Practical 3

Week 4

Artificial Intelligence Summer 2019

Dijkstra’s Algorithm:

This week the objective is to learn how Dijkstra's Algorithm works. Dijkstra's Algorithm finds the shortest path from the root node to every other node.

Get an understanding of the algorithm from here:

https://www.codingame.com/playgrounds/1608/shortest-paths-with-dijkstras-algorithm/dijkstras-algorithm

Difference between Algorithm and Uniform Cost Search:

Uniform Cost searches for shortest paths in terms of cost from the root node to a goal node. Uniform Cost Search is Dijkstra's Algorithm which is focused on finding a single shortest path to a single finishing point rather than the shortest path to every point.

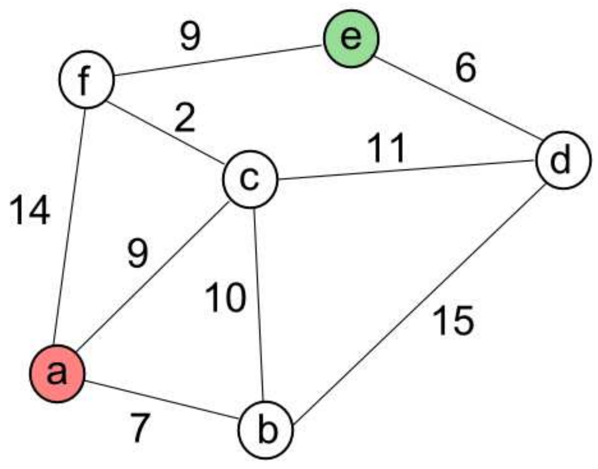
UCS does this by stopping as soon as the finishing point is found. For Dijkstra, there is no goal state and processing continues until all nodes have been removed from the priority queue, i.e. until shortest paths to all nodes (not just a goal node) have been determined.

UCS has fewer space requirements, where the priority queue is filled gradually as opposed to Dijkstra's, which adds all nodes to the queue on start with an infinite cost.

As a result of the above points, Dijkstra is more time consuming than UCS. Also UCS is usually formulated on trees while Dijkstra is used on general graphs

Djikstra is only applicable in explicit graphs where the entire graph is given as input. UCS starts with the source vertex and gradually traverses the necessary parts of the graph. Therefore, it is applicable for both explicit graphs and implicit graphs (where states/nodes are generated as traversal proceeds).

Exercise 1: Recall the exercises we did last week in using various constructors provided in NetworkX to draw undirected and directed graphs. Re-use the code to create an undirected graph as shown below:



Exercise 2: Once you have created the above graph, you will need to use the built-in functions within NetworkX to implement Dijkstra's algorithm. The two functions are:

`shortest_path_length`(*G*, *source*, *target*, *weight*)

Compute shortest path lengths in the graph.

Parameters:

- **G** (*NetworkX graph*) –
- **source** (*node, optional*) – Starting node for path. If not specified, compute shortest path lengths using all nodes as source nodes.
- **target** (*node, optional*) – Ending node for path. If not specified, compute shortest path lengths using all nodes as target nodes.
- **weight** (*None or string, optional (default = None)*) – If None, every edge has weight/distance/cost 1. If a string, use this edge attribute as the edge weight. Any edge attribute not present defaults to 1.

Returns:

length – If the source and target are both specified, return the length of the shortest path from the source to the target.

If only the source is specified, return a dictionary keyed by targets whose values are the lengths of the shortest path from the source to one of the targets.

If only the target is specified, return a dictionary keyed by sources whose values are the lengths of the shortest path from one of the sources to the target.

If neither the source nor target are specified return a dictionary of dictionaries with path[source][target]=L, where L is the length of the shortest path from source to target.

Return type: int or dictionary

`shortest_path`(*G*, *source=None*, *target=None*, *weight=None*)

Compute shortest paths in the graph.

Parameters:

- **G** (*NetworkX graph*) –
- **source** (*node, optional*) – Starting node for path. If not specified, compute shortest paths using all nodes as source nodes.
- **target** (*node, optional*) – Ending node for path. If not specified, compute shortest paths using all nodes as target nodes.

- **weight** (*None or string, optional (default = None)*) – If None, every edge has weight/distance/cost 1. If a string, use this edge attribute as the edge weight. Any edge attribute not present defaults to 1.

Returns:

path – All returned paths include both the source and target in the path.

If the source and target are both specified, return a single list of nodes in a shortest path from the source to the target.

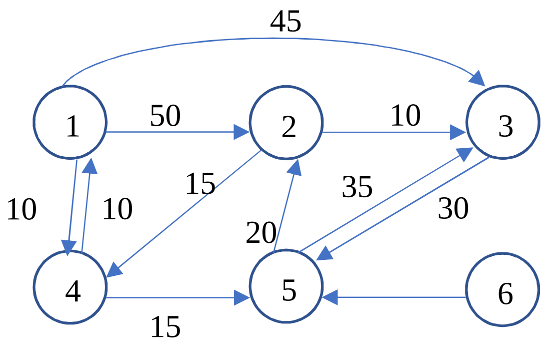
If only the source is specified, return a dictionary keyed by targets with a list of nodes in a shortest path from the source to one of the targets.

If only the target is specified, return a dictionary keyed by sources with a list of nodes in a shortest path from one of the sources to the target.

If neither the source nor target are specified return a dictionary of dictionaries with path[source][target]=[list of nodes in path].

Try to use the above two functions on the graph given in Exercise 1 and print out the shortest path length and the shortest path.

Exercise 3: Modify the above code to make it run on the following graph. Work out the solution manually and verify it with the result obtained from the above code.



Exercise 4: Look at the code given on the Moodle page. Run the code for the graph given above. Work out the shortest path length and the shortest path manually. Does it match with the results returned by the code?

Bonus Exercise 5: Try to understand the code given. Once you have a good understanding of the code, try to print out the priority queue at the end of every iteration.

-END-