Lecture notes for "Analysis of Algorithms": **Dynamic All-Pairs Shortest Paths** (*Second draft*) [*]

Lecturer: *Uri Zwick*

December 2009

1 Dijstra's algorithm

Dijkstra's [Dij59] classical single-source shortest paths algorithm, for graphs with *non-negative* edge weights, is given in Figure 1. When implemented using a priority queue, such as Fibonacci heaps (see [FT87]), that supports insert and decrease key operations in $O(1)$ time, and extract-min operations in $O(\log n)$ time, its running time is $O(m + n \log n)$.

We may view the operation of Dijkstra's algorithm as follows. Suppose that we have just found out that π is a shortest path from s to u, for some vertex $u \in V$. Then, for every edge $(u, v) \in E$, the path $\pi \cdot (u, v)$, obtained by appending the edge (u, v) to the path π, is made a *candidate* for being a shortest path from s to v. As subpaths of shortest paths are also shortest paths, at least one shortest path to any vertex v is made a candidate at some stage and then recognized as a shortest path at a later stage.

2 The algorithm of Karger, Koller and Phillips

Suppose now that we would like to solve the *All-Pairs Shortest Paths* (APSP) problem. One option is of course to run Dijkstra's algorithm independently from each vertex of the graph. Can we gain anything by trying to find all shortest paths together?

Suppose for simplicity that all edge lengths are strictly positive. We can try to find shortest paths between all pairs of vertices in increasing order of their (weighted) length. Suppose that we have just found out that π is a shortest path from s to u and that $(u, v) \in E$. Should we immediately make $\pi \cdot (u, v)$ a candidate for being a shortest path from s to v? If $\pi \cdot (u, v)$ is a shortest path,

[*]Burn after reading!

```
Function DIJKSTRA($G = (V, E, c), s$)
    $d[s] \leftarrow 0$
    $Q \leftarrow$ heap()
    heap-insert($Q, s, d[s]$)
    foreach $v \in V - \{s\}$ do
        $d[v] \leftarrow \infty$
        $p[v] \leftarrow null$
    while $Q \neq \emptyset$ do
        $u \leftarrow$ extract-min($Q$)
        foreach $e \in E_{out}(u)$ do
            relax($e$)
```

```
Function relax($e = (u, v)$)
    if $d[u] + c(u, v) < d[v]$ then
        $d[v] \leftarrow d[u] + c[e]$
        if $p[v] = null$ then
            heap-insert($Q, v, d[v]$)
        else
            decrease-key($Q, v, d[v]$)
        $p[v] \leftarrow u$
```

Figure 1: Dijkstra's single-source shortest paths algorithm.

```
Function KKP($G = (V, E, c)$)
    foreach $u \in V$ do
        $d[u, u] \leftarrow 0$
        $in[u], out[u] \leftarrow \emptyset$
    foreach $u \neq v \in V$ do
        $d[u, v] \leftarrow \infty$
        $p[u, v] \leftarrow null$
    $Q \leftarrow$ heap()
    foreach $e = (u, v) \in E$ do
        $d[u, v] \leftarrow c[e]$
        $p[u, v] \leftarrow e$
        heap-insert($Q, (u, v), d[u, v]$)
    while $Q \neq \emptyset$ do
        $(u, v) \leftarrow$ extract-min($Q$)
        insert($in[v], u$)
        foreach $e \in out[v]$ do
            relax($u, e$)
        if $start[p[u, v]] = u$ then
            insert($out[u], p[u, v]$)
            foreach $w \in in[u]$ do
                relax($w, p[u, v]$)
```

```
Function relax($u, e = (v, w)$)
    if $d[u, v] + c[e] < d[u, w]$ then
        $d[u, w] \leftarrow d[u, v] + c[e]$
        if $p[u, w] = null$ then
            heap-insert($Q, (u, w), d[u, w]$)
        else
            decrease-key($Q, (u, w), d[u, w]$)
        $p[u, w] \leftarrow e$
```

Figure 2: The algorithm of Karger, Koller and Phillips.

then in particular the edge (u, v) is a shortest path from u to v. If we do not know yet that (u, v) is a shortest path, there is no reason yet to make $\pi \cdot (u, v)$ a candidate. (If we know that (u, v) is *not* a shortest path, then there is definitely no reason to make $\pi \cdot (u, v)$ a candidate.)

An algorithm that exploits this idea was developed by Karger, Koller and Phillips [KKP93]. A version of their algorithm is presented in Figure 2. A similar algorithm was also devised by McGeouch [McG95].

For every vertex $u \in V$, we maintain list $in[u]$ of all the vertices w from which a shortest path from w to u was already found by the algorithm, and a list $out[u]$ of all the edges (u, w) emanating from u which were already identified as shortest paths.

Definition 2.1 (Essential edges) *Let $G = (V, E, c)$ be a weighted directed graph. An edge $e = (u, v) \in E$ is said to be* essential *if and only if it is a shortest path from u to v.*

Theorem 2.2 *Algorithm* KKP *finds shortest paths between all pairs of vertices in a graph with non-negative edge weights. Its running time is $O(m^*n + n^2 \log n)$, where m^* is the number of essential edges in the graph.*

3 The algorithm of Demetrescu and Italiano

3.1 Path systems

The algorithm maintains a collection of paths. Each path in the collection is said to be *generated*. Some of the paths are also marked as *selected*. If a path belongs to the collection maintained, then any subpath of it also belongs to the collection. Each path π has the following information associated with it:

$\ell[\pi]$ – The path obtained from π by removing its *last* edge.

$r[\pi]$ – The path obtained from π by removing its *first* edge.

$start[\pi]$ – The *first vertex* on π.

$end[\pi]$ – The *last vertex* on π.

$first[\pi]$ – The *first edge* on π.

$last[\pi]$ – The *last edge* on π.

$cost[\pi]$ – The cost of π.

$sel[\pi]$ – `true` if and only if the path π is selected.

$GL[\pi]$ – A list containing all the *generated* left extensions of π.

$GR[\pi]$ – A list containing all the *generated* right extensions of π.

$SL[\pi]$ – A list containing all the *selected* left extensions of π.

$SR[\pi]$ – A list containing references to all the *generated* right extensions of π.

```
Function path(v)
    $\pi \leftarrow$ new-path()
    $\ell[\pi] \leftarrow null$
    $r[\pi] \leftarrow null$
    $start[\pi] \leftarrow v$
    $end[\pi] \leftarrow v$
    $first[\pi] \leftarrow null$
    $last[\pi] \leftarrow null$
    $cost[\pi] \leftarrow 0$
    $sel[\pi] \leftarrow$ true
    $GL[\pi], GR[\pi] \leftarrow \emptyset$
    $SL[\pi], SR[\pi] \leftarrow \emptyset$
    return $\pi$
```

```
Function path(e = (u, v))
    $\pi \leftarrow$ new-path()
    $\ell[\pi] \leftarrow \pi[u]$
    $r[\pi] \leftarrow \pi[v]$
    $start[\pi] \leftarrow u$
    $end[\pi] \leftarrow v$
    $first[\pi] \leftarrow e$
    $last[\pi] \leftarrow e$
    $cost[\pi] \leftarrow c[e]$
    $sel[\pi] \leftarrow$ false
    $GL[\pi], GR[\pi] \leftarrow \emptyset$
    $SL[\pi], SR[\pi] \leftarrow \emptyset$
    insert($GL[\pi[v]], \pi$)
    insert($GR[\pi[u]], \pi$)
    return $\pi$
```

```
Function path($\pi_1, \pi_2$)
    if $r[\pi_1] \neq \ell[\pi_2]$ then error
    $\pi \leftarrow$ new-path()
    $\ell[\pi] \leftarrow \pi_1$
    $r[\pi] \leftarrow \pi_2$
    $start[\pi] \leftarrow start[\pi_1]$
    $end[\pi] \leftarrow end[\pi_2]$
    $first[\pi] \leftarrow first[\pi_1]$
    $last[\pi] \leftarrow last[\pi_2]$
    $cost[\pi] \leftarrow c[first[\pi]] + cost[\pi_2]$
    $sel[\pi] \leftarrow$ false
    $GL[\pi], GR[\pi] \leftarrow \emptyset$
    $SL[\pi], SR[\pi] \leftarrow \emptyset$
    insert($GL[\pi_2], \pi$)
    insert($GR[\pi_1], \pi$)
    return $\pi$
```

Figure 3: Generating a new path and inserting it into the path system.

New paths are generated using the constructors given in Figure 3. A call `path`(v), where $v \in V$, generates an empty path composed of the vertex v on its own. A call `path`(e), where $e \in E$, generates a path composed of the edge e. More interestingly, if π_1 and π_2 are such that $r[\pi_1] = \ell[\pi_2]$, i.e., if the path obtained from π_1 by removing its first edge is equal to the path obtained from π_2 by removing its last edge, then `path`(π_1, π_2) generates a path combining π_1 and π_2. The combined path is the path obtained by appending the last edge of π_2 to π_1, or equivalently, by prepending the first edge of π_1 and π_2.

```
Function apsp(G = (V, E, c))
    foreach $u \in V$ do
        $\pi[u] \leftarrow$ path($u$)
        $d[u, u] \leftarrow 0$
        $p[u, u] \leftarrow \pi[u]$
    foreach $u \neq v \in V$ do
        $P[u, v] \leftarrow$ heap()
    insert-edges($E$)
    build-paths()
```

```
Function insert-edges($E_{ins}$)
    foreach $e = (u, v) \in E_{ins}$ do
        $\pi \leftarrow$ path($e$)
        heap-insert($P[u, v], \pi, cost[\pi]$)
```

```
Function delete-edges($E_{del}$)
    foreach $e \in E_{ins}$ do
        remove-path($\pi[e]$)
```

Definition 3.1 *A collection of generated and selected paths is* well-maintained *if and only if the following conditions are satisfied:*

1. *π is generated iff $l[\pi]$ and $r[\pi]$ are selected.*

Function `build-paths()`

```
$Q \leftarrow$ heap()
init-build-paths()
while $Q \neq \phi$ do
    $(u,v) \leftarrow$ extract-min($Q$)
    new-shortest-path($p[u,v]$)
```

Function `init-build-paths()`

```
foreach $u \neq v \in V$ do
    $d[u,v] \leftarrow \infty$
    $p[u,v] \leftarrow null$
    if $P[u,v] \neq \phi$ then
        $\pi \leftarrow$ find-min($P[u,v]$)
        $d[u,v] \leftarrow cost[\pi]$
        $p[u,v] \leftarrow \pi$
        heap-insert($Q, (u,v), d[u,v]$)
```

Function `new-shortest-path(`π`)`

```
if $sel[\pi] =$ false then
    $sel[\pi] \leftarrow$ true
    insert($SL[r[\pi]], \pi$)
    insert($SR[l[\pi]], \pi$)
    foreach $\pi' \in SL[l[\pi]]$ do
        $\pi'' \leftarrow$ path($\pi', \pi$)
        examine($\pi''$)
    foreach $\pi' \in SR[r[\pi]]$ do
        $\pi'' \leftarrow$ path($\pi, \pi'$)
        examine($\pi''$)
```

Function `examine(`π`)`

```
$u \leftarrow start[\pi]$ ; $v \leftarrow end[\pi]$
heap-insert($P[u,v], \pi, cost[\pi]$)
if $cost[\pi] < d[u,v]$ then
    $d[u,v] \leftarrow cost[\pi]$
    if $p[u,v] = null$ then
        heap-insert($Q, (u,v), d[u,v]$)
    else
        decrease-key($Q, (u,v), d[u,v]$)
    $p[u,v] \leftarrow \pi$
```

Figure 4: The functions `build-paths` and `new-shortest-path`.

2. *If π is generated then $\pi \in GL[r[\pi]]$ and $\pi \in GR[l[\pi]]$.*

3. *If π is selected then $\pi \in SL[r[\pi]]$ and $\pi \in SR[l[\pi]]$.*

For every pair of vertices $u, v \in V$, we maintain a priority queue $P[u,v]$ that contains all the generated paths going from u to v. The key of each path π in the priority queue is its cost.

3.2 The static case

To simplify the presentation of the algorithm we start by making the following simplifying assumption:

Uniqueness assumption: For every two vertices $u, v \in V$, there is a *unique* shortest path from u to v in the graph.

Lemma 3.2 *If the collection of paths is well-maintained before a call to* `build-paths`, `insert-edges` *and* `remove-paths`*, then it is also well-maintained after the call.*

```
Function update(E_del, E_ins)
    delete-edges($E_{del}$)
    insert-edges($E_{ins}$)
    build-paths()
```

```
Function vertex-update(v, E_new)
    $E_{del} \leftarrow \{(x,y) \in E \mid x = v$ or $y = v\}$
    $E \leftarrow E - E_{del} \cup E_{new}$
    update($E_{del}, E_{new}$)
```

```
Function full-update(v, E_new)
    $t \leftarrow t + 1$
    $v[t] \leftarrow v$
    vertex-update($v, E_{new}$)
    $j \leftarrow 1$
    while $j < t$ do
        dummy-update($v[t-j]$)
        $j \leftarrow 2j$
```

```
Function dummy-update(v)
    $E_v \leftarrow \{(x,y) \in E \mid x = v$ or $y = v\}$
    vertex-update($v, E_v$)
```

Figure 5: Updating shortest paths.

Figure 6: F-update

Definition 3.3 (Locally Shortest Paths) *A path π is a* locally shortest path *(LSP) if the paths $\ell[\pi]$ and $r[\pi]$ obtained by removing the last or first edge of π are shortest paths.*

A shortest path is clearly a locally shortest path, but a locally shortest path is not necessarily a shortest path. Also note that every edge is a locally shortest path.

Theorem 3.4 *Suppose that* `build-paths` *is run on a collection of paths that initially contains $\pi[e]$, for every $e \in E$, and $\pi[v]$, for every $v \in V$, and that all shortest paths in the graph are unique. Then at the end of the run we have:*

1. *The selected paths are exactly the shortest paths in the graph.*
2. *The generated paths are exactly the locally shortest paths in the graph.*
3. *For every $u, v \in V$, $p[u, v]$ is the shortest path from u to v in the graph.*

The running time of the algorithm is $O(|LSP| + n^2 \log n)$, where LSP is the set of locally shortest paths in the graph.

Lemma 3.5 *Under the uniqueness assumption, $|LSP| \leq m^*n \leq mn$, where m^* is the number of essential edges of the graph.*

Lemma 3.6 *The number of locally shortest paths passing through a given vertex v is at most $3n^2$.*

Function `remove-path`(π)

```
if $\ell[\pi] \neq nu$ then
    delete($GR[\ell[\pi]], \pi$)
    delete($GL[r[\pi]], \pi$)
    if $sel[\pi] = $ true then
        delete($SR[\ell[\pi]], \pi$)
        delete($SL[r[\pi]], \pi$)
$u \leftarrow start[\pi]$ ; $v \leftarrow end[\pi]$
heap-delete($P[u, v], \pi$)
foreach $\pi' \in GL[\pi] \cup GR[\pi]$ do
    remove-path($\pi'$)
```

Figure 7: Removing a path and all its extensions from the system.

3.3 The dynamic case

E_{del} - the set of edges *deleted* or that have their cost changed.

E_{ins} - the set of edges *inserted* or that have their cost changed.

Note that edges that have their cost changed belong to both E_{del} and E_{ins}. Such edges are first removed from the graph and then reinserted with their modified cost.

Definition 3.7 (Vertex updates) *A vertex update operation may add, delete, or change the cost of edges incident on a vertex v. An update is said to be an* increasing *update if it only deletes edges or increases the cost of existing edges. An update is said to be an* decreasing *update if it only inserts edges or decreases the cost of existing edges. (Note that an increasing update can only increase distances in the graph, while a decreasing update can only decrease distances.)*

Suppose that we have just performed an update on vertex v. Generated paths passing through v may now use edges that are no longer part of the graph, or they may use edges whose cost was either increased or decreased. Shortest paths passing through v may stop being shortest paths

The first step we take in order to recompute the shortest paths in the graph following an update of vertex v is to remove all paths passing through v from our system.

All paths passing through an edge e are removed by calling procedure `remove-path` given in Figure 7. All these paths are extensions of $\pi[e]$ and can therefore be recursively reached by following the links in the lists $GL[\cdot]$ and $GR[\cdot]$. (This is where the lists $GL[\cdot]$ and $GR[\cdot]$ are used. Only $SL[\cdot]$ and $SR[\cdot]$ were used in the static case.) When a path π, connecting u and v, is removed from the system, it is also removed from the priority queue $P[u, v]$. Removing π from $P[u, v]$ takes $O(\log n)$ time.

Lemma 3.8 *A call* `remove-paths`(π) *correctly removes all extensions of π. The resulting system is well-maintained. The running time is $O(|DEL| \log n)$, where DEL is the number of paths removed.*

Theorem 3.9 *Suppose that* `build-paths` *is run on a well-maintained collection of paths that includes $\pi[e]$, for every $e \in E$, and possibly other generated and selected paths. If all shortest paths are unique, then at the end of the run we have:*

1. *All shortest paths are selected.*
2. *All* newly *selected paths are shortest paths.*
3. *For every $u, v \in V$, $p[u, v]$ is the shortest path from u to v in the graph.*

The running time of `build-paths` *is $O(|NEW|+n^2 \log n)$, where NEW is the set of newly generated paths.*

Theorem 3.10 *A call* `vertex-update`(v, E_{new}) *recomputes all shortest paths after an update of vertex v. The running time used is $O((|DEL| + n^2) \log n + |NEW|)$, where DEL is the set of generated paths passing through v before the update, and NEW is the set of newly generated paths.*

Note that while all paths in DEL pass through v, the updated vertex, the paths in NEW do not necessarily pass through v.

How large can DEL and NEW be?

3.4 Increasing updates only

If all updated are increasing, things are particularly nice. All selected paths are shortest paths and all generated paths are locally shortest paths.

Lemma 3.11 *Suppose that all selected paths before an increasing update are shortest paths. Then, the same holds after the update. In particular, all generated paths, before and after the update are locally shortest paths.*

Lemma 3.12 *The total running time of k increasing updates is $O(mn + kn^2 \log n)$, where m is the number of edges in the graph after the updates.*

Proof: Since all updates are increasing, all selected paths are shortest and all generated paths are locally shortest. Let d_i be the number of locally shortest paths *destroyed* by the i-th update operation, and let c_i be the number of locally shortest paths *created* (generated) by the i-th update operation. By Theorem 3.10, the running time of the i-th update is $O((d_i + n^2) \log n + c_i)$, and the total running time is $O((kn^2 + \sum_{i=1}^{k} d_i) \log n + \sum_{i=1}^{k} c_i)$.

All paths destroyed by the i-th update are locally shortest paths passing through v_i. By Lemma 3.6, there are at most $3n^2$ such paths, and therefore $d_i = O(n^2)$. All that remains, therefore, is to bound $\sum_{i=1}^{k} c_i$.

The number of LSPs after the k update operations is at least $\sum_{i=1}^{k} c_i - \sum_{i=1}^{k} d_i$. By Lemma 3.5, the number of LSPs after the k-th update is at most mn. Thus $\sum_{i=1}^{k} c_i \leq mn + \sum_{i=1}^{k} d_i$. As we have seen, $d_i = O(n^2)$. Thus, $\sum_{i=1}^{k} c_i = O(mn + kn^2)$. Putting everything together, we get that the time required to process the k updates is indeed $O(mn + kn^2 \log n)$. $\square$

3.5 Decreasing updates only

4 General updates

Definition 4.1 (Historical paths) *Let π be a path in the graph in time t. Let $t' \leq t$ be the last time a vertex on π was updated. Then, π is said to be* historical *at time t if it has been a shortest path at least once during the time interval $[t', t]$.*

Note that a path can stop being a historical path only as a result of an update of one of its vertices.

Definition 4.2 (Locally historical paths) *A path π at time t is said to be a* locally historical path *if and only if it consists of a single vertex, or if $l[\pi]$ and $r[\pi]$ are both historical paths at time t.*

Lemma 4.3 *If, at some time, there at most z historical paths between any pair of vertices in the graph, then there are at most zmn locally historical paths.*

Lemma 4.4 *If, at some time, there at most z historical paths between any pair of vertices in the graph, then there are at most $O(zn^2)$ locally historical paths passing through a given vertex v.*

Let $v[t]$ be the vertex updated at time $t \geq 0$.

A *dummy* update of a vertex v is a call to `update`(v) without changing any edge weights.

Theorem 4.5 *Suppose that after performing the requested update on vertex $v[t]$ at time t, we perform dummy updates on vertices $v[t-1], v[t-2], \ldots, v[t-2^i] \ldots$. Then, the number of historical paths between any pair of vertices in the graph is $O(\log t)$.*

Lemma 4.6 *Suppose that π_1 and π_2 are historical paths from x and y at time t. Let t_1 and t_2 be the times of the last (non-dummy) updates on π_1 and π_2, respectively. Then, $t_1 \neq t_2$.*

Lemma 4.7 *Suppose that π_1 and π_2 are historical paths from x and y at time t. Let t_1 and t_2 be the times of the last (non-dummy) updates on π_1 and π_2. Assume that $t_1 < t_2$. Then, if $t_1 + 2^j \leq t$, then $t_1 + 2^j < t_2$. In particular, $t - t_2 < \frac{t-t_1}{2}$.*

5 Getting rid of the uniqueness assumption

Assign each edge e a distinct tag $tag[e]$. All edges that participate in an update are assigned tags that are larger than all previous tags. If π is a path, we let $TAG[\pi]$ be the set of the tags of the edges appearing on π, and $tag[\pi]$ be the maximum tag of an edge appearing on π. We can easily add the maintenance of $tag[\pi]$ to the constructors $\texttt{path}(e)$ and $\texttt{path}(\pi_1, \pi_2)$.

We next define a total order on sets of numbers. Let A and B be two distinct sets of numbers. We say that $A \prec B$ if and only if $\max(A - B) < \max(B - A)$. (In words: the largest element in A that does not belong to B is smaller than the largest element of B that does not belong to A.) We say that $A \preceq B$ if and only if $A = B$ or $A \prec B$.

We next define a total order of paths. We say that $\pi_1 \prec \pi_2$ if and only if $cost[\pi_1] < cost[\pi_2]$ or $cost[\pi_1] = cost[\pi_2]$ and $TAG[\pi_1] \prec TAG[\pi_2]$.

We say that a path π from u to v in the graph is *optimal* if and only if $\pi \prec \pi'$ for every other path from u to v in the graph. If there is a path from u to v in the graph, then there is a unique optimal path from u to v. Also note that a subpath of an optimal path must also be optimal.

We change $\texttt{path}()$ such that it also maintains $tag[\pi]$ for every π. (But not $TAG(\pi)$. That would be too time consuming!). The key associated with each path in $P[u, v]$ and each pair (u, v) in Q is now the pair $(cost[\pi], tag[\pi])$, and comparisons are done lexicographically.

Lemma 5.1 *If up to a certain point in time all selected paths are optimal at the time of their selection, then up to that point*

1. *if π_1 and π_2 are two* selected *paths between u and v, then $cost[\pi_1] \neq cost[\pi_2]$ and $tag[\pi_1] \neq tag[\pi_2]$.*
2. *if π_1 and π_2 are two* generated *paths between u and v, then $cost[\pi_1] \neq cost[\pi_2]$ or $tag[\pi_1] \neq tag[\pi_2]$.*

Theorem 5.2 *The algorithm finds all the optimal paths in the graph.*

References

[DI04] C. Demetrescu and G.F. Italiano. A new approach to dynamic all pairs shortest paths. *Journal of the ACM*, 51(6):968–992, 2004.

[Dij59] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[FT87] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[KKP93] D.R. Karger, D. Koller, and S.J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22:1199–1217, 1993.

[McG95] C.C. McGeoch. All-pairs shortest paths and the essential subgraph. *Algorithmica*, 13:426–441, 1995.

[Tho04] M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *Proc. of 9th SWAT*, pages 384–396, 2004.