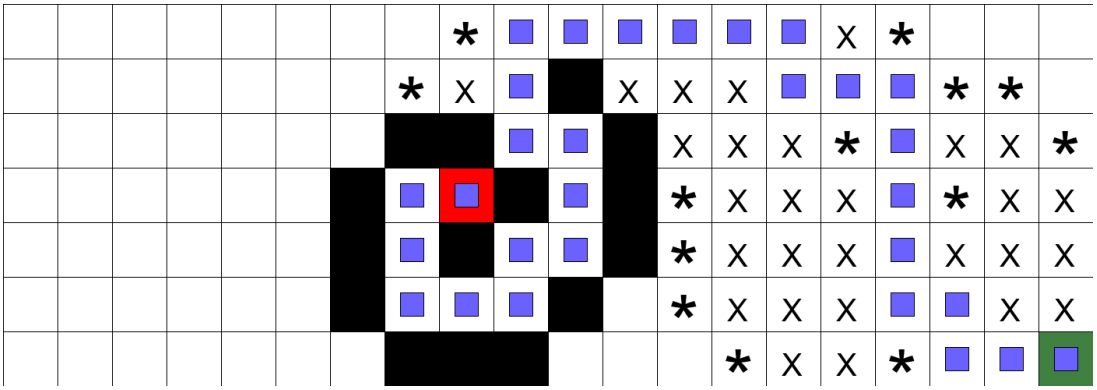


Figure 1: Board 1-3 with A* search

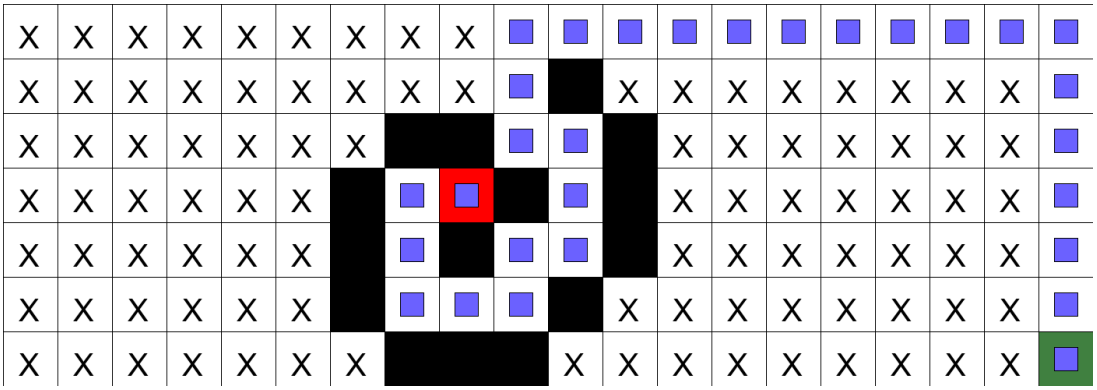


Figure 2: Board 1-3 with BFS search

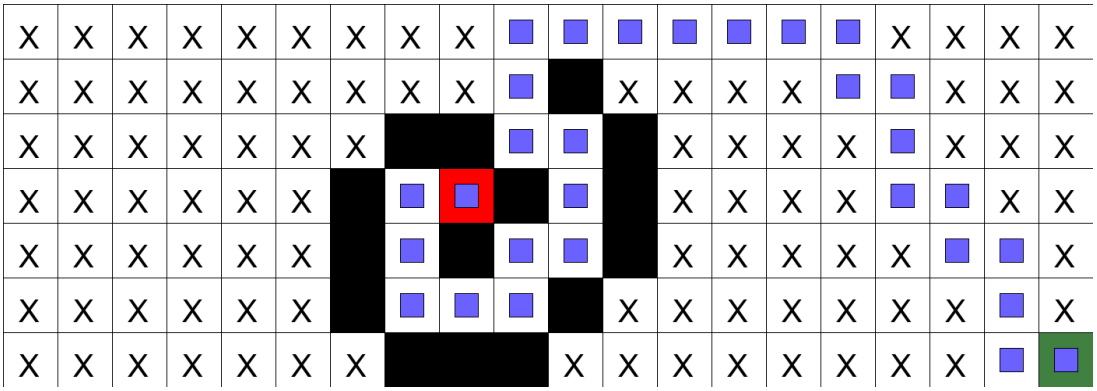


Figure 3: Board 1-3 with Djikstra search

Board 2-4

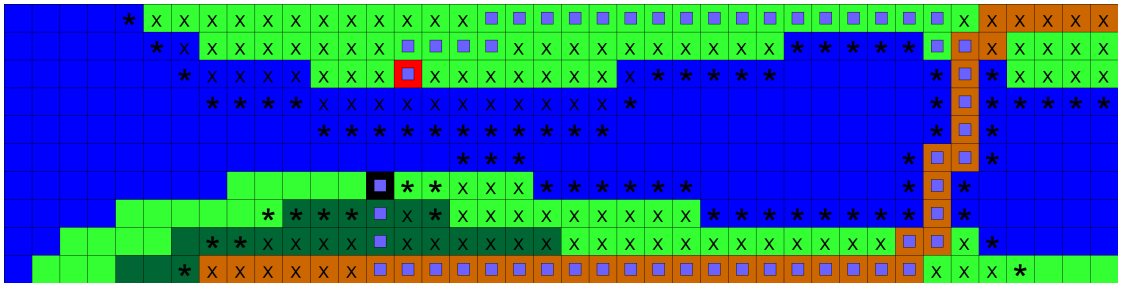


Figure 4: Board 2-4 with A* search

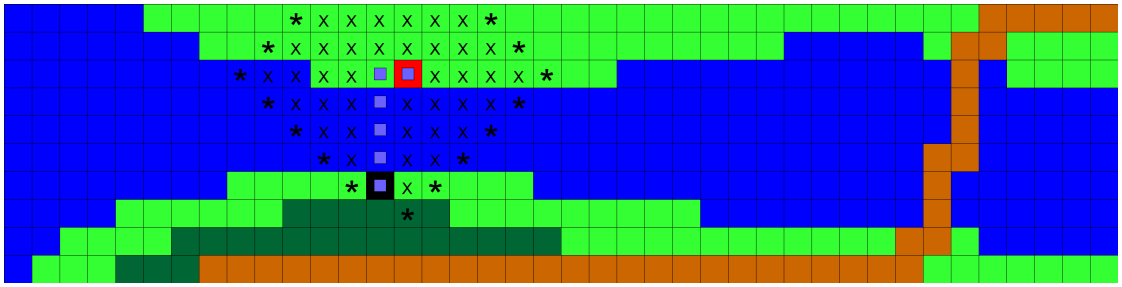


Figure 5: Board 2-4 with BFS search

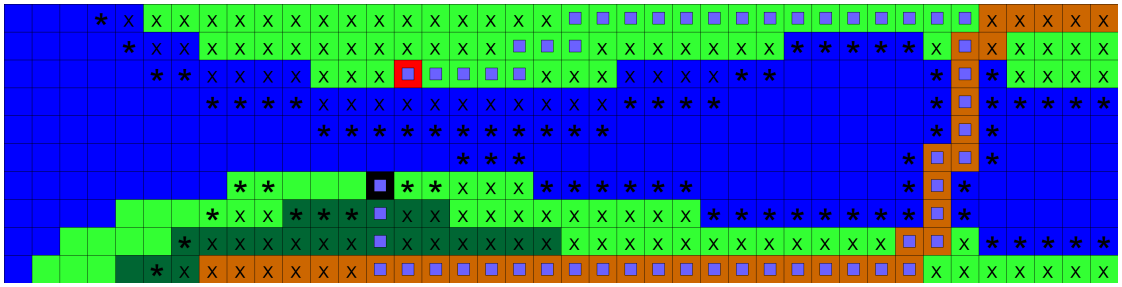


Figure 6: Board 2-4 with Djikstra search

Comments:

Board 1-3 has no weights and therefore even BFS finds a optimal route. BFS and Djikstra opens and processes a lot of more tiles than A*. This is the kind of task where A* shines. It heads directly en route to the endpoint, compared to Djikstra which basicially has found the shortest path to every tile on the map by the time it's finished. The path that Djikstra and A* finds are equally as good, they are just different because of the implimentation.

Board 2-4 is weigthed and therefore BFS does not do a good job. It finds a path to the endpoint, which is simply the first way it sees. But it is a lot more costly to go this way. Djikstra and A* basicially finds the same way. This is a task where Djistra does comparatibily a lot better than last time. The number of opened and closed tiles is about the same as A*.