AI - Øving 2

Martin Tysseland og Ernst Torsgård

September 2018

Python

```
import math
from PIL import Image


BOARDNAME = "board-1-1.txt"

IMAGENAME = "board_1_1.jpeg"

BOAR = "small"
#BOAR = "big"
ALGORITHM = "A*"
#ALGORITHM = "Dijkstras"
#ALGORITHM = "Breadth-First Search"

if BOAR == "big":
    ROWS = 10
    COLUMNS = 40+1
elif BOAR == "small":
    ROWS = 7
    COLUMNS = 20 + 1

# for Image
SCALAR = 40
SCALAR2 = 30
SCALAR3 = 15

class Node:
    child = []
    def __init__(self, F, G, H, child, parent, value, status):
        self.F = F
        self.G = G
        self.H = H
        self.child.append(child)
        self.parent = parent
        self.value = value
        self.status = status

def loadBoard(boardname, columns, rows):
    board = open(boardname, "r")
    boardMatrix = [[0 for y in range(columns)] for x in range(rows)]
```

```
    column = 0
    row = 0
    startPosition = [0, 0]
    endPosition = [0, 0]
    for line in board:
        for element in line:
            if element == 'r':
                boardMatrix[column][row] = Node(0, 0, 0, None, None, 1, None)
            elif element == '.':
                boardMatrix[column][row] = Node(0, 0, 0, None, None, 1, None)
            elif element == 'g':
                boardMatrix[column][row] = Node(0, 0, 0, None, None, 5, None)
            elif element == 'f':
                boardMatrix[column][row] = Node(0, 0, 0, None, None, 10, None)
            elif element == 'm':
                boardMatrix[column][row] = Node(0, 0, 0, None, None, 50, None)
            elif element == 'w':
                boardMatrix[column][row] = Node(0, 0, 0, None, None, 100, None)
            elif element == 'A':
                boardMatrix[column][row] = Node(0, 0, 0, None, None, 0, "start")
                startPosition[0] = column
                startPosition[1] = row
            elif element == 'B':
                boardMatrix[column][row] = Node(0, 0, 0, None, None, 0, "end")
                endPosition[0] = column
                endPosition[1] = row
            else:
                boardMatrix[column][row] = Node(0, 0, 0, None, None, math.inf, None)
            row += 1
        row = 0
        column += 1
    board.close()
    return boardMatrix, startPosition, endPosition


def printBoard(boardMatrix, width, height):
    xn = 0
    yn = 0
    img = Image.new('RGB', (width * SCALAR, height * SCALAR), (255, 255, 255))

    # make map
    for line in boardMatrix:
        for element in line:
            if element.value == 1 and BOAR == "big":
                for x in range(SCALAR):
                    for y in range(SCALAR):
                        img.putpixel((x+xn,y+yn),(139,69,19))
```

```
            elif element.value == 5:
                for x in range(SCALAR):
                    for y in range(SCALAR):
                        img.putpixel((x + xn, y + yn), (124,252,0))
            elif element.value == 10:
                for x in range(SCALAR):
                    for y in range(SCALAR):
                        img.putpixel((x + xn, y + yn), (34,139,34))
            elif element.value == 50:
                for x in range(SCALAR):
                    for y in range(SCALAR):
                        img.putpixel((x + xn, y + yn), (128,128,128))
            elif element.value == 100:
                for x in range(SCALAR):
                    for y in range(SCALAR):
                        img.putpixel((x + xn, y + yn), (0,0,255))
            elif element.status == "start":
                for x in range(SCALAR):
                    for y in range(SCALAR):
                        img.putpixel((x + xn, y + yn), (255,0,0))
            elif element.status == "end":
                for x in range(SCALAR):
                    for y in range(SCALAR):
                        img.putpixel((x + xn, y + yn), (0,255,0))
            elif element.value == math.inf:
                for x in range(SCALAR):
                    for y in range(SCALAR):
                        img.putpixel((x + xn, y + yn), (0,0,0))
            xn += SCALAR
        xn = 0
        yn += SCALAR

    xn = 0
    yn = 0

    # make path, openList and closedList
    for line in boardMatrix:
        for element in line:
            if element.status == "bestPath":
                for x in range(SCALAR - SCALAR2):
                    for y in range(SCALAR - SCALAR2):
                        img.putpixel((x+xn+SCALAR3,y+yn+SCALAR3),(0,0,0))
            elif element.status == "openList":
                for x in range(SCALAR - SCALAR2):
                    for y in range(SCALAR - SCALAR2):
                        img.putpixel((x+xn+SCALAR3,y+yn+SCALAR3),(255,215,0))
            elif element.status == "closedList":
                for x in range(SCALAR - SCALAR2):
                    for y in range(SCALAR - SCALAR2):
                        img.putpixel((x+xn+SCALAR3,y+yn+SCALAR3),(255,0,255))
            xn += SCALAR
        xn = 0
        yn += SCALAR
```

```python

    # make grid
    for x in range(width*SCALAR):
        for y in range(SCALAR, height*SCALAR, SCALAR):
            img.putpixel((x, y), (0, 0, 0))
    for x in range(SCALAR, width*SCALAR, SCALAR):
        for y in range(height*SCALAR):
            img.putpixel((x, y), (0, 0, 0))

    img.show()
    img.save(IMAGENAME)

def costFunction(boardMatrix, endPosition, child, currentPosition):
    H = abs(child[0]-endPosition[0]) + abs(child[1]-endPosition[1])
    G = boardMatrix[currentPosition[0]][currentPosition[1]].G + boardMatrix[child[0]][child[1]].value
    F = G + H
    return F, G, H

def findChild(boardMatrix, currentPosition,):
    newChildren = []
    for x in range(-1, 2, 2):
        if 0 <= currentPosition[1]+x < COLUMNS:
            if boardMatrix[currentPosition[0]][currentPosition[1]+x].value < math.inf:
                if boardMatrix[currentPosition[0]][currentPosition[1]+x].parent == None:
                    newChildren.append([currentPosition[0], currentPosition[1]+x])
    for y in range(-1, 2, 2):
        if 0 <= currentPosition[0]+y < ROWS:
            if boardMatrix[currentPosition[0]+y][currentPosition[1]].value < math.inf:
                if boardMatrix[currentPosition[0] + y][currentPosition[1]].parent == None:
                    newChildren.append([currentPosition[0]+y, currentPosition[1]])
    return newChildren

def addChildrenAndParent(newChildren, currentPosition, boardMatrix, endPosition):
    boardMatrix[currentPosition[0]][currentPosition[1]].child = newChildren
    for child in newChildren:
        F, G, H = costFunction(boardMatrix, endPosition, child, currentPosition)
        boardMatrix[child[0]][child[1]].F = F
        boardMatrix[child[0]][child[1]].G = G
        boardMatrix[child[0]][child[1]].H = H
        boardMatrix[child[0]][child[1]].child = None
        boardMatrix[child[0]][child[1]].parent = currentPosition

def checkNeighborPath(boarMatrix, currentPosition):
    gNode = boarMatrix[currentPosition[0]][currentPosition[1]].G
    for x in range(-1, 2, 2):
        if 0 <= currentPosition[1]+x < COLUMNS:
            gNeighbor = boarMatrix[currentPosition[0]][
```

```
currentPosition[1]+x].G
            hNeighbor = boarMatrix[currentPosition[0]][currentPosition[1]+x].H
            valueNeighbor = boarMatrix[currentPosition[0]][currentPosition[1]+x].value
            if gNode + valueNeighbor < gNeighbor:
                boarMatrix[currentPosition[0]][currentPosition[1]+x].parent = currentPosition
                boarMatrix[currentPosition[0]][currentPosition[1]+x].G = gNode + valueNeighbor
                boarMatrix[currentPosition[0]][currentPosition[1]+x].F = gNode + valueNeighbor + hNeighbor
                neighborPosition = [currentPosition[0], currentPosition[1]+x]
                checkNeighborPath(boarMatrix, neighborPosition)
    for y in range(-1, 2, 2):
        if 0 <= currentPosition[0]+y < ROWS:
            gNeighbor = boarMatrix[currentPosition[0]+y][currentPosition[1]].G
            hNeighbor = boarMatrix[currentPosition[0]+y][currentPosition[1]].H
            valueNeighbor = boarMatrix[currentPosition[0]+y][currentPosition[1]].value
            if gNode + valueNeighbor < gNeighbor:
                boarMatrix[currentPosition[0]+y][currentPosition[1]].parent = currentPosition
                boarMatrix[currentPosition[0]+y][currentPosition[1]].G = gNode + valueNeighbor
                boarMatrix[currentPosition[0]+y][currentPosition[1]].F = gNode + valueNeighbor + hNeighbor
                neighborPosition = [currentPosition[0]+y, currentPosition[1]]
                checkNeighborPath(boarMatrix, neighborPosition)


def pathfinding(boardMatrix, startPosition, endPosition):
    openList = []
    closedList = []
    H = abs(startPosition[0] - endPosition[0]) + abs(startPosition[1] - endPosition[1])
    currentPosition = startPosition
    boardMatrix[currentPosition[0]][currentPosition[1]].H = H
    boardMatrix[currentPosition[0]][currentPosition[1]].F = H
    openList.append(currentPosition)

    while openList:

        if ALGORITHM == "A*":
            bestF = boardMatrix[openList[-1][0]][openList[-1][1]].F
            bestNode = openList[-1]
            for node in openList:
                if bestF > boardMatrix[node[0]][node[1]].F:
                    bestF = boardMatrix[node[0]][node[1]].F
                    bestNode = node

        elif ALGORITHM == "Dijkstras":
            bestG = boardMatrix[openList[-1][0]][openList[-1][1]].G
```

```
            bestNode = openList[-1]
            for node in openList:
                if bestG > boardMatrix[node[0]][node[1]].G:
                    bestG = boardMatrix[node[0]][node[1]].G
                    bestNode = node

        elif ALGORITHM == "Breadth-First Search":
            bestNode = openList[0]

        currentPosition = bestNode
        if currentPosition == endPosition:
            break
        openList.remove(currentPosition)
        closedList.append(currentPosition)
        newChildren = findChild(boardMatrix, currentPosition)
        if startPosition in newChildren:
            newChildren.remove(startPosition)
        for child in newChildren:
            openList.append(child)
        addChildrenAndParent(newChildren, currentPosition, boardMatrix, endPosition)
        checkNeighborPath(boardMatrix, currentPosition)

    nodeList = ["start", "end"]
    for node in openList:
        if boardMatrix[node[0]][node[1]].status not in nodeList:
            boardMatrix[node[0]][node[1]].status = "openList"
    for node in closedList:
        if boardMatrix[node[0]][node[1]].status not in nodeList:
            boardMatrix[node[0]][node[1]].status = "closedList"
    path = boardMatrix[endPosition[0]][endPosition[1]].parent
    while path is not None:
        if boardMatrix[path[0]][path[1]].value > 0:
            boardMatrix[path[0]][path[1]].status = "bestPath"
        path = boardMatrix[path[0]][path[1]].parent
    return boardMatrix



def main():

    boardMatrix, startPosition, endPosition = loadBoard(BOARDNAME, COLUMNS, ROWS)

    boardMatrix = pathfinding(boardMatrix, startPosition, endPosition)

    printBoard(boardMatrix, COLUMNS, ROWS)

main()
```

Tabell

Fargekoder			
Farge	Bokstav	Forklaring	Verdi
	A	Start	0
	B	Slutt	0
		OpenList	
		ClosedList	
	r	Vann	1
	g	Fjell	5
	f	Skog	10
	m	Gress	50
	w	Vei	100
	#	Vegg	inf

Part 1

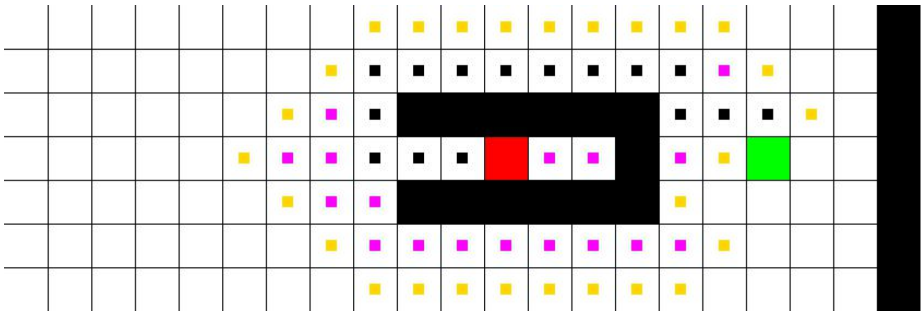

Figure 1: Board-1-1 med A*

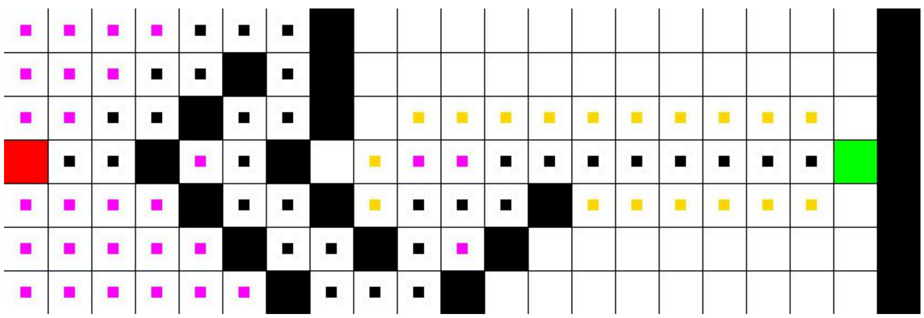

Figure 2: Board-1-2 med A*

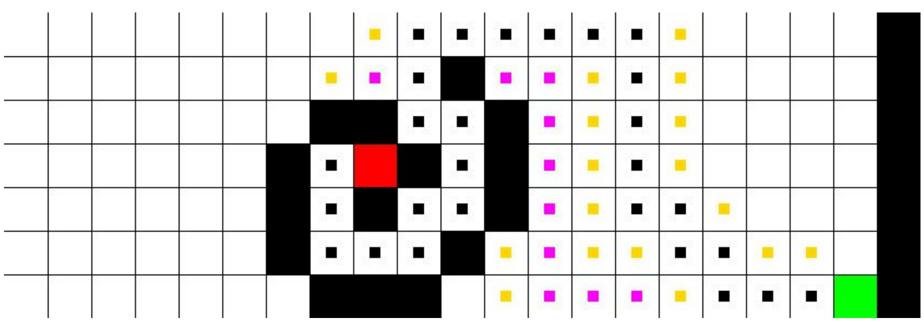

Figure 3: Board-1-3 med A*

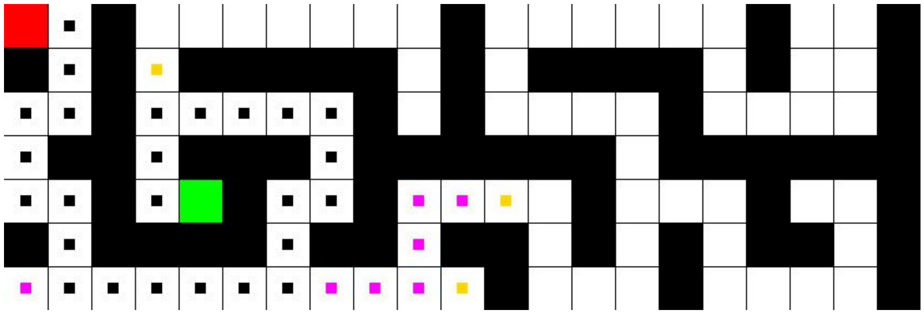

Figure 4: Board-1-4 med A*

Part 2

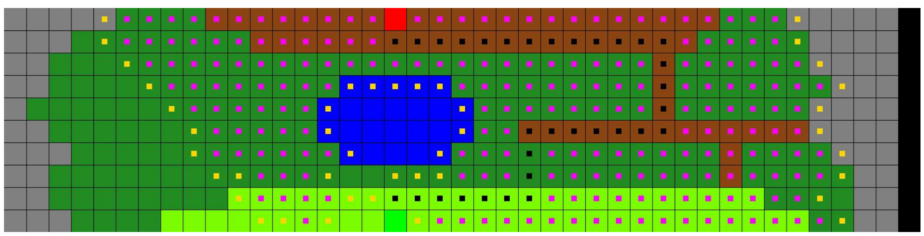

Figure 5: Board-2-1 med A*

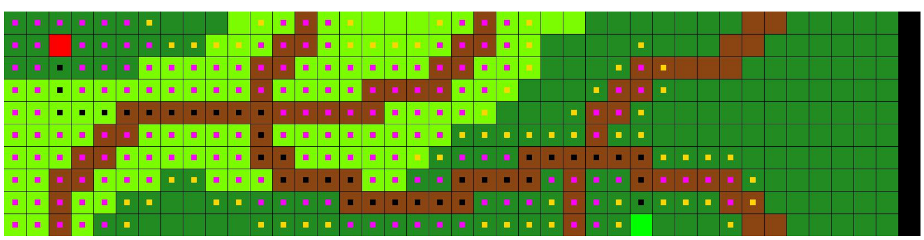

Figure 6: Board-2-2 med A*

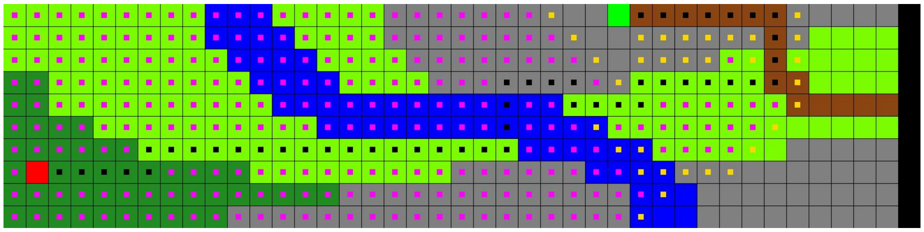

Figure 7: Board-2-3 med A*

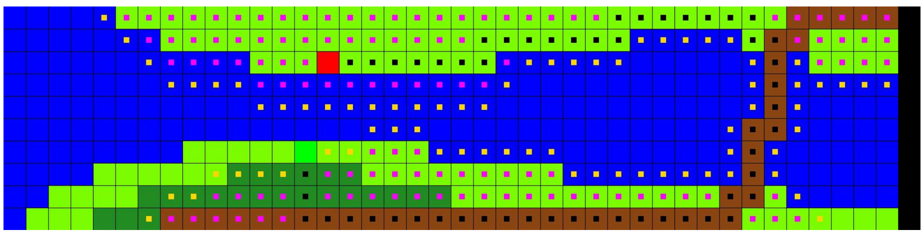

Figure 8: Board-2-4 med A*

Part 3

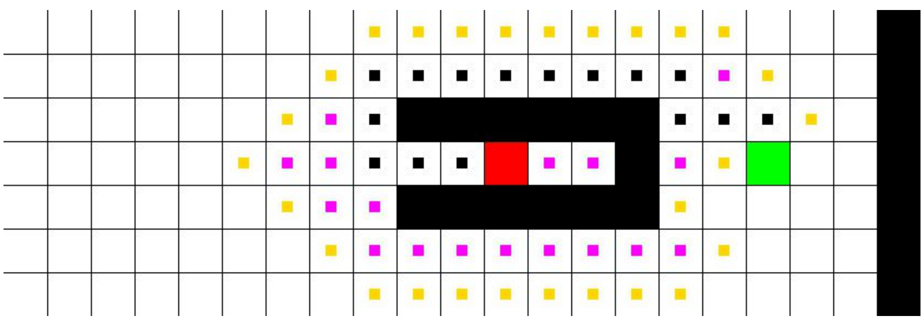

Figure 9: Board-1-1 med A*

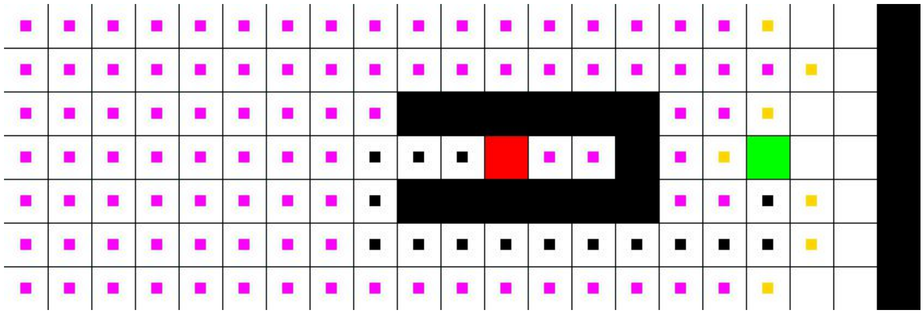

Figure 10: Board-1-1 med Dijkstra's

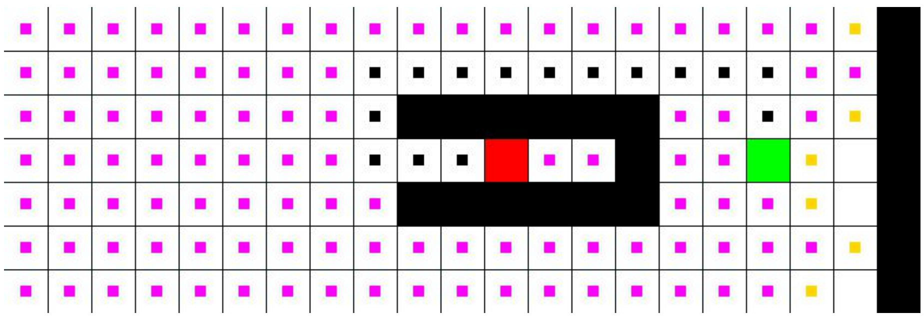

Figure 11: Board-1-1 med Breadth-First Search

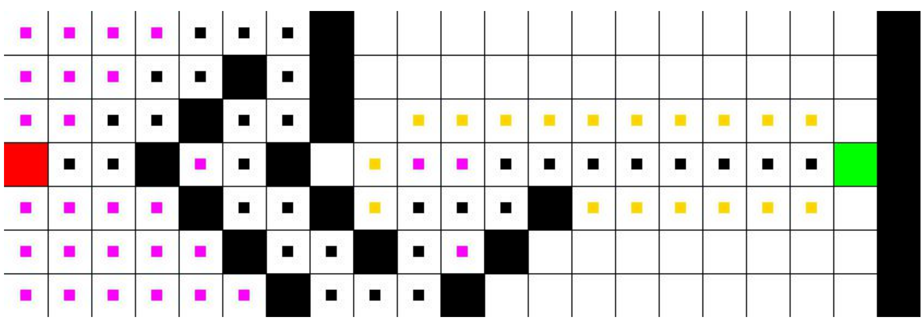

Figure 12: Board-1-2 med A*

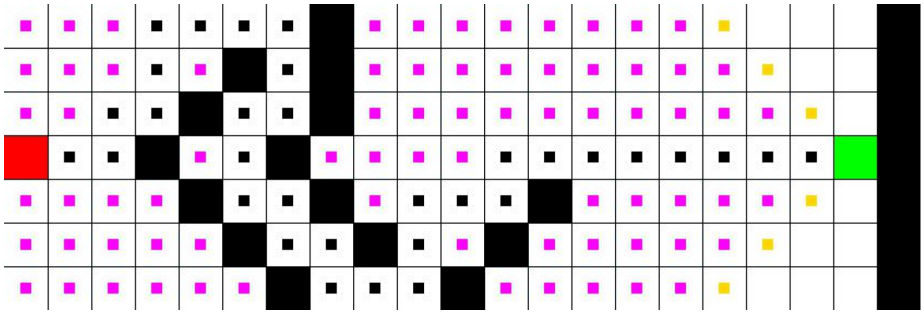

Figure 13: Board-1-2 med Dijkstra's

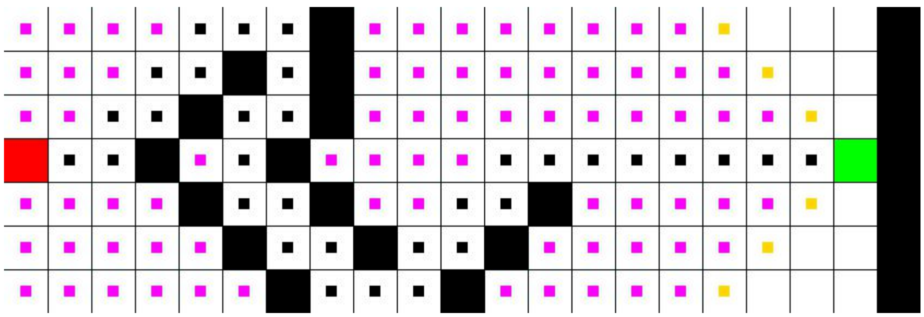

Figure 14: Board-1-2 med Breadth-First Search

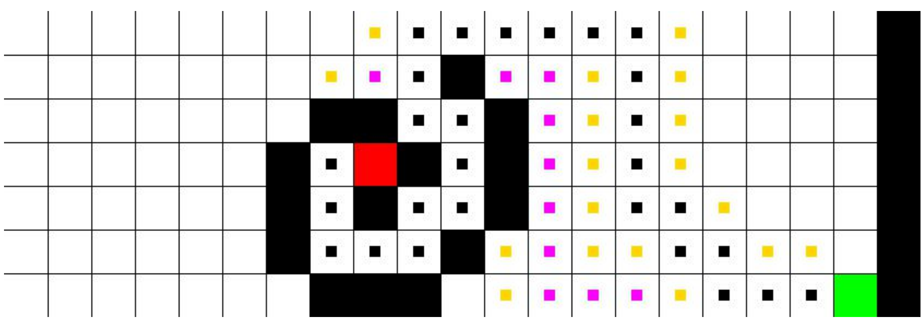

Figure 15: Board-1-3 med A*

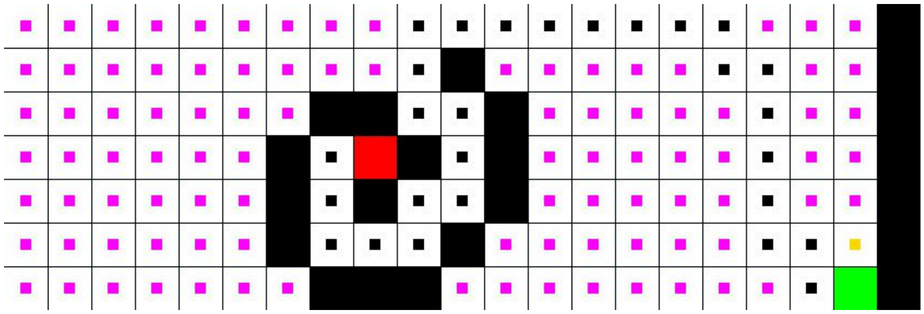

Figure 16: Board-1-3 med Dijkstra's

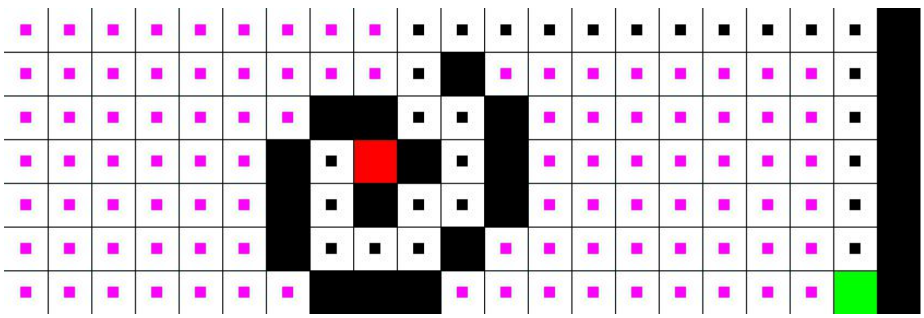

Figure 17: Board-1-3 med Breadth-First Search

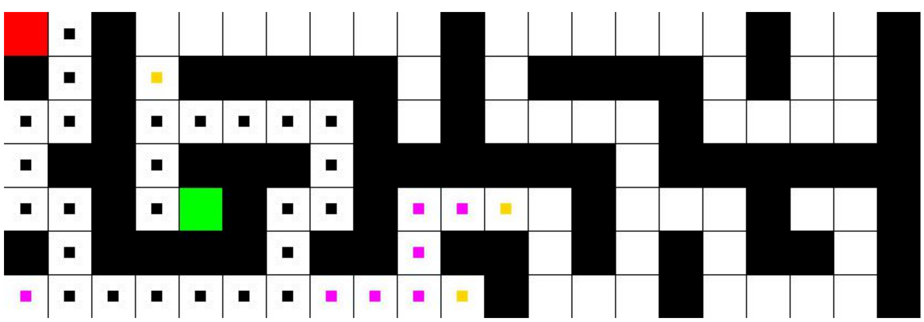

Figure 18: Board-1-4 med A*

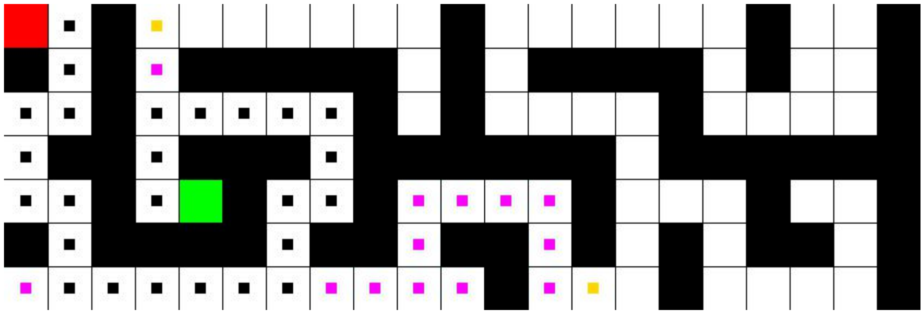

Figure 19: Board-1-4 med Dijkstra's

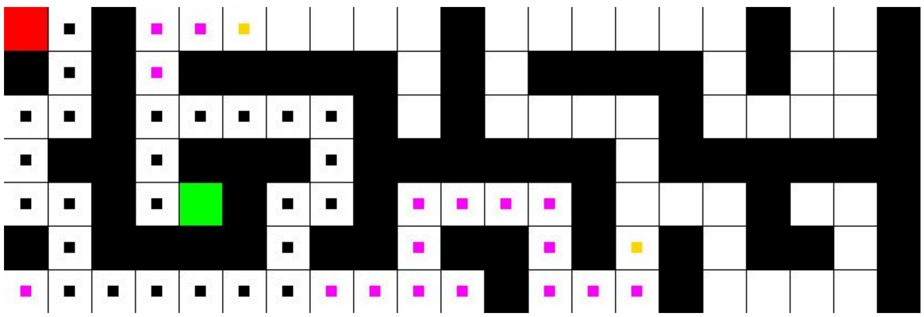

Figure 20: Board-1-4 med Breadth-First Search

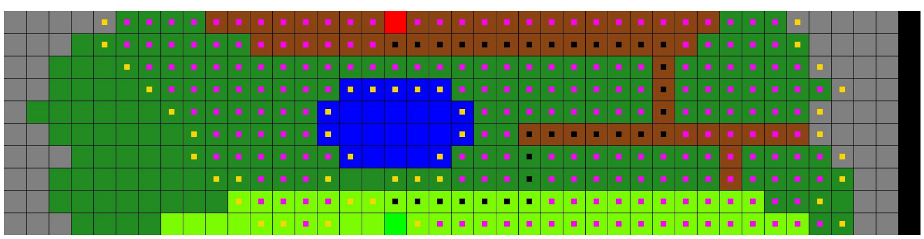

Figure 21: Board-2-1 med A*

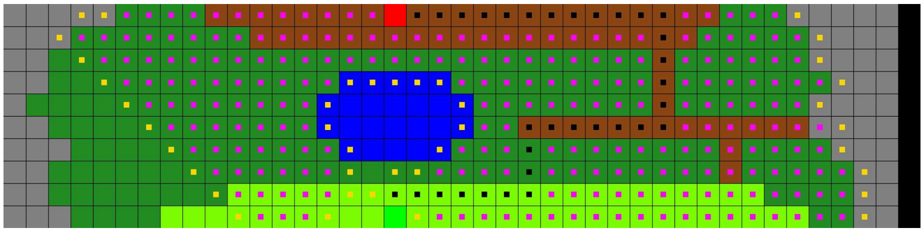

Figure 22: Board-2-1 med Dijkstra's

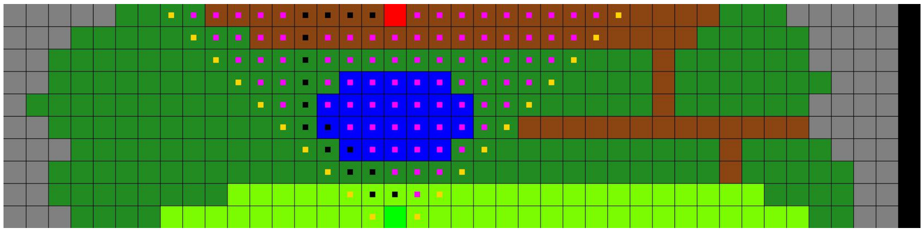

Figure 23: Board-2-1 med Breadth-First Search

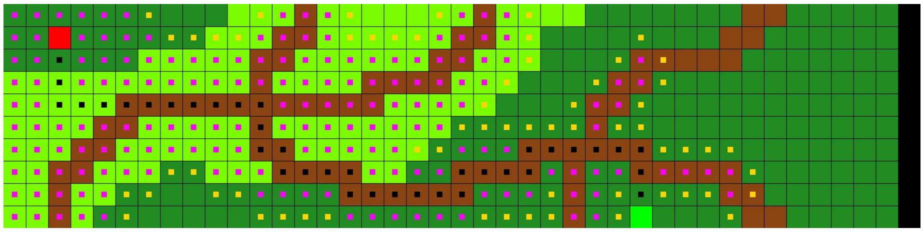

Figure 24: Board-2-2 med A*

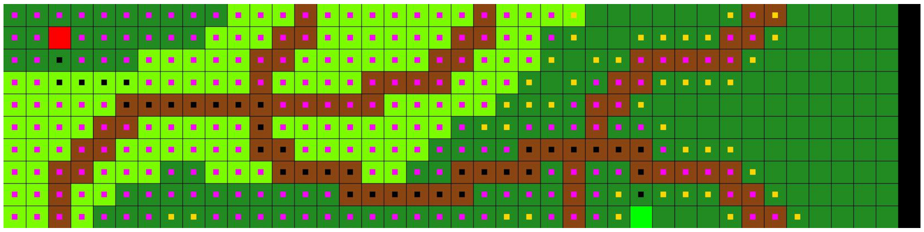

Figure 25: Board-2-2 med Dijkstra's

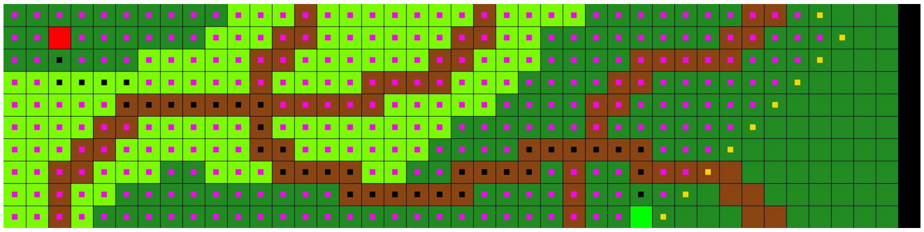

Figure 26: Board-2-2 med Breadth-First Search

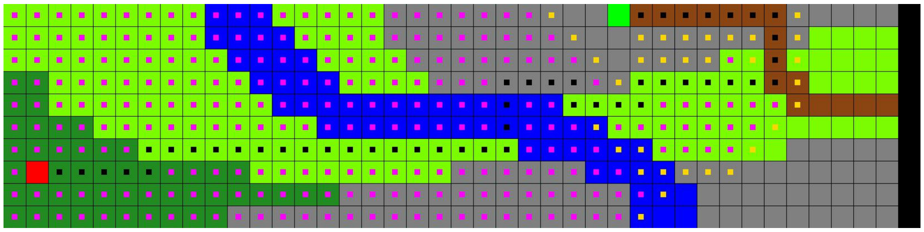

Figure 27: Board-2-3 med A*

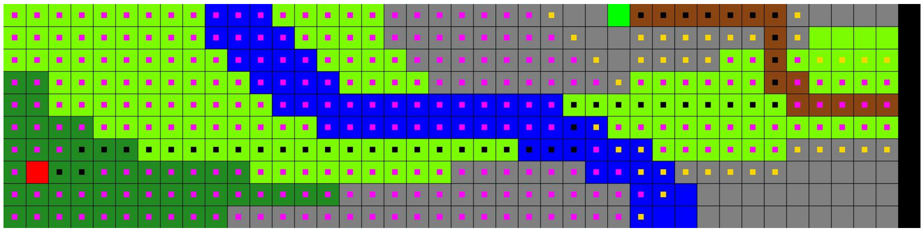

Figure 28: Board-2-3 med Dijkstra's

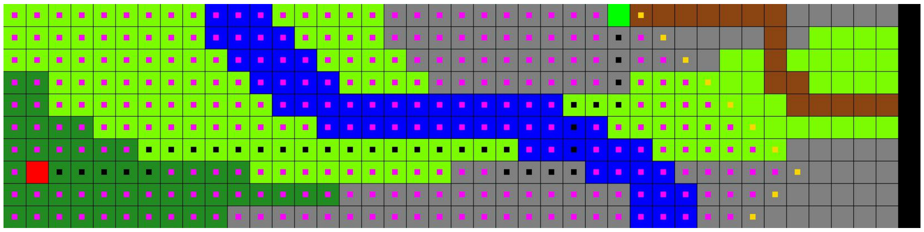

Figure 29: Board-2-3 med Breadth-First Search

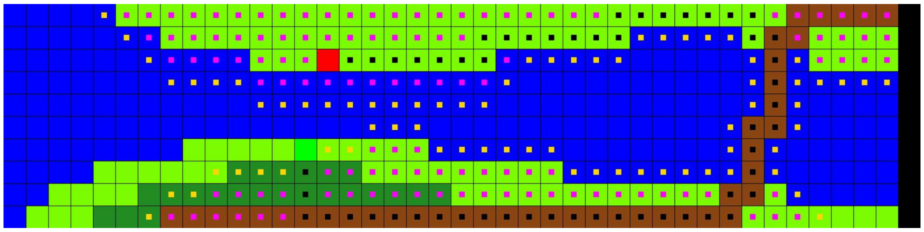

Figure 30: Board-2-4 med A*

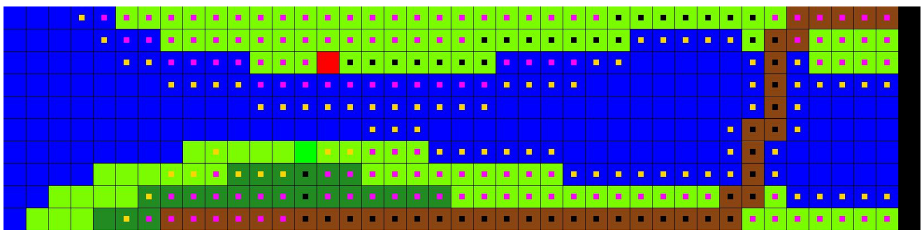

Figure 31: Board-2-4 med Dijkstra's

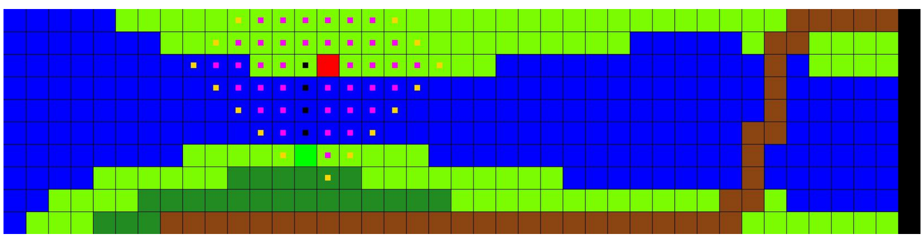

Figure 32: Board-2-4 med Breadth-First Search

3.3

For each game board processed above, a brief analysis of (a) any differences in the path found by A*, BFS and Dijkstra, and (b) any interesting differences in the number of open and closed between for the different algorithms.

(a) Man ser at når kostnadden for alle rutene er 1 så finner alltid alle algoritmene den raskeste veien. Her er A* helt tydelig raskest da den bruker F = G + H for å velge neste rute den skal undersøke, mens Dijkstra bruker bare H og BFS bruker første noden som ble lagt til i OpenList. Når rutene har forskjellige verdier vil A* og Dijkstra alltid finne den raskeste veien, mens BFS finner den raskeste veien innenfor det området den har undersøkt i det den finner den siste ruten. For eksempel finner BFS den raskeste veien fra A til B på brett 2-2 (Figure 26), siden her er den raskeste veien innenfor det området den allerede har søkt i. Mens på brett 2-4 (Figure 32) finner den ikke den raskeste veien, siden her finner den B tidlig og setter opp den raskeste veien ut i fra de rutene den har søkt gjennom. Ser at det er små forskjeller på de beste rutene da det er flere av dem på noen av brettene. Dette varierer ut i fra hvilken node som blir først sjekket av de nodene som er like gode.

(b) Man ser at når alle algoritmene finner den raskeste veien er det alltid A* som har færrest noder i ClosedList. Det betyr at A* alltid er den raskeste algoritmen for denne typen problem av de tre algoritmene, siden den søker gjennom færrest noder.