Introduction to Algorithms

Meng-Tsung Tsai

12/03/2019

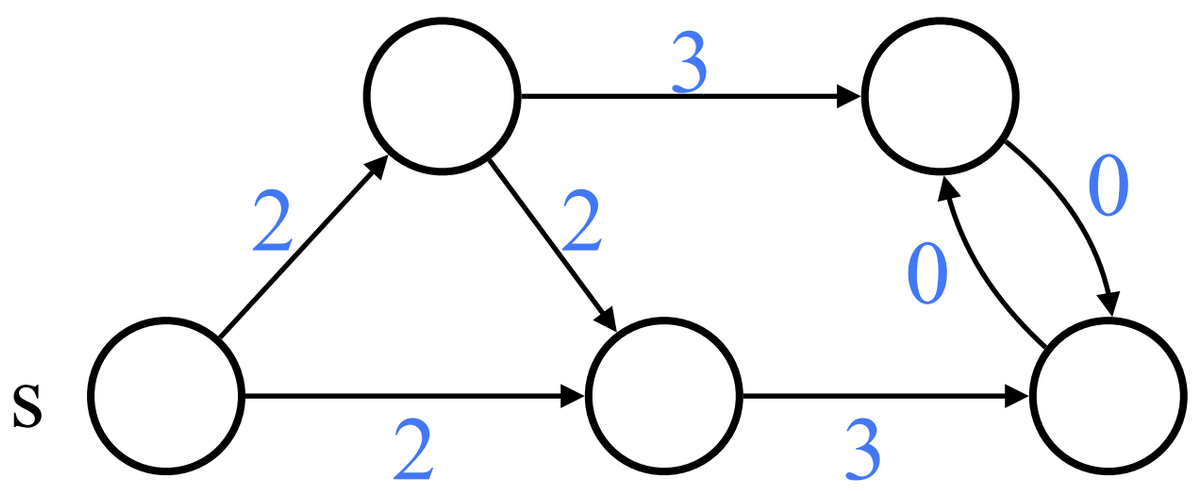
Dijkstra's algorithm

Problem

Input: a directed graph G, a weight function $\omega : E \to \mathbf{R}^+ \cup \{0\}$, and a (source) node s in G.

Output: for each node v in G, output the shortest (may be not simple) path P from s to v ($\omega(P) = \sum_{e \in P} \omega(e)$ is minimized).

Example.

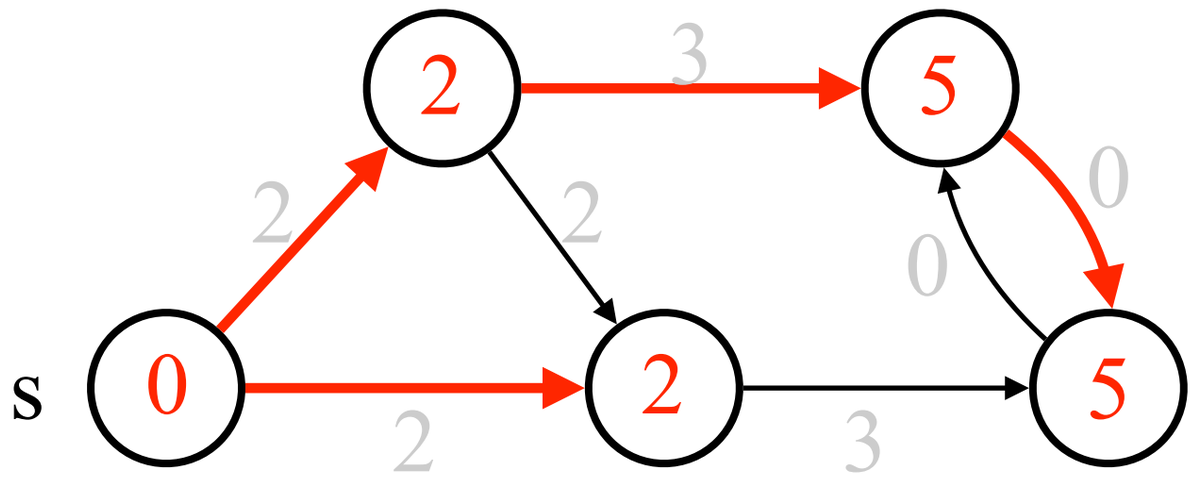


Problem

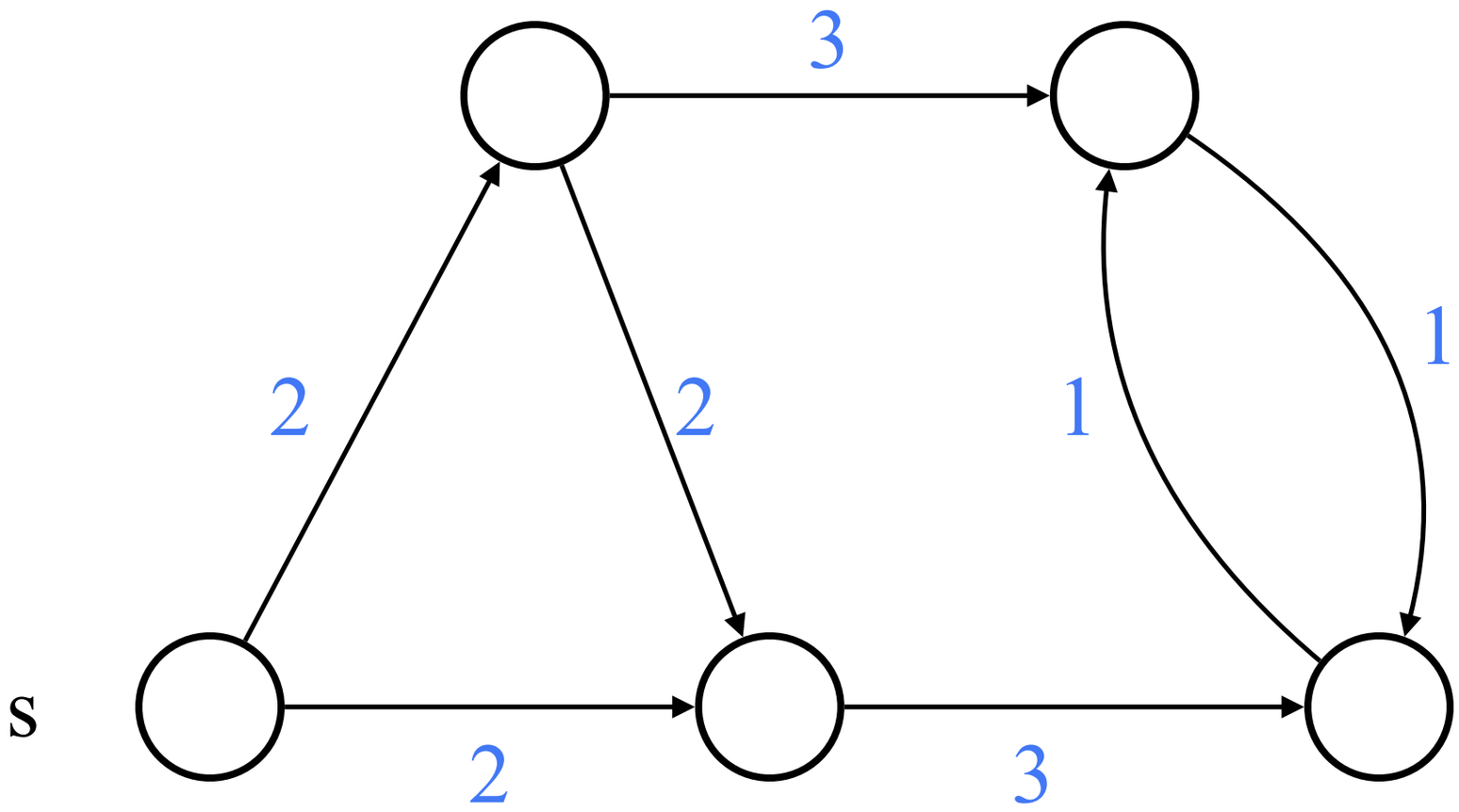
Input: a directed graph G, a weight function $\omega : E \to \mathbf{R}^+ \cup \{0\}$, and a (source) node s in G.

Output: for each node v in G, output the shortest (may be not simple) path P from s to v ($\omega(P) = \sum_{e \in P} \omega(e)$ is minimized).

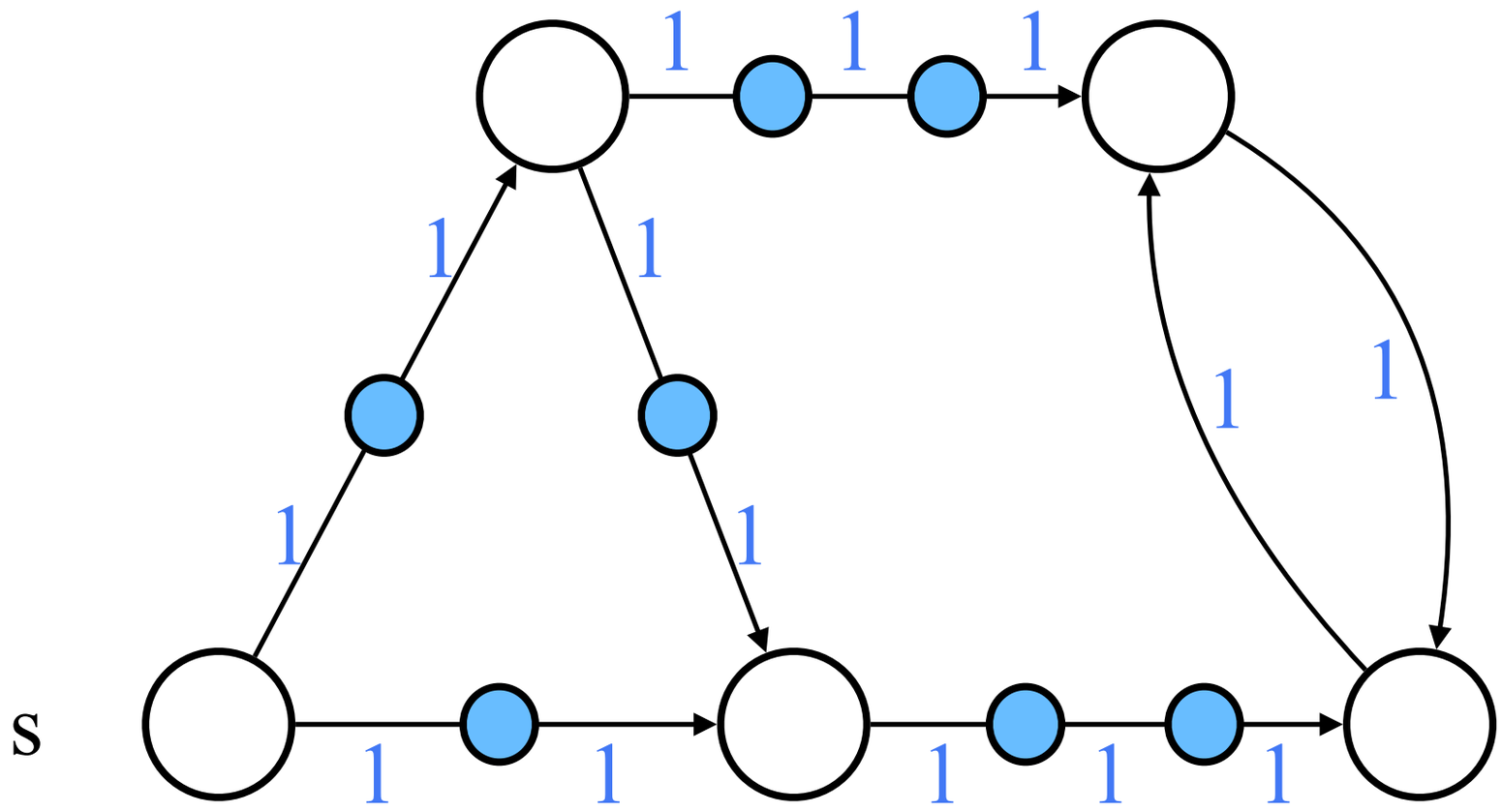
Example.



A simple case: if ω(e)'s are positive integers

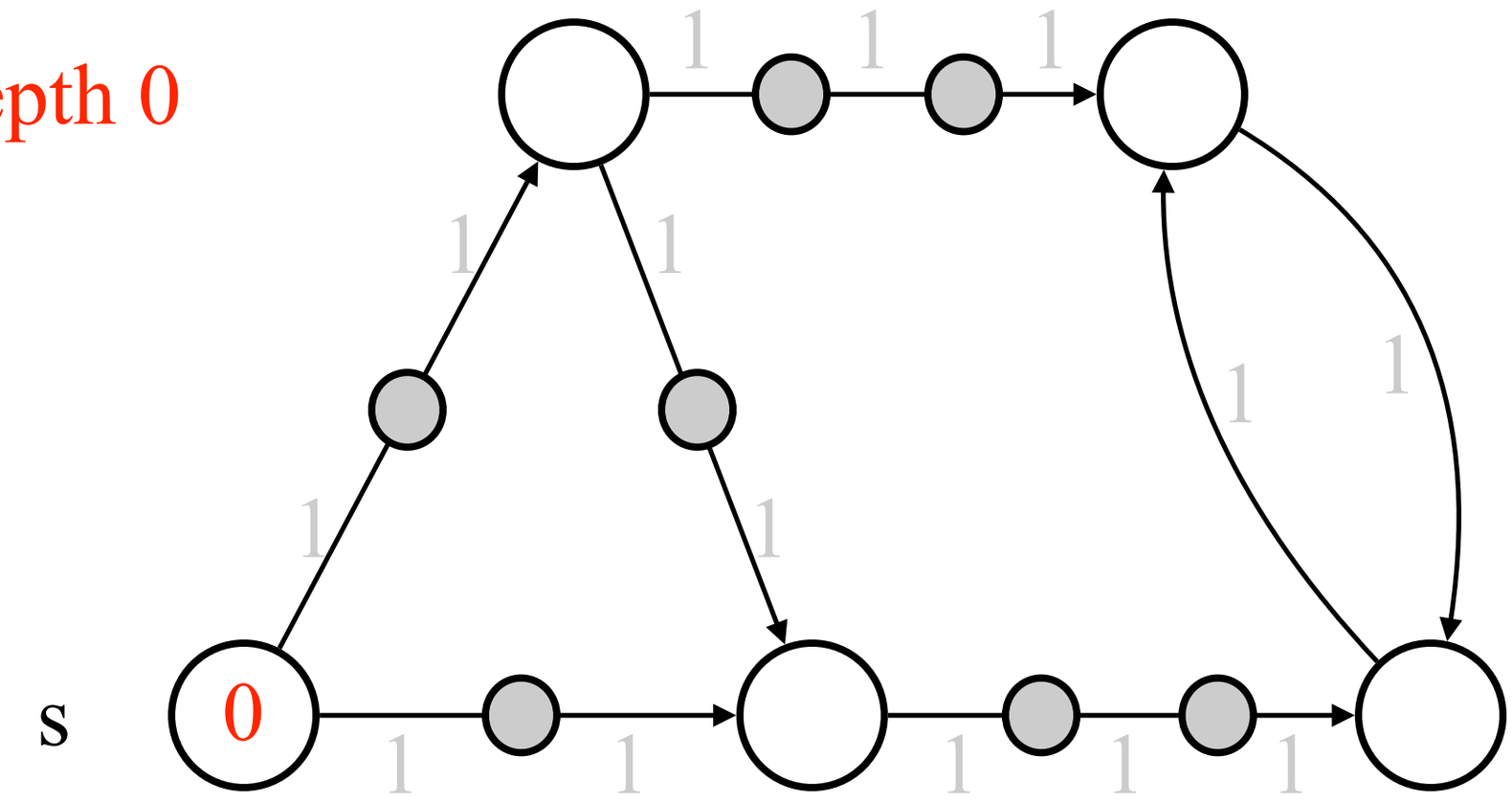


A simple case: if ω(e)'s are positive integers



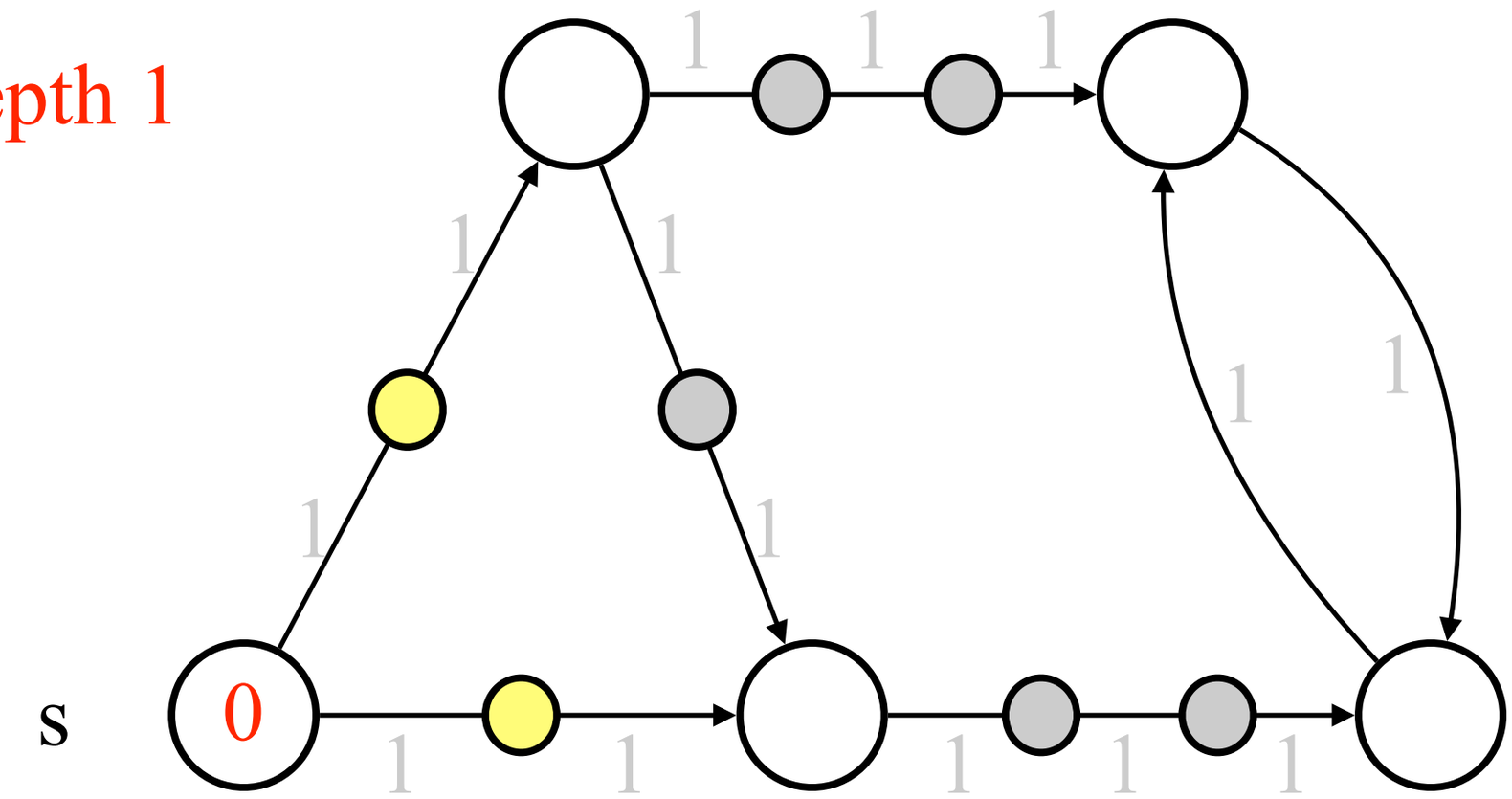
Because ω(e)'s are 1, running BFS(G, s) gives you the shortest path to every node in G.

A simple case: if ω(e)'s are positive integers



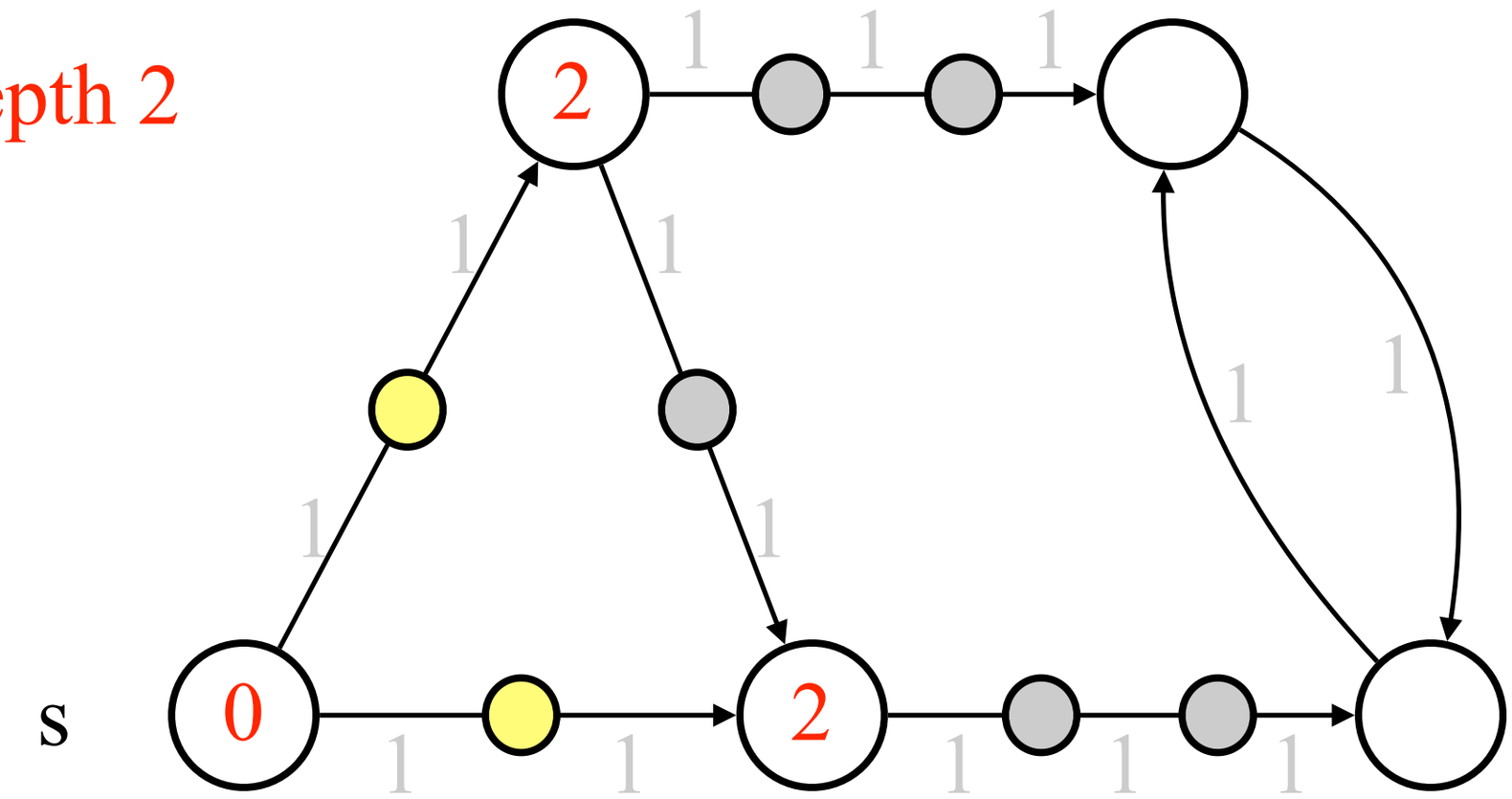
A simple case: if ω(e)'s are positive integers

depth 1



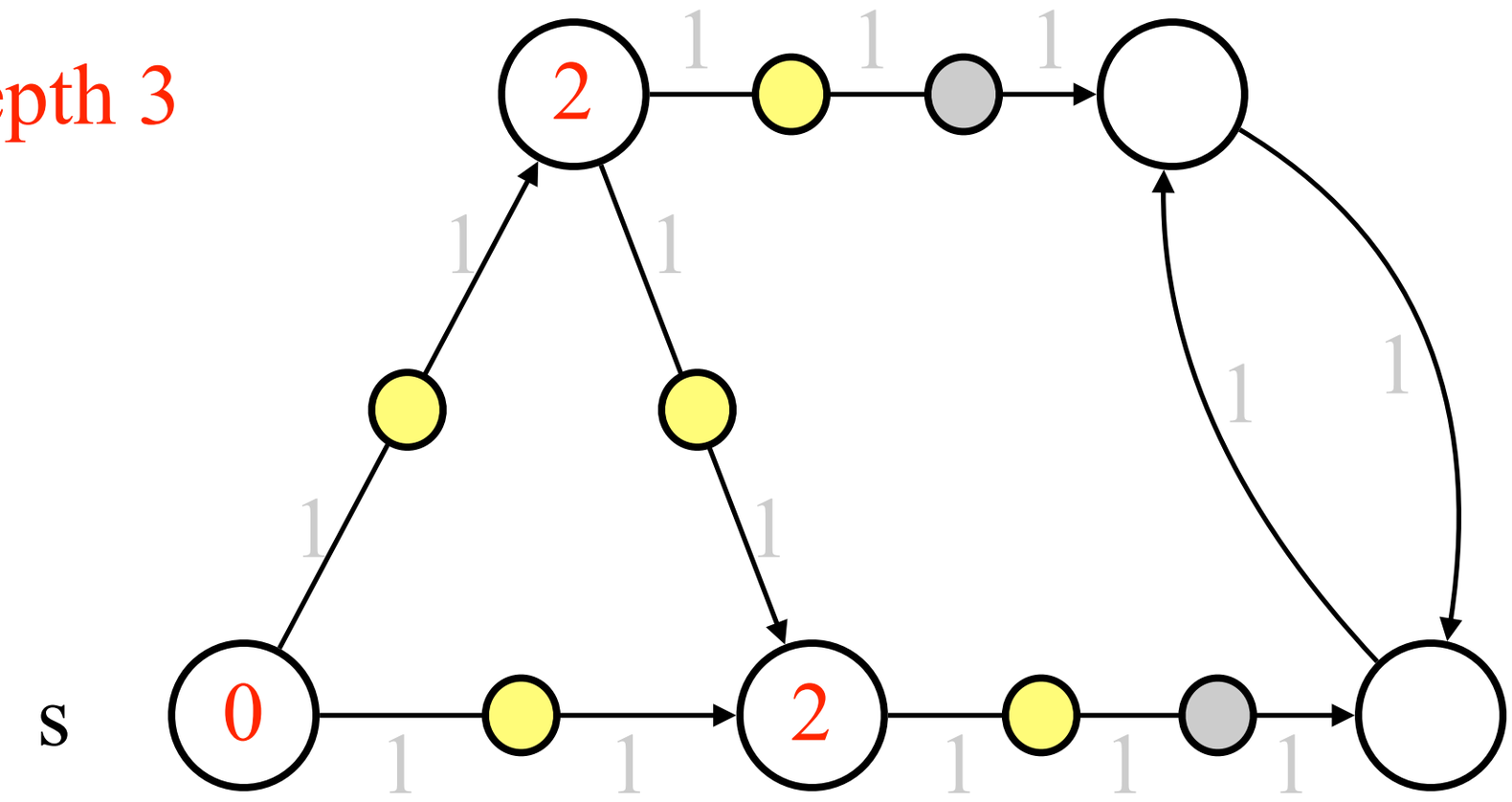
A simple case: if ω(e)'s are positive integers

depth 2



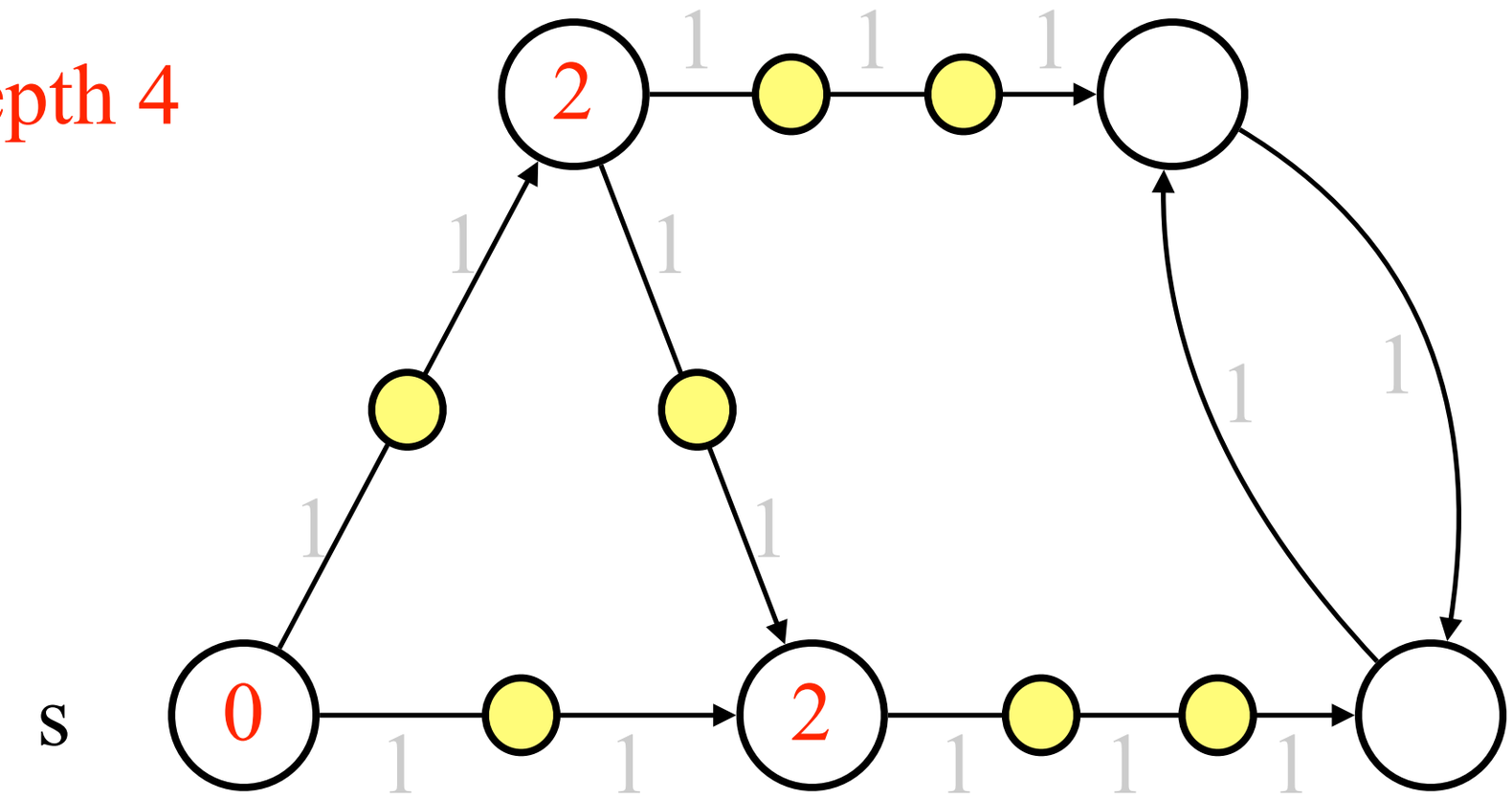
A simple case: if ω(e)'s are positive integers

depth 3

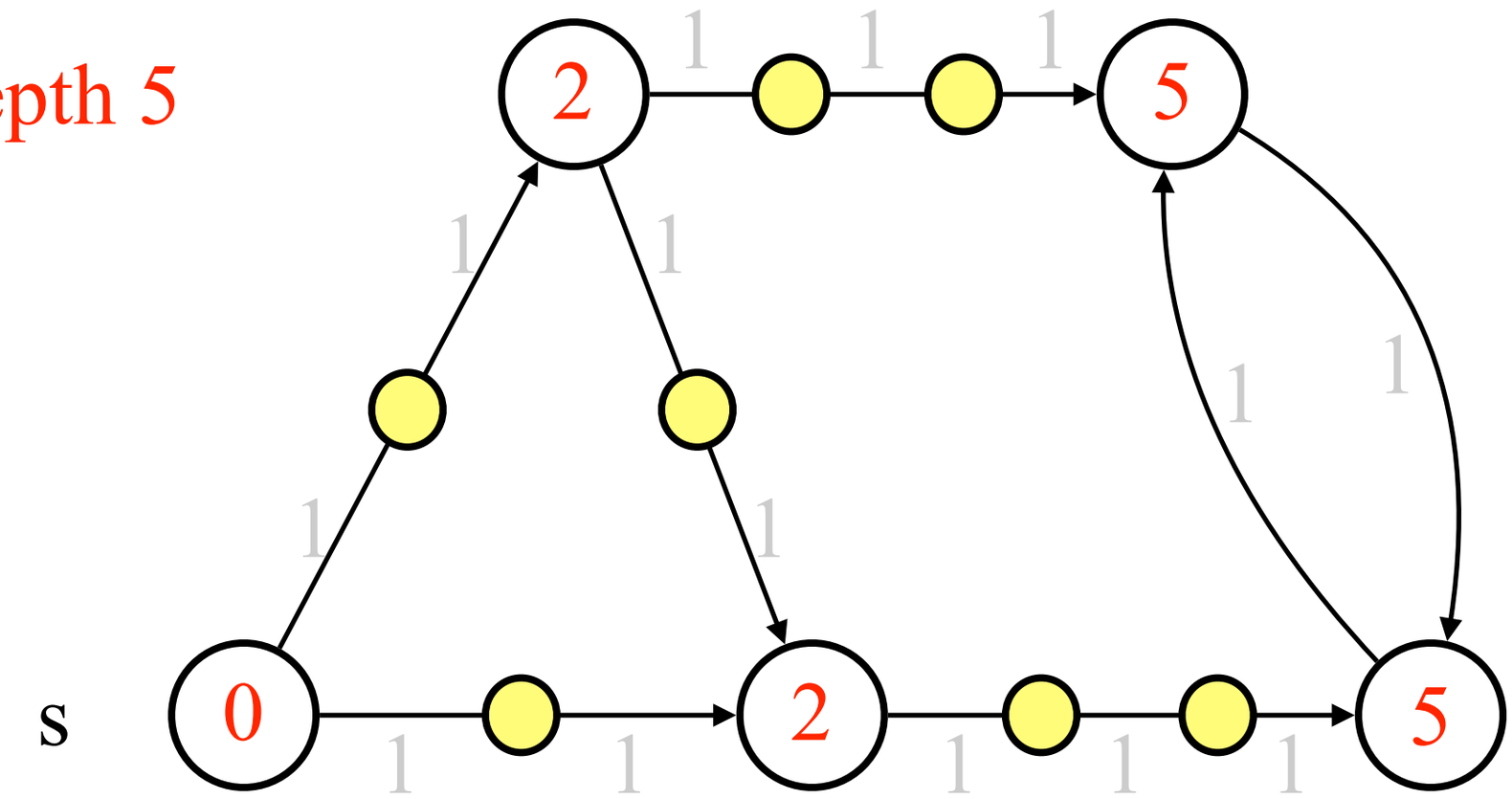


A simple case: if ω(e)'s are positive integers

depth 4



A simple case: if ω(e)'s are positive integers



If some ω(e) is large, then the corresponding edge will be subdivided into too many nodes, making this approach slow.

Dijkstra's algorithm is an implementation of the BFS approach without subdividing the edges

```
Dijkstra(G, ω, s){
    Initialization(G, s);
    S ← ∅;
    Q ← {v : v is a node in G}; // Q is a min-heap, with the
    est(v) attributes as keys
    while(Q ≠ ∅){
        u ← Extract-Min(Q); // find v whose est(v) is smallest in Q
        S ← S ∪ {u};
        foreach node v in Adj[u]{
            Relax(u, v, ω); // an implicit Decrease-Key(v)
        }
    }
}
```

Dijkstra's algorithm is an implementation of the BFS approach without subdividing the edges

```
Dijkstra(G, ω, s){
    Initialization(G, s);
    S ← ∅;
    Q ← {v : v is a node in G}; // Q is a min-heap, with the
    est(v) attributes as keys
    while(Q ≠ ∅){
        u ← Extract-Min(Q); // find v whose est(v) is smallest in Q
        S ← S ∪ {u};
        foreach node v in Adj[u]{
            Relax(u, v, ω); // an implicit Decrease-Key(v)
        }
```

Like Prim's algorithm, we cannot find v in a heap efficiently.
We need to use a direct-address table to find v in O(1) time.

Correctness

```
while(Q ≠ ∅){
    u ← Extract-Min(Q); // find v whose est(v) is smallest in Q
    S ← S ∪ {u};
    foreach node v in Adj[u]{
        Relax(u, v, ω); // an implicit Decrease-Key(v)
    }
```

Claim. Let est(v) be the est(v) at the moment that v is extracted from Q. Let $v_1, v_2, ..., v_n$ be the order that nodes are extracted from Q. We have $\text{est}(v_1) \leq \text{est}(v_2) \leq ... \leq \text{est}(v_n)$.

v_1 is s, and est(s) is 0.
v_2 can be relaxed only by $v_1 \Rightarrow$
[1] $\text{est}(v_2) = \text{est}(v_1)+\omega(v_1, v_2) \geq \text{est}(v_1)$, or
[2] $\text{est}(v_2) = \infty$. // not relaxed, then $\text{est}(v_2) = ... = \text{est}(v_n) = \infty$.

Correctness

```
while(Q ≠ ∅){
    u ← Extract-Min(Q); // find v whose est(v) is closest to s
    S ← S ∪ {u};
    foreach node v in Adj[u]{
        Relax(u, v, ω); // an implicit Decrease-Key(v)
    }
```

Claim. Let est(v) be the est(v) at the moment that v is extracted from Q. Let $v_1, v_2, ..., v_n$ be the order that nodes are extracted from Q. We have $\text{est}(v_1) \le \text{est}(v_2) \le ... \le \text{est}(v_n)$.

v_3 can be relaxed by either v_1 or $v_2 \Rightarrow$
[1] $\text{est}(v_3) = \text{est}(v_1)+\omega(v_1, v_3) \ge \text{est}(v_1)$, or
[2] $\text{est}(v_3) = \text{est}(v_2)+\omega(v_2, v_3) \ge \text{est}(v_2) \ge \text{est}(v_1)$, or [3] $\text{est}(v_3) = \infty$.
If [3] or $\text{est}(v_3) \ge \text{est}(v_2)$ holds, then we are done. Otherwise, [1] holds $\Rightarrow$ $\text{est}(v_3) = \text{est}(v_1)+\omega(v_1, v_3) \ge \text{est}(v_2)$ because $\text{est}(v_2)$ is the smallest value among est(v)'s relaxed by v_j for some $j < 2$.

Correctness

```
while(Q ≠ ∅){
    u ← Extract-Min(Q); // find v whose est(v) is closest to s
    S ← S ∪ {u};
    foreach node v in Adj[u]{
        Relax(u, v, ω); // an implicit Decrease-Key(v)
    }
```

Claim. Let est(v) be the est(v) at the moment that v is extracted from Q. Let $v_1, v_2, ..., v_n$ be the order that nodes are extracted from Q. We have $\text{est}(v_1) \leq \text{est}(v_2) \leq ... \leq \text{est}(v_n)$.

By repeating this argument over all v_i's (formally by induction), we complete the proof.

Correctness

```
while(Q ≠ ∅){
    u ← Extract-Min(Q); // find v whose est(v) is closest to s
    S ← S ∪ {u};
    foreach node v in Adj[u]{
        Relax(u, v, ω); // an implicit Decrease-Key(v)
    }
```

Claim. Let est(v) be the est(v) at the moment that v is extracted from Q. Let $v_1, v_2, ..., v_n$ be the order that nodes are extracted from Q. We have $\underline{est}(v_1) \leq \underline{est}(v_2) \leq ... \leq \underline{est}(v_n)$.

Observe that only $\underline{est}(v_i)$ can be used to relax $est(v_j)$. If $j < i$, then $\underline{est}(v_j) \leq \underline{est}(v_i)$, thus $est(v_j) = \underline{est}(v_j)$ after v_j is extracted from Q.

Correctness

```
while(Q ≠ ∅){
    u ← Extract-Min(Q); // find v whose est(v) is closest to s
    S ← S ∪ {u};
    foreach node v in Adj[u]{
        Relax(u, v, ω); // an implicit Decrease-Key(v)
    }
```

Claim. $\underline{est}(v_i) = dis(v_i)$. Let $P = (s, x_1), (x_1, x_2), ..., (x_k, v_i)$ be a shortest path $s \rightsquigarrow v_i$ (thus $\omega(P) = dis(v_i)$).

Clearly, this holds for $v_1 = s$. We assume that this holds for v_2, ..., v_{i-1}. If $\omega(x_k, v_i) > 0$, then $dis(x_k) < dis(v_i)$. By the previous claim and the above assumption, $x_k = v_j$ for some $j < i$. Hence, $\underline{est}(v_i) \leq \underline{est}(x_k) + \omega(x_k, v_i) = dis(v_i)$.

Running time

```
while(Q ≠ ∅){
    u ← Extract-Min(Q); // find v whose est(v) is closest to s
    S ← S ∪ {u};
    foreach node v in Adj[u]{
        Relax(u, v, ω); // an implicit Decrease-Key(v)
    }
```

[1] If we linear scan Q to find the minimum, then it takes O(n) time for each node v. In total, it runs in $O(n^2)$ time. The number of Relax operations is at most $O(m) = O(n^2)$.

[2] If we use binary heaps to find the minimum, then it takes O(log n) time for each node v. In total, it runs in O(n log n). Relax() now requires perform Decrease-Key(). Thus, the total running time is O((n+m) log n).

[3] By Fibonacci-heaps, the total running time is O(n log n+m).

Exercise

If $\omega(x_k, v_i) = 0$, then x_k might be extracted from Q after v_i. Please complete this proof.

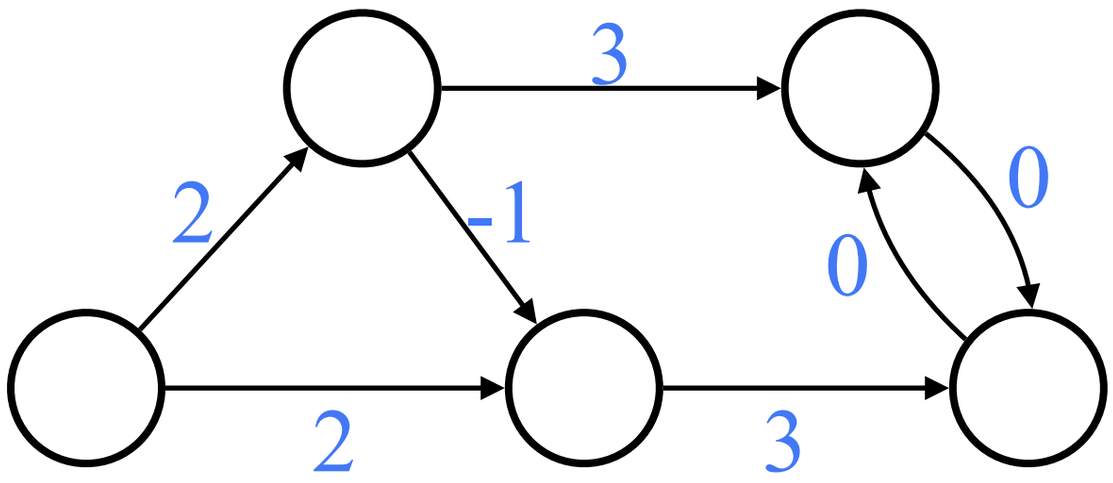
All-Pairs Shortest Paths

Problem

Input: a directed graph G, a weight function $\omega : E \to \mathbf{R}$.

Output: for every pair of nodes u, v in G, output the shortest (may be not simple) path P from u to v ($\omega(P) = \sum_{e \in P} \omega(e)$ is minimized).

Example.

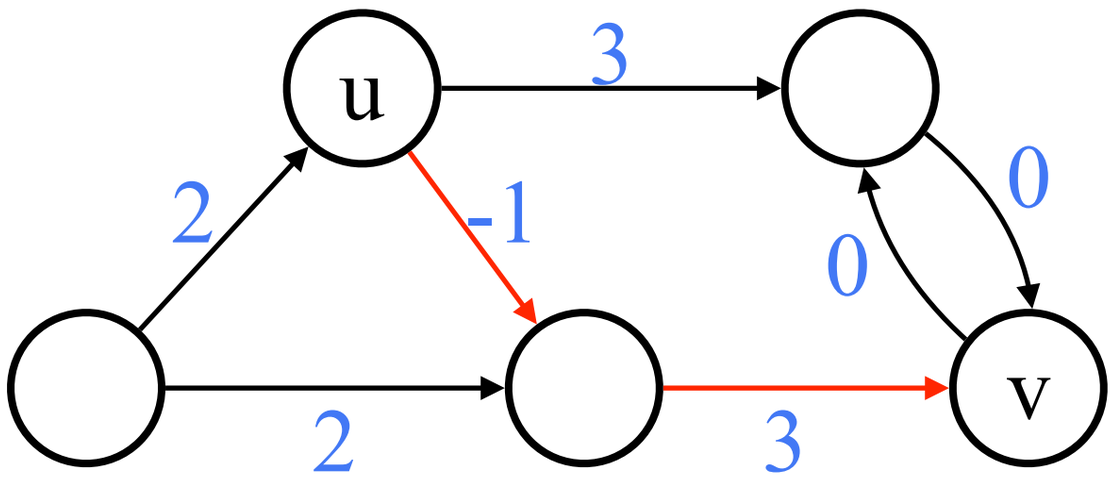


Problem

Input: a directed graph G, a weight function $\omega : E \to \mathbf{R}$.

Output: for every pair of nodes u, v in G, output the shortest (may be not simple) path P from u to v ($\omega(P) = \sum_{e \in P} \omega(e)$ is minimized).

Example.

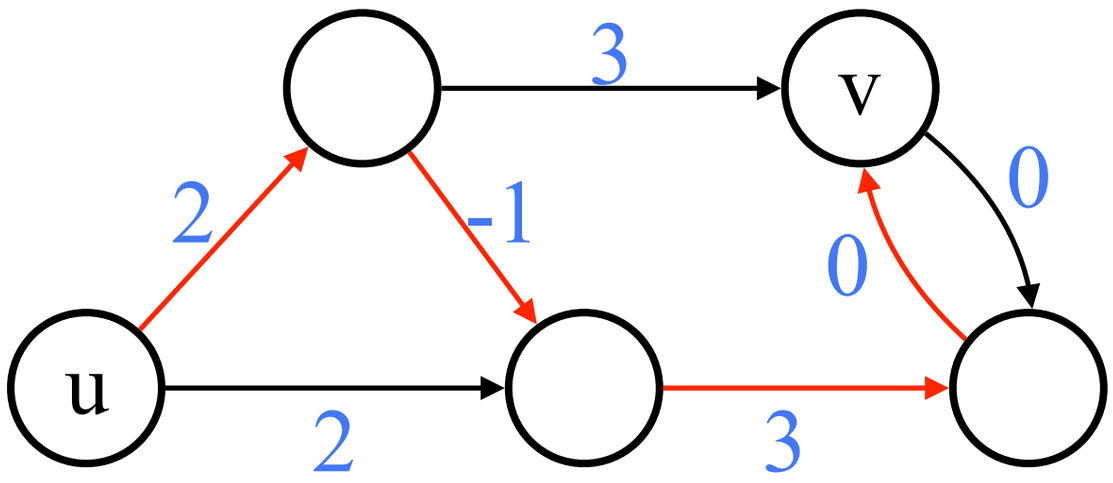


Problem

Input: a directed graph G, a weight function $\omega : E \to \mathbf{R}$.

Output: for every pair of nodes u, v in G, output the shortest (may be not simple) path P from u to v ($\omega(P) = \sum_{e \in P} \omega(e)$ is minimized).

Example.

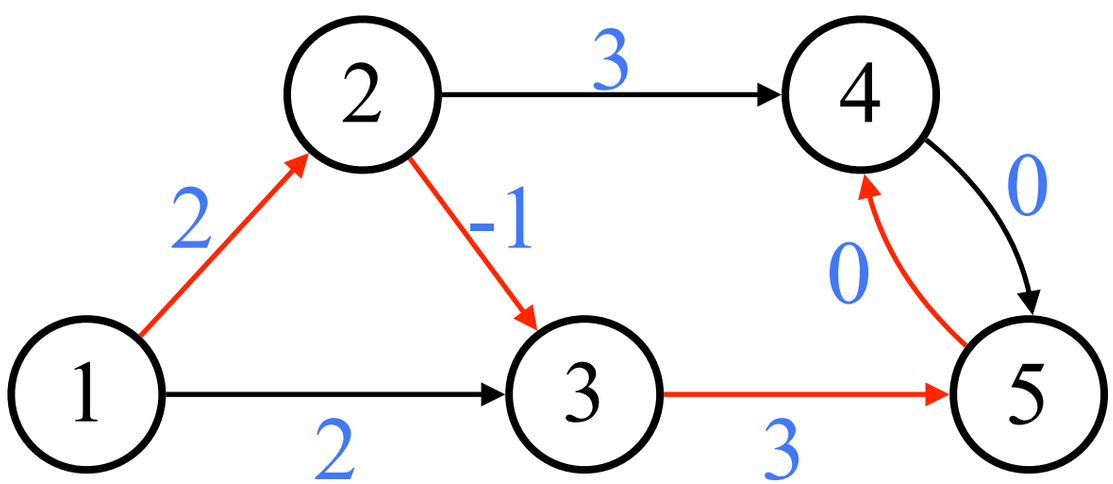


Adjacency matrix representation

Input: a matrix W in which (1) $\omega_{ij} = 0$ if i = j (2) $\omega_{ij} = \infty$ if (i, j) is not an edge of the input graph (3) ω_{ij} = the weight of the directed edge (i, j)

Output: a matrix D in which d_{ij} denotes the weight of a shortest path $i \rightsquigarrow j$, and a matrix P in which p_{ij} denotes the predecessor of j on the shortest path $i \rightsquigarrow j$.

Example.



The nodes are numbered from 1 to n.

$\omega_{12} = 2$, $\omega_{33} = 0$, $\omega_{53} = \infty$.

$d_{12} = 2$, $d_{25} = 2$, $d_{14} = 4$

$p_{14} = 5$, $p_{25} = 3$

The choices of graph representation

	single-source shortest paths	all-pairs shortest paths
graph representation	adjacency list	adjacency matrix

The choices of graph representation

	single-source shortest paths	all-pairs shortest paths
graph representation	adjacency list	adjacency matrix

The adjacency list representation use $O(m)$ space, which may be $o(n^2)$. To store the computed results for all of n^2 pairs, it already needs $O(n^2)$ space. Hence, saving space by using adjacency list is pointless for all-pairs shortest paths.

Restriction

There exists no negative cycle in the input graph.

Given this restriction, we have:

[1] for every pair of nodes u, v, there exists a simple shortest path u ⇝ v. // Otherwise, iteratively cut a cycle on the path until it has no cycle.

[1] ⇒ [2] for every pair of nodes u, v, there exists a simple shortest path of length at most n-1. // We define the length of a path to be the number of edges on the path, and define the weight of a path to be the sum of edge weights on the path.

Solving APSP by SSSP

By SSSP

For each node v, compute the SSSP with the source v. By repeating this n rounds, we obtain APSP.

	running time	restriction
Dijkstra's (array)	O(n (n^2))	no negative edge
Dijkstra's (binary heaps)	O(n (n+m) log n)	no negative edge
Dijkstra's (Fibonacci heaps)	O(n (n log n + m))	no negative edge
Bellman-Ford	O(n (nm))	no negative cycle (from some s?)

Dynamic Programming

Observation

Because the input graph has no negative cycle, it suffices to find a shortest path u ⇝ v of length at most n-1 for every u, v.

Let the shortest path u ⇝ v be a shortest path u ⇝ k plus the edge (k, v) for some unknown k. // Why? The subpath (u ⇝ k) of a shortest path (u ⇝ v) is also a shortest path.

We can find the unknown k by computing

$$d_{uv} = \ell_{uv}^{(n-1)} = \min_k \ell_{uk}^{(n-2)} + \omega_{kv}$$

where $\ell_{uv}^{(t)}$ denotes the minimum weight of any path from u to v that have length at most t.

$$\ell_{uv}^{(1)} = \omega_{uv}$$

By definition, $\ell_{uv}^{(1)}$ is the minimum weight of any path from u to v that have length at most 1. Thus, the path is either an edge (u, v) or a null path (u, u).

Given $\ell_{uv}^{(t-1)}$ and ω_{uv}, compute $\ell_{uv}^{(t)}$

Extend-Shortest-Paths($L^{(t-1)}$, W){
Let $L^{(t)} = \{\infty\}$;
for(u = 1 to n){
for(v = 1 to n){
for(k = 1 to n){ // can we exclude both k = u and k=v?
if($\ell_{uk}^{(t-1)} + \omega_{kv} < \ell_{uv}^{(t)}$){

$\ell_{uv}^{(t)} \leftarrow \ell_{uk}^{(t-1)} + \omega_{kv}$;

}
}
}
}
}

Running time and space

Running time:

We need to compute $D = L^{(n-1)}$ from $L^{(1)}$. Since it takes $O(n^3)$ time to obtain $L^{(t)}$ from $L^{(t-1)}$, the total running time is $O(n^4)$.

Space:

While computing $L^{(t)}$ from $L^{(t-1)}$, we no longer need $L^{(t-2)}$. Thus, we need only two L matrices at a time, which uses $O(n^2)$ space.

Matrix P:

The computation of predecessor matrix P is omitted for this algorithm.

Faster Dynamic Programming

Observation

Because the input graph has no negative cycle, it suffices to find a shortest path u ⇝ v of length at most n-1 for every u, v.

Let the shortest path u ⇝ v be a shortest path u ⇝ k plus another shortest path (k, v) for some unknown k. // Why?
The subpaths of a shortest path (u ⇝ v) are shortest paths.

We can find the unknown k by computing

$$d_{uv} = \ell_{uv}^{(2^r)} = \min_k \ell_{uk}^{(2^{r-1})} + \ell_{kv}^{(2^{r-1})},$$

where $2^r \geq n-1$.

Note that $L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots = L^{(\infty)}$.

$$\ell_{uv}^{(1)} = \omega_{uv}$$

By definition, $\ell_{uv}^{(1)}$ is the minimum weight of any path from u to v that have length at most 1. Thus, the path is either an edge (u, v) or a null path (u, u).

Given $\ell_{uv}^{(2^{r-1})}$, compute $\ell_{uv}^{(2^r)}$

Extend-Shortest-Paths($L^{(2^{r-1})}$, W){
  Let $L^{(2^r)}$= {∞};
  for(u = 1 to n){
    for(v = 1 to n){
      for(k = 1 to n){
        if($\ell_{uk}^{(2^{r-1})} + \ell_{kv}^{(2^{r-1})} < \ell_{uv}^{(2^r)}$){
          $\ell_{uv}^{(2^r)} \leftarrow \ell_{uk}^{(2^{r-1})} + \ell_{kv}^{(2^{r-1})}$;
        }
      }
    }
  }
}

Running time and space

Running time:

We need to compute D = $L^{(2^r)}$ from $L^{(1)}$, where $2^r \geq n-1$. Since it takes $O(n^3)$ time to obtain $L^{(2^t)}$ from $L^{(2^{(t-1)})}$, the total running time is $O(n^3 \log n)$.

Space:

By the same argument, the space usage is $O(n^2)$.

Matrix P:

The computation of predecessor matrix P is omitted for this algorithm.

Floyd-Warshall Algorithm

Observation

Let path $u \rightsquigarrow^{k} v$ be the shortest path from u to v and in which each intermediate node, excluding u and v, has an identifier (id) $\leq k$.

By definition, we have $\omega(u \rightsquigarrow v) = \omega(u \rightsquigarrow^{n} v)$, and

$$\omega(u \rightsquigarrow^{k} v) = \min(\underline{\omega(u \rightsquigarrow^{k-1} v)}, \underline{\omega(u \rightsquigarrow^{k-1} k)+\omega(k \rightsquigarrow^{k-1} v)}).$$

doesn't pass through node k　　pass through node k

$$\omega(u \rightarrow^0 v) = \omega(u, v)$$

By definition, $u \rightarrow^0 v$ is the shortest path from u to v and in which each intermediate node, excluding u and v, has an identifier (id) ≤ 0. Since every node has an identifier in {1, 2, ..., n}, $u \rightarrow^0 v$ has no intermediate node.

Given $\omega(u \rightsquigarrow^{k-1} v)$, compute $\omega(u \rightsquigarrow^{k} v)$

Floyd-Warshall(W){ // let $F^{(k)}$ be the matrix containing the weight $\omega(u \rightsquigarrow^{k} v)$ for every u, v

$F^{(0)} = W$;

for(k = 1 to n){

$F^{(k)} = F^{(k-1)}$; // doesn't pass through node k

for(u = 1 to n){

for(v = 1 to n){

if($f_{uk}^{(k-1)} + f_{kv}^{(k-1)} < f_{uv}^{(k)}$){

$f_{uv}^{(k)} \leftarrow f_{uk}^{(k-1)} + f_{kv}^{(k-1)}$;

}

}}}}

Running time and space

Running time:

There are 3 nested loops and each loop has n interations, the total running time is thus $O(n^3)$.

Space:

By the same argument, the space usage is $O(n^2)$.

Matrix P:

The computation of predecessor matrix P is shown in the following slides.

Predecessor matrix

We have $P_{uv}^{(0)}$ = u if there exists edge (u, v) and = NIL otherwise.

If $\omega(u \rightsquigarrow^{k} v) = \omega(u \rightsquigarrow^{k-1} v)$, then it means the shortest path from u to v doesn't pass through node k. Hence, $P_{uv}^{(k)} = P_{uv}^{(k-1)}$.

If $\omega(u \rightsquigarrow^{k} v) = \omega(u \rightsquigarrow^{k-1} k) + \omega(k \rightsquigarrow^{k-1} v)$, then the predecessor of v can be found in the latter subpath. That is, $P_{uv}^{(k)} = P_{kv}^{(k-1)}$.

Pseudocode

Floyd-Warshall(W){ // let $F^{(k)}$ be the matrix containing the weight $\omega(u \rightsquigarrow^{k} v)$ for every u, v

$F^{(0)} = W$; $P_{uv}^{(0)} = (\omega_{uv} < \infty)?u : \text{NIL}$;

for(k = 1 to n){

$F^{(k)} = F^{(k-1)}$; $P^{(k)} = P^{(k-1)}$;

for(u = 1 to n){

for(v = 1 to n){

if($f_{uk}^{(k-1)} + f_{kv}^{(k-1)} < f_{uv}^{(k)}$){

$f_{uv}^{(k)} \leftarrow f_{uk}^{(k-1)} + f_{kv}^{(k-1)}$;

$P_{uv}^{(k)} \leftarrow P_{kv}^{(k-1)}$;

}

}}}}

Sparse Graphs

By SSSP

Dijkstra's algorithm needs $O(n^2 \log n + nm)$ time, which is very fast if m is small. However, this approach works only for graphs that has no negative edge.

	running time	restriction
Dijkstra's (array)	$O(n\ (n^2))$	no negative edge
Dijkstra's (binary heaps)	$O(n\ (n+m) \log n)$	no negative edge
Dijkstra's (Fibonacci heaps)	$O(n\ (n \log n + m))$	no negative edge
Bellman-Ford	$O(n\ (nm))$	no negative cycle (from some s?)

For the graphs with negative edges

Idea:

Assign a new weight to each edge so that

[1] for each pair of nodes u, v, every shortest path u ⇝ v before the reweighting remains a shortest path u ⇝ v after the reweighting,

[2] all edges have non-negative weight. (thus, we can use the approach of running Dijkstra's algorithm n times)

Johnson's algorithm - the 1st condition

Given any function h: V $\rightarrow$ **R**, if we let the new weight function ω'_{uv} be $h(u)-h(v)+\omega_{uv}$, then every path P from u to v has weight

$$\sum_{e\in P} \omega'(e) = h(u)-h(v)+\sum_{e\in P} \omega(e).$$

Since h(u)-h(v) is a constant when we fix u and v, a path is a min ω'-weighted shortest path iff it is a min ω-weighted shortest path.

Johnson's algorithm - the 2nd condition

Create a super source node s, and add edge (s, v) for each node v in the input graph with $\omega_{sv} = 0$.

Running the Bellman-Ford algorithm, we have d_{sv} for each v.

Let $h(u) = d_{su}$ and $h(v) = d_{sv}$. Clearly, $d_{sv} \leq d_{su} + \omega_{uv}$.

shortest path — a path

Hence, $\omega'_{uv} = h(u) + \omega_{uv} - h(v) \geq 0$. We are done.

Exercise

Prim's algorithm and Dijkstra's algorithm look similar. What are the differences between these two algorithms?

Exercise

In practice, Floyd-Warshall algorithm is usually implemented as follows:

```
dis[,] = {∞};
foreach(edge u, v) dis[u, v] = ω[u, v];
foreach(nod v) dis[v, v] = 0;
for(k = 1 to n){
    for(u = 1 to n){
        for(v = 1 to n){
            if(dis[u, k] + dis[k, v] < dis[u, v]){
                dis[u, v] = dis[u, k] + dis[k, v];
            }
        }
    }
}
```

How to update the predecessor matrix accordingly?