Notes - Algorithms & Data Structures - Graphs - Dijkstra - Part 1

- Dr Nick Hayward

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

Contents

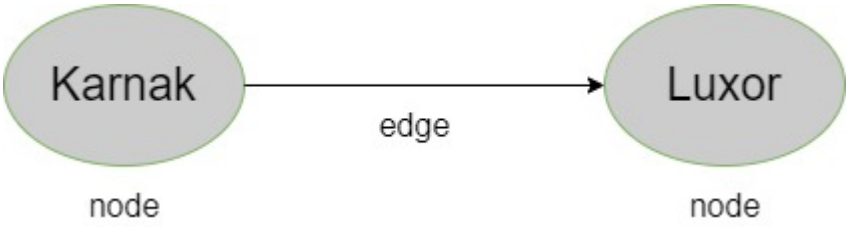
Intro

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



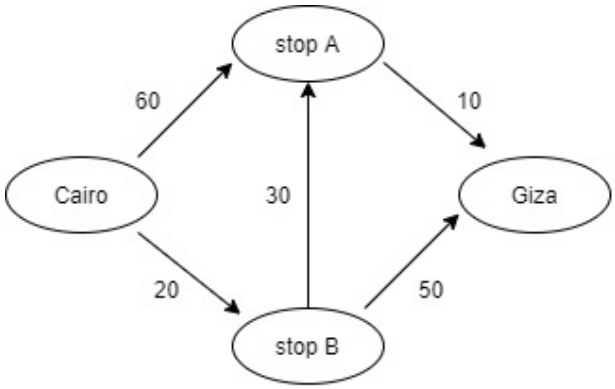
Each node may be connected to many other nodes in the graph, commonly referenced as *neighbour* nodes.

A consideration of terminology

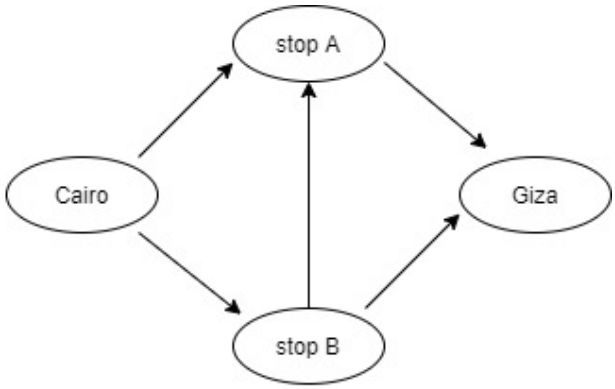
A key concept for working with Dijkstra's algorithm is the association of values, numbers for example, for each edge in the graph. These are the weights assigned to the edge in the graph.

So, when we assign weights to an edge, we're now creating a *weighted* graph. Likewise, if we do not assign weights to edges, we're defining an *unweighted* graph.

For example,



Weighted



Unweighted

The choice of weighted versus unweighted will also affect our choice of algorithm, and the context of its usage.

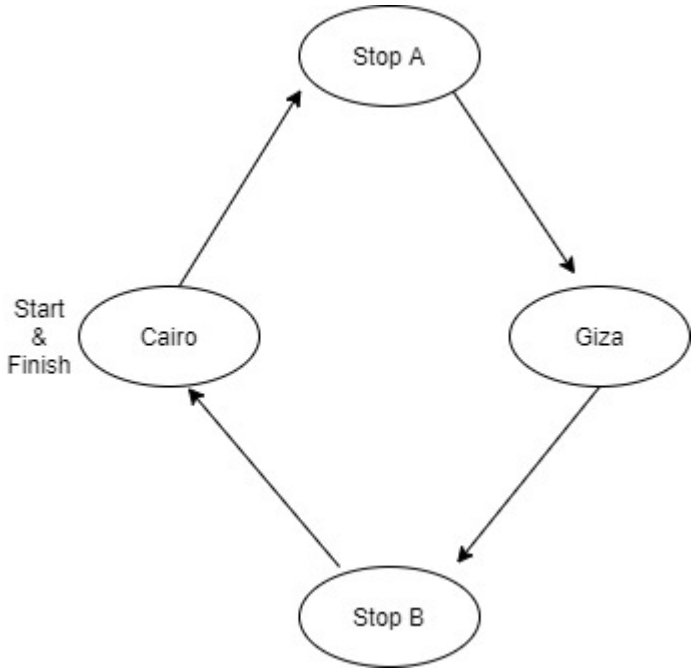
For example, if we want to calculate the shortest path in an *unweighted* graph, we may use the *breadth-first* search algorithm.

However, to perform a similar calculation for a *weighted* graph, we may use Dijkstra's algorithm.

graph cycles

We may also encounter a graph with *cycles*. In effect, we can cycle from `Stop A` back around to `Stop A`.

For example,



So, we may start and finish at the same node in the graph.

If we consider a graph with a cycle segment, we may need to calculate the shortest path between two defined nodes. For most calculations, we will commonly choose a path that avoids the cycle. The cycle will usually add greater weight to the calculation.

If we then follow the cycle more than once, we are simply adding extra weight to the calculation for each completed cycle.

If we consider an *undirected graph*, we are now working with a cycle. Connected nodes in an undirected graph point to each other, effectively a cycle.

So, each edge will add another cycle to an undirected graph.

Dijkstra's algorithm only works with *directed acyclic graphs* (DAGs).

Dijkstra's algorithm

A common requirement for working with graphs is a consideration of weighted and unweighted edges.

With weighted graphs, we're interested in options for assigning more or less weight to edges within the graph.

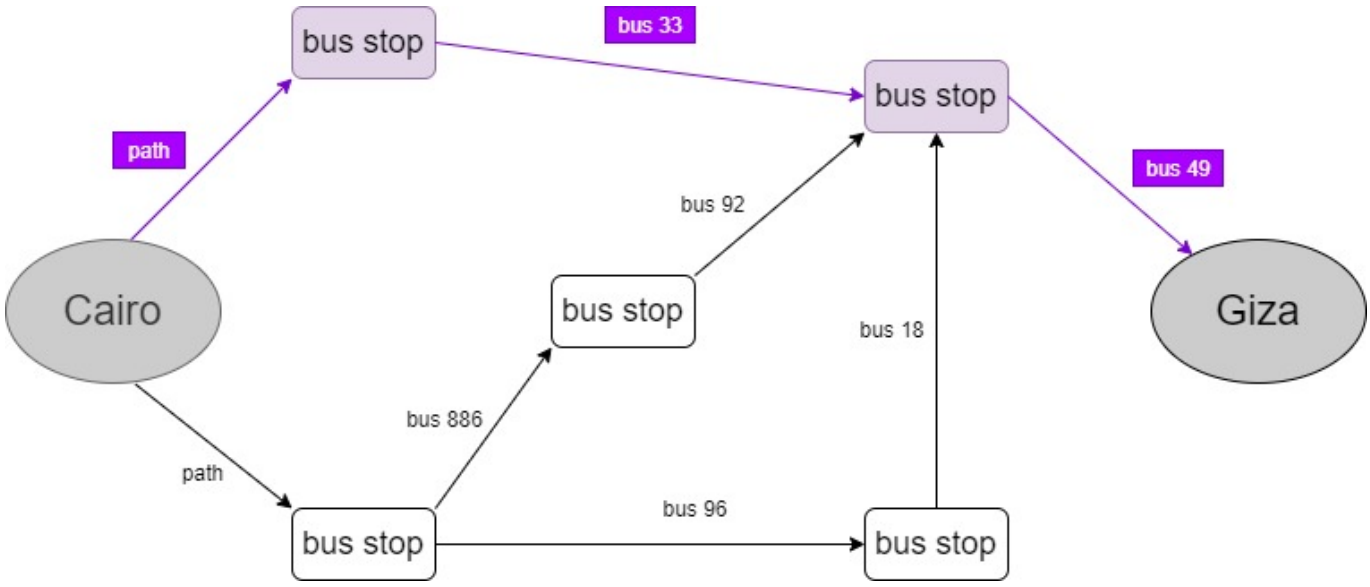
Dijkstra's algorithm helps us work with queries for paths in our graphs. For example, we might need to answer the question,

> which path is the shortest to a node?

e.g. which is the shortest path to node A?

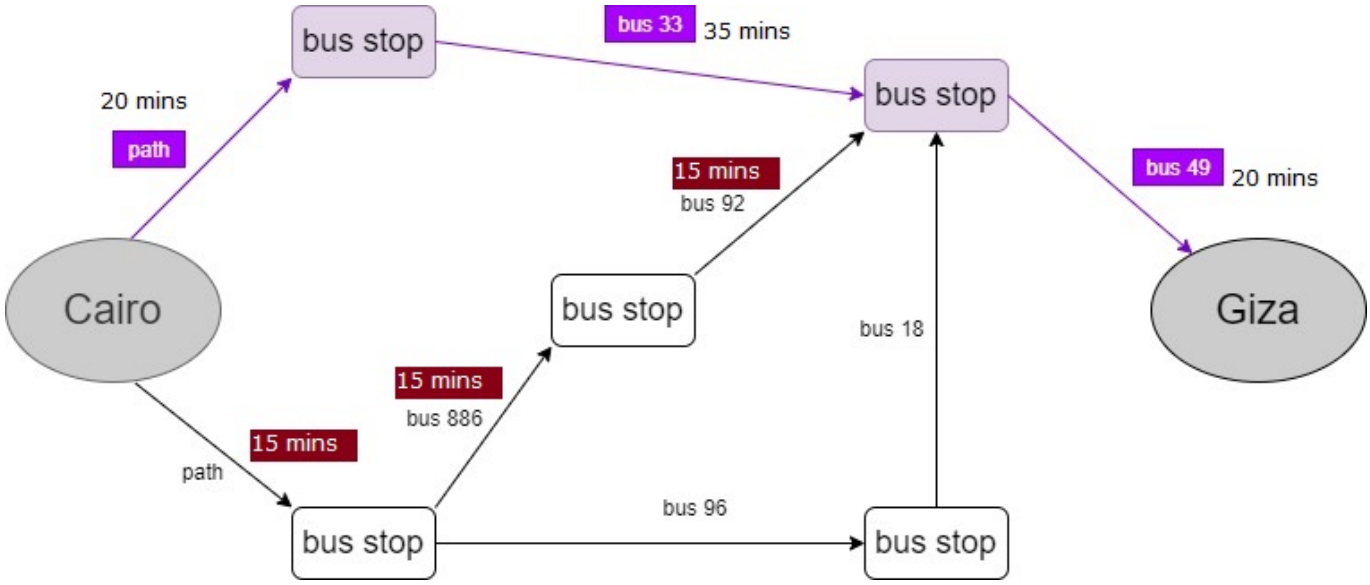
However, this path may not be fastest, but it will be the shortest in the graph. It will be the shortest because it will include the least number of edges between nodes.

For example, we might consider the following graph for the shortest route



We can clearly see the shortest path with the least number of segments, three. We may use *breadth-first* search to find the path with the fewest segments, as we see in this example.

However, if we then added travel times to edges for competing routes, costs for each edge, we may find a quicker path for traveling.

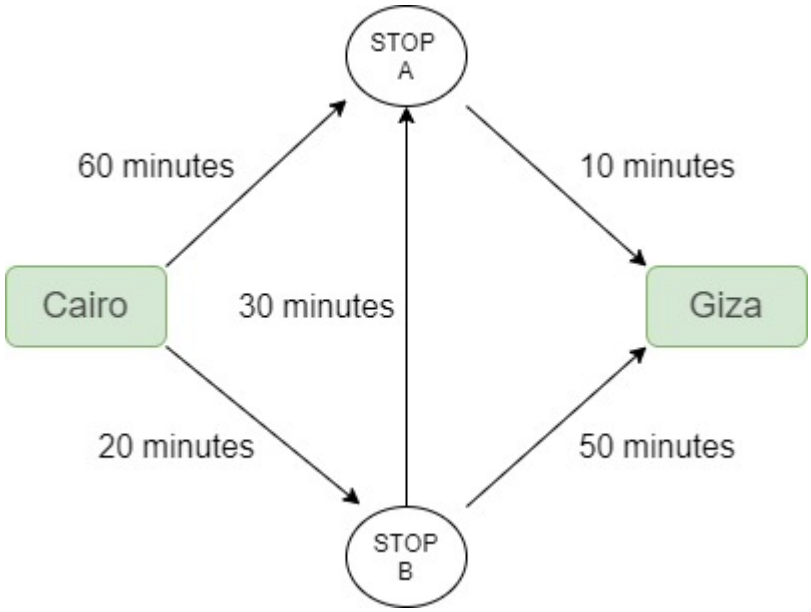


For example, we can see the time difference between the purple and red paths from Cairo to Giza. The red path is faster, in spite of one more segment.

To find the fastest path, we may use a different option to breadth-first search. Instead, we may use *Dijkstra's algorithm* to help us query the graph for the fastest path.

working example - part 1

If we consider the following basic weighted graph, we may initially see how Dijkstra's algorithm works to help us find the fastest path.



Each segment in this graph has a corresponding cost, time in minutes, which we may use with Dijkstra's algorithm to calculate the shortest possible time from Cairo to Giza.

So, we may initially use the following steps with Dijkstra's algorithm to calculate the fastest path from a defined start to finish in the graph.

For the current start node

- 1. identify the *cheapest* node - i.e. the next node we can reach in the least amount of time

We simply check the neighbour nodes for the current node, and identify the path with the shortest time. For example, from our starting point of *Cairo*, we can see there are two options, either *Stop A* with a time of `60 minutes` or *Stop B* with a time of `20 minutes`.

Obviously, we do not know the values of the other nodes at this point of the search. So, as we don't yet know how long it will take to get to the finish, *Giza*, we still note an overall time of `infinity`.

However, we know the closest node from the start, *Cairo*, will be *Stop B* with a time of `20 minutes`.

node	time
Stop A	60 minutes
Stop B	20 minutes
Giza	infinity

- 2. update any costs for neighbours of this node

Now, we need to calculate all times from *Stop B* to available neighbour nodes. In the current example, this includes *Stop A* and our finish node of *Giza*.

So, we may now update our times from the start, *Cairo*, to each node currently known in the graph

node	time
Stop A	50 minutes
Stop B	20 minutes
Giza	70 minutes

The first improvement is a faster time from *Cairo* to *Stop A*, even though we have to go through the node *Stop B*.

With the current known neighbour nodes, we may also follow a path from the start node, *Cairo*, to the finish in *Giza*, which takes `70 minutes`.

So, we currently have a shorter path from *Cairo* to *Stop A*, and a shorter path to the finish from the start.

- 3. repeat this pattern for each node in the graph

We may now repeat the above pattern to check for other neighbour nodes, and potentially faster routes from the start to the finish.

So, we repeat the first step again, which means we need to find the next node with the shortest travel time. We've obviously checked all of the neighbour nodes for *Stop B*, so we can now check the next fastest neighbour of the start node *Cairo*. In the current example, this will be the node *Stop A*.

We don't need to update the time from the start node, *Cairo*, to node *Stop A*, as we've already identified a faster route. However, we may check the times for the quickest route to the finish.

We now have an extra path to check, from *Stop A* to the finish in *Giza*. This gives us a shorter time from the start to the finish, due to its time of 10 minuutes.

We may now update our times as follows,

node	time
Stop A	50 minutes
Stop B	20 minutes
Giza	60 minutes

In effect, we may define the fastest routes for the following paths

- Cairo to Stop A = 50 minutes
- Cairo to Stop B = 20 minutes
- Cairo to Giza = 60 minutes

So, we've been able to identify a quicker path from the start to *Stop A* and, likewise, a quicker path from the start to the finish.

- calculate the time for the final path

So, we may currently define the final path with a calculated fastest time of `60 minutes`. If we compare this calculation with a search using *breadth-first*, we may see that it would not have found that path. Instead, it would have found the shorter path, but a slower path in this example graph.

benefits of Dijkstra

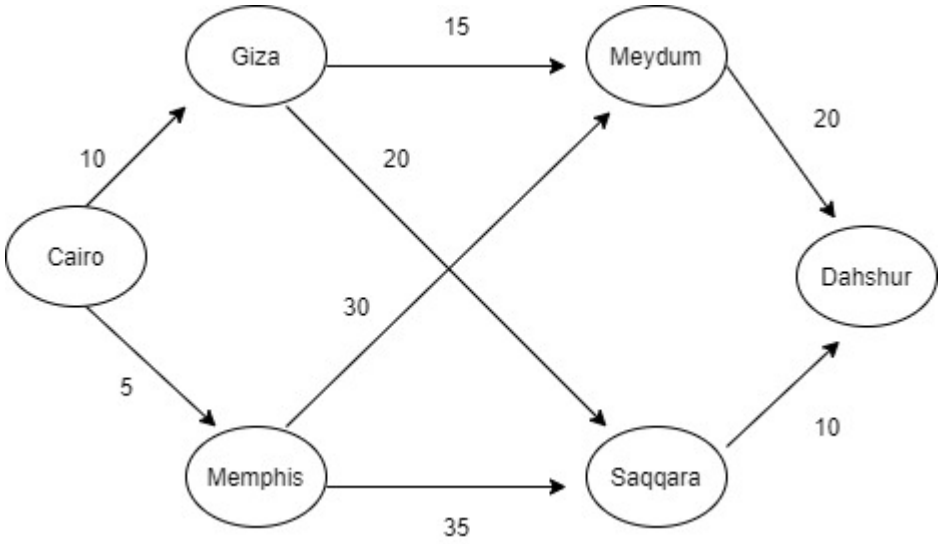
In this current example, we may see an initial benefit, relative to this context, for a search with Dijkstra's algorithm compared with a breadth-first search.

We may use breadth-first search to find the shortest path between defined nodes in the graph. By contrast, we may use Dijkstra's algorithm to assign weights to the graph, which we may use to find the path with the smallest total of calculated weights.

working example - part 2

Let's consider another working example for a graph with weighted edges.

For example, we have a graph with values for cost to travel from one node to another, perhaps from Cairo to Giza or Giza to Saqqara, and so on.



In this graph, we may define weights for the associated costs of travel along each edge. In effect, we may travel from *Memphis* to *Meydum* for $30 or, perhaps, from *Giza* to *Meydum* for only $15.

If we consider this graph, we may need to calculate the cheapest route from *Cairo*, our start point, to an end point of *Dahshur*.

We may, of course, again use Dijkstra's algorithm to perform this calculation. So, we may follow the defined four steps for this algorithm.

Our initial costings may be defined as follows,

parent	node	cost
Cairo	Giza	10
Cairo	Memphis	5
N/A	Meydum	infinity
N/A	Saqqara	infinity
N/A	Dahshur	infinity

and we may set an initial parent for each node. We may then update this table as we execute the algorithm.

- 1. We may start by finding the cheapest node
  - in this graph, from our starting node of *Cairo*, we may quickly see that the cheapest edge is `5` to Memphis
  - we can't make this initial path any cheaper, so the cheapest node will be Cairo to Memphis
- 2. Then, calculate the cost to the neighbours of this cheapest node, i.e. from Memphis
  - we now have costs for Meydum, `30`, and Saqqara, `35`
  - we can update our table of costs from our starting point to each neighbour, Cairo to Meydum and Cairo to Saqqara
  - Cairo -> Memphis -> Meydum
  - Cairo -> Memphis -> Saqqara

parent	node	cost

parent	node	cost
Cairo	Giza	10
Cairo	Memphis	5
Memphis	Meydum	35
Memphis	Saqqara	40
N/A	Dahshur	infinity

So, we now have costs for Meydum and Saqqara. We may define their costs as we travel through the Memphis node. This means, as we can see in the table, that their parent node may also be updated to Memphis.

We may now repeat these two steps for the next cheapest node from Cairo, which is, of course, Giza at `10`. We can update its values as well.

parent	node	cost
Cairo	Giza	10
Cairo	Memphis	5
Giza	Meydum	25
Giza	Saqqara	30
N/A	Dahshur	infinity

We can see that the costs for travel from the starting point, Cairo, to Meydum and Saqqara have been updated. They are cheaper now, so we update the values in the table. In effect, it's now cheaper to travel to Meydum and Saqqara via Giza.

So, we may check the cost to travel to the end point, Dahshur. We check the cheapest node from Giza, which is currently Meydum at `15`. We may update its neighbours, which gives us an initial cost for Dahshur of `20`.

If we update the table at this point, we get the following travel cost from Cairo to Dahshur

parent	node	cost
Cairo	Giza	10
Cairo	Memphis	5
Giza	Meydum	25
Giza	Saqqara	30
Meydum	Dahshur	45

So, we finally have an initial travel cost from start to finish of `45`. i.e. from Cairo -> Giza -> Meydum -> Dahshur

However, we may also check the next cheapest node from Giza, which is Saqqara. Then, we may travel from Saqqara to Dahshur for a total of `40` from Cairo.

In effect, we may now update the cost of travel from start to finish to a lower overall cost of `40`

parent	node	cost
Cairo	Giza	10
Cairo	Memphis	5
Giza	Meydum	25
Giza	Saqqara	30
Saqqara	Dahshur	40

So, we may see that the shortest path costs `40`. Using this overall cost, we may now define the path for travel from start to finish in this graph.

To help with this path definition, we may check the parent node set in the last table. In this example, we ended up with *Saqqara* as the parent for the end point *Dahshur*.

So, we know that we need to travel from Saqqara to Dahshur. We may then follow the path to the parent of Saqqara, which was set to Giza. In effect, to travel to Saqqara we need to begin at Giza.

Likewise, we follow Giza back to its parent, the starting point at *Cairo*.

We now have a complete route for traveling from the start point to the end point in the least cost, `40`.