Analysis of Algorithms, I

CSOR W4231

Eleni Drinea
Computer Science Department

Columbia University

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

Outline

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

Recall: We can find all the connected components of an undirected graph with either BFS or DFS.

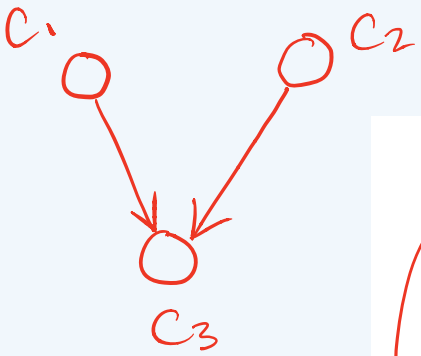
- **Undirected** graphs: find all connected components
- **Directed** graphs: find all strongly connected components (SCCs)
  - SCC(u) = set of nodes that are reachable from u and have a path back to u
  - SCCs provide a hierarchical view of the connectivity of the graph:
    - on a top level, the meta-graph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

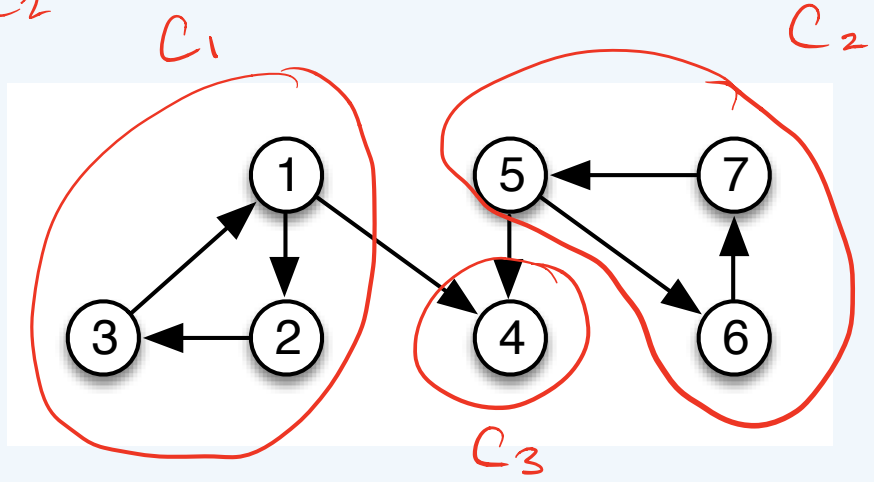
How can we find SCC(u) using BFS?

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





2 source SCCs (C_1 and C_2)

1 sink SCC (C_3)

Consider the meta-graph of all SCCs of G.

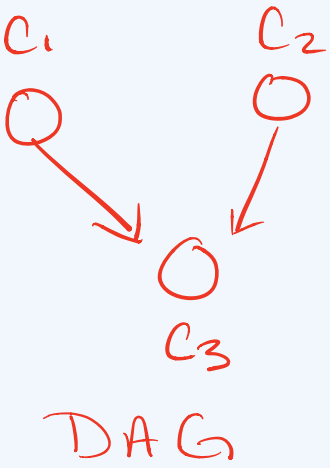
- Make a (super)vertex for every SCC.
- Add a (super)edge from SCC C_i to SCC C_j if there is an edge from some vertex u of C_i to some vertex v of C_j.

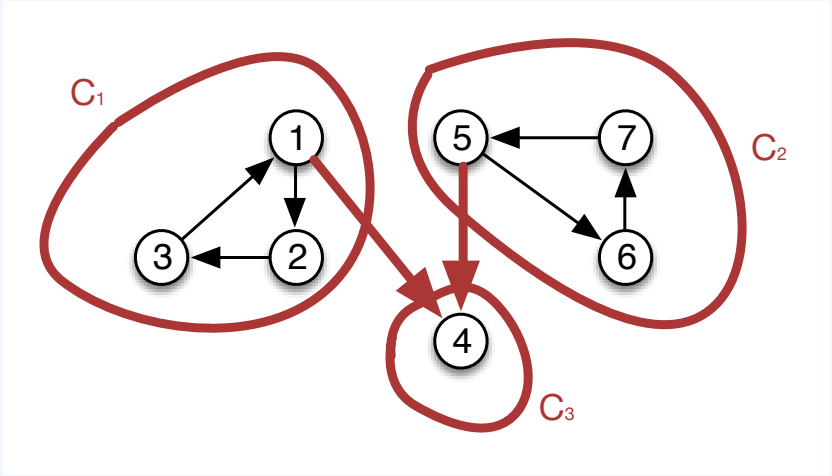
What kind of graph is the meta-graph of SCC's? DAG

The meta-graph of SCCs will always be a DAG.

The meta-graph of SCCs of a directed graph

The metagraph of SCCs is always a DAG.





and we know that every DAG has at least one source and at least one sink.

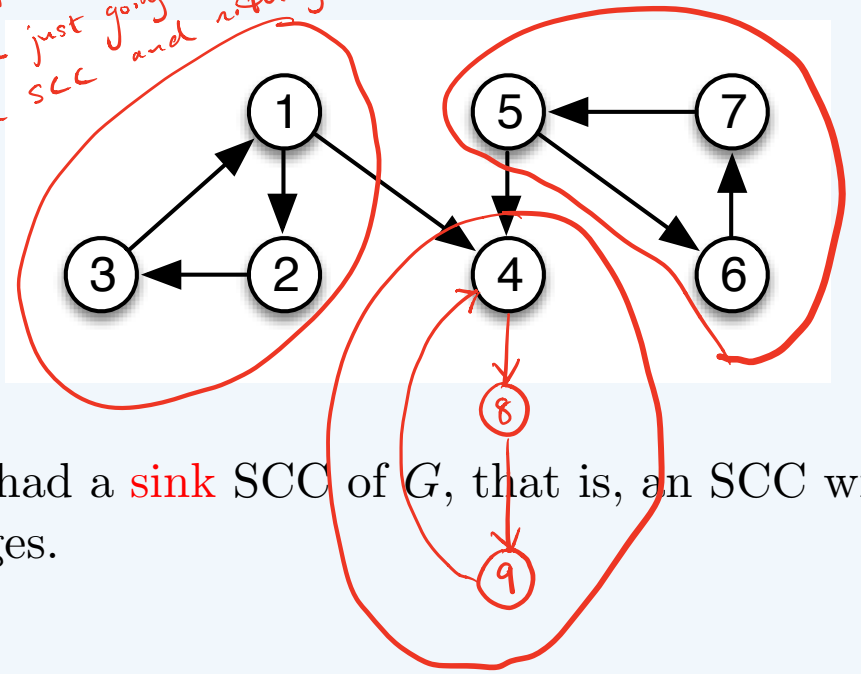
Consider the meta-graph of all SCCs of G.

- 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?

If you run DFS from a sink SCC, you're just going to discover the sink SCC and nothing else.



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?* Just the 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?*

→ Remove the sink SCCs, recursively.

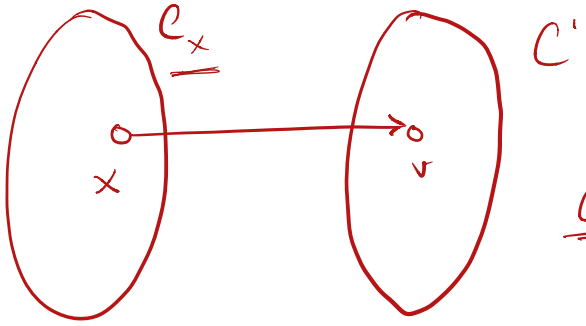
1. *What will DFS discover starting at a node of a sink SCC?*

A: the sink SCC and only the sink SCC

2. *How do we find a node that for sure lies in a sink SCC?*

First, consider how to find node in a source SCC:

- We can find quickly a node that for sure lies in a source SCC.
  - Run DFS in G.
  - The node v with largest finish time must belong to a source SCC.



Pf: Suppose v has the largest finish time. There are two cases:

Case I: C_x was discovered before C'. In this case, we will discover C' since there is a path from x to v.
⇒ The start time and finish time of any node in C will be less than that of C_x because DFS will back-track back to C_x.
⇒ Impossible that v has the largest finish time.

Case II: C_x discovered after C.

The start time of any node in C_x would be greater than the finish time of any node in C. Therefore, it is impossible that v is the node w/ the largest finish time.

The Finish time of Connected Component:

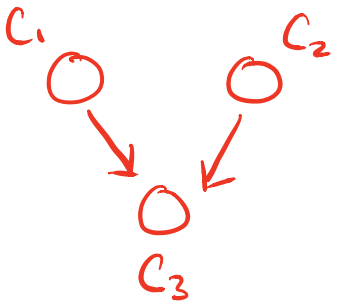
- $f(C) \triangleq \max_{u \in C} f(u)$

How do we find a node that for sure lies in a sink SCC?

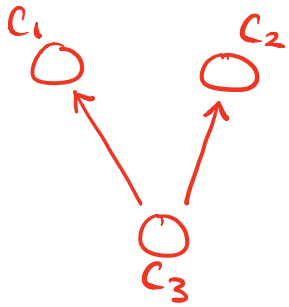
Use the reverse graph, G^r.

Source SCCs in G become sink SCCs in G^r, and sink SCCs in G become source SCCs in G^r.

Meta-graph G:



G^r:

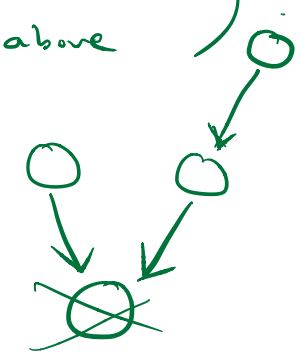


1. Run DFS on G^r

2. Compute finish times for all nodes.

$\Rightarrow$ nodes v with max finish time in G^r $\in$ sink SCC in G.

- Run DFS on G from node v.
  DFS Search (G, v)
- Output C (the nodes we found from search procedure above)



Remove a sink SCC

the metagraph of SCCs remains a DAG

$\Rightarrow$ still a sink SCC in the metagraph of of SCCs.

Looking for a new sink SCC in a metagraph.

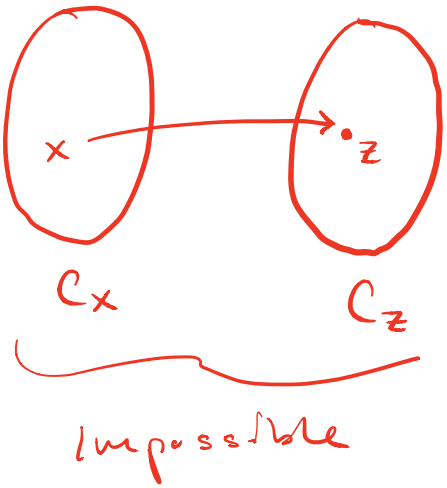
- Let $V_1 = V - C$
- Let G_1 be the induced graph (the graph that has vertices in V_1)
- There exists another source SCC in G_1. (since G_1 and G_1^r are DAGs).

Search for node w/ next largest finish time. Claim: Node w/ next largest finish time belongs to a source SCC in G_1^r.

Why is that?

Suppose z is the node w/ the next largest finish time; The only reason it wouldn't be is because there is some other component C_x w/ edge going from C_x to C_z. This implies the finish time of C_x is greater than the finish time of C_z. ⨳ A contradiction.

G_1^r:



Impossible

Therefore, this component is a source component in G_1^r and therefore a sink component of G_1

Now you can restart DFS from that node

- The node z w/ the largest finish time in G_1^r belongs to a source SCC in G_1^r.

Apply this procedure repeatedly to extract all SCCs:

$\Rightarrow$ nodes v with max finish time in G^r $\in$ sink SCC in G.

- Run DFS on G from node v.
  DFS Search (G, v)
- Output C_1 (the nodes we found from search procedure above)
- Let $V_1 = V - C_1$
- Let G_1 be the induced graph (the graph that has vertices in V_1)
- Exists another source SCC in G_1^r.
- The node z w/ largest finish time in G_1^r belongs to a source SCC in G_1^r.

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

Fact 1.

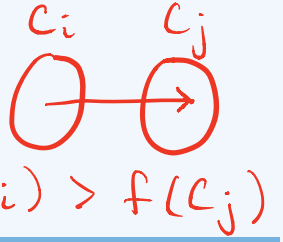
The node assigned the largest finish time when we run DFS(G) *belongs to a source SCC in* G.

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 `DFS`(G^r); compute $finish$ times.
2. Run `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

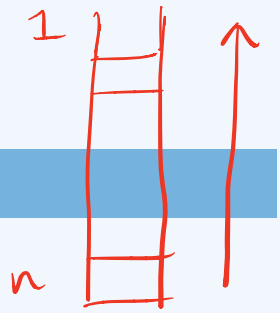
$\texttt{SCC}(G = (V, E))$

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

2x DFS

Not computing finish times in step 3 since done in step 2.

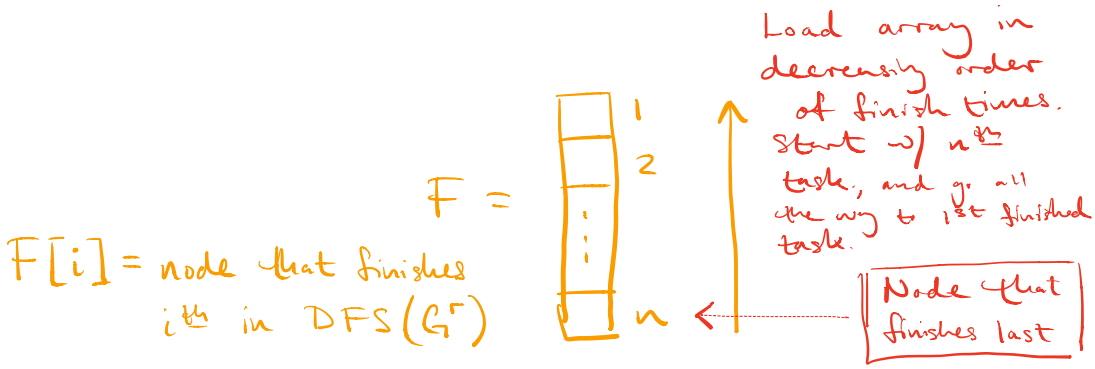
FINISH TIMES



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?*

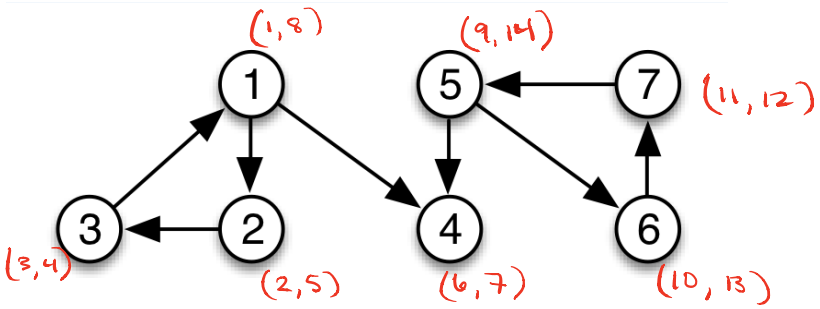
Storing finish times the same way as done for topological sort using DFS:



F[i] = node that finishes ith in DFS(G^r)

Doing it this way, you don't have to sort.

Example:

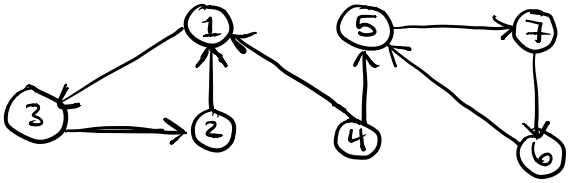


F =

3	1
2	2
4	3
1	4
7	5
6	6
5	7

Recall F[i] = node that finishes i^{th} in DFS

G^r:

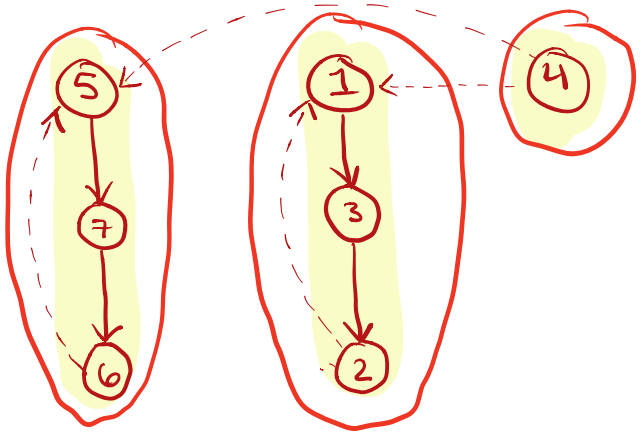


Now have to run DFS in G^r in decreasing order of finish times.

F =

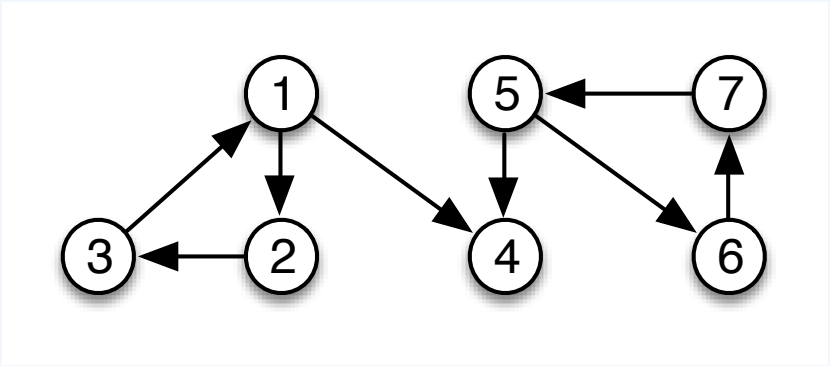
	1
	2
	3
	4
	5
	6
	7

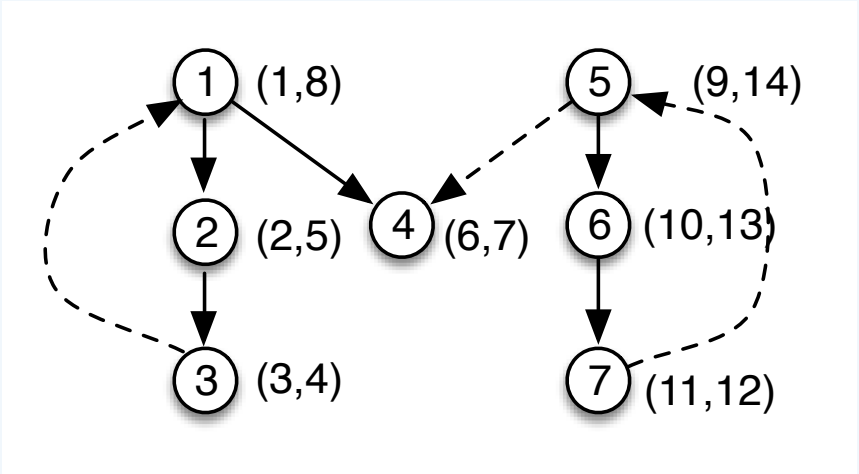
DFS(G^r ...):



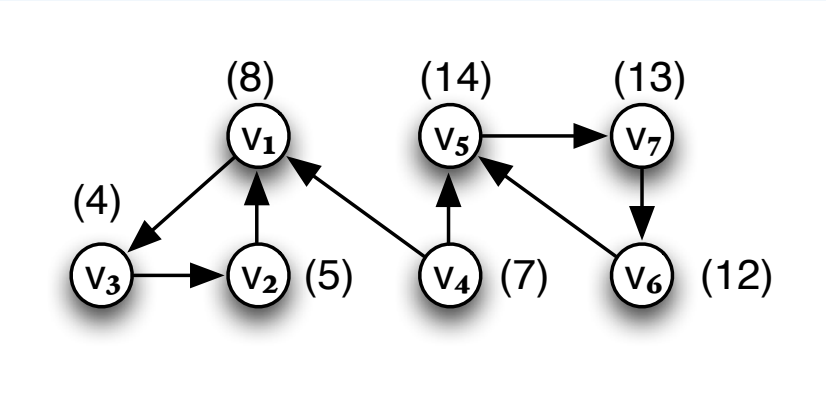
SCC_1 SCC_2 SCC_3

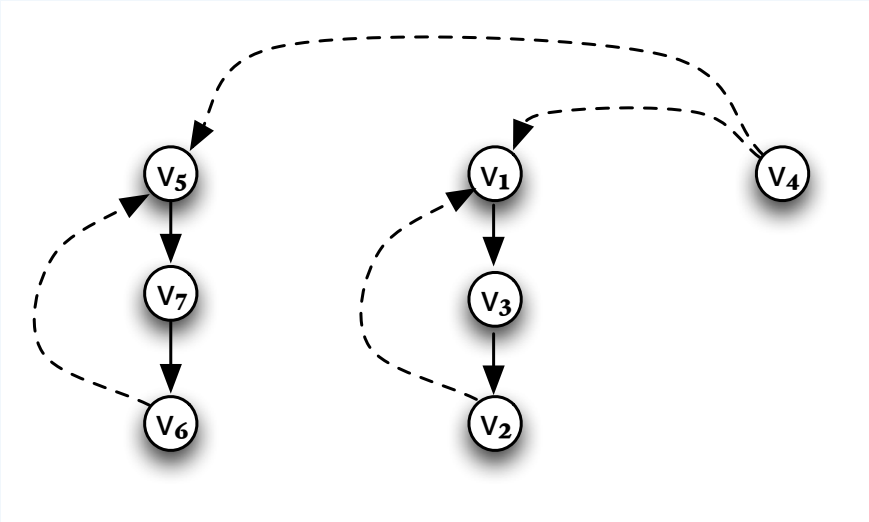
A directed graph and its DFS forest with time intervals





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

$$finish(C_j) < start(u) < finish(u)$$
$$\Rightarrow finish(C_j) < finish(C_i)$$

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

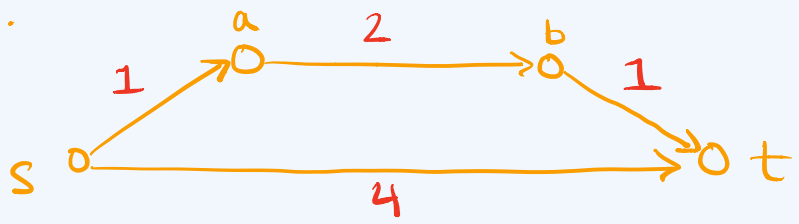
- In unweighted graphs we can use BFS to find shortest paths in linear time.

  - 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: or weight or cost

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

  - In weighted graphs, a shortest path from u to v is a path of **minimum** length among all paths from u to v.

Ex:



Two shortest paths here

Notation

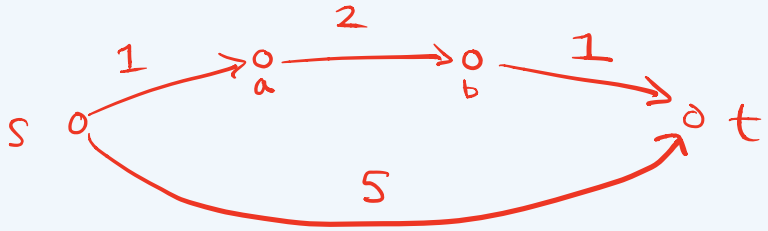
If undirected, an s-t path is a path between s and t.

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

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

- $dist(t)$: 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.

Ex:



- $dist(s,t) = 4$
- (s, a, b, t)
- shortest s-t path. cost = 4

In this context, weight ≡ length ≡ cost are used interchangably. They all mean: $\sum_{e \in P} w_e$ (P = path)

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.

For single-origin shortest-path problems:

The output is an array dist[v] that gives the shortest path from s to every vertex $v \in V$.
and the length of the shortest path.

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

$A(G, s)$: shortest path from s to every vertex in G.

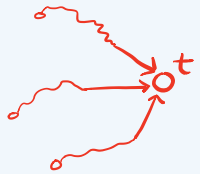
We can also solve

1. ▸ **single-pair** shortest-path problem → Run $A(G, s)$ with $G = (V, E, w)$ for pair (s, t)
   ↳ same complexity as single-origin.
2. ▸ **single-destination** shortest-paths problem: find a shortest path from every vertex to a destination t
   ↳ Run A on the reverse graph G^r & origin t.
3. ▸ **all-pairs** shortest-paths: find a shortest path between every pair of vertices

Run $A(G, s)$ computing shortest paths from s to all nodes including s to t.

(s, t)

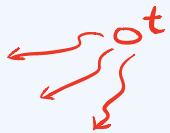
G



2. Run $A(G^r, t)$

reverse graph — origin

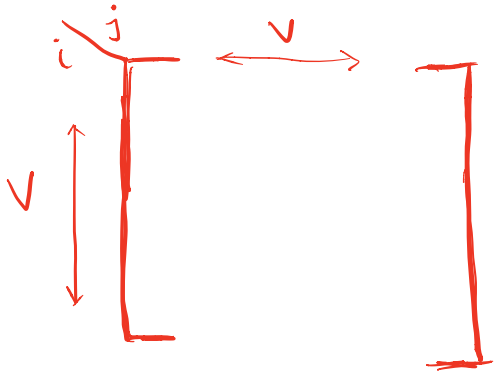
G^r:



gives the shortest paths from t to all nodes in the reverse graph G^r

The complexity of this is the same as complexity of A plus complexity to reverse the graph, which takes linear time.

3. The output for this problem is a matrix.. Indeed, for each origin node, the single-origin shortest path problem outputs an array dist[v].



To solve this Run single-origin for every vertex.

Running Time: $n \cdot$ (time complexity of A in G)

- We will be able to improve on this run-time by using a specialized algorithm on this problem.

(Running $A(G, v)$ for all $v \in V$)

Graphs with **non-negative** weights

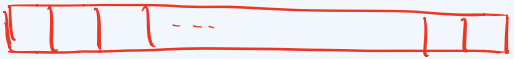
Input

- 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.

dist

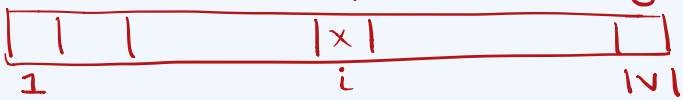


→ store in an array dist[i] = distance of node i from s.

Can use a linked list or dictionary. However, a simple array will suffice since we know the # of nodes in the graph.

Use back-pointers w/ another array

prev



prev[i] = node preceding node i in the shortest s-i path

s-i path: $s \longrightarrow \dots \longrightarrow prev[x] \longrightarrow prev[i] \longrightarrow i$ (x)

prev array

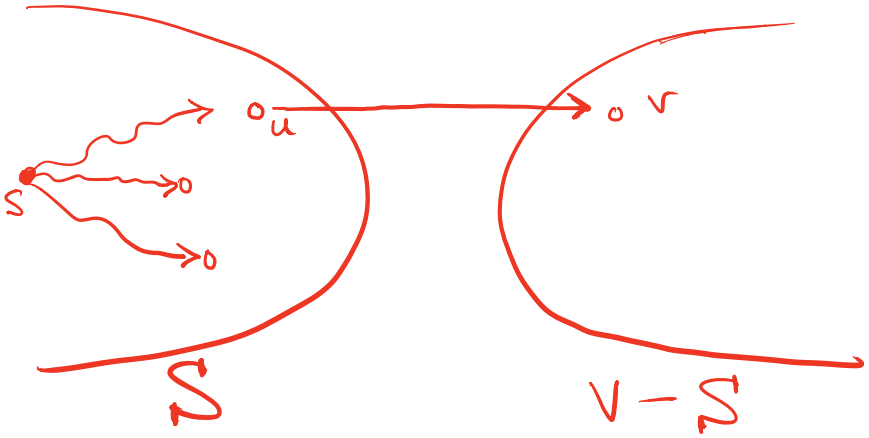
- Keep scanning this array until you find the origin vertex.

So we are filling in 2 arrays during the algorithm: the distance array and the previous array.

- The dist array tells you the distance to every node from s.
- The prev array tells you the node the preceeds node i in the shortest s-i path.

Sketch of Dijkstra's algorithm

The main idea is to maintain a set of nodes S that have been explored in the sense that we have already shortest path from the origin vertex s to every node in this set S.



For $u \in S$, the alg. assumes that we have correctly computed the shortest paths from the origin vertex to every node in S.

"explored" part of the graph

Initially, we can set

$S = \{s\}$ because we know

$dist[s] = 0$

$prev[s] = NIL$

$dist[v] = \infty$

↑ Since we don't know if v is reachable from s.

- At every time step / iteration of the algorithm, we will add a node to the set S from the set $V - S$ (every other node in the graph).
- In every iteration we will be identifying a $v \in V - S$ that can be joined via a cheapest path from the origin vertex. First, you travel within the explored part of the graph to some node u and then by a single edge you move from u to v.

$dist[v] = \infty$ if v has no incoming edge from S.

- At every time step, add one node $v \in V - S$ to S as follows:
  - For every node v in $V - S$, compute

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

Computing the min for every node v in $V - S$. There are multiple min computations.

In the next step, you select the node in $V - S$ that achieves the min of all the min computations

  - Select v s.t. $d(v) = \min_{x \in V - S} d(x)$
  - Set $S = S \cup \{v\}$
  - Set $dist[v] = d(v)$
  - Set $prev[v] = u$

The node responsible for shortest path. The node preceding v in the shortest s–v path

- This algorithm assumes that we have already computed shortest paths for every vertex in S from the origin vertex.

- We use the previous array ($prev[\cdot]$) in order to reconstruct the shortest paths in the end.

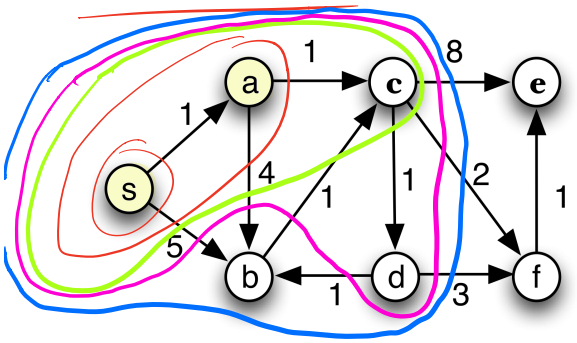
Clearly, we can only add those nodes to S from $V-S$ that are joined by an edge to a node in S (i.e., neighbors in $V-S$). Then we can simplify the algorithm and improve complexity by only considering the out neighbors of S at every iteration. That is,

$$N(S) = \text{out neighbors of } S$$
$$= \left\{ v \in V-S \text{ s.t. } \exists (u,v) \in E \text{ for some } u \in S \right\}$$

Then only need to compute $d(v)$ for the nodes in the set $N(S)$.

Example Below

EXAMPLE



$$N(S) = \{ v \in V - S \text{ s.t. } u \in S \text{ and } (u,v) \in E \}$$

Time step	S	$N(S)$	dist[v]	prev[v]
1	{s}	{a, b}	dist[s] = 0	prev[s] = NIL
2	{s, a}	{b, c}	dist[a] = 1	prev[a] = s
3	{s, a, c}	{b, d, e, f}	dist[c] = 2	prev[c] = a
4	{s, a, c, d}	{b, e, f}	dist[d] = 3	prev[d] = c
5	{s, a, c, d, b}	{e, f}	dist[b] = 4	prev[b] = d
6	{s, a, c, d, b, f}	{e}	dist[f] = 4	prev[f] = c
7	V	$\emptyset$	dist[e] = 5	prev[e] = f

Reconstructing shortest $s \rightsquigarrow b$ path:

$$s \leftarrow a \leftarrow c \leftarrow d \leftarrow b$$

$$b \leftarrow d \leftarrow c \leftarrow a \leftarrow s$$

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.

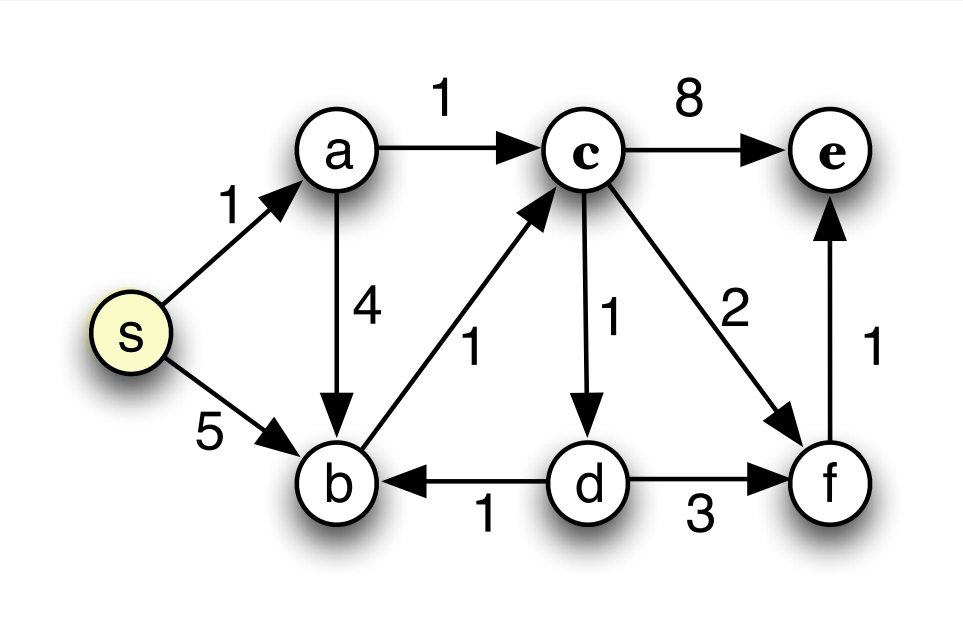
- Initially, $dist[s] = 0$, $S = \{s\}$.
- Each time, add to S the node $v \in V - S$ that
  1. has an edge from some node in S;
  2. minimizes the following quantity among all nodes $v \in V - S$

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

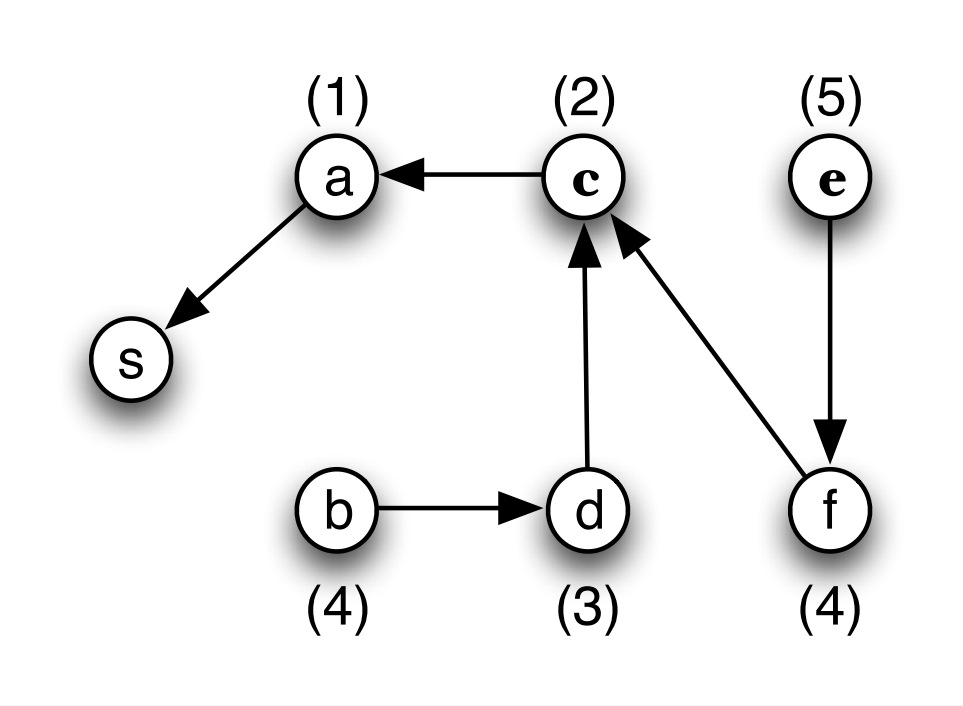
- Set $prev[v] = u$
- Set $dist[v] = d(v)$.

Algorithm assumes that any time during execution, the set S correctly contains all the nodes to which the shortest path from the origin has been determined. Therefore, this is what we need to prove to prove correctness.

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?*).

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.

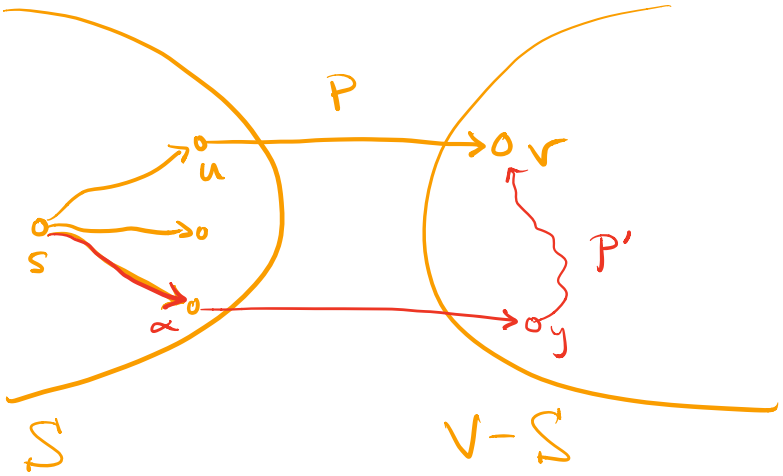
Pf: Induction on $|S|$.

Base: $|S| = 1$ $(S = \{s\})$

$dist[s] = 0$

Hyp: Assume Claim holds for $|S| = k \geq 1$

Step: Show Claim holds for $|S| = k+1$



$d(v) = \min_{x \in V-S} d(x)$

Let $P' \neq P$ be s.t. $w(P') < w(P)$.

$w(P'_{s \to y}) = dist[x] + w_{xy} \geq d(y) \geq d(v)$

$= w(P)$

$+ w(P'_{y \to v})$

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) → $O(n)$

$S = \{s\}$

while $S \neq V$ **do** → Enter while-loop n times (technically $n-1$ times)

n-times

$\sum_{x \in V} deg(x) = O(m)$

For every $x \in V - S$ with at least one edge from S compute

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

→ Fix x: $O(indeg(x))$ to compute whole thing.

→ fix x and fix u, compute inside in $O(1)$.

→ $O(1) \cdot deg(x)$

→ $\sum_{x \in V} O(1) \cdot deg(x) = O(m)$

$O(n)$

Select v such that $d(v) = \min_{x \in V - S} d(x)$

→ $O(1)$ for each computation and there are n of them. Total = $O(n)$.

$O(1)$ for each of them and there are n of them.

$S = S \cup \{v\}$

$dist[v] = d(v)$

$prev[v] = u$

} $O(1)$

end while

Initialize(G, s)

for $v \in V$ **do**

$dist[v] = \infty$

$prev[v] = NIL$

end for

$dist[s] = 0$

} $O(n)$

Initialize(G, s)
$S = \{s\}$ Enter while-loop n times

n times → **while** $S \neq V$ **do**

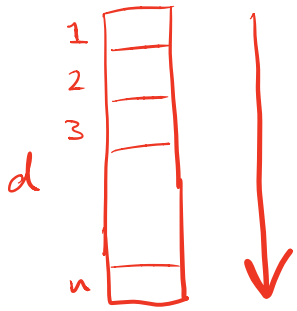
1. For every $x \in V - S$ with at least one edge from S compute
$$d(x) = \min_{u \in S, (u,x) \in E} \{dist[u] + w_{ux}\}$$
2. Select v such that $d(v) = \min_{x \in V - S} d(x)$
3. $S = S \cup \{v\}$
4. $dist[v] = d(v)$
5. $prev[v] = u$

end while

1. Fix v. Compute $d(v)$ in time $O(\deg(v))$

For all v, spend $\sum_{v \in V} O(\deg(v)) = O(m)$

2. $O(n)$



Total time:

$$n \cdot (O(m) + O(n))$$

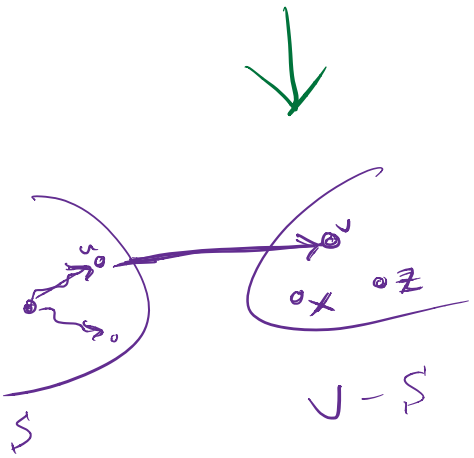
$$= O(mn)$$ // (Assuming here that $m = \Omega(n)$)

Total Time : $O(n \cdot m)$

(Because you enter the while-loop n times)

- Not a great running time
- Issue: computing $d(x)$ for every node in the graph, and then we are discarding them for the next iteration. Then we recompute them from scratch.

Improved implementation to follow (Below)



pick v such that dist[v] = dist[u] + w_{uv} + c_v

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 $(u, v) \in E$.

While-loop only loops over out neighbors of u instead of all nodes in G like in Dijkstra-v1.

```
Dijkstra-v2(G = (V, E, w), s ∈ V)
    Initialize(G, s)                          ← O(n)
    S = ∅
    while S ≠ V do                            ← Enter while-loop n-times.
O(n) →  Pick u so that dist[u] is minimum among all nodes in V − S
O(1) →  S = S ∪ {u}
O(1)·  for (u, v) ∈ E do
outdeg(u)      Update(u, v)                   → O(1) for fixed (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
```

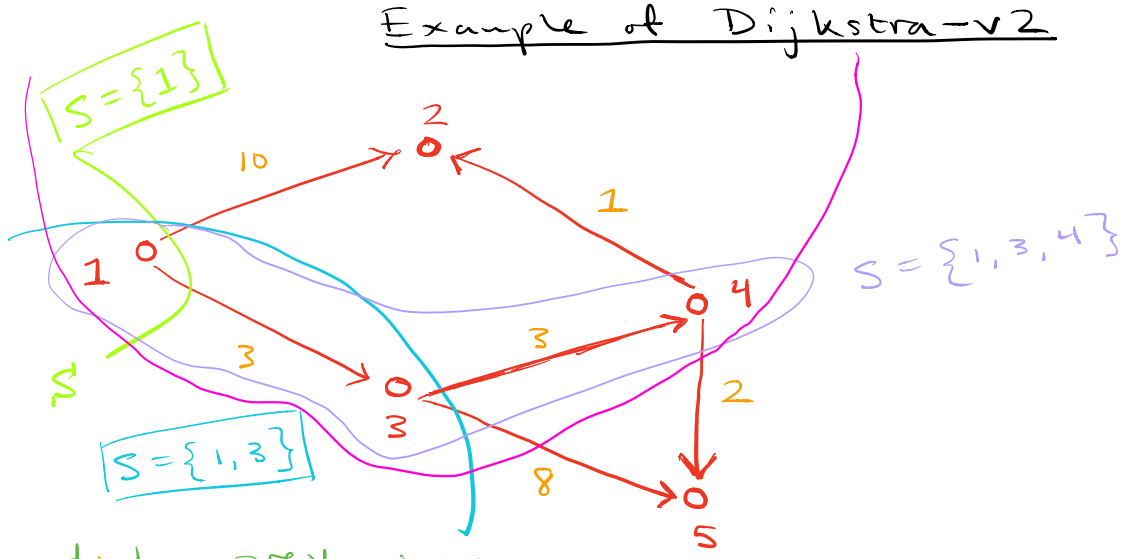
↳ $O(n)$ since dist array has n entries.

$$O(n^2) + \sum_{u \in V} O(1) \cdot \deg(u)$$

$$= O(n^2) + O(m) = O(n^2)$$

Total time: $O(n^2)$

Example of Dijkstra-v2



	dist	prev	in S
1	0	NIL	1
2	10	1	0
3	3	1	0
4	∞	NIL	0
5	∞	NIL	0

Scan array

	dist	prev	in S
1	0	NIL	1
2	10	1	0
3	3	1	1
4	6	3	0
5	11	3	0

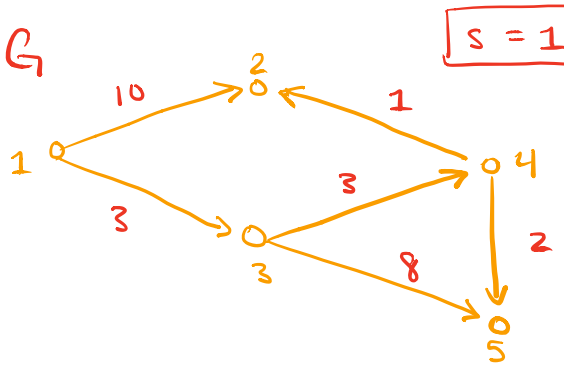
	dist	prev	in S
1	0	NIL	1
2	7	4	1
3	3	1	1
4	6	3	1
5	8	4	0

	dist	prev	in S
1	0	NIL	1
2	7	4	0
3	3	1	1
4	6	3	1
5	8	4	0

	dist	prev	in S
1	0	NIL	1
2	7	4	1
3	3	1	1
4	6	3	1
5	8	4	1

Updating dist[v] as v are added to S

G



s = 1

rep = 1

Scan dist to find node whose distance is minimum

	dist	prev	S
1	0	–	1
2	10	1	0
3	3	1	0
4	∞	–	0
5	∞	–	0

rep = 2

	dist	prev	S
1	0	–	1
2	10	1	0
3	3	1	1
4	6	3	0
5	11	3	0

rep = 3

	dist	prev	S
1	0	–	1
2	7	4	0
3	3	1	1
4	6	3	1
5	8	4	0

rep = 4

	dist	prev	S
1	0	–	1
2	7	4	1
3	3	1	1
4	6	3	1
5	8	4	0

rep = 5

	dist	prev	S
1	0	–	1
2	7	4	1
3	3	1	1
4	6	3	1
5	8	4	1

DONE

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

- `Initialize`(G, s) → $O(n)$
- $Q =$ `BuildQueue`$(\{V; dist\})$ → $O(n)$
- $S = \emptyset$
- **while** $Q \neq \emptyset$ **do**
  - $u =$ `ExtractMin`(Q) → $O(\log n)$ ⟹ Total : $O(n \log n)$
  - $S = S \cup \{u\}$
  - **for** $(u, v) \in E$ **do** → $O(\log n) \cdot \deg(u)$ ⟹ Total : $\sum_{u \in V} \deg(u) \cdot O(\log n) = O(m \log n)$
    - `Update`(u, v) → $O(\log n)$ for each fixed (u, v) since might require a Decrease-key(Q, x, k) operation to update distances.
  - **end for** ($O(\log n) \cdot \deg(u)$)
- **end while**

> - *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

Running time: $O(n \log n + m \log n) = O(m \log n)$ ← Assuming graph is sparse

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

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

- Total time for ExtractMin operations:

$$O(n \cdot \log n)$$

- Total time for Update(u,v) operations:
  - # Update(u,v): $\sum \deg(u) = m$
  - time per Update(u,v): $O(\log n)$

$\Rightarrow O(m \log n)$

Total Running Time: $O(n \log n + m \log n)$

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	$O(n\log n + m)$

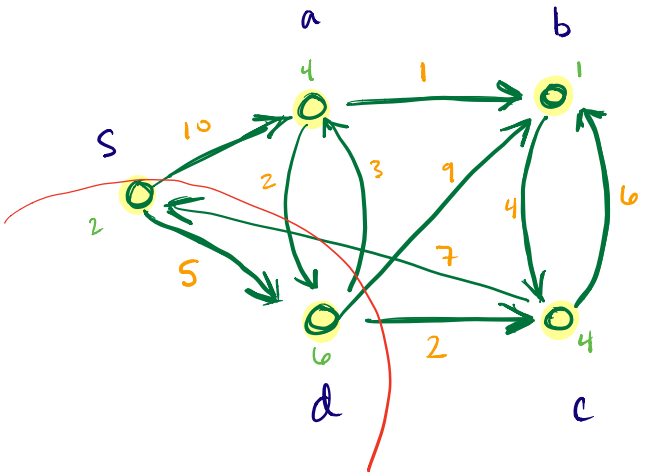
- Optimal choice is $d \approx m/n$ (the *average* degree of the graph)
- d-ary heap works well for both sparse and dense graphs
  - *If $m = n^{1+x}$, what is the running time of Dijsktra's algorithm using a d-ary heap?*

Sparse meaning that m is $O(n)$

↳ # edges close to minimal # of edges.

dense : # of edges is close to the maximal # of edges.

- For sparse graphs V3 of Dijkstra's is best
- For dense graphs V2 of Dijkstra's is good, but V3 is still better.



$13 + 7 = 20$

$13 + 10 = 23$

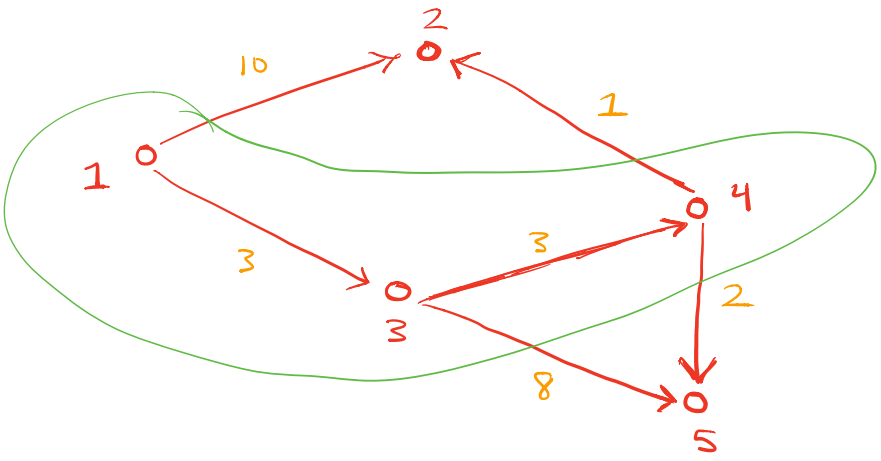
$13 + 2 + 4 = 19$

S	$dist[v]$	$prev[v]$	$N(S)$
$\{s\}$	2	NIL	$\{a, d\}$
$\{s, d\}$	$2 + 5 + 6 = 13$	$prev[d] = s$	$\{a, b, c\}$
$\{s, d, c\}$	$13 + 2 + 4 = 19$	$prev[c] = d$	

$$dist[d] = c_s + w_{sd} + c_d$$

$$= dist[s] + w_{sd} + c_d$$

$$dist[c] = dist[d] + w_{dc} + c_c$$



S	dist[v]	prev[v]	N(S)
{1}	dist[1] = 0	NIL	{2, 3}
{1, 3}	dist[3] = 3	prev[3] = 1	{2, 4, 5}
{1, 3, 4}	dist[4] = 6	prev[4] = 3	{2, 5}
{1, 3, 4, 2}	dist[2] = 7	prev[2] = 4	{5}
{1, 3, 4, 2, 5}	dist[5] = 8	prev[5] = 4	∅

dist =

0
7
3
6
8

prev =

NIL
4
1
3
4