CSE 101 (WI21, Kane)

Homework 2

Jiahong Long
[redacted]

29 January 2021

1. *Marble run*

We present a algorithm which runs in $O(|V| + |E|)$. We provide an algorithm, a proof of correctness, and an argument as to its runtime.

(a) *Algorithm description*

First, we describe some notation:

- $f : V \times V \mapsto \mathbb{R}$ returns the probability of the marble traversing the edge from v to w if an edge (v, w) exists, ie. $f(v, w)$ is the probability that a marble at v will leave through the edge (v, w). If no edge (v, w) exists, then $f(v, w) = 0$.

Then, the algorithm is as follows:

`compute_p(G, s):`

i. Mark all vertices unexplored.
ii. Run `explore` from s, marking all reachable vertices as explored.
iii. Let V' denote the subset of vertices reachable from s, ie. V' contains all and only vertices marked as explored.
iv. Using a standard topological sort algorithm, sort all vertices in V' into a topological order O.
v. For each vertex $v \in O$ in order:
   A. Set $p(v) = 0$.
   B. If v is a source (ie. has no incoming edges (w, v) where $w \in V'$) then set $p(v) = 1$ and continue.
   C. Otherwise, for each incoming edge (w, v) where $w \in V'$, perform the update

$$p(v) \leftarrow p(v) + p(w)f(w, v)$$

   ie. increment $p(v)$ by $p(w)f(w, v)$.

We posit that after the algorithm returns, $p(v)$ correctly returns the probability that a marble will run through vertex v.

(b) *Proof of correctness*

To show that the algorithm is correct, we will begin with a probability theory proof, and then demonstrate that the algorithm replicates the probabilistic calculations. Let v be an arbitrary vertex in the DAG G. Suppose there exist the n incoming edges

$$(w_1, v),\ (w_2, v),\ (w_3, v),\ \ldots\ ,\ (w_n, v)$$

Trivially, the only way to arrive at v is through one of the incoming edges, and as such the only paths through v must pass through one of $w_1, w_2, \ldots, w_n$. Then, by the law of total probability, (since the event of traversing the edge (w_x, v) is pairwise disjoint from traversing the edge (w_y, v) if $x \neq y$ explicitly because the graph is acyclic) the probability that any given path passes through v is given by

$$p(v) = \sum_{i=1}^{n} p(\text{passing through } v \text{ from } w_i) \tag{1}$$

Let W_i denote the event of passing through vertex w_i and V denote the event of passing through vertex v. Then:

$$= \sum_{i=1}^{n} P(V \cap W_i) \tag{2}$$

Bayes' theorem informs us that $P(V \cap W_i) = P(W_i)P(V \mid W_i)$. Then, $P(W_i)$ is simply equivalent to $p(w_i)$, and $P(V \mid W_i)$ is the probability of following the edge (w_i, v) given that the marble is at w_i so it is given by $f(w_i, v)$. Then:

$$p(v) = \sum_{i=1}^{n} p(w_i) f(w_i, v) \tag{3}$$

We will now show that this is exactly the calculation performed by the algorithm. To begin, recall that `explore` will find all and only the vertices reachable from a start vertex s: thus, parts ii. and iii. of the algorithm correctly generate a subset V' of the vertices V which are eventually reachable from s. In addition, because we process vertices in a toplogical order O at step (v.) it must be true that if a vertex has a predecessor, then the predecessor is processed prior to the vertex itself. This holds for arbitrarily many predecessors.

Then, consider (A.) where all $p(v)$ are initialised to zero. Recall that the subset V' that we are concerned with is effectively the subset of all vertices V that are reachable from the start vertex s. This subset will have exactly one source, because `explore(s)` will find exactly one tree, rooted at s. Therefore, we are guaranteed by step (B.) that $p(s)$ is set equal to 1, since it is the only vertex that we are guaranteed to visit with certainty.

Then, if v is not a source (ie. it is not s) then we let $p(v)$ equal to the sum of all $p(w)f(w, v)$ for $w \in V'$, ie. the sum across all incoming edges from vertices in V' of the probability of visiting any of those vertices, times the probability of taking an edge from that vertex to v. This is exactly equation (3). Furthermore, we are guaranteed that this calculation can be performed because of the topological ordering: when we visit each vertex, we are sure of the fact that all predecessors thereof have already been visited, and hence $p(x)$ where x is any predecessor has been properly updated. Finally, we need not concern ourselves with vertices which are unreachable from s, since there is zero probability that they are visited (by virtue of being unreachable). The algorithm correctly accounts for this since the accumulation logic to calculate $p(v)$ ignores all edges which are incoming from edges not in V'. Thus we have shown the correctness of the algorithm. ■

(c) *Proof of runtime*

We consider each step of the algorithm:

- Step (i.) occurs in $O(|V|)$ time (vacuously).
- Step (ii.) occurs in at most $O(|V| + |E|)$ time at most, since if s is a source of G, then `explore(s)` is equivalent to running DFS on G. If s is *not* a source of G, then `explore(s)` is faster than DFS since not every vertex and not every edge needs to be explored.

- Step (iii.) occurs in $O(|V|)$ time, as we iterate over each $v \in V$ and decide if it should be included in V'.
- Step (iv.) occurs in $O(|V| + |E|)$ time, as topological sort can be done simply by running DFS on G and returning vertices in reverse post-order, while restricting the subset of visitable vertices to those in V'. Using clever logic when assigning post-orders, vertices can simply be pushed to a deque, and so no additional runtime is required to reverse the post-orders.
- In step (v.), we choose to consider the total time amortised over all iterations of the loop:
  - (A.) and (B.) occur in linear time per loop iteration (vacuously). Then, this amortises to $O(|V|)$ over all iterations.
  - Over all iterations, (C.) amortises to $|E|$ time, since each edge is scanned (at most) once when the loop is called on the destination vertex of the edge.

  Hence step (v.) occurs in $O(|V| + |E|)$ time.

It follows that the total runtime is $O(|V| + |E|)$, which linear over the size of the graph. ■

2. *Contracting cycles*

We define a meta-vertex as

> a 'vertex' obtained by taking a cycle and replacing all vertices with a single 'meta-vertex'. Meta-vertices may contain nested meta-vertices. Edges originating at vertices in the cycle are modified to originate at the meta-vertex, and edges terminating at vertices in the cycle are modified to terminate at the meta-vertex. This is a valid abstraction since all vertices (in an optionally directed graph) in a cycle are equally reachable, ie. to be able to reach one vertex in a cycle implies that all vertices in the cycle are reachable.

We will now prove that a set of vertices is strongly connected if and only if it can be represented by a meta-vertex.

> We begin with the forwards direction. Suppose that vertices u and v are strongly connected. Then there exists a sequence of vertices
>
> $$u, x_1, x_2, \ldots, x_n, v$$
>
> such that edges $(u, x_1), (x_1, x_2), \ldots, (x_n, v)$ exist. In addition, a sequence of vertices
>
> $$v, y_1, y_2, \ldots, y_n, u$$
>
> exists such that edges $(v, y_1), (y_1, y_2), \ldots, (y_n, u)$ exist. Hence, the vertices in the circular path $u, x_1, x_2, \ldots, x_n, v, y_1, y_2, \ldots, y_n, u$ form a cycle. Because all vertices in the cycle sequence are strongly connected, they can be represented by a meta-vertex.
>
> In the reverse direction, suppose that m is a meta-vertex. Then, vertices and meta-vertices represented by m must be part of a cycle, and therefore they are all equally reachable from one another. It follows that vertices represented by m are strongly connected.

Observe that the algorithm finds meta-vertices:

> *... find some cycle C in G and let G' be the graph obtained from G by replacing all of the vertices of C with a single vertex with edgest to/from the same vertices that have edge from/to some vertex of C ...*

and then recursively continues this procedure, by calling itself on the graph generated by treating meta-vertices as normal vertices:

> *... Then recursively compute the metagraph of G'.*

until the input is a DAG:

If G is a DAG, return G. ...

The algorithm thus 'lumps' together cycles of vertices and meta-vertices into enveloping meta-vertices. Because we know that meta-vertices represent groups of strongly connected vertices, each such 'lumped' meta-vertex must also represent a set of strongly connected vertices. From *Algorithms* p. 92, we are informed that

Every directed graph is a DAG of its strongly connected components.

It follows that once the meta-vertices are lumped together to such an extent that the graph is a DAG, the remaining graph *must* be a metagraph, since each meta-vertex represents exactly a strongly connected component. Hence the algorithm is accurate. ■

3. *Graph cycle*

We claim that the statement is false. Consider the directed graph G:

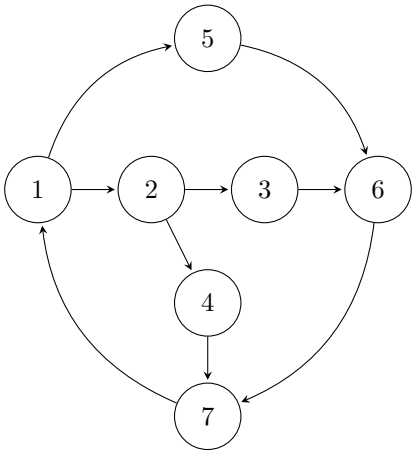

Figure 1: Directed graph G

We will note that G is:

- **trivially finite,** since $|V| = 7$ and $|E| = 9$
- **trivially directed,** since all edges are directed
- **strongly connected,** since DFS from vertex 1 will find every other vertex

Thus G satisfies the conditions of the statement. We will now show that no such cycle that visits each vertex at least once, using each edge at *most* once, exists. Consider vertices 3 and 4. Both these vertices are reachable only from vertex 2, which is itself only reachable from vertex 1. It follows that for any cycle to visit both 3 and 4, it must include the edge between 1 and vertex 2 twice. Hence no such cycle exists, and G itself is a counterexample to the statement. ■

4. *Dijkstra at small distances*

We present an algorithm which runs in $O(|V|+|E|+L)$. We provide an algorithm, a proof of correctness, and an argument as to its runtime.

(a) *Algorithm description*

To motivate the algorithm, we note that in the most optimal implementation of Dijkstra's algorithm, the runtime is

$$O(|V|\texttt{deletemin} + (|V| + |E|)\texttt{insert})$$

where `deletemin` and `insert` are slow, by virtue of the need to use a priority queue, particularly because the edge weights may be fractional or arbitrarily large. Since we are guaranteed that the edge weights are to be positive integers *and* that there exists an upper bound L on the distances we care about, this begs an implementation which uses some hash table, where collisions are resolved using a list. To facilitate constant-time `deletemin` and constant-time `insert`, it is prudent to also store a mapping between pointers into the hash table and vertices themselves. With this in mind, we proceed, and implement Dijkstra's algorithm using a hash table with $L+1$ slots such that vertices v where $dist(v) = x$ are put into the xth slot.

Let G denote the graph, $l(a, b)$ denote the edge length between vertices a and b where the edge (a, b) exists, s denote the start vertex, and L denote the maximum path length. Then the algorithm is described as:

`uint_edge_dijkstra(G, l, s, L):`

i. For all $u \in V$:
   A. Set $dist(u) = \infty$.
   B. Then, set $prev(u) = \varnothing$.
ii. Set $dist(s) = 0$.
iii. Create a length $L+1$ array Q of linked lists, and maintain a `top` pointer to the first nonempty 'head' node in the list. Then, insert s at $Q[dist(s) = 0]$, the zeroth position.
iv. While Q is not empty:
   A. Let u be the top element in Q. Remove u from Q, and edit `top` if the slot at $dist(u)$ is exhausted.
   B. If $dist(u) = \infty$, skip this iteration.
   C. For each outgoing edge (u, v), let $d = dist(u) + l(u, v)$. Then, if $dist(v) > d$ and $d \leq L$, set $dist(v) = d$, then set $prev(v) = u$, and (re)insert v at the dth slot.
v. Return all vertices v where $dist(v) \leq L$.

We posit that the algorithm correctly returns all vertices which can be visited in a distance less than or equal to the threshold L.

(b) *Proof of correctness*

In order to prove correctness, we must show that every vertex at distance of at most L is found and returned. Towards this, it will be sufficient to demonstrate that we

- check every vertex,
- calculate all minimum distances correctly,
- and return all vertices with minimum distances $\leq L$.

To begin, we note that parts (i.), (ii.)[1], and (iii.) are (nearly) identical to the reference Dijkstra implementation given by *Algorithms* p. 110, Dasgupta et al. Hence, the setup for Dijsktra does not change, in that all distances (except for the start vertex) are initialised to ∞, previous pointers are set to `null`, and a priority queue is initialised with all vertices. Then, in iterating over the queue, we check all vertices:

[1] We intialise $dist(s) = 0$, since there is no cost in not leaving a vertex. Dasgupta et al. choose 1 for this value; this makes no practical difference other than that all subsequent distances are increased by 1. However, since in this case we care about the total weights of the edges, it does not make sense for it to cost a nonzero amount to visit a vertex from itself.

- Step (A.) is identical to that in the reference Dijkstra implementation. During the first iteration, the vertex which is popped is s, as it is the only vertex with distance *dist* equal to zero (all others are infinity). During all subsequent iterations, this will be the/a vertex u with (one of) the lowest $dist(u)$. By virtue of this, we are guaranteed that we will always find the lowest cost for each vertex, because we have established an ordering relationship such that vertices are visited in order of lowest cost from the start vertex first. In addition, because we are guaranteed that all edge weights are positive integers, we know that $l(x, y) > 0$ for all $x, y \in V$. Therefore, propagating from the start vertex s, the distances of all subsequent vertices must monotonically increase: this fact immediately implies that inserting new elements will *never* create a situation in which the distance of the newly inserted element is lower than the distance corresponding to the slot pointed to by the `top` pointer.

- In step (C.), assuming $dist(u) \neq \infty$, we calculate prospective distances to all of its neighbours v, ie. for all edges (u, v), we calculate a new prospective distance $d = dist(u) + l(u, v)$. d then represents the distance to v through u. In the event that this is lower than both the existing distance to v *and* less than or equal to the threshold L, then the $dist(v)$ is updated to equal d. This step ensures that for each vertex, we store *only* distances which are shorter than or equal to the threshold L. Thus step (C.) is correct.

- Step (B.) now becomes clear: if no distances greater than L are ever stored, then $dist(v) = \infty$ for all vertices v which are of a distance greater than L. By skipping all such vertices, we avoid extraneously calculating prospective distances to its neighbours. We prove that the algorithm will never erroneously skip over a vertex v by contradiction:

  Assume towards a contradiction that v is a vertex which was skipped over by (B.) erroneously. For the skip to be erroneous, a sequence of vertices

  $$(x_0 = s),\ x_1,\ x_2,\ \ldots\ ,\ x_{n-1},\ (x_n = v)$$

  must exist such that

  $$L\ \geq\ \sum_{i=1}^{n} l(x_{i-1}, x_i)$$

  Then, it must be true that for each x_i where $i = 0, 1, \ldots, n$ we will eventually update $dist(x_i)$ to some value less than or equal to L. If this is true, then the loop logic at step (C.) will assign proper distances less than or equal to L to all x_i, including $x_n = v$. This is a contradiction: hence no vertices can be erroneously skipped over.

  Thus step (B.) is correct.

When the loop concludes, the only vertices v where $dist(v) \neq \infty$ will satisfy the condition $dist(v) \leq L$. All such vertices are returned by step (v.) We have thus shown that the algorithm checks every vertex, calculates all minimum distances correctly (where the minimum distance is less than L as shown by contradiction), and returns all such vertices. Hence, we have shown that the algorithm is correct. ■

(c) *Proof of runtime*

To begin, we restate as a given that Dijkstra's algorithm runs in

$$O(|V|\texttt{deletemin}\ +\ (|V| + |E|)(\texttt{insert, decreasekey}))$$

The difference between `uint_edge_dijkstra` and reference implementations of Dijsktra is purely in step (iii.): instead of a priority queue, we initialise a list of length $L+1$. Hence this initialisation step runs in additional $O(L)$ time.

We posit that usage of a hash table implies that `deletemin` operations run in $O(1)$ time: because we already have a pointer to the slot containing the elements with the minimum distance, `deletemin` will at *most* need to modify the `top` pointer to point to a higher-distance slot. Then this is constant time.

We also posit that the usage of a hash table makes `insert` and `decreasekey` operations run in $O(1)$. Crucial to this is the fact that we only concern ourselves with vertices v such that $dist(v) \leq L$. Since there exist distinct slots for each distance less than or equal to L that can be accessed randomly and immediately, insertion takes place in $O(1)$ time: all we need is to modify at most one pointer to point to the new vertex. Similarly, `decreasekey` is constant time because it is equivalent to removing a vertex and then reinserting it into a different slot. Removal of an element from a hash table resolved by lists runs in constant time (since we have stored a table mapping vertices into pointers into the table), and we have just shown that insertion is done in constant time. Hence `decreasekey` runs in $O(1)$.

With this, we can state that the total runtime of the algorithm is equivalent to

$$O(|V| + |V| + |E|) + O(L) = \boxed{O(|V| + |E| + L)}$$

Thus we have shown the runtime to be linear. ■

5. *Time spent*

6hr55min. Tracking done via `wakatime.com`.