Comp 363 - Design and Analysis of Computer Algorithms

Spring Semester 2020 - Week 13 - Part 2

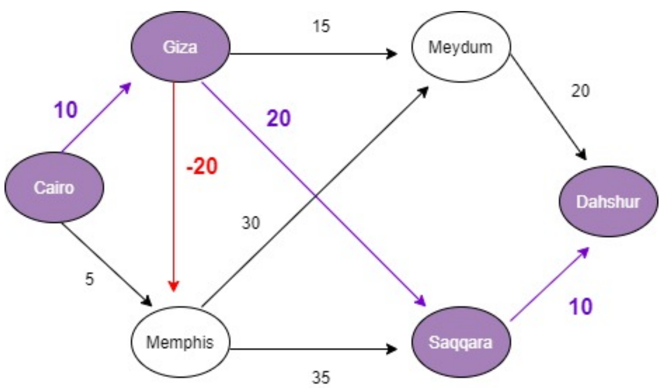
Dr Nick Hayward

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*

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*

Algorithms and Data Structures

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`*...*

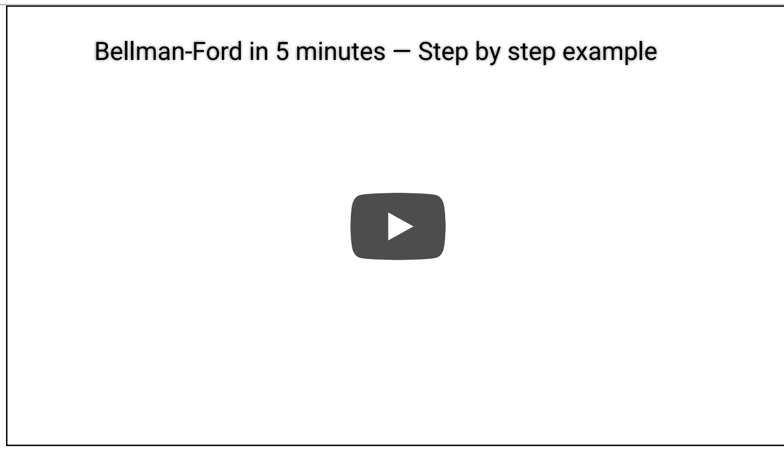
Algorithms and Data Structures

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

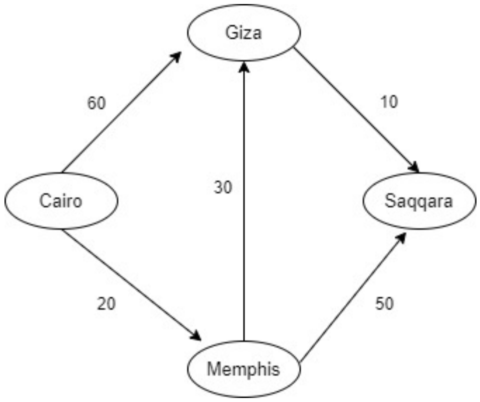


Graphs - Bellman-Ford algorithm example - UP TO 4:51

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

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

Algorithms and Data Structures

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,

```
# 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

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

```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
```

- 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