Hacettepe University
Department of Computer Science

BBM-405 Fundamentals of Artificial Intelligence, Spring 2020

Homework- Analysis & Comparison of Search Algorithms

Name: Mert ÇÖKELEK

ID: 21727082

e-mail: mert.cokelek0699@gmail.com

Date: 09.05.2020

Ingredients

Implementation and Testing

In the beginning, I implemented the four algorithms (Depth-First Search, Breadth-First Search, Uniform-Cost Search, A* Search) in Python and ran them on a regular path-finding problem to check whether they work successfully or not.

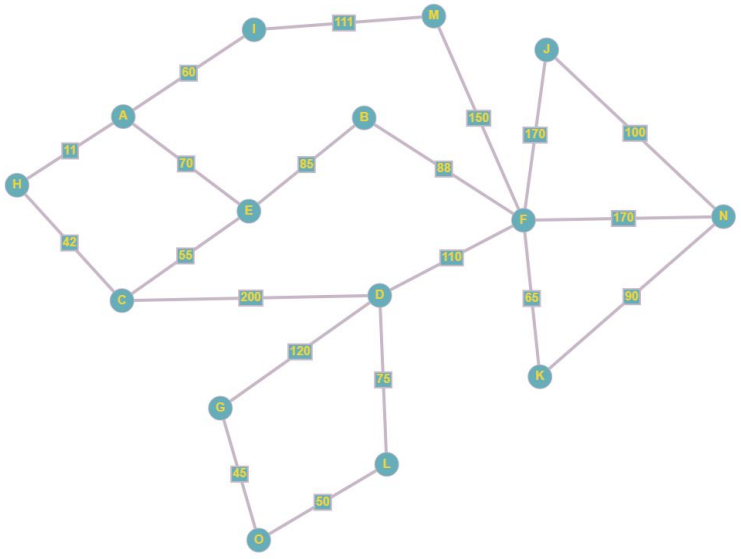
The below example is a simple pathfinding problem on a graph, and the algorithms will be tested in this simple example.

Example: Regular Pathfinding on a Graph, Represented with an Adjacency Matrix

1.1- Sample Graph and Representation in Python

Here is the graphical view of the problem.

For the representation and generation, I used [graphonline.ru.](graphonline.ru) In this website, you can generate graphs as you wish, and save them **as image** and **adjacency matrix.**



Here, the nodes are represented with uppercase letters, and they're stored as numbers(e.g A = 0, B = 1, and so on.)

The adjacency matrix is as follows, where the rows [x] and columns[y] correspond to different nodes, and the **[x][y]th & [y][x]th** element correspond to the **path cost** between node **x** and node **y**:

```
0, 0, 0, 0, 70, 0, 0, 11, 60, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 85, 88, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 200, 55, 0, 0, 42, 0, 0, 0, 0, 0, 0, 0,
0, 0, 200, 0, 0, 110, 120, 0, 0, 0, 0, 75, 0, 0, 0,
70, 85, 55, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 88, 0, 110, 0, 0, 0, 0, 0, 170, 65, 0, 150, 170, 0,
0, 0, 0, 120, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 45,
11, 0, 42, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
60, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 111, 0, 0,
0, 0, 0, 0, 0, 170, 0, 0, 0, 0, 0, 0, 0, 100, 0,
0, 0, 0, 0, 0, 65, 0, 0, 0, 0, 0, 0, 0, 90, 0,
0, 0, 0, 75, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 50,
0, 0, 0, 0, 0, 150, 0, 0, 111, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 170, 0, 0, 0, 100, 90, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 45, 0, 0, 0, 0, 50, 0, 0, 0,
```

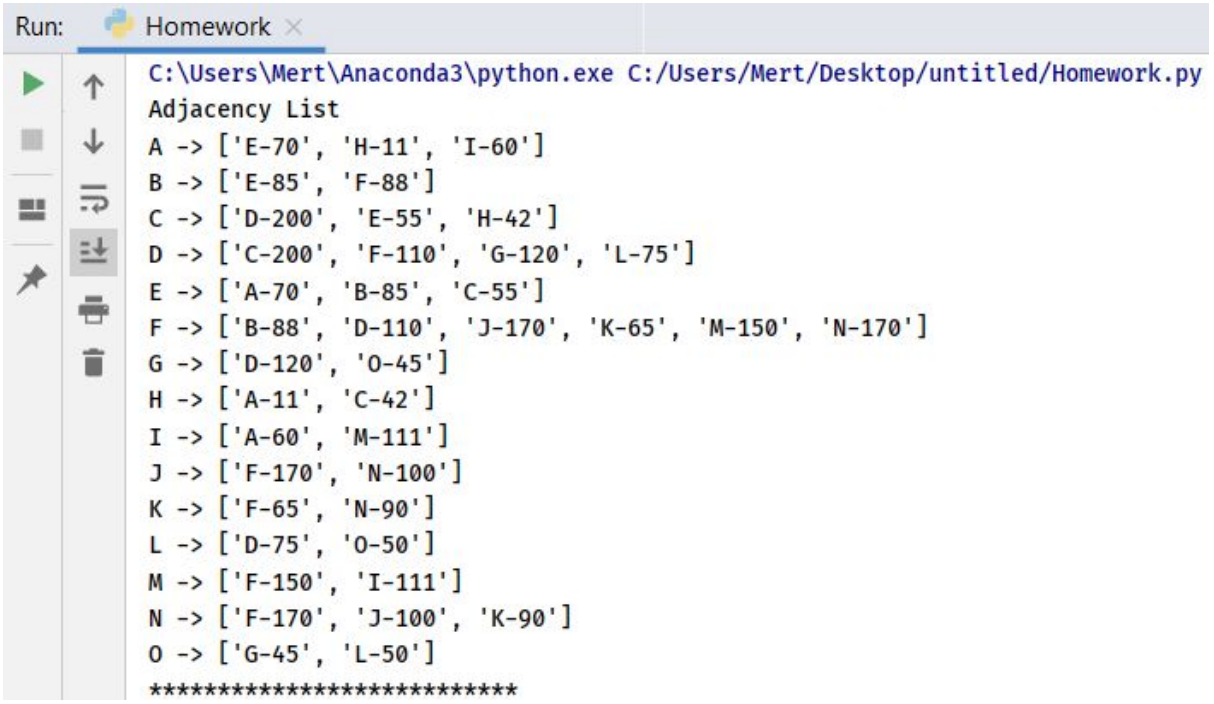
This text is read into 2d array with the following function:

```python
def txt_to_matrix(path):
    # open file in read mode
    adjacency_file = open(path, "r")

    # read the txt file into a 2d numpy matrix
    nparray = np.asarray(adjacency_file.read().splitlines())
    size = nparray.shape[0]
    matrix = np.ones((size, size))

    # convert characters into integers
    for i in range(nparray.shape[0]):
        matrix[i] = np.asarray(nparray[i].split(', '))[:size]
    matrix = matrix.astype(int)

    return matrix
```

and converted to **Adjacency List**, with the following function:

```python
def matrix_to_list(matrix):
    # graph: dictionary, structure: {node: [neighbor of node, path cost], [neigh..., cost..], ...}
    graph = {}
    for i, node in enumerate(matrix):
        adj = []
        for j, connected in enumerate(node):
            if connected:
                adj.append((j, connected))
        graph[i] = adj
    return graph
```

Here, is a sample output of the adjacency list:



(The numbers are converted to letters, for readability, and this list corresponds to the same graph, mentioned in the first page.)

1.1.1- DFS Implementation & Testing

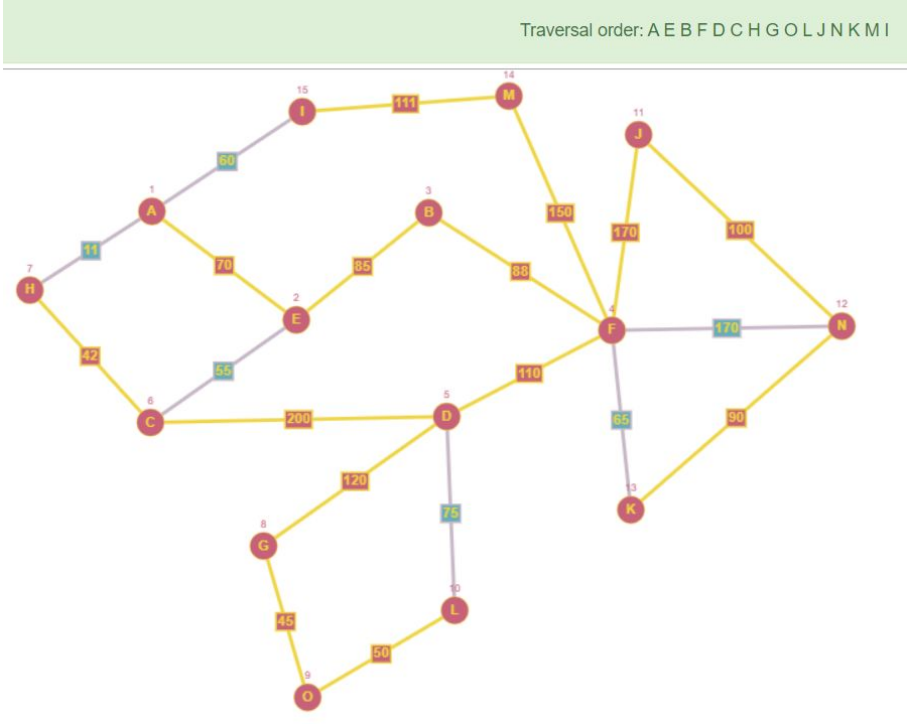
The DFS code is adapted from:
https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

```
def DFSUtil(graph, v, visited):
    visited[v] = True
    print(to_letter(v), end=' ')

    for i, cost in graph[v]:
        if visited[i] == False:
            DFSUtil(graph, i, visited)

def DFS(graph, v):
    visited = [False] * (max(graph) + 1)
    DFSUtil(graph, v, visited)
```

And when it's run with starting from node **"A"**, the paths to other nodes are found as:

```
**************************
DFS EXPANDING NODES IN ORDER
   A E B F D C H G O L J N K M I
**************************
Process finished with exit code 0
```

It can be visualized and verified by going to this link and clicking on ***Algorithms -> DFS.*** The visiting order on this graph is correct, even it has cycles. In the code, the possible cycles are prevented by storing the visited nodes. Conclusion: the DFS works well. *(The **yellow-red** parts show the expanded nodes-edges.)*



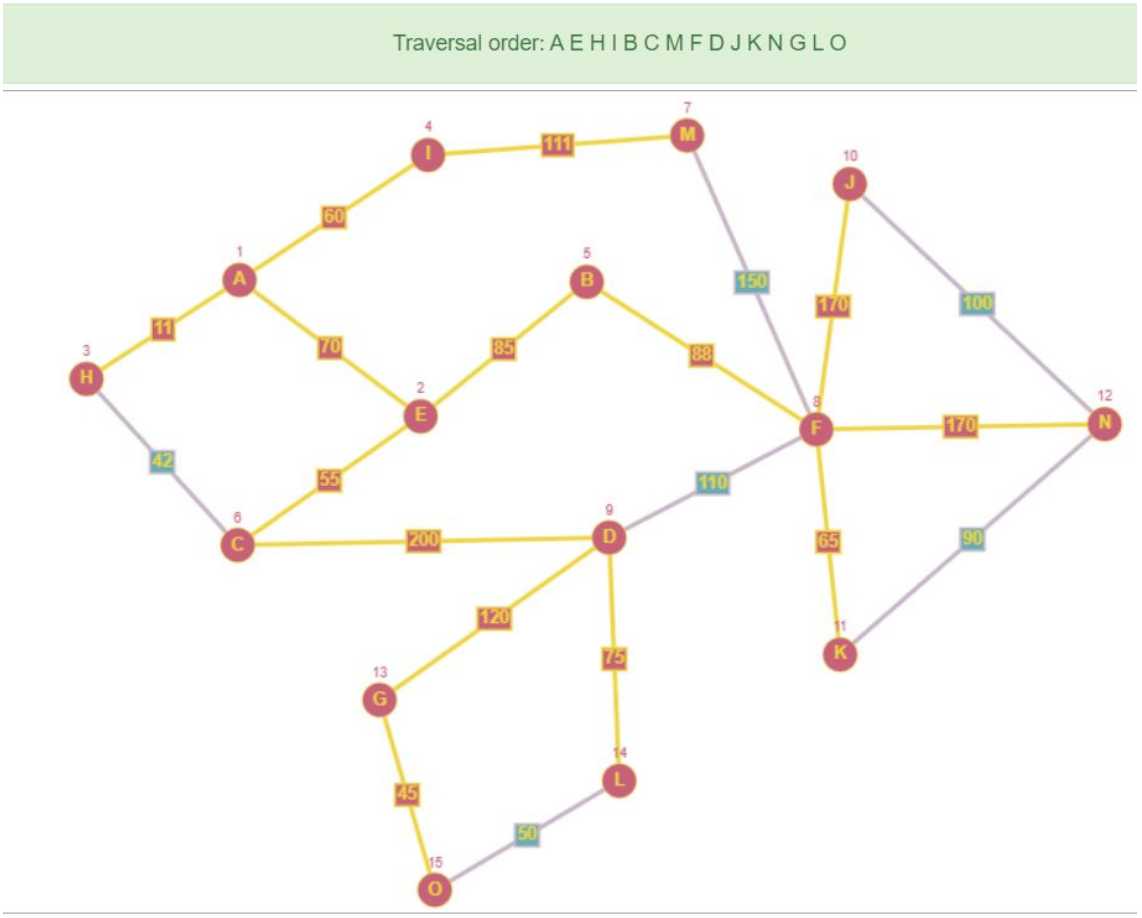
1.1.2- BFS Implementation & Testing

```
def bfs(graph, v):
    queue = list()
    visited = []
    queue.append(v)
    while queue != []:
        v = queue.pop(0)
        visited.append(v)
        for n, cost in graph[v]:
            if n not in queue and n not in visited:
                queue.append(n)
    path = ""
    for i in visited:
        path += to_letter(i) + " "
    return path
```

The BFS code is not taken/adapted from anywhere.

```
**************************
BFS EXPANDING NODES IN ORDER
    A E H I B C M F D J K N G L O
**************************
Process finished with exit code 0
```

Again, it can be visualized and verified by the previous link, and clicking on ***"BFS"*** *option. (The* ***yellow-red*** *parts show the expanded nodes-edges.)*



1.1.3- UCS Implementation & Testing

The UCS code is adapted from
https://stackoverflow.com/questions/43354715/uniform-cost-search-in-python
The visited nodes' path costs from **"A"** to **"O"** (start to end) are:

```
**************************
UCS EXPANDING NODES IN ORDER
    [('A', 0), ('H', 11), ('C', 53), ('I', 60), ('E', 70), ('B', 155), ('M', 171), ('F', 243),
('D', 253), ('K', 308), ('L', 328), ('G', 373), ('O', 378)]
**************************
```

Note that **"J" and "N"** nodes are not visited, since they are not chosen due to their high costs, on the path to **"O".**

Here, the output is as expected,

1- Starting from **"A",** in the frontier **("H=11, I=60, E=70")**, the node with the lowest cost is selected. The total cost is **11**.

2- The frontier priority queue of visited nodes **(A, H),** with summation of total cost and their path costs is: **("C=53, E=70, I=600"),** the node with lowest cost **(C)** is selected.

3- **The total cost is now 53,** and the frontier is **(I=60, E=70, D=253), so I is selected.**

4- **The total cost is now 60,** and the frontier is **(E=70, M=171, D=253) so E is selected.**

and so on. With this approach, the **lowest cost from A to O** is found as **378**, and the path is **A(0), H(11), C(11+42), D(53+200), L(253+75), O(328+50).**

1.1.4- A* Search Implementation & Testing

A code is the upgraded version of UCS. I only changed the heuristics with heuristics + path costs.*

A* is harder to follow on paper, compared to the other methods. So, I will give 3 examples with 3 different heuristics to explain it better.

The map is designed such that the closer the nodes are, the closer their names. (For example, **A, B, C** will be closer in map, where they are further from **J, K, L**)

(I especially added some exceptions(i.e A is closer to H, than G) to see the heuristic functions' performances.)

The heuristic function is as follows:

Current Node: The last node visited so far.
Node: In the frontier of Current Node to be expanded
End: The target node.

g(N) = Cost from **Current Node** to **Node**
h(N) = Heuristic = (Alphabetic order difference **between Node** and **End**.) * Multiplier
(Multiplier = variable parameter)
f(N) = g(N) + h(N)

Experiment 1.1.4.1: Heuristic Multiplier = 0 (Equivalent to UCS)

```
**************************
A  (Cost:0)  ->  H  (Cost:11)  ->  C  (Cost:53)  ->  I  (Cost:60)  ->  E
   (Cost:70)  ->  B  (Cost:155)  ->  M  (Cost:171)  ->  F  (Cost:243)  ->  D
   (Cost:253)  ->  K  (Cost:308)  ->  L  (Cost:328)  ->  G  (Cost:373)  ->  O
   (Cost:378)  ->  End
**************************
Process finished with exit code 0
```

Here, A* Algorithm behaves like UCS, so the outputs are the same. Because the heuristics are zero, which results in the current information being only the current path cost + edge cost.

Experiment 1.1.4.2: Heuristic Multiplier = 25

```
**************************
A  (Cost:0)  ->  H  (Cost:11)  ->  C  (Cost:53)  ->  E  (Cost:70)  ->  B
   (Cost:155)  ->  F  (Cost:243)  ->  D  (Cost:253)  ->  K  (Cost:308)  ->  L
   (Cost:328)  ->  G  (Cost:373)  ->  O  (Cost:378)  ->  End
**************************
```

Here, the costs are again the actual path costs, however the selected nodes are different than the previous one. This is because of the different heuristic function. For example, "I" is not selected in this path, where it was selected before. Because the heuristic value for "I" is "25 * (abs(14-8)) + 60 = 210". (0 = A, 8 = I). The heuristic value for "E" is "25 * (14-4) + 70 = 320". So, I is selected, instead of E.

The detailed output containing the steps with calculated heuristic values are given in **appendix #1.** (It is too long to fit in this report)

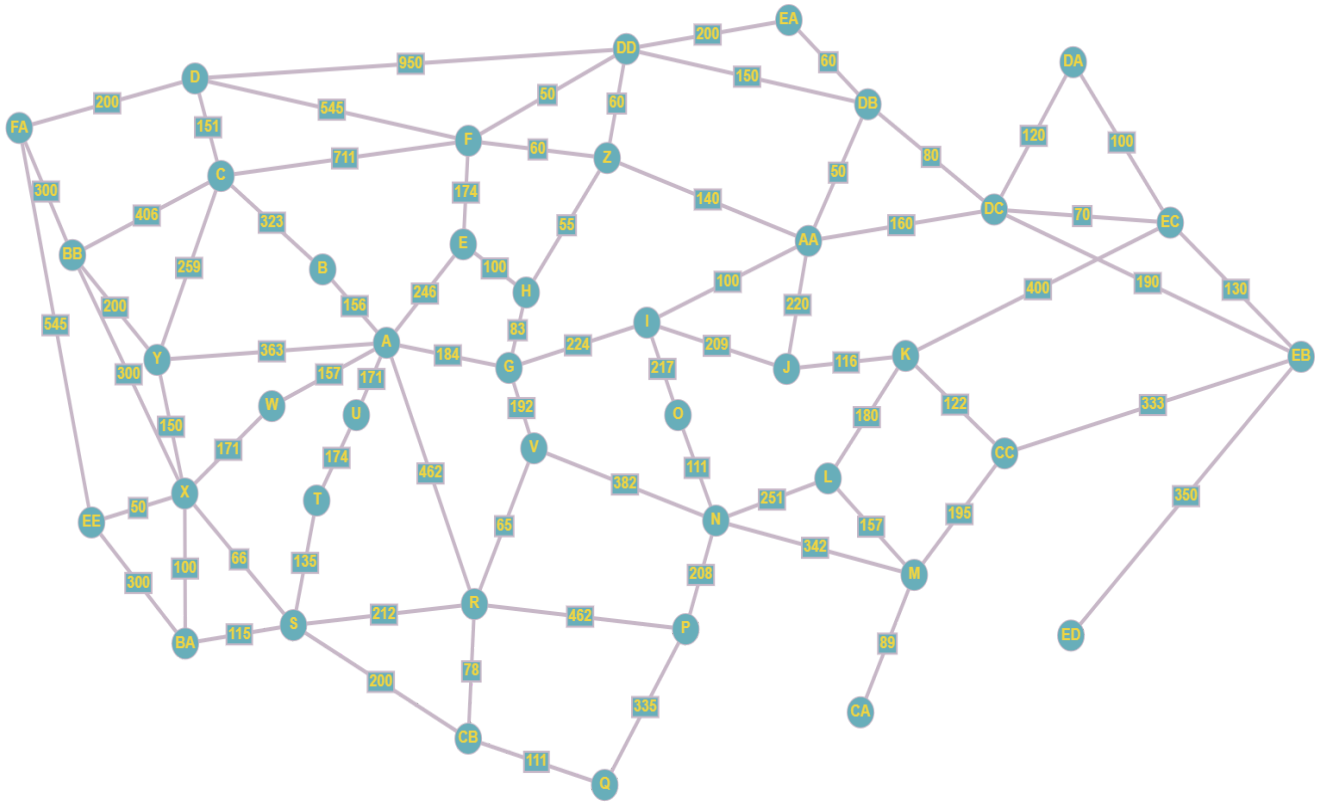
Experiment #1.4.3: Heuristic Multiplier = 100

```
**************************
A (Cost:0) -> H (Cost:11) -> C (Cost:53) -> I (Cost:60) -> E
   (Cost:70) -> B (Cost:155) -> M (Cost:171) -> F (Cost:243) -> D
   (Cost:253) -> K (Cost:308) -> L (Cost:328) -> G (Cost:373) -> O
   (Cost:378) -> End
**************************
```

In this situation, the heuristic multiplier is selected so large that the decisions converge to the actual edge costs. Because all the values are growing so large that the numeric relationship (greater than, less than) between them do not change. Again, the detailed output of the decisions, and their reasons are printed in **appendix #2.**

This part is intentionally left blank.

1.2- Testing on a Larger Graph (43x43)



The second graph is shown here.

The expanded nodes in order in this graph are as follows:

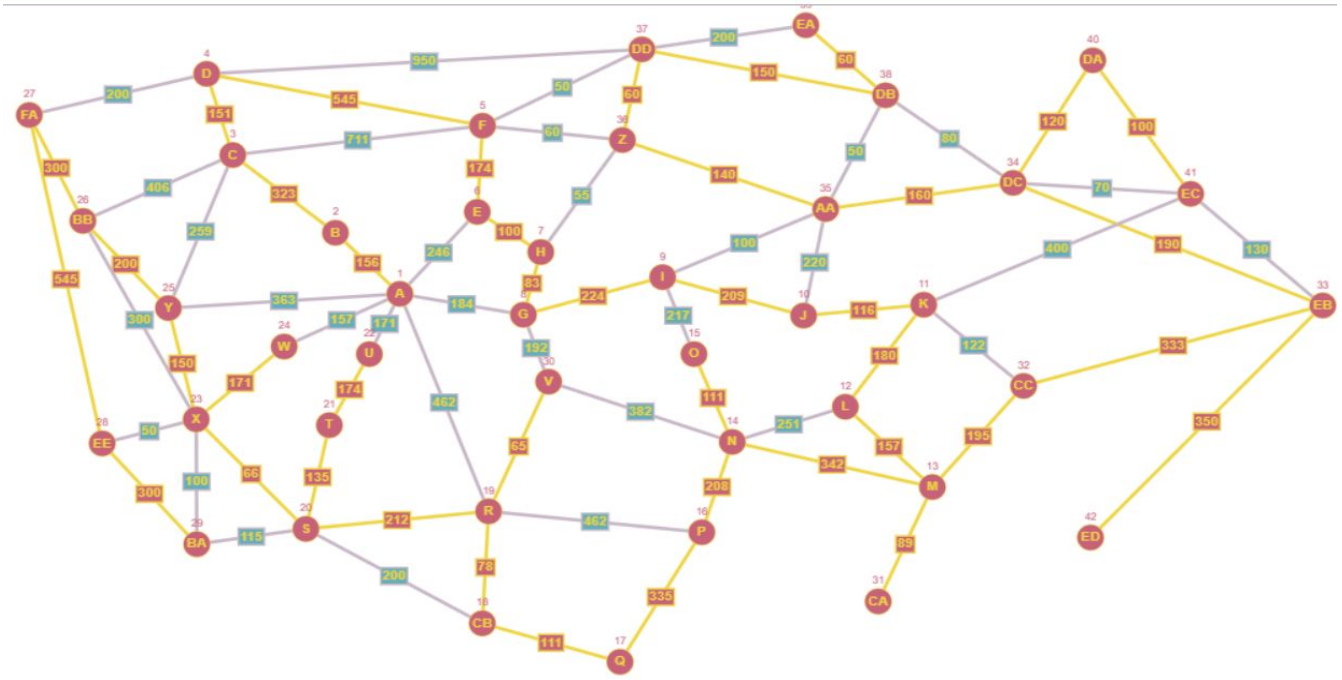
```
**************************
BFS NODES IN EXPANDING ORDER
    A B E G R U W Y C F H I V P S CB T X BB D Z DD J O AA N Q BA EE FA DB EA K DC L M CC EC DA EB CA ED
**************************
DFS NODES IN EXPANDING ORDER
   A B C D F E H G I J K L M N O P Q CB R S T U X W Y BB FA EE BA V CA CC EB DC AA Z DD DB EA DA EC ED
**************************
UCS NODES IN EXPANDING ORDER       [('A', 0), ('B', 156), ('W', 157), ('U', 171), ('G', 184),
    ('E', 246), ('H', 267), ('Z', 322), ('X', 328), ('T', 345), ('Y', 363), ('V', 376), ('EE',
    378), ('F', 382), ('DD', 382), ('S', 394), ('I', 408), ('BA', 428), ('R', 441), ('AA',
    462), ('C', 479), ('DB', 512), ('CB', 519), ('BB', 563), ('EA', 572), ('DC', 592), ('J',
    617), ('O', 625), ('D', 630), ('Q', 630), ('EC', 662), ('DA', 712), ('K', 733), ('N', 736),
    ('EB', 782), ('FA', 830)]
**************************
```

Note: The outputs are corresponding to the expanded nodes in order.

The results can be verified by clicking the link, and selecting the visualization method, as previous ones.

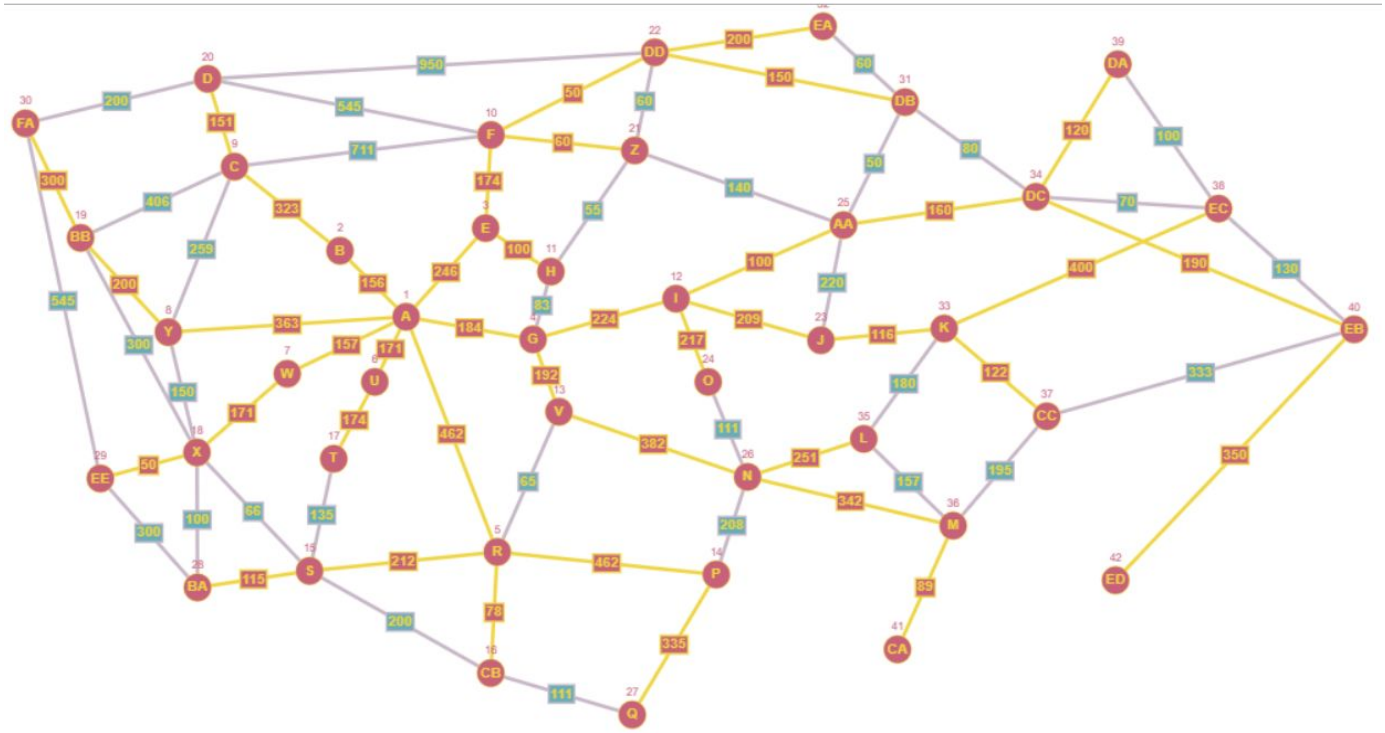
Graph State after DFS Executing (Red-Yellow nodes-edges are expanded)

Traversal order: A B C D F E H G I J K L M N O P Q CB R S T U X W Y BB FA EE BA V CA CC EB DC AA Z DD DB EA DA EC ED



Graph State after BFS Executing (Red-Yellow nodes-edges are expanded)

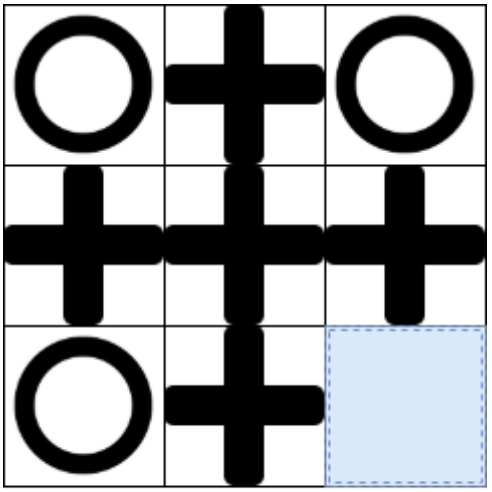
Traversal order: A B E G R U W Y C F H I V P S CB T X BB D Z DD J O AA N Q BA EE FA DB EA K DC L M CC EC DA EB CA ED



1.3- Five-Tac-Toe Game

Problem Definition: This problem is similar to **tic-tac-toe** game, however since the algorithms covered are not for “Constraint Satisfaction Problems”, the game is adapted to search problems.

- In this problem, there is a 9-grid puzzle map, which contains:
  - 5 (‘+’) symbols,
  - 3 (‘O’) symbols,
  - 1 blank tile.
- Initially, the elements in the puzzle are randomly distributed. The player’s (in this case the computer) aim is to re-order the tiles so that their final positions will be as:



5-Tac-Toe

- The player can only replace the blank tile with its adjacent neighbors. It is similar to 8-puzzle game, but there is a big difference.
  - In in 8-puzzle game, there are 9 unique tiles, and their expected final position is known. (1 will be on coordinates <0,0>; 3 will be on <0,2>, and so on).
  - In **five-tac-toe,** since the “+”s and “o”s are distinct, their final positions are not deduced directly, thus, we can not define a specific heuristic value for every single tile.
    - For example we can’t say that “This !+! will belong to this place, and that ‘+’ will belong to that place”. Instead, we will say that “+”s will be in the “2nd, 4th, 5th, 6th, 8th” tiles. Their order will not matter.
  - So we define one overall heuristic value for the puzzle, which takes into account the minimum summation of manhattan distances for each tile.

The A* search algorithm should be adapted to this problem, so the heuristic function is changed as:

```
def heuristic(self):
    n = self.n

    # The 2d map is taken as a 1d array.
    plus_indices = [i for i in range(n * n) if self.puzzle[i] == '+']
    p_row = [i // n for i in plus_indices] # X coordinates of '+'s
    p_col = [i % n for i in plus_indices]  # Y coordinages of '-'s
    plus_indices = list(zip(p_row, p_col)) # Merge coordinate pairs
    # print(plus_indices)
    ground_truth = [1, 3, 4, 5, 7]          # The '+'s should be in these indices.
    ground_truth_row = [i // n for i in ground_truth]
    ground_truth_col = [i % n for i in ground_truth]
    ground_truth = list(zip(ground_truth_row, ground_truth_col))
    # print(ground_truth)

    # minimum sum of manhattan distances of elements of the current state and the final state.
    soldan = [abs(x1 - x2) + abs(y1 - y2) for (x2, y2) in ground_truth for (x1, y1) in plus_indices]
    sagdan = [abs(x1 - y1) + abs(x2 - y2) for (x2, y2) in ground_truth for (x1, y1) in plus_indices]
    manhattan = [min(a, b) for (a, b) in zip(soldan, sagdan)]
    return sum(manhattan)
```

Since the other algorithms do not need a heuristic function, they will be kept same.
A sample output for A* search on this problem is as follows (This is only for demonstration, the algorithms' performances (time, memory, completeness, optimality) will be discussed in the next sections.)

```
0 th step
_ 0 0
0 + +
+ + +

1 th step
0 _ 0
0 + +
+ + +

2 th step
0 + 0
0 _ +
+ + +

3 th step
0 + 0
0 + +
+ _ +

4 th step
0 + 0
0 + +
_ + +

5 th step
0 + 0
_ + +
0 + +

6 th step
0 + 0
+ _ +
0 + +

7 th step
0 + 0
+ + _
0 + +

8 th step
0 + 0
+ + +
0 + _
```

A* algorithm reaches the correct positions in 8 steps.

```
0 th step
_   0   0
0   +   +
+   +   +

1 th step
0   0   0
_   +   +
+   +   +

2 th step
0   0   0
+   +   +
_   +   +
```

.....

```
33 th step
0   +   +
_   +   +
+   0   0

34 th step
0   +   +
+   +   +
_   0   0

35 th step
0   +   +
+   +   +
0   _   0
```

```
36 th step
0   +   +
+   _   +
0   +   0

37 th step
0   _   +
+   +   +
0   +   0

38 th step
0   +   _
+   +   +
0   +   0
```

DFS Reaches in 38 steps.

Evaluation & Detailed Analysis of Algorithms

In this section, the algorithms will be evaluated on 2 different problems, path finding on a graph with 100 nodes, and Five-Tac-Toe game.

Path Finding on a Graph with 100 Nodes

2.1- Time Complexity

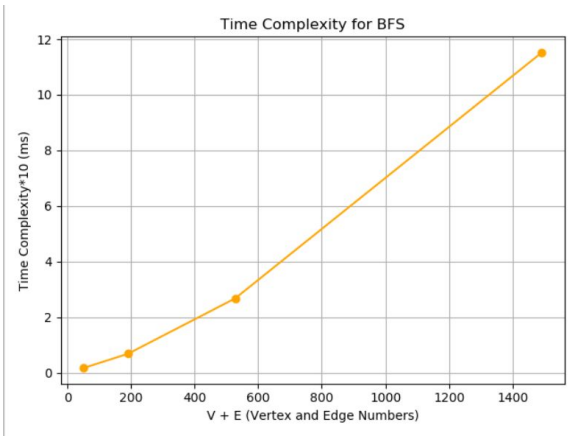
Here is the code for calculating the elapsed time, for DFS (and other algorithms)

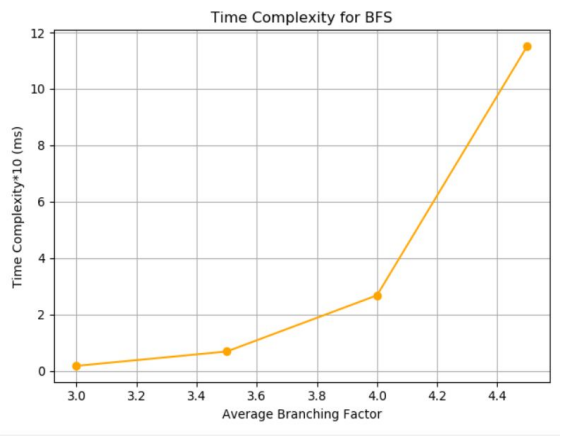
```
total = 0.0
for i in range(1000): # Run BFS 1000 times, and get their average.
    start_time = time.time() * 1000.0 # Convert seconds -> ms
    DFS(adjacency_list, 0, random.randint(1, 40)) # random = select random node as target, in the graph
    end_time = time.time() * 1000.0 # Convert seconds -> ms
    elapsed = end_time - start_time
    if elapsed == 0:                # Sometimes, the time is calculated as 0. Prevent this.
        i -= 1
    else:
        total += elapsed            # Total time
total /= 1000                       # BFS Ran 1000 times, find average.
```

The algorithms are run 1000 times with different randomly selected goal states, and their average elapsed times are taken. Sometimes, the randomly generated nodes might be close to the starting node and the algorithms finish very quick (elapsed time shows 0), so I prevented this situation to get a more accurate score.

2.1.1 BFS Time Complexity

```
**************************
BFS on Graph with 15 nodes and 37 edges
    Average Time to Find All Paths From Starting Point A:
     0.017983642578125 ms

Process finished with exit code 0

**************************
BFS on Graph with 43 nodes and 148 edges
    Average Time to Find All Paths From Starting Point A:
     0.06881689453125 ms

Process finished with exit code 0
**************************
BFS on Graph with 100 nodes and 428 edges
    Average Time to Find All Paths From Starting Point A:
     0.268244384765625 ms

Process finished with exit code 0

BFS on Graph with 240 nodes and 1252 edges
    Average Time to Find All Paths From Starting Point A: 1.15208872412 ms

Process finished with exit code 0
```

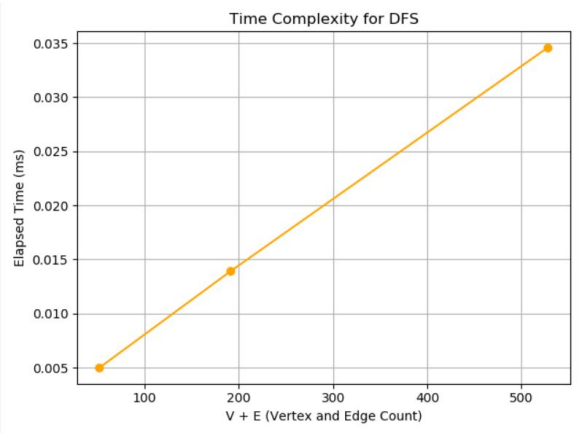


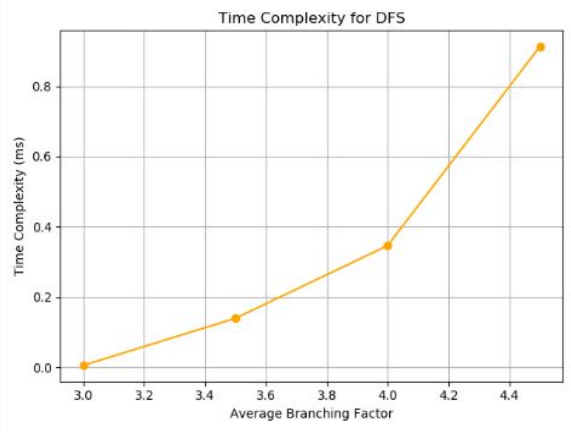


As it can be seen, the time and the number of edges + nodes has a **linear relationship**. However, the branching factor and time complexity has an **exponential relationship.**

2.1.2- DFS Time Complexity

```
**************************
DFS on Graph with 15 nodes and 37 edges
    Average Time to Find All Paths From Starting Point A:
     0.0049951171875 ms

Process finished with exit code 0

**************************
DFS on Graph with 43 nodes and 148 edges
    Average Time to Find All Paths From Starting Point A:
     0.013927978515625 ms

Process finished with exit code 0

**************************
DFS on Graph with 100 nodes and 428 edges
    Average Time to Find All Paths From Starting Point A:
     0.03465478515625 ms

Process finished with exit code 0
```



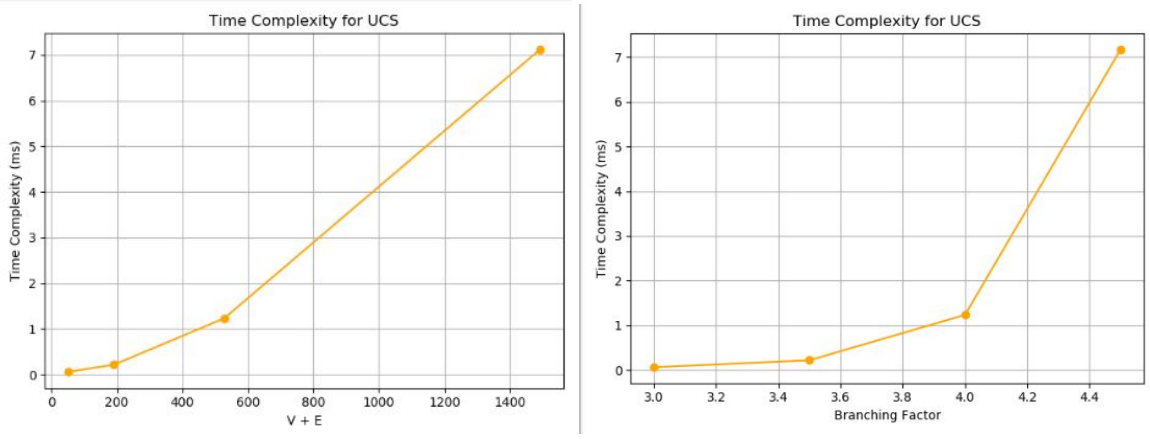


There is again a linear relationship between vertex and edge numbers and elapsed time, and an exponential relationship between branching factor and time required.

- Note that **the effect of branching factor to time complexity in BFS is higher than DFS**.
- This can be because BFS traverses the branches in order and the amount of expanded nodes increases in the power of branching factor, where in DFS the expansion is done through a single direction, until reaching the terminal, and then, going to the next branch.

2.1.3- UCS Time Complexity

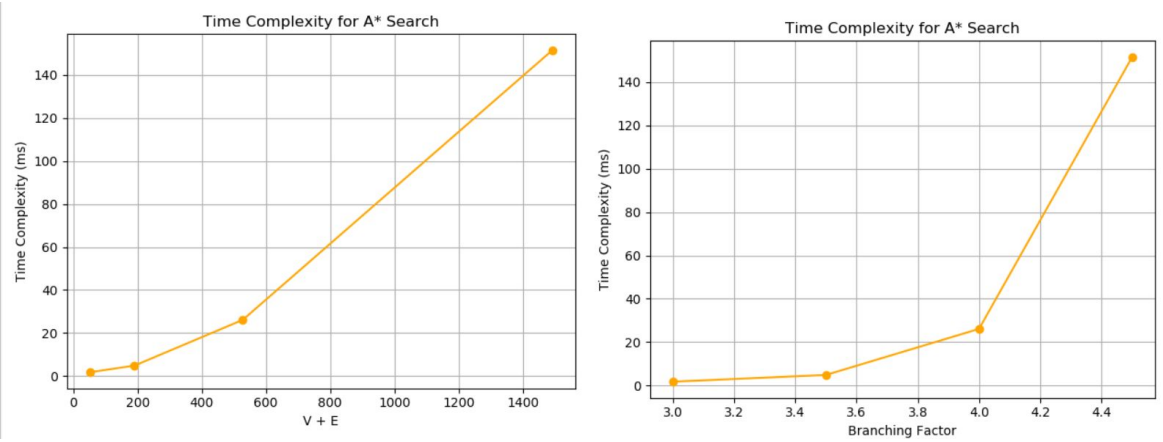
```
**************************
UCS on Graph with 15 nodes and 37 edges
    Average Time to Find All Paths From Starting Point A:
     0.062520263671875 ms

Process finished with exit code 0

**************************
UCS on Graph with 43 nodes and 148 edges
    Average Time to Find All Paths From Starting Point A:
     0.218707275390625 ms

Process finished with exit code 0

**************************
UCS on Graph with 100 nodes and 428 edges
    Average Time to Find All Paths From Starting Point A:
     1.23404345703125 ms

Process finished with exit code 0
```



UCS behaves similar to BFS, when all the edge costs are the same. In the graphs that I generated, the different path costs caused UCS to take into account different number of edges. The average elapsed time requirements are increased in UCS, since it uses Priority Queue, which is a more complex data structure.

2.1.4- A* Search Time Complexity

```
A* on Graph with 15 nodes and 37 edges
    Average Space Required to All Paths From Starting Point A: 1.7816477
A* on Graph with 43 nodes and 148 edges
    Average Space Required to All Paths From Starting Point A: 4.889491821
A* on Graph with 100 nodes and 428 edges
    Average Space Required to All Paths From Starting Point A: 26.135821367
A* on Graph with 240 nodes and 1252 edges
    Average Time to Find All Paths From Starting Point A: 151.483799178 ms

Process finished with exit code 0
```



Here, the time complexity of **UCS** and **A* Search** depends on the **branching factor** and **number of nodes**. So, the **x axis is changed as the average branch factor**. (For branching factor = 4.5, another experiment (with 240 nodes and 1252 edges) is done, however its graph is not displayable. (too complex)). The first 3 samples are the graphs mentioned in previous sections.

Conclusion on Time Complexity

DFS performs better than **BFS** in some situations (this is because it reaches the target in a deeper level quickly). However, it has a main drawback which is in deeper graphs, depending on the child node selection it can need a longer time to reach a shallower node because it first has to complete a full traversal through an end, to start the next node in the same level. This situation also implies DFS to be not optimal (will be covered in detail later), since it may reach the target in a deep level and stop executing, where there might be another(shorter) path which it hasn't expanded yet. In such cases, BFS performs better and always finds the optimal path.

UCS and A* work similar to BFS by expanding the nodes in a manner that is similar to level order, but their main drawback in time performance is they use priority queues(a more complex data structure than Stack & Queues) and they also need to take into account some cost estimating functions, which increase the time overhead to select the next state.

2.2.- Space Complexity

In order to compare the algorithms, by the memory space they've used; I used the "**resource**" module in python.
The sample code is as follows:

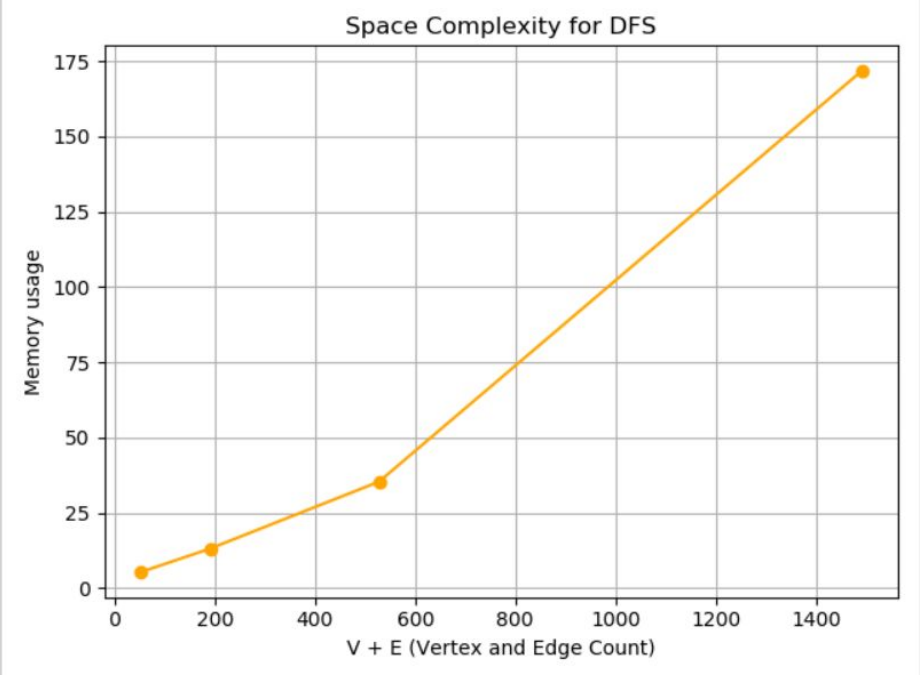
```
before = resource.getrusage(resource.RUSAGE_SELF)
matrix = txt_to_matrix("405 HW\\Simple Graph\\Adj. Matrix.txt")
adjacency_list = matrix_to_list(matrix)
bfs(adjacency_list, 0)
after= resource.getrusage(resource.RUSAGE_SELF)
used = after.ru_maxrss      - before.ru_maxrss
print("DFS on Graph with 100 nodes and 428 edges \n\tAverage Space Required to Find All Paths From Starting Point A:", used)
```

Again, the function calls are done 1000 times, on random target nodes, and their average result is taken into account.

This part is intentionally left blank.

2.2.1- DFS Space Complexity

```
DFS on Graph with 15 nodes and 37 edges
    Average Space Required to All Paths From Starting Point A: 5.32833e-06
DFS on Graph with 43 nodes and 148 edges
    Average Space Required to All Paths From Starting Point A: 1.315346e-05
DFS on Graph with 100 nodes and 428 edges
    Average Space Required to All Paths From Starting Point A: 3.536267e-05
DFS on Graph with 240 nodes and 1252 edges
    Average Space Required to All Paths From Starting Point A: 0.00017184901

Process finished with exit code 0
```

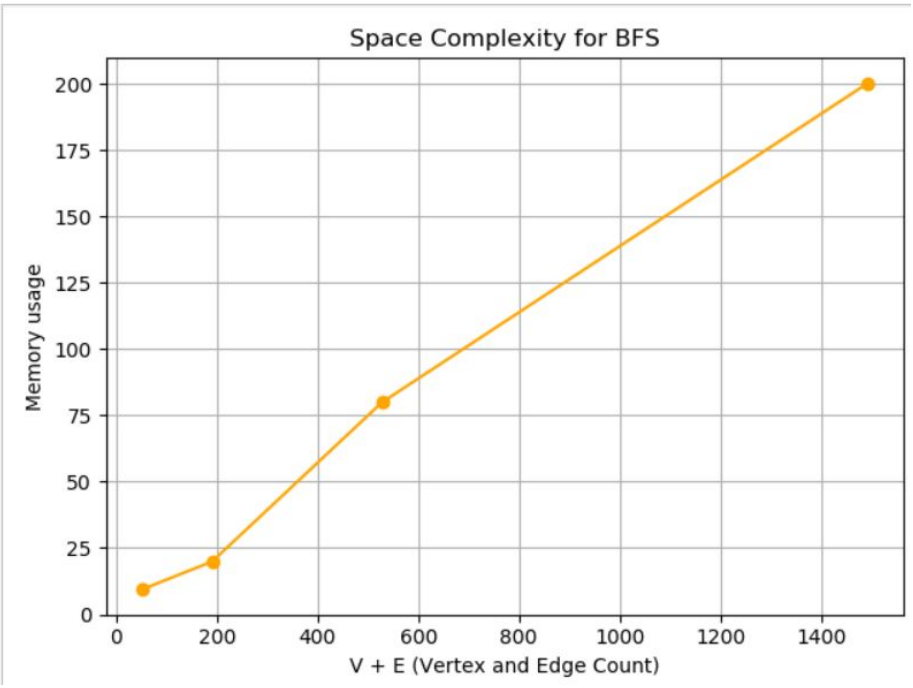


As it can be seen, there is a linear relationship between the node and edge counts, and the space complexity. This is feasible, since

- the nodes and edges are stored in adjacency list;
- visited paths stored in an array(the deeper the level is, the more storage required);
- and the recursive structure of DFS.

2.2.2- BFS Space Complexity

```
BFS on Graph with 15 nodes and 37 edges
    Average Space Required to All Paths From Starting Point A: 9.42833e-06
BFS on Graph with 43 nodes and 148 edges
    Average Space Required to All Paths From Starting Point A: 1.992346e-05
BFS on Graph with 100 nodes and 428 edges
    Average Space Required to All Paths From Starting Point A: 7.9885613e-05
BFS on Graph with 240 nodes and 1252 edges
    Average Space Required to All Paths From Starting Point A: 0.000200184091

Process finished with exit code 0
```



Similar things can be said about BFS. It uses a Queue data structure and every time it reaches a new node, adds it to the queue. So, the memory space required for BFS will be in a linear relationship with the number of nodes.

2.2.3- Space Complexity for UCS and A* Search

Similar to DFS and BFS, the space complexities are in a linear relationship with the node-edge numbers. I did not add their graphs to the report not to keep it longer.
The main difference between UCS & A* and BFS & DFS is that UCS and A* algorithms require more space, than the others. The main reason behind that is they use a different data structure (priority queue).
They also have a linear relationship between node number and space complexity but their memory overhead is higher.

2.3- Optimality

Optimality is the ability of the algorithm to find the best possible path.

- For BFS and DFS, the smallest number of edges,
- for UCS and A* Search the shortest path length.

For this purpose, let us consider the algorithms on **five-tac-toe** game.
In five-tac-toe game, for a particular step, there are 4 different options:

1. Move blank tile right
2. If can not move to right(boundary or visited)
    a. Move blank tile left
3. If can not move to left(boundary or visited)
    a. Move blank tile down
4. If can not move to down(boundary or visited)
    a. Move blank tile up
5. If can not move to up(boundary or visited)
    a. stop.

Even the order of these operations will change the results in the DFS. So, I will show 3 different possibilities on the step orders. (**First Order:** Up, down, left, right),(**Second Order:** Right, left, down, up), and (**Third Order**: Left, Up, Right, Down)

Before starting, I added some extra functionalities to the codes, so that the moves will be printed, the max. search depth will and number of expanded nodes will be stored.

Comparisons of Algorithms on Optimality

Initial State:

```
0 th step
_   +   0
+   0   +
0   +   +
```

Goal State:

```
26 th step
0   +   0
+   +   +
0   +   _
```

2.3.1- Optimality of DFS

- **DFS with First Move Order:**

```
DFS steps in order: Right Right Down Left Left Down Right Right Up Left Left Down Right
Right Up Left Left Up Right Right Down Left Left Down Right Right

Number of Nodes Visited: 27

Goal reached at Depth = 26

Process finished with exit code 0
```

- **DFS with Second Move Order:**

```
DFS steps in order: Down Down Right Up Up Left Down Down Right Up Up Left Down Down Right Up Up
Left Down Down Right Right Up Up Left Down Down Right Up Up Left Down Down Left

Number of Nodes Visited: 35

Goal reached at Depth = 34

Process finished with exit code 0
```

- **DFS with Third Move Order:**

```
DFS steps in order:
Down Down Right Right Up Up Left Down Down Right Up Up Left Down Down Right Up Up Left Down
Down Right Up Up Left Left Down Down Right Right Up Up Left Down Down Right Up Up Left Down
Right Up Left Down Down Right Up Up Left Down Down Right Up Up Left Left Down Down Right Right
Up Up Left Down Down Right Up Up Left Down Down Right Up Up Left Down Down Left Up Right
Down Right Up Up Left Down Down Right Up Up Left Down Down Right Up Left Down Right Up Up
Left Left Down Down Right Right Up Up Left Left Down Down Right Right Up Up Left Left Down Down
Right Right Up Up Left Left Down Down Right Right Up Up Left Down Down Left Up Up Right Down
Down Right Up Up Left Down Right Up Left Down Down Right

Number of Nodes Visited: 167

Goal reached at Depth = 152

Process finished with exit code 0
```

As it can be seen, the order of moves seriously effect the performance of DFS. And, for the same problem (initial states and goal states are same), DFS may find many different paths, depending on the child node selection. ***So, DFS is not Optimal.***

2.3.2- Optimality of BFS

In BFS, again the order of the movements may change the exact result, however the found paths will be in the same level(depth) on the graph, due to the logic behind BFS (It does not pass to the next level, until it finishes the current level)

Again, I will show 3 outputs corresponding to 3 move orders and show that the max. depth will be the same.

- **BFS with First Move Order**

```
BFS steps in order:
Right Down Up Left Right Down Right Down

Number of Nodes Visited: 136

Goal reached at Depth = 8

Process finished with exit code 0
```

- **BFS with Second Move Order**

```
BFS steps in order:
Right Down Left Up Right Down Right Down

Number of Nodes Visited: 136

Goal reached at Depth = 8

Process finished with exit code 0
```

- **BFS with Third Move Order**

```
BFS steps in order:
Right Down Up Left Right Down Right Down

Number of Nodes Visited: 136

Goal reached at Depth = 8

Process finished with exit code 0
```

As it can be seen, the movements may change, however the max. depth is the same. And in each way, the path with the smallest number of edges is selected. ***So, BFS is optimal.***

2.3.3- Optimality of UCS

- **UCS with First and Third Move Order**

```
UCS steps in order:
Right Down Left Up Right Down Right Down

Number of Nodes Visited: 136

Goal reached at Depth = 8

Process finished with exit code 0
```

- **UCS with Second Move Order**

```
UCS steps in order:
Right Down Left Up Right Down Right Down

Number of Nodes Visited: 144

Goal reached at Depth = 8

Process finished with exit code 0
```

The main difference between UCS and BFS is that UCS takes into account the path costs, while BFS takes into account the path count. Since they are in a correlation, UCS reached the goal state in same depth, with a little difference in number of visited nodes. ***So, UCS is optimal.***

This part is intentionally left blank.

2.3.4- Optimality of A* Search

The change in move orders for A* will change the next state, if the f(n) = h(n)+g(n) results are the same, for two different states. (The first found path will be selected, by implementation, however again, this will not effect the optimality of the algorithm).

- **A* Search with First and Third Move Order**

```
A* Search steps in order:
Down Right Up Left Down Right Down Right

Number of Nodes Visited: 132

Goal reached at Depth = 8

Process finished with exit code 0
```

- **A* Search with Second Move Order**

```
A* Search steps in order:
Right Down Left Up Right Down Right Down

Number of Nodes Visited: 132

Goal reached at Depth = 8

Process finished with exit code 0
```

As it can be seen, the first two steps are different but after 3rd state, the remainings are the same. The heuristic function was selected as the minimum sum of manhattan distances of the current state and the target state, and we can see that this heuristic function is **admissible**.

Let us investigate the effect of different heuristic functions:

- For this purpose, I will change the heuristic code in page 13
  - The manhattan distance closer to the center of the map will be more important.
  - The manhattan distance in the peripheries will be less important.

- **A* Search with Second Move Order, Heuristic Changed**

```
A* Search steps in order:
Right Right Down Left Down Right Up Up Left Left

Number of Nodes Visited: 158

Goal reached at Depth = 10

Process finished with exit code 0
```

As it can be seen, the change in the heuristic caused A* to fail to find the optimal path.

Admissibility and Consistency of Heuristic Functions in A*

Admissibility

In the previous page, an example of **inadmissible heuristic** function is shown. **The provement of admissibility** can be done by experiments like *that*, and **by formulating** the heuristic function. Now, let us prove that the chosen heuristic is admissible.

By definition, admissible heuristic implies that for every node **n, h(n) ≤ h*(n),** where h*(n) is the true cost to reach the goal state from n.

And the chosen heuristic function was the min. sum of manhattan distances. In order to prove that,

- There are 2 cases for every single '+':
    - Either it is on a correct position (in one of the coordinates <0,1>, <1,0>, <1,1>,<1,2>,<2,1>;
    - or not.
- If a '+' is in correct place, than the heuristic value does not increase.
- If it is in incorrect place, than the heuristic value will increase by the distance of closest non-occupied coordinate.
- So, the heuristic value will always be less than or equal to the actual cost.
- **If we change the heuristic** (i.e the example shown in previous page), it is not admissible anymore. (While the misplaced ones will increment the heuristic value by 1, the misplacements closer to the center will increment more than 1), this breaks the "less than or equal to" statement.

Consistency

In order to show the consistency of the heuristic function, I printed the current state's F() values in every selected state. The path costs are expected to be non-decreasing, if the heuristic is consistent.

- One example for **planar graph** is shown in **Appendix 1 and Appendix 2.**
    - According to this example, the path costs are approximated correctly.
    - And the remaining path's cost are not increasing.
    - So, our first heuristic (the alphabetical difference between node names) is consistent.
- In the five-tac-toe game, F(n) = g(n) + h(n) where g(n) is the cost of moving the blank tile (it is hyperparameter, selected as 1), and h(n) = h * min. sum of manhattan distances. (h is another tunable parameter).

- For an example in **five-tac-toe** game, the output when h = ⅓:

```
0 th step
Current F() Value :  1
+   _   O
+   O   +
O   +   +

1 th step
Current F() Value :  2
+   O   O
+   _   +
O   +   +

2 th step
Current F() Value :  3
+   O   O
_   +   +
O   +   +

3 th step
Current F() Value :  4
_   O   O
+   +   +
O   +   +

4 th step
Current F() Value :  5
O   _   O
+   +   +
O   +   +

5 th step
Current F() Value :  6
O   +   O
+   _   +
O   +   +

6 th step
Current F() Value :  6
O   +   O
+   +   +
O   _   +

7 th step
Current F() Value :  7
O   +   O
+   +   +
O   +   _
```

- Here, it can be seen that the current F value is non-decreasing. I am showing one single example here, but I tried tens of different combinations, the F() value does not decrease in any of them when h = ⅓ .
- **When parameter h is changed to another value** (say 1), the heuristic function becomes **inconsistent** for this graph. As an example below, it is not non-decreasing.

```
0 th step
Current F() Value :  5
+   _   O
+   O   +
O   +   +

1 th step
Current F() Value :  6
+   O   O
+   _   +
O   +   +

2 th step
Current F() Value :  6
+   O   O
_   +   +
O   +   +

3 th step
Current F() Value :  6
_   O   O
+   +   +
O   +   +

4 th step
Current F() Value :  7
O   _   O
+   +   +
O   +   +

5 th step
Current F() Value :  9
O   +   O
+   _   +
O   +   +

6 th step
Current F() Value :  7
O   +   O
+   +   +
O   _   +

7 th step
Current F() Value :  7
O   +   O
+   +   +
O   +   _
```

2.4- Completeness

- For the planar graph problems in (Part 1, 2), a solution always existed. And the algorithms were able to find it.
- So, the algorithms are complete, in planar graph problems.(Unless the depth is not limited. For example if we set the depth limit to 10, the DFS would fail to find the path in the second example).

- For the five-tac-toe game, the same situation is valid, but also if the initial state is given properly(e.g the number of ‘+’s, ‘_’s, ‘O’s are correct), the algorithms always find the solution, without stucking in dead ends and infinite loops.
    - I prevented infinite loops by storing the visited nodes in an array.
- Again, if the depth was limited, (i.e 25), DFS would fail to reach the goal state in the example in page 14.