Bellman-Ford and Dynamic Programming

Lecture 18

Part I

No negative edges: Dijkstra

Dijkstra's Algorithm

```
Initialize for each node v, dist(s, v) = ∞
Initialize X = ∅, dist(s, s) = 0
for i = 1 to |V| do
    Let v be such that dist(s, v) = min_{u∈V−X} dist(s, u)
    X = X ∪ {v}
    for each u in Adj(v) do
        dist(s, u) = min(dist(s, u), dist(s, v) + ℓ(v, u))
```

Priority Queues to maintain *dist* values for faster running time

1. Using heaps and standard priority queues: $O((m+n)\log n)$
2. Best-first-search

Dijkstra's Algorithm using Priority Queues

$Q \leftarrow$ makePQ()
insert(Q, $(s, 0)$)
for each node $u \neq s$ **do**
    insert(Q, (u, ∞))
$X \leftarrow \emptyset$
for $i = 1$ to $|V|$ **do**
    $(v, \mathrm{dist}(s, v)) =$ *extractMin(Q)*
    $X = X \cup \{v\}$
    for each u in $\mathrm{Adj}(v)$ **do**
        decreaseKey$\Big(Q, \big(u, \min\big(\mathrm{dist}(s, u),\ \mathrm{dist}(s, v) + \ell(v, u)\big)\big)\Big)$.

Priority Queue operations:

1. $O(n)$ **insert** operations
2. $O(n)$ **extractMin** operations
3. $O(m)$ **decreaseKey** operations

Implementing Priority Queues via Heaps

Using Heaps

Store elements in a heap based on the key value

1. All operations can be done in $O(\log n)$ time

Implementing Priority Queues via Heaps

Using Heaps

Store elements in a heap based on the key value

1. All operations can be done in $O(\log n)$ time

Dijkstra's algorithm can be implemented in $O((n+m)\log n)$ time.

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $O(\log n)$ time
2. **decreaseKey** in $O(1)$ *amortized* time:

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $O(\log n)$ time
2. **decreaseKey** in $O(1)$ *amortized* time: ℓ **decreaseKey** operations for $\ell \geq n$ take *together* $O(\ell)$ time

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $O(\log n)$ time
2. **decreaseKey** in $O(1)$ *amortized* time: ℓ **decreaseKey** operations for $\ell \geq n$ take *together* $O(\ell)$ time
3. Relaxed Heaps: **decreaseKey** in $O(1)$ worst case time but at the expense of **meld** (not necessary for Dijkstra's algorithm)

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $O(\log n)$ time
2. **decreaseKey** in $O(1)$ *amortized* time: ℓ **decreaseKey** operations for $\ell \geq n$ take *together* $O(\ell)$ time
3. Relaxed Heaps: **decreaseKey** in $O(1)$ worst case time but at the expense of **meld** (not necessary for Dijkstra's algorithm)

1. Dijkstra's algorithm can be implemented in $O(n \log n + m)$ time.

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $O(\log n)$ time
2. **decreaseKey** in $O(1)$ *amortized* time: ℓ **decreaseKey** operations for $\ell \geq n$ take *together* $O(\ell)$ time
3. Relaxed Heaps: **decreaseKey** in $O(1)$ worst case time but at the expense of **meld** (not necessary for Dijkstra's algorithm)

1. Dijkstra's algorithm can be implemented in $O(n \log n + m)$ time.
2. Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps (European Symposium on Algorithms, September 2009!)

Key takeaways of Dijkstra

1. Non-negative edges: In order to get to t, only need nodes whose shortest distance is smaller than t.

Key takeaways of Dijkstra

1. Non-negative edges: In order to get to t, only need nodes whose shortest distance is smaller than t.
    - The intermediate set X keeps the $i - 1$ closest nodes

Key takeaways of Dijkstra

1. Non-negative edges: In order to get to t, only need nodes whose shortest distance is smaller than t.
   - The intermediate set X keeps the $i - 1$ closest nodes
   - Give us an evaluation order: $d'(s, u)$ only updated when v is added to X, and $u \in Adj(v)$ and $u \in V - X$

Key takeaways of Dijkstra

1. Non-negative edges: In order to get to t, only need nodes whose shortest distance is smaller than t.
   - The intermediate set X keeps the $i-1$ closest nodes
   - Give us an evaluation order: $d'(s,u)$ only updated when v is added to X, and $u \in Adj(v)$ and $u \in V - X$
   - In particular, once a node is in X, $d'(s,u)$ no longer changes as $d'(s,u) = d(s,u)$, and it is never updated again

Key takeaways of Dijkstra

1. Non-negative edges: In order to get to t, only need nodes whose shortest distance is smaller than t.
   - The intermediate set X keeps the $i-1$ closest nodes
   - Give us an evaluation order: $d'(s,u)$ only updated when v is added to X, and $u \in Adj(v)$ and $u \in V - X$
   - In particular, once a node is in X, $d'(s,u)$ no longer changes as $d'(s,u) = d(s,u)$, and it is never updated again

2. How to recognize the i-th closest node?

$$d'(s,u) = min\Big(d'(s,u),\ \mathrm{dist}(s,v) + \ell(v,u)\Big)$$

Key takeaways of Dijkstra

1. Non-negative edges: In order to get to t, only need nodes whose shortest distance is smaller than t.
   - The intermediate set X keeps the $i-1$ closest nodes
   - Give us an evaluation order: $d'(s,u)$ only updated when v is added to X, and $u \in Adj(v)$ and $u \in V - X$
   - In particular, once a node is in X, $d'(s,u)$ no longer changes as $d'(s,u) = d(s,u)$, and it is never updated again

2. How to recognize the i-th closest node?

$$d'(s,u) = min\Big(d'(s,u),\ \mathrm{dist}(s,v) + \ell(v,u)\Big)$$

   - $d'(s,u) \geq d(s,u)$

Key takeaways of Dijkstra

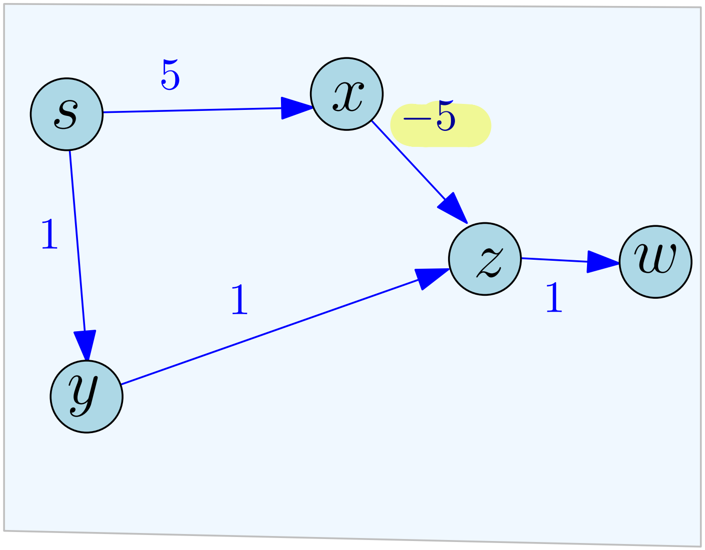
1. Non-negative edges: In order to get to t, only need nodes whose shortest distance is smaller than t.
   - The intermediate set X keeps the $i-1$ closest nodes
   - Give us an evaluation order: $d'(s,u)$ only updated when v is added to X, and $u \in Adj(v)$ and $u \in V - X$
   - In particular, once a node is in X, $d'(s,u)$ no longer changes as $d'(s,u) = d(s,u)$, and it is never updated again

2. How to recognize the i-th closest node?
$$d'(s,u) = min\Big(d'(s,u),\ \mathrm{dist}(s,v) + \ell(v,u)\Big)$$
   - $d'(s,u) \geq d(s,u)$
   - $d'(s,v) = \min_{u \in V-X} d'(s,u)$ is the i-th closest node, and $d'(s,v) = d(s,v)$

Part II

Negative Edges: Bellman-Ford

What are the distances computed by Dijkstra's algorithm?

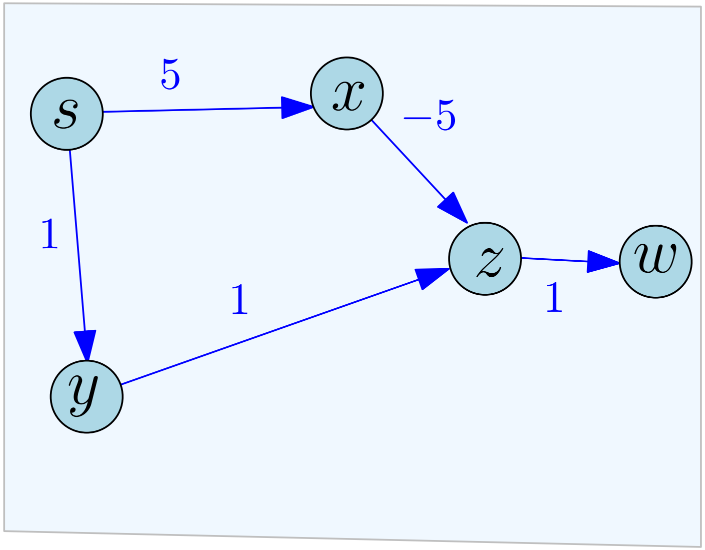


The distance as computed by Dijkstra algorithm starting from s:

- (A) $s = 0$, $x = 5$, $y = 1$, $z = 0$.
- (B) $s = 0$, $x = 1$, $y = 2$, $z = 5$.
- (C) $s = 0$, $x = 5$, $y = 1$, $z = 2$.
- (D) IDK.

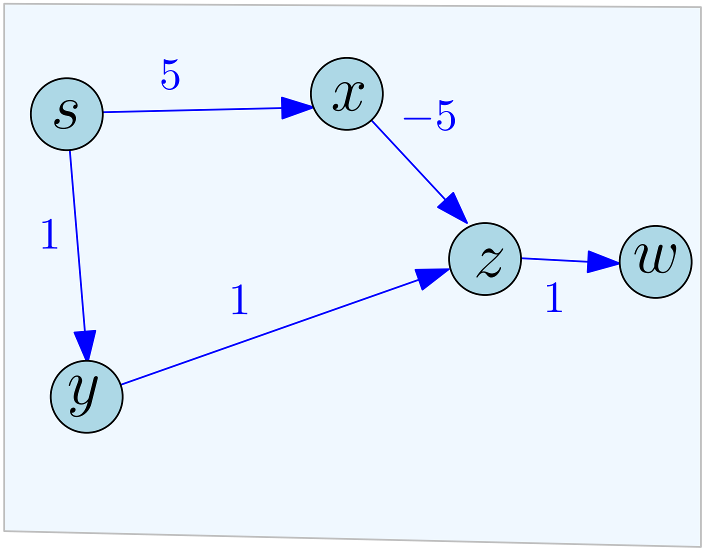
Dijkstra's Algorithm and Negative Lengths

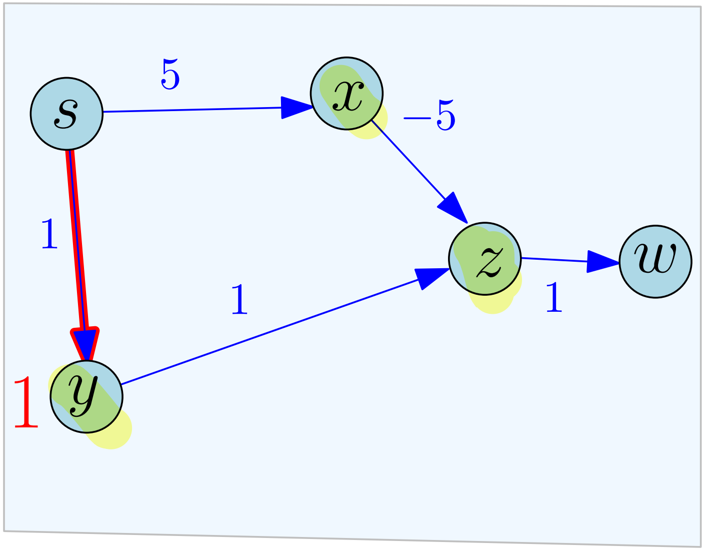
With negative length edges, Dijkstra's algorithm can fail



Dijkstra's Algorithm and Negative Lengths

With negative length edges, Dijkstra's algorithm can fail





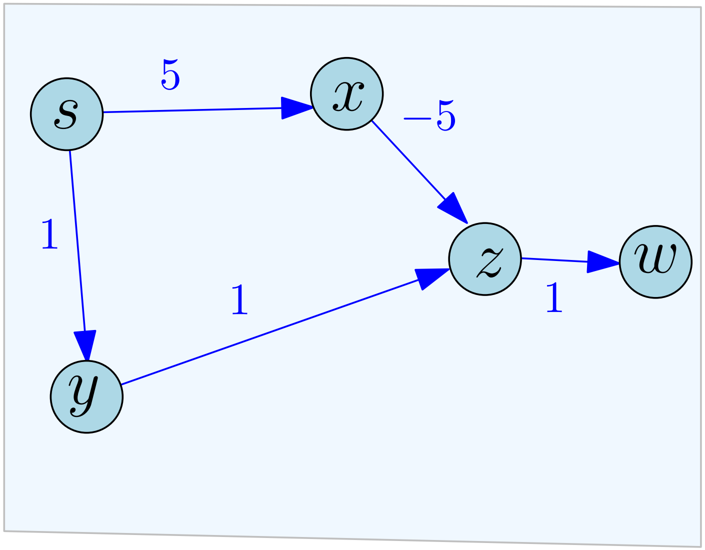
$X = \{s, y\}$

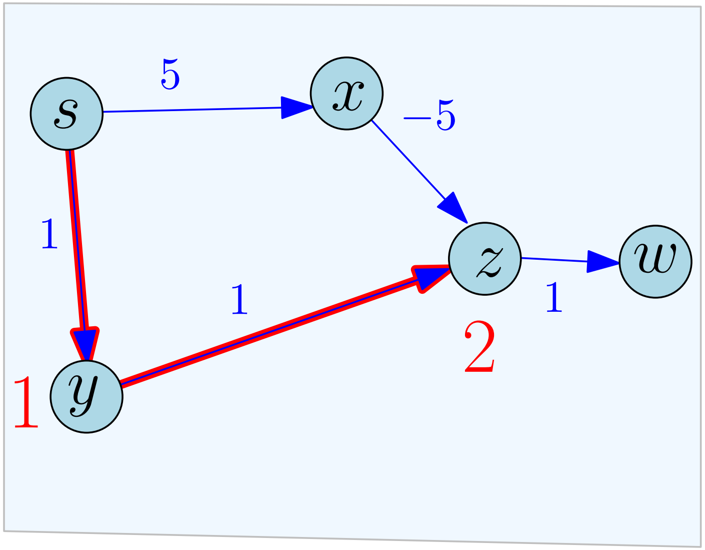
$s \to y \to z \quad d'(s,z) = 2$

$< d'(s,x)$

Dijkstra's Algorithm and Negative Lengths

With negative length edges, Dijkstra's algorithm can fail





$X = \{ s, y, z \}$

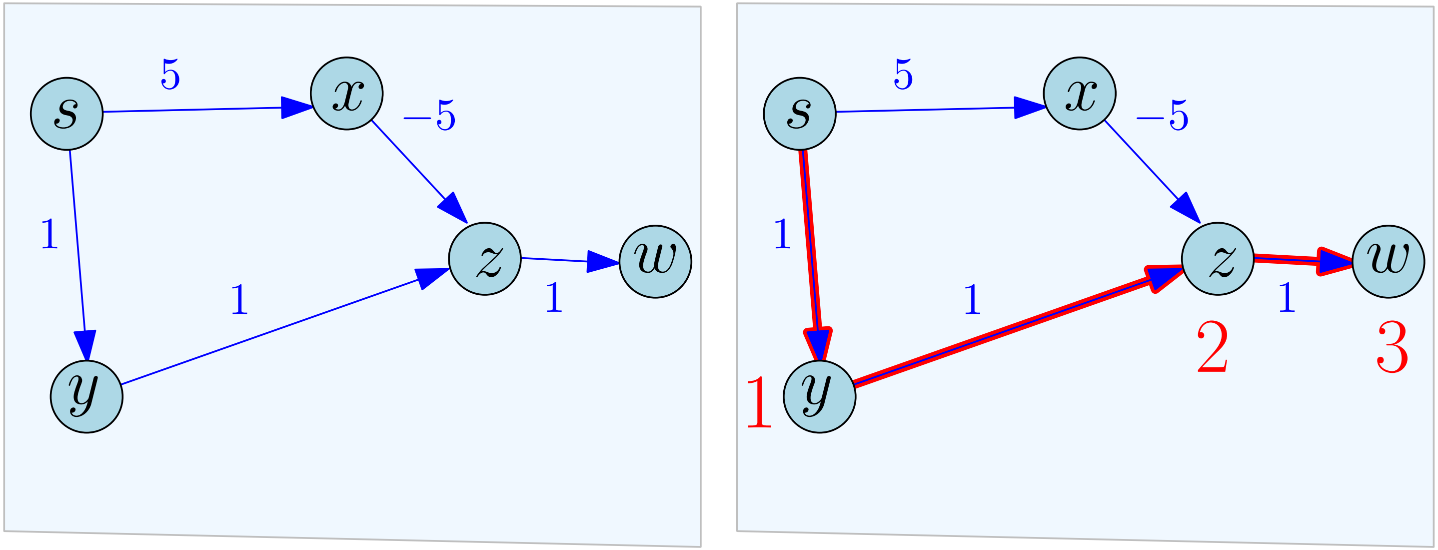
x, w

$d'(s,w) = 3$

$< d'(s,x)$

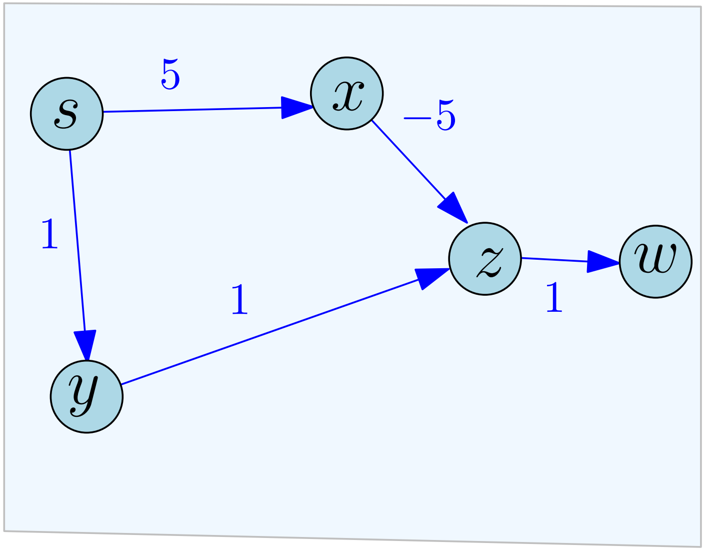
Dijkstra's Algorithm and Negative Lengths

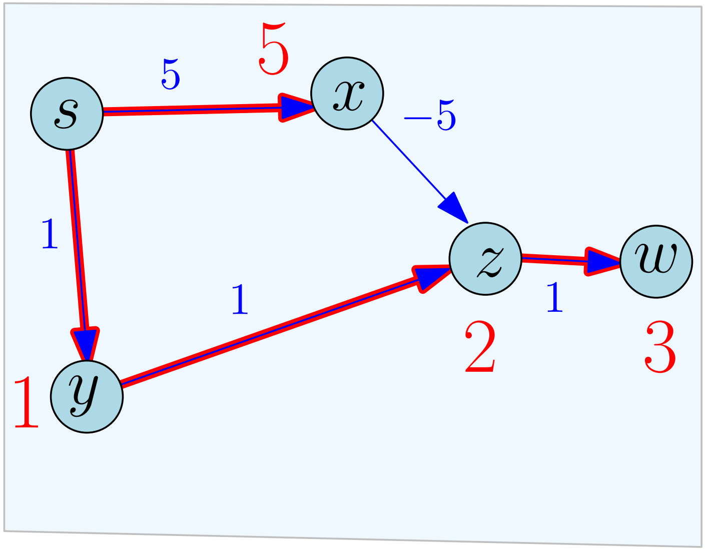
With negative length edges, Dijkstra's algorithm can fail



Dijkstra's Algorithm and Negative Lengths

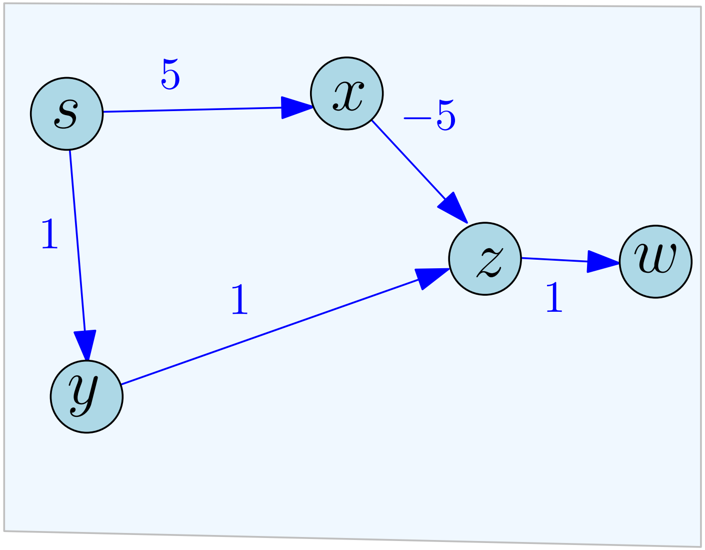
With negative length edges, Dijkstra's algorithm can fail

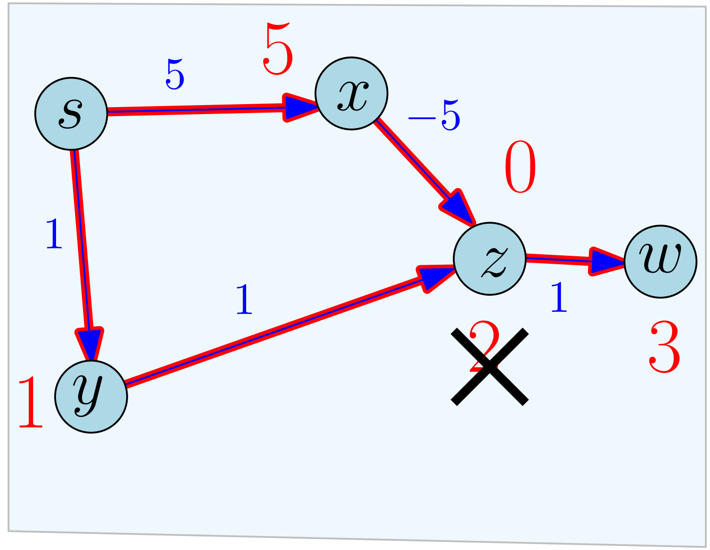




Dijkstra's Algorithm and Negative Lengths

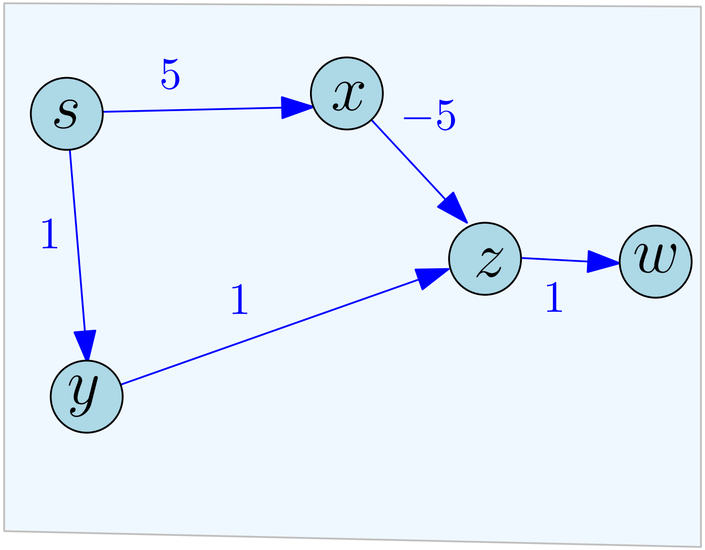
With negative length edges, Dijkstra's algorithm can fail

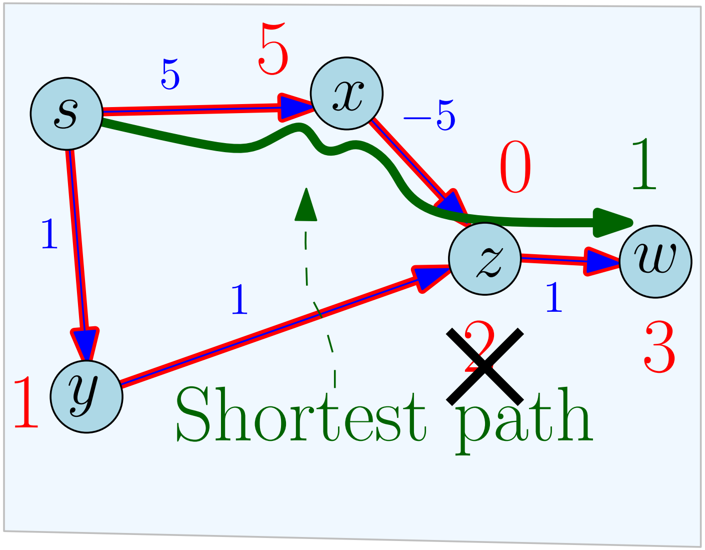




Dijkstra's Algorithm and Negative Lengths

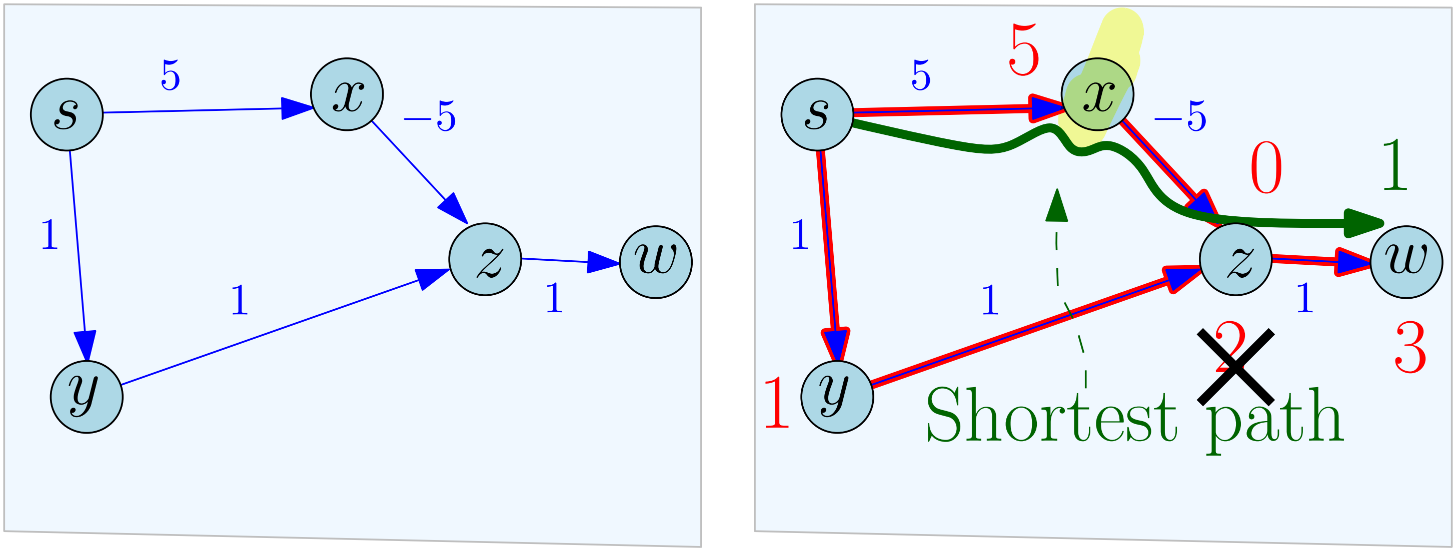
With negative length edges, Dijkstra's algorithm can fail





Dijkstra's Algorithm and Negative Lengths

With negative length edges, Dijkstra's algorithm can fail



False assumption: Dijkstra's algorithm is based on the assumption that if $s = v_0 \rightarrow v_1 \rightarrow v_2 \ldots \rightarrow v_k$ is a shortest path from s to v_k then $dist(s, v_i) \leq dist(s, v_{i+1})$ for $0 \leq i < k$. Holds true only for non-negative edge lengths.

$d(s, x) > d(s, z)$

Anything we can learn from Dijkstra?

$$d'(s,u) = min\Big(d'(s,u),\ \mathrm{dist}(s,v) + \ell(v,u)\Big)$$

- $d'(s,u) \geq d(s,u)$ still true.

Anything we can learn from Dijkstra?

$$d'(s,u) = min\Big(d'(s,u),\ \mathrm{dist}(s,v) + \ell(v,u)\Big)$$

- $d'(s,u) \geq d(s,u)$ still true.

if $s = v_0 \to v_1 \to v_2 \ldots \to v_k$ is a shortest path from s to v_k

- for $1 \leq i < k$: $s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ is a shortest path from s to v_i, i.e. subpath of a shortest path is still a shortest path.
- Not true: $dist(s, v_i) \leq dist(s, v_{i+1})$, the intermediate set is no longer X; in fact, it can be anything

Anything we can learn from Dijkstra?

$$d'(s,u) = min\Big(d'(s,u),\ \mathrm{dist}(s,v) + \ell(v,u)\Big)$$

- $d'(s,u) \geq d(s,u)$ still true.

if $s = v_0 \to v_1 \to v_2 \ldots \to v_k$ is a shortest path from s to v_k

- for $1 \leq i < k$: $s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ is a shortest path from s to v_i, i.e. subpath of a shortest path is still a shortest path.
- Not true: $dist(s, v_i) \leq dist(s, v_{i+1})$, the intermediate set is no longer X; in fact, it can be anything

Solution: Update all edges $|V| - 1$ times!

Bellman-Ford Algorithm

```
for each u ∈ V do
    d(u) ← ∞
d(s) ← 0

for k = 1 to n − 1 do
    for each v ∈ V do
        for each edge (u, v) ∈ In(v) do
            d(v) = min{d(v), d(u) + ℓ(u, v)}

for each v ∈ V do
    dist(s, v) ← d(v)
```

Running time: $O(mn)$

Part III

Bellman-Ford and DP

Shortest Paths and Recursion

1. Compute the shortest path distance from s to t recursively?
2. What are the smaller sub-problems?

Shortest Paths and Recursion

1. Compute the shortest path distance from s to t recursively?
2. What are the smaller sub-problems?

Lemma

Let G be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from s to v_k then for $1 \leq i < k$:

1. *$s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from s to v_i*

Shortest Paths and Recursion

1. Compute the shortest path distance from s to t recursively?
2. What are the smaller sub-problems?

Lemma

Let G be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from s to v_k then for $1 \leq i < k$:

1. *$s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from s to v_i*

Sub-problem idea: paths of fewer hops/edges

Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source s.

$d(v, k)$: shortest path length from s to v using at most k edges.

Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source s.

$d(v, k)$: shortest path length from s to v using at most k edges.

Note: $dist(s, v) = d(v, n - 1)$.

Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source s.
$d(v, k)$: shortest path length from s to v using at most k edges.
Note: $dist(s, v) = d(v, n - 1)$.

Recursion for $d(v, k)$:

Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source s.
$d(v, k)$: shortest path length from s to v using at most k edges.
Note: $dist(s, v) = d(v, n-1)$.

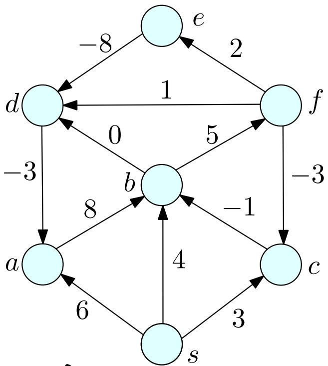
Recursion for $d(v, k)$:

$$d(v,k) = \min \begin{cases} \min_{u \in In(v)} (d(u, k-1) + \ell(u, v)). \\ d(v, k-1) \end{cases}$$

at most k-1 edges

Base case: $d(s, 0) = 0$ and $d(v, 0) = \infty$ for all $v \neq s$.

Example



	0	1	2	3	4	5	6
s	0	0	0	0	0	0	0
a	∞	6	6	1	−1	−1	−2
b	∞	4	2	2	2	2	2
c	∞	3	3	3	3	3	3
d	∞	∞	4	2	2	1	1
e	∞	∞	∞	11	9	9	9
f	∞	∞	9	7	7	7	7

s $\xrightarrow{6}$ a
d $\xrightarrow{-3}$ a

s $\xrightarrow{4}$ b
a $\xrightarrow{8}$ b
c $\xrightarrow{-1}$ b

s $\xrightarrow{3}$ c
f $\xrightarrow{-3}$ c

b $\xrightarrow{0}$ d
f $\xrightarrow{1}$ d
e $\xrightarrow{-8}$ d

f $\xrightarrow{2}$ e

b $\xrightarrow{5}$ f

Bellman-Ford Algorithm

```
for each u ∈ V do
    d(u, 0) ← ∞
d(s, 0) ← 0

for k = 1 to n − 1 do
        for each v ∈ V do
            d(v, k) ← d(v, k − 1)
            for each edge (u, v) ∈ In(v) do
                d(v, k) = min{d(v, k), d(u, k − 1) + ℓ(u, v)}

for each v ∈ V do
        dist(s, v) ← d(v, n − 1)
```

Bellman-Ford Algorithm

```
for each u ∈ V do
    d(u, 0) ← ∞
d(s, 0) ← 0

for k = 1 to n − 1 do
        for each v ∈ V do
            d(v, k) ← d(v, k − 1)
            for each edge (u, v) ∈ In(v) do
                d(v, k) = min{d(v, k), d(u, k − 1) + ℓ(u, v)}

for each v ∈ V do
        dist(s, v) ← d(v, n − 1)
```

Running time:

Bellman-Ford Algorithm

for each $u \in V$ **do**
    $d(u,0) \leftarrow \infty$
$d(s,0) \leftarrow 0$

for $k = 1$ to $n-1$ **do**
    for each $v \in V$ **do**
        $d(v,k) \leftarrow d(v,k-1)$
        for each edge $(u,v) \in In(v)$ **do**
            $d(v,k) = \min\{d(v,k), d(u,k-1) + \ell(u,v)\}$

for each $v \in V$ **do**
    $\text{dist}(s,v) \leftarrow d(v,n-1)$

Running time: $O(mn)$

Bellman-Ford Algorithm

```
for each u ∈ V do
    d(u, 0) ← ∞
d(s, 0) ← 0

for k = 1 to n − 1 do
        for each v ∈ V do
            d(v, k) ← d(v, k − 1)
            for each edge (u, v) ∈ In(v) do
                d(v, k) = min{d(v, k), d(u, k − 1) + ℓ(u, v)}

for each v ∈ V do
        dist(s, v) ← d(v, n − 1)
```

Running time: $O(mn)$ Space:

Bellman-Ford Algorithm

```
for each u ∈ V do
    d(u, 0) ← ∞
d(s, 0) ← 0

for k = 1 to n − 1 do
        for each v ∈ V do
            d(v, k) ← d(v, k − 1)
            for each edge (u, v) ∈ In(v) do
                d(v, k) = min{d(v, k), d(u, k − 1) + ℓ(u, v)}

for each v ∈ V do
        dist(s, v) ← d(v, n − 1)
```

Running time: $O(mn)$ Space: $O(n^2)$

Bellman-Ford Algorithm

```
for each u ∈ V do
    d(u, 0) ← ∞
d(s, 0) ← 0

for k = 1 to n − 1 do
        for each v ∈ V do
            d(v, k) ← d(v, k − 1)
            for each edge (u, v) ∈ In(v) do
                d(v, k) = min{d(v, k), d(u, k − 1) + ℓ(u, v)}

for each v ∈ V do
        dist(s, v) ← d(v, n − 1)
```

Running time: $O(mn)$ Space: $O(n^2)$

Space can be reduced to $O(n)$.

Bellman-Ford Algorithm

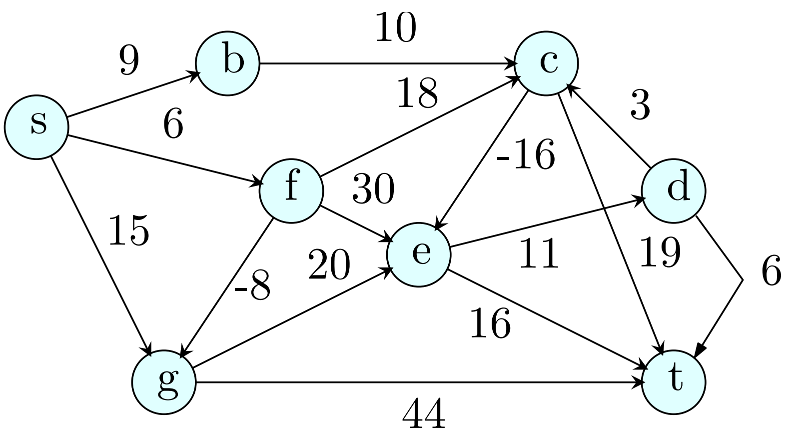
```
for each u ∈ V do
    d(u) ← ∞
d(s) ← 0

for k = 1 to n − 1 do
    for each v ∈ V do
        for each edge (u, v) ∈ In(v) do
            d(v) = min{d(v), d(u) + ℓ(u, v)}

for each v ∈ V do
    dist(s, v) ← d(v)
```

Running time: $O(mn)$ Space: $O(n)$

Negative Length Cycles

Definition

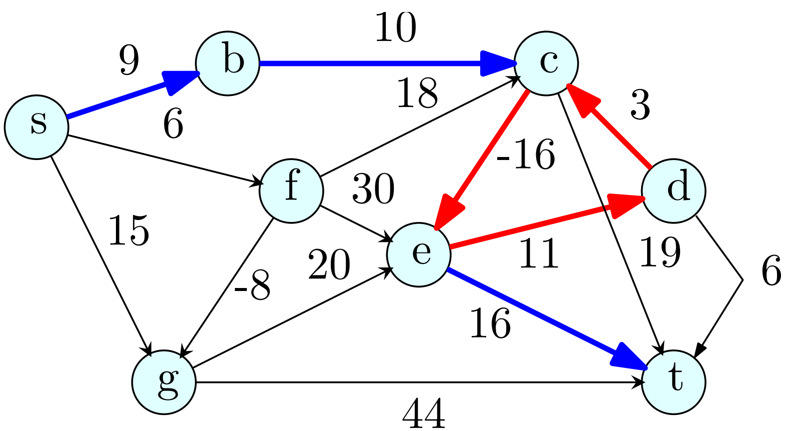
A cycle C is a negative length cycle if the sum of the edge lengths of C is negative.



Negative Length Cycles

Definition

A cycle C is a negative length cycle if the sum of the edge lengths of C is negative.



Shortest Paths and Negative Cycles

Given $G = (V, E)$ with edge lengths and s, t. Suppose

1. G has a negative length cycle C, and
2. s can reach C and C can reach t.

Question: What is the shortest **distance** from s to t?

Shortest Paths and Negative Cycles

Given $G = (V, E)$ with edge lengths and s, t. Suppose

1. G has a negative length cycle C, and
2. s can reach C and C can reach t.

Question: What is the shortest **distance** from s to t?

$-\infty$

Bellman-Ford: Negative Cycle Detection

Check if distances change in iteration n.

```
for each u ∈ V do
    d(u) ← ∞
d(s) ← 0

for k = 1 to n − 1 do
        for each v ∈ V do
            for each edge (u, v) ∈ In(v) do
                d(v) = min{d(v), d(u) + ℓ(u, v)}
(* One more iteration to check if distances change *)
for each v ∈ V do
    for each edge (u, v) ∈ In(v) do
        if (d(v) > d(u) + ℓ(u, v))
            Output ``Negative Cycle''

for each v ∈ V do
        dist(s, v) ← d(v)
```

Negative Cycle Detection

Negative Cycle Detection

Given directed graph G with arbitrary edge lengths, does it have a negative length cycle?

Negative Cycle Detection

Negative Cycle Detection

Given directed graph G with arbitrary edge lengths, does it have a negative length cycle?

1. Bellman-Ford checks whether there is a negative cycle C that is reachable from a specific vertex s. There may negative cycles not reachable from s.
2. Run Bellman-Ford $|V|$ times, once from each node u?

Negative Cycle Detection

1. Add a new node s' and connect it to all nodes of G with zero length edges. Bellman-Ford from s' will find a negative length cycle if there is one. Exercise: why does this work?
2. Negative cycle detection can be done with one Bellman-Ford invocation.