Shortest Path Algorithms

Lecture 5

February 3, 2015

Part I

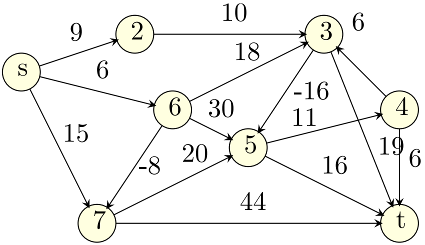
Shortest Paths with Negative Length Edges

Single-Source Shortest Paths with Negative Edge Lengths

Single-Source Shortest Path Problems

Input: A *directed* graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.

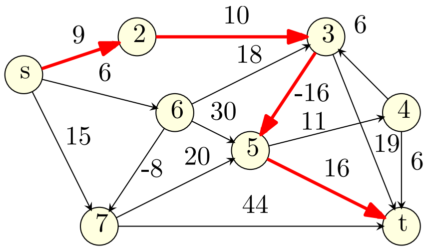


Single-Source Shortest Paths with Negative Edge Lengths

Single-Source Shortest Path Problems

Input: A *directed* graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

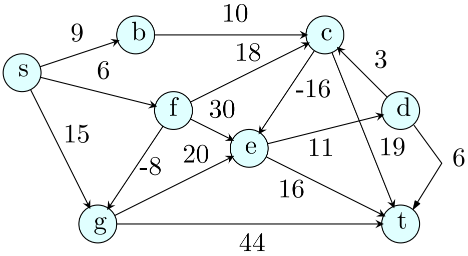
1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.



Negative Length Cycles

Definition

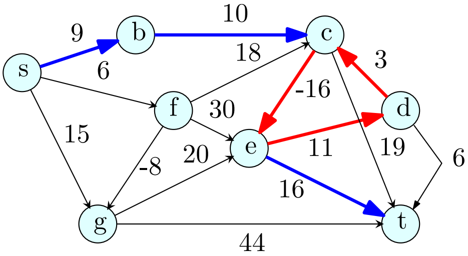
A cycle C is a **negative length cycle** if the sum of the edge lengths of C is negative.



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

1. 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.
2. **Question:** What is the shortest **distance** from s to t?
3. Possible answers: Define shortest distance to be:
   1. undefined, that is $-\infty$, OR
   2. the length of a shortest **simple** path from s to t.

Shortest Paths and Negative Cycles

1. 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.
2. **Question:** What is the shortest **distance** from s to t?
3. Possible answers: Define shortest distance to be:
   1. undefined, that is $-\infty$, OR
   2. the length of a shortest **simple** path from s to t.

Shortest Paths and Negative Cycles

1. 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.
2. **Question:** What is the shortest **distance** from s to t?
3. Possible answers: Define shortest distance to be:
   1. undefined, that is $-\infty$, OR
   2. the length of a shortest **simple** path from s to t.

Shortest Paths and Negative Cycles

1. 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.
2. **Question:** What is the shortest **distance** from s to t?
3. Possible answers: Define shortest distance to be:
   1. undefined, that is $-\infty$, OR
   2. the length of a shortest **simple** path from s to t.

Shortest Paths and Negative Cycles

1. 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.
2. **Question:** What is the shortest **distance** from s to t?
3. Possible answers: Define shortest distance to be:
    1. undefined, that is $-\infty$, OR
    2. the length of a shortest **simple** path from s to t.

Shortest Paths and Negative Cycles

1. 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.
2. **Question:** What is the shortest **distance** from s to t?
3. Possible answers: Define shortest distance to be:
   1. undefined, that is $-\infty$, OR
   2. the length of a shortest **simple** path from s to t.

Shortest Paths and Negative Cycles

1. 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.
2. **Question:** What is the shortest **distance** from s to t?
3. Possible answers: Define shortest distance to be:
   1. undefined, that is $-\infty$, OR
   2. the length of a shortest **simple** path from s to t.

Shortest Paths and Negative Cycles

1. 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.
2. **Question:** What is the shortest **distance** from s to t?
3. Possible answers: Define shortest distance to be:
   1. undefined, that is $-\infty$, OR
   2. the length of a shortest **simple** path from s to t.

Shortest Paths and Negative Cycles

Lemma

If there is an efficient algorithm to find a shortest simple $s \to t$ *path in a graph with negative edge lengths, then there is an efficient algorithm to find the* longest *simple* $s \to t$ *path in a graph with positive edge lengths.*

Finding the $s \to t$ longest path is difficult. **NP-Hard**!

Algorithmic Problems

Input: A directed graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

Questions:

1. Given nodes s, t, either find a negative length cycle C that s can reach or find a shortest path from s to t.
2. Given node s, either find a negative length cycle C that s can reach or find shortest path distances from s to all reachable nodes.
3. Check if G has a negative length cycle or not.

Algorithmic Problems

Input: A directed graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

Questions:

1. Given nodes s, t, either find a negative length cycle C that s can reach or find a shortest path from s to t.
2. Given node s, either find a negative length cycle C that s can reach or find shortest path distances from s to all reachable nodes.
3. Check if G has a negative length cycle or not.

Algorithmic Problems

Input: A directed graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

Questions:

1. Given nodes s, t, either find a negative length cycle C that s can reach or find a shortest path from s to t.
2. Given node s, either find a negative length cycle C that s can reach or find shortest path distances from s to all reachable nodes.
3. Check if G has a negative length cycle or not.

Algorithmic Problems

Input: A directed graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

Questions:

1. Given nodes s, t, either find a negative length cycle C that s can reach or find a shortest path from s to t.
2. Given node s, either find a negative length cycle C that s can reach or find shortest path distances from s to all reachable nodes.
3. Check if G has a negative length cycle or not.

Shortest Paths with Negative Edge Lengths

In Undirected Graphs

Note: With negative lengths, shortest path problems and negative cycle detection in undirected graphs cannot be reduced to directed graphs by bi-directing each undirected edge. Why?

Problem can be solved efficiently in undirected graphs but algorithms are different and more involved than those for directed graphs. Beyond the scope of this class. If interested, ask instructor for references.

Why Negative Lengths?

Several Applications

1. Shortest path problems useful in modeling many situations — in some negative lengths are natural
2. Negative length cycle can be used to find arbitrage opportunities in currency trading
3. Important sub-routine in algorithms for more general problem: minimum-cost flow

Why Negative Lengths?

Several Applications

1. Shortest path problems useful in modeling many situations — in some negative lengths are natural
2. Negative length cycle can be used to find arbitrage opportunities in currency trading
3. Important sub-routine in algorithms for more general problem: minimum-cost flow

Why Negative Lengths?

Several Applications

1. Shortest path problems useful in modeling many situations — in some negative lengths are natural
2. Negative length cycle can be used to find arbitrage opportunities in currency trading
3. Important sub-routine in algorithms for more general problem: minimum-cost flow

Why Negative Lengths?

Several Applications

1. Shortest path problems useful in modeling many situations — in some negative lengths are natural
2. Negative length cycle can be used to find arbitrage opportunities in currency trading
3. Important sub-routine in algorithms for more general problem: minimum-cost flow

Negative cycles

Application to Currency Trading

Currency Trading

Input: n currencies and for each ordered pair (a, b) the *exchange rate* for converting one unit of a into one unit of b.

Questions:

1. Is there an arbitrage opportunity?
2. Given currencies s, t what is the best way to convert s to t (perhaps via other intermediate currencies)?

Negative cycles

Application to Currency Trading

Currency Trading

Input: n currencies and for each ordered pair (a, b) the *exchange rate* for converting one unit of a into one unit of b.

Questions:

1. Is there an arbitrage opportunity?
2. Given currencies s, t what is the best way to convert s to t (perhaps via other intermediate currencies)?

Concrete example:

1. **1** Chinese Yuan = **0.1116** Euro
2. **1** Euro = **1.3617** US dollar
3. **1** US Dollar = **7.1** Chinese Yuan.

As such...

Thus, if exchanging **1 $** $\rightarrow$ Yuan $\rightarrow$ Euro $\rightarrow$ **$**, we get:
$\mathbf{0.1116} * \mathbf{1.3617} * \mathbf{7.1} = \mathbf{1.07896\$}$.

Negative cycles

Application to Currency Trading

Concrete example:

1. $\mathbf{1}$ Chinese Yuan $= \mathbf{0.1116}$ Euro
2. $\mathbf{1}$ Euro $= \mathbf{1.3617}$ US dollar
3. $\mathbf{1}$ US Dollar $= \mathbf{7.1}$ Chinese Yuan.

As such...

Thus, if exchanging $\mathbf{1}\ \$ \to$ Yuan $\to$ Euro $\to \mathbf{\$}$, we get:
$\mathbf{0.1116} * \mathbf{1.3617} * \mathbf{7.1} = \mathbf{1.07896\$}$.

Reducing Currency Trading to Shortest Paths

1. **Observation:** If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.
2. Create currency trading *directed* graph $G = (V, E)$:
   1. For each currency i there is a node $v_i \in V$
   2. $E = V \times V$: an edge for each pair of currencies
   3. edge length $\ell(v_i, v_j) = -\log(exch(i, j))$ can be negative
3. **Exercise:** Verify that
   1. There is an arbitrage opportunity if and only if G has a negative length cycle.
   2. The best way to convert currency i to currency j is via a shortest path in G from i to j. If d is the distance from i to j then one unit of i can be converted into 2^d units of j.

Reducing Currency Trading to Shortest Paths

1. **Observation:** If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.
2. Create currency trading *directed* graph $G = (V, E)$:
    1. For each currency i there is a node $v_i \in V$
    2. $E = V \times V$: an edge for each pair of currencies
    3. edge length $\ell(v_i, v_j) = -\log(exch(i, j))$ can be negative
3. **Exercise:** Verify that
    1. There is an arbitrage opportunity if and only if G has a negative length cycle.
    2. The best way to convert currency i to currency j is via a shortest path in G from i to j. If d is the distance from i to j then one unit of i can be converted into 2^d units of j.

Reducing Currency Trading to Shortest Paths

1. **Observation:** If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.
2. Create currency trading *directed* graph $G = (V, E)$:
    1. For each currency i there is a node $v_i \in V$
    2. $E = V \times V$: an edge for each pair of currencies
    3. edge length $\ell(v_i, v_j) = -\log(exch(i, j))$ can be negative
3. **Exercise:** Verify that
    1. There is an arbitrage opportunity if and only if G has a negative length cycle.
    2. The best way to convert currency i to currency j is via a shortest path in G from i to j. If d is the distance from i to j then one unit of i can be converted into 2^d units of j.

Reducing Currency Trading to Shortest Paths

1. **Observation:** If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.
2. Create currency trading *directed* graph $G = (V, E)$:
    1. For each currency i there is a node $v_i \in V$
    2. $E = V \times V$: an edge for each pair of currencies
    3. edge length $\ell(v_i, v_j) = -\log(exch(i, j))$ can be negative
3. **Exercise:** Verify that
    1. There is an arbitrage opportunity if and only if G has a negative length cycle.
    2. The best way to convert currency i to currency j is via a shortest path in G from i to j. If d is the distance from i to j then one unit of i can be converted into 2^d units of j.

Reducing Currency Trading to Shortest Paths

1. **Observation:** If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.
2. Create currency trading *directed* graph $G = (V, E)$:
   1. For each currency i there is a node $v_i \in V$
   2. $E = V \times V$: an edge for each pair of currencies
   3. edge length $\ell(v_i, v_j) = -\log(exch(i, j))$ can be negative
3. **Exercise:** Verify that
   1. There is an arbitrage opportunity if and only if G has a negative length cycle.
   2. The best way to convert currency i to currency j is via a shortest path in G from i to j. If d is the distance from i to j then one unit of i can be converted into 2^d units of j.

Reducing Currency Trading to Shortest Paths

1. **Observation:** If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.
2. Create currency trading *directed* graph $G = (V, E)$:
   1. For each currency i there is a node $v_i \in V$
   2. $E = V \times V$: an edge for each pair of currencies
   3. edge length $\ell(v_i, v_j) = -\log(exch(i, j))$ can be negative
3. **Exercise:** Verify that
   1. There is an arbitrage opportunity if and only if G has a negative length cycle.
   2. The best way to convert currency i to currency j is via a shortest path in G from i to j. If d is the distance from i to j then one unit of i can be converted into 2^d units of j.

Reducing Currency Trading to Shortest Paths

1. **Observation:** If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.
2. Create currency trading *directed* graph $G = (V, E)$:
   1. For each currency i there is a node $v_i \in V$
   2. $E = V \times V$: an edge for each pair of currencies
   3. edge length $\ell(v_i, v_j) = -\log(exch(i, j))$ can be negative
3. **Exercise:** Verify that
   1. There is an arbitrage opportunity if and only if G has a negative length cycle.
   2. The best way to convert currency i to currency j is via a shortest path in G from i to j. If d is the distance from i to j then one unit of i can be converted into 2^d units of j.

Reducing Currency Trading to Shortest Paths

1. **Observation:** If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.
2. Create currency trading *directed* graph $G = (V, E)$:
   1. For each currency i there is a node $v_i \in V$
   2. $E = V \times V$: an edge for each pair of currencies
   3. edge length $\ell(v_i, v_j) = -\log(exch(i, j))$ can be negative
3. **Exercise:** Verify that
   1. There is an arbitrage opportunity if and only if G has a negative length cycle.
   2. The best way to convert currency i to currency j is via a shortest path in G from i to j. If d is the distance from i to j then one unit of i can be converted into 2^d units of j.

Reducing Currency Trading to Shortest Paths

1. **Observation:** If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.
2. Create currency trading *directed* graph $G = (V, E)$:
   1. For each currency i there is a node $v_i \in V$
   2. $E = V \times V$: an edge for each pair of currencies
   3. edge length $\ell(v_i, v_j) = -\log(exch(i, j))$ can be negative
3. **Exercise:** Verify that
   1. There is an arbitrage opportunity if and only if G has a negative length cycle.
   2. The best way to convert currency i to currency j is via a shortest path in G from i to j. If d is the distance from i to j then one unit of i can be converted into 2^d units of j.

Reducing Currency Trading to Shortest Paths

1. **Observation:** If we convert currency i to j via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of i yields $exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)$ units of j.
2. Create currency trading *directed* graph $G = (V, E)$:
    1. For each currency i there is a node $v_i \in V$
    2. $E = V \times V$: an edge for each pair of currencies
    3. edge length $\ell(v_i, v_j) = -\log(exch(i, j))$ can be negative
3. **Exercise:** Verify that
    1. There is an arbitrage opportunity if and only if G has a negative length cycle.
    2. The best way to convert currency i to currency j is via a shortest path in G from i to j. If d is the distance from i to j then one unit of i can be converted into 2^d units of j.

Reducing Currency Trading to Shortest Paths

Math recall - relevant information

1. $\log(\alpha_1 * \alpha_2 * \cdots * \alpha_k) = \log \alpha_1 + \log \alpha_2 + \cdots + \log \alpha_k$.
2. $\log x > 0$ if and only if $x > 1$.

Shortest Paths with Negative Edge Lengths

Problems

Algorithmic Problems

Input: A directed graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

Questions:

1. Given nodes s, t, either find a negative length cycle C that s can reach or find a shortest path from s to t.
2. Given node s, either find a negative length cycle C that s can reach or find shortest path distances from s to all reachable nodes.
3. Check if G has a negative length cycle or not.

Algorithmic Problems

Input: A directed graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

Questions:

1. Given nodes s, t, either find a negative length cycle C that s can reach or find a shortest path from s to t.
2. Given node s, either find a negative length cycle C that s can reach or find shortest path distances from s to all reachable nodes.
3. Check if G has a negative length cycle or not.

Algorithmic Problems

Input: A directed graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

Questions:

1. Given nodes s, t, either find a negative length cycle C that s can reach or find a shortest path from s to t.
2. Given node s, either find a negative length cycle C that s can reach or find shortest path distances from s to all reachable nodes.
3. Check if G has a negative length cycle or not.

Algorithmic Problems

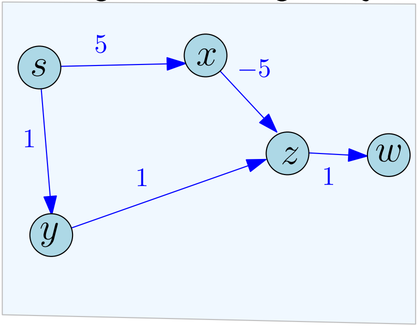
Input: A directed graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

Questions:

1. Given nodes s, t, either find a negative length cycle C that s can reach or find a shortest path from s to t.
2. Given node s, either find a negative length cycle C that s can reach or find shortest path distances from s to all reachable nodes.
3. Check if G has a negative length cycle or not.

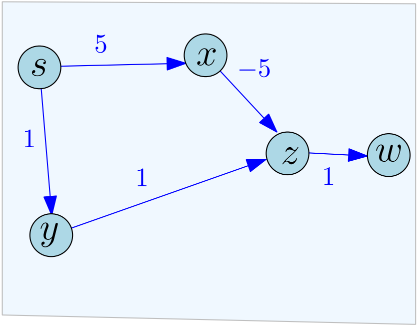
Dijkstra's Algorithm and Negative Lengths

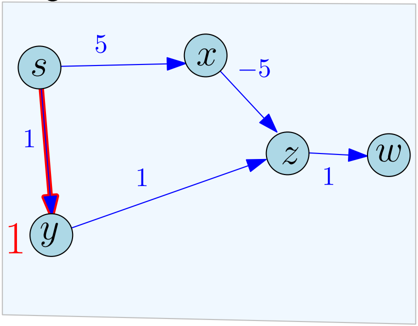
With negative cost edges, Dijkstra's algorithm fails



Dijkstra's Algorithm and Negative Lengths

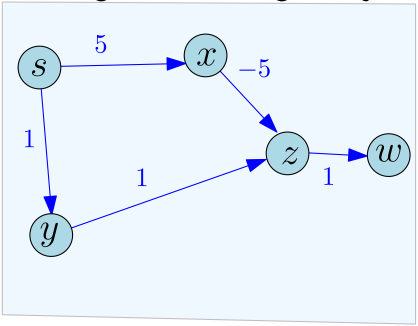
With negative cost edges, Dijkstra's algorithm fails

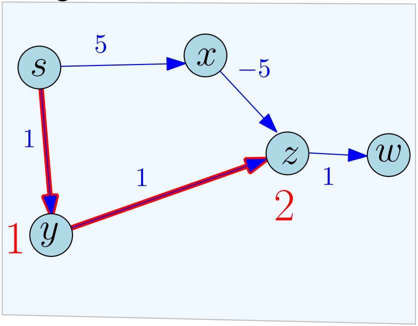




Dijkstra's Algorithm and Negative Lengths

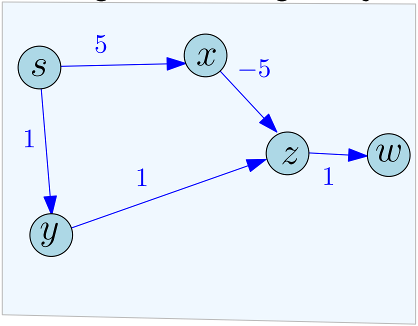
With negative cost edges, Dijkstra's algorithm fails

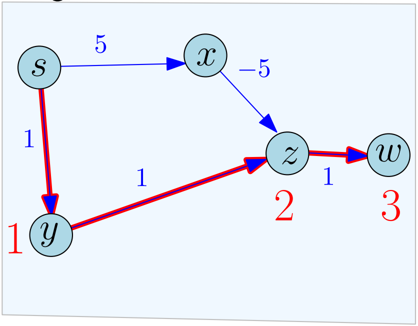




Dijkstra's Algorithm and Negative Lengths

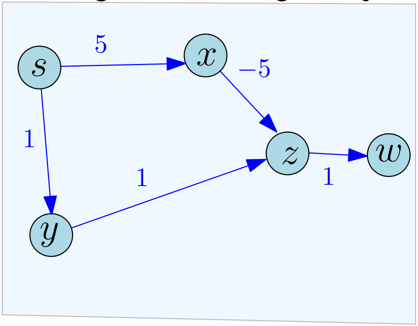
With negative cost edges, Dijkstra's algorithm fails

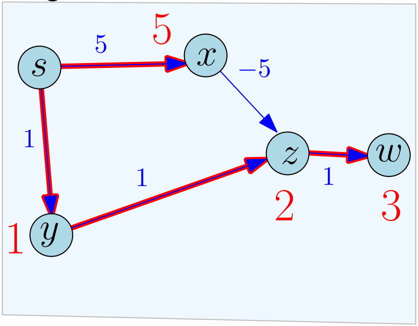




Dijkstra's Algorithm and Negative Lengths

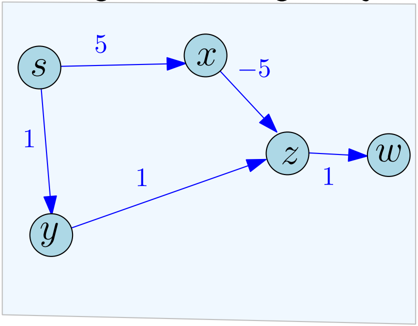
With negative cost edges, Dijkstra's algorithm fails

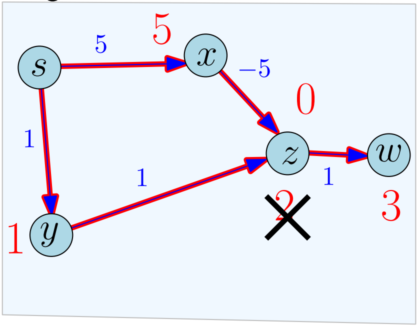




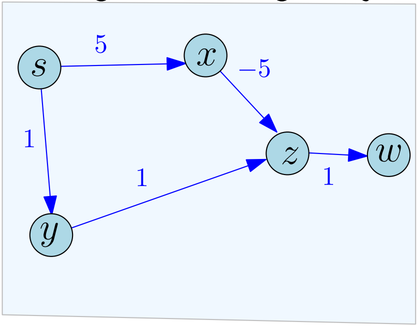
Dijkstra's Algorithm and Negative Lengths

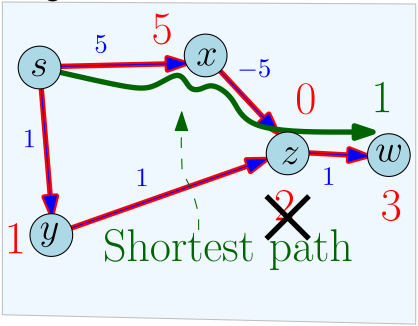
With negative cost edges, Dijkstra's algorithm fails





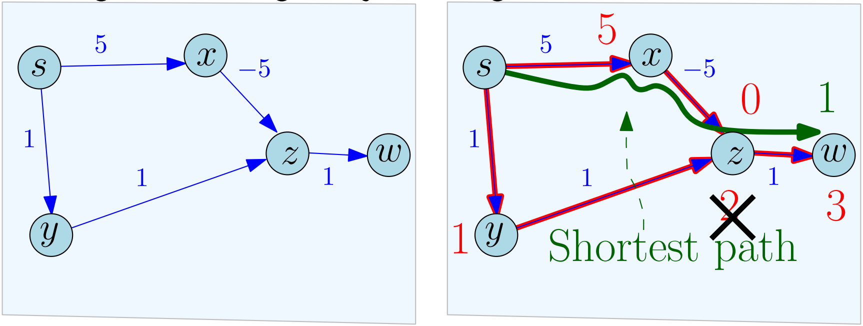
With negative cost edges, Dijkstra's algorithm fails





Dijkstra's Algorithm and Negative Lengths

With negative cost edges, Dijkstra's algorithm fails



False assumption: Dijkstra's algorithm is based on the assumption that if $s = v_0 \to v_1 \to v_2 \dots \to v_k$ 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.

Shortest Paths with Negative Lengths

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 $1 \le i < k$:

1. $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ *is a shortest path from* s *to* v_i
2. *False:* $dist(s, v_i) \le dist(s, v_k)$ *for* $1 \le i < k$. *Holds true only for non-negative edge lengths.*

Cannot explore nodes in increasing order of distance! We need a more basic strategy.

Shortest Paths with Negative Lengths

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
2. *False:* $dist(s, v_i) \leq dist(s, v_k)$ *for* $1 \leq i < k$. *Holds true only for non-negative edge lengths.*

Cannot explore nodes in increasing order of distance! We need a more basic strategy.

Shortest Paths with Negative Lengths

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
2. *False:* $dist(s, v_i) \leq dist(s, v_k)$ *for* $1 \leq i < k$. *Holds true only for non-negative edge lengths.*

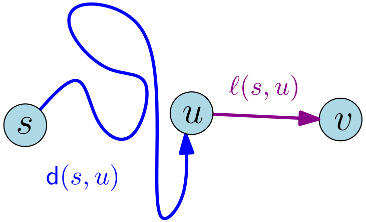
Cannot explore nodes in increasing order of distance! We need a more basic strategy.

A Generic Shortest Path Algorithm

1. Start with distance estimate for each node $d(s, u)$ set to ∞
2. Maintain the invariant that there is an $s \to u$ path of length $d(s, u)$. Hence $d(s, u) \geq dist(s, u)$.
3. Iteratively refine $d(s, \cdot)$ values until they reach the correct value $dist(s, \cdot)$ values at termination

Must hold that...

$d(s, v) \leq d(s, u) + \ell(u, v)$



A Generic Shortest Path Algorithm

Question: How do we make progress?

Definition

Given distance estimates $d(s, u)$ for each $u \in V$, an edge $e = (u, v)$ is **tense** if $d(s, v) > d(s, u) + \ell(u, v)$.

Relax$(e = (u, v))$
  if $(d(s, v) > d(s, u) + \ell(u, v))$ **then**
    $d(s, v) \Leftarrow d(s, u) + \ell(u, v)$

A Generic Shortest Path Algorithm

Question: How do we make progress?

Definition

Given distance estimates $d(s, u)$ for each $u \in V$, an edge $e = (u, v)$ is **tense** if $d(s, v) > d(s, u) + \ell(u, v)$.

Relax$(e = (u, v))$
    if $(d(s, v) > d(s, u) + \ell(u, v))$ **then**
        $d(s, v) \Leftarrow d(s, u) + \ell(u, v)$

A Generic Shortest Path Algorithm

Question: How do we make progress?

Definition

Given distance estimates $d(s, u)$ for each $u \in V$, an edge $e = (u, v)$ is **tense** if $d(s, v) > d(s, u) + \ell(u, v)$.

Relax($e = (u, v)$)
    if ($d(s, v) > d(s, u) + \ell(u, v)$) **then**
        $d(s, v) \Leftarrow d(s, u) + \ell(u, v)$

A Generic Shortest Path Algorithm

Invariant

If a vertex u has value $d(s, u)$ associated with it, then there is a $s \rightsquigarrow u$ walk of length $d(s, u)$.

Proposition

Relax *maintains the invariant on* $d(s, u)$ *values.*

Proof.

Indeed, if **Relax**((u, v)) changed the value of $d(s, v)$, then there is a walk to u of length $d(s, u)$ (by invariant), and there is a walk of length $d(s, u) + \ell(u, v)$ to v through u, which is the new value of $d(s, v)$. □

A Generic Shortest Path Algorithm

```
d(s, s) = 0
for each node u ≠ s do
    d(s, u) = ∞

while there is a tense edge do
    Pick a tense edge e
    Relax(e)

Output d(s, u) values
```

Technical assumption: If $e = (u, v)$ is an edge and $d(s, u) = d(s, v) = \infty$ then edge is not tense.

Properties of the generic algorithm

Proposition

If u *is not reachable from* s *then* $d(s, u)$ *remains at* ∞ *throughout the algorithm.*

Properties of the generic algorithm

Proposition

If a negative length cycle C is reachable by s then there is always a tense edge and hence the algorithm never terminates.

Proof

Let $C = v_0, v_1, \ldots, v_k$ be a negative length cycle. Suppose algorithm terminates. Since no edge of C was tense, for $i = 1, 2, \ldots, k$ we have $d(s, v_i) \leq d(s, v_{i-1}) + \ell(v_{i-1}, v_i)$ and $d(s, v_0) \leq d(s, v_k) + \ell(v_k, v_0)$. Adding up all the inequalities we obtain that length of C is non-negative!

Proof in more detail...

$$
\begin{aligned}
d(s, v_1) &\leq d(s, v_0) + \ell(v_0, v_1) \\
d(s, v_2) &\leq d(s, v_1) + \ell(v_1, v_2) \\
&\dots \\
d(s, v_i) &\leq d(s, v_{i-1}) + \ell(v_{i-1}, v_i) \\
&\dots \\
d(s, v_k) &\leq d(s, v_{k-1}) + \ell(v_{k-1}, v_k) \\
d(s, v_0) &\leq d(s, v_k) + \ell(v_k, v_k)
\end{aligned}
$$

$$
\sum_{i=0}^{k} d(s, v_i) \leq \sum_{i=0}^{k} d(s, v_i) + \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0)
$$

Proof in more detail...

$$
\begin{aligned}
d(s, v_1) &\leq d(s, v_0) + \ell(v_0, v_1) \\
d(s, v_2) &\leq d(s, v_1) + \ell(v_1, v_2) \\
&\cdots \\
d(s, v_i) &\leq d(s, v_{i-1}) + \ell(v_{i-1}, v_i) \\
&\cdots \\
d(s, v_k) &\leq d(s, v_{k-1}) + \ell(v_{k-1}, v_k) \\
d(s, v_0) &\leq d(s, v_k) + \ell(v_k, v_k)
\end{aligned}
$$

$$
\sum_{i=0}^{k} d(s, v_i) \leq \sum_{i=0}^{k} d(s, v_i) + \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0)
$$

Proof in more detail...

$$\sum_{i=0}^{k} d(s, v_i) \leq \sum_{i=0}^{k} d(s, v_i) + \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0)$$

Proof in more detail...

$$\sum_{i=0}^{k} d(s, v_i) \leq \sum_{i=0}^{k} d(s, v_i) + \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0)$$

$$0 \leq \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0).$$

Proof in more detail...

$$\sum_{i=0}^{k} d(s, v_i) \leq \sum_{i=0}^{k} d(s, v_i) + \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0)$$

$$0 \leq \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0) = \mathrm{len}(C).$$

Proof in more detail...

$$\sum_{i=0}^{k} d(s, v_i) \le \sum_{i=0}^{k} d(s, v_i) + \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0)$$

$$0 \le \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0) = \mathrm{len}(C).$$

C is a not a negative cycle. Contradiction.

Properties of the generic algorithm

Corollary

If the algorithm terminates then there is no negative length cycle C that is reachable from s.

Properties of the generic algorithm

Lemma

If the algorithm terminates then $d(s,u) = dist(s,u)$ *for each node* u *(and* s *cannot reach a negative cycle).*

Proof of lemma; see future slides.

Properties of the generic algorithm

If estimate distance from source too large, then ∃ tense edge...

Lemma

If ∃ *walk* $\pi \equiv s = v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k = u$ *such that*

$$\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_j) < d(s, u)$$

Then, there exists a tense edge in G*.*

Proof.

Assume π: shortest in number of edges (with property).

$\implies \ell(v_1 \rightarrow \cdots v_{k-1}) \geq d(s, v_{k-1})$.

$\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k)$

$\leq \ell(v_1 \rightarrow \cdots v_{k-1}) + \ell(v_{k-1}, v_k)$

$= \ell(\pi) < d(s, v_k)$.

$\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k) < d(s, v_k)$

$\implies$ edge (v_{k-1}, v_k) is tense. □

Properties of the generic algorithm

If estimate distance from source too large, then ∃ tense edge...

Lemma

If ∃ *walk* $\pi \equiv s = v_1 \to v_2 \to \cdots \to v_k = u$ *such that*

$$\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_j) < d(s, u)$$

Then, there exists a tense edge in G.

Proof.

Assume π: shortest in number of edges (with property).

$\implies \ell(v_1 \to \cdots v_{k-1}) \geq d(s, v_{k-1})$.

$\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k)$

$\leq \ell(v_1 \to \cdots v_{k-1}) + \ell(v_{k-1}, v_k)$

$= \ell(\pi) < d(s, v_k)$.

$\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k) < d(s, v_k)$

$\implies$ edge (v_{k-1}, v_k) is tense. □

Properties of the generic algorithm

If estimate distance from source too large, then ∃ tense edge...

Lemma

If ∃ walk $\pi \equiv s = v_1 \to v_2 \to \cdots \to v_k = u$ *such that*

$$\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_j) < d(s, u)$$

Then, there exists a tense edge in G*.*

Proof.

Assume π: shortest in number of edges (with property).

$\implies \ell(v_1 \to \cdots v_{k-1}) \geq d(s, v_{k-1})$.

$\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k)$

$\leq \ell(v_1 \to \cdots v_{k-1}) + \ell(v_{k-1}, v_k)$

$= \ell(\pi) < d(s, v_k)$.

$\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k) < d(s, v_k)$

$\implies$ edge (v_{k-1}, v_k) is tense. □

Properties of the generic algorithm

If estimate distance from source too large, then ∃ tense edge...

Lemma

If ∃ *walk* $\pi \equiv s = v_1 \to v_2 \to \cdots \to v_k = u$ *such that*

$$\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_j) < d(s, u)$$

Then, there exists a tense edge in G.

Proof.

Assume π: shortest in number of edges (with property).

$\implies \ell(v_1 \to \cdots v_{k-1}) \geq d(s, v_{k-1})$.

$\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k)$

$\leq \ell(v_1 \to \cdots v_{k-1}) + \ell(v_{k-1}, v_k)$

$= \ell(\pi) < d(s, v_k)$.

$\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k) < d(s, v_k)$

$\implies$ edge (v_{k-1}, v_k) is tense. □

Properties of the generic algorithm

If estimate distance from source too large, then ∃ tense edge...

Lemma

If ∃ walk $\pi \equiv s = v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k = u$ *such that*

$$\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_j) < d(s, u)$$

Then, there exists a tense edge in G.

Proof.

Assume π: shortest in number of edges (with property).

$\implies \ell(v_1 \rightarrow \cdots v_{k-1}) \geq d(s, v_{k-1})$.

$\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k)$

$\leq \ell(v_1 \rightarrow \cdots v_{k-1}) + \ell(v_{k-1}, v_k)$

$= \ell(\pi) < d(s, v_k)$.

$\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k) < d(s, v_k)$

$\implies$ edge (v_{k-1}, v_k) is tense. □

Properties of the generic algorithm

If estimate distance from source too large, then ∃ tense edge...

Lemma

If $\exists$ *walk* $\pi \equiv s = v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k = u$ *such that*

$$\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_j) < d(s, u)$$

Then, there exists a tense edge in G*.*

Proof.

Assume π: shortest in number of edges (with property).

$\Longrightarrow$ $\ell(v_1 \rightarrow \cdots v_{k-1}) \geq d(s, v_{k-1})$.

$\Longrightarrow$ $d(s, v_{k-1}) + \ell(v_{k-1}, v_k)$

$\leq \ell(v_1 \rightarrow \cdots v_{k-1}) + \ell(v_{k-1}, v_k)$

$= \ell(\pi) < d(s, v_k)$.

$\Longrightarrow$ $d(s, v_{k-1}) + \ell(v_{k-1}, v_k) < d(s, v_k)$

$\Longrightarrow$ edge (v_{k-1}, v_k) is tense. □

Properties of the generic algorithm

If estimate distance from source too large, then ∃ tense edge...

Lemma

If $\exists$ *walk* $\pi \equiv s = v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k = u$ *such that*

$$\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_j) < d(s, u)$$

Then, there exists a tense edge in G.

Proof.

Assume π: shortest in number of edges (with property).

$\cdots$

$\implies$ edge (v_{k-1}, v_k) is tense. □

$\implies$ If for any vertex u: $d(s, u) > \mathrm{dist}(s, u)$ then the algorithm will continue working!

Generic Algorithm: Ordering Relax operations

```
d(s,s) = 0
for each node u ≠ s do
    d(s,u) = ∞

While there is a tense edge do
    Pick a tense edge e
    Relax(e)

Output d(s,u) values for u ∈ V(G)
```

Question: How do we pick edges to relax?

Observation: Suppose $s \rightarrow v_1 \rightarrow \ldots \rightarrow v_k$ is a shortest path.

If **Relax**(s, v_1), **Relax**(v_1, v_2), ..., **Relax**(v_{k-1}, v_k) are done in *order* then $d(s, v_k) = dist(s, v_k)$!

Generic Algorithm: Ordering Relax operations

```
d(s,s) = 0
for each node u ≠ s do
    d(s,u) = ∞

While there is a tense edge do
    Pick a tense edge e
    Relax(e)

Output d(s,u) values for u ∈ V(G)
```

Question: How do we pick edges to relax?

Observation: Suppose $s \rightarrow v_1 \rightarrow \ldots \rightarrow v_k$ is a shortest path.

If **Relax**(s, v_1), **Relax**(v_1, v_2), $\ldots$, **Relax**(v_{k-1}, v_k) are done in *order* then $d(s, v_k) = dist(s, v_k)$!

Ordering Relax operations

1. **Observation:** Suppose $s \to v_1 \to \ldots \to v_k$ is a shortest path.

   If **Relax**(s, v_1), **Relax**(v_1, v_2), $\ldots$, **Relax**(v_{k-1}, v_k) are done in *order* then $d(s, v_k) = dist(s, v_k)$! (Why?)

2. We don't know the shortest paths so how do we know the order to do the Relax operations?

Ordering Relax operations

1. **Observation:** Suppose $s \to v_1 \to \ldots \to v_k$ is a shortest path.

   If **Relax**(s, v_1), **Relax**(v_1, v_2), $\ldots$, **Relax**(v_{k-1}, v_k) are done in *order* then $d(s, v_k) = dist(s, v_k)$! (Why?)
2. We don't know the shortest paths so how do we know the order to do the Relax operations?

Ordering Relax operations

1. We don't know the shortest paths so how do we know the order to do the Relax operations?
2. We don't!
    1. Relax *all* edges (even those not tense) in some arbitrary order
    2. Iterate $|V| - 1$ times
    3. First iteration will do **Relax**(s, v_1) (and other edges), second round **Relax**(v_1, v_2) and in iteration k we do **Relax**(v_{k-1}, v_k).

Ordering Relax operations

1. We don't know the shortest paths so how do we know the order to do the Relax operations?
2. We don't!
    1. Relax *all* edges (even those not tense) in some arbitrary order
    2. Iterate $|V| - 1$ times
    3. First iteration will do **Relax**(s, v_1) (and other edges), second round **Relax**(v_1, v_2) and in iteration k we do **Relax**(v_{k-1}, v_k).

The Bellman-Ford (**BellmanFord**) Algorithm

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

    for i = 1 to |V| − 1 do
        for each edge e = (u, v) do
            Relax(e)

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

BellmanFord Algorithm: Scanning Edges

One possible way to scan edges in each iteration.

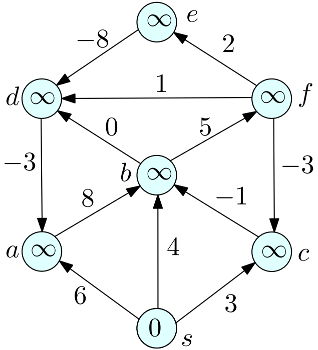
```
Q is an empty queue
for each u ∈ V do
    d(s, u) = ∞
    enq(Q, u)
d(s, s) = 0

for i = 1 to |V| − 1 do
    for j = 1 to |V| do
        u = deq(Q)
        for each edge e in Adj(u) do
            Relax(e)
        enq(Q, u)

for each u ∈ V do
    dist(s, u) = d(s, u)
```

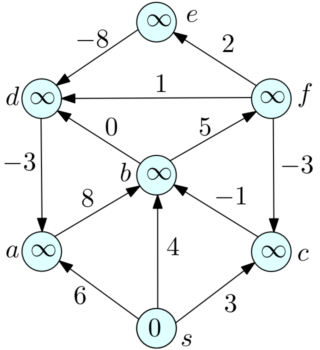
Example

Step 0



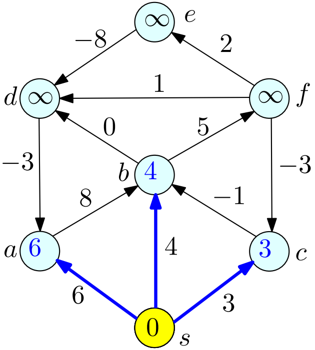
Example

Step 0



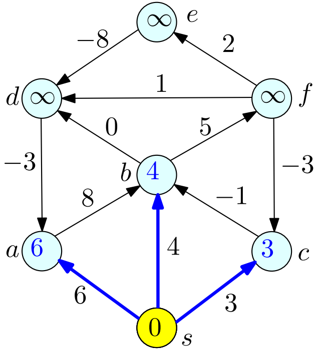
⟹

Step 1

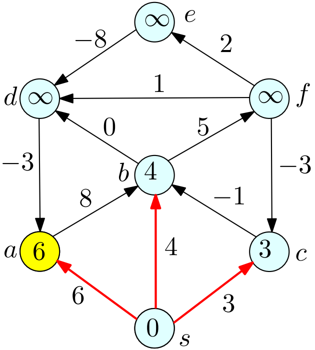


Example

Step 1

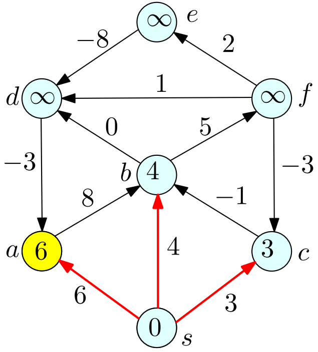


Step 2



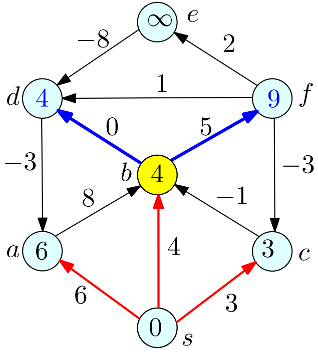
Example

Step 2

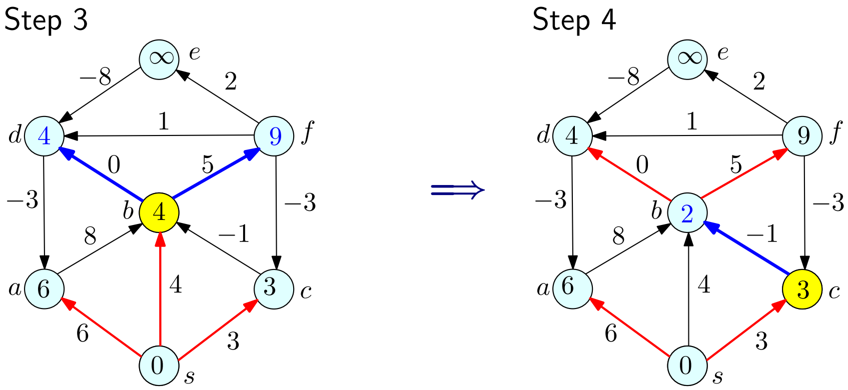


⟹

Step 3



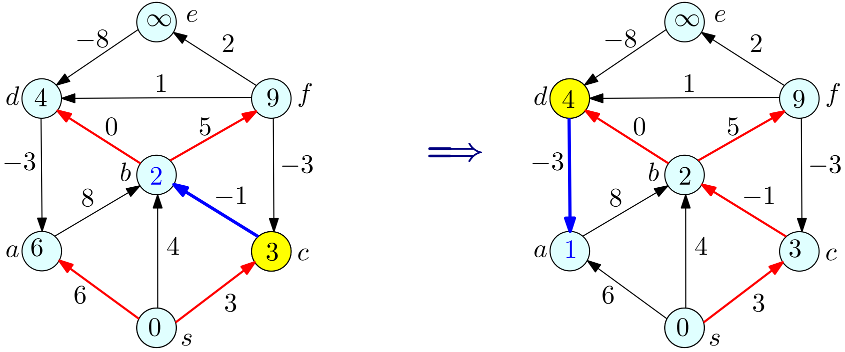
Example


Step 3
∞
e
−8
2
d
4
1
9
f
0
5
−3
b
4
−3
8
−1
a
6
4
3
c
6
0
3
s
Step 4
∞
e
−8
2
d
4
1
9
f
0
5
−3
b
2
−3
8
−1
a
6
4
3
c
6
0
3
s

Example

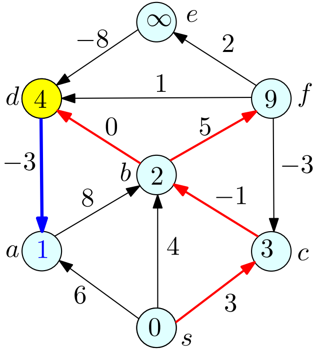
Step 4

Step 5



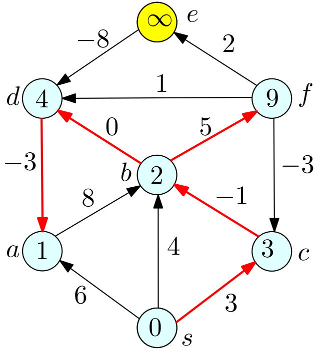
Example

Step 5



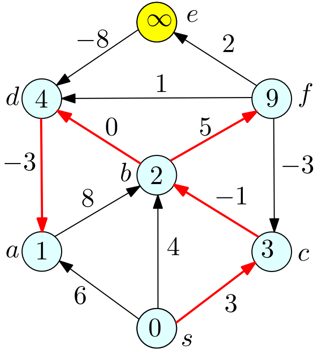
⟹

Step 6



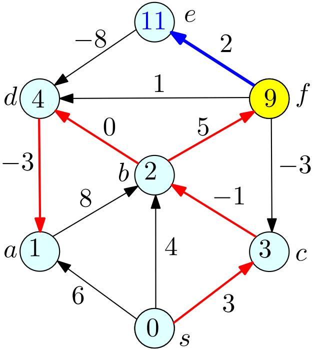
Example

Step 6



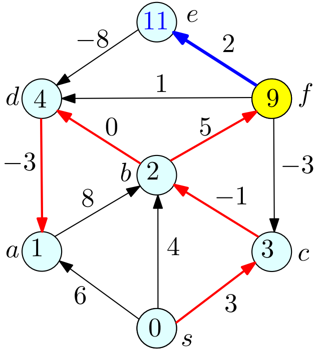
⟹

Step 7



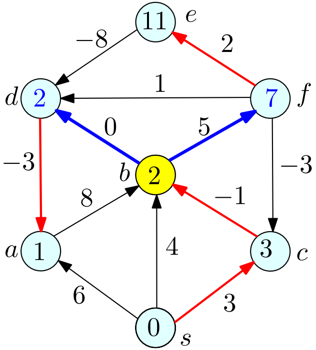
Example

Step 7



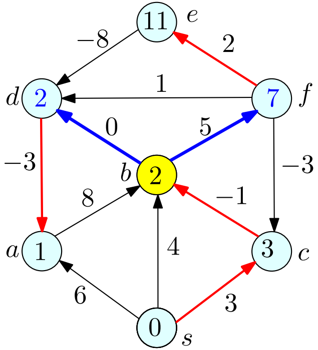
$\Longrightarrow$

Step 8

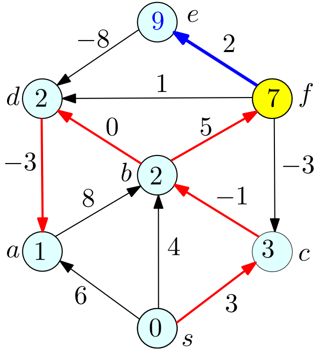


Example

Step 8

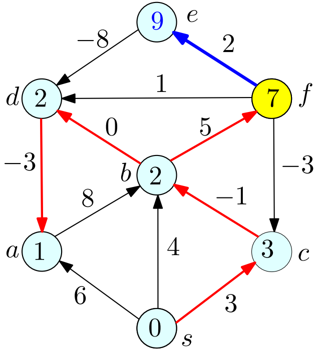


Step 9

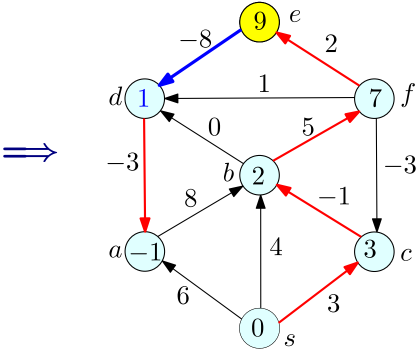


Example

Step 9

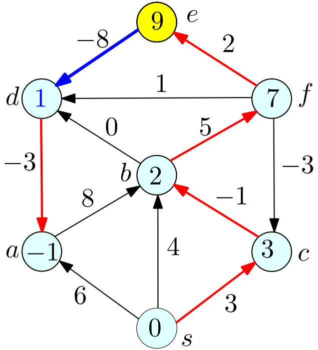


Step 10



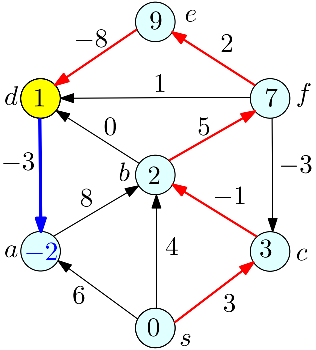
Example

Step 10



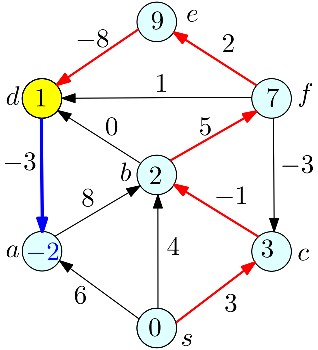
⟹

Step 11



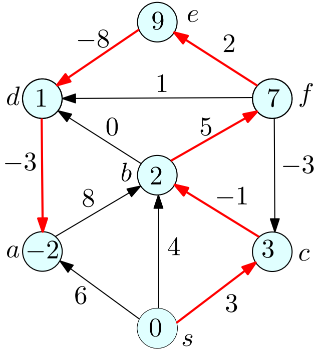
Example

Step 11



$\Longrightarrow$

Step 12



We are done! No edge is tense.

Example

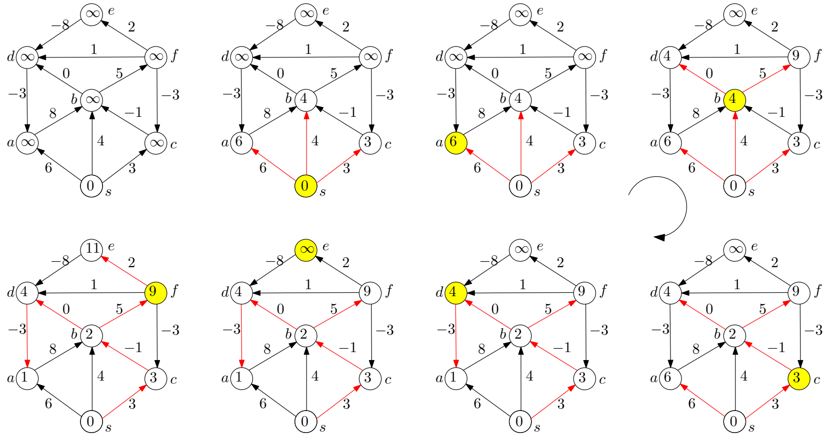


Figure: One iteration of **BellmanFord** that Relaxes all edges by processing nodes in the order ***s*, *a*, *b*, *c*, *d*, *e*, *f***. Red edges indicate the **prev** pointers (in reverse)

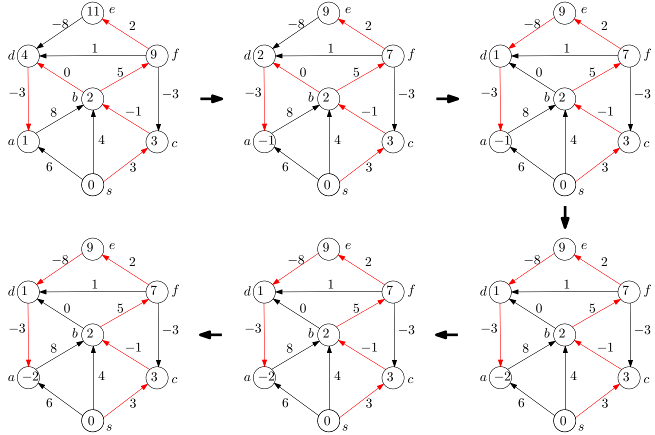


Figure: **6** iterations of **BellmanFord** starting with the first one from previous slide. No changes in **5**th iteration and **6**th iteration.

Correctness of the **BellmanFord** Algorithm

Lemma

G: a directed graph with arbitrary edge lengths, v: a node in V s.t. there is a shortest path from s to v with i edges. Then, after i iterations of the loop in **BellmanFord**, $d(s, v) = dist(s, v)$

Proof.

By induction on i.

1. Base case: $i = 0$. $d(s, s) = 0$ and $d(s, s) = dist(s, s)$.
2. Induction Step: Let $s \to v_1 \ldots \to v_{i-1} \to v$ be a shortest path from s to v of i hops.
    1. v_{i-1} has a shortest path from s of $i - 1$ hops or less. (Why?). By induction, $d(s, v_{i-1}) = dist(s, v_{i-1})$ after $i - 1$ iterations.
    2. In iteration i, **Relax**(v_{i-1}, v_i) sets $d(s, v_i) = \mathrm{dist}(s, v_i)$.
    3. Note: Relax does not change $d(s, u)$ once $d(s, u) = \mathrm{dist}(s, u)$.

Correctness of **BellmanFord** Algorithm

Corollary

After $|V| - 1$ iterations of **BellmanFord**, $d(s, u) = dist(s, u)$ *for any node* u *that has a shortest path from* s.

Note: If there is a negative cycle C such that s can reach C then we do not know whether $d(s, u) = dist(s, u)$ or not even if $dist(s, u)$ is well-defined.

Question: How do we know whether there is a negative cycle C reachable from s?

Correctness of **BellmanFord** Algorithm

Corollary

After $|V| - 1$ iterations of **BellmanFord**, $d(s, u) = dist(s, u)$ *for any node u that has a shortest path from s.*

Note: If there is a negative cycle C such that s can reach C then we do not know whether $d(s, u) = dist(s, u)$ or not even if $dist(s, u)$ is well-defined.

Question: How do we know whether there is a negative cycle C reachable from s?

Correctness of **BellmanFord** Algorithm

Corollary

After $|V| - 1$ iterations of **BellmanFord**, $d(s, u) = dist(s, u)$ *for any node* u *that has a shortest path from* s.

Note: If there is a negative cycle C such that s can reach C then we do not know whether $d(s, u) = dist(s, u)$ or not even if $dist(s, u)$ is well-defined.

Question: How do we know whether there is a negative cycle C reachable from s?

BellmanFord to detect Negative Cycles

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

for i = 1 to |V| − 1 do
        for each edge e = (u, v) do
            Relax(e)

for each edge e = (u, v) do
        if e = (u, v) is tense then
            Stop and output that s can reach
                    a negative length cycle

Output for each u ∈ V:   d(s, u)
```

Correctness

Lemma

G has a negative cycle reachable from s if and only if there is a tense edge e after $|V| - 1$ iterations of **BellmanFord**.

Proof Sketch.

G has no negative length cycle reachable from s implies that all nodes u have a shortest path from s. Therefore $d(s, u) = dist(s, u)$ after the $|V| - 1$ iterations. Therefore, there cannot be any tense edges left.

If there is a negative cycle C then there is a tense edge after $|V| - 1$ (in fact any number of) iterations. See lemma about properties of the generic shortest path algorithm. □

Correctness

Lemma

G has a negative cycle reachable from s if and only if there is a tense edge e after $|V| - 1$ iterations of **BellmanFord**.

Proof Sketch.

G has no negative length cycle reachable from s implies that all nodes u have a shortest path from s. Therefore $d(s, u) = dist(s, u)$ after the $|V| - 1$ iterations. Therefore, there cannot be any tense edges left.

If there is a negative cycle C then there is a tense edge after $|V| - 1$ (in fact any number of) iterations. See lemma about properties of the generic shortest path algorithm. □

Part II

Negative cycle detection

Finding the Paths and a Shortest Path Tree

```
BellmanFord:
    for each u ∈ V do
        d(s, u) = ∞
        prev(u) = null
    d(s, s) = 0
    for i = 1 to |V| − 1 do
        for each edge e = (u, v) do
            Relax(e)
    if there is a tense edge e then
        Output that s can reach a negative cycle C
    else
        for each u ∈ V do
            output d(s, u)

Relax(e = (u, v)):
    if (d(s, v) > d(s, u) + ℓ(u, v)) then
        d(s, v) = d(s, u) + ℓ(u, v)
        prev(v) = u
```

Negative Cycle Detection

Negative Cycle Detection

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

1. **BellmanFord** 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 **BellmanFord** $|V|$ times, once from each node u?

Negative Cycle Detection

Negative Cycle Detection

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

1. **BellmanFord** 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 **BellmanFord** $|V|$ 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.
2. **BellmanFord** from s' will fill find a negative length cycle if there is one.
3. Exercise: why does this work?
4. Negative cycle detection can be done with one **BellmanFord** invocation.

Negative Cycle Detection

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

Negative Cycle Detection

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

Negative Cycle Detection

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

Running time for **BellmanFord**

1. Input graph $G = (V, E)$ with $m = |E|$ and $n = |V|$.
2. n outer iterations and m **Relax**() operations in each iteration. Each **Relax**() operation is $O(1)$ time.
3. Total running time: $O(mn)$.

Dijkstra's Algorithm with **Relax**()

```
for each node u ≠ s do
    d(s, u) = ∞
d(s, s) = 0
S = ∅
while (S ≠ V) do
    Let v be node in V − S with min d value
    S = S ∪ {v}
    for each edge e in Adj(v) do
        Relax(e)
```

Part III

Shortest Paths in DAGs

Shortest Paths in a DAG

Single-Source Shortest Path Problems

Input A directed acyclic graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.

Simplification of algorithms for DAGs

1. No cycles and hence no negative length cycles! Hence can find shortest paths even for negative length edges
2. Can order nodes using topological sort

Shortest Paths in a DAG

Single-Source Shortest Path Problems

Input: A directed acyclic graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes s, t find shortest path from s to t.
2. Given node s find shortest path from s to all other nodes.

Simplification of algorithms for DAGs

1. No cycles and hence no negative length cycles! Hence can find shortest paths even for negative length edges
2. Can order nodes using topological sort

Algorithm for DAGs

1. Want to find shortest paths from s. Ignore nodes not reachable from s.
2. Let $s = v_1, v_2, v_{i+1}, \ldots, v_n$ be a topological sort of G

Observation:

1. shortest path from s to v_i cannot use any node from $v_{i+1}, \ldots, v_n$
2. can find shortest paths in topological sort order.

Algorithm for DAGs

1. Want to find shortest paths from s. Ignore nodes not reachable from s.
2. Let $s = v_1, v_2, v_{i+1}, \ldots, v_n$ be a topological sort of G

Observation:

1. shortest path from s to v_i cannot use any node from $v_{i+1}, \ldots, v_n$
2. can find shortest paths in topological sort order.

Algorithm for DAGs

1. Code:

```
ShortestPathDAG:
    for i = 1 to n do
        d(s, v_i) = ∞
    d(s, s) = 0

    for i = 1 to n − 1 do
        for each edge e in Adj(v_i) do
            Relax(e)

    return d(s, ·) values computed
```

2. Correctness: induction on i and observation in previous slide.
3. Running time: $O(m + n)$ time algorithm! Works for negative edge lengths and hence can find *longest* paths in a DAG.

Algorithm for DAGs

1. Code:

```
ShortestPathDAG:
    for i = 1 to n do
        d(s, v_i) = ∞
    d(s, s) = 0

    for i = 1 to n − 1 do
        for each edge e in Adj(v_i) do
            Relax(e)

    return d(s, ·) values computed
```

2. Correctness: induction on i and observation in previous slide.
3. Running time: $O(m + n)$ time algorithm! Works for negative edge lengths and hence can find *longest* paths in a DAG.

Algorithm for DAGs

1. Code:

```
ShortestPathDAG:
    for i = 1 to n do
        d(s, v_i) = ∞
    d(s, s) = 0

    for i = 1 to n − 1 do
        for each edge e in Adj(v_i) do
            Relax(e)

    return d(s, ·) values computed
```

2. Correctness: induction on i and observation in previous slide.
3. Running time: $O(m + n)$ time algorithm! Works for negative edge lengths and hence can find *longest* paths in a DAG.

Takeaway Points

1. Shortest paths with potentially negative length edges arise in a variety of applications.
2. Longest simple path problem is difficult (no known efficient algorithm and **NP-Hard**).
3. Restrict attention to shortest walks. Well defined only if there are no negative length cycles reachable from the source.
4. In this case shortest walk = shortest path.
5. Generic shortest path algorithm starts with distance estimates to the source. Iteratively relaxes the edges one by one.
6. ...Guaranteed to terminate with correct distances if no negative length cycle reachable from s.
7. If negative length cycle reachable from $s \implies$ no termination.
8. Dijkstra's algorithm also instantiation of generic algorithm.

Takeaway Points

1. Shortest paths with potentially negative length edges arise in a variety of applications.
2. Longest simple path problem is difficult (no known efficient algorithm and **NP-Hard**).
3. Restrict attention to shortest walks. Well defined only if there are no negative length cycles reachable from the source.
4. In this case shortest walk = shortest path.
5. Generic shortest path algorithm starts with distance estimates to the source. Iteratively relaxes the edges one by one.
6. ...Guaranteed to terminate with correct distances if no negative length cycle reachable from s.
7. If negative length cycle reachable from $s \implies$ no termination.
8. Dijkstra's algorithm also instantiation of generic algorithm.

Takeaway Points

1. Shortest paths with potentially negative length edges arise in a variety of applications.
2. Longest simple path problem is difficult (no known efficient algorithm and **NP-Hard**).
3. Restrict attention to shortest walks. Well defined only if there are no negative length cycles reachable from the source.
4. In this case shortest walk = shortest path.
5. Generic shortest path algorithm starts with distance estimates to the source. Iteratively relaxes the edges one by one.
6. ...Guaranteed to terminate with correct distances if no negative length cycle reachable from s.
7. If negative length cycle reachable from $s \implies$ no termination.
8. Dijkstra's algorithm also instantiation of generic algorithm.

Takeaway Points

1. Shortest paths with potentially negative length edges arise in a variety of applications.
2. Longest simple path problem is difficult (no known efficient algorithm and **NP-Hard**).
3. Restrict attention to shortest walks. Well defined only if there are no negative length cycles reachable from the source.
4. In this case shortest walk = shortest path.
5. Generic shortest path algorithm starts with distance estimates to the source. Iteratively relaxes the edges one by one.
6. ...Guaranteed to terminate with correct distances if no negative length cycle reachable from s.
7. If negative length cycle reachable from $s \implies$ no termination.
8. Dijkstra's algorithm also instantiation of generic algorithm.

Takeaway Points

1. Shortest paths with potentially negative length edges arise in a variety of applications.
2. Longest simple path problem is difficult (no known efficient algorithm and **NP-Hard**).
3. Restrict attention to shortest walks. Well defined only if there are no negative length cycles reachable from the source.
4. In this case shortest walk = shortest path.
5. Generic shortest path algorithm starts with distance estimates to the source. Iteratively relaxes the edges one by one.
6. ...Guaranteed to terminate with correct distances if no negative length cycle reachable from s.
7. If negative length cycle reachable from $s \implies$ no termination.
8. Dijkstra's algorithm also instantiation of generic algorithm.

Takeaway Points

1. Shortest paths with potentially negative length edges arise in a variety of applications.
2. Longest simple path problem is difficult (no known efficient algorithm and **NP-Hard**).
3. Restrict attention to shortest walks. Well defined only if there are no negative length cycles reachable from the source.
4. In this case shortest walk = shortest path.
5. Generic shortest path algorithm starts with distance estimates to the source. Iteratively relaxes the edges one by one.
6. ...Guaranteed to terminate with correct distances if no negative length cycle reachable from s.
7. If negative length cycle reachable from $s \implies$ no termination.
8. Dijkstra's algorithm also instantiation of generic algorithm.

Takeaway Points

1. Shortest paths with potentially negative length edges arise in a variety of applications.
2. Longest simple path problem is difficult (no known efficient algorithm and **NP-Hard**).
3. Restrict attention to shortest walks. Well defined only if there are no negative length cycles reachable from the source.
4. In this case shortest walk = shortest path.
5. Generic shortest path algorithm starts with distance estimates to the source. Iteratively relaxes the edges one by one.
6. ...Guaranteed to terminate with correct distances if no negative length cycle reachable from s.
7. If negative length cycle reachable from $s \implies$ no termination.
8. Dijkstra's algorithm also instantiation of generic algorithm.

Points continued

1. **BellmanFord** is instantiation of generic algorithm.
2. ...in each iteration relaxes all the edges.
3. Discovers negative length cycles if there is tense edge in the nth iteration.
4. For vertex u with a shortest path to the source with i edges the algorithm has the correct distance after i iterations.
5. Running time of **BellmanFord** algorithm is $O(nm)$.
6. **BellmanFord** can be adapted to find a negative length cycle in the graph by adding a new vertex.
7. If we have a DAG then it has no negative length cycle and hence shortest paths exists even with negative lengths.
8. Can compute single-source shortest paths in DAG in linear time.
9. Implies one can compute longest paths in a DAG in linear time.

Points continued

1. **BellmanFord** is instantiation of generic algorithm.
2. ...in each iteration relaxes all the edges.
3. Discovers negative length cycles if there is tense edge in the nth iteration.
4. For vertex u with a shortest path to the source with i edges the algorithm has the correct distance after i iterations.
5. Running time of **BellmanFord** algorithm is $O(nm)$.
6. **BellmanFord** can be adapted to find a negative length cycle in the graph by adding a new vertex.
7. If we have a DAG then it has no negative length cycle and hence shortest paths exists even with negative lengths.
8. Can compute single-source shortest paths in DAG in linear time.
9. Implies one can compute longest paths in a DAG in linear time.

Points continued

1. **BellmanFord** is instantiation of generic algorithm.
2. ...in each iteration relaxes all the edges.
3. Discovers negative length cycles if there is tense edge in the nth iteration.
4. For vertex u with a shortest path to the source with i edges the algorithm has the correct distance after i iterations.
5. Running time of **BellmanFord** algorithm is $O(nm)$.
6. **BellmanFord** can be adapted to find a negative length cycle in the graph by adding a new vertex.
7. If we have a DAG then it has no negative length cycle and hence shortest paths exists even with negative lengths.
8. Can compute single-source shortest paths in DAG in linear time.
9. Implies one can compute longest paths in a DAG in linear time.

Points continued

1. **BellmanFord** is instantiation of generic algorithm.
2. ...in each iteration relaxes all the edges.
3. Discovers negative length cycles if there is tense edge in the nth iteration.
4. For vertex u with a shortest path to the source with i edges the algorithm has the correct distance after i iterations.
5. Running time of **BellmanFord** algorithm is $O(nm)$.
6. **BellmanFord** can be adapted to find a negative length cycle in the graph by adding a new vertex.
7. If we have a DAG then it has no negative length cycle and hence shortest paths exists even with negative lengths.
8. Can compute single-source shortest paths in DAG in linear time.
9. Implies one can compute longest paths in a DAG in linear time.

Points continued

1. **BellmanFord** is instantiation of generic algorithm.
2. ...in each iteration relaxes all the edges.
3. Discovers negative length cycles if there is tense edge in the nth iteration.
4. For vertex u with a shortest path to the source with i edges the algorithm has the correct distance after i iterations.
5. Running time of **BellmanFord** algorithm is $O(nm)$.
6. **BellmanFord** can be adapted to find a negative length cycle in the graph by adding a new vertex.
7. If we have a DAG then it has no negative length cycle and hence shortest paths exists even with negative lengths.
8. Can compute single-source shortest paths in DAG in linear time.
9. Implies one can compute longest paths in a DAG in linear time.

Points continued

1. **BellmanFord** is instantiation of generic algorithm.
2. ...in each iteration relaxes all the edges.
3. Discovers negative length cycles if there is tense edge in the nth iteration.
4. For vertex u with a shortest path to the source with i edges the algorithm has the correct distance after i iterations.
5. Running time of **BellmanFord** algorithm is $O(nm)$.
6. **BellmanFord** can be adapted to find a negative length cycle in the graph by adding a new vertex.
7. If we have a DAG then it has no negative length cycle and hence shortest paths exists even with negative lengths.
8. Can compute single-source shortest paths in DAG in linear time.
9. Implies one can compute longest paths in a DAG in linear time.

Points continued

1. **BellmanFord** is instantiation of generic algorithm.
2. ...in each iteration relaxes all the edges.
3. Discovers negative length cycles if there is tense edge in the nth iteration.
4. For vertex u with a shortest path to the source with i edges the algorithm has the correct distance after i iterations.
5. Running time of **BellmanFord** algorithm is $O(nm)$.
6. **BellmanFord** can be adapted to find a negative length cycle in the graph by adding a new vertex.
7. If we have a DAG then it has no negative length cycle and hence shortest paths exists even with negative lengths.
8. Can compute single-source shortest paths in DAG in linear time.
9. Implies one can compute longest paths in a DAG in linear time.

Points continued

1. **BellmanFord** is instantiation of generic algorithm.
2. ...in each iteration relaxes all the edges.
3. Discovers negative length cycles if there is tense edge in the nth iteration.
4. For vertex u with a shortest path to the source with i edges the algorithm has the correct distance after i iterations.
5. Running time of **BellmanFord** algorithm is $O(nm)$.
6. **BellmanFord** can be adapted to find a negative length cycle in the graph by adding a new vertex.
7. If we have a DAG then it has no negative length cycle and hence shortest paths exists even with negative lengths.
8. Can compute single-source shortest paths in DAG in linear time.
9. Implies one can compute longest paths in a DAG in linear time.

Points continued

1. **BellmanFord** is instantiation of generic algorithm.
2. ...in each iteration relaxes all the edges.
3. Discovers negative length cycles if there is tense edge in the nth iteration.
4. For vertex u with a shortest path to the source with i edges the algorithm has the correct distance after i iterations.
5. Running time of **BellmanFord** algorithm is $O(nm)$.
6. **BellmanFord** can be adapted to find a negative length cycle in the graph by adding a new vertex.
7. If we have a DAG then it has no negative length cycle and hence shortest paths exists even with negative lengths.
8. Can compute single-source shortest paths in DAG in linear time.
9. Implies one can compute longest paths in a DAG in linear time.

Questions for a possible written quiz...

(A) Given a directed graph $G = (V, E)$ with n vertices and m edges, describe how to compute a cycle in G if such a cycle exist. What is the running time of your algorithm?

(B) As above, but assume edges have weights (negative or positive). Describe how to detect a negative cycle in G?

(C) Describe how to modify your algorithm from (B) so that it outputs the negative cycle.

Questions for a possible written quiz...

(A) Given a directed graph $G = (V, E)$ with n vertices and m edges, describe how to compute a cycle in G if such a cycle exist. What is the running time of your algorithm?

(B) As above, but assume edges have weights (negative or positive). Describe how to detect a negative cycle in G?

(C) Describe how to modify your algorithm from (B) so that it outputs the negative cycle.

Questions for a possible written quiz...

(A) Given a directed graph $G = (V, E)$ with n vertices and m edges, describe how to compute a cycle in G if such a cycle exist. What is the running time of your algorithm?

(B) As above, but assume edges have weights (negative or positive). Describe how to detect a negative cycle in G?

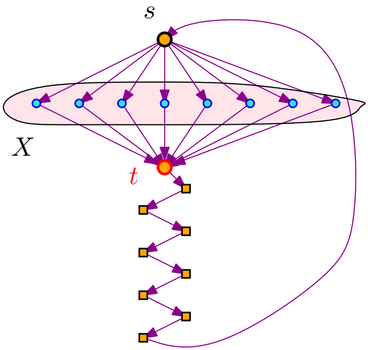
(C) Describe how to modify your algorithm from (B) so that it outputs the negative cycle.

Part IV

Not for lecture

A shortest walk that visits all vertices...

... in a graph might have to be of length $\mathbf{\Omega(n^2)}$



Notes

Notes

Notes

Notes