A faster algorithm for the single source shortest path problem with few distinct positive lengths

J. B. Orlin, K. Madduri, K. Subramani, and M. Williamson

Journal of Discrete Algorithms **8** (2010) 189–198.

Speaker: Joseph, Chuang-Chieh Lin
Supervisor: Professor Maw-Shang Chang

Computation Theory Laboratory
Department of Computer Science and Information Engineering
National Chung Cheng University, Taiwan

January 31, 2010

Outline

Outline

1. Introduction
2. Basic terminology
3. Reviewing Dijkstra's algorithm for SSSPP
4. An $O(m + nK)$ implementation of Dijkstra's algorithm
5. A faster algorithm if K is permitted to grow with the problem size

The single source shortest path problem (SSSPP)

Given a graph $G = (V, E)$ and $s \in V$ designated as the source, where each edge $(u, v) \in E$ has a positive length c_{uv}, determine the shortest path from s to each $v \in V$ in G.

Assume that $|V| = n$ and $|E| = m$.

Previous results

- $O(m + n \log n)$ time by **Fibonacci Heap implementation**.
  - Fredman and Tarjan (1987); *J. ACM*.
- $O(m + n\frac{\log n}{\log \log n})$ time by the **Atomic Heap implementation** (in a slightly different model of computation).
  - Fredman and Willard (1994); *J. Comput. Sys. Sci.*

The contributions of this paper

♠ Input: a graph $G = (V, E)$ with $|V| = n$, $|E| = m$, and K distinct edge lengths.

- Efficient methods for implementing Dijkstra's algorithm for SSSPP parameterized by K.
  - An $O(m + nK)$ algorithm ($O(m)$ if $nK \leq 2m$);
  - An $O(m \log \frac{nK}{m})$ algorithm for the case that $nK > 2m$.
- Experimental results.
  - Demonstration of the superiority of their approach when K is small (except for dense graphs).

Motivation

- The “gossip” problem for social networks.
- For example, consider a social network composed of clusters of participants.
  - We model the intra-cluster distance by 1 and the inter-cluster distance by $p > 1$.
  - **Goal:** determine a faster manner where gossip originating in a cluster can reach all the participants in the social network.

Outline

Preliminaries & notations

- $\emptyset$: empty set; $\varnothing$: nothing.
- $E_{out}(v)$: the set of edges directed out of v.
- $\delta(v)$: the length of the shortest path in G from s to v.
  - $\delta(v) = \infty$ if there is no path from s to v.

Preliminaries & notations (contd.)

- $L = \{\ell_1, \ell_2, \ldots, \ell_K\}$: the set of distinct nonnegative edge lengths in increasing order (stored in an array).
  - $\forall (i,j) \in E$, (i,j) has an edge length $c_{ij} \in L$.
- Assumption: $(i,j) \Leftrightarrow t_{ij}$.
  - $c_{ij} = \ell_{t_{ij}}$ (i.e., $t : E \mapsto \{1, 2, \ldots, K\}$).
  - This can be done in $O(m + K \log K)$ time.

Outline

Dijkstra's algorithm for SSSPP

```
Dijkstra(G, w, s)
for each v ∈ V(G) do
    d[v] ← ∞; pred[v] ← ∅;
end for
d[s] ← 0;
Q ← V(G); S ← ∅;
while (Q ≠ ∅) do
    u ← argmin{d[v] : v ∈ Q}
    if d[u] = ∞ then
        break;
    end if
    remove u from Q; S ← S ∪ {u};
    for each v ∈ Adj[u] do
        temp ← d[u] + c_uv;
        if temp < d[v] then
            d[v] ← temp;
            pred[v] ← u;
        end if
    end for
end while
```

Dijkstra's algorithm for SSSPP (modularized)

```
RELAX(u, v, c)
    temp ← d[u] + c_uv;
    if temp < d[v] then
        d[v] ← temp;
        pred[v] ← u;
    end if
```

```
INITIALIZE(G, s)
    for each v ∈ V(G) do
        d[v] ← ∞;
        pred[v] ← ∅;
    end for
    d[s] ← 0;
```

```
Dijkstra(G, c, s)
    INITIALIZE(G, s);
    S ← ∅;
    Q ← V(G);
    while (Q ≠ ∅) do
        u ← EXTRACT-MIN(Q);
        if d[u] = ∞ then
            break;
        end if
        S ← S ∪ {u};
        for each v ∈ Adj[u] do
            RELAX(u, v, c);
        end for
    end while
```

Improvements by implementing priority queues

- A series of EXTRACT-MIN() and DECREASE-KEY() is performed in Dijkstra's algorithm.
- The running time of Dijkstra's algorithm can be represented as $T(n, m) = n \times \texttt{EXTRACT-MIN()} + m \times \texttt{DECREASE-KEY()}$.

	EXTRACT_MIN()	DECREASE-KEY()
linked list:	$O(1)$	$O(n)$
binary heap:	$O(\log n)$	$O(\log n)$
Fibonacci Heap:	$O(\log n)$ (amortized)	$O(1)$ (amortized)

Outline

Further notations

- We maintain the following structures:
  - S: the set of permanently labeled vertices;
  - $T = V \setminus S$: the set of temporarily labeled vertices.
- $d(j)$: the distance label of vertex j.
  - If $j \in S$, then $d(j) = \delta(j)$.
- $d^* = \max\{d(j) : j \in S\}$:
  - the distance label of the vertex most recently added to S.
- $\texttt{FIND-MIN}()$: identifying $\min\{d(v) : v \in T\}$.
  - $\texttt{EXTRACT-MIN}() = \texttt{FIND-MIN}() +$ Deletion of $\operatorname{argmin}\{d(v) : v \in T\}$ from T.

Further notations (contd.)

- Recall that $L = \{\ell_1, \ell_2, \ldots, \ell_K\}$: the set of K distinct edge lengths.
- For each $1 \leq t \leq K$, $E_t(S) = \{(i,j) \in E : i \in S, c_{ij} = \ell_t\}$.
  - If (i,j) occurs prior to edge (i',j') on $E_t(S)$, then $d(i) \leq d(i')$.
- $\texttt{CurrentEdge}(t)$: the first edge $(i,j) \in E_t(S)$ such that $j \in T$.
  - $\texttt{CurrentEdge}(t) = \varnothing$ if no such edge exists.
  - If $\texttt{CurrentEdge}(t) = (i,j)$, then let $f(t) = d(i) + \ell_t$.
    - $f(t)$: the length of the shortest path from s to i followed by edge (i,j).
    - Note here that NOT NECESSARY that $f(t) = d(j)$.

Further notations (contd.)

- `UPDATE`(t): moving the pointer `CurrentEdge`(t) so that it points to the first edge whose endpoint is in T (or set `CurrentEdge`$(t) = \varnothing$).
  - If `CurrentEdge`$(t) = (i, j)$, then `UPDATE`(t) sets $f(t) = d(i) + c_{ij}$.
  - If `CurrentEdge`$(t) = \varnothing$, then `UPDATE`(t) sets $f(t) = \infty$.

An $O(m + nK)$ implementation of Dijkstra's algorithm

NEW-DIJKSTRA()
- INITIALIZE();
- **while** $(T \neq \emptyset)$ **do**
  - $r \leftarrow \text{argmin}\{f(t) : 1 \leq t \leq K\}$;
  - $(i, j) \leftarrow$ CurrentEdge(r);
  - $d(j) \leftarrow d(i) + \ell_r$; pred$(j) \leftarrow i$;
  - $S \leftarrow S \cup \{j\}$; $T \leftarrow T \setminus \{j\}$;
  - **for** (each edge $(j, k) \in E_{out}(j)$) **do**
    - Add (j, k) to the end of $E_t(S)$, where $\ell_t = c_{jk}$;
    - **if** (CurrentEdge$(t) = \varnothing$) **then**
      - CurrentEdge$(t) \leftarrow (j, k)$;
    - **end if**
  - **end for**
  - **for** ($t \leftarrow 1$ to K) **do**
    - UPDATE(t);
  - **end for**
- **end while**

An $O(m + nK)$ implementation of Dijkstra's algorithm

```
INITIALIZE()
    S ← {s}; T ← V \ {s};
    d(s) ← 0; pred(s) ← ∅;
    for (each v ∈ T) do
        d(v) ← ∞; pred(v) ← ∅;
    end for
    for t ← 1 to K do
        E_t(S) ← ∅;
        CurrentEdge(t) ← ∅;
    end for
    for each edge (s, j) do
        Add (s, j) to the end of E_t(S),
        where ℓ_t = c_sj;
        if (CurrentEdge(t) = ∅) then
            CurrentEdge(t) ← (s, j);
        end if
    end for
    for (t ← 1 to K) do
        UPDATE(t);
    end for
```

```
UPDATE(t)
    (i, j) ← CurrentEdge(t);
    if (j ∈ T) then
        f(t) ← d(i) + c_ij;
        return;
    end if
    while ((j ∉ T) and
    (CurrentEdge(t).next ≠ ∅)) do
        (i, j) ← CurrentEdge(t).next;
        CurrentEdge(t) ← (i, j);
    end while
    if (j ∈ T) then
        f(t) ← d(i) + c_ij;
    else
        CurrentEdge(t) ← ∅;
        f(t) ← ∞;
    end if
```

Time complexity

- Initialization: $O(n)$
- Computing $r = \text{argmin}\{f(t) : 1 \leq t \leq K\}$ over all iterations: $O(nK)$.
- Total time needed for $\texttt{UPDATE}(t)$: $O(m + nK)$.
  - Suppose that $(i, j) \leftarrow \texttt{CurrentEdge}(t)$.
  - ⋆ $O(nK)$ if $\texttt{CurrentEdge}(t)\texttt{.next}$ is never used.
  - ⋆ Otherwise, $O(m)$.
    - $\because$ (i, j) is never scanned again after updating CurrentEdge(t).

Outline

When K is not a constant...

- Let $q = \frac{nK}{m}$.
- If $q < 2$, previous algorithm runs in $O(m)$ time.
- Assume that $q \geq 2$.
- To simplify the discussion, let $h = \frac{K}{q}$.
- **Goal:** compute $r = \text{argmin}\{f(t) : 1 \leq t \leq K\}$ more efficiently and call $\texttt{UPDATE}(t)$ less frequently.

Revision of the previous algorithm

- Store the values $f()$ in a collection of h different binary heaps $H_1, H_2, \ldots, H_h$.
  - H_1 stores $f(j)$ for $1 \leq j \leq q$;
  - H_2 stores $f(j)$ for $q+1 \leq j \leq 2q$;

    $\vdots$

- `FIND-MIN()` in H_i: $O(1)$ time.
  - `FIND-MIN()` takes $O(hn) = O(m)$ time overall.
- Insert/Delete an element into H_i: $O(\log q)$ time.
  - Deletions after `FIND-MIN()` takes $O(n \log q)$ time overall.

Revision of the previous algorithm (contd.)

- Relax the requirement on `CurrentEdge`.
  - If $(i, j) \leftarrow$ `CurrentEdge`(t) we obtain that $i, j \in S$:
    - We say that `CurrentEdge`(t) is *invalid*.
  - `CurrentEdge`(t) is permitted to be invalid at some intermediate stages of the algorithm.

Revision of the previous algorithm (contd.)

- We modify `FIND-MIN()` as follows.
  - If the minimum element in heap H_i is $f(t)$ for some i and if `CurrentEdge`(t) is invalid, perform `UPDATE()`, followed by:
    - Finding the new minimum element in H_i until it corresponds to a valid edge.
  - Whenever the algorithm calls `UPDATE()`, it leads to such a modification of `CurrentEdge()`.
  - Whenever the algorithm selects the minimum element among the q heaps, the minimum element in each heap corresponds to a valid edge.
- Since there are $\leq m$ modifications of `CurrentEdge()`, the total running time for `UPDATE()` overall is $O(m \log q)$.

Theorem 5.1

The binary heap implementation of Dijkstra's algorithm with $O\left(\frac{K}{q}\right)$ binary heaps of size $O(q)$ with $q = \frac{nK}{m}$ determines the shortest path from s to all other vertices in $O(m \log q)$ time.

Interested audience may refer to Chapter 7 of the paper for experimental results.

My daughter, Sherry

Thank you!