Bidirectional Dijkstra

May 30, 2020

Dijkstra's algorithm computes lengths of shortest paths from a start vertex s to every other vertex in a weighted graph with nonnegative weights. It works by successively improving an approximation $d[v]$ to the shortest path length $\delta(s, v)$ from s to v, which is initially $d[s] = 0$ and $d[v] = \infty$ for $v \neq s$. The algorithm maintains a priority queue Q of vertices which haven't yet been processed, initially containing just s with priority 0, and a set S of vertices whose true distance to s is known. It works as follows, in pseudo-Python:

```
while Q is not empty:
    u = extract_min(Q)
    S.add(u)
    for v in adj(u):
        if v not in S and d[v] > d[u] + w(u, v):
            d[v] = d[u] + w(u, v)
            Q.add(v, priority=d[v])
```

...where `adj(v)` is the vertices adjacent to `v` and `w(u, v)` is the weight of the edge $u \to v$. The function `extract_min` pops a vertex with minimum `d`-value out of Q. The procedure inside the conditional is called *relaxing* the edge `(u, v)`.

The key result in proving correctness is that once a vertex `u` enters `S`, its `d`-value `d[u]` is equal to the true shortest path distance $\delta(s, u)$ - this is CLRS Theorem 24.6.

Dijkstra runs in $O(V \log V + E \log V)$ if Q is a minheap, which can be improved slightly with fancier data structures. If all we care about is $\delta(s, t)$ for some fixed t we can speed it up for certain graphs as follows. Suppose each vertex has outvalency about m, and the edge distance from s to t is n. Then if the edge weights are roughly equal we expect to relax on the order of m^n edges by the time we reach t. On the other hand, if we run two searches, one starting at s and the other at t, stopping when they meet in the middle, we would only relax $2m^{n/2}$ edges. This method is called **bidirectional Dijkstra**.

The subtlety in bidirectional Dijkstra is the stopping condition. When we detect an edge between the forward and backward sets S_f, S_b of processed vertices there is no guarantee a shortest path $s \rightsquigarrow t$ passes through that edge. "Stop as soon as you find an edge (u, v)

between vertices processed in the forward and backward searches, return $d_f[u] + w(u, v) + d_b[v]$ or $d_f[v] + w(u, v) + d_b[u]$ as appropriate" is not correct.

(here d_f is the approximation to $\delta(s, -)$ computed by the forward search and d_b the approximation to $\delta(-, t)$ computed by the backward search).

The point of this post is to give a description of bidirectional Dijkstra that is precise enough to implement, then a sketch of why it is correct. This isn't in CLRS, for example, and while there are slides on the internet (e.g. here or here or these 2016 recitation notes from 6.006) the explanations have gaps which would have been filled in by the lecturer.

I'm going to assume that the graph is undirected; if it is directed you need to do the backward search on the opposite graph.

We start with forward approximations $d_f[v]$ of the distance of a node from s and backward approximations $d_b[v]$ of the distance of a node to t, initially all infinity except $d_f[s] = 0 = d_b[t]$. We keep a forward priority queue Q_f and a backward priority queue Q_b, initially containing s and t respectively, and sets S_f and S_b of vertices processed in the forward and backward searches, initially empty. The priority of an element in the queue is its d_f or d_b value. We also keep a number μ equal to the length of the shortest path $s \rightsquigarrow t$ yet seen, initially infinity.

```
while Qf is not empty and Qb is not empty:
    u = extract_min(Qf); v = extract_min(Qb)
    Sf.add(u); Sb.add(v)
    for x in adj(u):
        relax(u, x)
        if x in Sb and df[u] + w(u, x) + db[x] < mu:
            mu = df[u] + w(u, x) + db[x]
    for x in adj(v):
        relax(v, x)
        if x in Sf and db[v] + w(v, x) + df[x] < mu:
            mu = db[v] + w(v, x) + df[x]
    if df[u] + db[v] >= mu:
        break # mu is the true distance s-t
```

(Thanks to Panagiotis Karras, Ciaperoni Martino, Nassos Katsamanis, and Aristides Gionis who emailed me with a correction to this algorithm). You can find example Python code with tests in my Github repository.

"Relax" refers to the same procedure as in the ordinary Dijkstra algorithm, so that for example `relax(u, x)` would do the following:

```
if (x is not in Sf) and df[x] > df[u] + weight(u, x):
    df[x] = df[u] + weight(u, x)
    Qf.add(x, priority=df[x])
```

I claim that when this algorithm leaves the while loop, μ equals the true distance $\delta(s,t)$. If you want to recover the actual shortest path you need to maintain a vertex as well as a best path length, and when μ gets updated also update that vertex to x. The shortest path is then a shortest path $s \rightsquigarrow x$ followed by a shortest path $x \rightsquigarrow t$.

First we need a result about *ordinary* Dijkstra.

> *At any stage of the ordinary Dijkstra algorithm, if $y \notin S$ then $\delta(y,s) \geq \max_{x \in S} \delta(x,s)$.*

If not there is a first relaxation in the algorithm setting the d-value of a vertex $y \in Q$ to a value less than $d[x]$ for some $x \in S$. Say the edge (r,y) was the one being relaxed, so r was just added to S. $d[y]$ gets set to something larger than $\lambda = d[r]$, so $d[r] < d[y] < d[x]$.

If x was in S when $d[r]$ got set to λ, we have a contradiction to the relaxation above being the earliest setting a d-value of something in Q to something less than a d-value in S. So the sequence of events must have been:

1. $d[r]$ was set to λ
2. x entered S (before r did)
3. (r,y) was relaxed when r entered S. This is impossible: at step 2, x can't have had the smallest d-value outside S because $d[r]$ was already $\lambda < d[x]$.

Now we're ready to prove the correctness of the termination condition in bidirectional Dijkstra. Suppose that the algorithm terminated at the `break` but that there is a path $s \overset{P}{\rightsquigarrow} t$ beating μ. It can't contain a vertex x outside $S_f \cup S_b$: such a vertex is at least $d_f[u]$ from s and $d_b[v]$ from t by the ordinary Dijkstra result above, so P would have length at least μ. Thus P is contained in $S_f \cup S_b$ - but then it was considered when one of its edges was first found to connect the two sets of processed vertices, and μ was updated to something at most the length of P.

Matthew Towers' homepage

Matthew Towers' homepage
m.towers@ucl.ac.uk

matthew-towers

Hello world.