Analysis of Algorithms, I
CSOR W4231

Eleni Drinea
Computer Science Department

Columbia University

Strongly connected components,
single-origin shortest paths in weighted graphs

Outline

1. Applications of DFS
   - Strongly connected components
2. Shortest paths in graphs with non-negative edge weights (Dijkstra's algorithm)
   - Correctness
   - Implementations

Finding your way in a maze

Depth-first search (DFS): starting from a vertex s, explore the graph as deeply as possible, then backtrack

1. Try the first edge out of s, towards some node v.
2. Continue from v until you reach a dead end, that is a node whose neighbors have all been explored.
3. Backtrack to the first node with an unexplored neighbor and repeat 2.

Remark: DFS answers s-t connectivity

Directed graphs: classification of edges

DFS constructs a forest of trees.

Graph edges that do not belong to the DFS tree(s) may be

1. forward: from a vertex to a *descendant* (other than a *child*)
2. back: from a vertex to an *ancestor*
3. cross: from right to left (no ancestral relation), that is
    - from tree to tree
    - between nodes in the same tree but on different branches

On the time intervals of vertices u, v

If we use an explicit stack, then

- $start(u)$ is the time when u is pushed in the stack
- $finish(u)$ is the time when u is popped from the stack (that is, all of its neighbors have been explored).

Intervals $[start(u), finish(u)]$ and $[start(v), finish(v)]$ either

- contain each other (u is an ancestor of v or vice versa); or
- they are disjoint.

Classifying edges using time

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

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

2. Edge $(u, v) \in E$ is a forward edge if

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

3. Edge $(u, v) \in E$ is a cross edge if

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

Today

1. Applications of DFS
   - Strongly connected components
2. Shortest paths in graphs with non-negative edge weights (Dijkstra's algorithm)
   - Correctness
   - Implementations

Exploring the connectivity of a graph

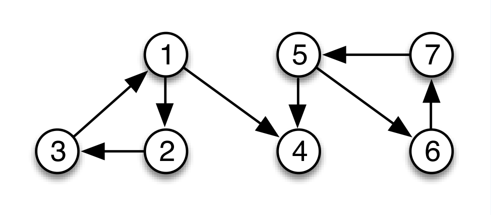
- **Undirected** graphs: find all connected components
- **Directed** graphs: find all strongly connected components (SCCs)
  - SCC(u) = set of nodes that are reachable from u and have a path back to u
  - SCCs provide a hierarchical view of the connectivity of the graph:
    - on a top level, the meta-graph 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 $\texttt{BFS}(u)$; 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 $\texttt{BFS}(u)$ 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

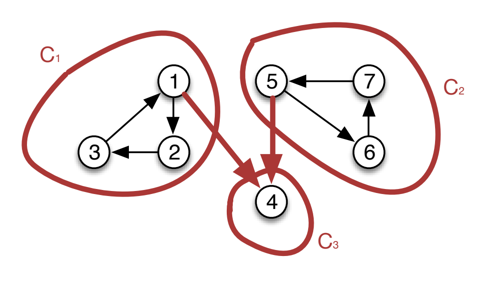


Consider the meta-graph of all SCCs of G.

- 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?

The meta-graph of SCCs of a directed graph

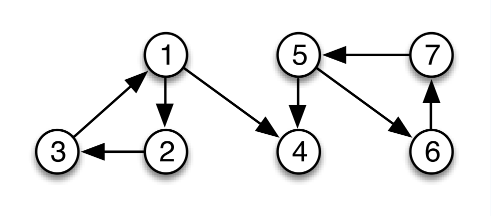


Consider the meta-graph of all SCCs of G.

- 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.

This graph is a DAG.

Is there an SCC we could process first?



Suppose we had a sink SCC of G, that is, an SCC with no outgoing edges.

1. *What will DFS discover starting at a node of a sink SCC?*
2. *How do we find a node that for sure lies in a sink SCC?*
3. *How do we continue to find all other SCCs?*

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

Fact 1.

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

Example: v_5 belongs to source SCC C_2.

Proof.

We will use Lemma 2 below. Let G be a directed graph. The meta-graph of its SCCs is a DAG. For an SCC C, let

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

Example: $finish(C_1) = finish(v_1) = 8$.

Lemma 2.

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 1 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 1). This is a sink SCC of G!

Using this observation to find all SCCs

We now know how to find a sink SCC in G.

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

How do we find all remaining SCCs?

- Remove T from G; let G' be the resulting graph.
- The meta-graph of SCCs of G' is a DAG, hence it has at least one sink SCC.
- Apply the procedure above recursively on G'.

Algorithm for finding SCCs in directed graphs

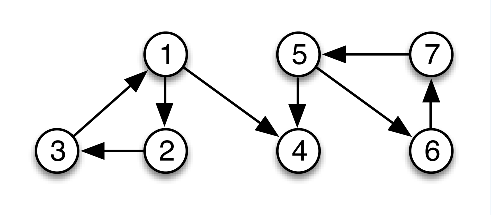
$\texttt{SCC}(G = (V, E))$

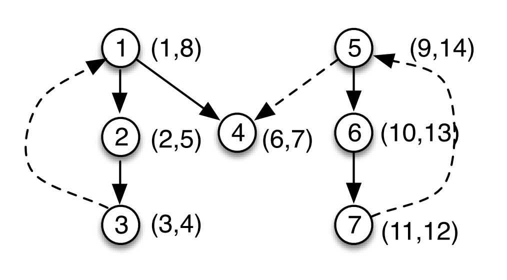
1. Compute G^r.
2. Run $\texttt{DFS}(G^r)$; compute $finish(u)$ for all u.
3. Run $\texttt{DFS}(G)$ in decreasing order of $finish(u)$.
4. Output the vertices of each tree in the DFS forest of line 3 as an SCC.

Remark 1.

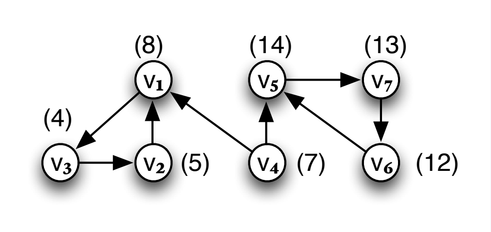
1. Running time: $O(n + m)$ —*why?*
2. Equivalently, we can (i) run $\texttt{DFS}(G)$, compute $finish$ times; (ii) run $\texttt{DFS}(G^r)$ by decreasing order of $finish$. *Why?*

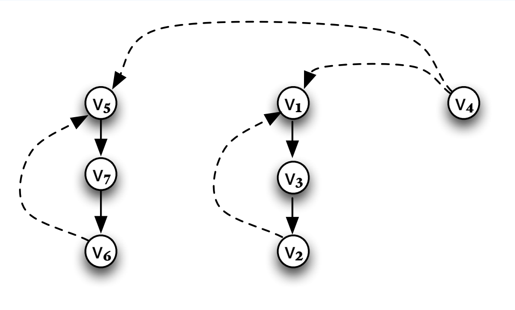
A directed graph and its DFS forest with time intervals





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





Still need to prove Lemma 2

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 3.

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 2

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.
   - 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 2 (cont'd)

2. $start(u) > start(v)$

   Since there is no edge from C_j to C_i (DAG!), DFS will finish exploring C_j before it discovers u. Thus

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

Today

1. Applications of DFS
   - Strongly connected components
2. 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 **length** 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** length among all paths from u to v.

Notation

- s-t path: a path from s to t.
- $dist(s,t)$: the length of the shortest s-t path;

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

- $dist(t)$: the length of the shortest s-t path, when s is fixed.
- We will refer to $w(P)$ as the **weight** or **cost** or **length** of P.

Single-origin (source) shortest-paths problem

Input:

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

Output: for every vertex $v \in V$

1. the length of a shortest s-v path;
2. a shortest s-v path.

Given an algorithm A for **single-origin** 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

Graphs with **non-negative** weights

Input

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

Output: for every vertex $v \in V$

1. the length of a shortest s-v path;
2. 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

1. $dist[v]$ = length of the shortest s-v path
2. $prev[v]$ = node before v on the shortest s-v path

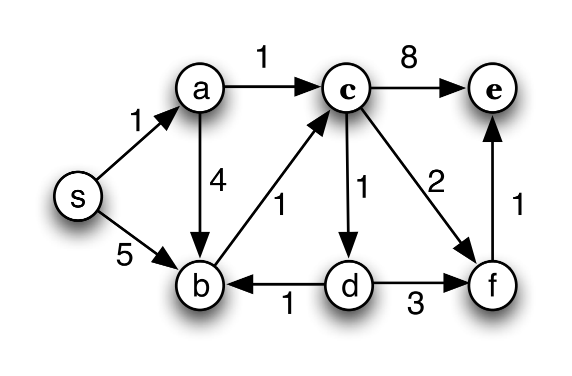
At all times, maintain a set S of nodes for which the distance from s has been determined.

- 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 $v \in V - S$

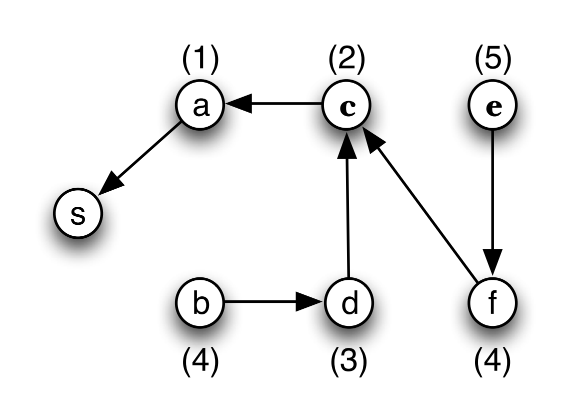
$$d(v) = \min_{u \in S:(u,v) \in E} \{dist[u] + w_{uv}\}$$

- 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 $|S| = k$, that is, 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 the edge (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 the algorithm added v in this iteration and not y, it must be that $d(v) \leq d(y)$. So just the subpath $s \to x \to y$ in P is at least as long as P_v! Hence so is P (*why?*).

Implementation

`Dijkstra-v1`$(G = (V, E, w), s \in V)$

  `Initialize`(G, s)
  $S = \{s\}$
  while $S \neq V$ **do**
    For every $x \in V - S$ with at least one edge from S compute

$$d(x) = \min_{u \in S, (u,x) \in E} \{dist[u] + w_{ux}\}$$

    Select v such that $d(v) = \min_{x \in V - S} d(x)$
    $S = S \cup \{v\}$
    $dist[v] = d(v)$
    $prev[v] = u$
  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 length of the shortest s-v path in $dist[v]$ as follows: when u is added to S, 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 $dist[v] > dist[u] + w_{uv}$ **then**
    $dist[v] = dist[u] + w_{uv}$
    $prev[v] = u$
  end if

Priority queues and binary heaps

- **Priority queue**: a priority queue is a data structure for maintaining a set S of n elements, each with an associated value called a *key*.

- *Operations* supported by a **min-priority queue** Q:
  1. `BuildQueue`$(\{S; keys\})$: builds a min-priority queue
  2. `Insert`(Q, x): insert element x into Q
  3. `Extract-min`(Q): extract the minimum element from Q
  4. `Decrease-key`(Q, x, k): decrease the *key* for x to a new (smaller) value k

- We can implement a min-priority queue as a **binary min-heap**. Then each of the four operations above requires time $O(n), O(\log n), O(\log n), O(\log n)$ respectively.

  See Chapter 6 in your textbook for more details on binary heaps.

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 = BuildQueue({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()`*?*

Further implementations of Dijsktra's algorithm

Notation: $|V| = n$, $|E| = m$

Implementation	`ExtractMin`	`Insert/` `DecreaseKey`	Time
Array	$O(n)$	$O(1)$	$O(n^2)$
Binary heap	$O(\log n)$	$O(\log n)$	$O((n+m)\log n)$
d-ary heap	$O(\log n)$	$O(\log n)$	$O((nd+m)\frac{\log n}{\log d})$
Fibonacci heap	$O(\log n)$	$O(1)$ amortized	$O(n\log n + m)$

- Optimal choice is $d \approx m/n$ (the *average* degree of the graph)
- d-ary heap works well for both sparse and dense graphs
  - *If* $m = n^{1+x}$, *what is the running time of Dijsktra's algorithm using a* d*-ary heap?*