Dijkstra's Python Implementations: with and without decrease-key

Nicola Amadio

AY 2019/2020

Abstract

The following is the report of the project work for the course in Combinatorial Optimization, MSc Computer Science, University of Milan. It is about Dijkstra's algorithm, and two of its possible implementations, both based on a Priority Queue realized through a Binary Heap. We deal with the "tree version" of the problem, i.e. the one where we want the shortest path from a source node to all the other nodes; we didn't store the actual paths but only their lengths, in a so-called "distance vector" (it's trivial to change the code to make it store the paths as well). The first version uses a decrease-key function that updates the distance vector if a smaller path to a vertex is found, while in the second we add the new and more efficient distances to the heap, without updating the ones already in the heap (possibly generating multiple nodes in the heap for the same vertex). We chose these two versions of the algorithm because they manage to achieve quasi-optimal asymptotical performance without being complicated to implement, therefore being a very good resource to use in practice. The community has generally agreed (through experimental results) that the fastest solution in practice is the one that doesn't use the decrease-key [1]. We started out expecting to achieve the same results, but ended up verifying that in our case the version with the decrease-key actually achieved better performance. This is due to the fact that we used Python in our implementation, which is not predictable when accessing lists' (python vectors) values: in fact this makes it not a good language for benchmarking algorithms. At the same time, it can happen that eventually we want to implement Dijkstra's algorithm in Python, and in that case the results of this report could be useful. It is worth pointing out that our implementation doesn't make use of any library function: we built the heap ourselves and didn't use the heapq Python API. The rest of the report is structured like this: in the first section we introduce the Shortest Path Problem and Dijkstra's algorithm; afterwards we dive into the two different implementations and finally we discuss our experimental results and conclude with some final considerations.

1 Shortest Path Problem

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. There are some variation of this problem, like the all-pairs shortest path, or the single-source shortest path - in which we have to find shortest paths from a source vertex v to all the other vertices in the graph (and which is the one we addressed in this work).

Shortest path algorithms are applied to many different domains, making it a very important problem in Computer Science. Examples include:

- Automatically finding directions between physical locations, such as driving directions on web mapping websites like Google Maps.
- Other applications, often studied in operations research, include plant and facility layout, robotics, transportation, and VLSI design.

The most important algorithms for solving the shortest path problem and its variations are:

a) Dijkstra's algorithm for solving the single-source shortest path problem with non-negative edge weight.

b) Bellman–Ford algorithm for solving the single-source problem if edge weights may be negative.

c) A* search algorithm for solving the single pair shortest path using heuristics to try to speed up the search.

d) Floyd–Warshall algorithm for solving all pairs shortest paths.

e) Johnson's algorithm for solving all pairs shortest paths, and may be faster than Floyd–Warshall on sparse graphs.

f) Viterbi algorithm for solving the shortest stochastic path problem with an additional probabilistic weight on each node.

2 Dijkstra's Algorithm

Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm) is an algorithm for finding the shortest paths between nodes in a graph. The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes [2], but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all the other nodes in the graph, producing a shortest-path tree. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.

Dijkstra's algorithm uses a data structure for storing and querying partial solutions sorted by distance from the start. The original algorithm uses a min-priority queue and runs in time $\mathcal{O}(|V|^2)$ (where $|V|$ is the number of nodes). Fredman & Tarjan in 1984 [3] proposed using a Fibonacci heap min-priority queue to optimize the running time complexity to $\mathcal{O}(|E| + |V|log|V|))$(where $|E|$ is the number of edges). This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights. The versions that we used both have a complexity of $\mathcal{O}(|V| + |E|log|V|)$.

2.1 Pseudocode

Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.

Algorithm 1 Dijkstra

```
create vertex set Q
for all vertex v in Graph do
    dist[v] ← INFINITY
    add v to Q
end for
dist[source] ← 0
while Q is not empty do
    u ← vertex in Q with min dist[u]
    remove u from Q
    for all neighbor v of u that are still in Q do
        if dist[u] + length(u, v) < dist[v] then
            dist[v] ← dist[u] + length(u, v)
        end if
    end for
end while
return dist
```

3 Implementations

The above is a generic, high-level pseudocode implementation of Dijkstra's algorithm. The versions that we chose for our tests use a Binary Heap data structure to implement the Priority Queue that stores the nodes that can still be visited along with their keys, which in Dijkstra's case are the distances of the vertices from source: for the heap property, the root node will be the closest to source.

The main difference between the two versions is that one uses a decrease-key function that actually updates the vertices' keys in case shorter paths from source are discovered, while the other just adds the vertices again in the heap if shorter paths are found, without decreasing the keys of the ones already in the heap. We describe the functions insert and decrease-key more in-depth:

- Q.insert adds a node, containing the vertex we are adding along with its distance from source, to the heap, placing it in the last position of the array implementing the heap. It then performs the so called "sift up" operation: the node's key is recursively confronted with that of its parent and in case it's smaller we swap the two nodes. This places the node in the right place of the heap.

- Q.decrease-key can be used if the vertex we want to add is already in the heap, only with a bigger key (which again in our case represents the distance from source). It works by looking up in a table the position of the vertex in the heap, updating its key with the new, smaller distance and "sifting up" the node to the right place.

The following is the actual Python code for the two Dijkstra's implementations. We point out that the PiorityQueue class that we are using was built by us from the ground up for educational purposes. We also tried using the official Python implementation for the Min-Heap (heapq [5]), which actually performs much better (see next section for actual numbers) since it is internally written in C. All our code, along with the following algorithms, was open sourced and it's available at [7].

```
def dijkstra_dec(self, src):
        Q = PriorityQueue()
        self.dist = [sys.maxsize] * (self.num_nodes + 1)
        self.dist[src] = 0
        Q.insert(src, 0)
        while Q.heap:
            heap_node = Q.extract_min()
            u = heap_node[0]
            self.dist[u] = heap_node[1]
            if u not in self.adjList.keys():
                continue
            for v, w in self.adjList[u]:
                if self.dist[u] + w < self.dist[v]:
                    if self.dist[v] == sys.maxsize:
                        Q.insert(v, self.dist[u] + w)
                    else:
                        Q.decrease_key(v, self.dist[u] + w)
                    self.dist[v] = self.dist[u] + w
```

```
def dijkstra_no_dec(self, src):
    Q = PriorityQueue()
    self.dist = [sys.maxsize] * (self.num_nodes + 1)
    self.dist[src] = 0
    Q.insert(src, 0)
    while Q.heap:
        heap_node = Q.extract_min()
        if heap_node[1] > self.dist[heap_node[0]]:
            continue
        u = heap_node[0]
        self.dist[u] = heap_node[1]
        if u not in self.adjList.keys():
            continue
        for v, w in self.adjList[u]:
            if self.dist[u] + w < self.dist[v]:
                Q.insert(v, self.dist[u] + w)
                self.dist[v] = self.dist[u] + w
```

4 Experimental Results

4.1 With our PriorityQueue

We tested our two algorithms over 14 directed graphs, with different density and size: the first 12 were randomly generated via the Randgraph script [4], while the last two represent the roads of NY and of Western USA respectively [4]. For each of these graphs, we ran Dijkstra's algorithm with 4 different source vertices, and then averaged the results we found on the 4 different runs. We then measured the time spent by each version of the algorithm and compared them to each other for the different experimental settings. We found that the only case in which the no-decrease-key version would beat the other was for very small (<1000 vertices) graphs. In all the other cases, the decrease-key version performed better, requiring 60 to 90 percent of the time spent by the other version. This result differs from the ones previously published by teams like [1]. This is mostly due to the fact that we used Python, which has a peculiar way of treating lists, and might lose time in ways for us difficult to understand. A possible explanation could be that modifying the heap-list is expensive in Python, giving the edge to the decrease-key version which rarely adds/removes new elements.

We can conclude that among our implementations, which follow a pretty standard way of coding Dijkstra's algorithm, the one using the decrease-key function would be a better choice in most cases. It's worth pointing out that changing some little detail in the implementation could alter the results, but it's also worth noting that, while there are ways to *potentially* improve the decrease-key version (for example using Pairing or Fibonacci Heaps), there aren't really many ways to improve the no-decrease-key version (the one that performed

worse in our experiments): an option would be to instantiate a very big heap (increasing the preprocessing time) and copying there the elements to be inserted, instead of actually inserting them by adding new nodes.

Performance of Dijstra's algorithm in Python				
Nr. of Vertices	decrease-key	no-decrease-key	time ratio	avg-deg
10	0.019	0.015	1.655	3
1000	0.111	0.106	1.053	3
100'000	0.176	0.216	0.651	3
10'000'000	38.09	42.92	0.764	3
13'500'000	52.28	58.54	0.774	3
1000	10.93	13.93	0.790	8
100'000	2504	3162	0.792	8
1'000'000	36874	46487	0.786	8
10'000'000	261070	268313	0.856	4
264'346	2552	2849	0.776	5.5
6'262'104	73510	77445	0.841	4.8

In the above figure we can see the results of the tests we conducted: the columns $w/\ decrease-key$ and $w/o\ decrease-key$ display the time required by Dijkstra's algorithm in milliseconds, while the time ratio was computed dividing the time required by the decrease-key version by the one required by the no-decrease-key one; the avg-deg column displays the average degree of the considered graph (i.e. $2 * |E|/|V|$).

4.2 Using heapq

We also tried using Python's heapq [5] class together with the no-decrease-key version, and we managed to get better results compared to the decrease-key algorithm (implemented with our PriorityQueue), requiring 80 percent of the time for sparse graph and as little as 10 percent for dense graph.

The heapq [5] class doesn't implement the decrease-key version, so we weren't able to test it also with that version of the algorithm. But there are ways to implement it, and it would be interesting to test the performance in that case, in order to find the most performing version of Dijkstra written in Python (even if in that case we would be using some internals written in C, due to the heapq [5] characteristics). It could be also worth it to try testing it with the HeapDict [6] class, which implements a performant heap containing a decrease-key method.

5 Final Considerations

The shortest path problem is of enormous importance in Computer Science and Dijkstra's algorithm is one of the most famous algorithms for graph problems in general, if not the most famous one. The relevance goes beyond theory, since, as we stated in the introduction, a lot of possible applications exist in this context. It's worth noting that in some practical domains other solutions might be preferred to Dijkstra's. For example, for the routing problem (i.e. Google Maps and navigation systems in general) the literature has moved to more engineered solutions that operate an initial preprocessing of the graph, in order to make successive operations on it faster: that's the line of worked followed by [8] [9] and [10] on Construction Hierarchies and related topics. Another case worth mentioning is the one related to the experimental results of the SPFA algorithm, an extension of the Bellman–Ford algorithm, which, although theoretically slower ($\mathcal{O}(|E||V|)$), is in most practical cases faster than all Dijkstra's implementations. That said, an infinite number of problems can be modeled as graph problems, and in many cases it would be worth it to consider Dijkstra as a possible solution.

References

[1] Mo Chen, Rezaul Alam Chowdhury, Vijaya Ramachandran, David Lan Roche, Lingling Tong *Priority Queues and Dijkstra's Algorithm.* In UTCS Technical Report TR-07-54, 2007.

[2] E. W. Dijkstra *A Note on Two Problems in Connexion with Graphs.* In Numerische Mathematik, 1959.

[3] M.L. Fredman, R.E. Tarjan *Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms.* In 25th Annual Symposium onFoundations of Computer Science, 1984.

[4] http://users.diag.uniroma1.it/challenge9/download.shtml

[5] https://docs.python.org/3.7/library/heapq.html

[6] https://pypi.org/project/HeapDict/

[7] https://github.com/amadionix/dijkstra-python

[8] Robert Geisberger, Peter Sanders, Dominik Schultes, Daniel Delling *Contraction hierarchies: Faster and simpler hierarchical routing in road networks.* In International Workshop on Experimental and Efficient Algorithms, 2008.

[9] Daniel Delling, Peter Sanders, Dominik Schultes, Dorothea Wagner *Engineering route planning algorithms.* In Algorithmics of large and complex networks, 2009.

[10] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, Renato F Werneck *Route planning in transportation networks*. In Algorithm Engineering, 2016.