Assignment 1: Bidirectional Dijkstra Algorithm

Shane McCarthy | 14200512 | shane.mc-carthy@ucdconnect.ie

Introduction

Many real world applications of network theory involve weighted networks in which the edges have weights or strengths representing, for instance, traffic on a road network, funds being transferred between accounts or social media likes between two friends. Therefore the shortest path (SP) cannot be determined using standard breadth-first search which would simply return the path with the minimum number of edges, hence the motivation for Dijkstra's algorithm.

Dijkstra's algorithm

Dijkstra's algorithm finds the shortest distance from a given source node to every other node within a component of a network, however unlike breadth-first search, it does so by factoring the edge weights. The algorithm works by recording the shortest distance it has found so far to each node and updating that record if a shorter one is found, once complete it can be shown that the shortest distance found is the shortest distance possible via any path. The time complexity of this algorithm depends on how it is implemented (Newman, 2010), the most basic implementation takes $O(n^2)$ in the worst case. If we implement the algorithm with a priority queue which uses a binary heap to store distance estimates, removing the smallest estimate in time $O(\log n)$, the efficiency of the algorithm improves dramatically to $O(\mathrm{nlog}\, n)$, this is close to $O(n)$ for the equivalent problem on a network that is unweighted.

Bidirectional Dijkstra Algorithm

The Bidirectional implementation of Dijkstra's algorithm assumes a single source and target node are known, such uses case include routing problems travelling from location A to B. In detail the algorithm follows a divide-and-conquer frontier search approach (Devadas, 2017), alternating between the forward frontier (traversing from the source *s*) and the backward frontier (traversing backwards from the target *t*) iteratively, on each iteration it;

1. *Finds the node v that has the smallest estimated distance from s/t. Marking this as certain.*
2. *Calculate the distance from s/t via v to each of the neighbours of v. If any of these distances are less than the current estimates to the same neighbour the new distance replaces the old one.*

The alternating search continues while the two frontiers don't overlap or more specifically the algorithm terminates when some node w (known as the link node) has been processed by both queues. Intuitively one would think that the link node *w* must sit on the SP as it is the point where both frontiers meet, however this is not always the case. The shortest source/target distance for all link nodes must be evaluated to determine the final SP. Figure 1 below illustrates the scenario where the link node that causes the algorithm to terminate does not sit on the SP between source and target.

Contrasting Dijkstra vs Bidirectional Dijkstra

Figure 1 below illustrates how both Dijkstra and Bidirectional Dijkstra determine the SP between source node 25 and target node 26. Dijkstra uses a brute-force approach and determines the SP to every other node in the network, essentially all nodes are visited. Bidirectional Dijkstra divides the problem into two and starts solving from the source and target respectively. Observe the groupings of blue and green nodes which form the Forward and Backward frontiers, on a larger graph these grouping would appear circular. Node 2 causes the alternating search to terminate before the nodes colored black could be visited and the SP is found through node 0. The important difference

between Dijkstra and Bidirectional Dijkstra illustrated in this toy example are the 19 nodes (of 40 in total) which were not visited, this reduces the complexity seen earlier from $O(\mathrm{nlog}\,n)$ to $\sim O(\frac{1}{2}\mathrm{nlog}\frac{1}{2}n)$.

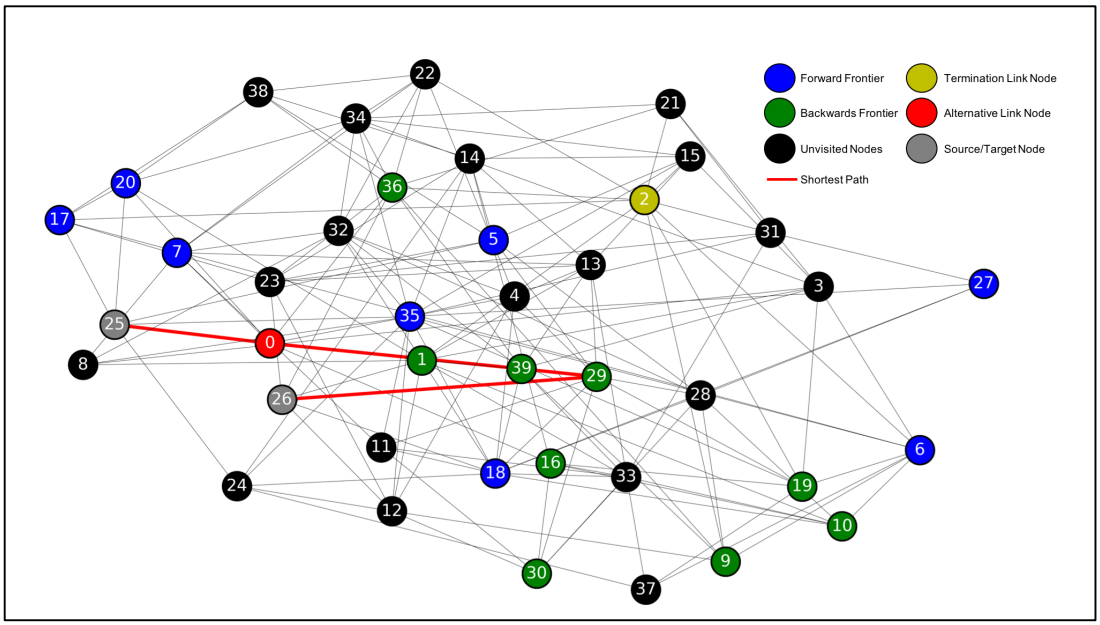

Figure 1 Bidirectional Dijkstra vs Dijkstra

This does not change the theoretical worst-case performance of Dijkstra however it does reduce the number of visited nodes in practice, depending on the graph structure. The underlying graph structure and scale dictates the number of nodes that can go unvisited, for example the gains resulting from Bidirectional Dijkstra would be far greater if the graph of interest had a Scale-free structure compared to a Random structure or if the graph of interest was sufficiently large. Figure 2 below illustrates the run-time behaviour of Dijkstra and Bidirectional Dijkstra over random graphs of varying sizes, the observed scaling behaviour is as expected.

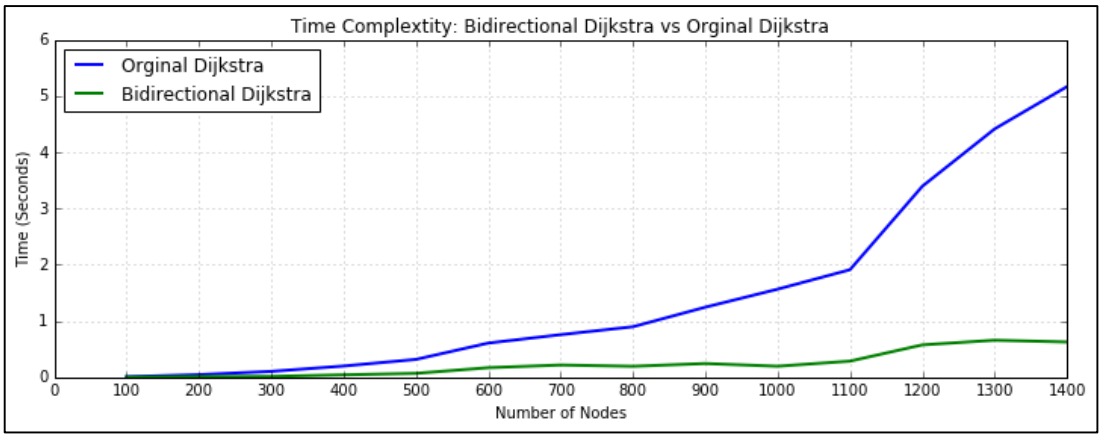

Figure 2 Time Complexity: Bidirectional Dijkstra vs Dijkstra (see appendix for raw data)

Conclusion

It has been demonstrated that a number of relatively minor modifications to Dijkstra's algorithm can result is dramatic performance efficiencies. A number of key learning outcomes were achieved through this assignment including self-instructed learning.

Bibliography

Devadas, S., 2017. *ocw.mit.edu.* [Online]
Available at: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-18-speeding-up-dijkstra/#p3s:254930&p3e:279160
[Accessed 2017].

Newman, M., 2010. *Networks: an introduction.* 2nd ed. New York: Oxford University Press Inc..

Appendix

Table 1 Time Complexity: Bidirectional Dijkstra vs Dijkstra

Graph Size (Nodes)	Average run-time for Dijkstra (seconds)	Average run-time for bidirectional Dijkstra (seconds)
100	0.00817	0.00231
200	0.04029	0.01361
300	0.10359	0.03164
400	0.25600	0.04000
500	0.31027	0.04388
600	0.65222	0.16165
700	0.75099	0.14284
800	0.85692	0.13364
900	1.15981	0.14249
1,000	1.65203	0.15744
1,100	2.22370	0.17000
1,200	2.53380	0.48062
1,300	3.14782	0.39907
1,400	3.39927	0.34461