Single-Source Shortest Paths

Chapter 24 of Cormen's book

Giulia Bernardini
giulia.bernardini@units.it

Algorithmic Design
a.y. 2022/2023

Does BFS work for weighted graphs too?

BFS(*G*,*s*) - *G* is represented by the adjacency lists `Adj[·]` of its vertices

```
for each u ∈ V\{s}
        u.color←white;
        u.distance←∞;
s.color←gray;
s.distance←0;
Q←∅;
enqueue(Q,s);
while Q ≠ ∅
        u←dequeue(Q);
        for each v ∈ Adj[u]
                if v.color = white
                    v.color←gray;
                    v.distance←u.distance + 1;
                    enqueue(Q,v);
        u.color←black;
```

BFS assigns to each *v* value *v.distance*, the least possible number of edges on any source-to-*v* path.

Does BFS work for weighted graphs too?

BFS(*G*,*s*) - *G* is represented by the adjacency lists `Adj[·]` of its vertices

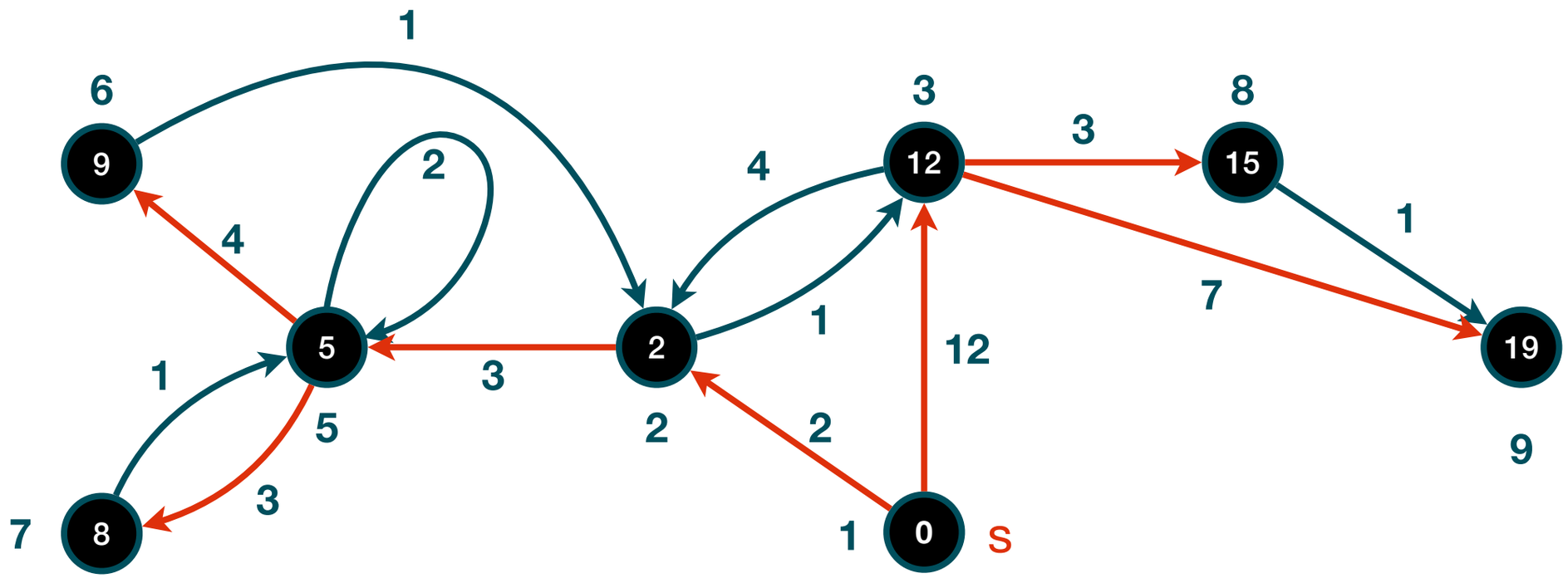
```
for each u ∈ V\{s}
        u.color←white;
        u.distance←∞;
s.color←gray;
s.distance←0;
Q←∅;
enqueue(Q,s);
while Q ≠ ∅
        u←dequeue(Q);
        for each v ∈ Adj[u]
                if v.color = white
                    v.color←gray;
                    v.distance←u.distance + w(u,v)
                    enqueue(Q,v);
        u.color←black;
```

BFS assigns to each *v* value *v.distance*, the least possible number of edges on any source-to-*v* path.

Can’t we just modify this instruction to make it work for weighted graphs?

Why does BFS not work for weighted graphs?

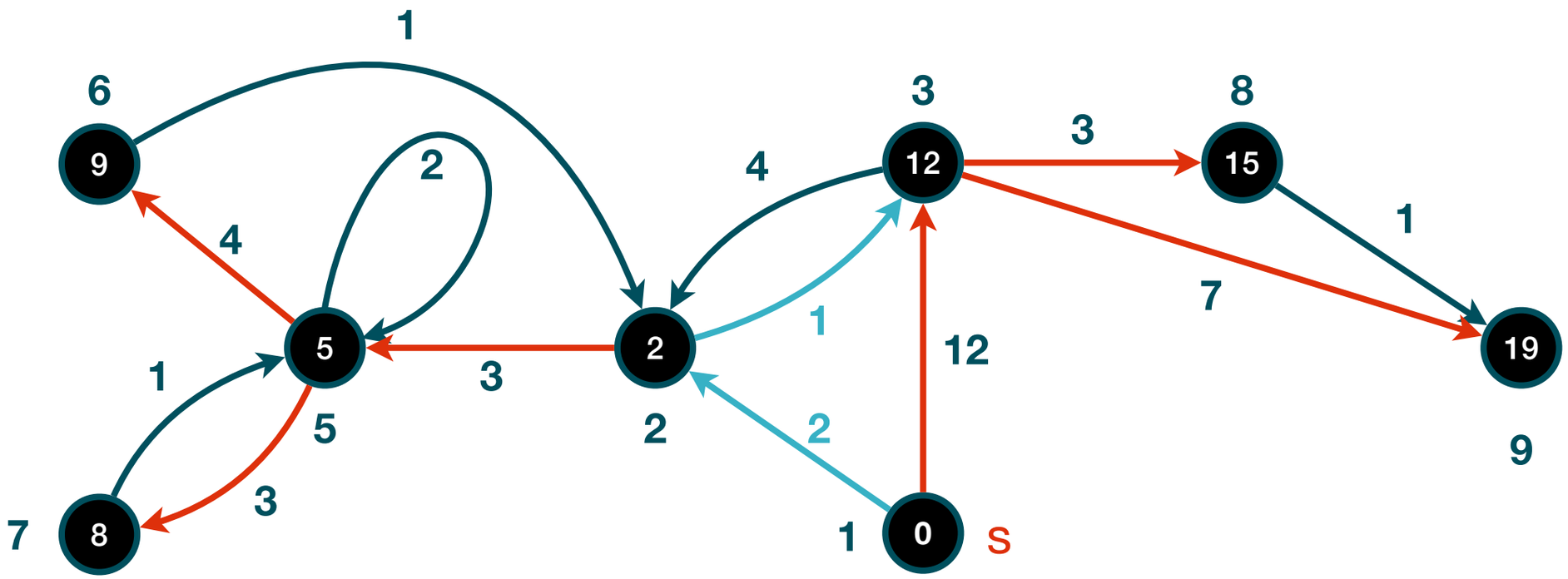
The shortest weighted path between two vertices may not be the one with the least number of edges!



Why does BFS not work for weighted graphs?

The shortest weighted path between two vertices may not be the one with the least number of edges!

The shortest path from S to vertex number 3 would be through vertex 2: the length of 1 2 3 is 2+1=3<12, even if this path has two edges in place of one.



Shortest paths on DAGs

1. Topological sort of the DAG
2. One pass over the vertices in the order given by the topological sort, starting from the source. Every node to the left of the source has infinite weight. The rest will get the weight of the shortest path.

Time complexity: $\Theta(|V| + |E|)$

Dijkstra's algorithm

If all the weights are nonnegative, we can use Dijkstra's (pronounced "Deikstra") algorithm.

Recall:

RELAX(u,v,w)
  if $v.d > u.d + w(u,v)$
    $v.d = u.d + w(u,v)$;
    $v.p \leftarrow u$;

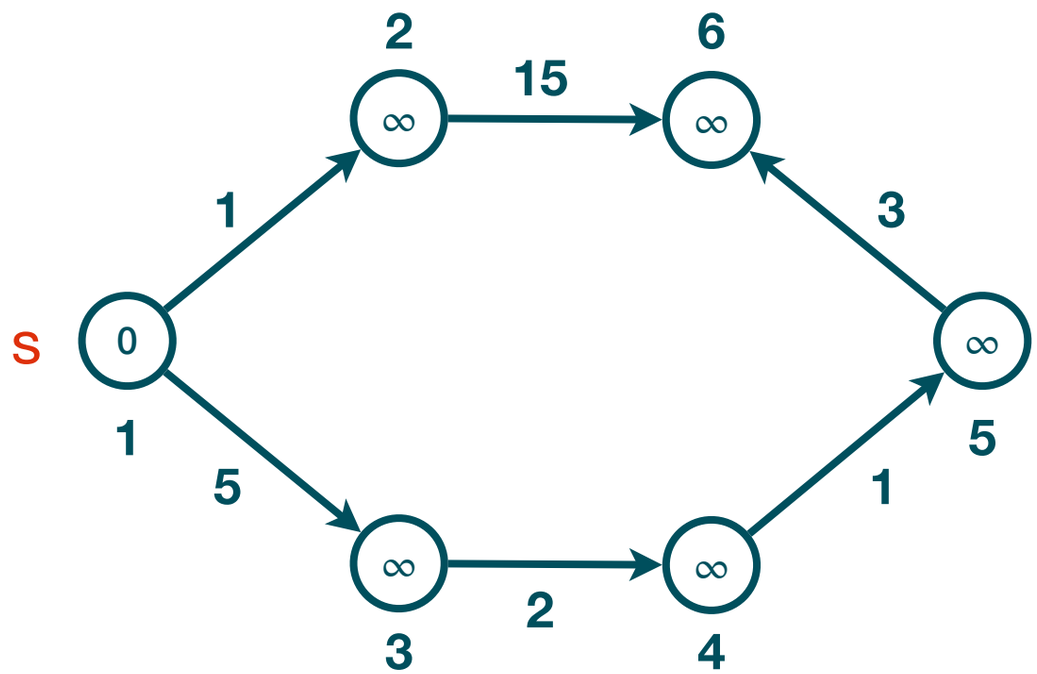
Dijkstra's algorithm

At each step, one edge is relaxed. The vertices that are still to be finalised are maintained in a min-priority queue (many different implementations are possible). S is the set of finalised vertices.

S={}

1 2 3 4 5 6

Q={0,∞,∞,∞,∞,∞}



DIJKSTRA(*G*,*w*,*s*)

INITIALISE(*G*,*s*);

S ← ∅;

Q ← V;

while Q ≠ ∅

  u ← EXTRACTMIN(Q);

  S ← SU{*u*};

  for each *v* ∈ Adj[*u*]

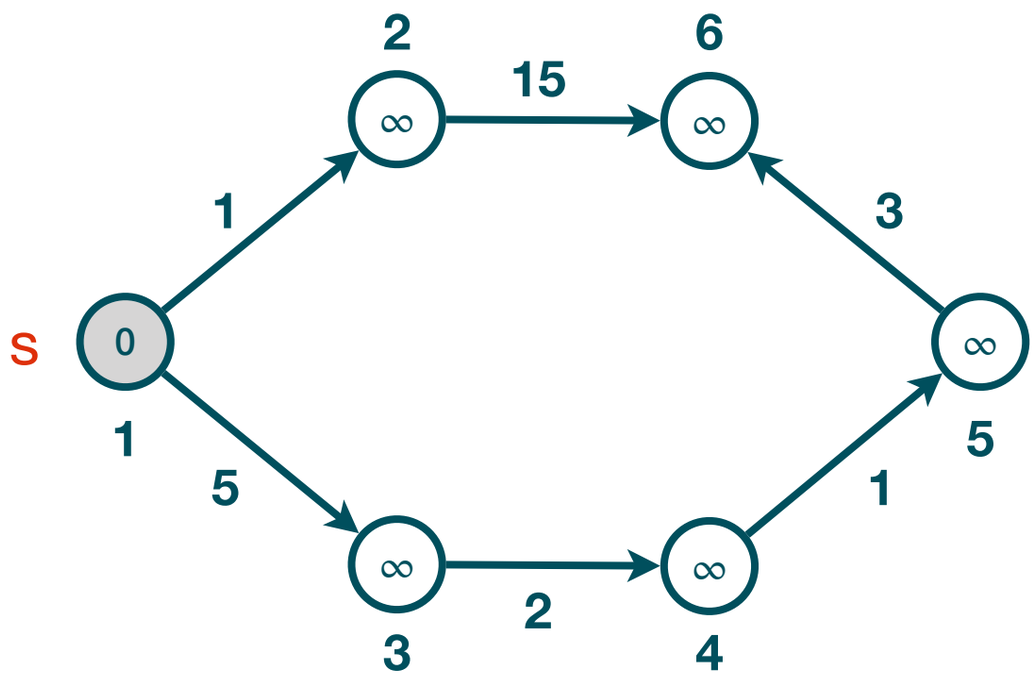
    RELAX(*u*,*v*,*w*);

Dijkstra's algorithm

At each step, one edge is relaxed. The vertices that are still to be finalised are maintained in a min-priority queue (many different implementations are possible). S is the set of finalised vertices.

S={}

	1	2	3	4	5	6
Q={	0,	∞,	∞,	∞,	∞,	∞}



DIJKSTRA(*G,w,s*)
INITIALISE(*G,s*);
S ← ∅;
Q ← V;
while Q ≠ ∅
u ← EXTRACTMIN(Q);
S ← S∪{*u*};
for each *v* ∈ Adj[*u*]
RELAX(*u,v,w*);

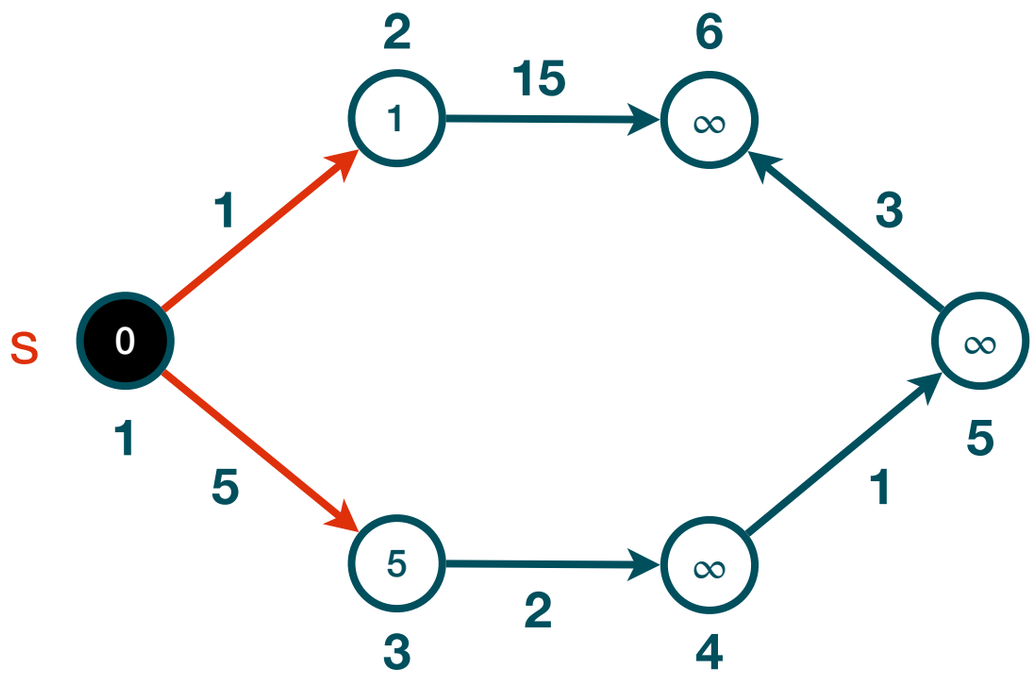
Dijkstra's algorithm

At each step, one edge is relaxed. The vertices that are still to be finalised are maintained in a min-priority queue (many different implementations are possible). S is the set of finalised vertices.

S={1}

2 3 4 5 6

Q={1,5,∞,∞,∞}




```
DIJKSTRA(G,w,s)
INITIALISE(G,s);
S ← ∅;
Q ← V;
while Q ≠ ∅
    u ← EXTRACTMIN(Q);
    S ← SU{u};
    for each v ∈ Adj[u]
        RELAX(u,v,w);
```

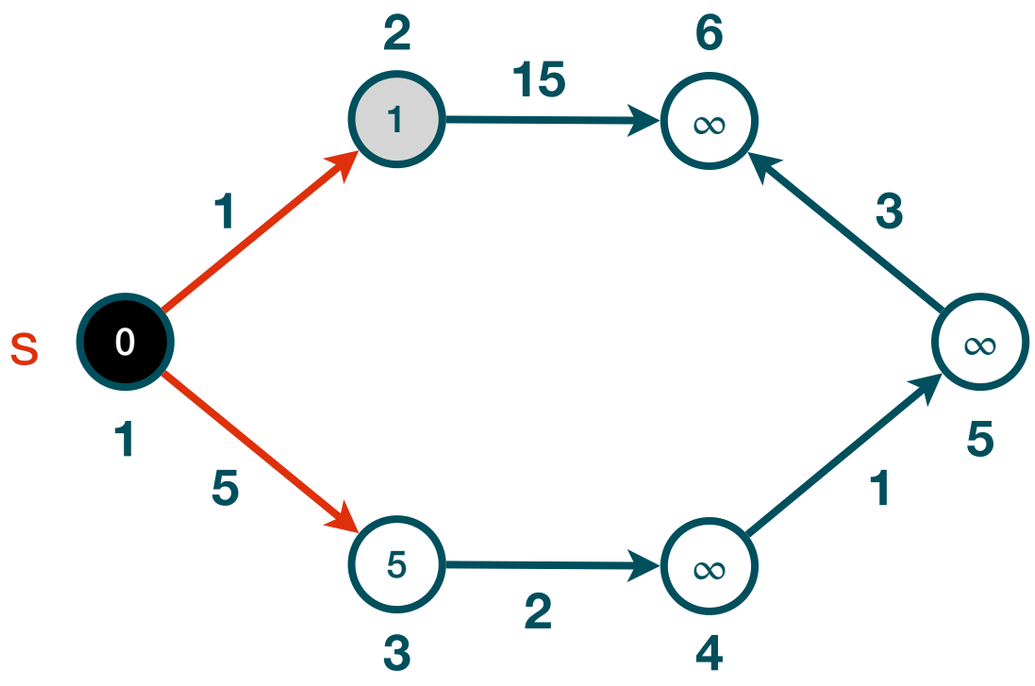
Dijkstra's algorithm

At each step, one edge is relaxed. The vertices that are still to be finalised are maintained in a min-priority queue (many different implementations are possible). S is the set of finalised vertices.

S={1}

2 3 4 5 6

Q={1,5,∞,∞,∞}




```
DIJKSTRA(G,w,s)
    INITIALISE(G,s);
    S ← ∅;
    Q ← V;
    while Q ≠ ∅
        u ← EXTRACTMIN(Q);
        S ← S∪{u};
        for each v ∈ Adj[u]
            RELAX(u,v,w);
```

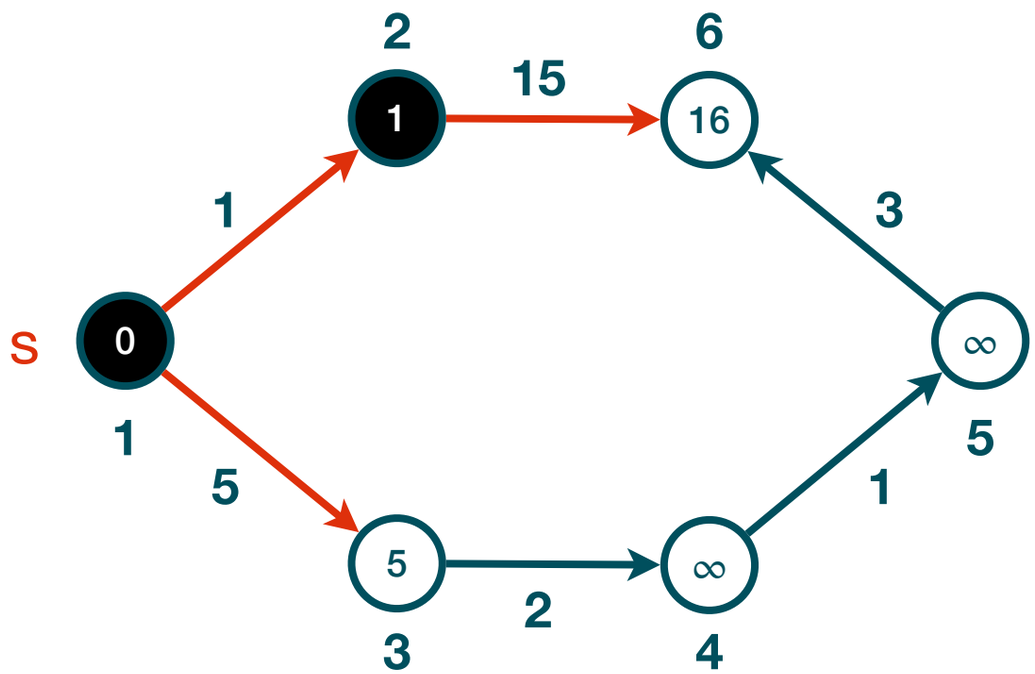
Dijkstra's algorithm

At each step, one edge is relaxed. The vertices that are still to be finalised are maintained in a min-priority queue (many different implementations are possible). S is the set of finalised vertices.

S={1,2}

3 4 5 6

Q={5,∞,∞,16}




```
DIJKSTRA(G,w,s)
INITIALISE(G,s);
S ← ∅;
Q ← V;
while Q ≠ ∅
    u ← EXTRACTMIN(Q);
    S ← SU{u};
    for each v ∈ Adj[u]
        RELAX(u,v,w);
```

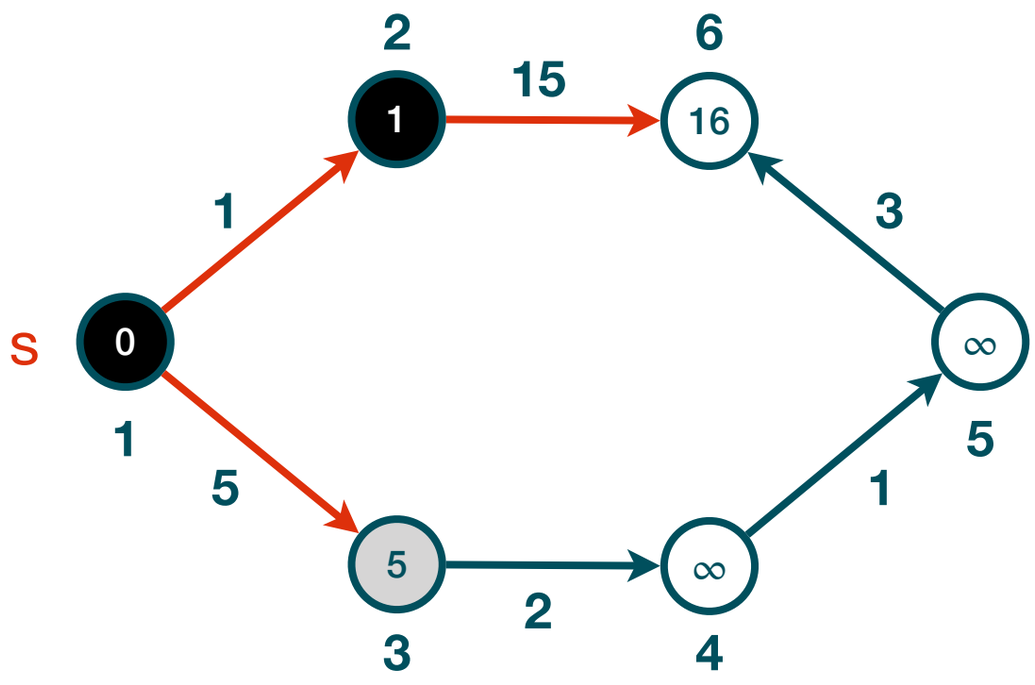
Dijkstra's algorithm

At each step, one edge is relaxed. The vertices that are still to be finalised are maintained in a min-priority queue (many different implementations are possible). S is the set of finalised vertices.

S={1,2}

3 4 5 6

Q={5,∞,∞,16}

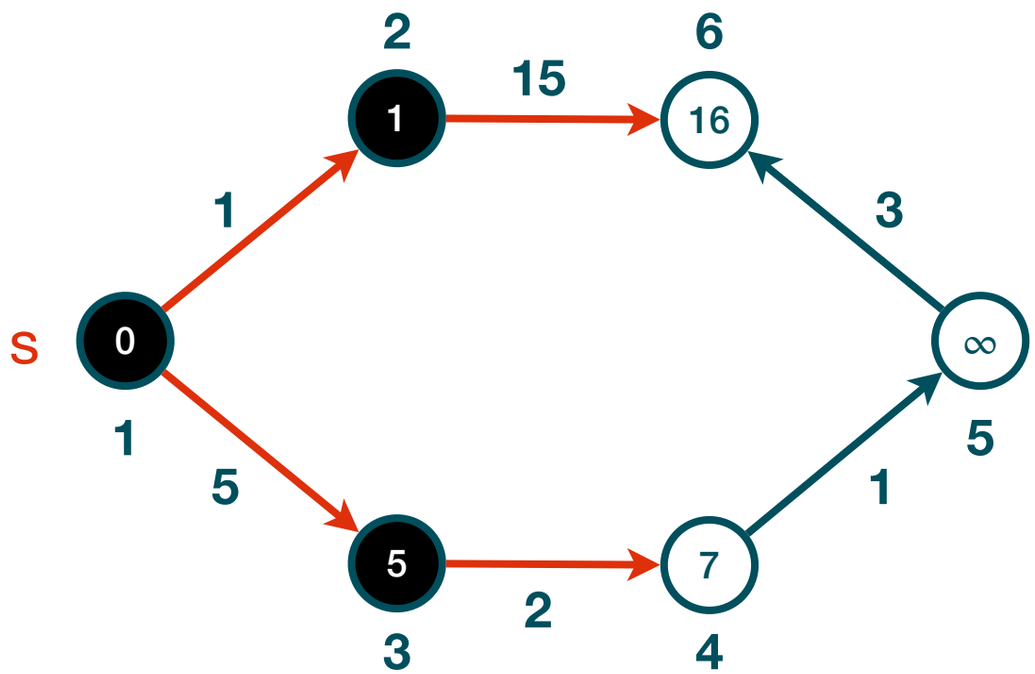



```
DIJKSTRA(G,w,s)
    INITIALISE(G,s);
    S ← ∅;
    Q ← V;
    while Q ≠ ∅
        u ← EXTRACTMIN(Q);
        S ← S∪{u};
        for each v ∈ Adj[u]
            RELAX(u,v,w);
```

Dijkstra's algorithm

At each step, one edge is relaxed. The vertices that are still to be finalised are maintained in a min-priority queue (many different implementations are possible). S is the set of finalised vertices.

S={1,2,3}
4 5 6
Q={7,∞,16}




```
DIJKSTRA(G,w,s)
INITIALISE(G,s);
S ← ∅;
Q ← V;
while Q ≠ ∅
    u ← EXTRACTMIN(Q);
    S ← SU{u};
    for each v ∈ Adj[u]
        RELAX(u,v,w);
```

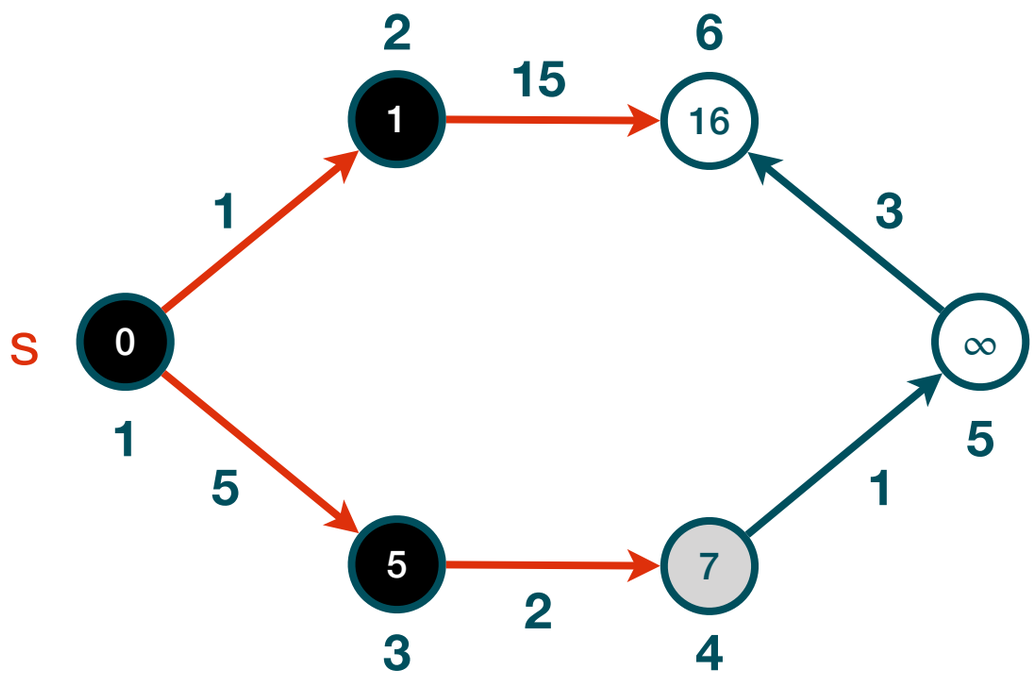
Dijkstra's algorithm

At each step, one edge is relaxed. The vertices that are still to be finalised are maintained in a min-priority queue (many different implementations are possible). S is the set of finalised vertices.

S={1,2,3}

4 5 6

Q={7,∞,16}



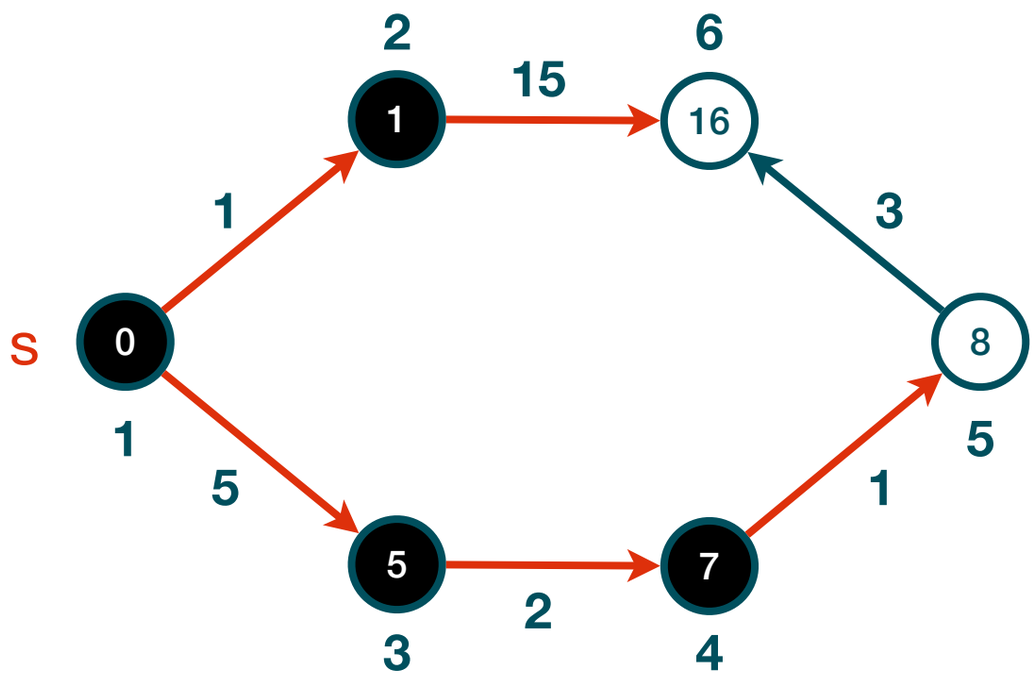

```
DIJKSTRA(G,w,s)
    INITIALISE(G,s);
    S ← ∅;
    Q ← V;
    while Q ≠ ∅
        u ← EXTRACTMIN(Q);
        S ← S∪{u};
        for each v ∈ Adj[u]
            RELAX(u,v,w);
```

Dijkstra's algorithm

At each step, one edge is relaxed. The vertices that are still to be finalised are maintained in a min-priority queue (many different implementations are possible). S is the set of finalised vertices.

S={1,2,3,4}
5 6

Q={8,16}



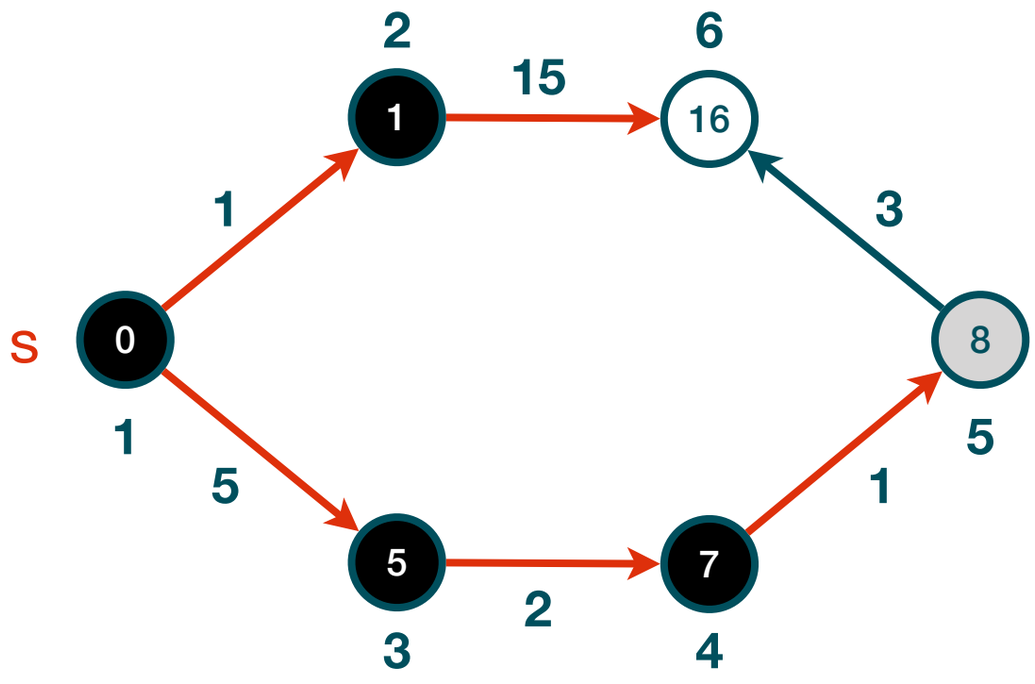

```
DIJKSTRA(G,w,s)
INITIALISE(G,s);
S ← ∅;
Q ← V;
while Q ≠ ∅
    u ← EXTRACTMIN(Q);
    S ← SU{u};
    for each v ∈ Adj[u]
        RELAX(u,v,w);
```

Dijkstra's algorithm

At each step, one edge is relaxed. The vertices that are still to be finalised are maintained in a min-priority queue (many different implementations are possible). S is the set of finalised vertices.

S={1,2,3,4}

Q={8,16}




```
DIJKSTRA(G,w,s)
    INITIALISE(G,s);
    S ← ∅;
    Q ← V;
    while Q ≠ ∅
        u ← EXTRACTMIN(Q);
        S ← S∪{u};
        for each v ∈ Adj[u]
            RELAX(u,v,w);
```

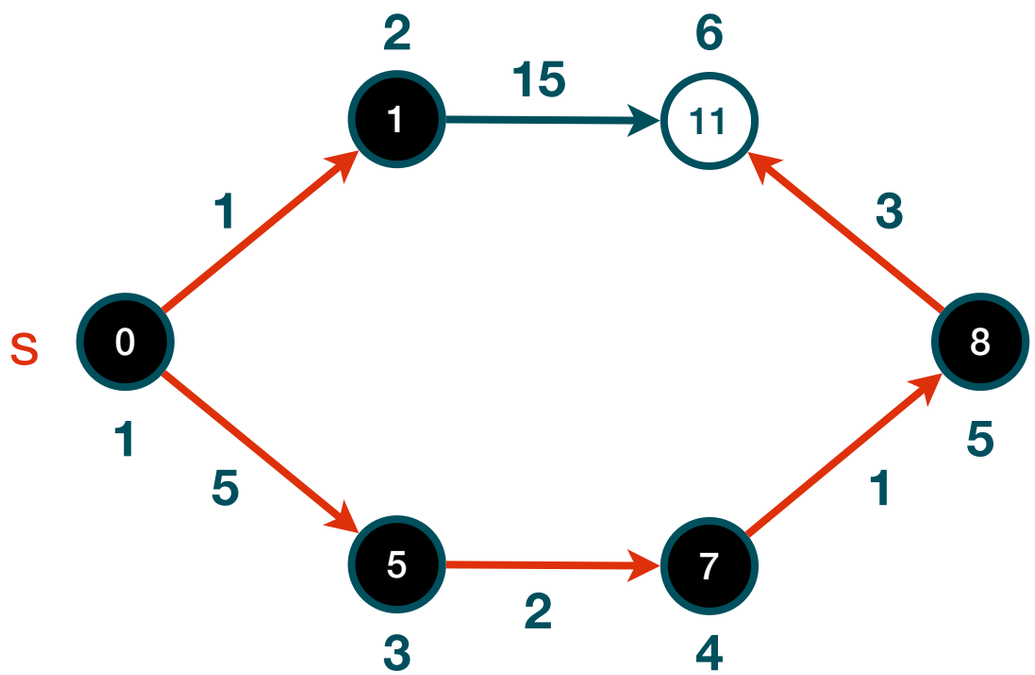
Dijkstra's algorithm

At each step, one edge is relaxed. The vertices that are still to be finalised are maintained in a min-priority queue (many different implementations are possible). S is the set of finalised vertices.

S={1,2,3,4,5}
6

Q={11}

RELAX makes 6.*d* change from 16 to 11!




```
DIJKSTRA(G,w,s)
INITIALISE(G,s);
S ← ∅;
Q ← V;
while Q ≠ ∅
    u ← EXTRACTMIN(Q);
    S ← SU{u};
    for each v ∈ Adj[u]
        RELAX(u,v,w);
```

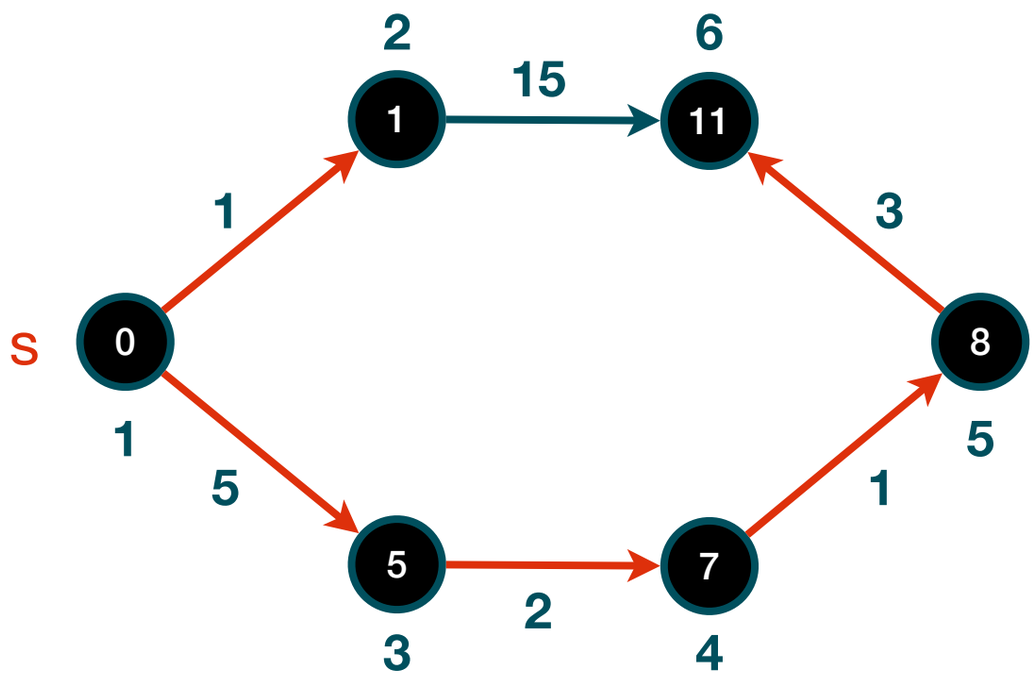
Dijkstra's algorithm

At each step, one edge is relaxed. The vertices that are still to be finalised are maintained in a min-priority queue (many different implementations are possible). S is the set of finalised vertices.

S={1,2,3,4,5,6}

Q={}

RELAX makes 6.*d* change from 16 to 11!



DIJKSTRA(*G,w,s*)
INITIALISE(*G,s*);
S ← ∅;
Q ← V;
while Q ≠ ∅
  u ← EXTRACTMIN(Q);
  S ← S∪{*u*};
  for each *v* ∈ Adj[*u*]
    RELAX(*u,v,w*);

Dijkstra's algorithm: complexity

Time complexity: $\Theta(|V|) + T_B(|V|) + |V| \cdot T_E(|V|) + |E| \cdot T_R(|V|)$

Queue data structure	$T_B(n)$	$T_E(n)$	$T_R(n)$	$T_D(G)$
Arrays	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(\|E\|+\|V\|^2)$
Binary Heaps	$\Theta(n)$	$O(\log n)$	$O(\log n)$	$O((\|E\|+\|V\|)\log \|V\|)$
Fibonacci Heaps	$\Theta(n)$	$O(\log n)$	$\Theta(1)$	$O(\|E\|+\|V\|\log \|V\|)$

Bellman-Ford algorithm

Can be applied to general graphs, even with negative cycles.

1. Do |V|-1 passes of the graph. At each pass, relax all the edges.

2. Once this is done, with a single pass check, for each edge, if it can be further relaxed. If an edge (u,v) can still be relaxed, mark the weight of v as undefined (it is reachable using a negative cycle!)

Time complexity: $\Theta(|V||E|) = O(|V|^3)$

Exercises

Cormen 24.3-6: We are given a directed graph G which each edge (u,v) has an associated value r(u,v), which is a real number in the range [0,1] that represents the reliability of a communication channel from vertex u to vertex v. We interpret r(u,v) as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices. *(Hint: either modify Dijkstra or transform the weights…)*

All-Pairs Shortest Paths

Chapter 25.2 of Cormen's book

Giulia Bernardini
giulia.bernardini@units.it

Algorithmic Design
a.y. 2022/2023

Dynamic programming: a summary

Basic steps:

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution, typically bottom-up
4. Construct an optimal solution from computed information

Dynamic programming vs divide-and-conquer

Divide-and-conquer algorithms partition the problem into disjoint subproblems, solve them recursively, and combine their solutions.

If the subproblems overlap, a divide-and-conquer algorithm would solve each subproblem repeatedly. A dynamic-programming algorithm solves each subproblem just once and saves its answer in a table, to be used over and over again.

Dynamic programming: a summary

To which problems does dynamic programming apply?

It applies to optimization problems with an optimal substructure: an optimal solution to the problem contains optimal solutions to subproblems.

Moreover, these subproblems must be independent: the solution to one subproblem does not affect the solution to another subproblem of the same problem.

All-Pairs Shortest Paths (APSP)

Input: a (weighted or not weighted, directed or undirected) graph G=(V,E,W?)

Output: a shortest path between each possible pair of vertices

Baseline solution: apply Dijkstra's algorithm |V| times, each time picking a different vertex as a source. This would take $O(|V|\cdot(|E|+|V|\log|V|))$ time (using Fibonacci heaps).

This only works for graphs with nonnegative weights.

For general graphs, we could apply Bellman-Ford |V| times: this would require $\Theta(|V|^2|E|)$ time, which is $\Theta(|V|^4)$ for dense graphs. Can we do better?

A recursive solution

Let $d_{ij}^{(k)}$ be the length of a shortest path from *i* to *j* with all its intermediate vertices in the set {1,2,…,k}. We can incrementally build shortest paths by admitting a new untouched intermediate vertex at each step, using the following recursion:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} \text{ if } k=0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\} \text{ if } k \geq 1 \end{cases}$$

Because for any path in G the intermediate vertices are in {1,2,…,n}, the matrix $D^{(n)}[i,j]=d_{ij}^{(n)}$ gives the desired output.

The Floyd-Warshall algorithm computes the whole sequence of matrixes $D^{(0)},D^{(1)},\ldots,D^{(n)}$

A dynamic programming solution: the Floyd-Warshall algorithm

FLOYD-WARSHALL(W) - W is the $|V| \times |V|$ weight matrix

```
D(0) ← W;
for k=1,…, |V|
    allocate a new |V| × |V| matrix D(k);
    for i=1,…, |V|
        for j=1,…, |V|
            dij(k) = min{dij(k-1), dik(k-1)+dkj(k-1)};
return D(|V|);
```

Time complexity: $\Theta(|V|^3)$

What about the predecessor matrix?

We can compute it at the same time of the weights.

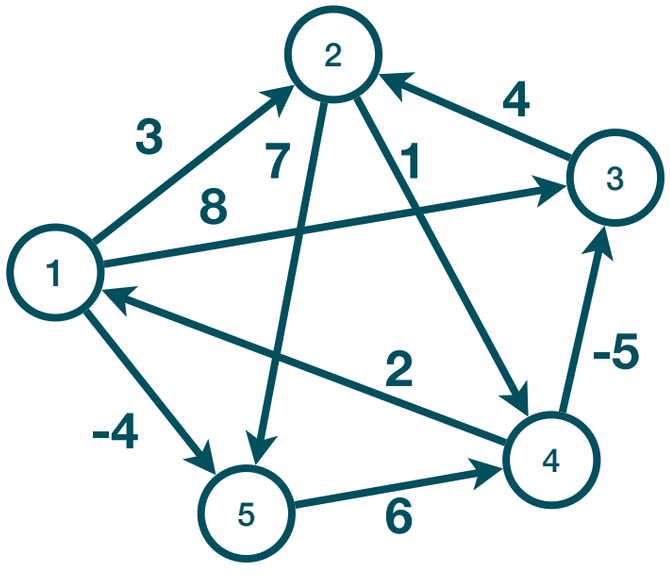
We define $\pi_{ij}^{(k)}$ as the predecessor of vertex *j* on a shortest path from vertex *i* with all intermediate vertices in the set {1,…,k}.

We can give a recursive formulation:

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \text{ (shortest p. doesn't use k)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \text{ (shortest path uses k)} \end{cases}$$

Floyd-Warshall algorithm: an example

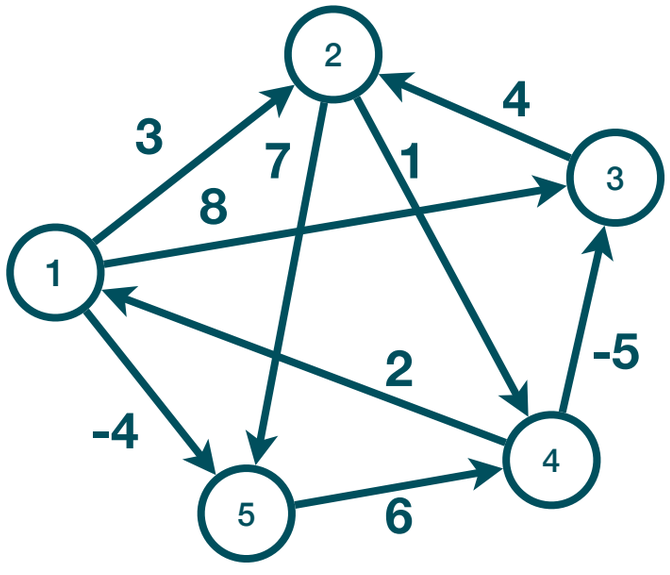


$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\} & \text{if } k\geq 1 \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)}+d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)}+d_{kj}^{(k-1)} \end{cases}$$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Floyd-Warshall algorithm: an example



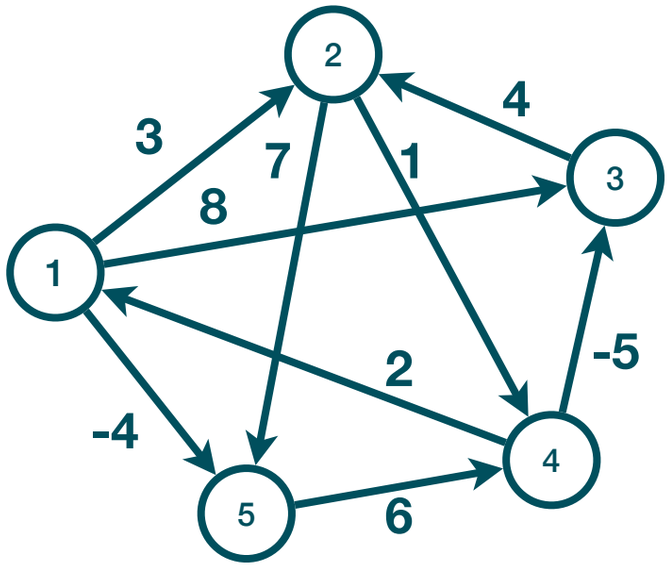
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\} & \text{if } k\geq 1 \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)}+d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)}+d_{kj}^{(k-1)} \end{cases}$$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Floyd-Warshall algorithm: an example



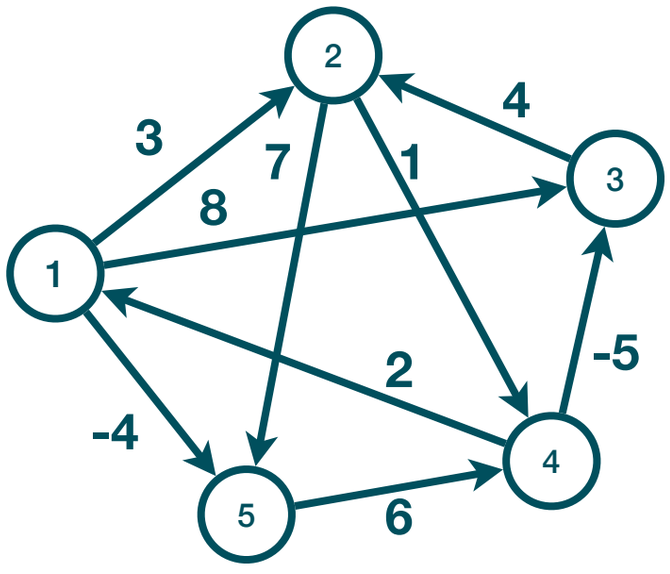
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\} & \text{if } k\geq 1 \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)}+d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)}+d_{kj}^{(k-1)} \end{cases}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Floyd-Warshall algorithm: an example



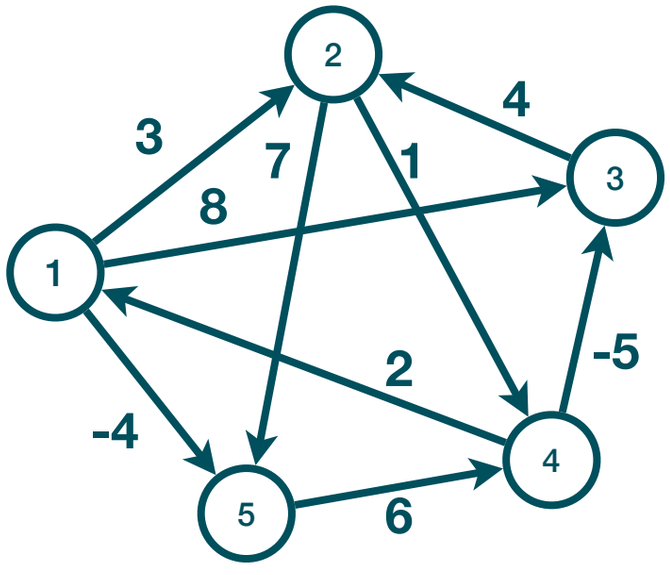
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\} & \text{if } k\geq 1 \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)}+d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)}+d_{kj}^{(k-1)} \end{cases}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Floyd-Warshall algorithm: an example



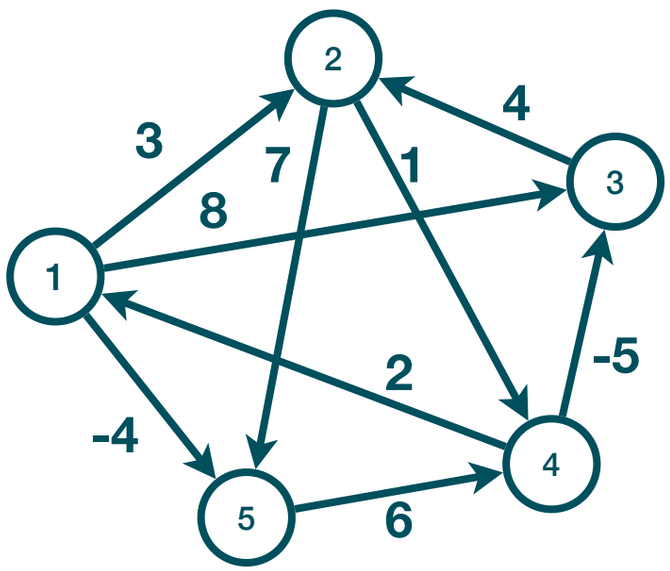
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\} & \text{if } k\geq 1 \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)}+d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)}+d_{kj}^{(k-1)} \end{cases}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Floyd-Warshall algorithm: an example



$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\} & \text{if } k\geq 1 \end{cases}$$

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)}+d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)}+d_{kj}^{(k-1)} \end{cases}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

An interactive tool

If you want to test the Floyd-Warshall algorithm:

https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-floyd-warshall/index_en.html

Exercises

Cormen Problem 24-3: *Arbitrage* is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 49 Indian rupees, 1 Indian rupee buys 2 Japanese yen, and 1 Japanese yen buys 0.0107 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy 49 x 2 x 0.0107 = 1.0486 U.S. dollars, thus turning a profit of 4.86 percent.

Suppose that we are given n currencies $c_1, c_2, \ldots, c_n$ and an n x n table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j .

Give an efficient algorithm to determine whether or not there exists a sequence of currencies $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ such that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdot \ldots \cdot R[i_{k-1}, i_k] > 1$$