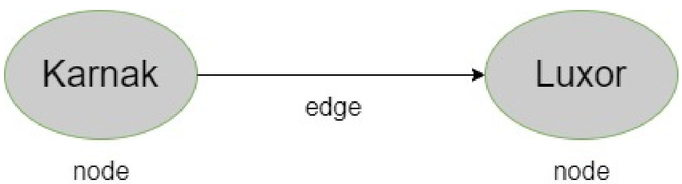
Comp 460 - Algorithms & Complexity

Spring Semester 2020 - Week 13

Dr Nick Hayward

graphs - intro recap

- graph data structure in computer science
- a way to model a given set of connections
- commonly use a *graph* to model patterns and connections for a given problem
- e.g. connections may infer relationships within data
- graph includes *nodes* and *edges*
  - *help us define such connections*
- e.g. we have two nodes with a single edge

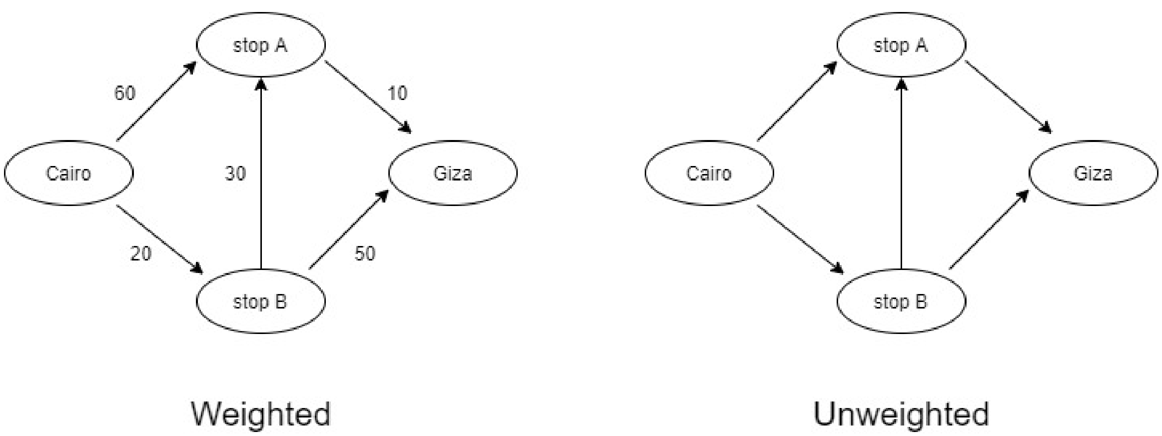


Graph Nodes and Edge

- each node may be connected to many other nodes in the graph
  - *commonly referenced as neighbour nodes*

graphs - Dijkstra’s algorithm - consideration of terminology - part 1

- key concept for working with Dijkstra’s algorithm is the association of values,
  - *e.g. numbers for each edge in the graph*
- valyes are the weights assigned to the edge in the graph
- when we assign weights to an edge
  - *creating a weighted graph*
- if we do not assign weights to edges
  - *defining an unweighted graph*
- e.g. weighted and unweighted graphs



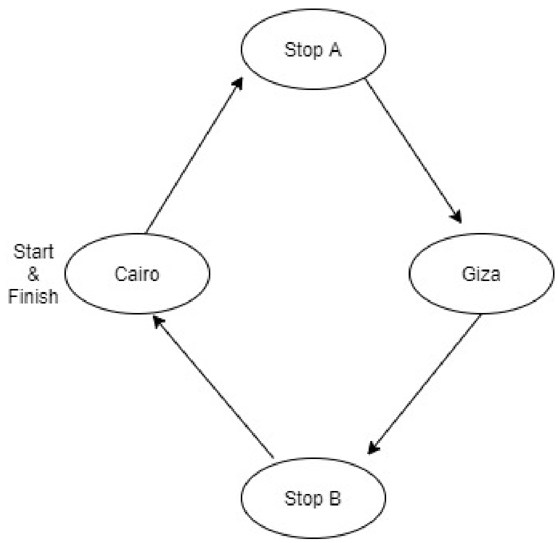
Graphs - Weighted and Unweighted

graphs - Dijkstra's algorithm - consideration of terminology - part 2

- choice of weighted versus unweighted
  - *also affecta our choice of algorithm*
  - *and the context of its usage*
- e.g. if we want to calculate the shortest path in an *unweighted* graph
  - *we may use the breadth-first search algorithm*
- to perform a similar calculation for a *weighted* graph
  - *we may use Dijkstra's algorithm...*

graphs - Dijkstra's algorithm - graph cycles - part 1

- may also encounter a graph with *cycles*
- i.e. we can cycle from `Stop A` back around to `Stop A`
- e.g.



Graph - Cycle

- we may start and finish at the same node in the graph

Algorithms and Data Structures

graphs - Dijkstra's algorithm - graph cycles - part 2

- if we consider a graph with a cycle segment
  - *need to calculate shortest path between two defined nodes*
- for most calculations commonly choose a path that avoids the cycle
- cycle will usually add greater weight to the calculation
- if we then follow cycle more than once
  - *simply adding extra weight to calculation for each completed cycle*
- if we consider an *undirected graph*
  - *now working with a cycle*
- connected nodes in an undirected graph point to each other
  - *effectively a cycle*
- each edge will add another cycle to an undirected graph
- *Dijkstra's* algorithm only works with *directed acyclic graphs* (DAGs)

Algorithms and Data Structures

graphs - Dijkstra’s algorithm - shortest path - part 1

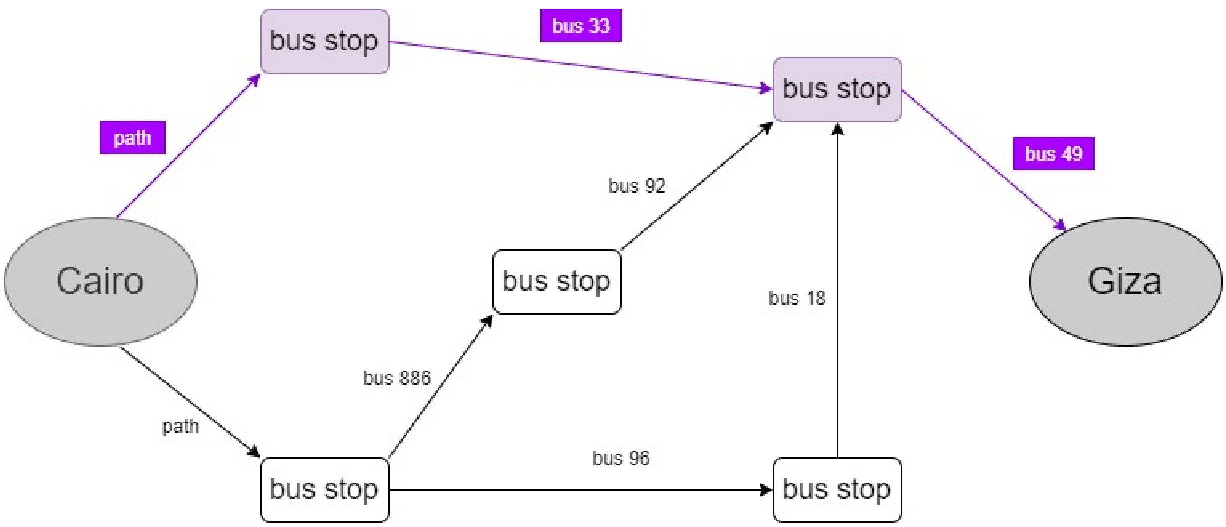
- common requirement for working with graphs
    - *a consideration of weighted and unweighted edges*
- with weighted graphs
    - *interested in options for assigning more or less weight*
    - *i.e. to edges within the graph*
- Dijkstra’s algorithm helps us work with queries for paths in our graphs
- e.g. 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?
    - *this path may not be fastest*
    - *but it will be the shortest in the graph*
- shortest because it will include least number of edges between nodes

graphs - Dijkstra's algorithm - shortest path - part 2

- e.g. we might consider the following graph for the shortest route



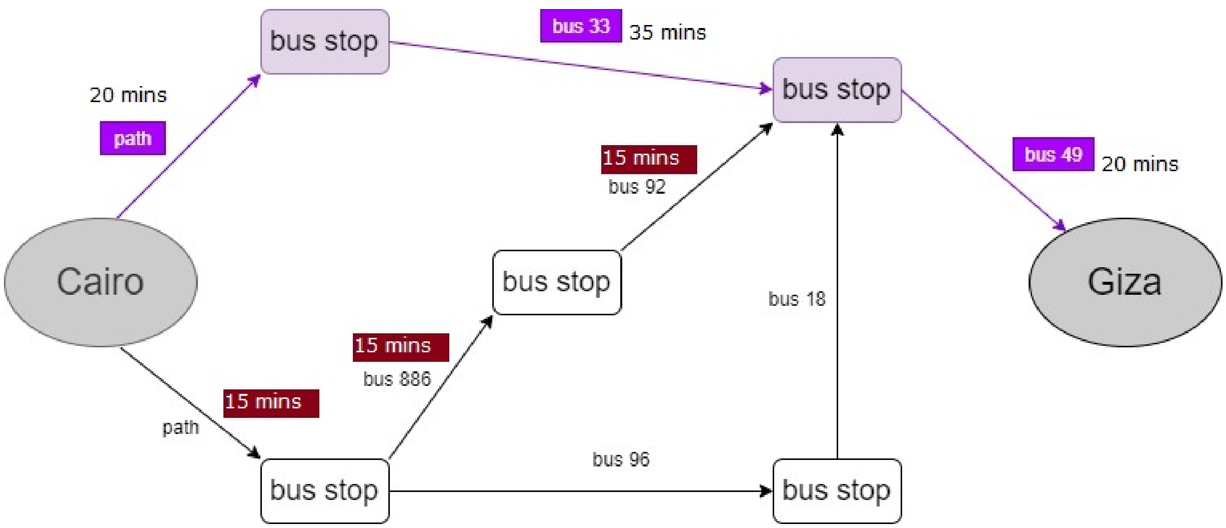
Graph Routes - shortest

- clearly see shortest path with least number of segments, three
- may use *breadth-first* search to find path with fewest segments

Algorithms and Data Structures

graphs - Dijkstra's algorithm - shortest path - part 3

- if we then added travel times to edges for competing routes
  - *i.e. costs for each edge*
- we may find a quicker path for traveling

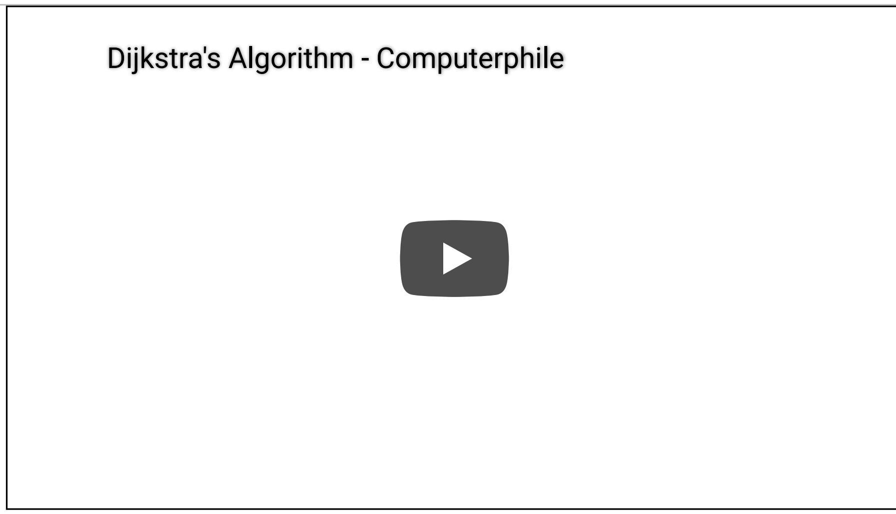


Graph Routes - shortest

- e.g. we can see time difference between purple and red paths
  - *from Cairo to Giza*
- red path is faster, in spite of one more segment
- to find the fastest path
  - *may use a different option to breadth-first search*
- we may use *Dijkstra's algorithm* to help us query graph for fastest path

Video - Algorithms and Data Structures

graphs - Dijkstra’s algorithm - part 1

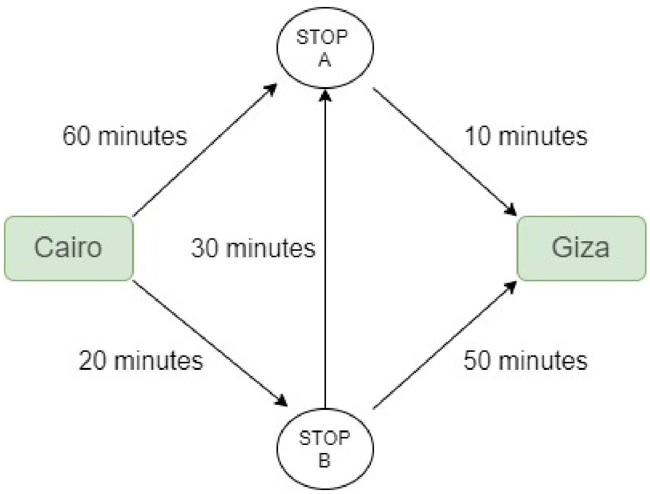


Graphs - Dijkstra’s algorithm - intro - UP TO 1:01

Source - Dijkstra’s algorithm - YouTube

graphs - Dijkstra's algorithm - working example 1 - part 1

- if we consider the following basic weighted graph
  - *may initially see how Dijkstra's algorithm works*
  - *i.e. to help us find the fastest path*



Dijkstra - example graph

- each segment in this graph has a corresponding cost
  - *time in minutes*
- may use *cost* with Dijkstra's algorithm
  - *calculate shortest possible time from Cairo to Giza...*

graphs - Dijkstra's algorithm - working example 1 - part 2

- may initially use the following steps with Dijkstra's algorithm
  - *i.e. to calculate fastest path from a defined start to finish in the graph*
  - *for the current start node*
- *1. identify the cheapest node*
  - *i.e. next node we can reach in least amount of time*
- simply check neighbour nodes for current node
  - *identify path with shortest time*
- e.g. from our starting point of *Cairo* there are two options
  - *either Stop A with a time of* `60 minutes`
  - *or Stop B with a time of* `20 minutes`
- we don't know values of other nodes at this point of search
- as we don't yet know how long it will take to get to finish
  - *Giza defined with an overall time of* `infinity`*...*

Algorithms and Data Structures

graphs - Dijkstra's algorithm - working example 1 - part 3

- we know closest node from start, *Cairo*
- *Stop B* with a time of `20 minutes`

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

Algorithms and Data Structures

graphs - Dijkstra's algorithm - working example 1 - part 4

- *2. update any costs for neighbours of this node*
- need to calculate all times from *Stop B* to available neighbour nodes
- in current example
  - *includes Stop A and our finish node of Giza*
- may now update our times from start, *Cairo*
  - *update to each node currently known in the graph*

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

graphs - Dijkstra's algorithm - working example 1 - part 5

- first improvement is a faster time from *Cairo* to *Stop A*
  - *even though we have to go through node Stop B*
- with current known neighbour nodes
  - *may also follow a path from start node, Cairo,*
  - *follow to finish in Giza*
  - *path takes* `70 minutes`
- currently have a shorter path from *Cairo* to *Stop A*
- plus a shorter path to finish from start...

graphs - Dijkstra's algorithm - working example 1 - part 6

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

- may now repeat pattern to check for other neighbour nodes
  - *potentially faster routes from start to finish...*
- repeat first step again
  - *need to find next node with shortest travel time*
- checked all of the neighbour nodes for *Stop B*
- we can now check next fastest neighbour of start node *Cairo*
- in current example
  - *this will be node Stop A...*

graphs - Dijkstra's algorithm - working example 1 - part 7

- don't need to update time from start node
  - *i.e. from Cairo to node Stop A*
  - *already identified a faster route...*
- we may check times for quickest route to finish
- now have an extra path to check
  - *i.e. from Stop A to finish in Giza*
- gives us a shorter time from start to finish
  - *due to its time of* `10` *minuutes...*

Algorithms and Data Structures

graphs - Dijkstra’s algorithm - working example 1 - part 8

- now update our times as follows

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

- i.e. define fastest routes for following paths
  - *Cairo to Stop A = `50` minutes*
  - *Cairo to Stop B = `20` minutes*
  - *Cairo to Giza = `60` minutes*
- able to identify a quicker path from start to *Stop A*
  - *and a quicker path from start to finish…*

graphs - Dijkstra's algorithm - working example 1 - part 9

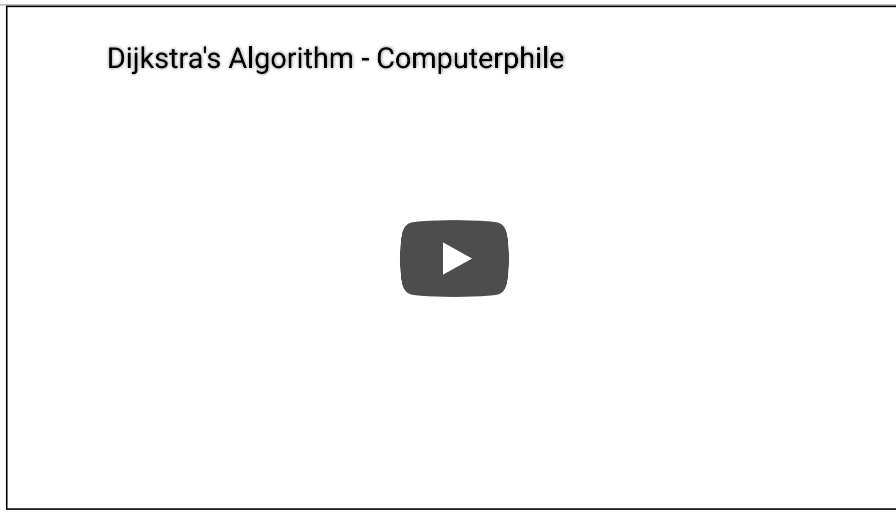
- calculate the time for the final path
  - *may currently define final path*
  - *calculated fastest time of* `60 minutes`*...*
- if we compare this calculation with a search using *breadth-first*
  - *may see that it would not have found that path*
- *breadth-first* would have found shorter path
 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

graphs - Dijkstra's algorithm - working example 1 - benefits

- in current example, we may see an initial benefit
  - *benefit relative to this context*
  - *e.g. for a search with Dijkstra's algorithm compared with a breadth-first search*
- may use breadth-first search to find shortest path
  - *e.g. between defined nodes in graph*
- may use Dijkstra's algorithm to assign weights to graph
  - *use to find path with smallest total of calculated weights...*

Video - Algorithms and Data Structures

graphs - Dijkstra’s algorithm - part 2

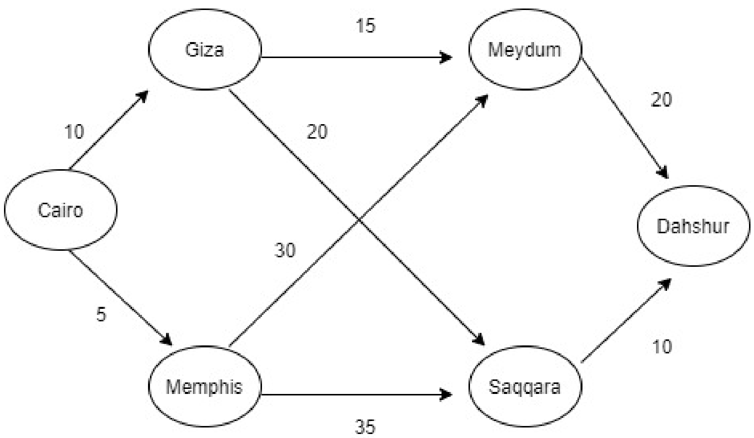


Graphs - Dijkstra’s algorithm - example outline - UP TO 3:57

Source - Dijkstra’s algorithm - YouTube

graphs - Dijkstra's algorithm - working example 2 - part 1

- consider another working example for a graph with weighted edges
- e.g. a graph with values for cost to travel from one node to another
  - *perhaps from Cairo to Giza or Giza to Saqqara...*



Graph Weighted

- in this graph
  - *define weights for associated costs of travel along each edge*
- i.e. travel from *Memphis* to *Meydum* for `30`
  - *or, perhaps, from Giza to Meydum for only* `15`*...*

Algorithms and Data Structures

graphs - Dijkstra’s algorithm - working example 2 - part 2

- if we consider this graph
  - *may need to calculate cheapest route from Cairo*
  - *e.g. start point to an end point of Dahshur...*
- may use Dijkstra’s algorithm to perform this calculation
  - *follow defined four steps for this algorithm*
- 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 set an initial parent for each node...

graphs - Dijkstra's algorithm - working example 2 - part 3

- then update this table as we execute the algorithm
- *1. start by finding cheapest node*
  - *in this graph, from a starting node of Cairo cheapest edge is `5` to Memphis*
  - *we can't make this initial path any cheaper*
  - *cheapest node will be Cairo to Memphis*
- *2. then, calculate cost to 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*

Algorithms and Data Structures

graphs - Dijkstra's algorithm - working example 2 - part 4

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

- now have costs for Meydum and Saqqara
- may define their costs as we travel through Memphis node
- as we can see in the table
  - *their parent node may also be updated to Memphis...*

graphs - Dijkstra's algorithm - working example 2 - part 5

- may now repeat these two steps for next cheapest node from Cairo
  - *i.e. Giza at a cost of `10`*
  - *update its values in the table as well*

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

- the costs for travel from starting point, Cairo, to Meydum and Saqqara updated
  - *they are cheaper now*
  - *so we update the values in the table*
- i.e. it's now cheaper to travel to Meydum and Saqqara via Giza…

Algorithms and Data Structures

graphs - Dijkstra’s algorithm - working example 2 - part 6

- we may check cost to travel to end point, *Dahshur*
- check cheapest node from Giza
  - *currently Meydum at 15*
- may update its neighbours
  - *gives us an initial cost for Dahshur of 20*
- if we update 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

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

graphs - Dijkstra's algorithm - working example 2 - part 7

- we may also check next cheapest node from Giza, *Saqqara*
- then, we may travel from Saqqara to Dahshur
  - *for a total of `40` from Cairo*
- i.e. may now update cost of travel from start to finish
  - *update 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

Algorithms and Data Structures

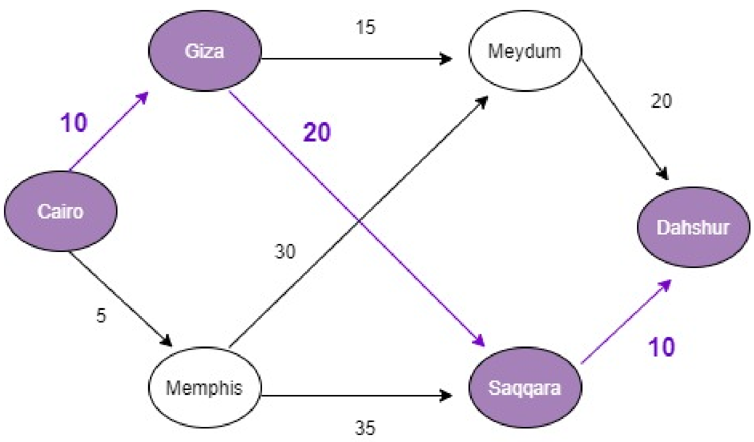
graphs - Dijkstra's algorithm - working example 2 - part 8

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

- we may see that shortest path costs `40`
- using this overall cost
  - *now define path for travel from start to finish in this graph*
- to help with this path definition
  - *may check parent node set in last table*
- i.e. ended up with *Saqqara* as parent for end point *Dahshur*

graphs - Dijkstra's algorithm - working example 2 - part 9

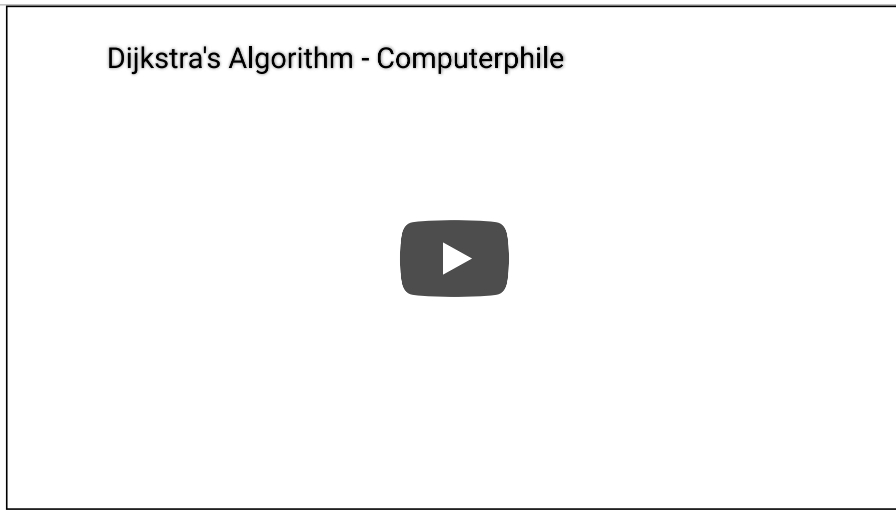
- we know that we need to travel from Saqqara to Dahshur
- may then follow path to parent of Saqqara, set to Giza
- i.e. to travel to Saqqara we need to begin at Giza
- we follow Giza back to its parent
  - *starting point at Cairo*
- now have a complete route for traveling from start point to end point in least cost, `40`



Graph Weighted - Final Costs

Video - Algorithms and Data Structures

graphs - Dijkstra’s algorithm - part 3



Graphs - Dijkstra’s algorithm - example usage - UP TO 8:48

Source - Dijkstra’s algorithm - YouTube

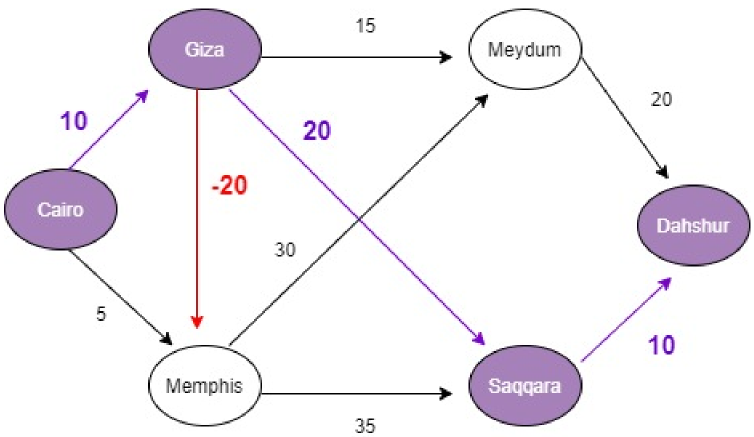
graphs - Dijkstra's algorithm - edges with negative weight - part 1

- in last example
  - *we have weighted edges from Cairo to Giza, and Cairo to Memphis*
- each of these routes has a cost involved,
  - *i.e. the weight of the edge*
- we may now add a path directly from Giza to Memphis
- in current example
  - *this edge will pay us* `20`
  - *we're able to claim the cost back…s*

Algorithms and Data Structures

graphs - Dijkstra’s algorithm - edges with negative weight - part 2

- edge may be defined with a negative weight of `-20`
- now have two routes to consider to allow us to travel from Cairo to Memphis
- might take previous route
  - *direct from Cairo to Memphis*
  - *route will cost 5*
- or we might consider updated route via Giza
- second route, `Cairo -> Giza -> Memphis`
  - *now costs `-10`*



Graph - Negative Weighted Edges

graphs - Dijkstra’s algorithm - Dijkstra and negative weights - part 1

- if we continue path through graph to end point at Dahshur
  - *might consider following this route with a negative weighted edge*
- if we try to perform our usual calculation with *Dijkstra’s* algorithm
  - *end up following more expensive route*
- i.e. negative-weighted edges will break use of Dijkstra’s algorithm
- issue may not be final predicted route, as seen above
- issue is commonly with defined calculations
  - *performed at various stages during algorithm’s execution*

graphs - Dijkstra’s algorithm - Dijkstra and negative weights - part 2

- if we run Dijkstra’s algorithm again on this graph
  - *this time with negative weighted edge*
- we get a false definition for cheapest route to Memphis
- e.g. following standard pattern of calculation
  - *we get the following table of costs*

node	cost
Giza	10
Memphis	5
Saqqara	infinity

- then, we find lowest-cost node
  - *and update costs for each of its neigbours*
- Memphis is initial lowest code node from Cairo with a cost of 5

Algorithms and Data Structures

graphs - Dijkstra's algorithm - Dijkstra and negative weights - part 3

- according to the Dijkstra algorithm
  - *there is no cheaper path to travel from Cairo to Memphis*
- due to *negative-weighted* edge from Giza to Memphis
  - *we know this calculation and assertion is incorrect*
- if we continue to follow Dijkstra's algorithm
  - *we update the table as follows*

node	cost
Giza	10
Memphis	5
Saqqara	40

- then, we get next lowest cost node from Cairo
  - *Giza with a cost of `10`*
  - *and update cost of its neighbours*

graphs - Dijkstra's algorithm - Dijkstra and negative weights - part 4

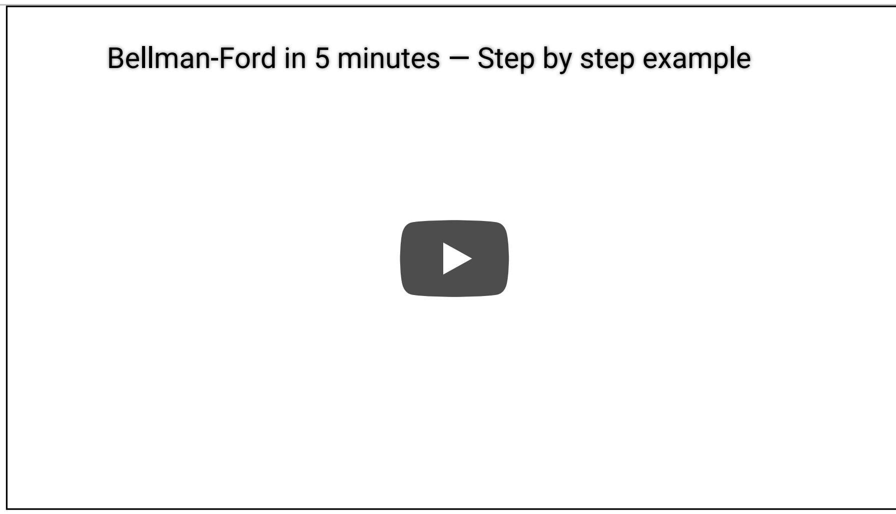
- if we consider neighbours of Giza in updated graph
  - *we have a negative weighted edge from Giza to Memphis*
- issue is trying to update cost for Memphis node
- a clear sign that something is not right with use of algorithm
- already processed Memphis node
  - *i.e. there should not now be a cheaper route to that node*
- due to negative weighted edge
  - *we've actually found a cheaper route*
- if we check the cost up to the node *Saqqara*
  - *algorithm will return already calculated cost of* `40`*...*

graphs - Dijkstra's algorithm - Dijkstra and negative weights - part 5

- due to negative weighted edge
  - *we know there is a cheaper route*
  - *but Dijkstra's algorithm did not find this route*
- algorithm makes an assumption about processing of nodes
  - *due to initial costs of weighted edge...*
- i.e. as we were processing the Memphis node
  - *Dijkstra's algorithm assumes there is now no faster way to that node*
- this assumption only holds true
  - *e.g. if we do not have negative weighted edges*
- *n.b.* we can't use negative weighted edges with Dijkstra's algorithm
- to calculate shortest path in a graph with negative weighted edges
  - *instead, use **Bellman-Ford** algorithm...*

Video - Algorithms and Data Structures

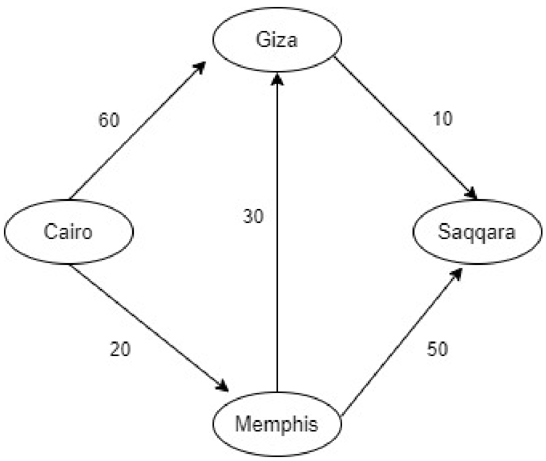
graphs - Bellman-Ford algorithm



Graphs - Bellman-Ford algorithm example - UP TO 4:51

Source - Bellman-Ford algorithm - simple example - YouTube *graphs - Dijkstra’s algorithm - working implementation - part 1*

- now consider a basic coded example of implementing Dijkstra’s algorithm in Python
- for this example, we’ll start with following graph



Graph - Coded Example

Algorithms and Data Structures

graphs - Dijkstra's algorithm - working implementation - part 2

- to help implement a working example for this graph
- define three *hash* tables for use with this implemented working example
- *1. graph*
  - *parent node*
  - *neighbour node*
  - *cost of weighted edge*
- *2. costs*
  - *node*
  - *current cost from start point*
- *3. parents*
  - *node*
  - *current parent node*

Algorithms and Data Structures

graphs - Dijkstra's algorithm - working implementation - part 3

- graph

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

graphs - Dijkstra's algorithm - working implementation - part 4

- costs

node	current cost
giza	60
memphis	20
saqqara	infinity

graphs - Dijkstra's algorithm - working implementation - part 5

- parents

node	current parent
giza	cairo
memphis	cairo
saqqara	-

- as we execute the algorithm
    - *update values for costs and parents tables...*

Algorithms and Data Structures

graphs - Dijkstra's algorithm - working implementation - part 6

implement the graph

- need to implement graph for this coded example
  - *we'll use a hash table for graph*

```
graph = {}
```

- in this hash table
  - *need to store multiple values for neighbours*
  - *then set cost for travel along that edge*
- e.g. for current graph
  - *we can see that Cairo has two neighbours, Giza and Memphis*

graphs - Dijkstra's algorithm - working implementation - part 7

- a number of options we might consider for structuring this pattern of data
  - *including nested hash tables for each node relative to the parent*
- e.g.

```
graph["cairo"] = {}
graph["cairo"]["giza"] = 60
graph["cairo"]["memphis"] = 20
```

- creates following structure for our data

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

- corresponds to structure and values defined in above table for graph

graphs - Dijkstra's algorithm - working implementation - part 8

- we might, of course, check its values as follows

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

- if we need to find weights for edges from `Cairo`
  - *we may call the following*

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

graphs - Dijkstra's algorithm - working implementation - part 9

- following this pattern
  - *add remaining nodes and neighbours to hash table for graph*

```python
# 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"] = {}
```

- hash table now represents graph with defined neighbour nodes and weighted edges
  - *e.g.*

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

graphs - Dijkstra's algorithm - working implementation - part 10

- next structure we need to create is a hash table for costs of each node
  - *i.e. using cost to define value of weighted edge from one node to another*
- *cost* of node will represent calculated total
  - *i.e. for weighted edges from start, Cairo, to a given node*
- e.g. we know it will cost `60` to get from Cairo to Giza
  - *and `20` to get from Cairo to Memphis…*

graphs - Dijkstra's algorithm - working implementation - part 11

- we can represent currently unknown costs as *infinity*
- we may represent this hash table as follows,

```python
# 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}
```

graphs - Dijkstra's algorithm - working implementation - part 12

- then, we may add our third table for *parent* nodes in graph

```python
# 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
```

- update such values as we work through algorithm and its execution
- also need to maintain a record of nodes already processed in graph
    - *i.e. to avoid duplicated effort...*

```python
nodes_checked = []
```

graphs - Dijkstra's algorithm - working implementation - part 13

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

Algorithms and Data Structures

graphs - Dijkstra's algorithm - working implementation - part 14

- initially define `while` loop
  - *and checks we need to perform for each iteration of the loop*

```
# 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)
```

- start by checking passed `cost` table
  - *check for lowest cost node in defined graph...*

graphs - Dijkstra's algorithm - working implementation - part 15

- custom function `find_low_cost_node()` may be implemented as follows

```
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
```

- simple implementation to check for current lowest common node
- use this custom function to get lowest common node
  - *we may then use with the `while` loop...*

graphs - Dijkstra's algorithm - working implementation - part 16

- loop itself may be considered as follows
    - *helps us further understand how implemented algorithm will work with a sample graph*

code breakdown

- e.g. begin by checking for node with lowest cost
    - *i.e. from start point in graph, Cairo*

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

- in the hash table
    - *this check will return Memphis with a cost of* `20`
- we can now get cost for this node
    - *and its neighbour nodes as well...*

graphs - Dijkstra's algorithm - working implementation - part 17

- then add these neighbour nodes to their own hash table

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

- use this structure to loop through stored neighbours

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

- each of these neighbour nodes will have their own cost
  - *detail cost from start node, Cairo, to that node*

graphs - Dijkstra's algorithm - working implementation - part 18

- in effect, we're calculating cost of node from start node
  - *i.e. if we went through the current node*
  - *e.g Cairo -> Memphis -> Giza with an updated cost of* `50`
- updated cost is lower than current cost
  - *for a route from start Cairo to Giza*
  - *cost was previously* `60`
- we can update the cost as follows

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

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

graphs - Dijkstra's algorithm - working implementation - part 19

- new cost is now updated in `cost` hash table as well

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

- may also update parent node for *Giza* in `parents` hash table

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

- now back at start of `while` loop
  - *we may now move on to next neighbour*
  - *Saqqara for current graph*
- we repeat above pattern
  - *checking and updating hash tables for cost of path to current node Saqqara*
  - *the finish node in the current graph...*

graphs - Dijkstra's algorithm - working implementation - part 20

- if we execute this algorithm with the current graph
  - *we get the following initial output*

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

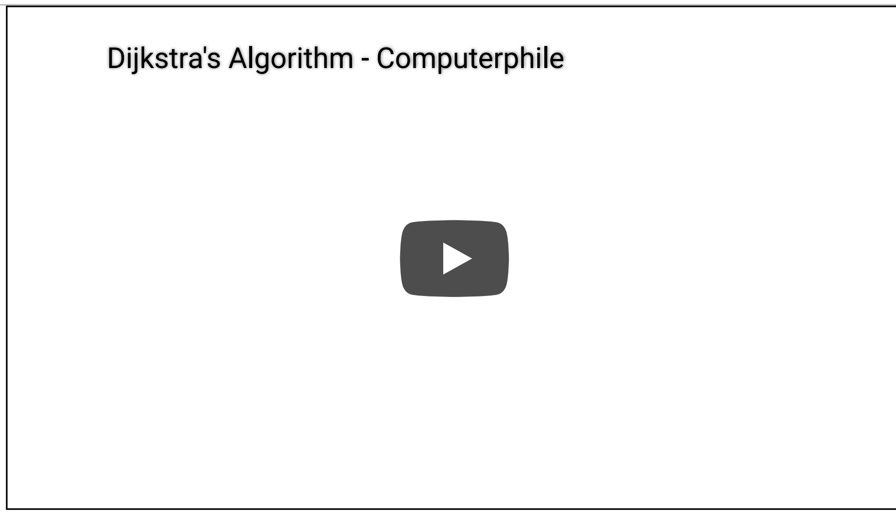
- updated as follows after we run Dijkstra's algorithm

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

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

Video - Algorithms and Data Structures

graphs - Dijkstra’s algorithm - part 4



Graphs - Dijkstra’s algorithm - improve usage - UP TO END

Source - Dijkstra’s algorithm - YouTube

Resources

various

- A* search algorithm
- Bellman-Ford algorithm
- Dijkstra’s algorithm
- Graph - abstract data type

videos

- A* (A star) search algorithm
- Bellman-Ford algorithm - simple example
- Dijkstra’s algorithm