Execution time of the shortest path algorithm

To the right is the invariant of the shortest path algorithm, which calculates shortest paths and their distances from node v to all nodes. Below is the algorithm itself. Our task is to analyze the expected execution time of the algorithm.

Here are our assumptions about the graph:

1. The graph is directed.
2. n nodes are reachable from start node v.
3. The n reachable nodes have a total of e outgoing edges.
4. The graph is implemented using some form of adjacency list representation.

We assume that frontier set F is implemented as a min-heap, with the priority of a node f in it being d[f]. We assume that settled set S is implemented as a hash set[1], so most operations are O(1) expected time and O(n) worst-case time. Both F and S have maximum size n.

Invariant

P1. For node s in settled set S, at least one shortest v-s path contains only settled nodes and d[s] is its distance.

P2. For f in *frontier set* F, at least one v-f path contains only *settled* nodes, except for f, and d[f] is the minimum distance from v to f over all such paths from v to f.

P3. Edges leaving S end in frontier set F.

P4. For w in S or F, bp[w] is w's backpointer on the shortest known path from v to w.

Theorem. For f in *frontier* set F with minimum d value (over nodes in F), d[f] is the shortest-path distance from v to f.

```
F= {v}; d[v]= 0; S= {};                          (1) 1 x (meaning executed 1 time).        O(1)
// invariant: P1, P2, and P3
while (F != {}) {                                (2) true n x, false 1 x.                  O(n)
   f= a node in F with minimum d value;          (3) n x.                                  O(n)
   Remove f from F and add it to S;              (4) n x.                                  Exp O(n log n). Worst O(n^2)
   for each w with (f, w) an edge {              (5) loop condition is evaluated n+e x.     O(n+e)
      if (w not in S or F) {                     (6) true n-1 x, false e-(n-1) x.          Exp O(n+e). Worst O(n*(n+e))
         d[w]= d[f] + wgt(f, w); bp[w]= f;       (7) n-1 x.                                O(n)
         add w to F;                             (8) n-1 x.                                O(n log n)
      }
      else if (d[f] + wgt(f, w) < d[w]) {        (9) e-(n-1) x.                            O(e-(n-1))
         d[w]= d[f] + wgt(f, w); bp[w]= f;       (10) from 0 to e-(n-1) x. .               Exp O(e log n). Worst O(n*e)
      }
   }
}
```

The lines of code in the algorithm above are numbered in red, at their right. Following that is the number of times that line is executed or evaluated. Below, we explain each line except for (1), which is obvious.

(2) Start node v is initially in F. The other n-1 reachable nodes are initially in the far-off set and are placed in F at some point; they are the only nodes to be placed in F. Each iteration of the while-loop removes one node from F and places it in S. After n nodes are removed, F is empty.

(3) and (4). The while-loop repetend is executed n times.

(5). This loop is executed n times, once for each reachable node. With an adjacency list representation of the graph, each time the loop is executed for a node f, the number of iterations is the outdegree of node f, so the loop condition *in total* evaluates to true e (i.e. the number of edges) times and to false n times.

(6) The if-condition is evaluated e times —once for each edge. It is true n-1 times, because each time it is true, node w is moved from the far-off set to the frontier set. Therefore, it is false e-(n-1) times.

(7) and (8). As explained for (6), the loop condition is true n-1 times.

(9). As explained for (6), the if-condition on (9) is false e-(n-1) times.

(10). This is the one place where exact information cannot be given. The if-condition on line (9) could be true 0, 1, …, or up to e-(n-1) times.

[1] We could implement S as a boolean array, but a hash set is closer to a real implementation, in which the nodes are not given by integers but are objects.

Above, we explained why each line is executed the number of times given in red. We now use this information to explain the expected and worst-case execution times. If the expected and worst-case times are different, the expected is in blue and the worst-case is in green.

(1) Adding `v` to empty heap `F` is constant time. An array assignment is constant time. Initializing `S` is constant time.

(2) Testing whether heap `F` is empty is constant time. The test is done `n+1` times.

(3) Peeking at the minimum value in a min-heap is constant time. It is done exactly `n` times.

(4) Removing f from heap `F` is O(`log n`) expected and worst-case time. Adding f to hash set `S` is O(1) expected time and O(`n`) worst-case time. The operations are done `n` times.

(5) The test to see whether there is another edge to process in an adjacency list takes constant time. The test is made `n+e` times.

(6) The tests whether `w` is in hash set `S` or whether `w` is in heap `F` take expected constant time and O(`n`) worst-case time. The tests are done `n+e` times.

(7) The two assignments take constant time. They are performed `n-1` times.

(8) Adding `w` to heap `F` takes expected and worst-case time O(`log n`). The operation is performed `n-1` times.

(9) Evaluating the if-condition takes constant time. It is done `e-(n-1)` times.

(10) We have to be careful here. The two assignments take constant time, but if `w` in in heap `F`, its priority in `F` is `d[w]`, so if `d[w]` changes, the priority of `w` in `F` has to be changed. This takes expected time O(`log n`) and worst-case time O(`n`). Line (10) is performed at most `e-(n-1)` times, which is bounded above by `e`.

Calculating the execution time

The execution time of the algorithm is the sum of the total execution times of each of its lines. Thus, to find the order of execution time, we just have to determine the line with the biggest O(…). We can find different values depending on whether the graph is sparse or dense.

If the graph is sparse, there are few edges, let's say e <= c*n for some constant c. In this case, lines 4, 8, and 10 tell us that the expected time is O(n log n) and 4, 6, and 10 tell us that the worst-case time is $O(n^2)$.

If the graph is dense, there are many edges, up to n(n-1) edges. Assume e is $O(n^2)$. In this case, line 10 tells us that the expected time is $O(n^2 \log n)$ and lines 6 and 10 tell us that the worst-case time is $O(n^3)$.