

**Assignment Cover Letter**

**(Group Work****)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student Information:** | **Surname** | **Given Name** | **Student ID Number** |
| 1. | Lay | Acheadeth | 2301956134 |
| 2. | Ellis | Mapakama | 2301960024 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Course Code** | **:** | COMP6340 |  | **Course Name** | **:** | Analysis Of Algorithm |
| **Class** | **:** | L3BC |  | **Name of Lecture(s)** | **:** | Tri Asih Budiono |
| **Major** | **:** | CS |  |  |  |  |
| **Assignment Title** | **:** |  |  |  |  |  |
| **Assignment Type** | **:** | Final Project |  |  |  |  |
| **Due Date** | **:** | 12/01/2021 |  | **Submission Date** | **:** | 12/01/2001 |

Signature of Student: (Group)

Group:

## **Project Details**

We are implementing two well-known shortest path algorithms to solve the maze problem.

Those algorithm are:

1. A\* shortest path finding algorithm
2. Dijkstra Shortest path finding algorithm

For interface we use python.

## **Main features**

For our visualization tools, there are some main feature to be expected here:

* The input option for our user whether to choose DijkStra or A\* algorithm to solve our maze.
* After we choose, a maze will be created and the algorithm will start implementing on the maze behind the scene.
* After that it analyzes some crucial details we need to know such as Node count, Time elapsed, for maze when started and Node explored, time elapsed when the algorithm is done implementing.
* Lastly it saves the image with the result of the implementation of the algorithm into it and shows us the image after saving.

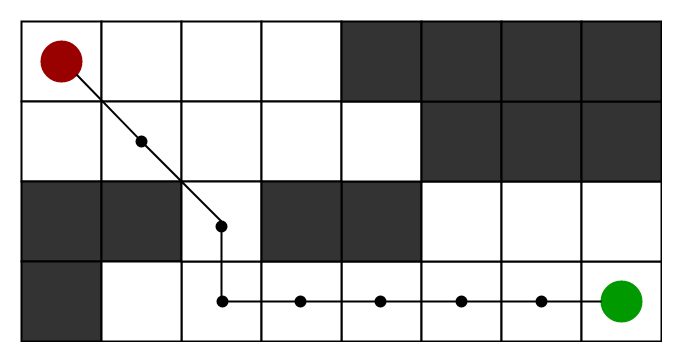
## **Algorithm explained:**

A\* path finding algorithm:

* Why?:This algorithm, unlike other traversal algorithms, has “brains” or its logic as human. What this really means is that it’s smarter than other conventional algorithms out there. Many game and web-based maps also use this algorithm to find the shortest path very efficiently (approximation).
* Explanation:

Consider a square grid having many obstacles and we are given a starting cell and a targeting cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here how A\* fix the problem.

What it does is at each step it picks the node according to the value -’f’ which is a parameter equal to the sum of two other parameters -’g’ and ‘h’. At each step it picks the node/cell having the lowest ‘f’ ,and processes that node/cell.



We define ‘g’ and ‘h’ as below:

g= the movement cost to move from the starting point to a given square on a grid, following the path generated to get there.

h= the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don’t know the actual distance until we find the path because all sorts of things can be in a way (wall,water,etc). There can be many ways to calculate this ‘h’.

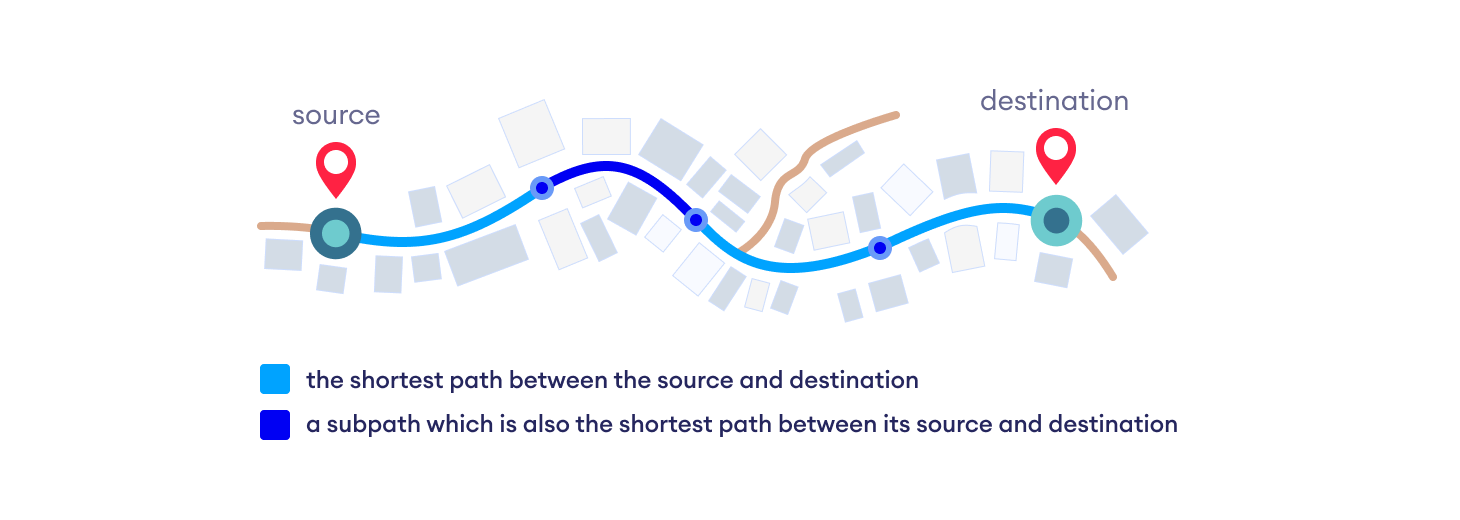
First, mantahann distance. (This is the simplest one).

Second, diagonal distance.

Third, Euclidean distance.

Dijkstra path finding algorithm:

* Explanation:

Dijkstra algorithm works on the basis that any subpath B->D of the shortest path A->D between vertices A and D is also the shortest path between B and D.

Dijkstra used this property in the opposite direction i.e we overestimate the distance of

each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.

The algorithm use a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

## 

## 

## **Application explained and some demo:**

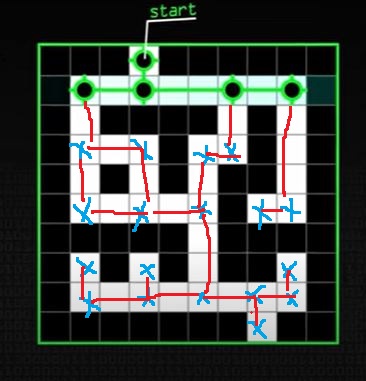
The Program take a maze file in png form that is only in black and white (Black for the wall and white for the path)

After uploading the image the program will convert the png maze and check for a possible path on the maze(the white areas) and create Nodes that will be regarded as the graph that is used by the path finding algorithm.

**Making The maze**

As the maze making algorithm will go through the image row by row analysing each pixel looking for places to place nodes.

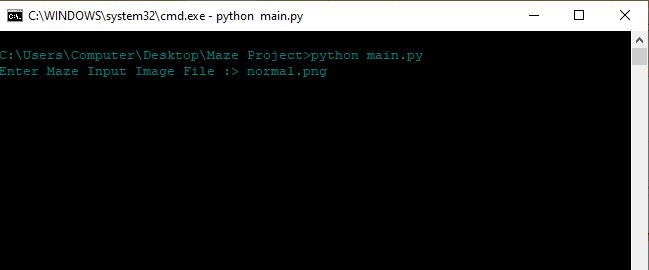
Nodes will be allocated to white pixels which are the path and a node is created only when the white box is in the end at the wall or if there are two possible directions from that position.



The image above shows the 23 nodes that have been created on this maze. These nodes will then now represent the graph that will be solved using the shortest path algorithm to get the path through the maze.

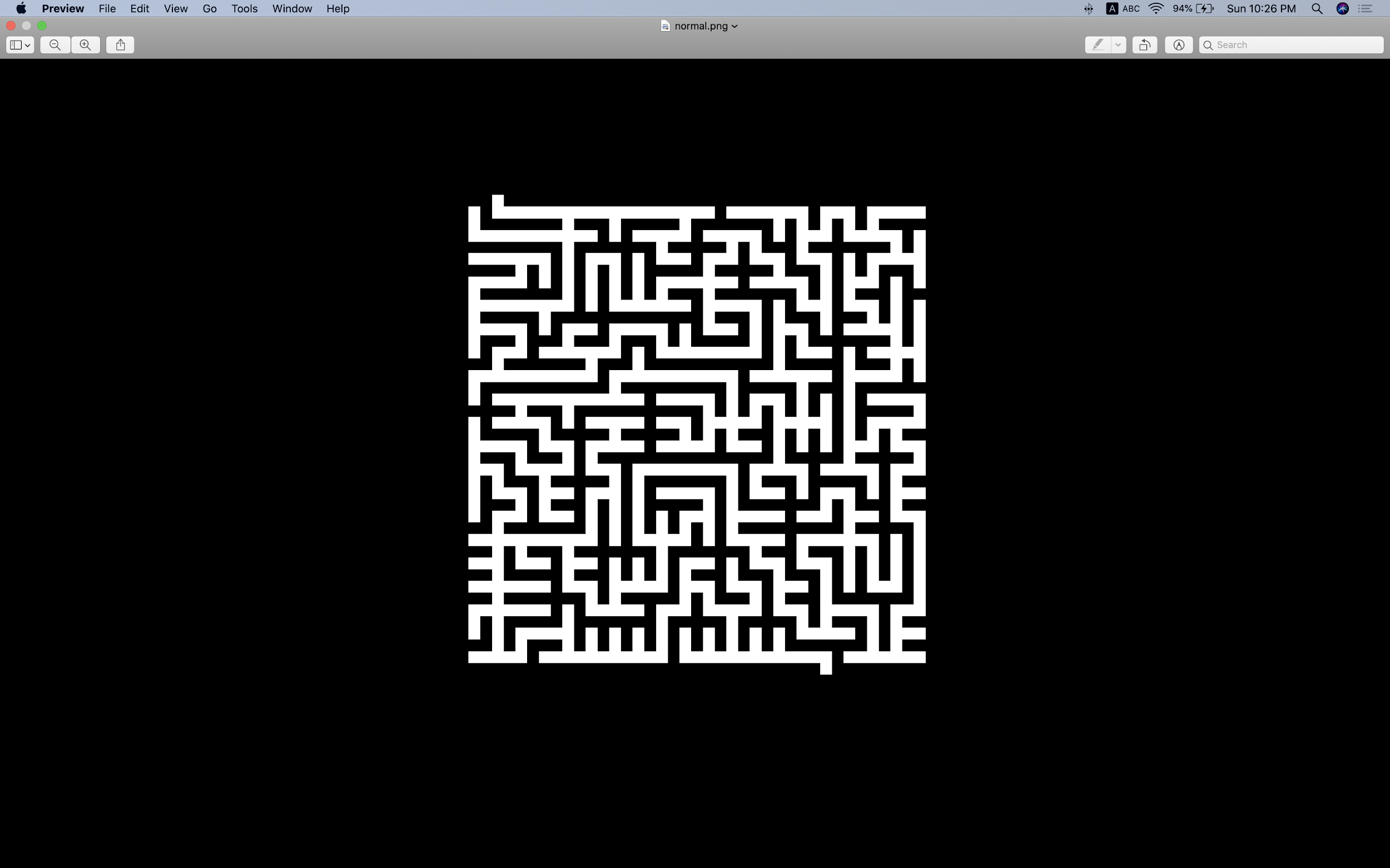
**Running the Application**

Open the maze project folder and then using your command prompt run the main.py

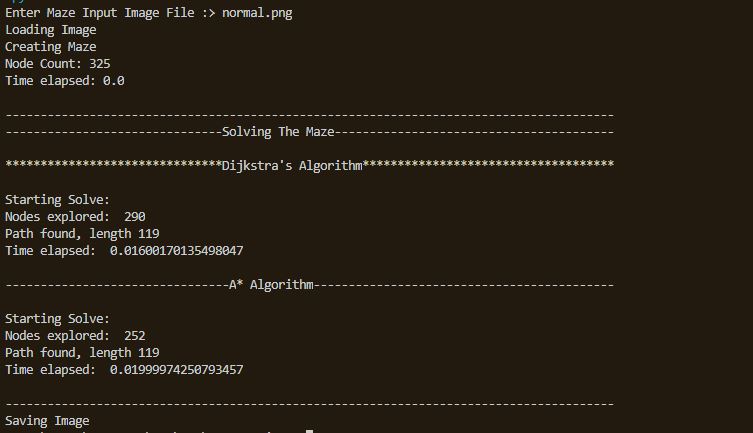


Enter the name of your maze png file (normal.png in this example). Make sure your maze file is in the maze project root folder or else put file path on the input file

Normal.png looks like this before solving the maze.

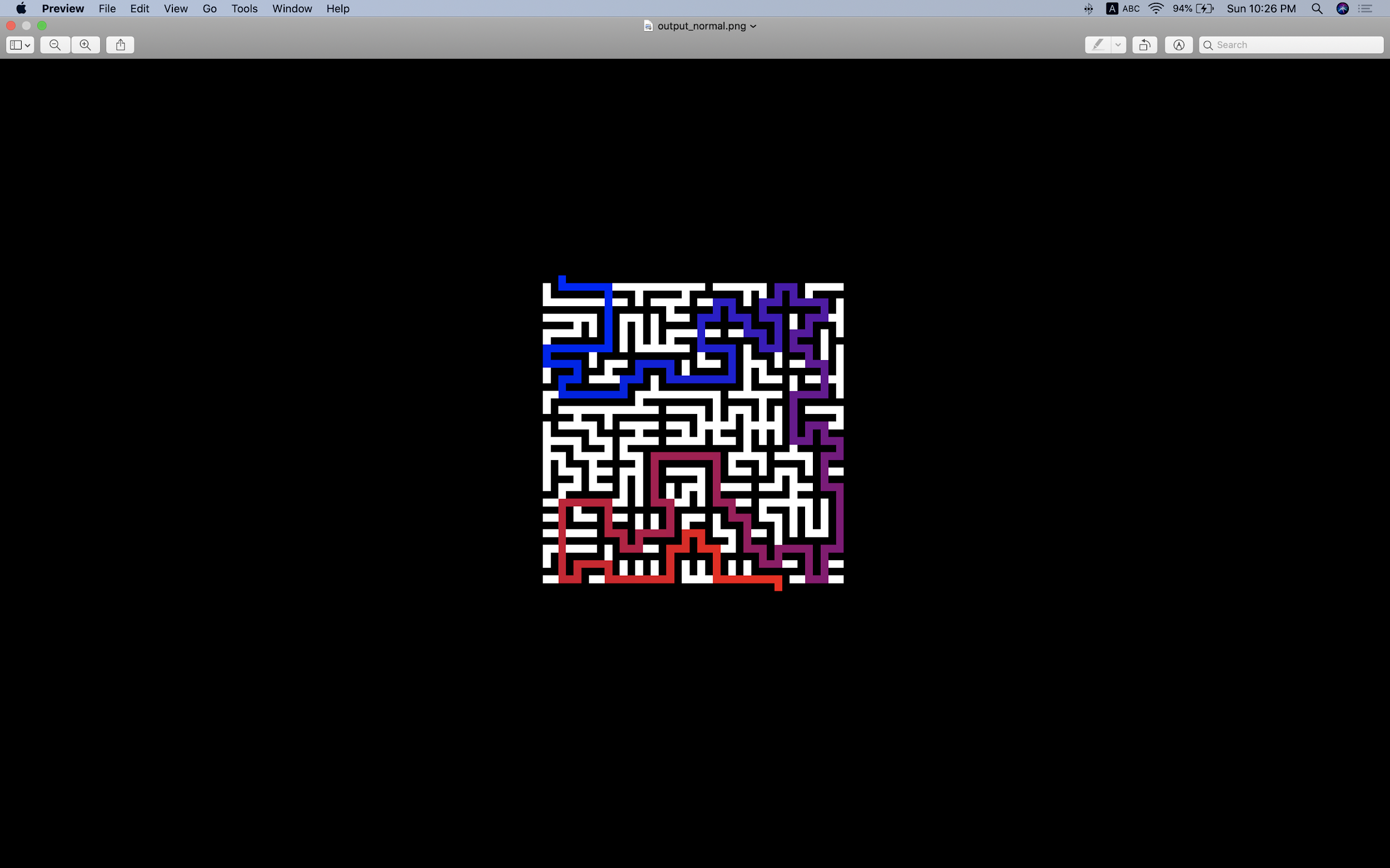


The program will run both the dijkstra and the A\* and then show the stats of how they both solved the maze for comparison



Because the way through the maze is always the same both algorithms will produce the same maze in the end but they take different runtimes to complete the maze.

The output maze looks like this.



This is the maze image you get after the algorithm is implementing and the maze is solved using the algorithm you have inputted.

## 

## 

## 

## 

## 

## **Implementation explained:**

MazeProject/main.py

The main function.  
def main():

while True:

print("To Exit Program Enter 'exit'\n")

input\_file = input("Enter Maze Input Image File :> ")

if (input\_file == "exit"):

sys.exit()

if (os.path.exists(input\_file) == True ):

break

else:

print(" Error on Inputs")

system('cls')

solve(input\_file, createOut(input\_file))

time.sleep(5)

im = Image.open(createOut(input\_file))

width, height = im.size

im.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()

The Solving Function

def solve(input\_file, output\_file):

# Load Image

print ("Loading Image")

im = Image.open(input\_file)

# Create the maze (and time it) - for many mazes this is more time consuming than solving the maze

print ("Creating Maze")

t0 = time.time()

maze = Maze(im)

t1 = time.time()

print ("Node Count:", maze.count)

total = t1-t0

print ("Time elapsed:", total, "\n")

print("---------------------------------------------------------------------------------------")

print("-------------------------------Solving The Maze----------------------------------------\n")

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Dijkstra's Algorithm\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n")

result = dijkstra("dijkstra",maze)

print("--------------------------------A\* Algorithm-------------------------------------------\n")

astar("astar", maze)

print("---------------------------------------------------------------------------------------")

"""

Create and save the output image.

This is simple drawing code that travels between each node in turn, drawing either

a horizontal or vertical line as required. Line colour is roughly interpolated between

blue and red depending on how far down the path this section is.

"""

print ("Saving Image")

im = im.convert('RGB')

impixels = im.load()

resultpath = [n.Position for n in result]

length = len(resultpath)

for i in range(0, length - 1):

a = resultpath[i]

b = resultpath[i+1]

# Blue - red

r = int((i / length) \* 255)

px = (r, 0, 255 - r)

if a[0] == b[0]:

# Ys equal - horizontal line

for x in range(min(a[1],b[1]), max(a[1],b[1])):

impixels[x,a[0]] = px

elif a[1] == b[1]:

# Xs equal - vertical line

for y in range(min(a[0],b[0]), max(a[0],b[0]) + 1):

impixels[a[1],y] = px

im.save(output\_file)

The above function will call the maze creating function and the algorithm once the maze has been created.

Making the Maze

class Maze:

class Node:

def \_\_init\_\_(self, position):

self.Position = position

self.Neighbours = [None, None, None, None]

#self.Weights = [0, 0, 0, 0]

def \_\_init\_\_(self, im):

width = im.size[0]

height = im.size[1]

data = list(im.getdata(0))

self.start = None

self.end = None

# Top row buffer

topnodes = [None] \* width

count = 0

# Start row

for x in range (1, width - 1):

if data[x] > 0:

self.start = Maze.Node((0,x))

topnodes[x] = self.start

count += 1

break

for y in range (1, height - 1):

#print ("row", str(y)) # Uncomment this line to keep a track of row progress

rowoffset = y \* width

rowaboveoffset = rowoffset - width

rowbelowoffset = rowoffset + width

# Initialise previous, current and next values

prv = False

cur = False

nxt = data[rowoffset + 1] > 0

leftnode = None

for x in range (1, width - 1):

# Move prev, current and next onwards. This way we read from the image once per pixel, marginal optimisation

prv = cur

cur = nxt

nxt = data[rowoffset + x + 1] > 0

n = None

if cur == False:

# ON WALL - No action

continue

if prv == True:

if nxt == True:

# PATH PATH PATH

# Create node only if paths above or below

if data[rowaboveoffset + x] > 0 or data[rowbelowoffset + x] > 0:

n = Maze.Node((y,x))

leftnode.Neighbours[1] = n

n.Neighbours[3] = leftnode

leftnode = n

else:

# PATH PATH WALL

# Create path at end of corridor

n = Maze.Node((y,x))

leftnode.Neighbours[1] = n

n.Neighbours[3] = leftnode

leftnode = None

else:

if nxt == True:

# WALL PATH PATH

# Create path at start of corridor

n = Maze.Node((y,x))

leftnode = n

else:

# WALL PATH WALL

# Create node only if in dead end

if (data[rowaboveoffset + x] == 0) or (data[rowbelowoffset + x] == 0):

#print ("Create Node in dead end")

n = Maze.Node((y,x))

# If node isn't none, we can assume we can connect N-S somewhere

if n != None:

# Clear above, connect to waiting top node

if (data[rowaboveoffset + x] > 0):

t = topnodes[x]

t.Neighbours[2] = n

n.Neighbours[0] = t

# If clear below, put this new node in the top row for the next connection

if (data[rowbelowoffset + x] > 0):

topnodes[x] = n

else:

topnodes[x] = None

count += 1

# End row

rowoffset = (height - 1) \* width

for x in range (1, width - 1):

if data[rowoffset + x] > 0:

self.end = Maze.Node((height - 1,x))

t = topnodes[x]

t.Neighbours[2] = self.end

self.end.Neighbours[0] = t

count += 1

break

self.count = count

self.width = width

self.height = height

MazeProject/dijkstra.py

from FibonacciHeap import FibHeap

from priority\_queue import FibPQ

def solve(maze):

# Width is used for indexing, total is used for array sizes

width = maze.width

total = maze.width \* maze.height

# Start node, end node

start = maze.start

startpos = start.Position

end = maze.end

endpos = end.Position

# Visited holds true/false on whether a node has been seen already. Used to stop us returning to nodes multiple times

visited = [False] \* total

# Previous holds a link to the previous node in the path. Used at the end for reconstructing the route

prev = [None] \* total

# Distances holds the distance to any node taking the best known path so far. Better paths replace worse ones as we find them.

# We start with all distances at infinity

infinity = float("inf")

distances = [infinity] \* total

# The priority queue. There are multiple implementations in priority\_queue.py

unvisited = FibPQ()

# This index holds all priority queue nodes as they are created. We use this to decrease the key of a specific node when a shorter path is found.

# This isn't hugely memory efficient, but likely to be faster than a dictionary or similar.

nodeindex = [None] \* total

# To begin, we set the distance to the start to zero (we're there) and add it into the unvisited queue

distances[start.Position[0] \* width + start.Position[1]] = 0

startnode = FibHeap.Node(0, start)

nodeindex[start.Position[0] \* width + start.Position[1]] = startnode

unvisited.insert(startnode)

# Zero nodes visited, and not completed yet.

count = 0

completed = False

# Begin Dijkstra - continue while there are unvisited nodes in the queue

while len(unvisited) > 0:

count += 1

# Find current shortest path point to explore

n = unvisited.removeminimum()

# Current node u, all neighbours will be v

u = n.value

upos = u.Position

uposindex = upos[0] \* width + upos[1]

if distances[uposindex] == infinity:

break

# If upos == endpos, we're done!

if upos == endpos:

completed = True

break

for v in u.Neighbours:

if v != None:

vpos = v.Position

vposindex = vpos[0] \* width + vpos[1]

if visited[vposindex] == False:

# The extra distance from where we are (upos) to the neighbour (vpos) - this is manhattan distance

d = abs(vpos[0] - upos[0]) + abs(vpos[1] - upos[1])

# New path cost to v is distance to u + extra

newdistance = distances[uposindex] + d

# If this new distance is the new shortest path to v

if newdistance < distances[vposindex]:

vnode = nodeindex[vposindex]

# v isn't already in the priority queue - add it

if vnode == None:

vnode = FibHeap.Node(newdistance, v)

unvisited.insert(vnode)

nodeindex[vposindex] = vnode

distances[vposindex] = newdistance

prev[vposindex] = u

# v is already in the queue - decrease its key to re-prioritise it

else:

unvisited.decreasekey(vnode, newdistance)

distances[vposindex] = newdistance

prev[vposindex] = u

visited[uposindex] = True

# We want to reconstruct the path. We start at end, and then go prev[end] and follow all the prev[] links until we're back at the start

from collections import deque

path = deque()

current = end

while (current != None):

path.appendleft(current)

current = prev[current.Position[0] \* width + current.Position[1]]

return [path, [count, len(path), completed]]

MazeProject/astar.py

from FibonacciHeap import FibHeap

from priority\_queue import FibPQ

# This implementatoin of A\* is almost identical to the Dijkstra implementation. So for clarity I've removed all comments, and only added those

# Specifically showing the difference between dijkstra and A\*

def solve(maze):

width = maze.width

total = maze.width \* maze.height

start = maze.start

startpos = start.Position

end = maze.end

endpos = end.Position

visited = [False] \* total

prev = [None] \* total

infinity = float("inf")

distances = [infinity] \* total

# The priority queue. There are multiple implementations in priority\_queue.py

unvisited = FibPQ()

nodeindex = [None] \* total

distances[start.Position[0] \* width + start.Position[1]] = 0

startnode = FibHeap.Node(0, start)

nodeindex[start.Position[0] \* width + start.Position[1]] = startnode

unvisited.insert(startnode)

count = 0

completed = False

while len(unvisited) > 0:

count += 1

n = unvisited.removeminimum()

u = n.value

upos = u.Position

uposindex = upos[0] \* width + upos[1]

if distances[uposindex] == infinity:

break

if upos == endpos:

completed = True

break

for v in u.Neighbours:

if v != None:

vpos = v.Position

vposindex = vpos[0] \* width + vpos[1]

if visited[vposindex] == False:

d = abs(vpos[0] - upos[0]) + abs(vpos[1] - upos[1])

# New path cost to v is distance to u + extra. Some descriptions of A\* call this the g cost.

# New distance is the distance of the path from the start, through U, to V.

newdistance = distances[uposindex] + d

# A\* includes a remaining cost, the f cost. In this case we use manhattan distance to calculate the distance from

# V to the end. We use manhattan again because A\* works well when the g cost and f cost are balanced.

remaining = abs(vpos[0] - endpos[0]) + abs(vpos[1] - endpos[1])

# Notice that we don't include f cost in this first check. We want to know that the path \*to\* our node V is shortest

if newdistance < distances[vposindex]:

vnode = nodeindex[vposindex]

if vnode == None:

# V goes into the priority queue with a cost of g + f. So if it's moving closer to the end, it'll get higher

# priority than some other nodes. The order we visit nodes is a trade-off between a short path, and moving

# closer to the goal.

vnode = FibHeap.Node(newdistance + remaining, v)

unvisited.insert(vnode)

nodeindex[vposindex] = vnode

# The distance \*to\* the node remains just g, no f included.

distances[vposindex] = newdistance

prev[vposindex] = u

else:

# As above, we decrease the node since we've found a new path. But we include the f cost, the distance remaining.

unvisited.decreasekey(vnode, newdistance + remaining)

# The distance \*to\* the node remains just g, no f included.

distances[vposindex] = newdistance

prev[vposindex] = u

visited[uposindex] = True

from collections import deque

path = deque()

current = end

while (current != None):

path.appendleft(current)

current = prev[current.Position[0] \* width + current.Position[1]]

return [path, [count, len(path), completed]]

## **Comparison between algorithm:**

We will now start analysing both of our algorithm deeply by comparing its time complexity and space complexity in solving the maze:

A\* path finding algorithm:

* Its time complexity usually depend on the heuristic. In the case of the unbound search space, the number of node expanded is exponential in the depth of solution (the shortest path) d:O(b^d), where b is the branching factor ( the average number of success per rate).
* Its space complexity O(|V|)=O(b^d), as it store all generated nodes in memory.

Dijkstra path finding algorithm:

* Its time complexity is O(ElogV). It can be reduced to O(E+VlogV) using the Fibonacci heap. The reason is Fibonacci heap take O(1) time for decrease-key operation while binary heap take O(logn) time.
* Space complexity: O(|V| ) or in worst case O(|V|+|E|)

**For dijkstra path finding time elapsed:**

Choose The Algorithm you want to use. (astar or dijkstra)

Enter Solving Method :> **dijkstra**

Enter Maze Input Image File :> **normal.png**

Loading Image

Creating Maze

Node Count: **325**

Time elapsed: **0.000926971435546875**

Starting Solve: **Dijkstra's Algorithm**

Nodes explored: **290**

Path **found**, length **119**

Time elapsed: **0.003423929214477539**

Saving Image

Process finished with exit code 0

**For A\* path finding time elapsed:**

Choose The Algorithm you want to use. (astar or dijkstra)

Enter Solving Method :> **astar**

Enter Maze Input Image File :> **normal.png**

Loading Image

Creating Maze

Node Count: **325**

Time elapsed: **0.001116037368774414**

Starting Solve: **A-star Search**

Nodes explored: **252**

Path **found**, length **119**

Time elapsed:  **0.004088163375854492**

Saving Image

Process finished with exit code 0

**CONCLUSION**

The project’s primary objective was accomplished and we were able to demonstrate how the two path finding algorithms can be used to solve maze.

We did implement a rather complicated method using an image file, but that is how we explored the creativity that comes with studying computer science.

Github link to Project files:

https://github.com/ellispax/Semester3/tree/main/Analysis%20Of%20Algorithms/Final%20Project(COMP6340)