Notes - Algorithms & Data Structures - Graphs - Dijkstra - Part 2

- Dr Nick Hayward

A brief intro to Dijkstra's algorithm for *graph* data structure usage.

Contents

- Intro
- Edges with negative weight
    - Dijkstra and negative weights
- Working implementation
    - implement the graph
    - implement the algorithm
    - code breakdown
    - output

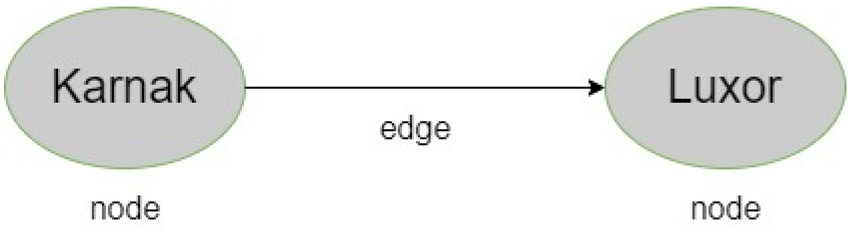
Intro

A graph data structure may be defined in computer science as a way to model a given set of connections.

We may commonly use a *graph* to model patterns and connections for a given problem. For example, such connections may infer relationships within data.

A graph includes *nodes* and *edges*, which help us define such connections.

For example, we have two nodes with a single edge



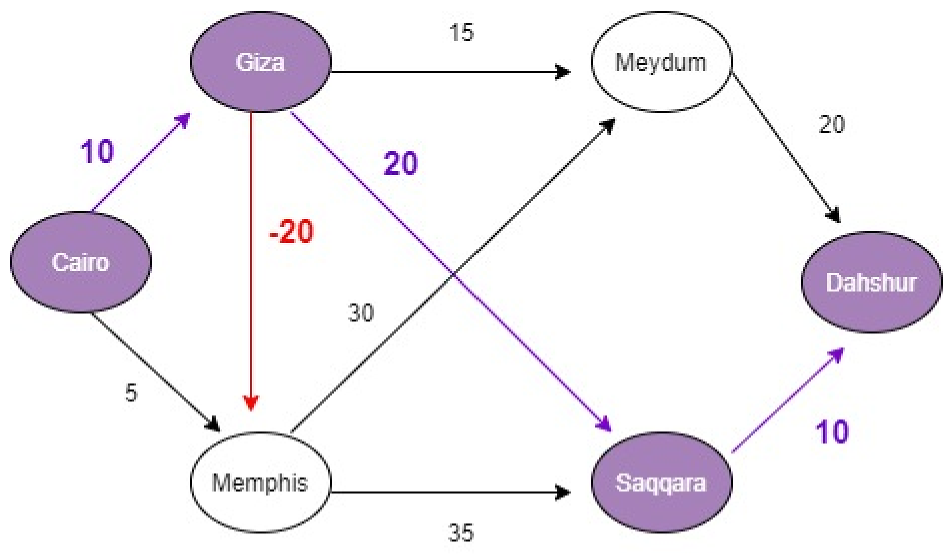
Each node may be connected to many other nodes in the graph, commonly referenced as *neighbour* nodes.

Edges with negative weight

In the last example, outline in part 1, we have weighted edges from Cairo to Giza, and Cairo to Memphis. Each of these routes has a cost involved, the weight of the edge.

However, we may now add a path directly from Giza to Memphis. In our example, this edge will pay us `20`, as we're able to claim the cost back. So, the edge may be defined with a negative weight of `-20`.

So, we now have two routes to consider to allow us to travel from Cairo to Memphis. We might take the previous route, direct from Cairo to Memphis, which will cost `5`, or we might consider the updated route via Giza. The second route, Cairo -> Giza -> Memphis, will now cost `-10`.



Dijkstra and negative weights

If we continue the path through the graph to the end point at Dahshur, we might consider following this route with a negative weighted edge.

However, if we try to perform our usual calculation with Dijkstra's algorithm, we would end up following the more expensive route. In effect, negative-weighted edges will break the use of the Dijkstra algorithm.

The issue may not be the final predicted route, as seen above, it is commonly with the defined calculations performed at various stages during the algorithm's execution.

If we run Dijkstra's algorithm again on this graph, this time with the negative weighted edge, we get a false definition for cheapest route to Memphis.

For example, following the standard pattern of calculation, we get the following table of costs

node	cost
Giza	10
Memphis	5
Saqqara	infinity

Then, we find the lowest-cost node, and update the costs for each of its neigbours. As we can see in the table, Memphis is the initial lowest code node from Cairo with a cost of 5.

So, according to the Dijkstra algorithm, there is no cheaper path to travel from Cairo to Memphis. However, due to the negative-weighted edge from Giza to Memphis, we know this calculation and assertion is *incorrect*.

However, if we continue to follow Dijkstra's algorithm, we update the table as follows

node	cost
Giza	10
Memphis	5

node	cost
Saqqara	40

Then, we get the next lowest cost node from Cairo, Giza with a cost of `10`, and, again, update the cost of its neighbours.

If we consider the neighbours of Giza in the updated graph, we have a negative weighted edge from Giza to Memphis. However, the issue is that we're trying to update the cost for the Memphis node. This is a sign that something is not right with the use of the algorithm.

We've already processed the Memphis node, which means there should not now be a cheaper route to that node. However, due to the negative weighted edge, we have actually found a cheaper route.

If we check the cost up to the node Saqqara, the algorithm will return the already calculated cost of `40`.

However, due to the negative weighted edge, we know there is a cheaper route but Dijkstra's algorithm did not find this route. The algorithm makes an assumption about the processing of the nodes due to initial costs of the weighted edge.

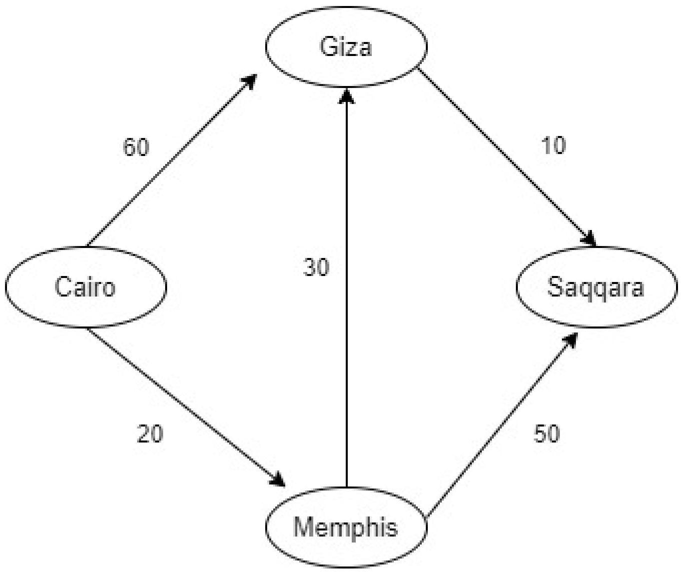
In effect, as we were processing the Memphis node, Dijkstra's algorithm assumes there is now no faster way to that node. However, this assumption only holds true if we do not have negative weighted edges. So, we can't use negative weighted edges with Dijkstra's algorithm.

To allow us to calculate the shortest path in a graph with negative weighted edges we may, instead, use the **Bellman-Ford** algorithm.

Working implementation

We may now consider a basic coded example of implementing Dijkstra's algorithm in Python.

For this example, we'll start with the following graph



To help implement a working example for this graph, we may define three *hash* tables as follows

- graph

parent	node	cost
cairo	giza	60
	memphis	20
giza	saqqara	10
memphis	giza	10
	saqqara	50
saqqara		

- costs

node	current cost
giza	60
memphis	20
saqqara	infinity

- parents

node	current parent
giza	cairo
memphis	cairo
saqqara	-

As we execute the algorithm, we'll update the values for the *costs* and *parents* tables.

implement the graph

We'll need to implement the graph for this coded example. So, we'll use a hash table for the graph

```
graph = {}
```

In this hash table, we need to store multiple values for the neighbours and then set cost for travel along that edge. For example, for the current graph, we can see that Cairo has two neighbours, *Giza* and *Memphis*.

There are a number of options we might consider for structuring this pattern of data, including nested hash tables for each node relative to the parent.

For example,

```
graph["cairo"] = {}
graph["cairo"]["giza"] = 60
```

```
graph["cairo"]["memphis"] = 20
```

This will create the following structure for our data,

```
{'cairo': {'giza': 60, 'memphis': 20}}
```

which corresponds to the structure and values defined in the above table for the graph. We might, of course, check its values as follows,

```
print(graph["cairo"].keys())
```

Likewise, if we need to find the weights for the edges from `Cairo` we may call the following

```
print(graph["cairo"]["giza"])
print(graph["cairo"]["memphis"])
```

So, following this pattern, we may then add the remaining nodes and neighbours to the hash table for the graph.

```
# update other graph nodes and weights
graph["giza"] = {}
graph["giza"]["saqqara"] = 10
graph["memphis"] = {}
graph["memphis"]["giza"] = 30
graph["memphis"]["saqqara"] = 50
# no current neighbour nodes for saqqara - graph end point
graph["saqqara"] = {}
```

The hash table now represents the graph with defined neighbour nodes and weighted edges. For example,

```
{'cairo': {'giza': 60, 'memphis': 20}, 'giza': {'saqqara': 10}, 'memphis':
{'giza': 30, 'saqqara': 50}, 'saqqara': {}}
```

The next structure we need to create is a hash table for the costs of each node. In effect, we're using *cost* to define the value of the weighted edge from one node to another. The *cost* of the node will represent the calculated total for the weighted edges from the start, Cairo, to a given node. For example, we know it will cost `60` to get from Cairo to Giza, and `20` to get from Cairo to Memphis.

As we saw in the above worked examples, we can represent currently unknown costs as *infinity*. So, we may represent this hash table as follows,

```
# cost table - weighted edges from start node
infinity = float("inf")
cost = {}
cost["giza"] = 60
cost["memphis"] = 20
cost["saqqara"] = infinity
```

This may be represented as follows

```
{'giza': 60, 'memphis': 20, 'saqqara': inf}
```

Then, we may add our third table for the parent nodes in the graph

```
# parents table - parent nodes in graph
parents = {}
# define inital parents
parents["giza"] = "cairo"
parents["memphis"] = "cairo"
# parent for end point - updated during execution...
parents["saqqara"] = None
```

We'll update such values as we work through the algorithm and its execution.

We also need to maintain a record of the nodes we've already processed in the graph to avoid duplicated effort.

```
nodes_checked = []
```

implement the algorithm

For this coded example, we need to implement the following pattern for the algorithm

- while nodes exist to continue processing
    - get the node closest to the start node
    - update any costs for the node's neighbours
    - if any costs for the neighbours have been updated
        - update the parents
        - mark node as processed
    - repeat this process as necessary...

So, we may implement this pattern in Python to add Dijkstra's algorithm to an app.

Initially, we'll define the `while` loop, and the checks we need to perform for each iteration of the loop

```python
# execute check for lowest cost node that has not been processed...
node = find_low_cost_node(cost)
# loop through nodes to check - exit when all nodes are processed
while node is not None:
    node_cost = cost[node]
    # add neighbour nodes to hash table
    neighbours = graph[node]
    # loop through all neighbours of current node
    for neighbour in neighbours.keys():
        # update cost where available
        new_node_cost = node_cost + neighbours[neighbour]
        # check updated cost to see if it's now cheaper
        if cost[neighbour] > new_node_cost:
            # update cost for this node
            cost[neighbour] = new_node_cost
            # current node becomes new parent for this neighbour
            parents[neighbour] = node
    # mark node as now processed...
    nodes_checked.append(node)
    # find next node to process - then loop through again...
    node = find_low_cost_node(cost)
```

We start by checking the passed `cost` table for the lowest cost node in the defined graph. The custom function `find_low_cost_node()` may be implemented as follows,

```python
def find_low_cost_node(cost):
    low_cost = float("inf")
    low_cost_node = None
    # check each node
    for node in cost:
        # get current cost
        node_cost = cost[node]
        # check if current cost is lowest & hasn't been processed...
        if node_cost < low_cost and node not in nodes_checked:
            # update as current lowest cost node...
            low_cost = node_cost
            low_cost_node = node
    return low_cost_node
```

This is a simple implementation to check for the current lowest common node.

So, we can use this custom function to get the lowest common node, which we may then use with the `while` loop.

The loop itself may be considered as follows, which helps us further understand how the implemented algorithm will work with a sample graph.

code breakdown

For example, we begin by checking for the node with the lowest cost from the start point in the graph. i.e. from Cairo.

```
# execute check for lowest cost node that has not been processed...
node = find_low_cost_node(cost)
```

In our hash table, this check will return *Memphis* with a cost of `20`. So, we can now get the cost for this node, and its neighbour nodes as well. We may then add these neighbour nodes to their own hash table,

```
neighbours = graph[node]
```

We can use this structure to loop through the stored neighbours,

```
# loop through all neighbours of current node
for neighbour in neighbours.keys():
```

Each of these neighbour nodes will have their own cost, which will detail the cost from the start node, *Cairo*, to that node. In effect, we're calculating the cost of that node from the start node if we went through the current node.

e.g Cairo -> Memphis -> Giza with an updated cost of `50`

This updated cost is, of course, lower than the current cost for a route from the start *Cairo* to Giza, which was previously `60`.

So, we can update the cost as follows

```
# update cost where available
new_node_cost = node_cost + neighbours[neighbour]
```

In effect, this is calculated as the cost of *Memphis*, `20`, plus the cost from *Memphis* to *Giza*, `30`. We now have an updated lowest cost of `50` for a path from *Cairo* to *Giza*.

This new cost is now updated in the `cost` hash table as well,

```
# update cost for this node
cost[neighbour] = new_node_cost
```

We may also update the parent node for *Giza* in the `parents` hash table,

```
# current node becomes new parent for this neighbour
parents[neighbour] = node
```

We're now back at the start of the `while` loop, so we may now move on to the next neighbour, which will be *Saqqara* for the current graph.

We repeat the above pattern, again checking and updating the hash tables for the cost of the path to the current node *Saqqara*, the finish node in the current graph.

output

If we execute this algorithm with the current graph we get the following initial output

```
initial costs
{'giza': 60, 'memphis': 20, 'saqqara': inf}
```

which is then updated as follows after we run Dijkstra's algorithm

```
updated lowest cost from start to each node:
{'giza': 50, 'memphis': 20, 'saqqara': 60}
```

Once we've processed each node in the graph, the algorithm is complete and we have an output for the lowest cost from the start node *Cairo* to the finish node *Saqqara*.