Application of Dijkstra's Shortest Path Algorithm to Internet Routers

MSIN0023

Computational Thinking
Individual Project
Word count: 1881

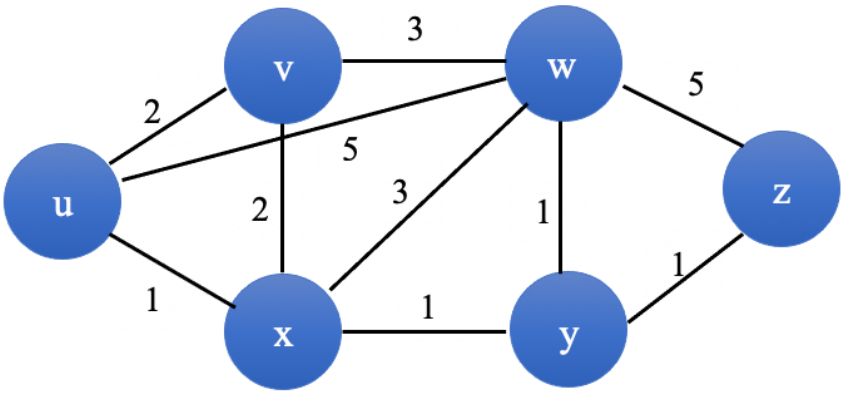
1. Algorithm and System Requirements

Whenever people carry out a browser search via a search engine, the search request must first 'travel' over the local area network through a router. Then the same request travels back, arriving at a router where the server is placed, ultimately generating the search result we see in the browser.

These routers are interconnected and must collectively work together to enable information to move around each. There are costs associated with each link between pairs of routers resulting from many factors such as intensity of people using the server and various routers exchanging information at different times. Maintaining this connection of routers and establishing an efficient path with shortest distances i.e. least costly are crucial to save money, time, and effort.

Cisco is the worldwide leader in IT, networking, and cybersecurity. Their systems' high performance highly depends on the functionality of their multiprotocol router systems. As a reputable company with ambitious claims, they aim to provide non-interrupted connections, which highly relies on establishing the network of routers in the most cost-efficient way to enhance profit and success.

These networks of routers can be represented "as a graph with weighted edges" where the path with the smallest total weight is chosen to route any given request. This problem can be tackled and implemented using Dijkstra's Shortest Path algorithm. Dijkstra's algorithm utilizes weights of the edges to find the path that minimizes the total distance between the selected node and other nodes represented my numbers in Graph 1.[1]



Graph 1 (n=6)

In terms of the functional requirement of the system, the code works by taking inputs from a network which are vertices and weights of edges, as a list of sets. Then the code converts them into the shortest distances between the chosen vertex and all the other nodes.

[1] (Bradfields School of Computer Science., n.d)

through implementation of a priority queue. The logic that governs key functions in the specific business application is, reducing the costs of the inter-connections of routers aiming to achieve cost-efficiency.

In terms of non-functional requirements; the size of the data increases proportionally to the number of vertices and edges. The weights assigned to each path linking to the selected node also contribute to the growth of the data considering they ultimately affect the shortest distances. The total running time of the algorithm performing the process with 6 vertices is $\cong 0.000507$, which has been averaged over 5 different runs. When the complexity increases, i.e. when 14 vertices are introduced, the running time increases to $\cong 0.00320$, which is almost 6 times the initial value. We must note that even though the increased data load isn't very large we observe a much slower response suggesting the algorithm may not perform well under data load.

Since the nature of the data also affects the running time of the algorithm, the way the data is presented to algorithm matters, for instance presenting the list of distances of each node with the selected node in a sorted way would decrease the running time as fewer steps are required. The number of links made with each node and their corresponding weights also affects the running time highly since they require additional computational power.

2. Overview of Code

Dijkstra's algorithm is implemented by assigning a label to each vertex within a graph identifying the shortest path from the starting vertex to all other vertices in the graph. The code works in a decomposed way; in a hierarchy of functions building up to define our priority queue. Defining the priority queue requires us to define functions; `my_heappush()` and `my_heappop()`. These subroutines rely on other subroutines `my_siftdown()` and `my_siftup()`.These functions are a part of the source codes[2] of Python's Heap queue algorithm implementing a priority queue(`pq`): deciding which paths will be included in shortest paths to a given node.

Initially, the customized function `calculate_distances()` is defined to train the algorithm to give out distances between a selected node on the network and the chosen starting vertex. `Distances` dictionary comprises 2 distances; the initial distance from the starting vertex to itself is defined to be zero and since the distance between the starting vertex to other vertices is unknown, their distance to the starting vertex is set be infinity.

Dijkstra's Algorithm utilizes a priority queue where the entries are coupled as (`distance, vertex)` ensuring that the order of vertices are sorted by distance. This is handled by re-entering another couple with different distance information for the same vertex.

[2] (Delfino, 2020)

The code lines build up on each other establishing a priority listing of paths leading to the selected node. `my_heappop()` ensures that at each iteration through vertices, the lowest costing path is being chosen.

The algorithm iterates only once for each vertex and the order of this iteration is controlled by the priority queue with implementations of loops, it determines the order with respect to the distance from the starting vertex. The algorithm relies on the mission of updating this priority queue as distances from vertices reduce, hence giving the `(distance,vertex)` tuple a different priority.

Algorithm works sequentially aiming to decrease the distances to selected nodes, when all vertices are visited the algorithm stops. If not, the second if loop ensures that from the unvisited vertices, the smallest distance path to selected vertex is chosen, for each neighbouring vertex the distances are iteratively checked to see if the distance from the starting node to that vertex is smaller than the previously noted distance, if so the distances are adjusted accordingly. `my_heappush()` pushes the `(distance, neighbour)` couple onto the priority queue, maintaining the queue invariant.

Printing the customized function `calculate_distances()` to an example network, and specifying a selected node, would print the output as the shortest distance of each vertex from that network to that selected node.

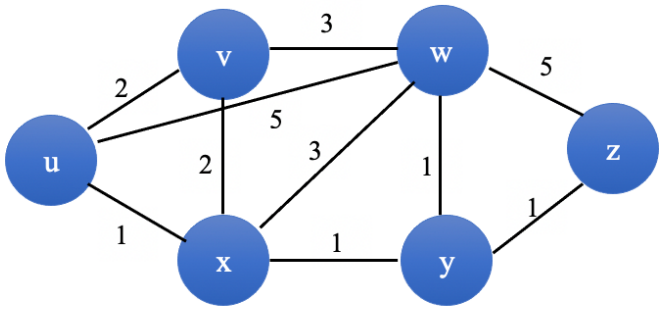
3. Complexity of the Code

Considering there are millions of internet routers around the globe that continuously exchange information, the graph network could easily get complex as there are many possible combinations of vertex numbers and associated distances.

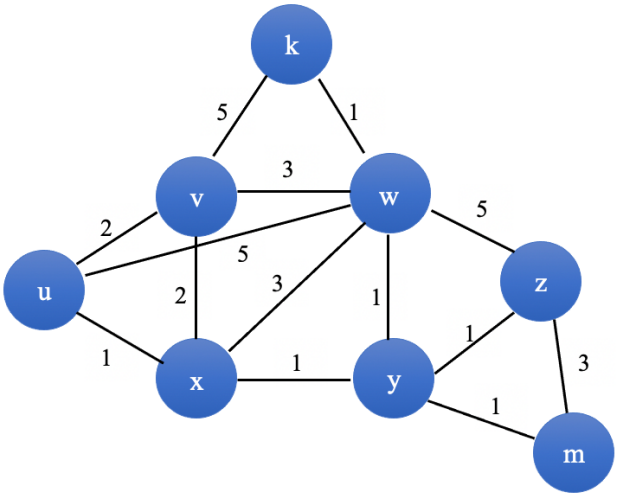
'Big(O)' expressions are used to indicate complexity in terms of "order of growth".
To assess the complexity, we can count the number of iterations made around the loops as the function of our input parameter returns distances.

In the algorithm there are nested loops that contribute to the complexity, however in the calculation of order of growth, we only consider the highest order complexity term which consists of a nested for loop inside a while loop. Hence, the order of growth of the algorithm could be assumed to be $O(n^2)$ where n is the number of vertices.

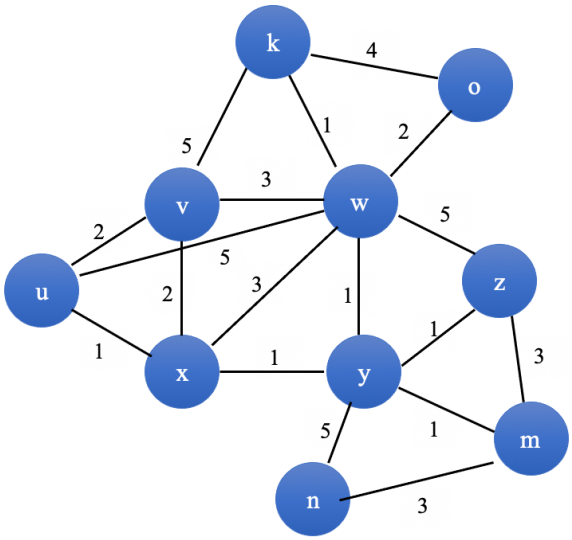
Big(O) plot can be generated considering a range of different input size values based on the running times of the algorithm. The algorithm has been executed four different times with varying sample sizes. As sample size increases (i.e., vertices get added), the algorithm is expected to run slower as it gets more complex. To consider various sample sizes, the inter-connections of below graphs should be visualized:



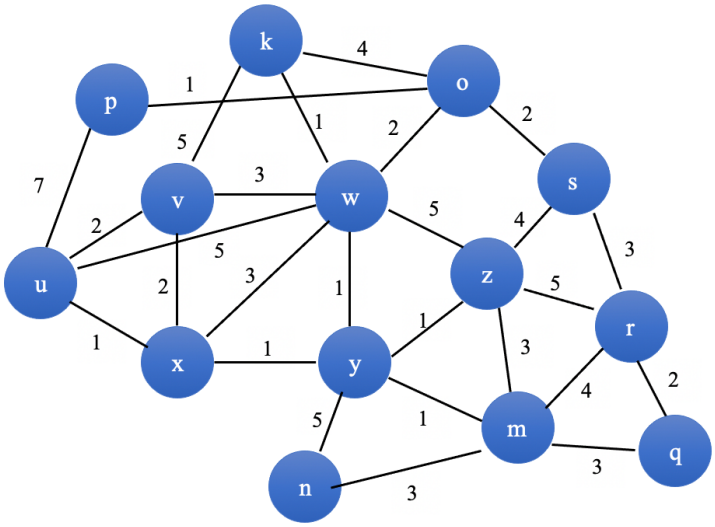
Example Graph (n=6)



Example Graph 2 (n=8)



Example Graph 3 (n=10)



Example Graph 4 (n=14)

To count the iterations computed in a while loop, a counter variable “c” is defined, initially set to zero. As the algorithm processes new entries to the Priority Queue, c gets incremented by 1. This is done until all shortest paths between nodes are discovered, hence the final c value will represent the number of iterations computed.

As sample size increases, additional iterations are required to be performed as the data gets more complex. Implementing `time.time()` function from python's `time` module, we can note the total running times as well:

Sample Size (number of vertices)	Number of Iterations (c value)	Total Running Time (secs)
6	6	0.0005068778991699219
8	9	0.0006279945373535156
10	12	0.0009620189666748047
14	17	0.0031957626342773438

Based on the above calculations with four different n values, a plot can be generated displaying the increase of running time over increase in sample size. As sample size gets larger, the total running time plot starts to resemble $f(n) = n^2$ curve. $O(n^2)$ indicates a polynomial running time.

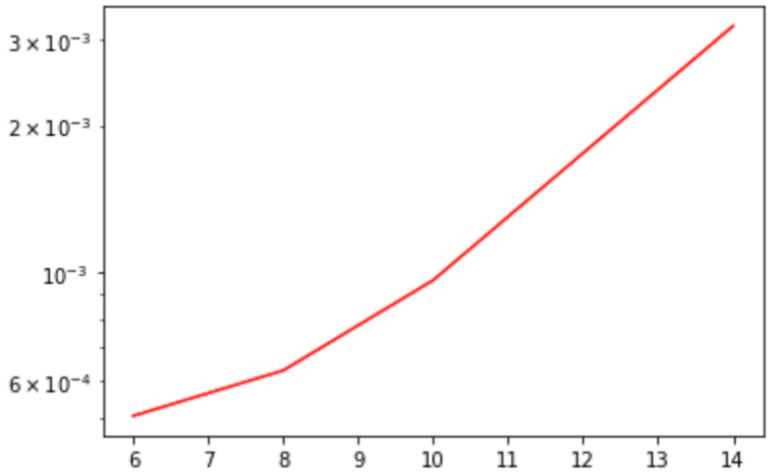


Figure 1

As it can be observed in Figure 1, there is a non-linearly increasing trend in the execution time of the algorithm when the number of vertices is increased. This can be attributed to the nested loop structure of the algorithm, which leads to a polynomial increase in the number of calculations once a new vertex is added.

From this plot it can be derived that if sample size is very large meaning a very large number of computing steps are required the algorithm might potentially take a lot of running time. This might prove problematic when the problem being dealt with requires a near-immediate

solution as in our case where the router's ability to connect and exchange information in the fastest way holds paramount importance.

4. Description of Data

The data used to illustrate how the algorithm helps achieve cost-efficiency in router systems is an open-source data taken from the internet. However, for simplicity an initial exemplary graph was formed with representative number resembling the router systems

The data consists of, each node, their neighbouring nodes, and distances to each given node. This way for any given sample size the shortest distances to given nodes are printed

Additional nodes were manually added to the data to assess complexity, appending on the initial graph (i.e. adding new nodes and weights). Number of vertices added and the distances between these vertices were randomly selected.

The algorithm relies on data being represented in the form of network, based on the number of vertices and entries added to priority queue the size of the data can be varied. The network is presented to the algorithm as a list of sets based on the order and weights connecting the nodes.

5. Conclusion

Code shows good design principles in terms of the hierarchical order of functions building onto each other and its recursive nature strengthened by the implementation of loops. Additionally, due to its iterative nature there are relatively few steps required to perform the algorithm even without importing related libraries.

However, the main limitation of Dijkstra's algorithm is that it only works when all of the weights assigned are positive. In this regard an alternative algorithm for finding shortest path of all other nodes with respect to a selected node is Bellman-Ford Algorithm[3]; although it is not as generalizable as Dijkstra's, Bellman-Ford is able to work also with negative weights. Another important limitation of the algorithm specifically applied to the problem of internet router systems is that for the algorithm to work the input must be the complete representation of the graph, including all vertices and weights in order. So, the assumption that every router has a full coordinate of all routers on the internet must be made. This

[3] (Myrouteonline.com. 2020.)

assumption in the real world does not usually hold up given its impossible that every router is connected and flow information to one another.

Dijkstra's algorithm can be generalized to any problem requiring the solution of shortest distance, for instance GPS systems, telephone networks, computer-routing algorithms, and game design. Just like labelling routers as nodes, weighted links among various interconnected networks utilizes Dijkstra's algorithm to find the shortest path. Also considering that algorithm doesn't require initial data for its progress, its easily implemented to any defined set.

References

Bradfields School of Computer Science. n.d. Shortest Path with Dijkstra's Algorithm. [online] Available at: <https://bradfieldcs.com/algos/graphs/dijkstras-algorithm/> [Accessed 14 February 2022].

Delfino, A., 2020. *cpython/heapq.py at main · python/cpython*. [online] GitHub. Available at: <https://github.com/python/cpython/blob/main/Lib/heapq.py#L130> [Accessed 13 February 2022].

Myrouteonline.com. 2020. What Is the Best Shortest Path Algorithm?. [online] Available at: <https://www.myrouteonline.com/blog/what-is-the-best-shortest-path-algorithm#:~:text=Similar%20to%20Dijkstra's%20algorithm%2C%20the,other%20nodes%20in%20the%20graph> [Accessed 14 February 2022].

Code Appendix

```
import time

# functions defined from the source code of heapq()

def my_heappush(heap, item):
    # Push item onto heap, to maintain the heap invariant
    heap.append(item)
    my_siftdown(heap, 0, len(heap) -1)


# 'heap' is a heap at all indices >= startpos except possibly for pos
# pos is the index of a leaf with a possibly out-of-order value.
def my_siftdown(heap, startpos, pos):
    newitem = heap[pos]
    #Follow path to the root, moving parents down until finding a place
'newitem' fits
    while pos> startpos:
        parentpos = (pos - 1) >> 1
        parent = heap[parentpos]
        if newitem < parent:
            heap[pos] = parent
            pos = parentpos
            continue
        break
    heap[pos] = newitem

def my_heappop(heap):
    # Pops the smallest item off the heap, to maintain heap invariant
    lastelt = heap.pop() #if heap is empty raise appropriate IndexError
    if heap:
        returnitem = heap[0]
        heap[0] = lastelt
        my_siftup(heap, 0)
        return returnitem
    return lastelt

def my_siftup(heap, pos):
    endpos = len(heap)
    startpos = pos
    newitem = heap[pos]
    # Smaller child is 'bubbled up' until hitting a leaf.
    childpos = 2*pos + 1 # leftmost child position
    while childpos < endpos:
         # Index of smaller child is set to childpos
        rightpos = childpos + 1
        if rightpos < endpos and not heap[childpos] < heap[rightpos]:
            childpos = rightpos
        # Move  smaller child up
        heap[pos] = heap[childpos]
        pos = childpos
        childpos = 2*pos + 1
    # Leaf at pos is empty now.
    # Insert and bubble up newitem by sifting its parents down
    heap[pos] = newitem
```

```
    my_siftdown(heap, startpos, pos)


# open-source code for Dijkstra's algorithm

def calculate_distances(graph, starting_vertex):
    distances = {vertex: float('infinity') for vertex in graph} #unkown
distance
    distances[starting_vertex] = 0

    c=0   # iteration starts

# a customized function 'pq' is created

    pq = [(0, starting_vertex)]
    while len(pq) > 0:
        current_distance, current_vertex = my_heappop(pq)

        # Nodes can get added to the priority queue multiple times.
        # if a vertex is processed the first time we remove it from the
priority queue.
        if current_distance > distances[current_vertex]:     # ensures a
vertex is processed only once
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight  #ensures the distances
build up
            print(f"Neighbor: {neighbor}, weight: {weight}, distance:
{distance}")
 # Only consider this new path if it's better than any path we've
            # already found.
            if distance < distances[neighbor]:  #consider all neighbor
vetrices
                distances[neighbor] = distance
                my_heappush(pq, (distance, neighbor))
                c=c+1  # iteration ends
    print(c)   # prints how many iterations are done given a sample size

    return distances


example_graph = {   # for n=6
    'U': {'V': 2, 'W': 5, 'X': 1},
    'V': {'U': 2, 'X': 2, 'W': 3},
    'W': {'V': 3, 'U': 5, 'X': 3, 'Y': 1, 'Z': 5},
    'X': {'U': 1, 'V': 2, 'W': 3, 'Y': 1},
    'Y': {'X': 1, 'W': 1, 'Z': 1},
    'Z': {'W': 5, 'Y': 1},
}



start = time.time()
print(calculate_distances(example_graph, 'X'))
end = time.time()
print(end-start)      # calculate total running time
```

```
example_graph_2 = {       # for n=8
    'U': {'V': 2, 'W': 5, 'X': 1},
    'V': {'U': 2, 'X': 2, 'W': 3, 'K': 5},
    'W': {'V': 3, 'U': 5, 'X': 3, 'Y': 1, 'Z': 5, 'K': 1},
    'X': {'U': 1, 'V': 2, 'W': 3, 'Y': 1, },
    'Y': {'X': 1, 'W': 1, 'Z': 1, 'M': 1},
    'Z': {'W': 5, 'Y': 1, 'M': 3},
    'K': {'V': 5, 'W': 1},
    'M': {'Y': 1, 'Z': 3},
}



start = time.time()
print(calculate_distances(example_graph_2, 'X'))
end = time.time()
print(end-start)
example_graph_3 = {     # for n=10
    'U': {'V': 2, 'W': 5, 'X': 1},
    'V': {'U': 2, 'X': 2, 'W': 3, 'K': 5},
    'W': {'V': 3, 'U': 5, 'X': 3, 'Y': 1, 'Z': 5, 'K': 1, 'O': 2},
    'X': {'U': 1, 'V': 2, 'W': 3, 'Y': 1, },
    'Y': {'X': 1, 'W': 1, 'Z': 1, 'M': 1, 'N': 5},
    'Z': {'W': 5, 'Y': 1, 'M': 3},
    'K': {'V': 5, 'W': 1, 'O': 4 },
    'M': {'Y': 1, 'Z': 3, 'N': 3},
    'N': {'Y': 5, 'M': 3},
    'O': {'K': 4, 'W': 2},

}
start = time.time()
print(calculate_distances(example_graph_3, 'X'))
end = time.time()
print(end-start)



example_graph_4 = {    # for n=14
    'U': {'V': 2, 'W': 5, 'X': 1, 'P': 7},
    'V': {'U': 2, 'X': 2, 'W': 3, 'K': 5},
    'W': {'V': 3, 'U': 5, 'X': 3, 'Y': 1, 'Z': 5, 'K': 1, 'O': 2},
    'X': {'U': 1, 'V': 2, 'W': 3, 'Y': 1, },
    'Y': {'X': 1, 'W': 1, 'Z': 1, 'M': 1, 'N': 5},
    'Z': {'W': 5, 'Y': 1, 'M': 3, 'S': 4, 'R': 5},
    'K': {'V': 5, 'W': 1, 'O': 4 },
    'M': {'Y': 1, 'Z': 3, 'N': 3, 'R': 4, 'Q': 3},
    'N': {'Y': 5, 'M': 3},
    'O': {'K': 4, 'W': 2, 'P': 1, 'S': 2},
    'P': {'U': 7, 'O': 1},
    'S': {'O': 2, 'Z': 4, 'R': 3},
    'R': {'S': 3, 'Z': 5, 'M': 4, 'Q': 2},
    'Q': {'R': 2, 'M': 3},
}
start = time.time()
print(calculate_distances(example_graph_4, 'X'))
end = time.time()
print(end-start)
```

```
import matplotlib.pyplot as plt    # plot relationship between sample size
and total running time
#n is the number of vertices
n=[6, 8, 10, 14]
times= [0.0005068778991699219, 0.0006279945373535156,
0.0009620189666748047, 0.0031957626342773438]
plt.plot(n,times, 'r')
plt.yscale('log')
plt.show()
```