Shortest Paths

Jonathan Backer
backer@cs.ubc.ca

Department of Computer Science
University of British Columbia

June 24, 2007

Introduction

Reading:

- CLRS: "Single-Source Shortest Paths" 24 (except 24.4)
- GT: "Single-Source Shortest Paths" 7.1

Given a weighted graph, we define the cost of a path as the sum of weights between consecutive path vertices. We explore two different approaches to finding all of the shortest paths from a given source vertex.

The first algorithm is similar to Prim's algorithm and is greedy. The second algorithm uses dynamic programming algorithm (our next topic).

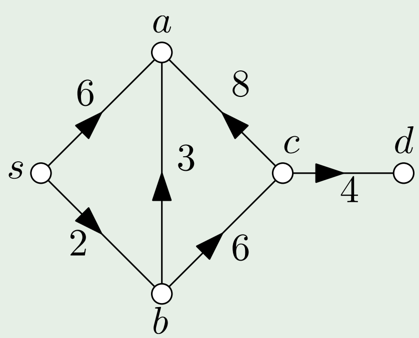
Directed graphs

Edges of a directed graph have direction and can only be traversed one way.

- An edge $(u, v) \in E$ from u to v is an ordered pair.
- In particular, $(u, v) \neq (v, u)$.

A path from u to v is a sequence of vertices $u = x_0, x_1, \ldots, x_t = v$ where $(x_{i-1}, x_i) \in E$, for $1 \leq i \leq t$.

Example

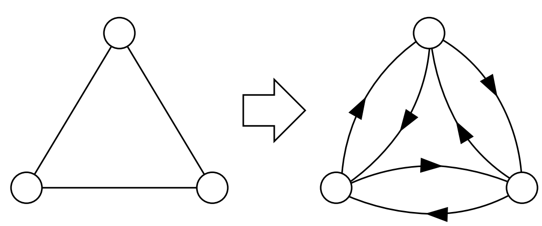


Vertices are intersections and edges are one-way streets. The weight of an edge is the street length or the expected travel time.

What are the shortest paths from s?

Directed vs. undirected graphs

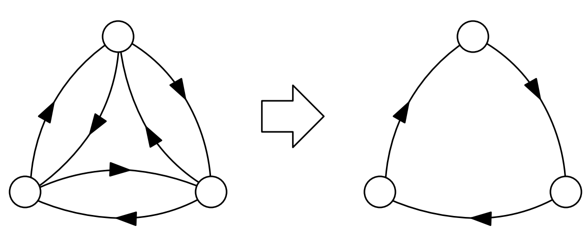
Every undirected graph has a directed counterpart.



Problems differ on directed graphs

- Cycle guarantees connectivity in directed graphs.
- Tree guarantees connectivity in undirected graphs.

The solution to the directed counterpart is not necessarily a solution to the undirected original.



Single source shortest paths

Problem

Given a weighted (directed) graph G and a source vertex s, find the shortest paths from s to every other vertex of G.

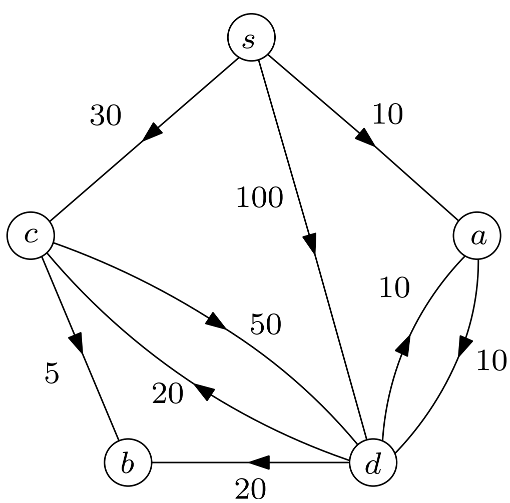
Important properties:

- No vertex is visited twice on a shortest path.
- The prefix of a shortest path is a shortest path.

Outline of Dijkstra's algorithm:

- Grow a shortest path tree rooted at s and directed from s
- Track the cost of the shortest path to other vertices using just vertices in tree (plus the destination).
- Repeatedly add the vertex that is cheapest to reach from the tree.

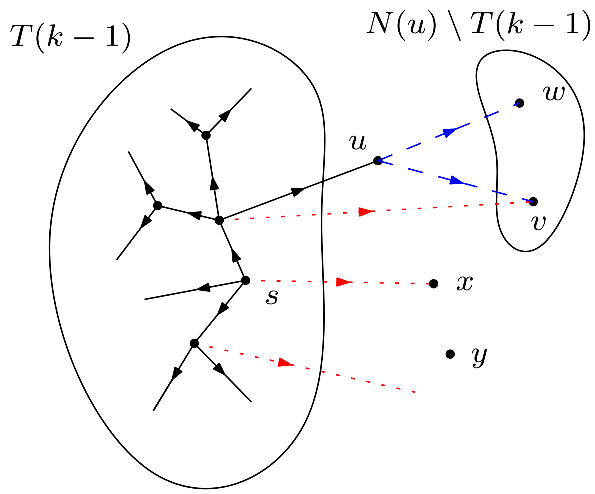
Dijkstra's example



vertex	s	a	b	c	d	Tree
costs	0	∞	∞	∞	∞	$\emptyset$
	-	10,s	∞	30,s	100,s	$\{s\}$
	-	-	∞	30,s	20,a	$\{s, a\}$
	-	-	40,d	30,s	-	$\{s, a, d\}$
	-	-	35,c	-	-	$\{s, a, c, d\}$

Efficient cost update

How do we update the costs once we add a vertex to the shortest path tree?



Suppose u is added to get $T(k)$.

- Is it cheaper to reach vertices outside of $T(k-1)$ by going through u?
- Update neighbours of u that aren't in $T(k-1)$ (e.g. v and w).

- Other vertices are unaffected (e.g. x and y).

To find u efficiently, we keep $V \setminus T(k-1)$ in a heap.

Initialization

- *cost*$[v]$ is the cost of the shortest path from s to v.
- *prev*$[v]$ is used for path recovery — it indicates what edge was used to get the minimum *cost*$[v]$.

```
Algorithm Dijkstra(V, E, s)
   for v ∈ V do
      tree[v] ← false
      if (v = s) then
         cost[v] = 0
      else
         cost[v] = ∞
      Q.insert(v,cost[v])
      prev[v] = ∅
```

Dijkstra's algorithm: greedy loop

```
for k ← 1 to |V| do
    v ← Q.deleteMin()
    tree[v] ← true
    if v = s then
        T ← ∅
    else
        T ← T ∪ {(v, prev[v])}
    for each (v, w) ∈ E do
        if tree[w] = false and
            cost[w] > cost[v] + w((v, w))
        then
            prev[w] ← v
            cost[w] ← cost[v] + w((v, w))
            Q.updatePriority(w, cost[w])
```

Run-time complexity

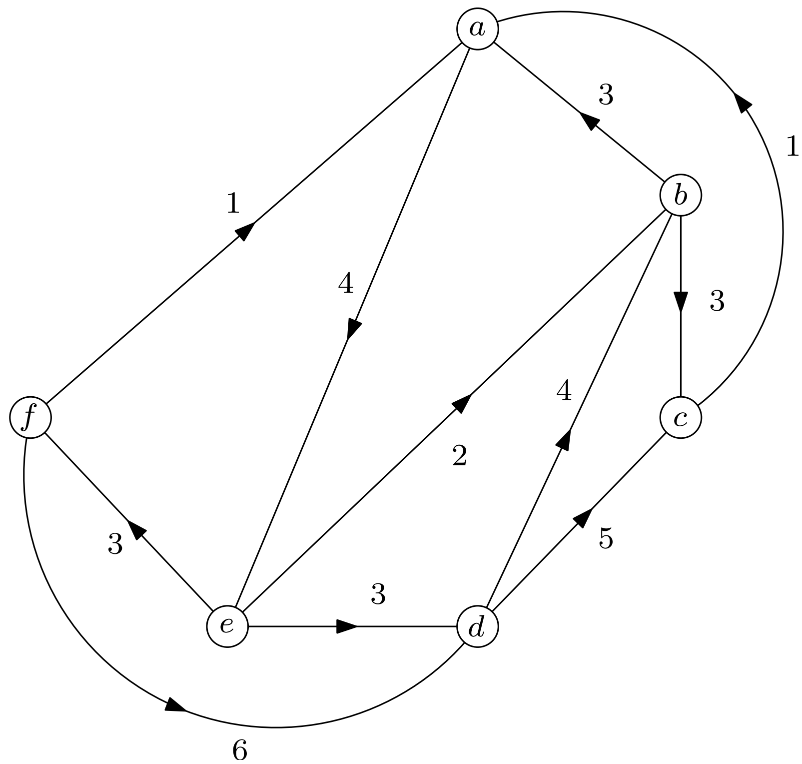
We count the priority queue operations because they are inside each loop and are the only non-constant time operations.

- Adding vertices to the queue:
  $|V| \times O(\log |V|) = O(|V| \log |V|)$
- Removing the minimum vertex from the queue:
  $|V| \times O(\log |V|) = O(|V| \log |V|)$
- Updating the priority of a vertex in the queue (at most once for each edge):
  $|E| \times O(\log |V|) = O(|E| \log |V|)$

Total cost: $O([|V| + |E|] \log |V|)$

Another example

Find the shortest paths from d.



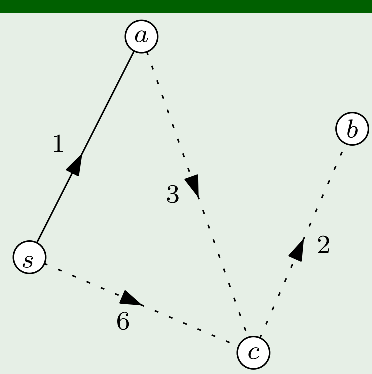
Notation

Let $T(k)$ be the shortest path tree with k vertices built after k adding k vertices.

Let $SP(T, v)$ be the shortest cost path from s to v in the subgraph T of G.

- We explicitly allow $v \notin T$.
- In this case, we include the edges of G from T to v.

Example



Let T be the tree with the solid edges. Then

- $SP(T, a) = 1$,
- $SP(T, c) = 4$, and
- $SP(T, b) = \infty$.

Correctness

Proof

We inductively prove the following about *cost* after k iterations.

$$cost[v] = \begin{cases} SP(G, v) & \text{if } v \in T(k) \\ SP(T(k), v) & \text{otherwise} \end{cases}$$

Base case (k=1): Result of initialization

$$cost[v] = \begin{cases} 0 & = SP(G, v) & \text{if } v = s \\ \infty & = SP(T(k), v) & \text{otherwise} \end{cases}$$

Induction step:

Assume that the hypothesis holds after k iterations. Suppose u is added on the $(k+1)$th iteration. We now look at $cost[v]$ for every $v \in V$ after the updates.

Correctness (cont'd)

Proof (cont'd)

1. $v \in T(k)$

   After the kth iteration, $cost[v] = SP(G, v)$. It is unchanged by the $(k+1)$th iteration.
2. $v \notin T(k)$ and $v \neq u$

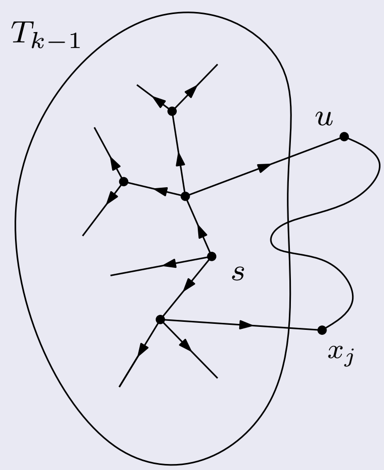
   If $(u, v) \in E$, then $cost[v]$ is properly updated and equals $SP(T(k+1), v)$. Otherwise $cost[v]$ is not updated, which is correct because $SP(T(k), v) = SP(T(k+1), v)$.
3. $v \notin T(k)$ and $v = u$

   Then $cost[u]$ is not updated, so it suffices to show that $SP(T(k), u) = SP(G, u)$.

Correctness (cont'd)

Proof (cont'd)

Let $s = x_1, x_2, \ldots, x_i = u$ be a shortest s, u-path in G. Consider the smallest j such that $x_j \notin T(k)$. If $j = i$, then $SP(T(k), u) = SP(G, u)$.



So suppose $i \neq j$. Then by the greedy choice of u instead of x_j

$$\begin{aligned} SP(T(k), u) &\leq SP(T(k), x_j) \\ &= SP(G, x_j). \\ \text{But } SP(G, x_j) &\leq SP(G, u). \end{aligned}$$

Hence $SP(T(k), u) \leq SP(G, u)$ by transitivity. $\square$

Bellman-Ford algorithm

- Works with negative edge weights!
- Our first dynamic programming algorithm.
  - Divide-and-conquer breaks problems into subproblems (top-down).
  - Dynamic programming combines subproblems into problems (bottom-up).
- If there is an negative cost cycle, there is no shortest path.
- If no negative cost cycles, a shortest path visits each vertex.
  - Each shortest path uses at most $|V| - 1$ edges.
- Bellman-Ford iteratively finds shortest paths using at most $1, 2, 3, \ldots, |V| - 1$ edges.

Pseudo-code

```
Algorithm Bellman-Ford(V, E, s)
    for v ∈ V do
        if (v = s) then cost[v] = 0
        else cost[v] = ∞
        prev[v] = ∅

    for k ← 1 to |V| − 1 do
        for (u, v) ∈ E do
            if cost[u] ≠ ∞ and
                cost[v] > cost[u] + w((u, v))
            then
                cost[v] ← cost[u] + w((u, v))
                prev[v] ← u

    for (u, v) ∈ E do
        if cost[u] ≠ ∞ and
            cost[v] > cost[u] + w((u, v))
        then throw new Exception("negative cycle!")
```

Correctness

Proof

Clearly the algorithm finds paths and calculates their costs correctly.

To prove correctness, we argue inductively that after the kth iteration of the "`for` k" loop that

- $cost[v]$ is no larger than the length of the shortest s, v-path that has at most k edges.

Base case (k=0): We can only reach s. Property holds by initialization.

Inductive step: Assume that it holds for k. The shortest s, v-path using at most $(k+1)$ edges must have been a shortest s, u-path using at most k edges plus a shortest u, v-path using 1 edge, for some u. After considering every edge, we have found u and updated $cost[v]$ and $prev[v]$ accordingly. □