**The implementations of the BFS, DFS, Greedy and A\* Search Algorithms with Python**

**BADS 7601 Artificial Intelligence**

**TERM PROJECT**

**By**

**Prasit Chulanutrakul 6120412007**

**Table of Contents**

[**1.** **Introduction** 2](#_Toc47802382)

[**2.** **Description of File and Directory** 3](#_Toc47802383)

[**3.** **The instruction “How to run the program”** 5](#_Toc47802384)

[**4.** **The drawing of the proposed testing graph** 8](#_Toc47802385)

[**5.** **The images capture of the program running** 10](#_Toc47802386)

[**6.** **Share Source Code** 25](#_Toc47802387)

[**Appendix: Input File Format** 29](#_Toc47802388)

# **Introduction**

In this project, 4 search algorithms are implemented with python language which are:

* 1. Breadth-First Search (BFS),
  2. Depth-First Search (DFS),
  3. Greedy best-first search
  4. A\* algorithm.

# **Description of File and Directory**

1. “bads7601\_project.py” file

This file contains the source code of the class name "Graph". This class contains a graph data structure and several groups of methods which describes below:

**Method for creating, manipulating, and viewing the graph data structure**

1. “add\_node()” method: Add graph's node and its neighbor nodes.
2. “add\_goal()” method: Add goal node for 'weighted\_graph'.
3. “remove\_node()” method: Remove graph's node.
4. “graph\_from\_file()” method: Create graph from file.
5. “get\_graph()” method: Get graph structure
6. “get\_graph\_type()” method: Get graph type

**Method for searching path**

1. “uninform\_search()” method: Search the graph using “BFS” or “DFS”
2. “inform\_search()” method: Search the graph using “Greedy” or “A\*”

**Method for visualize the result**

1. “print\_search\_result()” method: Display the result in string format
2. “plot\_g()” method: Plot the graph structure and search result.

For more details, how to implement this class to solve the problem, please read the docstrings of each method. Implemented of BFS, DFS Greedy best-first search, and A\* search algorithms

1. “Testing\_Program.ipynb” and “Testing\_Program.html” file

In this file contains the testing and example of class “Graph” which some part shows in section 5.2. Testing by implementing the "Graph" class on Jupyter Notebook

Testing\_Program.html is the exported version of Testing\_Program.ipynb in HTML format. (In case of your computer do not install the Jupyter Notebook)

1. “input” Folder

In this folder contains all input files for testing the program:

in.txt: In-class example (Slide Chapter3, P.15) input file

in\_aStar.txt: In-class example (Homework2, Ex1) input file

in\_inform.txt: In-class example (Slide Chapter4, P.6) input file

in\_state.txt: Proposed graph’s input file

1. “Generate\_Input” Folder

create\_graph.py: Script for creating propose graph

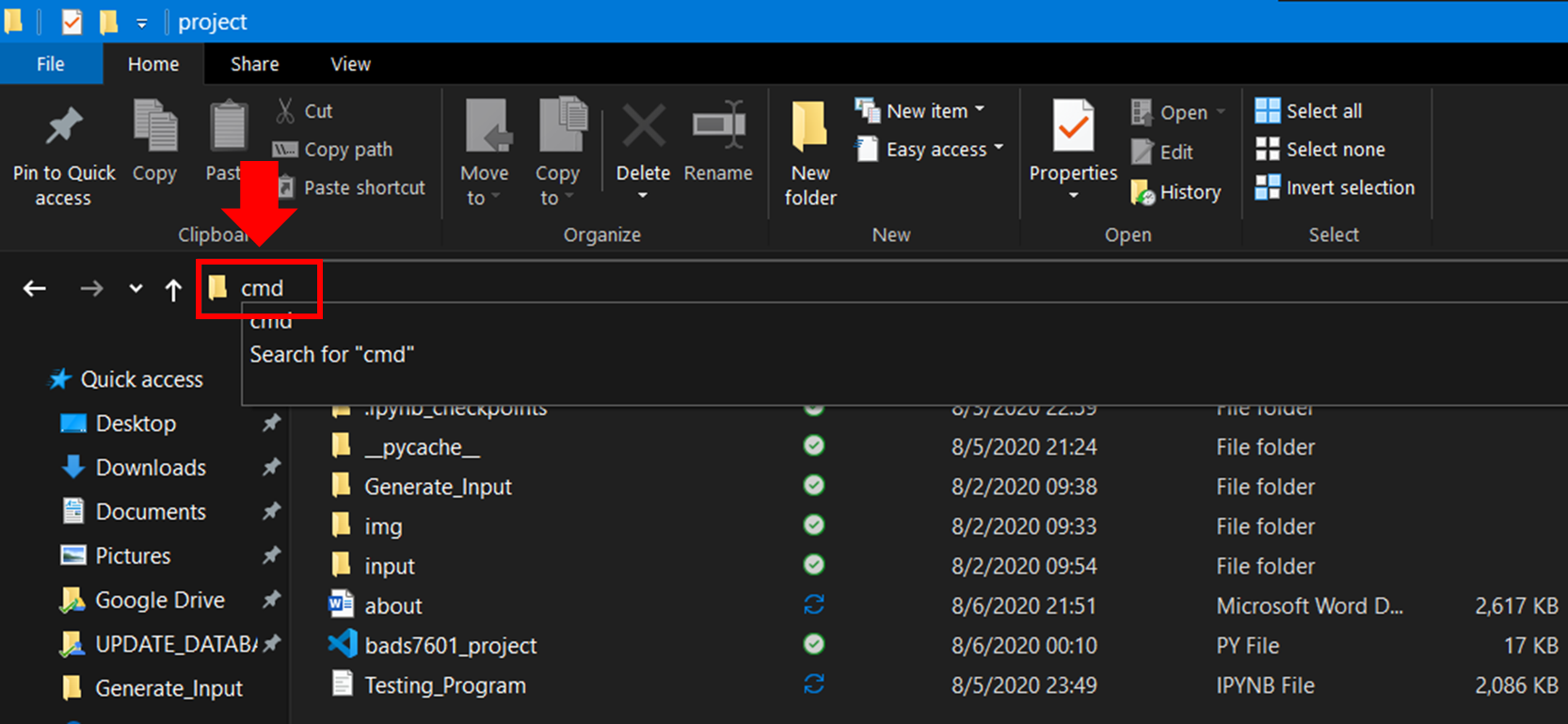
raw\_state\_graph folder: This folder contains raw data for creating propose graph

# **The instruction “How to run the program”**

There are two ways to apply the class “Graph” to solve the problems either directly run the bads7601\_project.py or apply the “Graph” class to the python code.

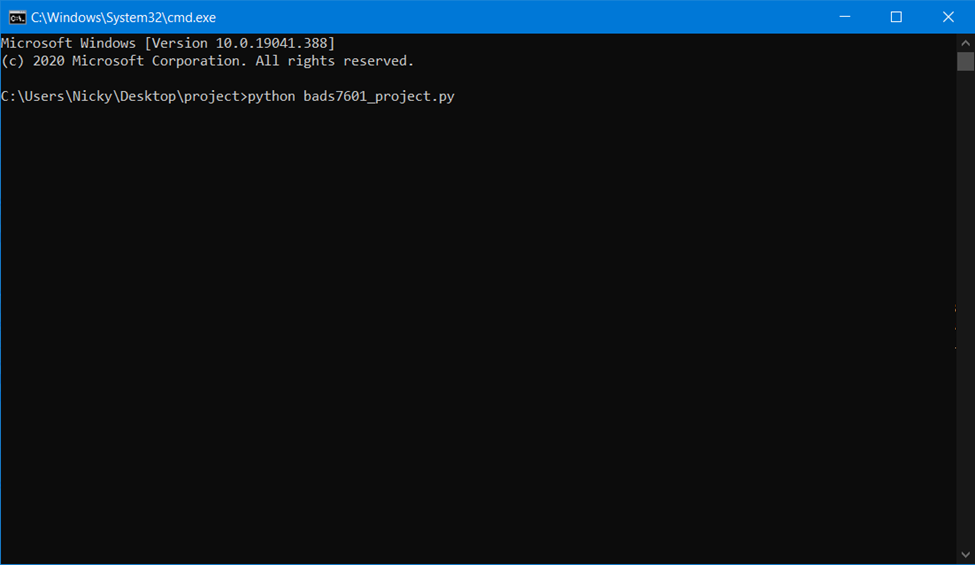
**3.1 Directly run the bads7601\_project.py**

1. To run the program, open command prompt on the directory which contains the “bads7601\_project.py” file, by typing “cmd” on the address bar and press enter.

****

2. The command prompt will appear as the picture below, then type

“python bads7601\_project.py” and press enter to run the program.



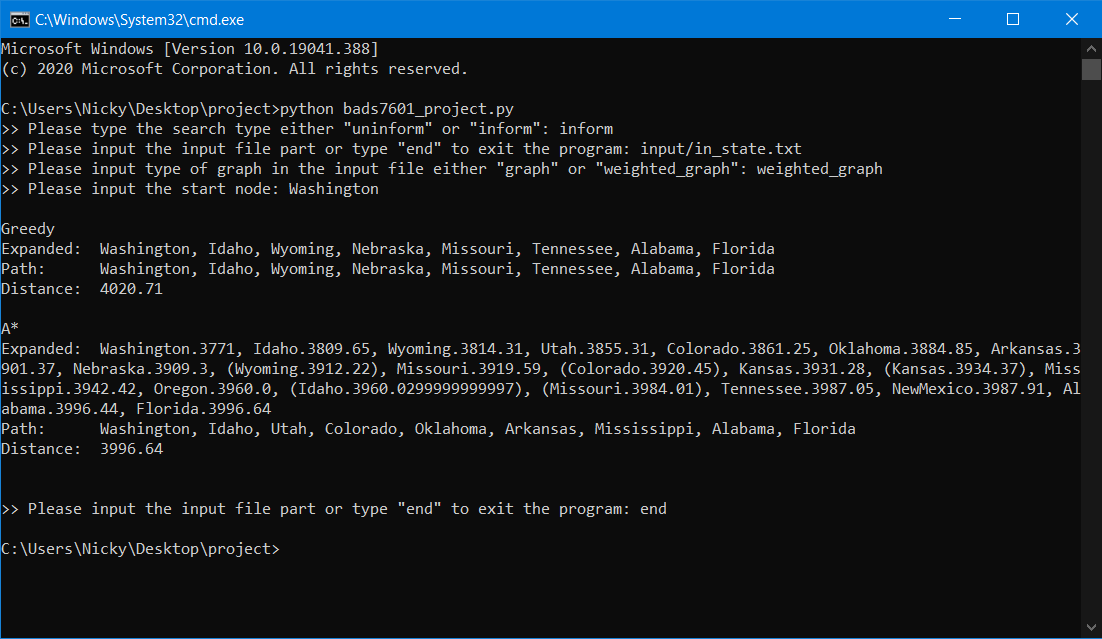
3. Input the parameters to the program:

- Input search type either “uninform” to search with BFS and DFS or “inform” to search with “Greedy” and “A\*”, then press enter.

- Input the input file path, then press enter.

- Input the graph type either “graph” or “weighted\_graph" as per the type of graph in the input file, then press enter.

- Input the start and end node if search type is “uninform” or input the start node if search type is “inform” then press enter, the result will show up as the figure below.



**Remark**

For “weighted\_graph” graph type can be searched with “uninform” and “inform” search type, but “graph” can be searched with “uninform” search type only.

**3.2 Implement the class “Graph” to solve the problem**

Before using the class “Graph” please ensure that the script or Jupyter Notebook file is in the same directory with “bads7601\_project.py”. Then import the “Graph” class from “bads7601\_project.py” into your script or Jupyter Notebook cell and then add the graph structure and finally apply the search method to search the data within the graph. The example codes are below.

**Example**

**Source**

from bads7601\_project import Graph

g = Graph('graph', verbose=False)

g.graph\_from\_file('input/in.txt')

g.plot\_g(g, size=(10, 5))

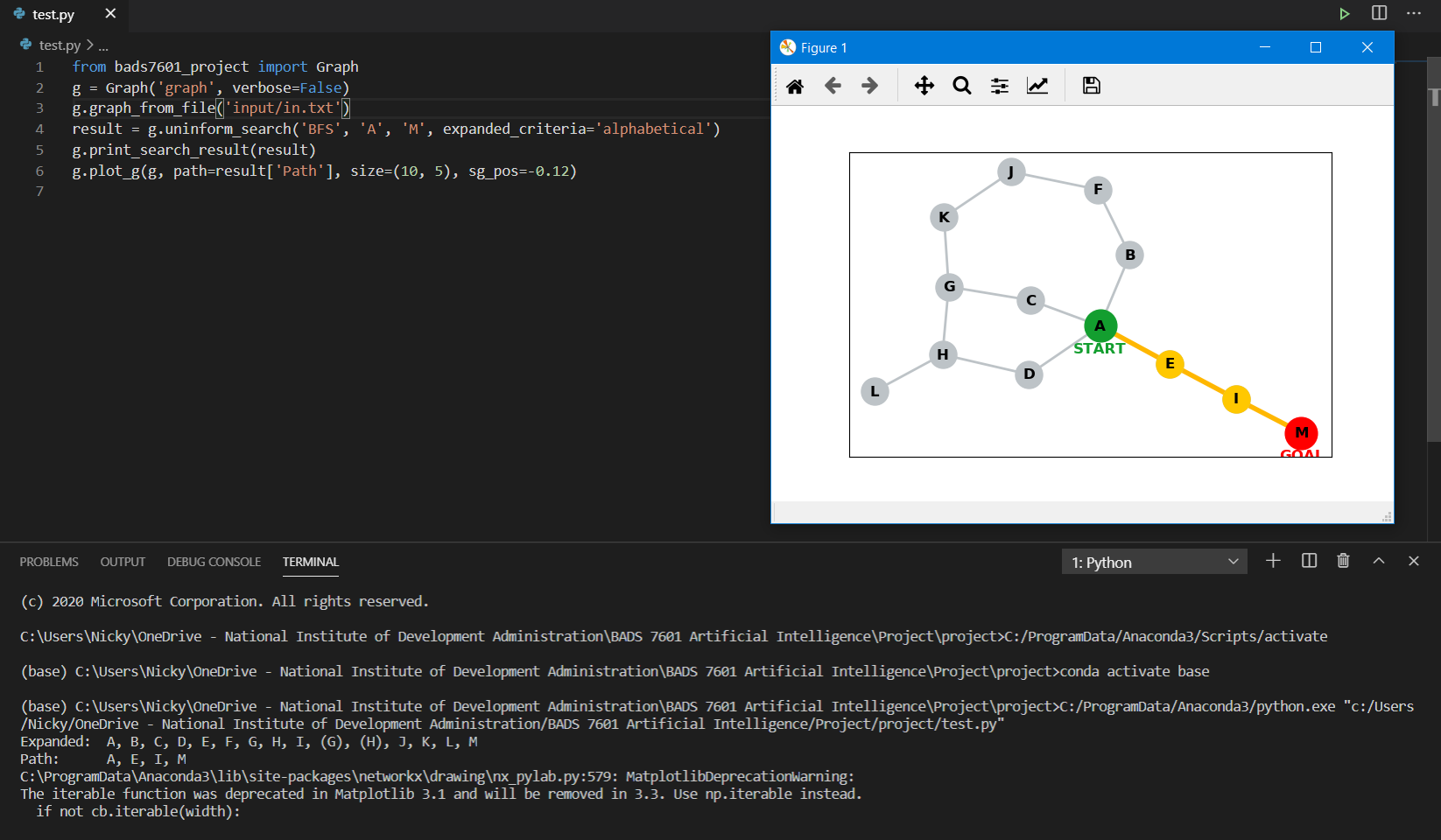
g.get\_graph()

result = g.uninform\_search('BFS', 'A', 'M', expanded\_criteria='alphabetical')

g.print\_search\_result(result)

g.plot\_g(g, path=result['Path'], size=(10, 5), sg\_pos=-0.12)

**Expected Result**

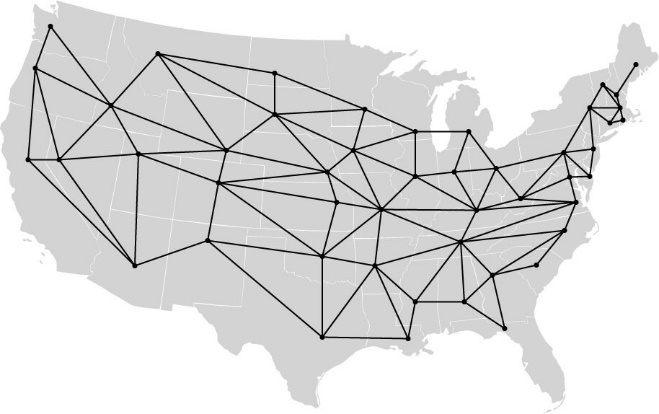
****

**Remark**

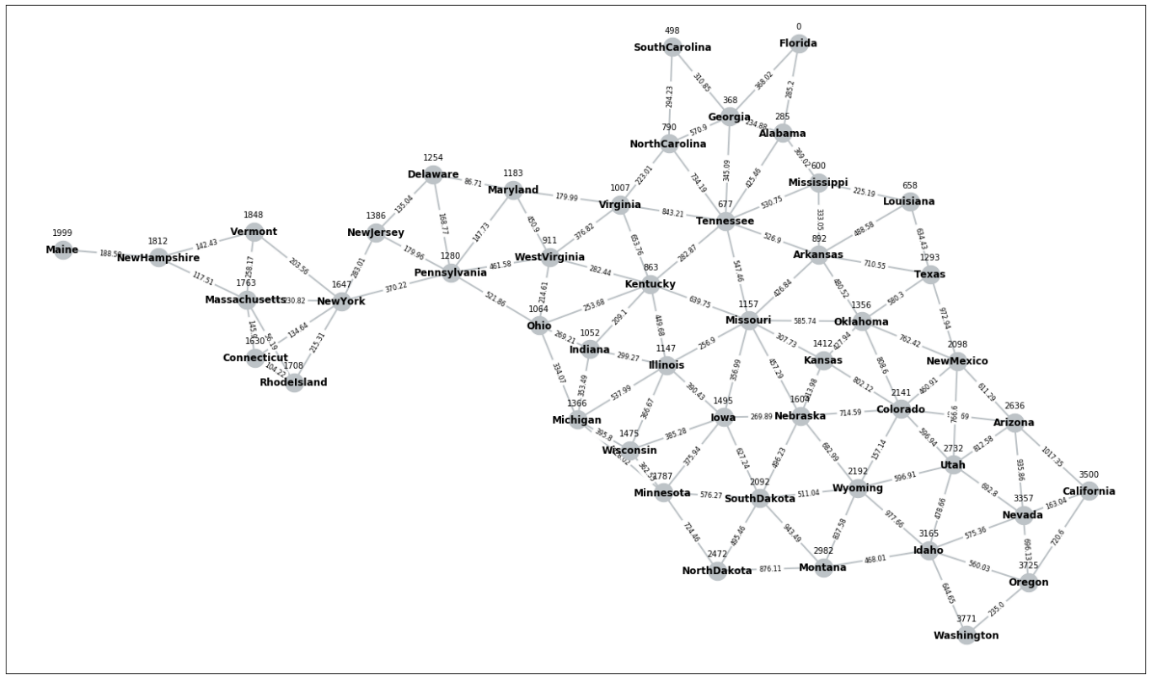
1. Not only using the method “graph\_from\_file()” to add the graph structure but also using “add\_node()” method to add graph's node and its neighbor nodes manually.

2. For more examples, please see the example source code in the Testing\_Program.ipynb (Please open this file with Jupyter Notebook) or Testing\_Program.html which is then exported version of Testing\_Program.ipynb in HTML format.

# **The drawing of the proposed testing graph**



The United States of America consists of 50 states but for the proposed graph for testing the algorithms, consist only 48 states, excluding Alaska and Hawaii which are not in the mainland. Thus, the proposed graph has 48 nodes and 110 edges. The figure below shows the plot of the proposed graph from the input file: in\_state.txt in the input folder (Plot with network package).

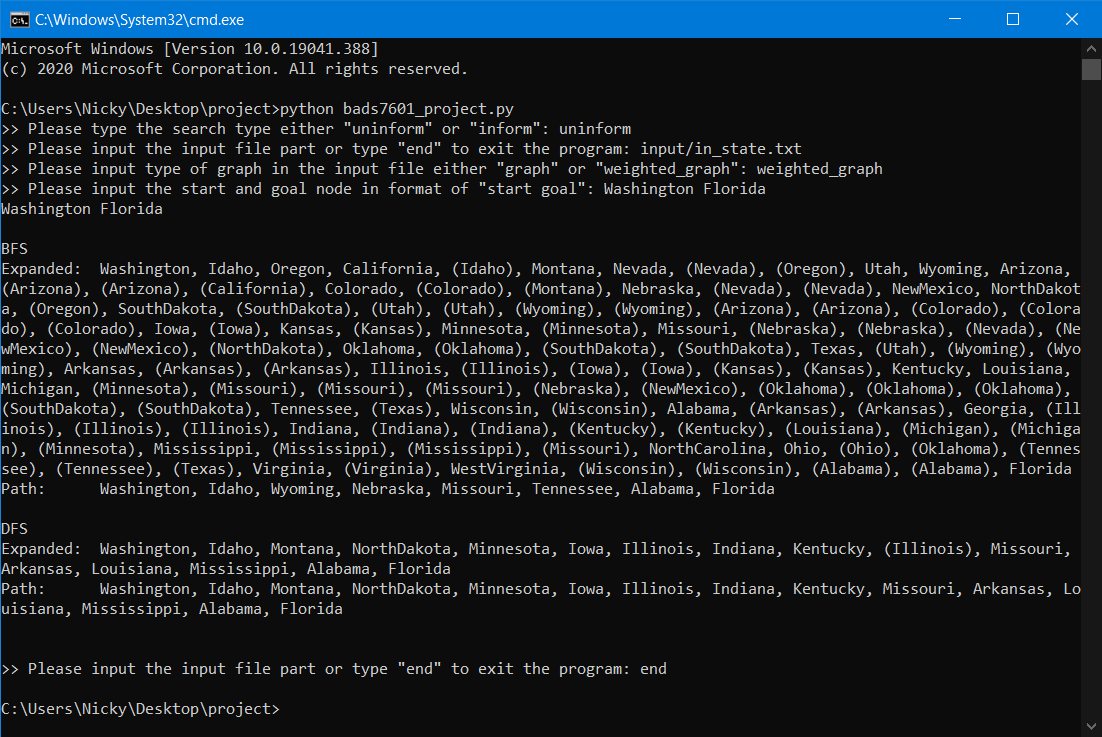


To create the states graph, the information for nodes and edges is from the link: <https://state.1keydata.com/bordering-states-list.php> which contains the state and its bordering states data. For the distance between the state (edge’s weight) and heuristic value is a displacement between two neighbor states and a displacement between the state and goal state, both values calculated using the latitude and longitude data form the link: <http://www.xfront.com/us_states/>. After gathering all necessary information and convert the data into the proper format, the special script (The script “create\_graph.py” and the raw input information files are in the “Generate Input” folder) is developed to generate the input file for the program. Indeed, the script can create the input file for an arbitrary state to goal state but for the simple way to test the search algorithms, the input file in which the goal state is “Florida” is used for demonstration.

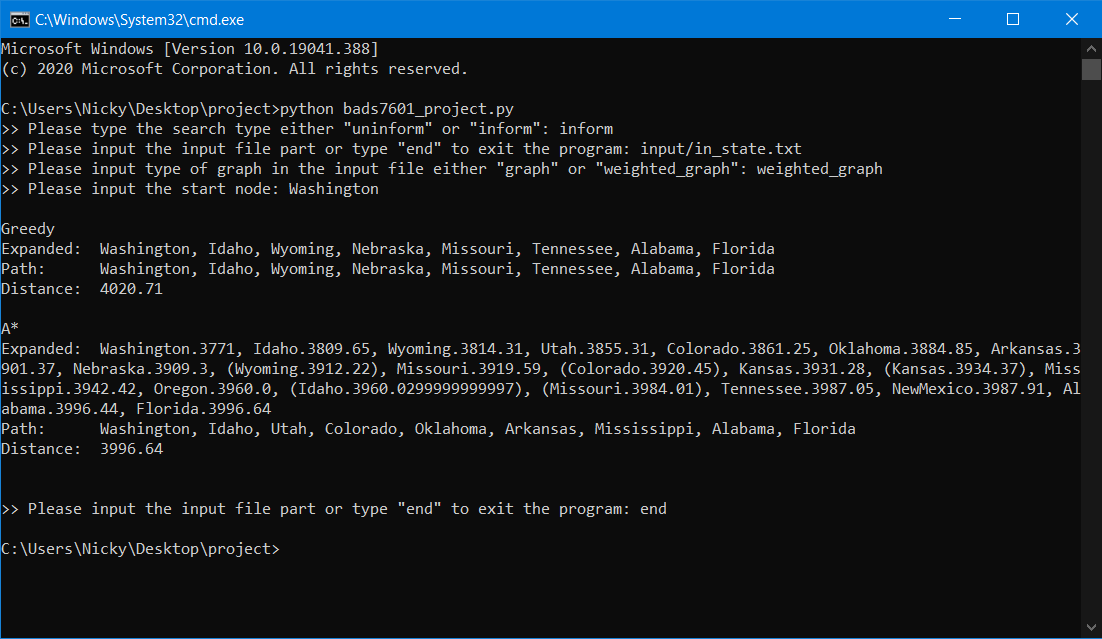
# **The images capture of the program running**

**5.1 Running the bads7601\_project.py directly**

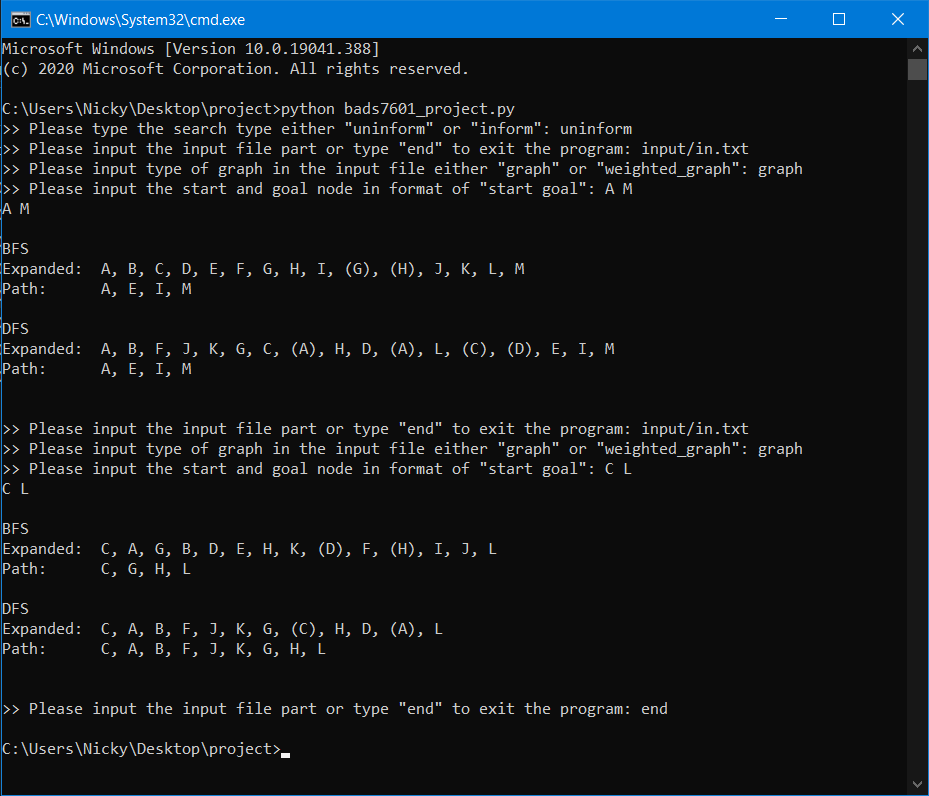
**5.1.1 Proposed Graph: Uninform Search**



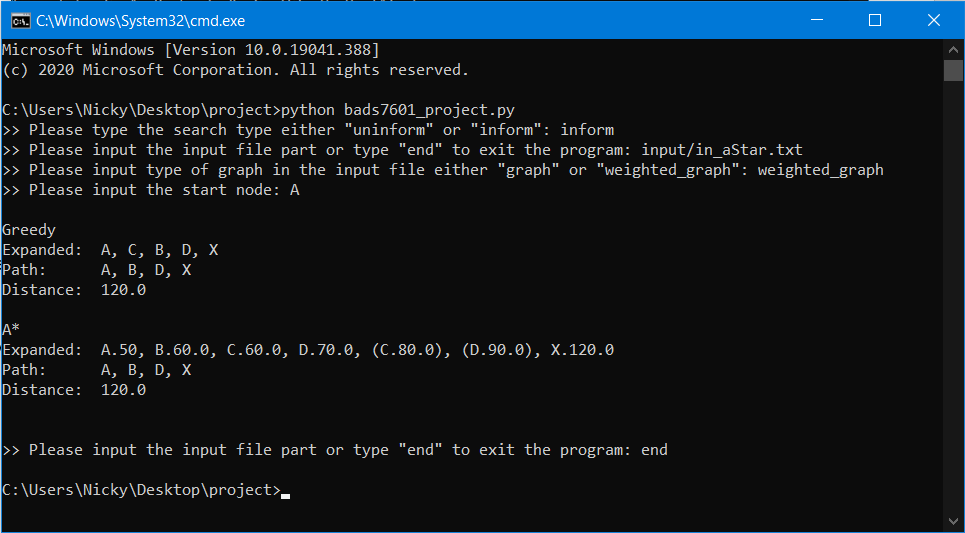
**5.1.1 Proposed Graph: Inform Search**



**5.1.3 Sample Run: Uninform Search**



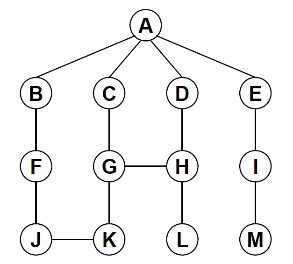
**5.1.4 Sample Run: Inform Search**

****

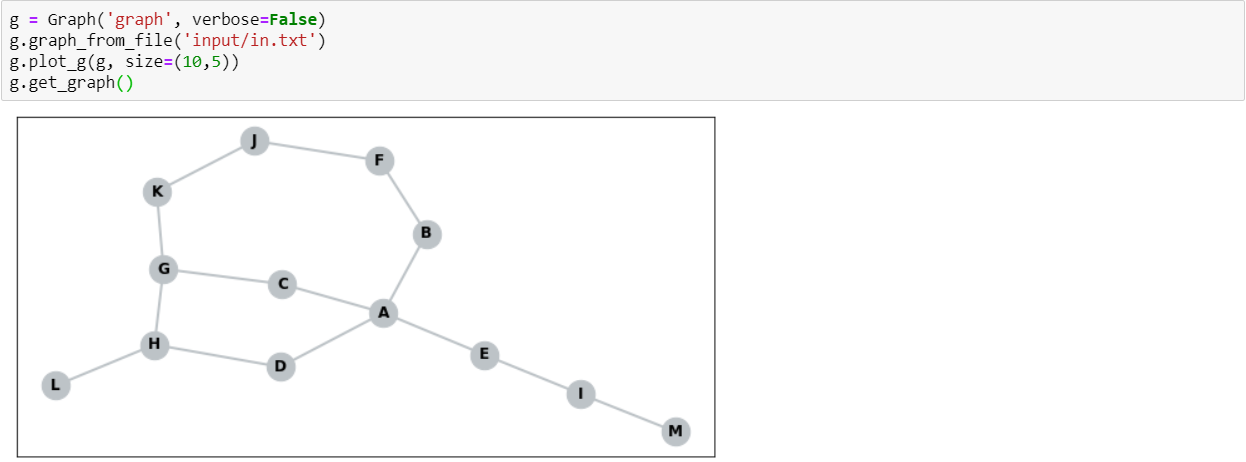
**5.2. Testing by implementing the “Graph” class on Jupyter Notebook**

All of the testing in this section are in the Testing\_Program.ipynb (Please open this file with Jupyter Notebook) and Testing\_Program.html (Exported content of Testing\_Program.ipynb in HTML format)

**5.2.1 In-Class Example: Uninform Search**



**Input**

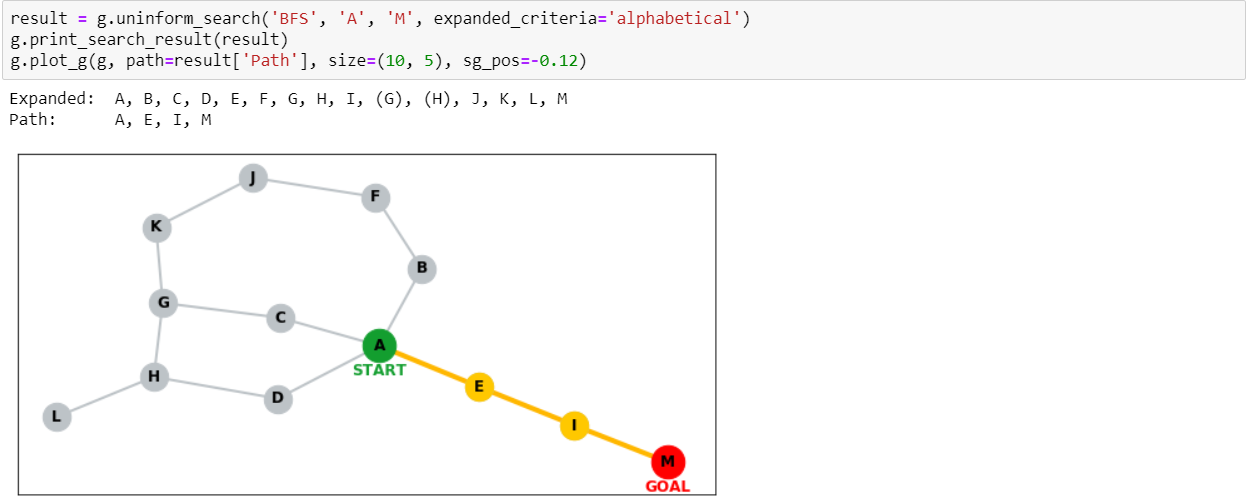


**BFS**

Expected result for BFS A TO M:

Path: A E I M

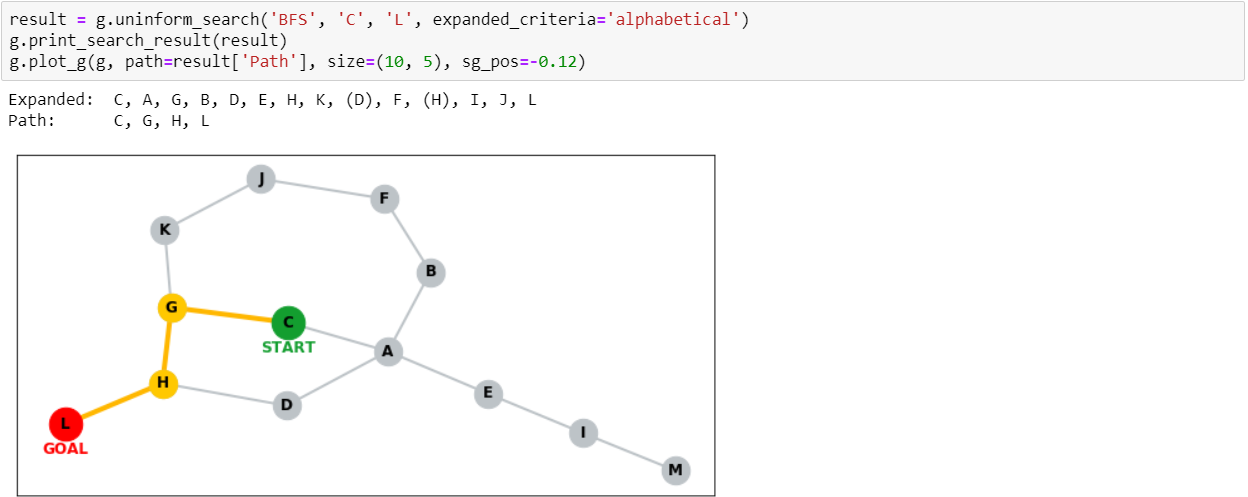
Expand: A B C D E F G H I (G) (H) J K L M



Expected result for BFS C TO L:

Path: C G H L

Expand: C A G B D E H K (D) F (H) I J L

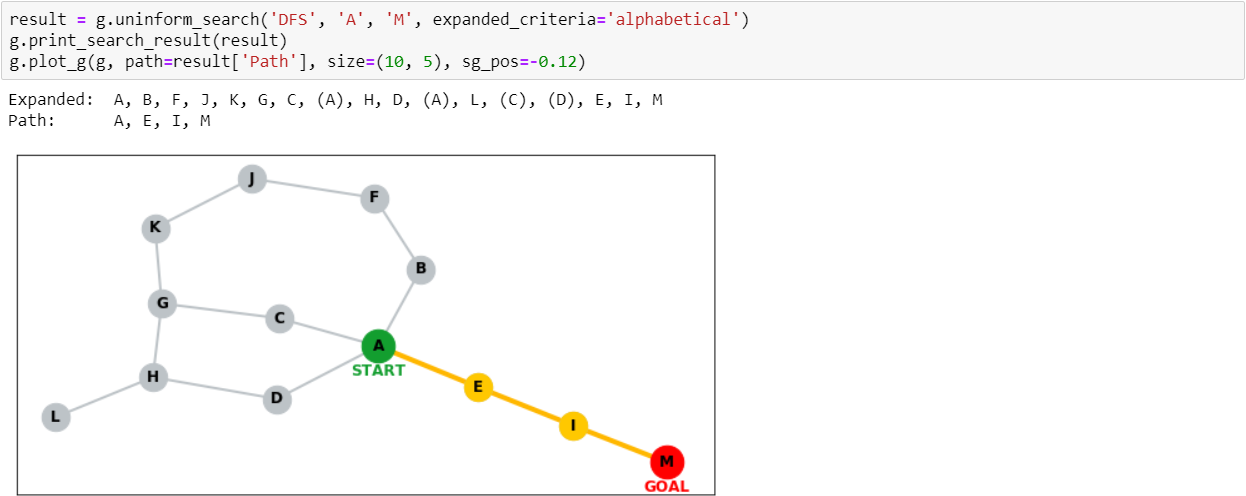


**DFS**

Expected result for DFS A TO M:

Path: A E I M

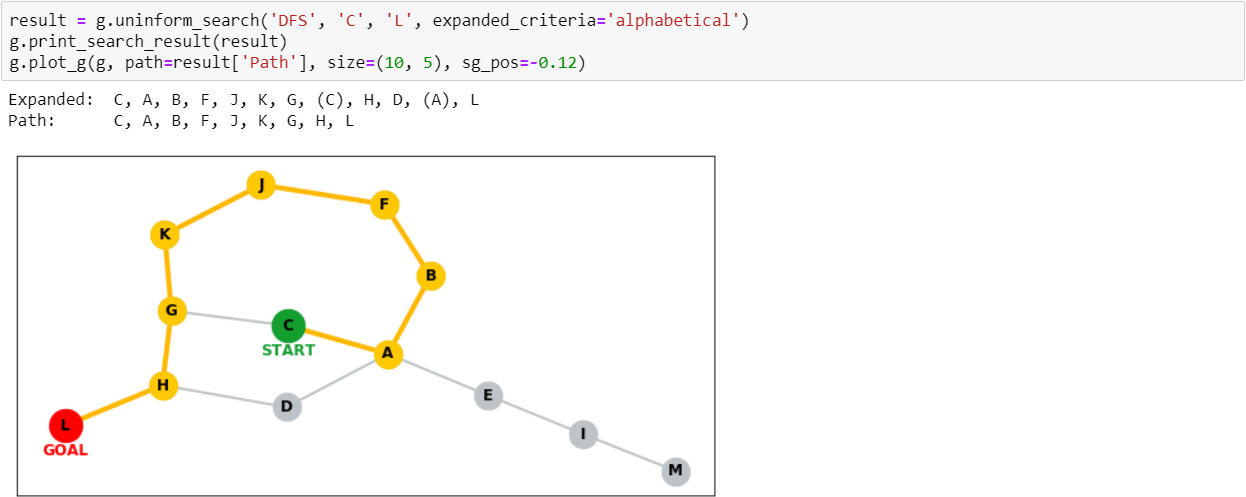
Expand: A B F J K G C (A) H D (A) L (C) (D) E I M



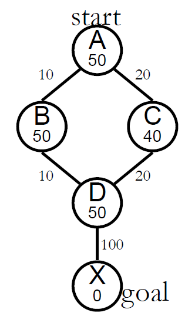
Expected result for DFS C TO L:

Path: C A B F J K G H L

Expand: C A B F J K G (C) H D (A) L



**5.2.2 In-Class Example: Uninform Search**



**Input**



**Greedy**

Expected result for A\* A TO X:

Path: A B D X

Expand: A C B D X

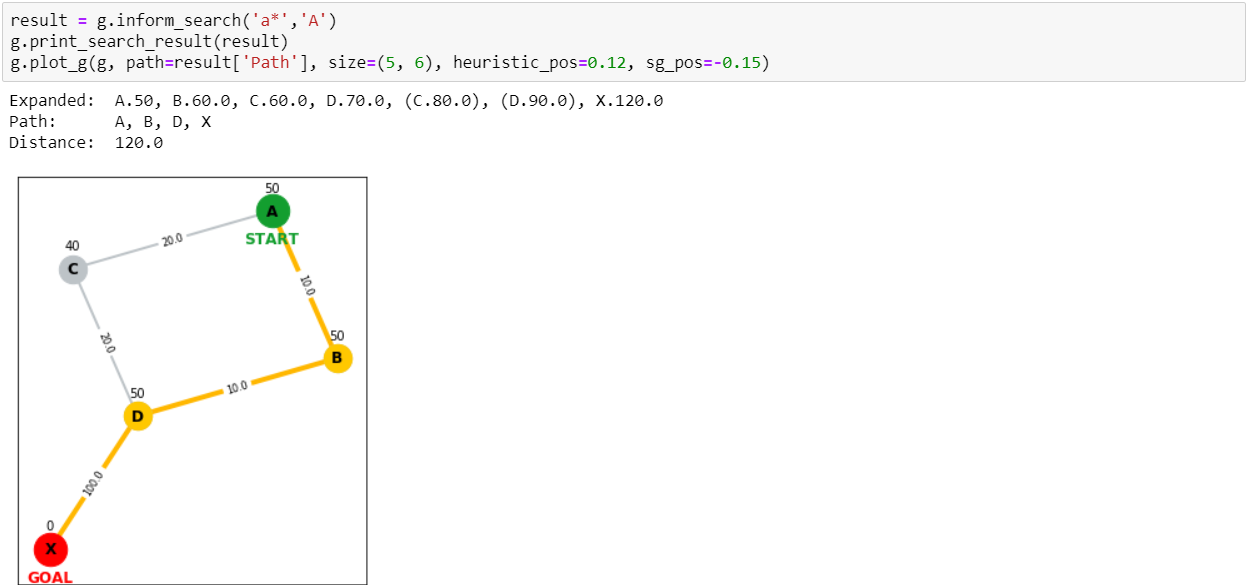
****

**A\***

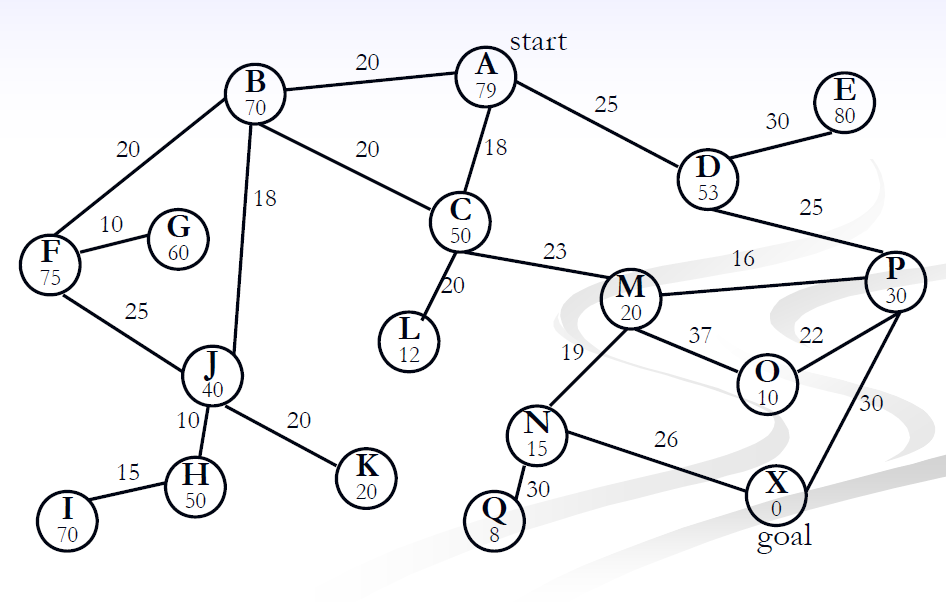
Expected result for A\* A TO X:

Path: A B D X

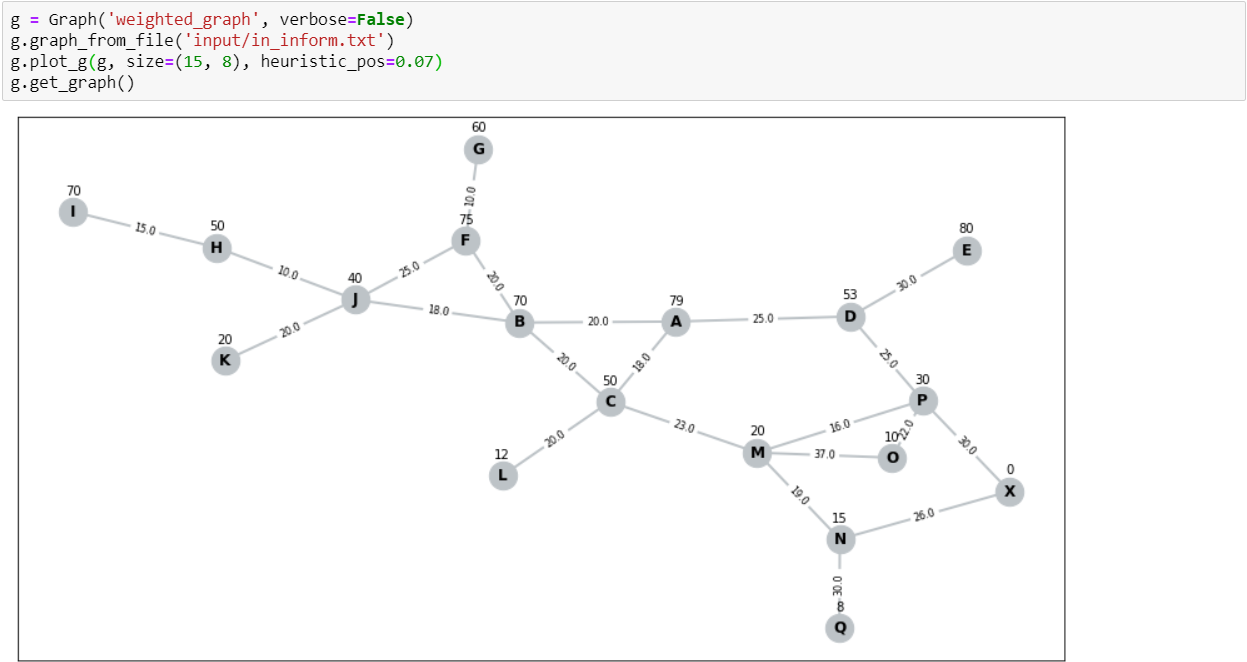
Expand: A.50 B.60 C.60 D.70 (C.80) (D.90) X.120

****

**5.2.3 In-Class Example: Uninform Search**



**Input**

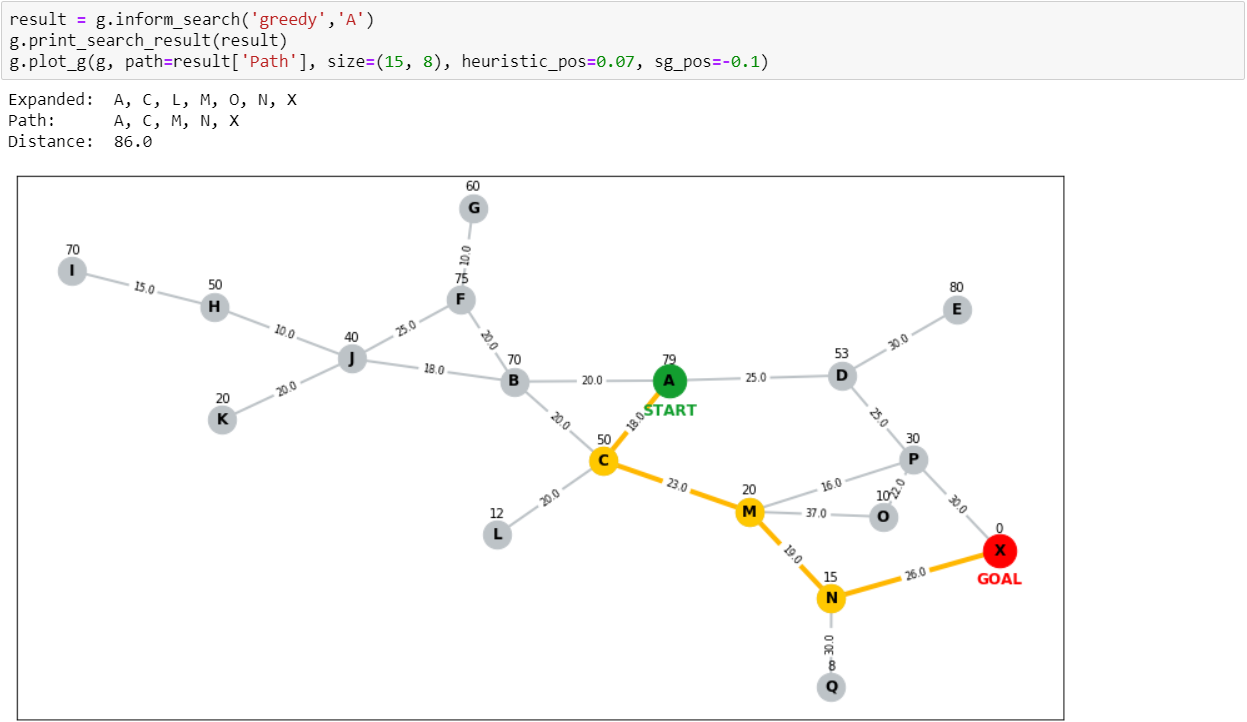
****

**Greedy**

Expected result for Greedy A TO X:

Path: A C M N X

Expand: A C L M O N X

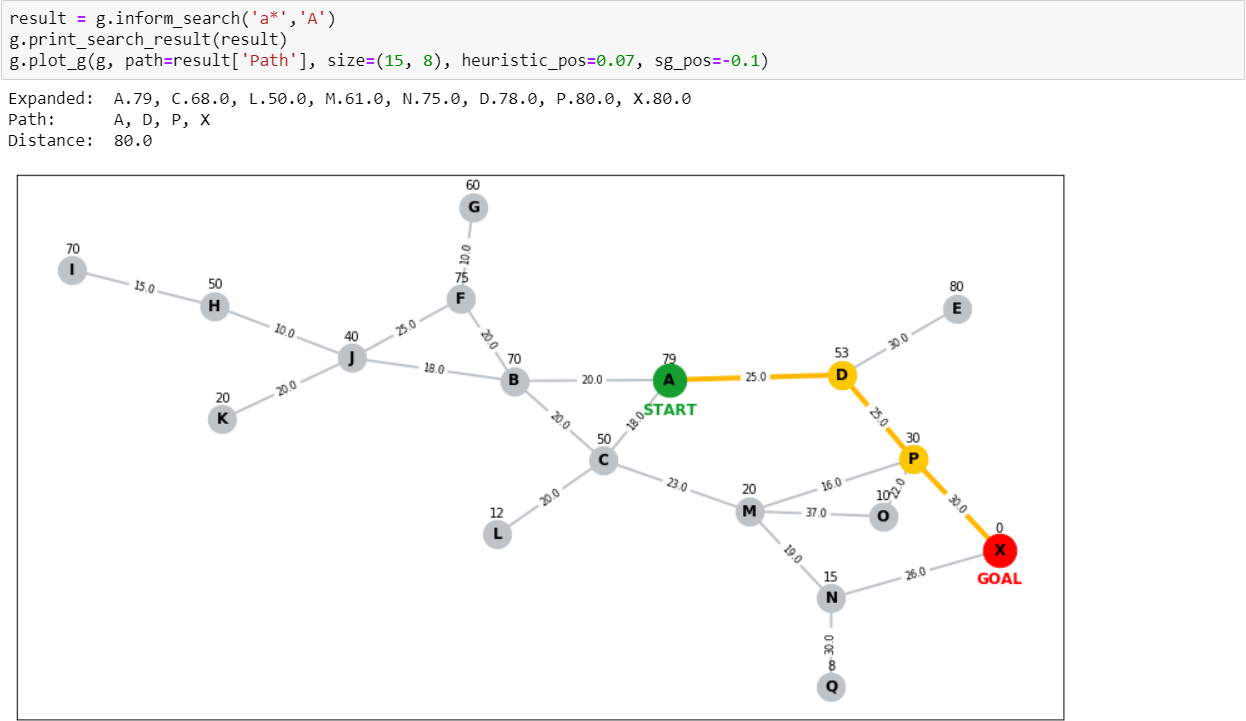
****

**A\***

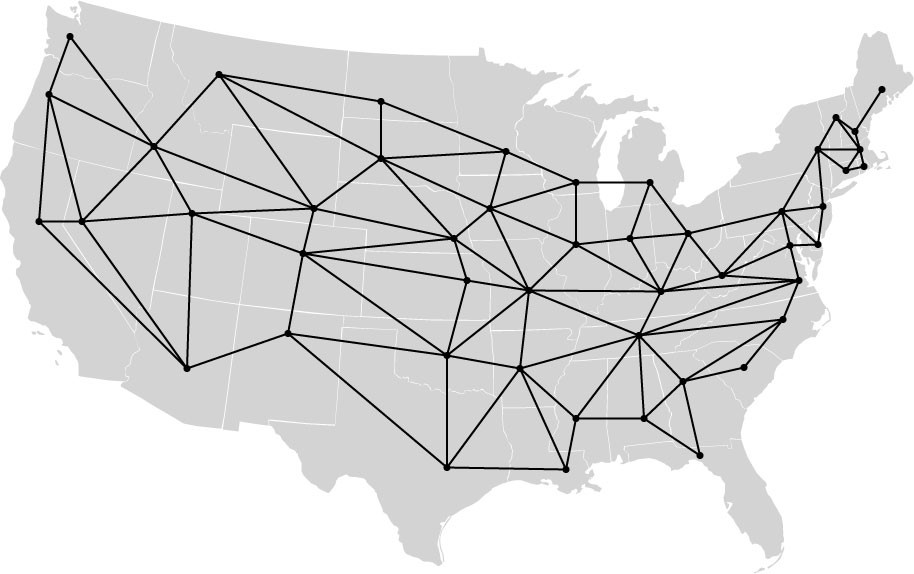
Expected result for A\* A TO X:

Path: A D P X

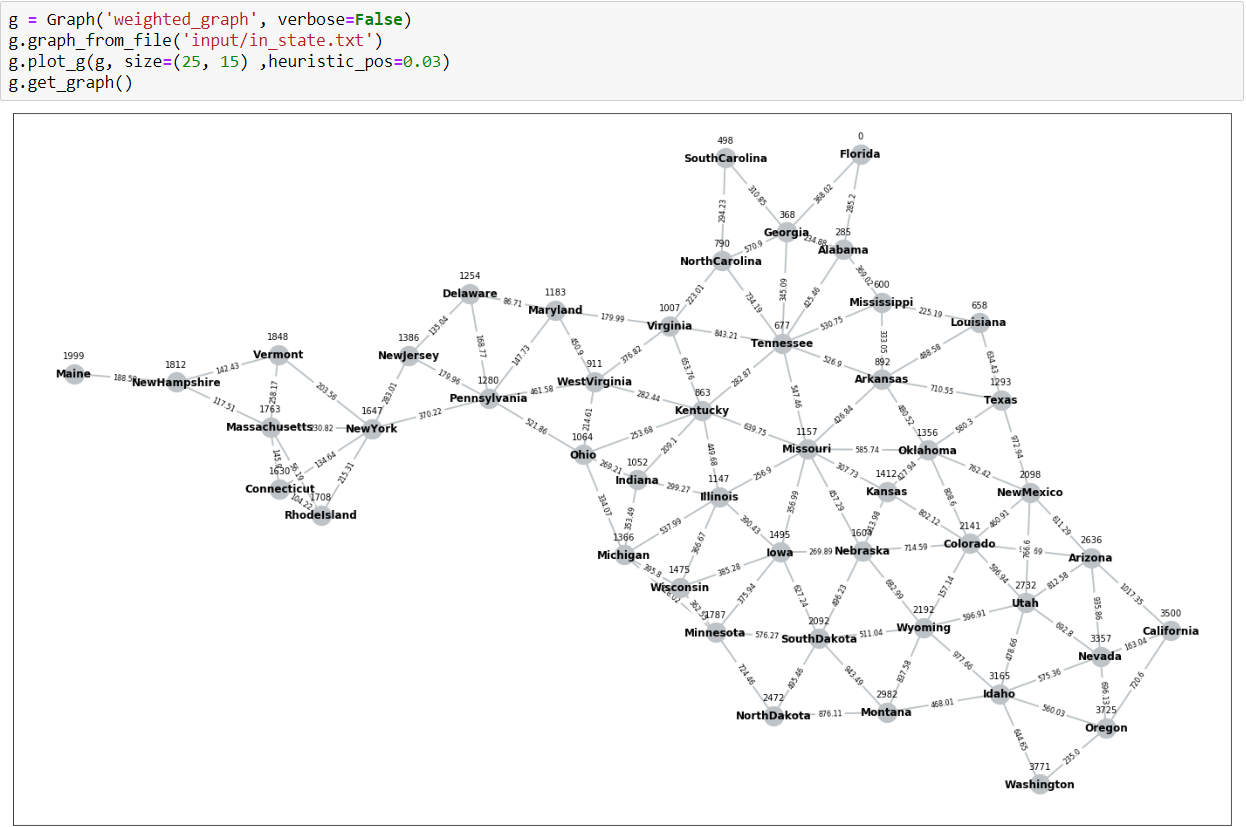
Expand: A.79 C.68 L.50 M.61 N.75 D.78 P.80 X.80

****

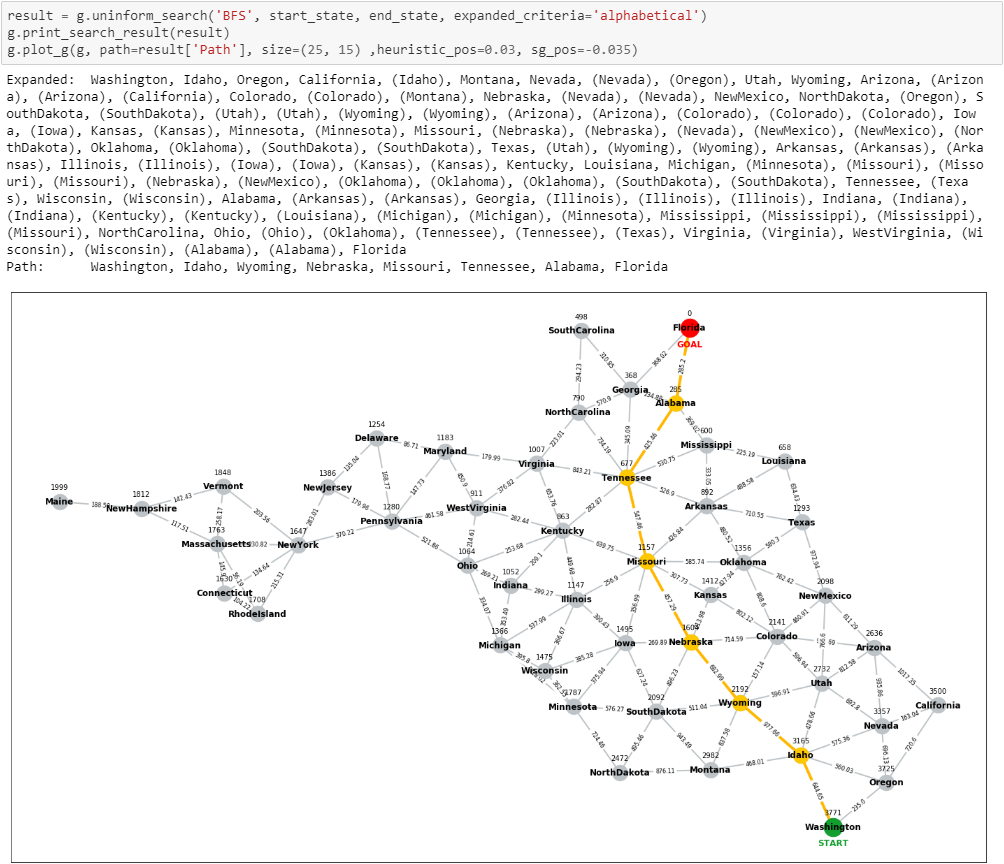
**5.2.4 Propose Graph**



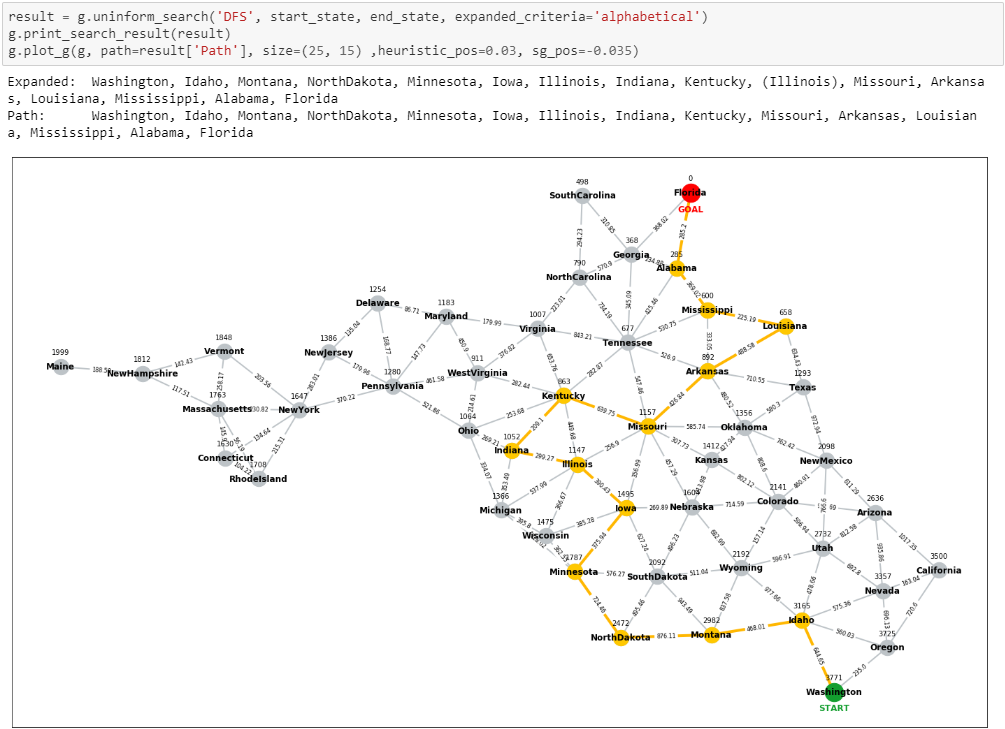
**Input**

****

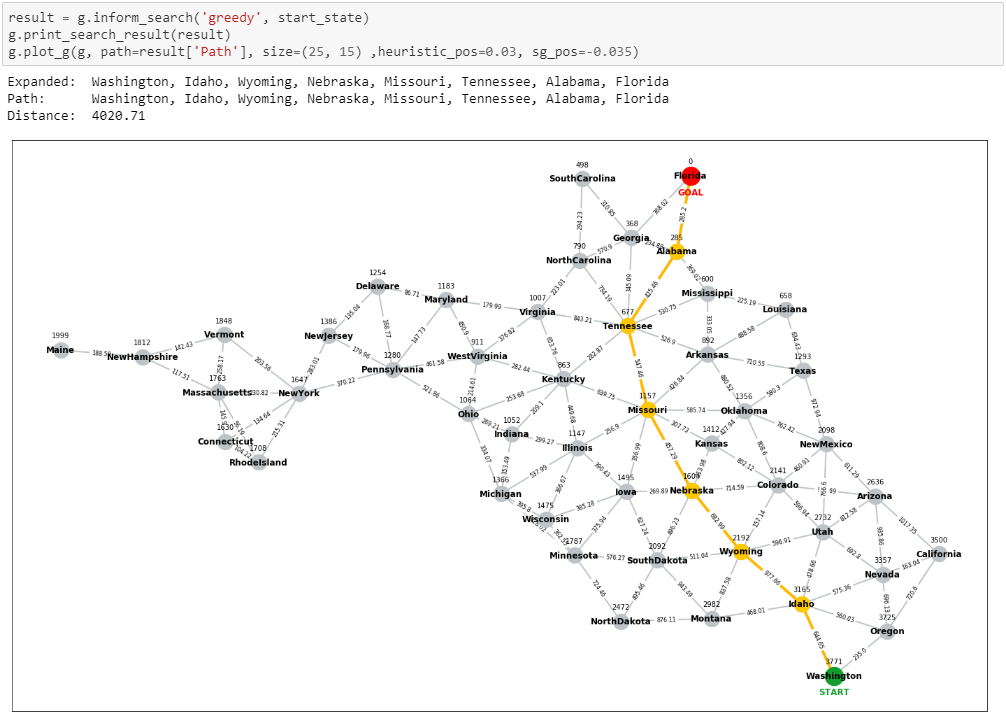
**BFS**

****

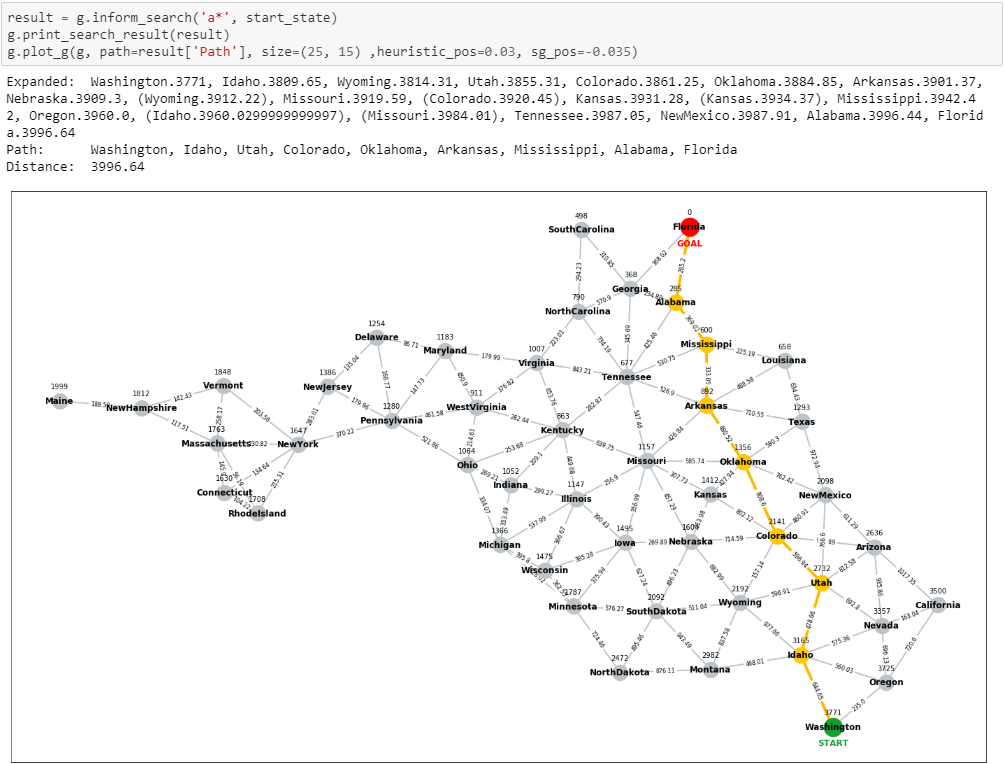
**DFS**

****

**Greedy**

****

**A\***

****

# **Share Source Code**

**3.1 Share source code between BFS&DFS**

    def uninform\_search(self, algorithm:str, start:str, goal:str, expanded\_criteria='alphabetical'):

        '''

        Uninform search

        Parameters: algorithm : {'BFS', 'DFS'}

                    start : str, name of start node

                    goal  : str, name of goal node

                    expanded\_criteria : {'alphabetical', 'reverse\_alphabetical', 'parent'}

        Returns:    str, Search result

        '''

        # Check arguments

        if algorithm not in ['BFS', 'DFS']:

            raise ValueError('algorithm must be "BFS" or "DFS" only.')

        if expanded\_criteria not in ['alphabetical', 'reverse\_alphabetical', 'parent']:

            raise ValueError('expanded\_criteria must be "alphabetical", "reverse\_alphabetical" or "parent" only.')

        # Initial queue

        # Member of queue consists of itself, it's parent, depth of search

        queue = [[start, None, 1]]

        # Initial visited

        visited = OrderedDict()

        expanded = list()

        while queue:

            # Print if verbose is true.

            if self.verbose:

                print('Queue: ', queue)

                print('Visited: ', visited)

            # Stop searching if the goal was expanded.

            if goal in visited:

                break

            # Getting the node to be expanded from queue

            node\_to\_expand = queue.pop(0)

            # Check the node whether it was expanded or was not expanded?

            if node\_to\_expand[0] not in visited:

                # Expanding node

                expanded\_nodes = list(self.\_\_graph[node\_to\_expand[0]].keys())

                # Add the node which was just expanded and it parent into visited.

                visited[node\_to\_expand[0]] = node\_to\_expand[1]

                expanded.append(node\_to\_expand[0])

                # Remove parent node

                if node\_to\_expand[1] in expanded\_nodes:

                    expanded\_nodes.remove(node\_to\_expand[1])

                if algorithm == 'BFS':

                    # --------------------------------------------------------------------------------

                    # QUEUE

                    # --------------------------------------------------------------------------------

                    for expanded\_node in expanded\_nodes:

                        queue.append([expanded\_node, node\_to\_expand[0], node\_to\_expand[2] + 1])

                    # Expanded criteria for BFS

                    if expanded\_criteria == 'alphabetical':

                        queue = sorted(queue, key = lambda x: (x[2], x[0]), reverse=False)

                    elif expanded\_criteria == 'reverse\_alphabetical':

                        queue = sorted(queue, key = lambda x: (x[2], x[0]), reverse=True)

                    else: # 'parent'

                        pass

                    # --------------------------------------------------------------------------------

                elif algorithm == 'DFS':

                    # --------------------------------------------------------------------------------

                    # STACK

                    # --------------------------------------------------------------------------------

                    # Expanded criteria for DFS

                    if expanded\_criteria == 'alphabetical':

                        sorted\_expanded\_nodes = sorted(expanded\_nodes, reverse=True)

                    elif expanded\_criteria == 'reverse\_alphabetical':

                        sorted\_expanded\_nodes = sorted(expanded\_nodes, reverse=False)

                    else: # 'parent'

                        sorted\_expanded\_nodes = expanded\_nodes.copy()

                    for expanded\_node in sorted\_expanded\_nodes:

                        queue.insert(0, [expanded\_node, node\_to\_expand[0], node\_to\_expand[2] + 1])

                    # --------------------------------------------------------------------------------

            else:

                expanded.append(f'({node\_to\_expand[0]})')

        return {'Expanded':expanded, 'Path':self.\_\_path(visited, goal)}

**3.2 Share source code between Greedy and A\***

    def inform\_search(self, algorithm:str, start:str):

        '''

        Inform search

        Parameters: algorithm : {'a\*', 'greedy'}

                    start : str, name of start node

        Returns:    str, Search result

        '''

        if self.\_\_graph\_type != 'weighted\_graph':

            raise TypeError('"graph\_type" must be weighted\_graph.')

        # Check arguments

        if algorithm not in ['a\*', 'greedy']:

            raise ValueError('algorithm must be "a\*" or "greedy" only.')

        # Initial queue

        # Member of queue consists of

            # 0 itself,

            # 1 it's parent,

            # 2 depth of search,

            # 3 sum\_distance,

            # 4 heuristic,

            # 5 sum\_distance + heuristic

        queue = [[start, None, 1, 0, self.\_\_heuristic[start], self.\_\_heuristic[start]]]

        # Initial visited

        visited = OrderedDict()

        expanded = list()

        while queue:

            # Print if verbose is true.

            if self.verbose:

                print('\n')

                print('Queue: ', queue)

                print('\n')

                print('Visited: ', visited)

            # Stop searching if the goal was expanded.

            if self.\_\_goal in visited:

                break

            # Getting the node to be expanded from queue

            if algorithm == 'a\*':

                index\_to\_pop = min(range(len(queue)), key = lambda i: (queue[i][5], queue[i][0], queue[i][1]))

            elif algorithm == 'greedy':

                index\_to\_pop = min(range(len(queue)), key = lambda i: (queue[i][4], queue[i][0], queue[i][1]))

            node\_to\_expand = queue.pop(index\_to\_pop)

            # Check the node whether it was expanded or was not expanded?

            if node\_to\_expand[0] not in visited:

                # Expanding node

                expanded\_nodes = self.\_\_graph[node\_to\_expand[0]].copy()

                # Add the node which was just expanded and it parent into visited.

                visited[node\_to\_expand[0]] = node\_to\_expand[1]

                if algorithm == 'a\*':

                    expanded.append(f'{node\_to\_expand[0]}.{node\_to\_expand[5]}')

                elif algorithm == 'greedy':

                    expanded.append(f'{node\_to\_expand[0]}')

                # Remove parent node

                if node\_to\_expand[1] in expanded\_nodes:

                    del expanded\_nodes[node\_to\_expand[1]]

                # QUEUE

                for expanded\_node in expanded\_nodes:

                    sum\_distance = node\_to\_expand[3] + expanded\_nodes[expanded\_node]

                    queue.append([expanded\_node, # itself

                                    node\_to\_expand[0], # it's parent

                                    node\_to\_expand[2] + 1, # depth of search

                                    sum\_distance, # sum\_distance

                                    self.\_\_heuristic[expanded\_node],

                                    sum\_distance + self.\_\_heuristic[expanded\_node]

                                ])

            else:

                if algorithm == 'a\*':

                    expanded.append(f'({node\_to\_expand[0]}.{node\_to\_expand[5]})')

                elif algorithm == 'greedy':

                    expanded.append(f'({node\_to\_expand[0]})')

        return {'Expanded':expanded, 'Path':self.\_\_path(visited, self.\_\_goal), 'Distance': node\_to\_expand[3]}

# **Appendix: Input File Format**

**Input File Format for BFS and DFS (Graph type: graph)**

---------------------------------------------------------------------------------------------------------------------------------------

node1 neighbor11 neighbor12 neighbor13 …

node2 neighbor21 neighbor22 neighbor23 …

node3 neighbor31 neighbor32 neighbor33 …

…

---------------------------------------------------------------------------------------------------------------------------------------

**Input File Format for Greedy and A\* (Graph type: weighted\_graph)**

---------------------------------------------------------------------------------------------------------------------------------------

goal

node1 h1 neighbor11 cost11 neighbor12 cost12 neighbor13 cost13 …

node2 h2 neighbor21 cost21 neighbor22 cost22 neighbor23 cost23 …

node3 h3 neighbor31 cost31 neighbor32 cost32 neighbor33 cost33 …

…

---------------------------------------------------------------------------------------------------------------------------------------