Analysis of Algorithms, I
CSOR W4231.002

Eleni Drinea
Computer Science Department

Columbia University

Thursday, March 5, 2015

Outline

1. Recap
2. Strongly connected components in directed graphs via depth-first search
3. Shortest paths in graphs with non-negative edge weights (Dijkstra's algorithm)
   - Correctness
   - Implementations

Today

1. Recap
2. Strongly connected components in directed graphs via depth-first search
3. Shortest paths in graphs with non-negative edge weights (Dijkstra's algorithm)
   - Correctness
   - Implementations

Review of the last lecture

- Depth-first search (DFS)
- Classification of graph edges in the DFS forest
- Applications
  1. Cycle detection
  2. Topological sorting

Today

1. Recap
2. Strongly connected components in directed graphs via depth-first search
3. Shortest paths in graphs with non-negative edge weights (Dijkstra's algorithm)
   - Correctness
   - Implementations

Classifying graph edges via DFS

Fact 1.

Edge $(u, v) \in E$ *is a back edge in a DFS tree if and only if*

$$start(v) < start(u) < finish(u) < finish(v).$$

Fact 2.

If $(u, v) \in E$ *is a forward edge in a DFS tree, then*

$$start(u) < start(v) < finish(v) < finish(u).$$

Fact 3.

If $(u, v) \in E$ *is a cross edge in the DFS forest, then*

$$start(v) < finish(v) < start(u) < finish(u).$$

Exploring the connectivity of a graph

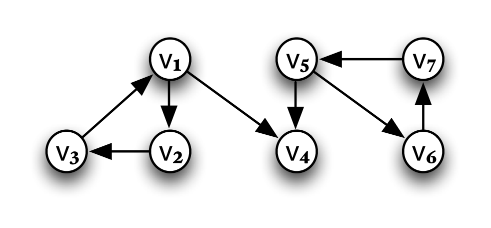
- **Undirected** graphs: find all connected components
- **Directed** graphs: find all strongly connected components (SCCs)
  - SCC(u) = set of nodes that are reachable from u and have a path back to u
  - SCCs provide a hierarchical view of the connectivity of the graph:
    - on a top level, the meta-graph of SCCs has a useful and simple structure *(coming up)*;
    - each meta-vertex of this graph is a fully connected subgraph that we can further explore.

How can we find SCC(u) using BFS?

1. Run `BFS`(u); the resulting tree T consists of the set of nodes to which there is a path **from** u
2. Define G^r as the **reverse** graph, where edge (i, j) becomes edge (j, i)
3. Run `BFS`(u) in G^r; the resulting BFS tree T' consists of the set of nodes that have a path **to** u
4. The common vertices in T, T' compose the strongly connected component of u.

What if we want all the SCCs of the graph?

The meta-graph of SCCs of a directed graph

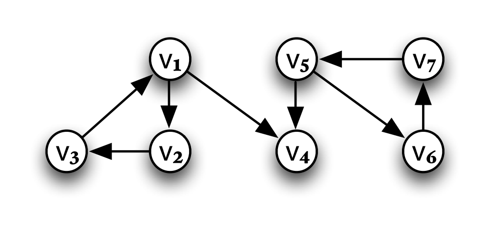


Consider the meta-graph of all SCCs of G.

- Make a (super)vertex for every SCC;
- add a (super)edge from SCC C_i to SCC C_j if there is an edge from some vertex u of C_i to some vertex v of C_j.

What kind of graph is the meta-graph of SCC's?

Is there an SCC we could process first?



Suppose we had a sink SCC of G.

What will DFS discover if it starts at some node of this SCC?

1. *How do we find a node that for sure lies in a sink SCC?*
2. *How do we continue to find all other SCCs?*

Easier to find a node in a *source* SCC!

Fact 4.

The node assigned the largest finish time when we run `DFS`(G) *belongs to a source SCC in* G.

Proof.

Let G be a directed graph. The meta-graph of its SCCs is a DAG. For an SCC C, let

$$finish(C) = \max_{v \in C} finish(v)$$

Lemma 5.

Let C_i, C_j *be SCCs in* G. *Suppose there is an edge* $(u, v) \in E$ *such that* $u \in C_i$ *and* $v \in C_j$. *Then* $finish(C_i) > finish(C_j)$.

□

G^r is useful again

- Fact 4 provides a direct way to find a node in a source SCC of G: pick the node with largest $finish$.
- But we want a node in a sink SCC of G!
- Consider G^r, the graph where the edges of G are reversed. *How do the SCCs of G and G^r compare?*
- Run DFS on G^r: the node with the largest $finish$ comes from a source SCC of G^r (Fact 4). That is a sink SCC of G!

Using this observation to find all SCCs

We know how to find a sink SCC in G:

1. run $\texttt{DFS}(G^r)$; compute $finish$ times
2. run DFS(G) starting from the node with the largest $finish$; the resulting tree T is a sink SCC in G.

How do we find all remaining SCCs?

Remove T from G: the remaining graph G' is a DAG, hence has at least one sink SCC.

Algorithm for finding SCCs in directed graphs

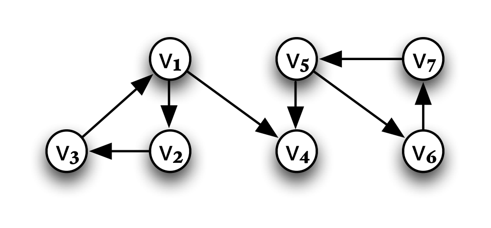
$\texttt{SCC}(G = (V, E))$

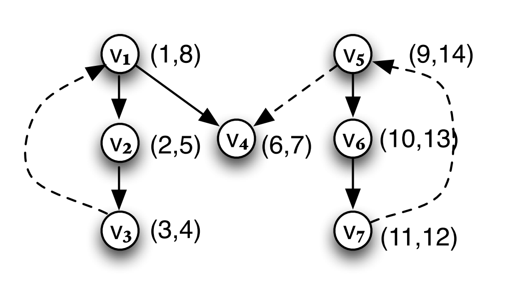
1. Compute G^r
2. Run $\texttt{DFS}(G^r)$; compute $finish(u)$ for all u
3. Run $\texttt{DFS}(G)$ in decreasing order of $finish(u)$
4. Output the vertices of each tree in the DFS forest of line 3 as an SCC.

Remark 1.

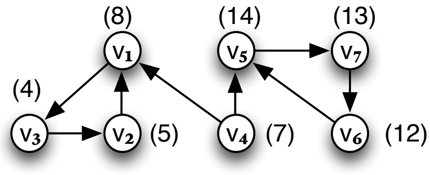
- *Running time: $O(n + m)$ —why?*
- *Equivalently, we can (i) run $\texttt{DFS}(G)$, compute $finish$ times; (ii) run $\texttt{DFS}(G^r)$ by decreasing order of $finish$. Why?*

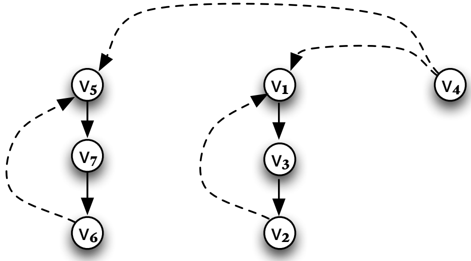
A directed graph and its DFS forest with time intervals





DFS forest of G^r; nodes are considered by decreasing $finish$





Still need to prove Lemma 5

Let G be a directed graph. The meta-graph of its SCCs is a DAG.

For an SCC C, let

$$finish(C) \quad = \quad \max_{v \in C} finish(v)$$

Lemma 6.

Let C_i, C_j be SCCs in G. Suppose there is an edge $(u, v) \in E$ such that $u \in C_i$ and $v \in C_j$. Then $finish(C_i) > finish(C_j)$.

Proof of Lemma 5

There are two cases to consider:

1. $start(u) < start(v)$ (DFS starts at C_i)
   - Before leaving u, DFS will explore edge (u, v).
   - Since $v \in C_j$, all of C_j will now be explored.
   - Since there is no edge from C_j back to C_i (DAG!), all vertices in C_j will be assigned $finish$ times before DFS backtracks to u and assigns a $finish$ time to u. Thus

$$finish(C_j) < finish(u) \leq finish(C_i)$$

Proof of Lemma 5 (cont'd)

2. $start(u) > start(v)$ (DFS starts at C_j)

   Since there is no edge from C_j to C_i, DFS will finish exploring C_j before it restarts from some vertex that will result in discovery of C_i. Thus

$$
\begin{aligned}
& finish(C_j) < start(u) < finish(u) \\
\Rightarrow\ & finish(C_j) < finish(C_i)
\end{aligned}
$$

Today

1. Recap
2. Strongly connected components in directed graphs via depth-first search
3. Shortest paths in graphs with non-negative edge weights (Dijkstra's algorithm)
   - Correctness
   - Implementations

Weighted graphs

- Edge weights represent distances (or time, cost, etc.)
- Consider a path $P = (v_0, \ldots, v_k)$. The weight of P is the sum of the weights of its edges:

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}).$$

- In weighted graphs, a shortest path from u to v is a path of minimum weight among all u-v paths, that is

$$distance(u, v) = \begin{cases} \min\limits_{P} \quad w(P) & \text{, if exists } u\text{-}v \text{ path} \\ \infty & \text{, otherwise} \end{cases}$$

Single-source shortest-paths problem

Input

- a weighted, directed graph $G = (V, E, w)$; function $w : E \to R$ maps edges to real-valued weights;
- a source (origin) vertex $s \in V$.

Output: for every vertex $v \in V$

- the length (weight) of a shortest s-v path;
- a shortest s-v path.

Given an algorithm A for single-source shortest-paths:

We can also solve

- **single-pair shortest-path problem**
- **single-destination shortest-paths problem:** find a shortest path from every vertex to a destination t
- **all-pairs shortest-paths**: find a shortest path between every pair of vertices

Shortest paths in graphs with non-negative weights

Single-source shortest-paths with non-negative edge weights:

Input

- a weighted, directed graph $G = (V, E, w)$; function $w : E \to R^+$ maps edges to non-negative real-valued weights;
- a source (origin) vertex $s \in V$.

Output: for every vertex $v \in V$

- the length (weight) of a shortest s-v path;
- a shortest s-v path.

Dijkstra's algorithm (Input: $G = (V, E, w), s \in V$)

Output: arrays $dist$, $prev$ with n entries such that

- $dist(v)$ stores the length of the shortest s-v path
- $prev(v)$ stores the node before v in the shortest s-v path

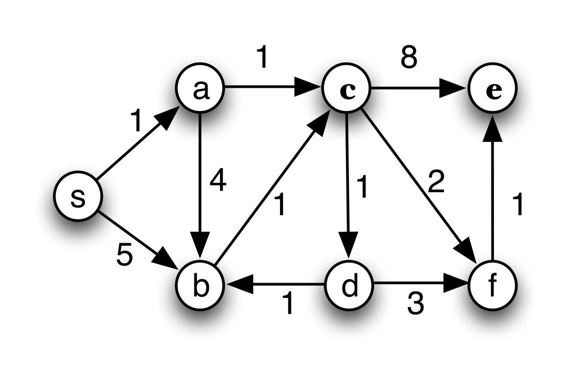
At all times, maintain a set S of nodes for which the distance from s has been determined.

- Initially, $dist(s) = 0$, $S = \{s\}$
- Each time, add to S the node $v \in V - S$ that
  1. has an edge from some node in S;
  2. minimizes the following quantity among all nodes $v \in V - S$

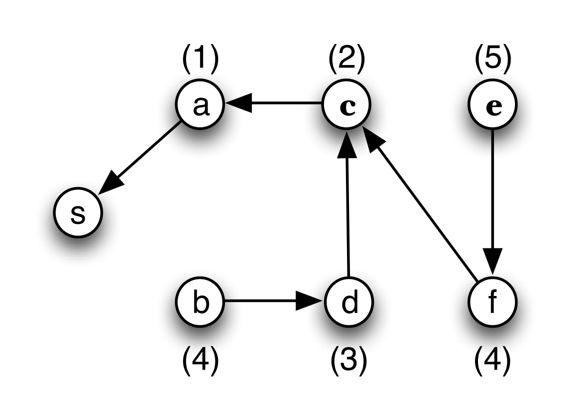
$$d(v) = \min_{u \in S:(u,v) \in E} \{dist(u) + w(u, v)\}$$

- Set $prev(v) = u$.

An example weighted directed graph



Dijkstra's output for example graph



The distances (in parentheses) and reverse shortest paths.

Another way of showing optimality of greedy algorithms

Greedy principle: a local decision rule is applied at every step.

- Dijkstra's algorithm is **greedy**: always form the shortest new s-v path by first following a path to some node u in S, and then a single edge (u, v).
- Proof of optimality: it *always stays ahead of any other solution*; when a path to a node v is selected, that path is shorter than every other possible s-v path.

Correctness of Dijkstra's algorithm

At all times, the algorithm maintains a set S of nodes for which it has determined a shortest-path distance from s.

Claim 1.

Consider the set S at any point in the algorithm's execution. For each u in S, the path P_u is a shortest s-u path.

Optimality of the algorithm follows from the claim (*why?*).

Proof of Claim 1

By induction on the size of S.

- **Base case**: $|S| = 1$, $dist(s) = 0$.
- **Hypothesis**: suppose the claim is true for $|S| = k$: for every $u \in S$, P_u is a shortest s-u path.
- **Step**: let v be the $k+1$-st node added to S. We want to show that P_v, which is P_u for some $u \in S$, followed by (u, v), is a shortest s-v path.
  Consider any other s-v path, call it P. P must leave S somewhere since $v \notin S$: let $y \neq v$ be the first node of P in $V - S$ and $x \in S$ the node before y in P. Since v was added in this iteration of the algorithm, it must be that $d(v) \leq d(y)$. So just the subpath $s \rightarrow x \rightarrow y$ in P is longer than P_v! Hence P is longer as well (*why?*).

Implementation

`Dijkstra-v1`$(G = (V, E, w), s \in V)$
  `Initialize`(G, s)
  $S = \{s\}$
  while $S \neq V$ **do**
    Select a node $v \in V - S$ with at least one edge from S so that
$$d(v) = \min_{u \in S, (u,v) \in E} \{dist[u] + w(u, v)\}$$
    $S = S \cup \{v\}$
    $dist[v] = d(v)$
    $prev[v] = u$
  end while

`Initialize`(G, s)
  for $v \in V$ **do**
    $dist[v] = \infty$
    $prev[v] = NIL$
  end for
  $dist[s] = 0$

Improved implementation (I)

Idea: Keep a conservative overestimate of the true shortest s-u path in $dist[u]$. Update $dist[v]$ for all v with $(u, v) \in E$.

`Dijkstra-v2`$(G = (V, E, w), s \in V)$

  `Initialize`(G, s)
  $S = \emptyset$
  while $S \neq V$ **do**
    Pick u so that $dist[u]$ is minimum among all nodes in $V - S$
    $S = S \cup \{u\}$
    for $(u, v) \in E$ **do**
      `Update`(u, v)
    end for
  end while

`Update`(u, v)

  if $dist[v] > dist[u] + w(u, v)$ **then**
    $dist[v] = dist[u] + w(u, v)$
    $prev[v] = u$
  end if

Improved implementation (II): binary min-heap

Idea: Use a priority queue implemented as a binary min-heap: store vertex u with key $dist[u]$. Required operations: `Insert`, `ExtractMin`; `DecreaseKey` for `Update`; each takes $O(\log n)$ time.

```
Dijkstra-v3(G = (V, E, w), s ∈ V)
   Initialize(G, s)
   Q = {V; dist}
   S = ∅
   while Q ≠ ∅ do
      u = ExtractMin(Q)
      S = S ∪ {u}
      for (u, v) ∈ E do
         Update(u, v)
      end for
   end while
```

Running time: $O(n \log n + m \log n) = O(m \log n)$

When is `Dijkstra-v3()` *better than* `Dijkstra-v2()`*?*