Dijkstra's Algorithm

Caleb Fernandes
Jadon Orr
Luke Shaw

Introduction

Dijkstra's Algorithm is a path finding algorithm that seeks to find the shortest path between one vertex and another in an undirected weighted graph [1]. An undirected graph is a graph in which the edges do not have a specific direction from one node to another (ie. the edges are bidirectional). A weighted graph is one such that the edges have weights associated with them that affect the traversal of the graph. In this case, the weights are the distances between the vertices of the graph [4]. The summation of the weights along one path is the distance it takes to travel from one node to another. The shortest path, found by Dijkstra's Algorithm, is the minimization of the summation of the edge weights between the start vertex and end vertex. Furthermore, the algorithm can be extended to return the shortest path itself along with the distance.

The algorithm utilizes a priority queue, vectors, and a graph structure. There are multiple variations of the algorithm as a priority queue can be substituted for an array-based queue, heap priority queue, fibonacci priority queue, or the set and the adjacency edges for a graph can be instantiated differently: adjacency list, adjacency matrix, incidence matrix etc. Our implementation uses an array-based priority queue and a pseudo-adjacency list. This list is considered pseudo as it is not implemented using a hashmap.

Main Body

Initialization of Algorithm

We begin the algorithm by creating the distance, parent vectors, and initializing the distance vector by inserting the pairs (V, ULONG_MAX) where V is an element of the vertice list. The parent vector is then initialized with the pair (StartVertex, "\0"). This can be interpreted

as (Child, Parent) where the first vertex in the pair comes from the second vertex. Thus, when the code finishes, it is easy to walk backwards and recreate the shortest path. Finally, the pair (StartVertex, 0) is inserted into the queue. This represents (Vertex, distance) where distance is the shortest amount of distance it takes to get to Vertex. As such, since you begin at StartVertex, the shortest distance to StartVertex is 0.

Core of the Algorithm

The core of the algorithm begins by looping until the priority queue is empty. Each iteration will take an edge in the adjacency list attributed to the vertex popped from the queue. In an iteration, the weight associated with the edge is added to the distance pair in the queue pair (remember, (Vertex, Distance) are the pairs in the queue where distance is the shortest distance to get to the Vertex) and stored in a temporary variable called total_distance. If this total distance is shorter than the distance in the (V, Distance) pair where V is equal to the neighbor vertex, then the distance and parent vectors are updated such that the new total distance replaces the old distance feature in the neighbors (V, Distance) pair in the distance vector and the parent of the neighbor vertex is updated to be the vertex from the vertex list that is currently being iterated on. Then, the (V, Distance) where V is the neighbor vertex is added to the queue. If the total distance is not lower than the associative element in the distance vector then the previously described sequence will not be performed and the loop will continue. This, again, will continue until there are no elements left in the queue. The result will be a distance vector that contains all the vertices in the graph and the shortest distances to get to them from the start vertex. The parent vertex will contain the vertices that they came from in order to create the shortest path [2].

Path Reconstruction and Final Distance

The final two steps are to find the total distance of the shortest path and to recreate this path. To recreate the path, a loop is created such that it ends when the element "\0" is hit. It initiates a variable current as the EndLabel, finds the end label in the parent vector, stores it in the path vector, then sets the current label to the second feature in the pair. The total distance is found easily as it is stored in the distance vector in the pair associated with the end label. Thus, the algorithm will return the second feature in the element in the distance vector pertaining to the end label. Now, the algorithm terminates [3].

Conclusions and Summary of Result

The algorithm was then tested on two test case graphs. The first test case was a simpler to interpret case as the vertices were numbered 1 through 6. The edges between the nodes were then implemented and the test case ran. It correctly outputs the answer for the shortest distance between nodes 1 and 5 as a distance of 20 and returns the correct path of 1 → 3 → 6 → 5. The code was further tested by removing a vertex and then an edge. First, the vertex 3 was removed and the correct result was output: a length of 23 with the shortest path being 1 → 6 → 5. Then, the edge ('5', '6', 9) was removed, the resulting output was again correct: a length of 28 with the shortest path being 1 → 2 → 4 → 5. The code was then tested on a more realistic scenario in which the vertices were labeled as building numbers at USF such as the "BEH" building. The distances between the buildings were then input as the edge weights and the algorithm correctly discerned the shortest path between the ENB and SUN buildings as 2558 with the shortest path being from ENB → SUN and the result of the output on LIB and CAS building was 1532 with the path LIB → ENB → CAS.

In conclusion, Dijktstra's algorithm proves to be a very elegant way of traversing a graph in order to find the shortest path between two vertices. It is also a good ending project as it combines various data structures and algorithms such as a priority queue, vector, graph, hashmap, and a variation to breadth first search. Breadth-first search algorithms are often used for finding the shortest path between two vertices in an unweighted graph. This algorithm shows how to modify it and its concepts for a weighted graph. Also, by implementing a second vector and traversing it, we can deconstruct the shortest path in terms of the vertices rather than just return a number. The complexity of our code, however, is uncertain as it uses an array-based priority queue which would result in a complexity of $O(|V|^2)$, however, we did not implement a hash map for access times for traversing through the vertex list and edges. Thus, the complexity of our algorithm is most likely larger than $O(|V|^2)$ as we have to access the individual vertices in the vertex list, then the edges of each edge list and then add them to the priority queue. Iterating through the vertex list would take $O(|V|)$ while iterating through all the edges would take $O(|V|)$ again. Thus, the most likely estimate for the complexity of our code is $O(|V|^3)$.

Surprising Discoveries:

The most important thing learned during this process is that the functionality would have been a lot faster and easier to implement had we used a hashmap instead of iterating through a vector to find the vertices everytime [1]. If we had a hash map, then it would have been much easier to find vertices and the code would have been much simpler. This would be an actual adjacency list as opposed to the pseudo-adjacency list that we created. Furthermore, using pointers to the vertices would have made this much easier, especially when traversing. The benefit, however, of not using pointers and creating new memory spaces is that we did not have

to create a constructor or destructor for graphs. Since the graph class did not allocate any new memory, the only allocated memory was in the vectors which automatically deconstruct. Thus, there was no need to involve a deconstructor past a basic declaration. In all, the traversal of the vertices amongst the queue, distance, and parent vectors was the most complex part of the code as it was difficult to keep track of what was going on. Therefore, proper variable naming was important in this code as to not get confused as to what elements were being modified or accessed.

Bibliography

References:

[1] *Dijkstra's algorithm* (2025) *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Dijkstra's_algorithm (Accessed: 18 April 2025).

[2] take U forward (no date) *G-32. Dijkstra's Algorithm - Using Priority Queue - C++ and Java - Part 1*, *YouTube*. Available at: https://www.youtube.com/watch?v=V6H1qAeB-l4&list=WL&index=14 (Accessed: 18 April 2025).

[3] take U forward (no date a) *G-35. Print Shortest Path - Dijkstra's Algorithm* , *YouTube*. Available at: https://www.youtube.com/watch?v=rp1SMw7HSO8&list=WL&index=15 (Accessed: 18 April 2025).

[4] *Introduction to graph data structure* (2024), *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/ (Accessed: 18 April 2025).