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

- Dr Nick Hayward

A brief intro to *graphs* for data structure and algorithm usage.

Contents

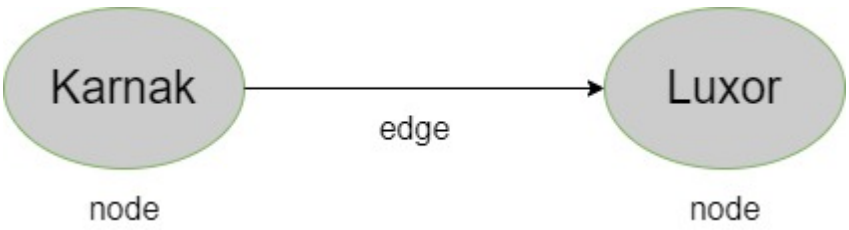
Intro

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.

Sample use case

A common use-case for describing conceptual use of graphs is to consider travel options and routes between various locations.

For example, we might consider traveling around Egypt to visit the many wonderful historical sites.

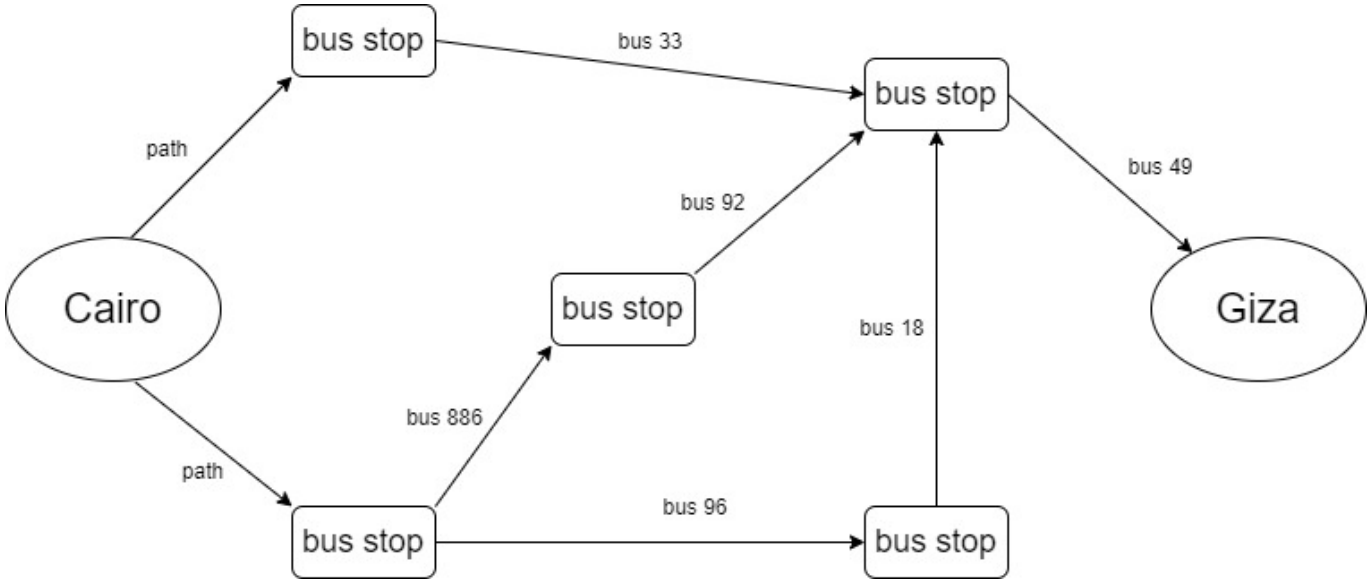
We might need to travel, for example, from the centre of Cairo to Giza to view the pyramids and the Sphinx.

We may use a bus to travel from the centre of Cairo to the Giza plateau, but we need to optimise this route with the minimum number of possible connections.

In effect, we may have numerous options for the available bus routes. However, there will be an optimal choice to allow us to find the path with the fewest steps.

The first step to solve this problem is to define it as a *graph*.

For example, we might consider the following routes



optimal path

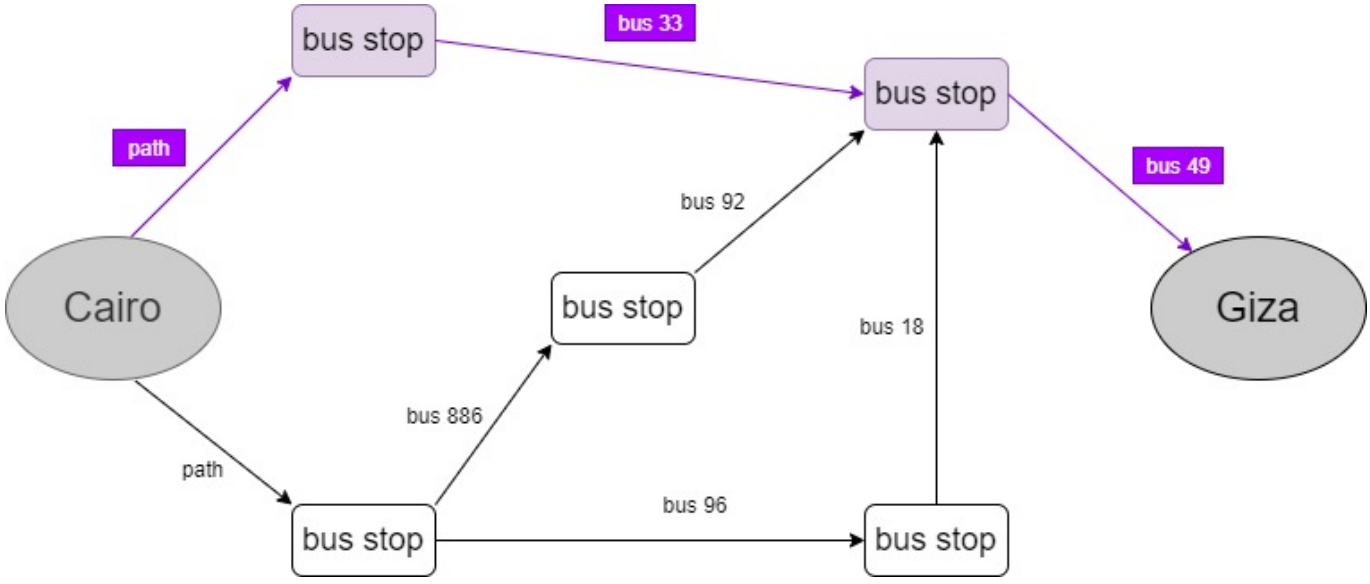
So, we need to define an algorithm to find the optimal path to travel from Cairo to Giza.

We may begin by checking if we can take a single *step* to get from Cairo to Giza. Obviously, this option is not available for the current routes.

We may then try two steps but, again, we can clearly see this is not possible.

However, if we try three steps we can travel from Cairo to Giza.

For example,



We need to take the path to the first bus stop, then take *bus 33* to the next bus stop, and then travel on *bus 49* to the final destination, *Giza*.

This means it will take us *three* steps to travel from the centre of Cairo to Giza. There are, of course, other possible routes using various combinations of buses. However, they are each longer than the optimal route with *three* steps.

This type of problem is formally known as the *shortest path problem*. We may use a *breadth-first search* algorithm to initially consider and solve this type of problem.

Breadth-first search

Breadth-first search is an algorithm we may use to query a *graph* data structure.

In effect, we may use this search algorithm to check for a couple of initial queries

- can we find a path from one node to another - does a path exist?
    - e.g. from node 'Cairo' to node 'Giza'
- what's the shortest path between two nodes
    - e.g. shortest path from 'Cairo' to 'Giza'

Conceptually, we used *breadth-first* to determine the shortest path for the previous use case, from 'Cairo' to 'Giza'.

does a path exist?

We may use *breadth-first* search to check for a given node in a defined graph.

For example, we might begin with an initial list of family members. We may use this list to check for any family member who has visited a specific location in Egypt, perhaps 'Giza' or 'Karnak'.

This seems like a straightforward initial search. We may begin by defining a list of current family members, which we'll use to start our search.

As we check each family member in the list, we simply check whether they have visited, perhaps, Karnak in Egypt or not.

However, an initial search shows us that there are no family members who have visited the site of Karnak. Instead of closing the search, we may, instead, expand the list to search through their family members as well.

So, as we search the records of each family member, we may also add all of their family members to the list.

We're now able to search all of our family members, and a growing network of additional, connected family members.

If a given family member has not visited Karnak, we may then add their family members and continue the search.

With this particular algorithm, we may search the entire network until we find someone who has visited Karnak. In effect, we're checking to see if a path does exist in the graph, someone who has visited Karnak.

find the shortest path

As we search our list, we may find multiple family members that have visited Karnak.

However, which family member is closest? In effect, what is the shortest path between nodes.

So, can we find the closest visitor to Karnak.

If we consider the list of family members, our initial family members are defined as a *first-degree* connection, whilst their family members are *second-degree* connections, and so on.

For search performance, we would naturally prefer a *first-degree* connection, then *second-degree*, until we find the shortest path to a match.

With this in mind, we need to search all *first-degree* connections before we check *second-degree*, and then continue to broaden the search.

This search pattern is, of course, *breadth-first* search.

In effect, this search algorithm will continue to radiate out from a defined starting point. As mentioned, we begin with first-degree connections, then radiate out to second-degree, then third-degree, and so on. We continue to check each level of connections until we find the nearest match for the given search query.

precedence

If we consider this radiated search of connections, we may also see how nodes may be checked as they're added to the search list.

As mentioned above, we will search nodes for first-degree connections before any of the second-degree connections.

So, we can see how *breadth-first* may be used to find a path from one node to another, and the shortest path as well.

This is possible because we define a search with a precedence of insertion. In effect, we search nodes in the same order they were added.

queues

To help search with an order of precendence, we may use a *queue* data structure to ensure we check nodes in the order they were added.

As we see with a stack, we may not access random elements in a queue. This is particularly useful as it enforces two operations we may use,

- enqueue
- dequeue

As expected, if we *enqueue* node `A` and then node `B`, node `A` will be *dequeued* before node `B`.

We can see how this data structure follows a pattern of *first in, first out*, or *FIFO*.

So, we may use this type of data structure to query the list of family members, and their connections as well, using breadth-first search.

Implement a graph

We may initially consider options for implementing a graph with Python.

As we've seen, a graph is a series of nodes with various connections to neighbouring nodes.

For example, we may represent a relationship such as

```
cairo -> giza
```

To implement this type of relationship in code, we may consider a *hash table*.

A *hash table* allows us to map a *key* to a *value*. In our current example, we need to match a node to all of its neighbours.

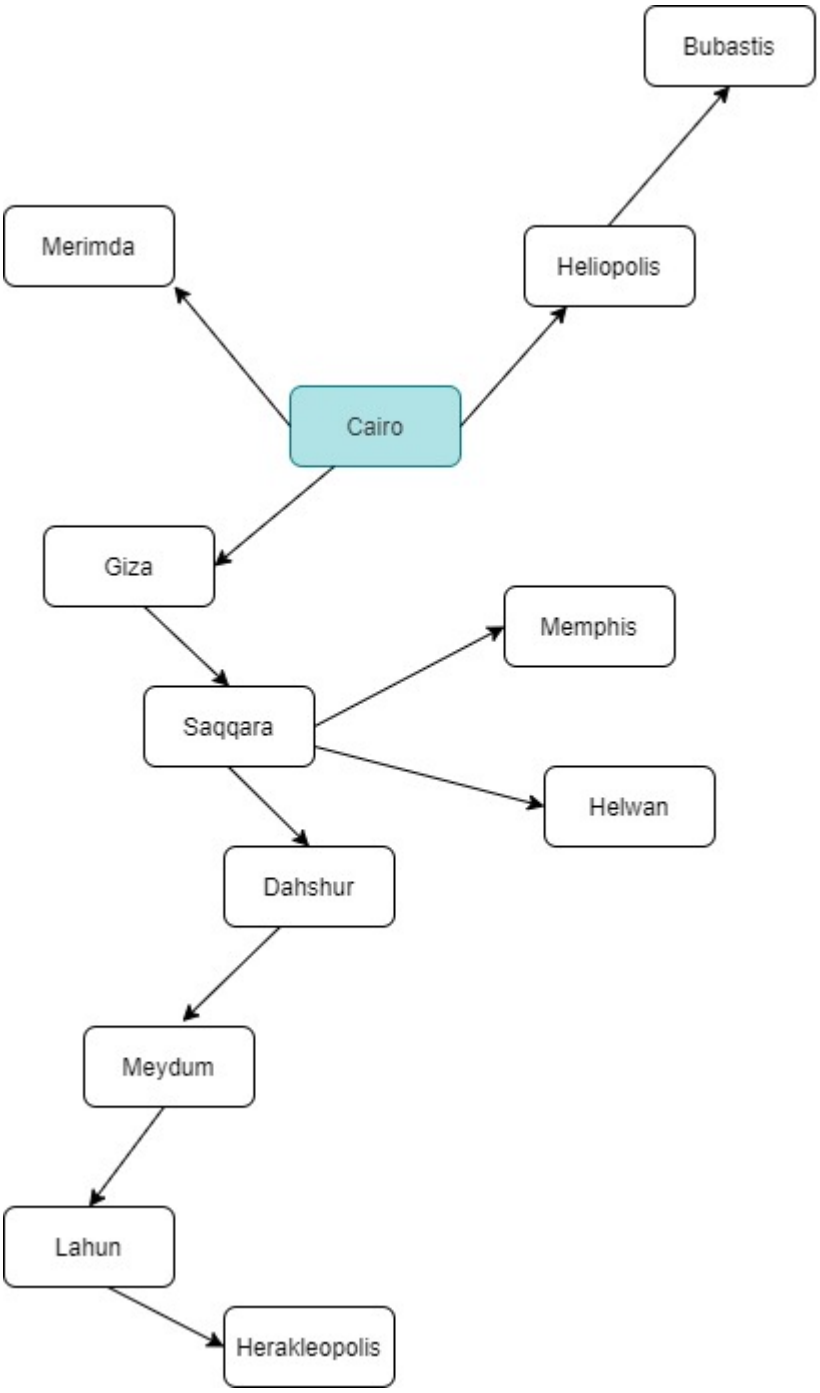
We might initially implement this structure in Python as follows,

```
graph = {}
graph["cairo"] = ["giza", "merimda", "heliopolis"]
```

So, we *map* the defined neighbouring nodes to an array for the node `cairo`.

As noted above, all we need for our graph in Python is a representation of its nodes and edges.

For example, if we consider a larger graph



we may implement it using Python as follows

```
graph = {}
graph["cairo"] = ["giza", "merimda", "heliopolis"]
graph["heliopolis"] = ["bubastis"]
graph["giza"] = ["saqqara"]
graph["saqqara"] = ["memphis", "dahshur", "helwan"]
graph["dahshur"] = ["meydum"]
graph["meydum"] = ["lahun"]
graph["lahun"] = ["herakleopolis"]
graph["merimda"] = []
graph["bubastis"] = []
graph["memphis"] = []
graph["helwan"] = []
graph["herakleopolis"] = []
```

As we compare the diagram of the graph, and the coded example with Python, we may consider whether the insert order matters.

If we consider the underlying data structure, a hash table, we don't need to worry about the order of insertion for the defined key/value pairs.

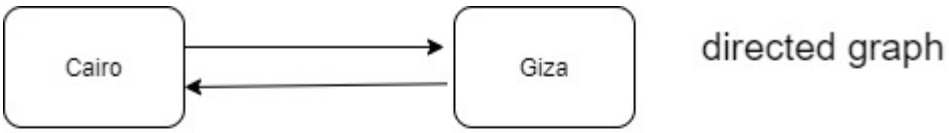
We can also clearly see how some of the nodes do not have any defined neighbours in this graph.

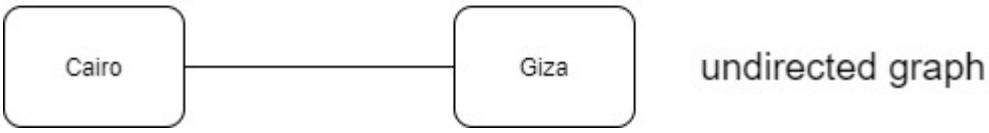
So, this current example is known as a *directed graph*, which reflects the one-way relationships for nodes and neighbours.

In the current example, *Saqqara* is the neighbour of *Giza* but *Giza* is not a neighbour of *Saqqara*. This is shown in the diagram as a single directed arrow.

An *undirected graph*, by contrast, defines both nodes as neighbours and does not use directed arrows in example diagrams.

However, we may represent such connections in both a directed and undirected graph. For example,





- directed graph - both nodes are represented as neighbours
- undirected graph - default usage, both nodes are neighbours