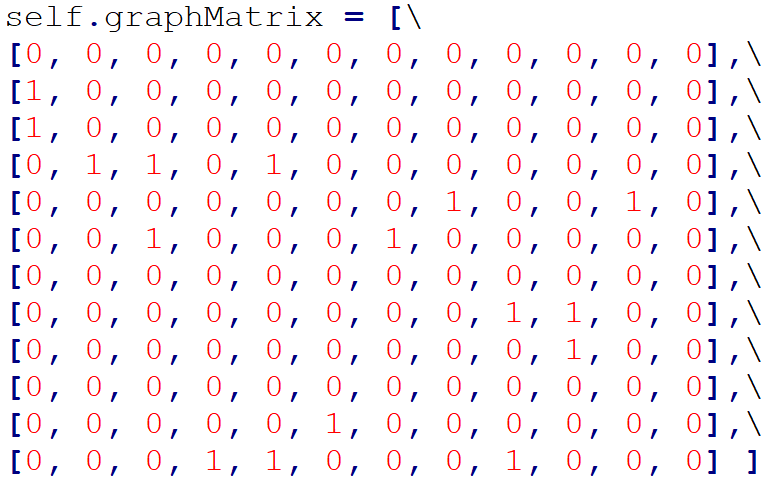
# Objective and functionality

Let be a graph, where is the set of all vertices and represent the set of all edges. This program finds , which includes all the vertices that create a path from to . The inputs of the program are, , and . The program uses Depth First Search (DFS), Breadth First Search (BFS), and Uniform Cost Search (UCS) to find a path. The output of the program is two strings. The former lists all the node expanded during the search procedure while the latter only shows the path.

# Inputs and Outputs

**Input 1.** The graph: The input graph can be represented using either an adjacency matrix or a vertex list. The former is a matrix, where the determines the weight of the edge that connects vertex to vertex, where and. A weight of 0 implies that the vertices are disconnected. An adjacency matrix that includes only 1’s and 0’s represents an unweighted graph. A matrix that is symmetric along the main diagonal implies an undirected graph. Alternatively, a graph can be represented using a vector list. There is an entry for each in the vector list. Entry lists all the vertices that is connected to, followed by the corresponding weights of each edge. For example “7:[8,9],[4,4]” implies that is connected to and over the edges with weights of 4 and 4, respectively. A node with no connection in a directed graph can be modeled as “6:[],[]”. Note that the vertex list is implemented using strings to improve program’s flexibility (rather than a dictionary). Also, the implemented vertex list uses numbers instead of letters to facilitate adding or removing new vertices to the matrix. Figures blew show an example of an adjacency matrix and a vector list:

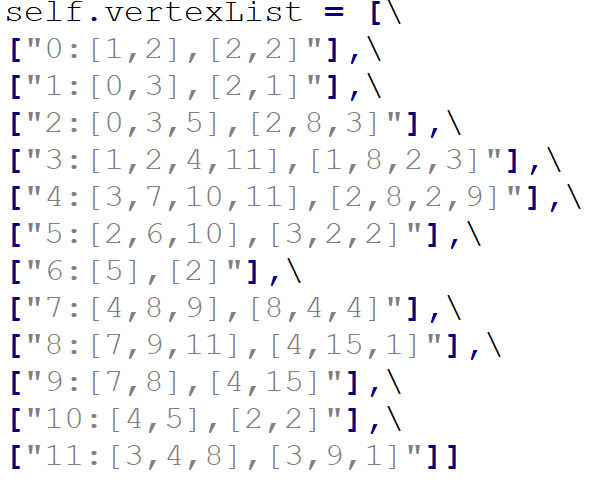


Figure 2. An example of an undirected weighted graph

Figure 1. An adjacency matrix example for an unweighted directed graph

**Input 2.** The start and goal states: and are represented by two global variable named startState and goalSate that can be configured in the code. By default, the startState is set to 11 (state S) and goalSate is set to 6 (state G).

**Output1.** States expanded: is a string that shows all the nodes that are expanded during the search process, separated by commas. The vertices of the graph are encoded by letters according to the graph provided in the assignment. The figure below shows an example of the output.

**Output2.** The returned path: is a string that shows all the nodes that are in the path separated by an arrow to imply the direction.

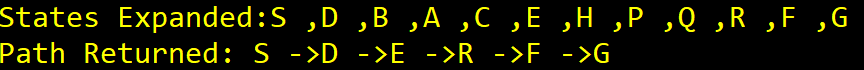
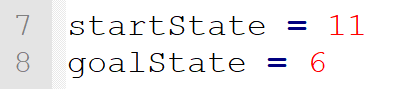


Figure 3. In this example, for the algorithm to find a path from S to G, it has to expand 12 vertices. The actual path, however, only includes 6.

# Utility and Application

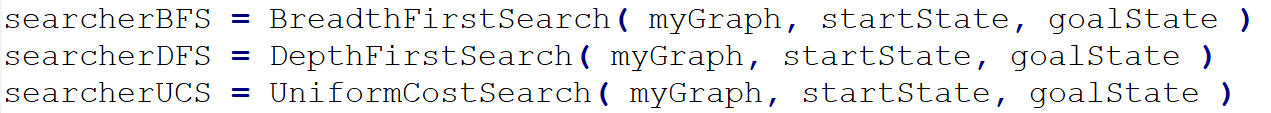
The code is written by Python 3 and uses libraries ‘collections’, ‘copy’, ‘parse’, and ‘heapq’. To use the code, first set the desired start and goal state.



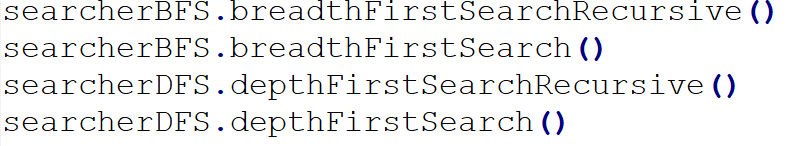
Then an object of GraphBasics must be created. The constructor receives three arguments. For graphs represented by adjacency matrix use “matrix” (all lowercase) as the first argument. Use anything else if it is a vector list (“vectorList” is recommended). If the graph is directed, use “directed” as the second input. Use anything else for undirected graphs (“undirected” is recommended). Finally, for weighted graph use “weighted” and anything else otherwise (“unweighted” is recommended). For example, the statement below creates an undirected unweighted graph that is implemented by an adjacency matrix.



Once a graph is created, instantiate an object of the desired search algorithm. Note that the constructor for all search classes receives three arguments. The first one is the graph to be searched. The second one is the start state and the third one is the goal sate.



Finally, from the search object created, call the appropriate function depending on whether the graph is represented by adjacency matrix or vertex list and whether recursive or iterative executions is desired. Below are a few examples.



To run the \_VertexList function, make sure the graph is created by “vertexList” option.

# Architecture and Structure

The program functionality is divided among five classes: GraphBasics, Search, DepthFirstSearch, BreadthFirstSearch, and UniformCostSearch.

**GraphBasics** stores the input graph represented using either adjacency matrix or vector list. It provides basic graph functionalities such as returning the weight of the edge that connects two vertices, altering the existing paths between nodes, checking if two nodes are connected, and other miscellaneous functionality. getNextLeastCostlyState is the most important member function of this class. Given a current node, a list of already expanded nodes, and the preferred graph representation (whether vector list or adjacency matrix), this function returns an edge of this node, which has the lowest weight and connects it to a node that is not expanded yet. Whenever there is tie, the function returns the vertex with the smallest index. Observe that there is always a tie in an unweighted graph. If no edge is detected (maybe because there is not any in a directed graph or maybe because all the connected nodes are already expanded), the function returns a negative value. GraphBasics is entirely responsible for handling the difference between the adjacency matrix and vertex list.

**Search** is the parent class of all search algorithms. It is an abstract entity, meaning that no object is created from this class. Instead, this class provides all the functionality that is common among any search algorithm. For example, it checks the success condition, the termination condition, etc. It also keeps track of the expanded nodes, current vertex the program is already expanding, the start and goal nodes, and the graph itself. It does not include the data structure to hold the fringe though, as the fringe is dependent on the actual algorithm. The most important function of this class is goToTheNextState, which is a wrapper function for GraphBasics. getNextLeastCostlyState. This enables the search class to fetch the next suitable node it has to expand.

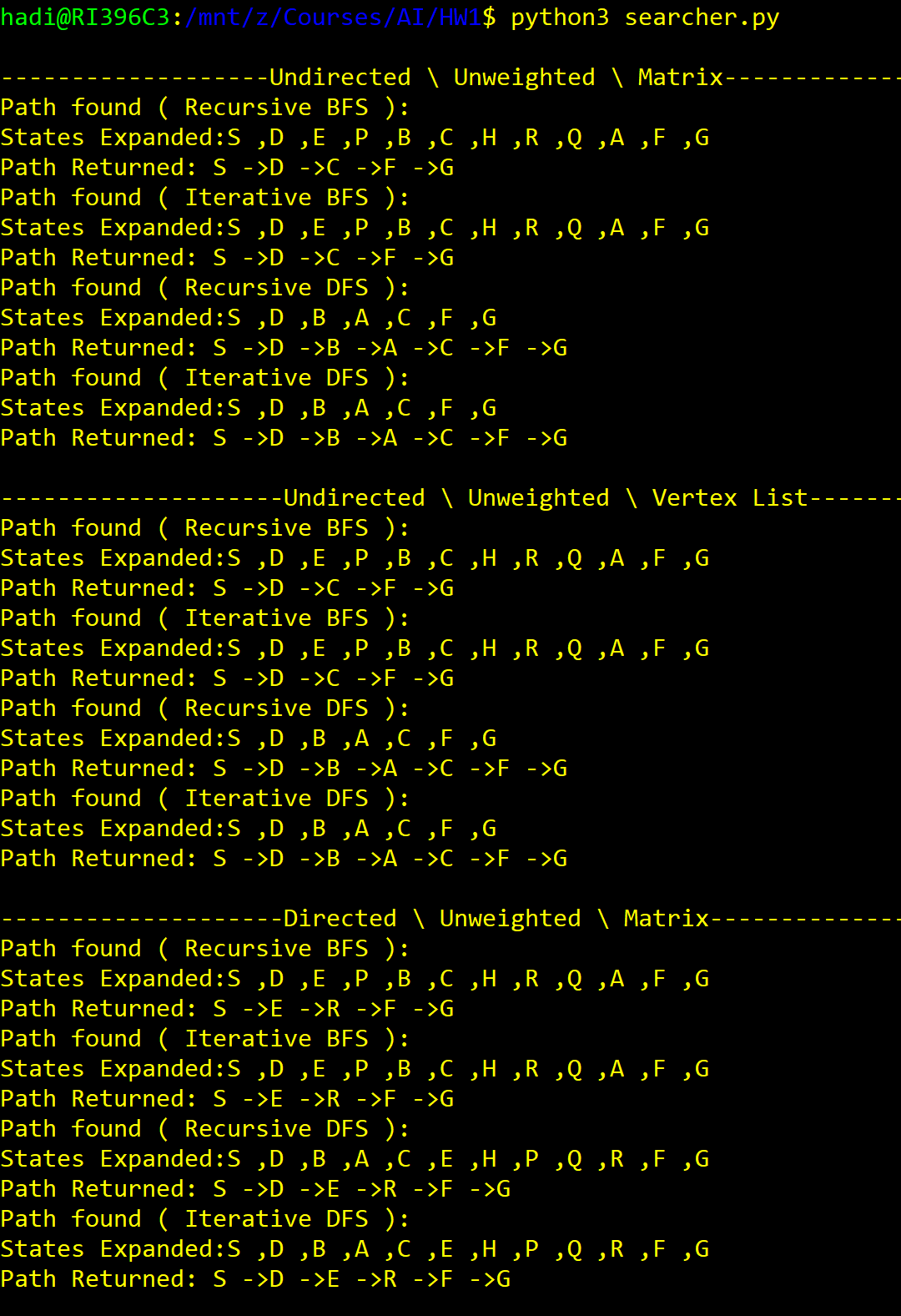
**DeapthFirstSearch** inherits all data members and member functions of its parent class. It also includes a stack (named fringeStack) that is used to hold the fringe. The class includes four functions named depthFirstSearch (which performs DFS search on a matrix that is implemented by adjacency matrix. The function is run iteratively rather than recursively) depthFirstSearchRecursive (which performs DFS search reclusively on graph that is represented by an adjacency matrix), depthFirstSearchRecursive\_VertexList (which performs DFS reclusively on a graph that is represented by and a vertex list), depthFirstSearch\_VertexList (which represents iterative DFS on a graph that is represented by an adjacency matrix). The search begins by inserting the current state in the stack. Search class then calls the GraphBasics to find the next node, which is the first one in alphabetical order in an unweighted graph. If the returned node is not the goal state, and is not a dead-end or already expanded node, the current state is set to the returned vertex. The process then continues in a loop or through recursion depending on the actual implementation of the function. Thanks to the modularity of the code, the implementation of the vertex list is similar to the adjacency matrix as the GraphBasics class handles the difference. This class handles any dead-end (by a function called resolveDeadend) by popping the current node from the stack, which returns the function to the previous node. If there is no unexplored edge in the start state is left, the search terminates.

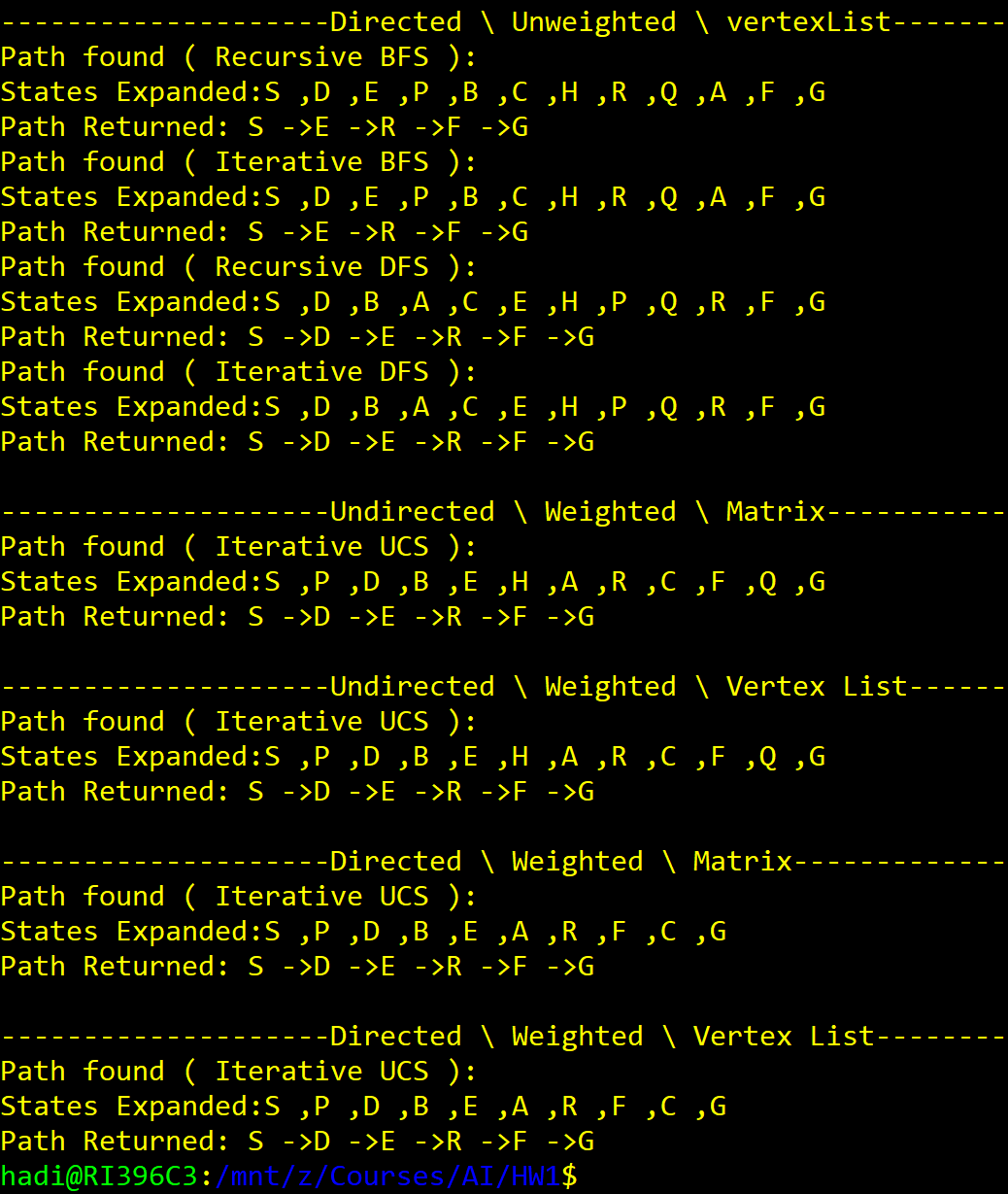
**BreadthFirstSearch** inherits all data members and member functions of the parent class. Similar to the previous class, it includes four different functions for recursive, iterative, adjacency matrix, and vector lists. The class uses a queue to keep track of the expanded nodes. The next state that needs to be expanded is computed similarly by calling the Search class and GraphBasics. This time, however, the fringe is pushed and popped from a queue.

**UniformCostSearch** is also inherited from Search. It has four functions that are used to solve the problem recursively, iteratively, using adjacency matrix, or vector list. The implementation of this algorithm is similar to the last two classes, however, the fringe is saved in a priority queue that is implemented by the heap. To avoid using two different queues, the fringe and its cost are saved in one queue. The first four letters of the fringe indicate the cost. A period then separates the cost from the actual fringe. Each vertex in the fringe is padded with zeros to make it four characters and vertices in the fringe are separated by commas. Padding elements with zeros is necessary as it allows us to safely compare strings. If the number of vertices in a graph or the total possible cost is in order of tens of thousands the padding width must be increased. This can easily be done, as the padding width is determined by a data member of this class. In case of a dead-end, UCS pops the least expensive fringe (which is always at location zero of the heap) and resumes the search from the one after that. This way, the least expensive path is guaranteed to be found (if it exists).

# Output of the program.

Pictures below show the output of the program, run on the graph provided in the assignment.





Sincerely,

Hadi Habibzadeh