Shortest Paths with Negative Lengths and DP

Lecture 18

March 30, 2023

Part I

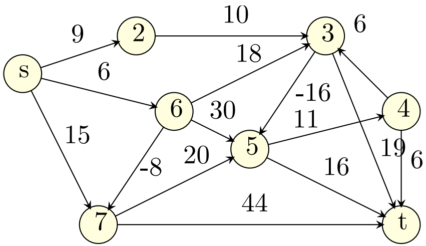
Shortest Paths with Negative Length Edges

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.

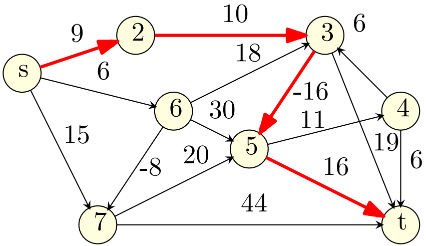


Single-Source Shortest Paths with Negative Edge Lengths

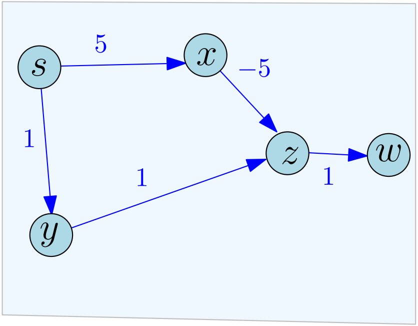
Single-Source Shortest Path Problems

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.



What are the distances computed by Dijkstra's algorithm?

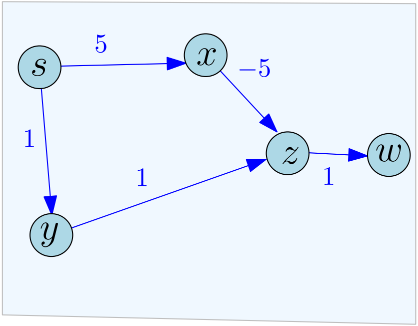


The distance as computed by Dijkstra algorithm starting from s:

(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.

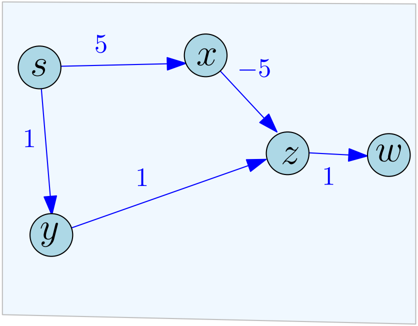
Dijkstra's Algorithm and Negative Lengths

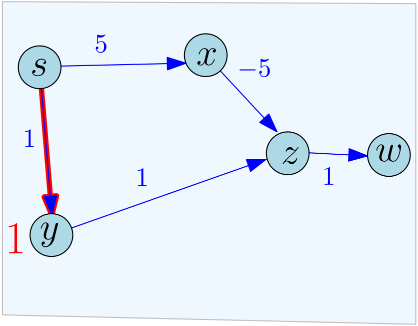
With negative length edges, Dijkstra's algorithm can fail



Dijkstra's Algorithm and Negative Lengths

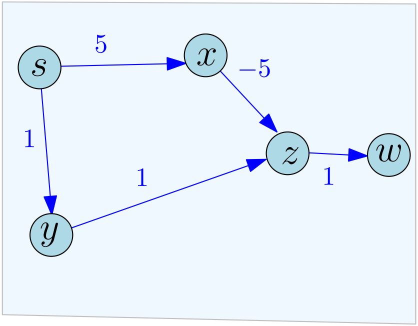
With negative length edges, Dijkstra's algorithm can fail

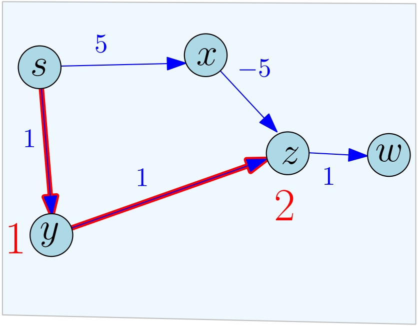




Dijkstra's Algorithm and Negative Lengths

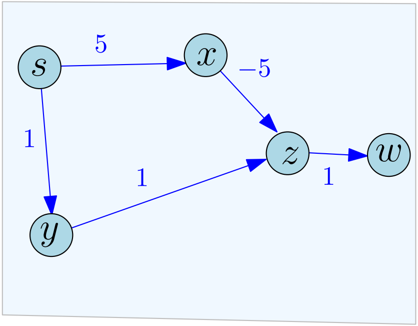
With negative length edges, Dijkstra's algorithm can fail

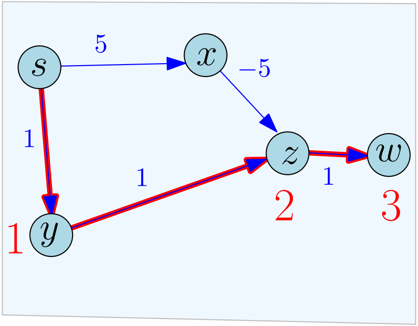




Dijkstra's Algorithm and Negative Lengths

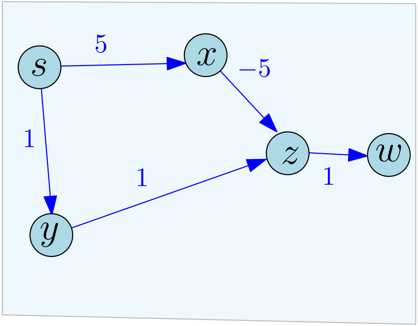
With negative length edges, Dijkstra's algorithm can fail

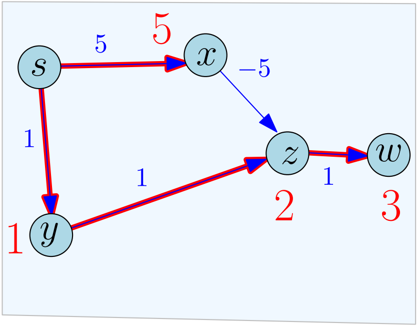




Dijkstra's Algorithm and Negative Lengths

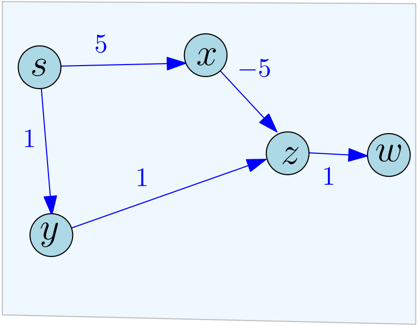
With negative length edges, Dijkstra's algorithm can fail

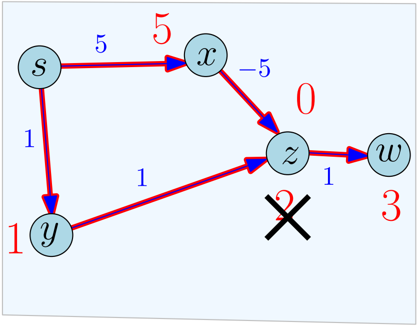




Dijkstra's Algorithm and Negative Lengths

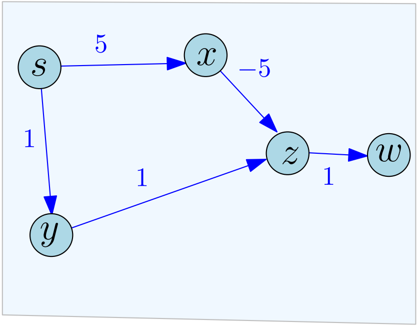
With negative length edges, Dijkstra's algorithm can fail

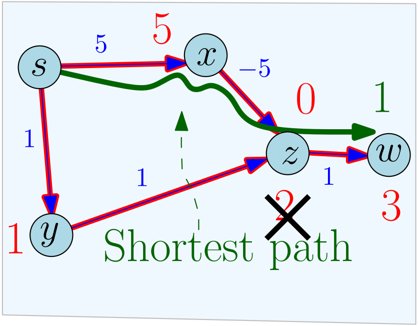




Dijkstra's Algorithm and Negative Lengths

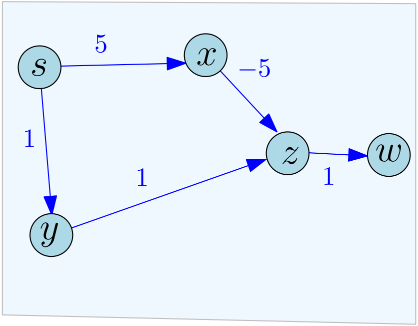
With negative length edges, Dijkstra's algorithm can fail

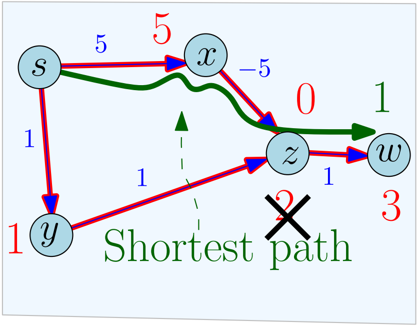




Dijkstra's Algorithm and Negative Lengths

With negative length edges, Dijkstra's algorithm can fail



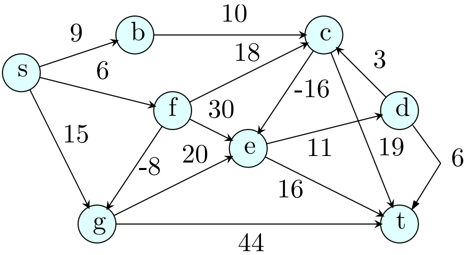


False assumption: Dijkstra's algorithm is based on the assumption that if $s = v_0 \rightarrow v_1 \rightarrow v_2 \ldots \rightarrow v_k$ is a shortest path from s to v_k then $dist(s, v_i) \leq dist(s, v_{i+1})$ for $0 \leq i < k$. Holds true only for non-negative edge lengths.

Negative Length Cycles

Definition

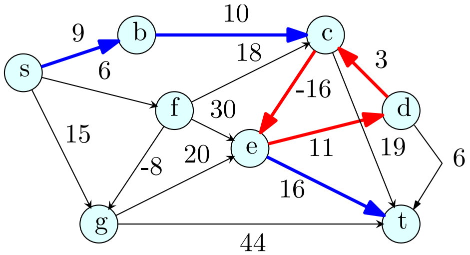
A cycle C is a negative length cycle if the sum of the edge lengths of C is negative.



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?
Possible answers:

1. undefined, that is $-\infty$, OR
2. the length of a shortest **simple** path 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?

Possible answers:

1. undefined, that is $-\infty$, OR
2. the length of a shortest **simple** path from s to t.

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**!

Alterantively: Finding Shortest Walks

Given a graph $G = (V, E)$:

1. A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$.
2. A walk is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. Vertices can repeat.

Define $dist(u, v)$ to be the length of a shortest walk from u to v.

1. If there is a walk from u to v that contains negative length cycle then $dist(u, v) = -\infty$
2. Else there is a path with at most $n - 1$ edges whose length is equal to the length of a shortest walk and $dist(u, v)$ is finite

Helpful to think about walks

Shortest Paths with Negative Edge Lengths

Problems

Algorithmic Problems

Input: A directed graph $G = (V, E)$ with edge lengths (could be negative). 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 lenths 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)?

Concrete example:

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

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

Reducing Currency Trading to Shortest Paths

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.

Reducing Currency Trading to Shortest Paths

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.

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 $\ell(v_i, v_j) =$

Reducing Currency Trading to Shortest Paths

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.

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

Reducing Currency Trading to Shortest Paths

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.

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

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

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 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

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.*

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 other strategies.

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.
Assume that all nodes can be reached by s in G

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

Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source s.
Assume that all nodes can be reached by s in G

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

Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source s.
Assume that all nodes can be reached by s in G

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

Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source s.
Assume that all nodes can be reached by s in G

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

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

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

Algorithm for Hop-Constrained Walks

Problem: Given $G = (V, E)$ with edge lengths, s and integer bound h. For each v find shortest s-v walk length with at most h edges. That is, $d(v, h)$ for all $v \in V$.

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

for k = 1 to h 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)}
```

Algorithm for Hop-Constrained Walks

Problem: Given $G = (V, E)$ with edge lengths, s and integer bound h. For each v find shortest s-v walk length with at most h edges. That is, $d(v, h)$ for all $v \in V$.

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

for k = 1 to h 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)}
```

Running time:

Algorithm for Hop-Constrained Walks

Problem: Given $G = (V, E)$ with edge lengths, s and integer bound h. For each v find shortest s-v walk length with at most h edges. That is, $d(v, h)$ for all $v \in V$.

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

for k = 1 to h 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)}
```

Running time: $O(mh)$

Algorithm for Hop-Constrained Walks

Problem: Given $G = (V, E)$ with edge lengths, s and integer bound h. For each v find shortest s-v walk length with at most h edges. That is, $d(v, h)$ for all $v \in V$.

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

for k = 1 to h 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)}
```

Running time: $O(mh)$ Space:

Algorithm for Hop-Constrained Walks

Problem: Given $G = (V, E)$ with edge lengths, s and integer bound h. For each v find shortest s-v walk length with at most h edges. That is, $d(v, h)$ for all $v \in V$.

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

for k = 1 to h 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)}
```

Running time: $O(mh)$ Space: $O(m + nh)$

Algorithm for Hop-Constrained Walks

Problem: Given $G = (V, E)$ with edge lengths, s and integer bound h. For each v find shortest s-v walk length with at most h edges. That is, $d(v, h)$ for all $v \in V$.

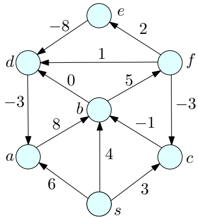
```
for each u ∈ V do
    d(u, 0) ← ∞
d(s, 0) ← 0

for k = 1 to h 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)}
```

Running time: $O(mh)$ Space: $O(m + nh)$

Space can be reduced to $O(m + n)$

Example



Bellman-Ford Algorithm

Based on the following three lemmas:

Lemma

There is an $O(mn)$ time and $O(m + n)$ space algorithm that computes $d(v, n - 1)$ and $d(v, n)$ for all $v \in V$.

Lemma

Suppose there is no negative length cycle in G then for each $v \in V$, $d(v, n - 1)$ is the shortest path distance from s to v.

Lemma

Suppose there is a negative length cycle C in G that is reachable from s. Then $d(v, n) < d(v, n - 1)$ for some $v \in V$.

Bellman-Ford Algorithm

1. Compute $d(v, n-1)$ and $d(v, n)$ for each $v \in V$
2. If there is any v such that $d(v, n) < d(v, n-1)$ then output that there is a negative length cycle.
3. Else, for each $v \in V$, $\text{dist}(s, v) = d(v, n-1)$.

$O(mn)$ time and $O(m+n)$ space

Bellman-Ford Algorithm

Keep track of only one value $d(v)$ for each v which stands for $d(v, k)$ as k changes from 0 to 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)
```

Correctness of the Bellman-Ford Algorithm

Lemma

There is an $O(mn)$ time and $O(m+n)$ space algorithm that computes $d(v, n-1)$ and $d(v, n)$ for all $v \in V$.

Proof via induction on k that $d(v, k)$ is the length of a shortest walk from s to v with *at most* k hops. We saw that the algorithm runs in $O(mn)$ time and $O(m+n)$ space.

Observation

If all vertices are reachable from s then $d(v, n-1) < \infty$ (finite).

Correctness of the Bellman-Ford Algorithm

Lemma

Suppose G does not have a negative length cycle and all nodes are reachable from s. Then for all v, $d(v, n-1) = d(v, n)$ and $\text{dist}(s, v) = d(v, n-1)$.

Proof.

No negative length cycle means shortest walk length is same as shortest path length. A path can have at most $n-1$ edges and hence $\text{dist}(s, v) = d(v, n-1)$ and $d(v, n-1) = d(v, n)$. □

Alternatively: suppose $d(v, n) < d(v, n-1)$. Consider s-v walk W that achieves $d(v, n)$. No negative length cycles $\Rightarrow$ can remove cycles from W to get s-v path P such that $\ell(W) = \ell(P)$. Then $d(v, h) = d(v, n)$ for some $h < n$, and $d(v, n-1) \leq d(v, h)$ which implies that $d(v, n-1) = d(v, n)$, a contradiction.

Detecting negative length cycle

Lemma

Suppose G has a negative cycle C reachable from s. Then there is some node $v \in C$ such that $d(v, n) < d(v, n-1)$.

Detecting negative length cycle

Lemma

Suppose G has a negative cycle C reachable from s. Then there is some node $v \in C$ such that $d(v, n) < d(v, n-1)$.

Proof by contradiction. Let $C = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_h \rightarrow v_1$ be a negative length cycle in G.

- $d(v_i, n-1)$ is finite for $1 \leq i \leq h$ by observation.
- Suppose $d(v, n) \geq d(v, n-1)$ for all $v \in C$
- This means $d(v_i, n-1) \leq d(v_{i-1}, n-1) + \ell(v_{i-1}, v_i)$ for $2 \leq i \leq h$ and $d(v_1, n-1) \leq d(v_n, n-1) + \ell(v_n, v_1)$. Because if $d(v_i, n-1) > d(v_{i-1}, n-1) + \ell(v_{i-1}, v_i)$ we would have $d(v_i, n) \leq d(v_{i-1}, n-1) + \ell(v_{i-1}, v_i)$ and $d(v_i, n) < d(v_i, n-1)$.
- Adding up all these inequalities results in the inequality $0 \leq \ell(C)$ which contradicts the assumption that $\ell(C) < 0$.

Detecting negative length cycle

A concrete setting. Assume cycle is $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_1$
Via the recursion and using the edges in the cycle we have

$$d(v_2, n-1) \leq d(v_1, n-1) + \ell(v_1, v_2) \text{ since } d(v_2, n) \geq d(v_2, n-1)$$
$$d(v_3, n-1) \leq d(v_2, n-1) + \ell(v_2, v_3) \text{ since } d(v_3, n) \geq d(v_3, n-1)$$
$$d(v_4, n-1) \leq d(v_3, n-1) + \ell(v_3, v_4) \text{ since } d(v_4, n) \geq d(v_4, n-1)$$
$$d(v_1, n-1) \leq d(v_4, n-1) + \ell(v_4, v_1) \text{ since } d(v_1, n) \geq d(v_1, n-1)$$

Adding up both sides:

$$d(v_1, n-1) + d(v_2, n-1) + d(v_3, n-1) + d(v_4, n-1) \leq d(v_1, n-1) + d(v_2, n-1) + d(v_3, n-1) + d(v_4, n-1) + \ell(C)$$

$\Rightarrow \ell(C) \geq 0$ a contradiction

An easier lemma about negative cycle detection

Lemma

Suppose G has a negative length cycle C reachable from s. Let v be any node on C. Then $d(v, n-1+p) < d(v, n-1)$ where p is the number of edges in C.

An easier lemma about negative cycle detection

Lemma

Suppose G *has a negative length cycle* C *reachable from* s*. Let* v *be any node on* C*. Then* $d(v, n-1+p) < d(v, n-1)$ *where* p *is the number of edges in* C*.*

Proof.

Consider s-v walk W that achieves $d(v, n-1)$. If we concatenate W and C we get another walk W' such that $\ell(W') = \ell(W) + \ell(C) < \ell(W)$ since $\ell(C) < 0$. W' has $|W| + p$ edges, hence $d(v, n-1+p) < d(v, n-1)$. □

The lemma shows that running Bellman-Ford for $2n-1$ iterations suffices to detect negative cycle. The stronger lemma says that n iterations suffice.

Finding the Paths and a Shortest Path Tree

How do we find a shortest path tree in addition to distances?

- For each v the $d(v)$ can only get smaller as algorithm proceeds.
- If $d(v)$ becomes smaller it is because we found a vertex u such that $d(v) > d(u) + \ell(u, v)$ and we update $d(v) = d(u) + \ell(u, v)$. That is, we found a shorter path to v through u.
- For each v have a $prev(v)$ pointer and update it to point to u if v finds a shorter path via u.
- At end of algorithm $prev(v)$ pointers give a shortest path tree oriented towards the source s.

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 fill 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 on a graph with one extra node.

Finding a negative length cycle

Question: How can we find a negative length cycle if it has one?

Finding a negative length cycle

Question: How can we find a negative length cycle if it has one?

Algorithm is simple

- In iteration n algorithm finds first v such that $d(v, n) < d(v, n-1)$ via u. Update $prev(v)$ pointer to u (interesting case is when $v = s$)
- There must be a cycle in the graph induced by $prev()$ pointers and it must be of negative length

Finding a negative length cycle

Question: How can we find a negative length cycle if it has one?

Algorithm is simple

- In iteration n algorithm finds first v such that $d(v, n) < d(v, n-1)$ via u. Update $prev(v)$ pointer to u (interesting case is when $v = s$)
- There must be a cycle in the graph induced by $prev()$ pointers and it must be of negative length

Proof is not straight forward to see. See next two slides

Note: Negative cycles can get created and removed along the way

Finding a negative length cycle

Properties of the algorithm:

- For each $v \neq s$, $prev(v)$ is a single back pointer and hence the graph induced by $prev()$ pointers consists of a forest rooted at s, collection of cycles, and isolated vertices (all disjoint)
- By induction one can show that if $prev(v) = u$ implies that there is an s-v walk whose last edge is (u, v) that achieves the current distance label $d(v)$ for v. In particular if there is a path from v to s using $prev$ pointers from v then that walk is a current shortest walk to v.
- By induction one can show that if there is a cycle in the graph induced by $prev()$ pointers at any stage of the algorithm then it must have negative length. This is the key property and the proof can be shown using the last edge that created the cycle and using a proof similar to the one for detecting negative cycle.

Finding a negative length cycle

- Consider the $prev()$ pointer graph after $n-1$ iterations. If there is a cycle then it must be negative
- Suppose there is no cycle. Then since all $d(v, n-1)$ values are finite, the $prev()$ pointers induce a tree rooted at s. Thus each node v has a *path* from s whose length is equal to $d(v, n-1)$.
- Algorithm found some v s.t $d(v, n) < d(v, n-1)$. There is $u \neq v$ such that $d(u, n-1) + \ell(u, v) < d(v, n-1)$.
  - Case 1: $v = s$. Implies that $d(s, n) < 0$, and the edge (u, s) together with the path from s to u in the current tree is a s-s cycle of length < 0
  - Case 2: $v \neq s$ and u is a decendent of v in the current tree of $prev()$ pointers — then updating $prev(v) = u$ will create a negative length cycle containing v
  - Case 3: $v \neq s$ and u is not a descent of v in current tree. Updating $prev(v) = u$ creates new tree and *path* to v with length $d(v, n)$, a contradition. Cannot happen.

Faster Algorithms?

Bellman-Ford algorithm is from 50's. Are there faster algorithms? Yes!

Bernstein-Nanongkai-WulffNilsen, 2022: randomized $O(m \log^8 n \log L)$ time algorithm where edge weights are integral and $L = \max_e |\ell(e)|$.
https://arxiv.org/pdf/2203.03456.pdf

Part II

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.

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

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

Assumption: s is first in the topological sort

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

for i = 1 to n − 1 do
        for each edge (v_i, v_j) in Adj(v_i) do
                d(s, v_j) = min{d(s, v_j), d(s, v_i) + ℓ(v_i, v_j)}

return d(s, ·) values computed
```

Correctness: induction on i and observation in previous slide.

Running time: $O(m + n)$ time algorithm! Works for negative edge lengths and hence can find *longest* paths in a DAG.

Algorithm for DAGs, a variant

Assumption: s is first in the topological sort

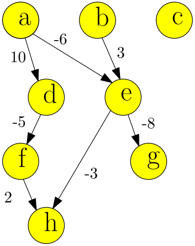
```
for i = 1 to n do
        d(s, v_i) = ∞
d(s, s) = 0

for i = 2 to n − 1 do
        for each edge (v_j, v_i) in In(v_i) do
            d(s, v_i) = min{d(s, v_i), d(s, v_j) + ℓ(v_j, v_i)}

return d(s, ·) values computed
```

When visiting v_i scan incoming edges to find shortest path to i.
Previous algorithm scanned all edges in $\mathbf{Adj}(v_i)$ *after* processing v_i.
Can see algorithms are same.

Algorithm for DAGs: Example



Want distances from a say. Consider topological sort:
a, b, c, d, f, e, h, g

Bellman-Ford and DAGs

Bellman-Ford relies on hop-constrained walks

We can find hop-constrained shortest walks via graph reduction.

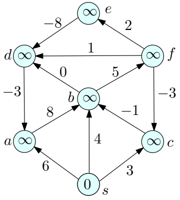
Given $G = (V, E)$ with edge lengths $\ell(e)$ and integer k construction new layered graph $G' = (V', E')$ as follows.

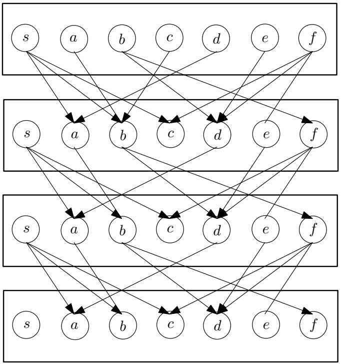
- $V' = V \times \{0, 1, 2, \ldots, k\}$.
- $E' = \{((u, i), (v, i+1) \mid (u, v) \in E, 0 \leq i < k\}$, $\ell((u, i), (v, i+1)) = \ell(u, v)$

Lemma

Shortest path distance from $(u, 0)$ to (v, k) in G' is equal to the shortest walk from u to v in G with exactly k edges.

Layered DAG: Figure

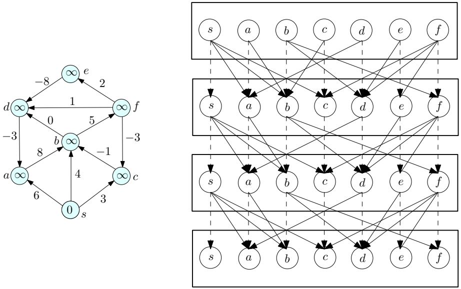




Edge lengths in DAG are same as in original graph

Layered DAG: Figure

Suppose we want $(u, 0)$ to (v, k) in G' to give us the shortest walk from u to v in G with *at most* k edges. We add 0 length edges between (u, i) and $(u, i+1)$ for each u, i as shown in the figure.



Edge lengths in DAG are same as in original graph except dashed edges which have 0 length

Part III

All Pairs Shortest Paths

Shortest Path Problems

Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths (or costs). 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.
3. Find shortest paths for *all* pairs of nodes.

Single-Source Shortest Paths

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$ with 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

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $G = (V, E)$ with 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.

Dijkstra's algorithm for non-negative edge lengths. Running time: $O((m + n) \log n)$ with heaps and $O(m + n \log n)$ with advanced priority queues.

Bellman-Ford algorithm for arbitrary edge lengths. Running time: $O(nm)$.

All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

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

1. Find shortest paths for all pairs of nodes.

All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

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

1. Find shortest paths for all pairs of nodes.

Apply single-source algorithms n times, once for each vertex.

1. Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.
2. Arbitrary edge lengths: $O(n^2 m)$.
   $\Theta(n^4)$ if $m = \Omega(n^2)$.

All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

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

1. Find shortest paths for all pairs of nodes.

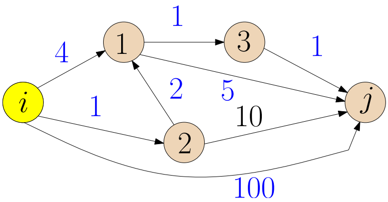
Apply single-source algorithms n times, once for each vertex.

1. Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.
2. Arbitrary edge lengths: $O(n^2 m)$.
   $\Theta(n^4)$ if $m = \Omega(n^2)$.

Can we do better?

All-Pairs: Recursion on index of intermediate nodes

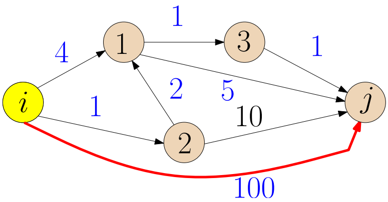
1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
2. $dist(i, j, k)$: length of shortest walk from v_i to v_j among all walks in which the largest **index** of an **intermediate node** is **at most** k (could be $-\infty$ if there is a negative length cycle).



$$dist(i, j, 0) =$$
$$dist(i, j, 1) =$$
$$dist(i, j, 2) =$$
$$dist(i, j, 3) =$$

All-Pairs: Recursion on index of intermediate nodes

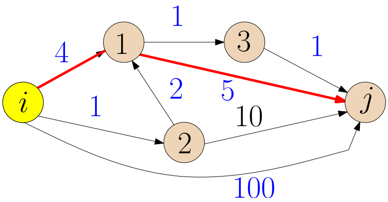
1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
2. $dist(i, j, k)$: length of shortest walk from v_i to v_j among all walks in which the largest **index** of an **intermediate node** is **at most** k (could be $-\infty$ if there is a negative length cycle).



$$
\begin{aligned}
dist(i, j, 0) &= 100 \\
dist(i, j, 1) &= \\
dist(i, j, 2) &= \\
dist(i, j, 3) &=
\end{aligned}
$$

All-Pairs: Recursion on index of intermediate nodes

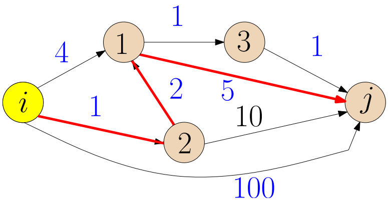
1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
2. $dist(i, j, k)$: length of shortest walk from v_i to v_j among all walks in which the largest **index** of an **intermediate node** is **at most** k (could be $-\infty$ if there is a negative length cycle).



$$
\begin{aligned}
dist(i, j, 0) &= 100 \\
dist(i, j, 1) &= 9 \\
dist(i, j, 2) &= \\
dist(i, j, 3) &=
\end{aligned}
$$

All-Pairs: Recursion on index of intermediate nodes

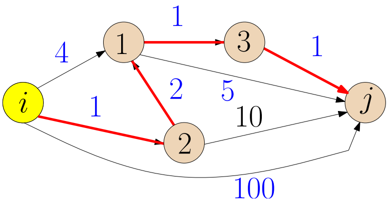
1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
2. $dist(i, j, k)$: length of shortest walk from v_i to v_j among all walks in which the largest **index** of an **intermediate node** is **at most** k (could be $-\infty$ if there is a negative length cycle).



$$dist(i, j, 0) = 100$$
$$dist(i, j, 1) = 9$$
$$dist(i, j, 2) = 8$$
$$dist(i, j, 3) =$$

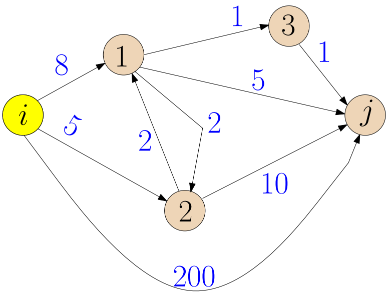
All-Pairs: Recursion on index of intermediate nodes

1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
2. $dist(i, j, k)$: length of shortest walk from v_i to v_j among all walks in which the largest **index** of an **intermediate node** is **at most** k (could be $-\infty$ if there is a negative length cycle).



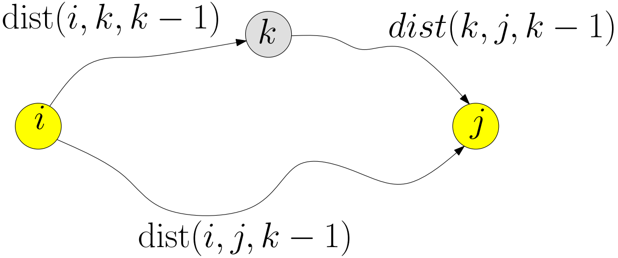
$$
\begin{aligned}
dist(i, j, 0) &= 100 \\
dist(i, j, 1) &= 9 \\
dist(i, j, 2) &= 8 \\
dist(i, j, 3) &= 5
\end{aligned}
$$

For the following graph, $\text{dist}(i, j, 2)$ is



- 9
- 10
- 11
- 12
- 15

All-Pairs: Recursion on index of intermediate nodes



$$dist(i,j,k) = \min \begin{cases} dist(i,j,k-1) \\ dist(i,k,k-1) + dist(k,j,k-1) \end{cases}$$

Base case: $dist(i,j,0) = \ell(i,j)$ if $(i,j) \in E$, otherwise ∞

Correctness: If $i \to j$ shortest walk goes through k then k occurs only once on the path — otherwise there is a negative length cycle.

All-Pairs: Recursion on index of intermediate nodes

If i can reach k *and* k can reach j *and* $dist(k, k, k-1) < 0$ then G has a negative length cycle containing k and $dist(i, j, k) = -\infty$.

Recursion below is valid only if $dist(k, k, k-1) = 0$. We can detect this during the algorithm or wait till the end.

$$dist(i, j, k) = \min \begin{cases} dist(i, j, k-1) \\ dist(i, k, k-1) + dist(k, j, k-1) \end{cases}$$

Alternatively:

$$dist(i, j, k) = \min \begin{cases} dist(i, j, k-1) \\ dist(i, k, k-1) + dist(k, k, k-1) + dist(k, j, k-1) \end{cases}$$

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

```
for i = 1 to n do
    for j = 1 to n do
        dist(i, j, 0) = ℓ(i, j)  (* ℓ(i, j) = ∞ if (i, j) ∉ E, 0 if i = j *)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i, j, k) = min { dist(i, j, k − 1),
                                  dist(i, k, k − 1) + dist(k, j, k − 1) }
for i = 1 to n do
    if (dist(i, i, n) < 0) then
        Output that there is a negative length cycle in G
```

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

```
for i = 1 to n do
    for j = 1 to n do
        dist(i, j, 0) = ℓ(i, j)  (* ℓ(i, j) = ∞ if (i, j) ∉ E, 0 if i = j *)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i, j, k) = min { dist(i, j, k − 1),
                                  dist(i, k, k − 1) + dist(k, j, k − 1) }
for i = 1 to n do
    if (dist(i, i, n) < 0) then
        Output that there is a negative length cycle in G
```

Running Time:

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

```
for i = 1 to n do
    for j = 1 to n do
        dist(i,j,0) = ℓ(i,j)  (* ℓ(i,j) = ∞ if (i,j) ∉ E, 0 if i = j *)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i,j,k) = min { dist(i,j,k−1),
                                dist(i,k,k−1) + dist(k,j,k−1) }
for i = 1 to n do
    if (dist(i,i,n) < 0) then
        Output that there is a negative length cycle in G
```

Running Time: $\Theta(n^3)$, Space: $\Theta(n^3)$.

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

```
for i = 1 to n do
    for j = 1 to n do
        dist(i,j,0) = ℓ(i,j)  (* ℓ(i,j) = ∞ if (i,j) ∉ E, 0 if i = j *)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i,j,k) = min { dist(i,j,k−1),
                                dist(i,k,k−1) + dist(k,j,k−1)
for i = 1 to n do
    if (dist(i,i,n) < 0) then
        Output that there is a negative length cycle in G
```

Running Time: $\Theta(n^3)$, Space: $\Theta(n^3)$.

Correctness: via induction and recursive definition

Floyd-Warshall Algorithm: Finding the Paths

Question: Can we find the paths in addition to the distances?

Floyd-Warshall Algorithm: Finding the Paths

Question: Can we find the paths in addition to the distances?

1. Create a $n \times n$ array Next that stores the next vertex on shortest path for each pair of vertices
2. With array Next, for any pair of given vertices i, j can compute a shortest path in $O(n)$ time.

Floyd-Warshall Algorithm

Finding the Paths

```
for i = 1 to n do
    for j = 1 to n do
        dist(i, j, 0) = ℓ(i, j)
(* ℓ(i, j) = ∞ if (i, j) not edge, 0 if i = j *)
        Next(i, j) = −1
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            if (dist(i, j, k − 1) > dist(i, k, k − 1) + dist(k, j, k − 1)) then
                dist(i, j, k) = dist(i, k, k − 1) + dist(k, j, k − 1)
                Next(i, j) = k

for i = 1 to n do
    if (dist(i, i, n) < 0) then
        Output that there is a negative length cycle in G
```

Exercise: Given *Next* array and any two vertices i, j describe an $O(n)$ algorithm to find a i-j shortest path.

Summary of results on shortest paths

Single source		
No negative edges	Dijkstra	$O(n \log n + m)$
Edge lengths can be negative	Bellman Ford	$O(nm)$

All Pairs Shortest Paths

No negative edges	n * Dijkstra	$O(n^2 \log n + nm)$
No negative cycles	n * Bellman Ford	$O(n^2 m) = O(n^4)$
No negative cycles	BF + n * Dijkstra	$O(nm + n^2 \log n)$
No negative cycles	Floyd-Warshall	$O(n^3)$
Unweighted	Matrix multiplication	$O(n^{2.38})$, $O(n^{2.58})$

Part IV

Dynamic Programming, DAGs, and Shortest Paths

Recursion and DAGs

Suppose we have a recursive program $foo(x)$ that takes an input x

- $foo(x)$ generates a recursion *tree* where a subproblem z is a child of subproblem y if $foo(y)$ calls $foo(z)$ during its execution. Note that the same subproblem/instance can occur many times in the tree.
- In DP we are interested in the *distinct* subproblems generated by $foo(x)$. We can create a natural *DAG* with the recursion
  - Each distinct subproblem corresponds to a node
  - If $foo(y)$ calls $foo(z)$ we add arc from y to z.

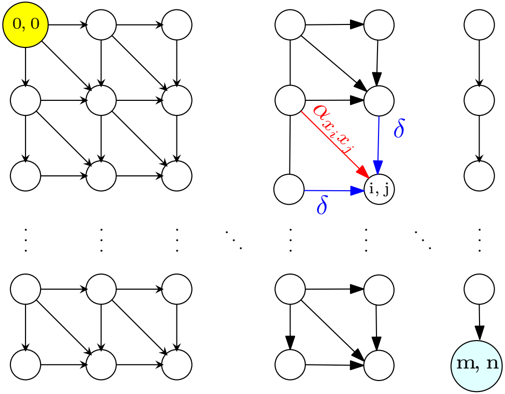
The *dependency graph* for recursion is naturally acyclic

DP and DAGs: Examples

Computing $\textit{Fib}(n)$ via recursion $\textit{Fib}(n) = \textit{Fib}(n-1) + \textit{Fib}(n-2)$

DP and DAGs: Examples

Edit Distance between strings $X[1..m]$ and $Y[1..n]$



DP and DAGs: Evaluation Order

Converting recursive algorithm to iterative algorithm in DP:

- Identify the structure of subproblems to estimate number
- Allocate appropriate data structure to store the subproblems
- Evaluate the subproblems in the *right order*

DP and DAGs: Evaluation Order

Converting recursive algorithm to iterative algorithm in DP:

- Identify the structure of subproblems to estimate number
- Allocate appropriate data structure to store the subproblems
- Evaluate the subproblems in the *right order*

Question: what is the right order? Can we automate it?

DP and DAGs: Evaluation Order

Converting recursive algorithm to iterative algorithm in DP:

- Identify the structure of subproblems to estimate number
- Allocate appropriate data structure to store the subproblems
- Evaluate the subproblems in the *right order*

Question: what is the right order? Can we automate it?

Yes. Compute the DAG. Evaluation order is the reverse of a topological sort of the DAG

DP and DAGs: Evaluation Order

Converting recursive algorithm to iterative algorithm in DP:

- Identify the structure of subproblems to estimate number
- Allocate appropriate data structure to store the subproblems
- Evaluate the subproblems in the *right order*

Question: what is the right order? Can we automate it?

Yes. Compute the DAG. Evaluation order is the reverse of a topological sort of the DAG

Question: Why not automate evaluation order?

DP and DAGs: Evaluation Order

Converting recursive algorithm to iterative algorithm in DP:

- Identify the structure of subproblems to estimate number
- Allocate appropriate data structure to store the subproblems
- Evaluate the subproblems in the *right order*

Question: what is the right order? Can we automate it?

Yes. Compute the DAG. Evaluation order is the reverse of a topological sort of the DAG

Question: Why not automate evaluation order?

- Creating the DAG explicitly is cumbersome, and wasteful in space/time.
- For many DP problems the subproblem and DAG structure is simple and can be exploited for efficiency

DP and DAGs: the other way

We saw that dependency graph of a recursion is a DAG and we can use graph/DAG properties to help us with DP.

Sometimes it is feasible to reduce a problem to a DAG computation directly without realizing that it came from a DP.

Examples:

- Longest Increasing Subsequence
- Bellman-Ford and Hop-Constrained Walks (we saw already)

Reducing Longest Increasing Subsequence to longest path in a DAG

LIS: given a sequence, find the longest increasing subsequence

Example

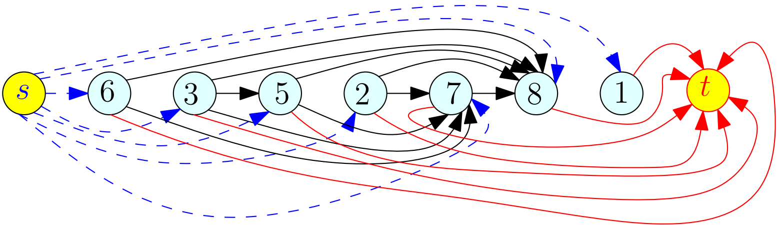
1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
3. Longest increasing subsequence: 3, 5, 7, 8

Reducing Longest Increasing Subsequence to longest path in a DAG

LIS: given a sequence, find the longest increasing subsequence

Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
3. Longest increasing subsequence: 3, 5, 7, 8



Dynamic Programming: Postscript

Dynamic Programming = Smart Recursion + Memoization

Dynamic Programming: Postscript

Dynamic Programming = Smart Recursion + Memoization

1. How to come up with the recursion?
2. How to recognize that dynamic programming may apply?

Some Tips

1. Problems where there is a *natural* linear ordering: sequences, paths, intervals, DAGs etc. Recursion based on ordering (left to right or right to left or topological sort) usually works.
2. Problems involving trees: recursion based on subtrees.
3. More generally:
    1. Problem admits a natural recursive divide and conquer
    2. If optimal solution for whole problem can be simply composed from optimal solution for each separate pieces then plain divide and conquer works directly
    3. If optimal solution depends on all pieces then can apply dynamic programming if *interface/interaction* between pieces is *limited*. Augment recursion to not simply find an optimum solution but also an optimum solution for each possible way to interact with the other pieces.

Examples

1. Longest Increasing Subsequence: break sequence in the middle say. What is the interaction between the two pieces in a solution?
2. Sequence Alignment: break both sequences in two pieces each. What is the interaction between the two sets of pieces?
3. Independent Set in a Tree: break tree at root into subtrees. What is the interaction between the subtrees?
4. Independent Set in an graph: break graph into two graphs. What is the interaction? Very high!
5. Knapsack: Split items into two sets of half each. What is the interaction?