Dynamic Programming: Shortest Paths

Algorithm Design & Analysis

Spring 2018
Outline
The Problem

Allow negative edge costs in the shortest paths problem.

Assume no negative cost cycles.
The RoadMap

• Establish appropriate optimality recurrence
• Determine complexity
• Reduce space requirements
• Preserve ability to find path given reduced space
The Bellman-Ford Algorithm

Assume $G$ directed with edge costs $c(u, v)$ for each $(u, v) \in E$

Let $opt(i, v) = \min \{ c(P) : P \text{ is a } v-t \text{ path of length } \leq i \}$

Heads-Up:

- Path length refers to number of edges on path
- Path cost refers to sum of costs of edges on path
The Bellman-Ford Algorithm

Assume $G$ directed with edge costs $c(u, v)$ for each $(u, v) \in E$

Let $opt(i, v) = \min\{c(P) : P$ is a $v - t$ path of length $\leq i\}$

Heads-Up:
- Path $length$ refers to number of edges on path
- Path $cost$ refers to sum of costs of edges on path

Note that
- $opt(i, t) = 0$ and $opt(0, v) = \infty$ if $v \neq t$.
- $opt(1, v) = c(v, t)$ if $(v, t) \in E$; $opt(1, v) = \infty$ otherwise.
**The Optimality Recurrence**

Let $P$ be a minimum-cost path from $v$ to $t$ using at most $i$ edges.
The Optimality Recurrence

Let $P$ be a minimum-cost path from $v$ to $t$ using at most $i$ edges.

Observe that
The Optimality Recurrence

Let $P$ be a minimum-cost path from $v$ to $t$ using at most $i$ edges.

Observe that

- If $P$ has length less than $i$, then $opt(i, v) = opt(i - 1, v)$.
- If $P$ has length $i$, $P$ consists of some edge $(v, u)$ and a path of length $i - 1$ from $u$ to $t$, so
  
  $opt(i, v) = c(v, u) + opt(i - 1, u)$.

Therefore, $opt(i, v) = \min_{(v, u) \in E} \{opt(i - 1, v), c(v, u) + opt(i - 1, u)\}$. 
The Optimality Recurrence

Let $P$ be a minimum-cost path from $v$ to $t$ using at most $i$ edges.

Observe that

- If $P$ has length less than $i$, then $opt(i, v) = opt(i - 1, v)$.
- If $P$ has length $i$, $P$ consists of some edge $(v, u)$ and a path of length $i - 1$ from $u$ to $t$, so
The Optimality Recurrence

Let $P$ be a minimum-cost path from $v$ to $t$ using at most $i$ edges.

Observe that

- If $P$ has length less than $i$, then $opt(i, v) = opt(i - 1, v)$.
- If $P$ has length $i$, $P$ consists of some edge $(v, u)$ and a path of length $i - 1$ from $u$ to $t$, so
  \[ opt(i, v) = c(v, u) + opt(i - 1, u). \]
The Optimality Recurrence

Let $P$ be a minimum-cost path from $v$ to $t$ using at most $i$ edges.

Observe that

- If $P$ has length less than $i$, then $opt(i, v) = opt(i - 1, v)$.
- If $P$ has length $i$, $P$ consists of some edge $(v, u)$ and a path of length $i - 1$ from $u$ to $t$, so
  \[
  opt(i, v) = c(v, u) + opt(i - 1, u).
  \]
- Therefore,
  \[
  opt(i, v) = \min_{(v, u) \in E} \{ opt(i - 1, v), c(v, u) + opt(i - 1, u) \}.
  \]
**Complexity Analysis**

**Observation:** \( opt(n - 1, s) \) is the cost of the optimal path from \( s \) to \( t \).

[That is: \( opt(k, -) \) needn’t be computed for any \( k \geq n \)]
Complexity Analysis

Observation: \( opt(n - 1, s) \) is the cost of the optimal path from \( s \) to \( t \).

[That is: \( opt(k, -) \) needn’t be computed for any \( k \geq n \)]

Proof: Every cycle cost is at least 0; removing cycles from a path doesn’t increase cost.
Complexity Analysis

Observation: $opt(n - 1, s)$ is the cost of the optimal path from $s$ to $t$. 

[That is: $opt(k, -)$ needn’t be computed for any $k \geq n$]

Proof: Every cycle cost is at least 0; removing cycles from a path doesn’t increase cost.

Space: $Opt[-, -]$ table takes $O(n^2)$ space.
Complexity Analysis

**Observation:** $opt(n - 1, s)$ is the cost of the optimal path from $s$ to $t$.

[That is: $opt(k, -)$ needn’t be computed for any $k \geq n$]

**Proof:** Every cycle cost is at least 0; removing cycles from a path doesn’t increase cost.

**Space:** $Opt[\_, \_]$ table takes $O(n^2)$ space.

**Time:** An entry of $Opt[\_, \_]$ might take $O(n)$ time to compute, for total of $O(n^3)$
Better Time Complexity Analysis

Let’s count table accesses in construction of $opt[-, -]$
Better Time Complexity Analysis

Let’s count table accesses in construction of $opt[-,-]$

- $opt[i,v]$ considers each neighbor of $v$, so it makes $outDeg(v)$ table accesses of $opt[-,-]$. 
Better Time Complexity Analysis

Let’s count table accesses in construction of $opt[-, -]$

- $opt[i, v]$ considers each neighbor of $v$, so it makes $outDeg(v)$ table accesses of $opt[-, -]$.
- Filling in $Opt[i, -]$ requires $\sum_{v \in V - \{t\}} outDeg(v)$ accesses.
Better Time Complexity Analysis

Let’s count table accesses in construction of $opt[−, −]$

- $opt[i, v]$ considers each neighbor of $v$, so it makes $outDeg(v)$ table accesses of $opt[−, −]$.
- Filling in $Opt[i, −]$ requires $\sum_{v \in V - \{t\}} outDeg(v)$ accesses.
- This sum is at most $m$, since each edge is used at most once.
Better Time Complexity Analysis

Let’s count table accesses in construction of $opt[−, −]$

- $opt[i, v]$ considers each neighbor of $v$, so it makes $outDeg(v)$ table accesses of $opt[−, −]$.
- Filling in $Opt[i, −]$ requires $\sum_{v \in V − \{t\}} outDeg(v)$ accesses.
- This sum is at most $m$, since each edge is used at most once.
- There are $n$ rows to the table, so total time is $O(mn)$
Better Time Complexity Analysis

Let’s count table accesses in construction of $opt[−, −]$

- $opt[i, v]$ considers each neighbor of $v$, so it makes $outDeg(v)$ table accesses of $opt[−, −]$.
- Filling in $Opt[i, −]$ requires $\sum_{v \in V − \{t\}} outDeg(v)$ accesses.
- This sum is at most $m$, since each edge is used at most once.
- There are $n$ rows to the table, so total time is $O(mn)$
- Actual path can be extracted from table in $O(m)$ time or built into table.
Improving Memory Requirements

Observation: \( \text{opt}[i, -] \) depends only on \( \text{opt}[i - 1, -] \)
Improving Memory Requirements

Observation: $opt[i, -]$ depends only on $opt[i - 1, -]$

- Use a 1-dim array $opt[]$, initialized to $opt[1, -]$, and a temporary array $hold[]$. How can we extract path now?
Observation: $opt[i, -] \text{ depends only on } opt[i - 1, -]$

- Use a 1-dim array $opt[]$, initialized to $opt[1, -]$, and a temporary array $hold[]$.
- Set $hold[v] \leftarrow \min_{(v, u) \in E} \{opt[v], c(v, u) + opt[u]\}$

How can we extract path now?
Observation: $opt[i, -]$ depends only on $opt[i - 1, -]$

- Use a 1-dim array $opt[]$, initialized to $opt[1, -]$, and a temporary array $hold[]$.
- Set $hold[v] \leftarrow \min_{(v, u) \in E} \{opt[v], c(v, u) + opt[u]\}$
- Then set $opt[] \leftarrow hold[]$; repeat $n - 3$ more times

This gives $O(n)$ space complexity beyond the storing of the graph.

How can we extract path now?
Improving Memory Requirements

Observation: \( opt[i, -] \) depends only on \( opt[i - 1, -] \)

- Use a 1-dim array \( opt[] \), initialized to \( opt[1, -] \), and a temporary array \( hold[] \).
- Set \( hold[v] \leftarrow \min_{(v, u) \in E} \{opt[v], c(v, u) + opt[u]\} \)
- Then set \( opt[] \leftarrow hold[] \); repeat \( n - 3 \) more times
- This gives \( O(n) \) space complexity beyond the storing of the graph.
Improving Memory Requirements

**Observation:** $opt[i, -]$ depends only on $opt[i - 1, -]$

- Use a 1-dim array $opt[]$, initialized to $opt[1, -]$, and a temporary array $hold[]$.
- Set $hold[v] \leftarrow \min_{(v,u) \in E} \{opt[v], c(v,u) + opt[u]\}$
- Then set $opt[] \leftarrow hold[]$; repeat $n - 3$ more times
- This gives $O(n)$ space complexity beyond the storing of the graph.

How can we extract path now?
Storing the Paths

**Idea:** Add an array $next[v]$ holding vertex after $v$ on the current shortest path from $v$ to $t$.

- $next[v]$ is initialized to $null$ for all $v$
- If $opt[v]$ changes, update $next[v]$ to hold the next vertex on the new (shorter) path from $v$ to $t$.
- Let $T$ be the graph containing all edges $(v, next[v])$. $T$ is dynamically changing.
- Claim: $T$ is a tree throughout process.
- After $i^{th}$ iteration, $T$ contains shortest $v - t$ paths of length at most $i$. 
Proof that $T$ (ignoring edge directions) is a tree

First show that $|V(T)| - 1 = |E(T)|$. 

Proof that $T$ (ignoring edge directions) is a tree

First show that $|V(T)| - 1 = |E(T)|$.

- Base Case: $T$ begins by containing $\{t\}$ and no edges.
Proof that $T$ (ignoring edge directions) is a tree

First show that $|V(T)| - 1 = |E(T)|$.

- Base Case: $T$ begins by containing $\{t\}$ and no edges
- Consider a point at which $opt[v]$ is being changed

Finally, observe that for every $v \in T$ there is a path from $v$ to $t$, so $T$ (undirected) is connected.
Proof that $T$ (ignoring edge directions) is a tree

First show that $|V(T)| - 1 = |E(T)|$.

- **Base Case:** $T$ begins by containing $\{t\}$ and no edges
- **Consider a point at which $opt[v]$ is being changed**
- **Then $opt[v] > c(v, u) + opt[u]$ for some neighbor $u$ of $v$**
Proof that $T$ (ignoring edge directions) is a tree

First show that $|V(T)| - 1 = |E(T)|$.

- Base Case: $T$ begins by containing $\{t\}$ and no edges
- Consider a point at which $\text{opt}[v]$ is being changed
- Then $\text{opt}[v] > c(v, u) + \text{opt}[u]$ for some neighbor $u$ of $v$
- So $u$ is in $T$ (since $\text{opt}[u] \neq \infty$)
Proof that \( T \) (ignoring edge directions) is a tree

First show that \(|V(T)| - 1 = |E(T)|\).

- Base Case: \( T \) begins by containing \( \{t\} \) and no edges
- Consider a point at which \( \text{opt}[v] \) is being changed
- Then \( \text{opt}[v] > c(v, u) + \text{opt}[u] \) for some neighbor \( u \) of \( v \)
- So \( u \) is in \( T \) (since \( \text{opt}[u] \neq \infty \))
- If \( v \) is in \( T \), then \( \text{next}[v] = w \neq \text{null} \) so \( (v, w) \in T \) is replaced by \( (v, u) \in T \)
Proof that $T$ (ignoring edge directions) is a tree

First show that $|V(T)| - 1 = |E(T)|$.

- Base Case: $T$ begins by containing $\{t\}$ and no edges
- Consider a point at which $opt[v]$ is being changed
- Then $opt[v] > c(v, u) + opt[u]$ for some neighbor $u$ of $v$
- So $u$ is in $T$ (since $opt[u] \neq \infty$)
- If $v$ is in $T$, then $next[v] = w \neq \text{null}$ so $(v, w) \in T$ is replaced by $(v, u) \in T$
- If $v$ is not in $T$, then we are adding a new vertex and a new edge to $T$
Proof that $T$ (ignoring edge directions) is a tree

First show that $|V(T)| - 1 = |E(T)|$.

- **Base Case:** $T$ begins by containing $\{t\}$ and no edges
- **Consider a point at which** $\text{opt}[v]$ **is being changed**
- **Then** $\text{opt}[v] > c(v, u) + \text{opt}[u]$ **for some neighbor** $u$ **of** $v$
- **So** $u$ **is in** $T$ **(since** $\text{opt}[u] \neq \infty$)**
- **If** $v$ **is in** $T$, **then** $\text{next}[v] = w \neq \text{null}$ **so** $(v, w) \in T$ **is replaced by** $(v, u) \in T$
- **If** $v$ **is not in** $T$, **then we are adding a new vertex and a new edge to** $T$

Finally, observe that for every $v \in T$ there is a path from $v$ to $t$, so $T$ (undirected) is connected.
Proof that $T$ is a tree

Now show that $T$ contains no cycles (just for fun, we already know that $T$ is a tree)
Proof that \( T \) is a tree

Now show that \( T \) contains no cycles (just for fun, we already know that \( T \) is a tree)

- Assume updating \( \text{opt}[v] \) creates a cycle in \( T \)
Proof that $T$ is a tree

Now show that $T$ contains no cycles (just for fun, we already know that $T$ is a tree)

- Assume updating $\text{opt}[\nu]$ creates a cycle in $T$
- Then the cycle looks like $\nu = \nu_0, \nu_1, \ldots, \nu_n = \nu$, where each $\nu_{i+1}(i < n) = \text{next}[\nu_i]$
Proof that $T$ is a tree

Now show that $T$ contains no cycles (just for fun, we already know that $T$ is a tree)

- Assume updating $\text{opt}[v]$ creates a cycle in $T$
- Then the cycle looks like $v = v_0, v_1, \ldots, v_n = v$, where each $v_{i+1} (i < n) = \text{next}[v_i]$
- By definition of $\text{next}[]$, $\text{opt}[v_0] > c(v_0, v_1) + \text{opt}[v_1]$

⇒ \leftarrow
Proof that $T$ is a tree

Now show that $T$ contains no cycles (just for fun, we already know that $T$ is a tree)

- Assume updating $\text{opt}[v]$ creates a cycle in $T$
- Then the cycle looks like $v = v_0, v_1, \ldots, v_n = v$, where each $v_{i+1}(i < n) = \text{next}[v_i]$
- By definition of $\text{next}[]$, $\text{opt}[v_0] > c(v_0, v_1) + \text{opt}[v_1]$
- Also $\text{opt}[v_i] = c(v_i, v_{i+1}) + \text{opt}[v_{i+1}]$, for all $i < n$
Proof that $T$ is a tree

Now show that $T$ contains no cycles (just for fun, we already know that $T$ is a tree)

- Assume updating $opt[v]$ creates a cycle in $T$
- Then the cycle looks like $v = v_0, v_1, \ldots, v_n = v$, where each $v_{i+1}(i < n) = next[v_i]$
- By definition of $next[]$, $opt[v_0] > c(v_0, v_1) + opt[v_1]$
- Also $opt[v_i] = c(v_i, v_{i+1}) + opt[v_{i+1}]$, for all $i < n$

Thus $opt[v_0] > (\sum_{i=0}^{i=n-1} c(v_i, v_{i+1})) + opt[v_n] \quad (1)$

$= (\sum_{i=0}^{i=n} c(v_i, v_{i+1})) + opt[v_0], \text{where } v_{n+1} = v_0 \quad (2)$
Proof that $T$ is a tree

Now show that $T$ contains no cycles (just for fun, we already know that $T$ is a tree)

- Assume updating $opt[v]$ creates a cycle in $T$
- Then the cycle looks like $v = v_0, v_1, \ldots, v_n = v$, where each $v_{i+1}(i < n) = next[v_i]$
- By definition of $next[]$, $opt[v_0] > c(v_0, v_1) + opt[v_1]$
- Also $opt[v_i] = c(v_i, v_{i+1}) + opt[v_{i+1}]$, for all $i < n$

Thus $opt[v_0] > (\sum_{i=0}^{i=n-1} c(v_i, v_{i+1})) + opt[v_n]$ \hfill (1)

$= (\sum_{i=0}^{i=n} c(v_i, v_{i+1})) + opt[v_0], \text{ where } v_{n+1} = v_0$ \hfill (2)

- But this is a negative weight cycle! $\Rightarrow \Leftarrow$. 
Summary

• $\text{opt} - \text{next}$ and $\text{next} - \text{opt}$ are of size $n$ and each of $n - 2$ updates takes $O(m)$ time.

• Upon completion, $\text{next}[v]$ contains first link in a cheapest path from $v$ to $t$.

• Total space required is $O(m + n)$.

• Total time required is $O(mn)$.

• Not quite as fast as Dijkstra, but more general.
Summary

- $opt[\cdot]$ and $next[\cdot]$ are of size $n$ and each of $n - 2$ updates takes $O(m)$ time.
Summary

• $opt[\cdot]$ and $next[\cdot]$ are of size $n$ and each of $n - 2$ updates takes $O(m)$ time.

• Upon completion, $next[v]$ contains first link in a cheapest path from $v$ to $t$. 
Summary

- $opt[\cdot]$ and $next[\cdot]$ are of size $n$ and each of $n - 2$ updates takes $O(m)$ time.
- Upon completion, $next[v]$ contains first link in a cheapest path from $v$ to $t$.
- Total space required is $O(m + n)$
Summary

- $opt[−]$ and $next[−]$ are of size $n$ and each of $n − 2$ updates takes $O(m)$ time.
- Upon completion, $next[v]$ contains first link in a cheapest path from $v$ to $t$.
- Total space required is $O(m + n)$
- Total time required is $O(mn)$. 
Summary

- $\text{opt}[\cdot]$ and $\text{next}[\cdot]$ are of size $n$ and each of $n - 2$ updates takes $O(m)$ time.
- Upon completion, $\text{next}[v]$ contains first link in a cheapest path from $v$ to $t$.
- Total space required is $O(m + n)$
- Total time required is $O(mn)$.
- Not quite as fast as Dijkstra, but more general.