Globalizing Dijkstra

June 11, 2010

1 Definitions

Let $G = (V, E, \omega)$ be a graph with strictly positive edge weights. Let $P(v, w) = \{e_1, e_2, \ldots, e_k\}$ be a sequence of edges that denotes a path between v, w, i.e., $e_1 = (v, v_1)$, $e_i = (v_{i-1}, v_i), \forall 1 < i < k$ and $e_k = (v_{k-1}, w)$. Let $cost(P) = \sum_{e \in P} \omega(e)$ denote the cost of a path. Without loss of generality we will assume that there is at least one path between any two vertices in G. Then, the distance of any two vertices v, w is defined as the minimal cost of any path between them. A path with minimal cost between its endvertices is called a shortest path. Note that a shortest path is also perfectly defined by the sequence of vertices it contains $(v_1, v_2, \ldots, v_k)$. In the case where there are more than one path between vertices, we will sort them lexicographically by the sequence of the labels of their vertices and define P_1 to be smaller than P_2 if both have the same cost and P_1 is sorted before P_2 in the lexikographic order. As an example: given two paths $P_1 = (v_1, v_3, v_6, v_2)$ and $P_2 = (v_1, v_4, v_2)$, $P_1 < P_2$ although the latter has fewer edges. With this, there is a unique smallest shortest path between any two given vertices.

The $APSP$-problem asks for the shortest path between all pairs of vertices. To avoid ties, we will restrict our analysis to the APSP with smallest shortest paths, w.l.o.g. Note that for one single vertex v, the union of all of its smallest shortest path to all other vertices constitute a uniquely determined tree, called the shortest path tree of v ($SPT(v)$).

The set of neighbors of vertex v, i.e., the set of vertices w with an edge $(v, w) \in G$ is denoted by $N(v)$.

We need to generalize the classical notion of relaxation: in the classical Dijkstra, a node z with current minimal known distance $d(v, z)$ is extracted from a priority queue in each step. For each of the edges (z, y) it would try to relax the distance of v to y by checking whether $d(v, z) + d(z, y)$ is smaller than the currently best known distance from $d(v, y)$. In our algorithm, a node v will ask all of his neighbors w whether the currently known distance $d(v, z)$ can be relaxed by their currently known distance to z, i.e., whether $d(v, w) + d(w, z) < d(v, z)$.

2 Globalizing Dijkstra

In this section we will describe a new algorithm for the $APSP$ with a runtime of $O(nm)$ *hopefully, that is.*

Theorem 1 *$APSP$ can be solved in $O(nm)$ for undirected graphs with strictly positive edge weights.*

The proof proceeds in the following steps:

1. We will first show that solving the single source shortest path problem (SSSP), which asks for a shortest path from a single vertex s to all other vertices, can be solved in $O(deg(s) \cdot n + n \log deg(s)$ if the shortest path trees of its neighbors are known.

2. We will then proof that whenever v accepts the $i-$th vertex z in its $SPT(v)$ and its shortest path to z contains neighbor w, then z must be one of the $i-$th closest neighbors of w, too.

3. It is clear that any node can find out quickly who is the closest and second closest vertex to it. The first is the vertex itself, the second is the neighbor to which it is connected with the minimum-cost edge.

4. If in round i each vertex holds a list of its i closest vertices, we show that it is possible to construct from this the list of the $i+1$th closest nodes for all nodes. This completes the proof.

Lemma 1 *Provided that the SSSP of all neighbors w of v is solved, the $SSSP(v)$ can be solved in $O(deg(v) \cdot n + n \cdot log(deg(v))$.*

To make the discussion easier, we will only describe how v can find the distance of itself to all other vertices. Constructing the shortest path tree just requires an extra array with pointers.
We will assume that every neighbor knows the distance to all other vertices and that they provide this information in the form of a sorted vertex array, where the node with the smallest distance to them is presented first. Let $r_w(i)$ denote the rank of node i with respect to node w and the distance of w to all other vertices.
v will now build its own list in which all nodes are sorted by their distance to it. The list $dist(v)$ is initialized with v itself. A node that is added to the list is said to be *accepted* by v and of course the invariant is that the distance to accepted nodes is correctly determined. The main idea is that v keeps a priority queue $PQ(v)$ that contains at all times for each neighbor w at most one information, namely the vertex z to which it currently provides the best distance $d(v,z) = d(v,w) + d(w,z)$ and that is not yet accepted by v. We assume that the objects stored in $PQ(v)$ contain these three informations and are sorted by $d(v,z)$ which can be seen as the current best guess on the distance of v to z if the shortest path contains w. In the beginning the priority queue is filled with the neighbors w of v, where the key is their distance to v. It is clear that if we

extract the minimum from $PQ(v)$ after the initialization, this picks the node with the second smallest distance to v as in the classical Dijkstra algorithm. Once we extract an object from $PQ(v)$ with minimal distance, we can be sure that this is the correct distance to v. Of course, we need to insert a new object for neighbor w with the next closest vertex to w that is not yet accepted by v.

To maintain the invariant on its priority queue $PQ(v)$, v maintains the following two arrays: a distance array of length n that is initialized to 0 for v itself and set to ∞ for all others; one array $inPQ$ of pointers, that points to the corresponding object in $PQ(v)$ if it contains vertex z. Let $dist_i(v)$ denote the list of the first i accepted nodes of v.

We have shown that $PQ(v)$ is correctly initialized and that a first extractMin-operation will yield the vertex with the second smallest distance to v after v itself. Let now w be the neighbor such that the shortest path from v to the last accepted vertex z contains w. In the order of their rank, starting from $r_w(z)$, w checks for all vertices $z^*, r_w(z) < r_w(z^*)$ whether it can decrease the currently known distance from v to z^* by looking into $dist(v)$. There are two cases: either the distance of z^* was ∞ before, than we can just insert a new object into $PQ(v)$ with the information $w, z^*, d(v,w) + d(w,z^*)$. If $d(v,z^*) < \infty$, then the object contains the information, over which neighbor w' the up to now best distance was provided. This neighbor is then replaced by w. Note that if w provides the same distance but is lexikographically smaller than the neighbor which previously provided the same distance, w will also replace w'. Iteratively, w' then search through its array to find a new node to which it can provide a better distance to v. Note that these replacements cannot happen arbitrarily often: since every node z is contained in each of the neighbor's arrays only once, its distance to v can also only be increased $deg(v)$ times. If a neighbor cannot find any node z whose distance to v is better than the currently known one, it will not provide any shortest paths anymore. It can thus safely be removed from $PQ(v)$.

It can be easily seen that the runtime for determining v's SPT is determined by at most $deg(v) \cdot n$ $decreaseKey$-Operations and exactly n $extractMin$-Operations. If the $PQ(v)$ is implemented as a Fibonacci-Heap, the total runtime is given by $O(deg(v) \cdot n + n \log deg(v))$.

We will later show that if this runtime is valid for constructing each single SPT, the global runtime is in $O(nm)$. To achieve this, we first need the following simple, but crucial theorem:

Theorem 2 *When v accepts its i-th closest node z because of a shortest path containing neighbor w, then z must be among the i-th closest nodes to w as well.*

Since the accepted shortest path $P(v,z)$ contains w, $d(w,z) < d(v,z)$. Assume, that z is not among the i closest nodes to w. Then, since v accepted z via w, the set of the i closest nodes of w contains at least one node z' that is not yet contained in v's list. Since z' is contained in w's list, it follows that $d(w,z') < d(w,z)$. But then, v must have accepted z' before accepting z, in contrast to the assumption and its implication.

This theorem implies that it is enough to know the i closest nodes of all neighbors of v in order to determine v's i-th closest node. It is clear that after the first round, every node knows its two closest nodes, namely itself and the neighbor to which it is connected with its minimum cost edge. We will now show that there is an order in which the nodes can determine their i-th closest neighbors without any cyclic dependencies. After this, we have shown that it is possible to construct the list $d_{i+1}(v)$ for all vertices from the lists $d_i(v)$.

Let now H_i denote the dependency graph of the i-th round, i.e., for each node v we add a directed edge to its neighbor w if the i-th accepted shortest path of v contains w. Since we have n vertices and one outgoing edge for each of them, this implies that in every connected component there is exactly one cycle. We will now proof the following theorem:

Theorem 3 *Let $C = v_1, v_2, \ldots, v_k$ denote a cycle in H_i. Then at least one of them does not accept the same node z as the node on which it depends accepts in that same round.*

Assume that all of the nodes in the cycle accept the same node z in the i-th round as their i-th closest node. It is clear that for each edge in the cycle, $d(v_i, z) > d(v_{(i+1)\mathrm{mod}_k,}, z)$ is true because a shortest path from v_i over $v_{(i+1)\mathrm{mod}_k}$ to z must be more expensive than the path from $v_{(i+1)\mathrm{mod}_k}$ to z. This leads to an immediate contradiction.

We will now state the following crucial theorem:

Theorem 4 *There is an order in which the vertices can be processed such that if every node knows its $i-1$ closest neighbors, before v decides who is its i-th closest node, its priority queue $PQ_i(v)$ knows for each neighbor the node z with the following properties:*

1. *z is in $d_i(w)$ and thus $d(w, z)$ is known.*
2. *The upper bound on the distance from v to z provided by w (given by $d(v, w) + d(w, z)$ is the minimal known distance from v to z, i.e., no other neighbor can provide a better distance to z at this time.*

We will now assume that every node v knows its $i-1$ closest neighbors, stored in $dist_i(v)$ (induction hypothesis). By Theorem 2 we know that whatever v accepts in the i-th round, it must be a node that was already accepted by one of its neighbors beforehand. The invariant on $PQ(v)$ states that before the decision of v, it keeps for all neighbors $w \in N(v)$ the node z with minimal (known) distance to w and minimal current distance to v that is not yet accepted by v. By the induction hypothesis we assume that before the $i-1$th decision of v, $PQ_{i-1}(v)$ asserted this invariant. Let now v remove the minimum from $PQ_{i-1}(v)$ and thereby accept its $i-1$th closest node. The next closest node is then either provided by one of the other neighbors or by the same neighbor that provided the minimum in round $i-1$. Let $w_{i-1}(v)$ denote the neighbor of w through which the shortest path to the $i-1$th accepted node $z_{i-1}(v)$ runs. The

update procedure on $PQ(v)$ thus needs to find the next minimum provided by w_{i-1}. Let $z_{i-1}(v)$ be the jth closest neighbor of $w_{i-1}(v)$. Check for all vertices $z_k(w), j < k < i$ whether w can relax the distance ot $z_k(w)$ to v. If there is such a vertex z and its distance to v is currently ∞, we can directly insert the new information $w, z, d(v, w) + d(w, z)$ into $PQ(v)$, otherwise we will follow the pointer to the object in $PQ(v)$, decrease its key accordingly and replace the former providing neighbor by w.

By Theorem 3 we know that in each cycle in H_i there exists a node v that accepts a different node z' as its i-th closest node as the node w it depends on. Since z' must be among the i-th closest nodes of w but is not the i-th closest node, it must be in $d_{i-1}(w)$. Thus we know that if each node v checks for all higher-ranked vertices in the list of the neighbor from which it accepted the last node whether they can relax one of its distances to other nodes, we will find in each cycle at least one such node. It implies that within the cycle at least one node's $PQ(v)$ is ready for the i-th decision. Thus, all nodes whose decision depends on this node's can now also update their priority queue. Since v breaks the dependency cycle and since every connected component contains exactly one such cycle, all other dependencies can be resolved in the topological sorting order of the resulting directed tree.

We have thus shown that with the knowledge of the i closest neighbors to each node v we can compute the $i + 1$th closest neighbors. We have not yet shown how to maintain the invariant on $PQ(v)$, namely that for each neighbor we store the node z which fulfills the above given properties. At the beginning of the first round this is certainly true, because for each neighbor w we store the information $w, w, d(v, w)$. But whenever we extract the minimum object in $PQ(v)$, and call the $update(PQ(v), w_i(v)$ procedure, two things can happen:

1. The relaxation of node $d_i(w)$ causes some other neighbor w' to vanish from $PQ(v)$.

2. There is no other neighbor in $d_i(w_i(v))$'s list that can relax any node z with respect to v;

The first case can be cured by a call of the procedure $update(PQ(v), w')$; if this leads to another replaced neighbor, this needs to be done iteratively until no neighbor is replaced anymore. The second case says that currently there is no interesting shortest path provided by $w_i(v)$. This does not mean that there might not be promising shortest paths in the future. Thus, we need to set a flag in $w_i(v)$ to check on v's PQ whenever it accepts a new vertex. I.e., in all following rounds $j > i$ and as long as some neighbor w is not contained in $PQ(v)$, it needs to check for its newly accepted node $z_j(w)$ whether it can relax v's distance to z. If so, it inserts itself into $PQ(v)$ or updates it.

Analysis of the runtime: Note that in each round, each node v extracts one minimum from a priority queue with at most $deg(v)$ entries. The updating procedure of its priority queue checks for each neighbor w for each of the nodes z that it accepted so far whether they can relax its current distance to z. Note that every node accepted by w is only checked once with respect to v in the

whole algorithm. Thus, we have $deg(v) * n$ many decreaseKey operations and n extractMin operations. If PQ is a Fibonacci-Heap, the runtime for each single vertex is given by $O(n(deg(v) + \log(deg(v))$. The total runtime thus amounts to:

$$\begin{aligned} n \sum_{v \in V} deg(v) + n \sum_{v \in V} \log(deg(v)) &= 2nm + n \log(\Pi_{v \in V} deg(v) \quad (1) \\ &\leq 2nm + n^2 \log(2m/n) = O(nm), \quad (2) \end{aligned}$$

where the last line comes from the fact that $\Pi_{v \in V} deg(v)$ under the condition that $\sum_{v \in V} deg(v) = 2m$ is maximized if $deg(v) = 2m/n$ for all v. Theorem 1 thus follows.