Khyati Mahajan

Graph Algorithms

Shortest Path and Minimum Spanning Tree

Aim

To study graph methods for shortest path detection and constructing a minimum spanning tree for graphs with different properties, listed below:

1. Directed or Undirected
2. Cyclic or acyclic

Shortest Path

This project aims to study single source shortest path given that there are no negative edges in the graph. Single source shortest path refers to the path in the graph which maintains the least cost to reach any vertex in the graph from the source. This project utilizes Djikstra's algorithm for calculating the shortest path tree.

Minimum Spanning Tree

A spanning tree of a graph is defined as a subgraph containing all the vertices of the graph and the minimum number of edges required to keep the graph connected. In a weighted graph, a minimum spanning tree is the spanning tree which has the minimum weight for the edges included than all other spanning trees of the graph. This project uses Prim's algorithm for calculating the minimum spanning tree.

Project Code Details

Assumptions

The code assumes that the input format provided is followed strictly, such that:

1. The number of vertices (v) and edges (e) specified are followed by exactly e lines of edge specifiers and use exactly v vertices.
2. When the source node is not specified, the line which specified the last edge is the last line of the input text file.

Program Main Code and Helper Functions

Method	What It Does
read_input	Converts the text file into chunks of information that are used to get information about the graph Input: Textfile as specified Returns: init_graph (a raw format of the graph as specified by strings)
gather_info	Gathers information about a given graph graph Input: init_graph obtained after calling read_input Returns: number of vertices and edges, graph type, source vertex

build_graph	Builds an adjacency list from the given graph Input: graph obtained after calling gather_info Returns: adjacency_list, source vertex

The rest of the script calls these functions and prints out information about the graph and the solutions obtained for the shortest path and the minimum spanning tree. If a source vertex is not provided as input, the script chooses one and runs the shortest_path_djikstra and minimum_spanning_tree_prim using this vertex.

Djikstra's Shortest Path Code

The shortest_path_djikstra code calculates the shortest path by using Python's heap library to maintain a priority queue prioritized by the distance of the vertex from the given or chosen source. The runtime listed on the Python documentation for heapsort is O(nlogn) where n refers to the number of elements in the heap. Heapsort is used when an element is pushed into the heap or popped from the heap.

For each iteration, the algorithms pops the closest vertex from the heap taking O(VlogV) time. For all iterations, since there are at most E edges, we perform O(ElogV) push updates to the heap. Thus the overall runtime turns out to be O((V+E)logV), shortened to O(ElogV) since edges are generally more than vertices for a connected graph.

The project code stores priority queue based on the edges, and since the heapq priority queue implementation in Python does not allow updating priority in heap, the vertices could also be repeated in the queue based on the edges. Thus the runtime of this project's code turns out to be O((E+E)logE), which is O(ElogE). The runtime could be improved to O(ElogV) by allowing to update the priority queue, as noted on the documentation page[1].

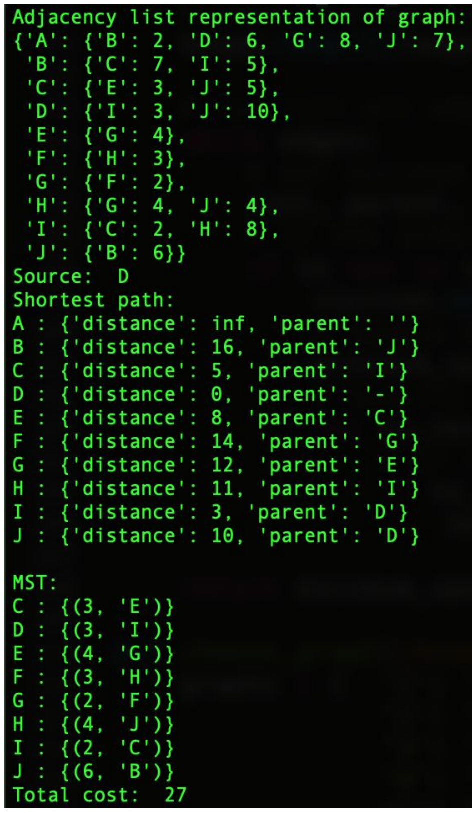
Prim's Minimum Spanning Tree Code

The minimum_spanning_tree_prim calculates the minimum spanning tree using Python's heap library to maintain a priority queue prioritized by the edge weight of the vertex from the vertices already visited.

The edges from the source vertex are first heapified. The heap prioritizes the outgoing edges. Then the edges are traversed, and if the end vertex obtained from heap pop has not been visited yet, it is added to the minimum spanning tree. The neighbors of this vertex are then checked to be added to the heap unless they have already been visited. Thus the runtime is O(ElogE).

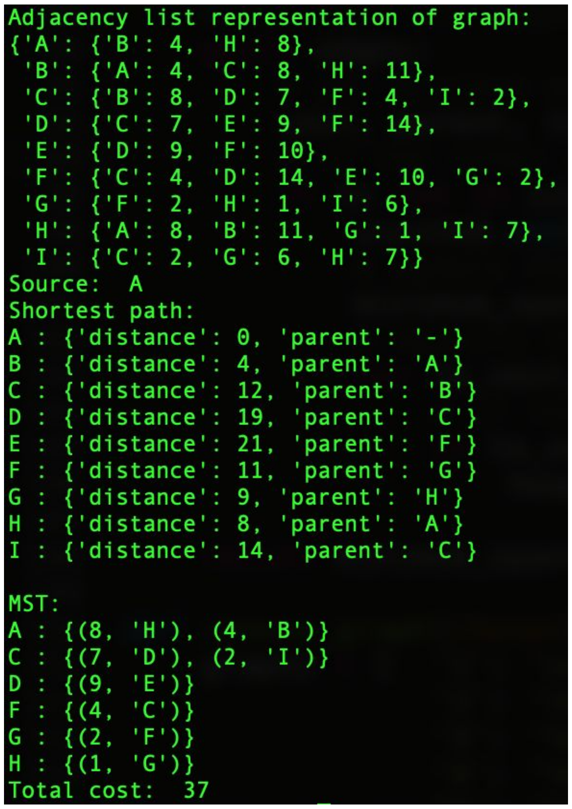
1: https://docs.python.org/3/library/heapq.html

Sample outputs



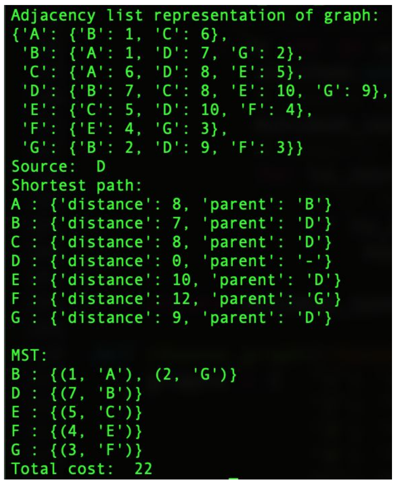
```
Adjacency list representation of graph:
{'A': {'B': 2, 'D': 6, 'G': 8, 'J': 7},
 'B': {'C': 7, 'I': 5},
 'C': {'E': 3, 'J': 5},
 'D': {'I': 3, 'J': 10},
 'E': {'G': 4},
 'F': {'H': 3},
 'G': {'F': 2},
 'H': {'G': 4, 'J': 4},
 'I': {'C': 2, 'H': 8},
 'J': {'B': 6}}
Source:  D
Shortest path:
A : {'distance': inf, 'parent': ''}
B : {'distance': 16, 'parent': 'J'}
C : {'distance': 5, 'parent': 'I'}
D : {'distance': 0, 'parent': '-'}
E : {'distance': 8, 'parent': 'C'}
F : {'distance': 14, 'parent': 'G'}
G : {'distance': 12, 'parent': 'E'}
H : {'distance': 11, 'parent': 'I'}
I : {'distance': 3, 'parent': 'D'}
J : {'distance': 10, 'parent': 'D'}

MST:
C : {(3, 'E')}
D : {(3, 'I')}
E : {(4, 'G')}
F : {(3, 'H')}
G : {(2, 'F')}
H : {(4, 'J')}
I : {(2, 'C')}
J : {(6, 'B')}
Total cost:  27
```



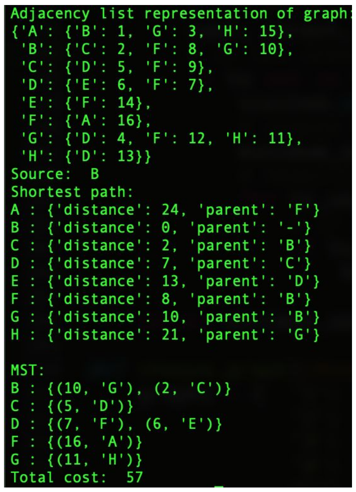
```
Adjacency list representation of graph:
{'A': {'B': 4, 'H': 8},
 'B': {'A': 4, 'C': 8, 'H': 11},
 'C': {'B': 8, 'D': 7, 'F': 4, 'I': 2},
 'D': {'C': 7, 'E': 9, 'F': 14},
 'E': {'D': 9, 'F': 10},
 'F': {'C': 4, 'D': 14, 'E': 10, 'G': 2},
 'G': {'F': 2, 'H': 1, 'I': 6},
 'H': {'A': 8, 'B': 11, 'G': 1, 'I': 7},
 'I': {'C': 2, 'G': 6, 'H': 7}}
Source:  A
Shortest path:
A : {'distance': 0, 'parent': '-'}
B : {'distance': 4, 'parent': 'A'}
C : {'distance': 12, 'parent': 'B'}
D : {'distance': 19, 'parent': 'C'}
E : {'distance': 21, 'parent': 'F'}
F : {'distance': 11, 'parent': 'G'}
G : {'distance': 9, 'parent': 'H'}
H : {'distance': 8, 'parent': 'A'}
I : {'distance': 14, 'parent': 'C'}

MST:
A : {(8, 'H'), (4, 'B')}
C : {(7, 'D'), (2, 'I')}
D : {(9, 'E')}
F : {(4, 'C')}
G : {(2, 'F')}
H : {(1, 'G')}
Total cost:  37
```



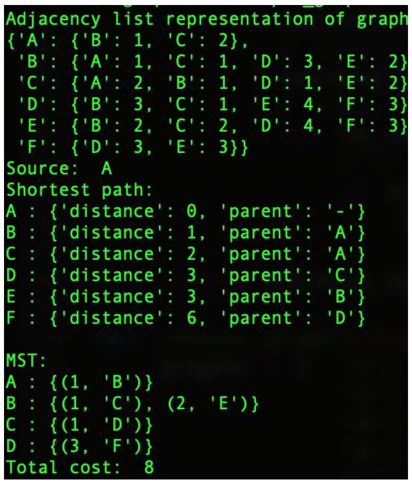
```
Adjacency list representation of graph:
{'A': {'B': 1, 'C': 6},
 'B': {'A': 1, 'D': 7, 'G': 2},
 'C': {'A': 6, 'D': 8, 'E': 5},
 'D': {'B': 7, 'C': 8, 'E': 10, 'G': 9},
 'E': {'C': 5, 'D': 10, 'F': 4},
 'F': {'E': 4, 'G': 3},
 'G': {'B': 2, 'D': 9, 'F': 3}}
Source:  D
Shortest path:
A : {'distance': 8, 'parent': 'B'}
B : {'distance': 7, 'parent': 'D'}
C : {'distance': 8, 'parent': 'D'}
D : {'distance': 0, 'parent': '-'}
E : {'distance': 10, 'parent': 'D'}
F : {'distance': 12, 'parent': 'G'}
G : {'distance': 9, 'parent': 'D'}

MST:
B : {(1, 'A'), (2, 'G')}
D : {(7, 'B')}
E : {(5, 'C')}
F : {(4, 'E')}
G : {(3, 'F')}
Total cost:  22
```



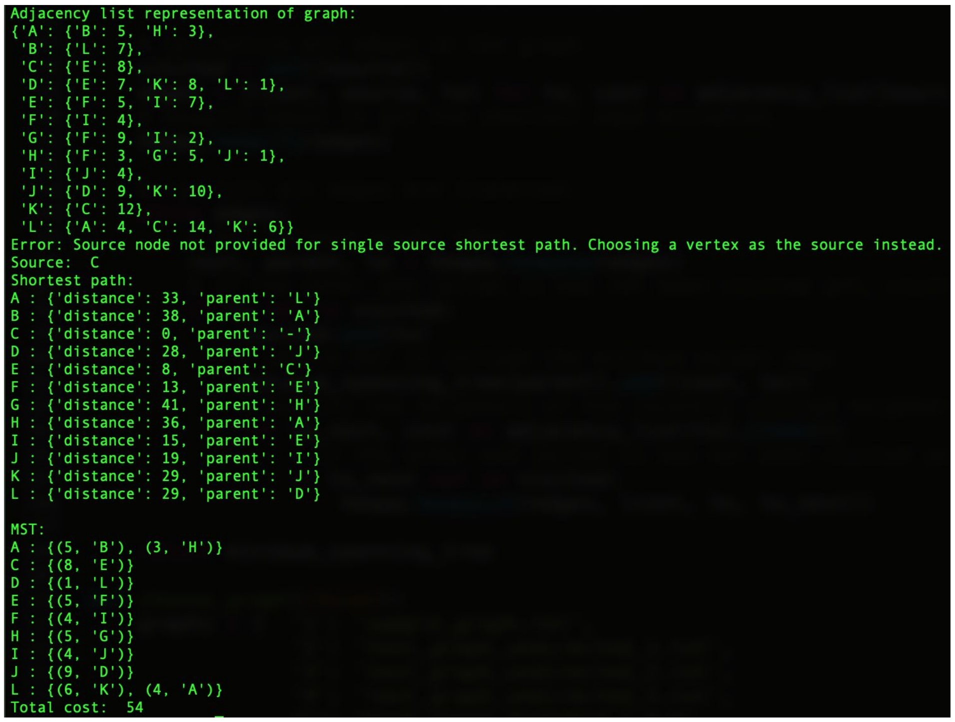
```
Adjacency list representation of graph
{'A': {'B': 1, 'G': 3, 'H': 15},
 'B': {'C': 2, 'F': 8, 'G': 10},
 'C': {'D': 5, 'F': 9},
 'D': {'E': 6, 'F': 7},
 'E': {'F': 14},
 'F': {'A': 16},
 'G': {'D': 4, 'F': 12, 'H': 11},
 'H': {'D': 13}}
Source:  B
Shortest path:
A : {'distance': 24, 'parent': 'F'}
B : {'distance': 0, 'parent': '-'}
C : {'distance': 2, 'parent': 'B'}
D : {'distance': 7, 'parent': 'C'}
E : {'distance': 13, 'parent': 'D'}
F : {'distance': 8, 'parent': 'B'}
G : {'distance': 10, 'parent': 'B'}
H : {'distance': 21, 'parent': 'G'}

MST:
B : {(10, 'G'), (2, 'C')}
C : {(5, 'D')}
D : {(7, 'F'), (6, 'E')}
F : {(16, 'A')}
G : {(11, 'H')}
Total cost:  57
```



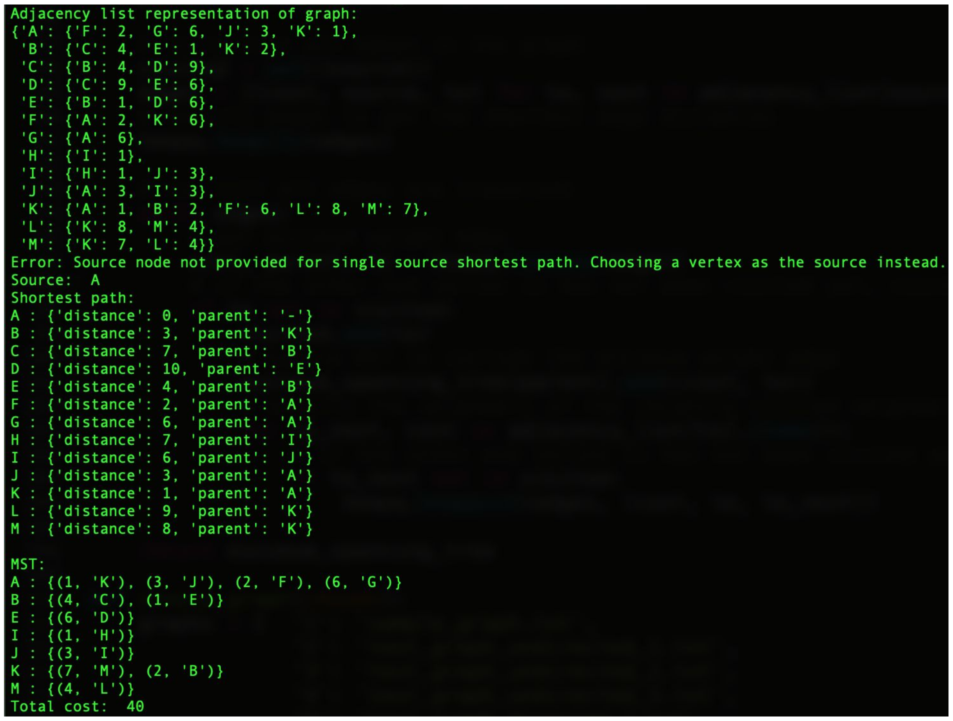
```
Adjacency list representation of graph
{'A': {'B': 1, 'C': 2},
 'B': {'A': 1, 'C': 1, 'D': 3, 'E': 2}
 'C': {'A': 2, 'B': 1, 'D': 1, 'E': 2}
 'D': {'B': 3, 'C': 1, 'E': 4, 'F': 3}
 'E': {'B': 2, 'C': 2, 'D': 4, 'F': 3}
 'F': {'D': 3, 'E': 3}}
Source:  A
Shortest path:
A : {'distance': 0, 'parent': '-'}
B : {'distance': 1, 'parent': 'A'}
C : {'distance': 2, 'parent': 'A'}
D : {'distance': 3, 'parent': 'C'}
E : {'distance': 3, 'parent': 'B'}
F : {'distance': 6, 'parent': 'D'}

MST:
A : {(1, 'B')}
B : {(1, 'C'), (2, 'E')}
C : {(1, 'D')}
D : {(3, 'F')}
Total cost:  8
```



```
Adjacency list representation of graph:
{'A': {'B': 5, 'H': 3},
 'B': {'L': 7},
 'C': {'E': 8},
 'D': {'E': 7, 'K': 8, 'L': 1},
 'E': {'F': 5, 'I': 7},
 'F': {'I': 4},
 'G': {'F': 9, 'I': 2},
 'H': {'F': 3, 'G': 5, 'J': 1},
 'I': {'J': 4},
 'J': {'D': 9, 'K': 10},
 'K': {'C': 12},
 'L': {'A': 4, 'C': 14, 'K': 6}}
Error: Source node not provided for single source shortest path. Choosing a vertex as the source instead.
Source:  C
Shortest path:
A : {'distance': 33, 'parent': 'L'}
B : {'distance': 38, 'parent': 'A'}
C : {'distance': 0, 'parent': '-'}
D : {'distance': 28, 'parent': 'J'}
E : {'distance': 8, 'parent': 'C'}
F : {'distance': 13, 'parent': 'E'}
G : {'distance': 41, 'parent': 'H'}
H : {'distance': 36, 'parent': 'A'}
I : {'distance': 15, 'parent': 'E'}
J : {'distance': 19, 'parent': 'I'}
K : {'distance': 29, 'parent': 'J'}
L : {'distance': 29, 'parent': 'D'}

MST:
A : {(5, 'B'), (3, 'H')}
C : {(8, 'E')}
D : {(1, 'L')}
E : {(5, 'F')}
F : {(4, 'I')}
H : {(5, 'G')}
I : {(4, 'J')}
J : {(9, 'D')}
L : {(6, 'K'), (4, 'A')}
Total cost:  54
```



```
Adjacency list representation of graph:
{'A': {'F': 2, 'G': 6, 'J': 3, 'K': 1},
 'B': {'C': 4, 'E': 1, 'K': 2},
 'C': {'B': 4, 'D': 9},
 'D': {'C': 9, 'E': 6},
 'E': {'B': 1, 'D': 6},
 'F': {'A': 2, 'K': 6},
 'G': {'A': 6},
 'H': {'I': 1},
 'I': {'H': 1, 'J': 3},
 'J': {'A': 3, 'I': 3},
 'K': {'A': 1, 'B': 2, 'F': 6, 'L': 8, 'M': 7},
 'L': {'K': 8, 'M': 4},
 'M': {'K': 7, 'L': 4}}
Error: Source node not provided for single source shortest path. Choosing a vertex as the source instead.
Source:  A
Shortest path:
A : {'distance': 0, 'parent': '-'}
B : {'distance': 3, 'parent': 'K'}
C : {'distance': 7, 'parent': 'B'}
D : {'distance': 10, 'parent': 'E'}
E : {'distance': 4, 'parent': 'B'}
F : {'distance': 2, 'parent': 'A'}
G : {'distance': 6, 'parent': 'A'}
H : {'distance': 7, 'parent': 'I'}
I : {'distance': 6, 'parent': 'J'}
J : {'distance': 3, 'parent': 'A'}
K : {'distance': 1, 'parent': 'A'}
L : {'distance': 9, 'parent': 'K'}
M : {'distance': 8, 'parent': 'K'}

MST:
A : {(1, 'K'), (3, 'J'), (2, 'F'), (6, 'G')}
B : {(4, 'C'), (1, 'E')}
E : {(6, 'D')}
I : {(1, 'H')}
J : {(3, 'I')}
K : {(7, 'M'), (2, 'B')}
M : {(4, 'L')}
Total cost:  40
```