UCD Smurfit Graduate School of Business

Network Software Modelling

Assignment One - Dijkstra's Algorithm

Author:
Conor Reid
(16202630 - MSc B.A Part-time)

Lecturer:
Dr. James McDermott

March 12, 2017

Dijkstra's Algorithm

Dijkstra's Algorithm is used for finding the shortest path (that of lowest cost) between a initial point in a directed graph, and all others. Formally; Given an initial vertex s in a weighted directed graph $G = (V, E)$ where edges are non-negative, Dijkstra's Algorithm finds the shortest path from s to all other vertices in G.

The algorithm works as follows; Every node in a graph is assigned a tentative distance value, which is zero for the initial node and infinity otherwise. The initial node is marked the current node and thus visited, all others are marked as unvisited and are passed into a set. Visited nodes are not visited again. The neighbours of the current node are calculated for actual tentative distance. If this distance is lower than the previously assigned tentative distance to this node then it overwrites the same. The process is repeated, assigning the next unvisited node with the lowest tentative distance as the current node. The algorithm stops when the unvisited set is empty.

Dijkstra's algorithm has time complexity $O(n^2)$ where n is the number of nodes in the graph.

The Bidirectional Variant

The bi-directional version of Dijkstra's algorithm can be used in an effort to *speed up* the searching process, by reducing the number of visited vertices. Normally, the searching process happens in the direction of s (the initial node) to t (the target node). In this version, searching direction alternates between forward and backwards. That is, we alternate between searching from s to t and from t to s, the latter involving following edges backwards. We denote distances for forward search as $d_f(u)$ and distances for backward search as $d_b(u)$. The algorithm terminates when some vertex has been removed from the queues of unvisited nodes of both searches (i.e we have reached the same node coming from each direction and thus we have a path) [1].

A Comparison

A graphic comparison of both versions is shown below, which illustrates a *bubbles* analogy for how the algorithm work. The left image is a representation of the original algorithm, which starts at s and finds each t [1] by growing a bubble centred at s. Conversely, the image in the right represents the bi-directional variant, which grows two bubbles centred at each of s and t and terminates when they meet. In theory, the total area of the bubbles in the second case is less, making it quicker.

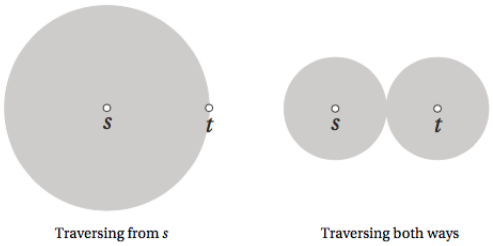

Figure 1: Dijkstra's Algorithms [2]

Time Complexity and Efficiency

Interestingly, the bi-directional algorithm does not improve the worst case behaviour of the original algorithm (big 0 notation). Bi-directional searching still has time complexity of $O(n^2)$ (which is really $O(\frac{n^2}{2})$ which is $\sim O(n^2)$). Having said this, this variant is still known to be more efficient. In the majority of cases it leads to significantly less path calculations since it (in theory) considers less nodes totally (see diagram above). Thus the Dijkstra bi-directional algorithm is known as a heuristic algorithm, since improvements are expected but not guaranteed.

Implementation

The source code written for this submission can be found in the `code/` directory. The below is a high level overview of implementation and design decisions. For more detail see the source code comments.

First, `priority_queue_dijkstra.py` contains code that was provided prior to beginning this assignment, that computes a list of shortest paths between some initial node and all others in a given graph.

[1] In the original algorithm, this could be a target node which when found, terminates the algorithm. One can also provide no target and find the shortest path to all other nodes from s

An addition has been made, to return a generator as an entry of the *results{}* dictionary. This generator will be used later in the bi-directional variant of Dijkstra's algorithm.

The bi-directional variant is found in `bidirectional_dijkstra.py`. This algorithm is designed to call the normal dijkstra algorithm twice; once in the forward direction moving from the initial node to the target node, and again for the reverse direction. This is why the above algorithm was modified to return a generator, so that the bi-directional algorithm can stop when we meet a common node (between forward and backwards direction) without having to compute distances beyond this common node.

A third party implementation of the Dijkstra algorithm has also been sourced and modified (see `code/tools/referencedCode.py`), since the provided implementation was shown to be slow to return a generator of results [2]. This second version was shown to be much quicker (see results below) benefiting from much lower overheads and dramatically improved the runtime of the bi-directional algorithm.

Run-Time behavior

Code written to test the implementations of the above are contained in `code/dijkstraTest.py`. This file contains functions that call; original Dijkstra, the bi-directional variant, Dijkstra as per the `networkx` package, and the bi-directional variant as per the same. The graphs used are randomly generated, using a custom written python module, at `code/tools/genGraphs.py`. This creates a number of complete graphs with node counts between 1 and 1352 [2], generated before testing begins and each used for one test iteration (i.e. one record in the table below).

The runtime behaviour of this program depends on which implementation of Dijkstra is being used (provided or third party, both modified). Testing was carried out using both, and summarised below in separate tables. Console output is available in `notes/` for test instances not shown here.

Provided Dijkstra - Own Bi-Directional Dijkstra.				
Graph Size(node - edge)	Dijkstra	Dijkstra(nx)	Bi-Directional Dijkstra	Bi-Directional Dijkstra(nx)
91 - 4095	0.011757	0.00061	0.022578	0.000812
200 - 19900	0.060883	0.002057	0.472891	0.006331
488 - 118828	1.065864	0.011387	1.448897	0.017168
779 - 303031	2.456892	0.030297	4.555177	0.015358
1219 - 742371	5.438428	0.073147	10.093776	0.012997

The table above refers to the use of the provided (modified) Dijkstra's algorithm, and the bi-direction variant for a number of different sized graphs. Counterintuitively, the original algorithm took less time to run than the variant in all cases, even though in theory it should be computing more nodes and thus be slower. Indeed, the `networkx` algorithms support this theory, and are significantly quicker than the offered solution.

3^{rd} Party Dijkstra - Own Bi-Directional Dijkstra.				
Graph Size(node - edge)	Dijkstra	Dijkstra(nx)	Bi-Directional Dijkstra	Bi-Directional Dijkstra(nx)
91 - 4095	4e-06	0.000447	0.001603	0.00315
200 - 19900	4e-06	0.001895	0.008461	0.006511
488 - 118828	7e-06	0.012082	0.009251	0.006266
779 - 303031	5e-06	0.032724	0.013184	0.020178
1219 - 742371	6e-06	0.079735	0.013706	0.009067

The alternative Dijkstra algorithm is used in generating the above, and it is clear that it yields significantly improved runtime results for the variant, bringing it very close to the `networkx` implementation. Runtime is also less than for the `networkx` Dijkstra algorithm, as expected.

Conclusions

Algorithms that computed shortest path were successfully built, using Dijkstra's algorithm and the bi-directional variant. While output results were shown to be correct, runtime performance was shown to suffer when using Dijkstra's as implemented using a priority queue. By using an alternative that utilises `heapq` and generators, performance close to `networkx` algorithms was achieved.

[2] Each node's name is a combination of an upper case letter, a lower case letter and a number(either a 1 or 2). This amounts to 26 * 26 * 2 = 1352 combinations.

References

[1] https://courses.csail.mit.edu/6.006/fall11/lectures/lecture18.pdf

[2] Hetland, M.L., 2014. Python Algorithms: Mastering basic algorithms in the Python Language. Apress.