Bellman-Ford and Dynamic Programming

Lecture 18
Part I

No negative edges: Dijkstra
Dijkstra’s Algorithm

Initialize for each node \( v \), \( \text{dist}(s, v) = \infty \)
Initialize \( X = \emptyset \), \( \text{dist}(s, s) = 0 \)
for \( i = 1 \) to \( |V| \) do

Let \( v \) be such that \( \text{dist}(s, v) = \min_{u \in V - X} \text{dist}(s, u) \)
\( X = X \cup \{ v \} \)

for each \( u \) in \( \text{Adj}(v) \) do

\[ \text{dist}(s, u) = \min \left( \text{dist}(s, u), \text{dist}(s, v) + \ell(v, u) \right) \]

Priority Queues to maintain \( \text{dist} \) values for faster running time

1. Using heaps and standard priority queues: \( O((m + n) \log n) \)
2. Best-first-search
Dijkstra’s Algorithm using Priority Queues

\[ Q \leftarrow \text{makePQ}() \]
\[ \text{insert}(Q, (s, 0)) \]
\[ \text{for each node } u \neq s \text{ do} \]
\[ \quad \text{insert}(Q, (u, \infty)) \]
\[ X \leftarrow \emptyset \]
\[ \text{for } i = 1 \text{ to } |V| \text{ do} \]
\[ \quad (v, \text{dist}(s, v)) = \text{extractMin}(Q) \]
\[ X = X \cup \{v\} \]
\[ \text{for each } u \text{ in } \text{Adj}(v) \text{ do} \]
\[ \quad \text{decreaseKey}(Q, (u, \min(\text{dist}(s, u), \text{dist}(s, v) + \ell(v, u)))) \]

Priority Queue operations:

1. \(O(n)\) insert operations
2. \(O(n)\) extractMin operations
3. \(O(m)\) decreaseKey operations
Implementing Priority Queues via Heaps

Using Heaps
Store elements in a heap based on the key value

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

Dijkstra’s algorithm can be implemented in $O((n + m) \log n)$ time.
Implementing Priority Queues via Heaps

Using Heaps

Store elements in a heap based on the key value

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

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

1. extractMin, insert, delete, meld in $O(\log n)$ time
2. decreaseKey in $O(1)$ amortized time:
Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

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

Relaxed Heaps:

- decreaseKey in $O(1)$ worst case time but at the expense of meld (not necessary for Dijkstra's algorithm)
- Dijkstra's algorithm can be implemented in $O(n \log n + m)$ time.

Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps (European Symposium on Algorithms, September 2009!)
## Fibonacci Heaps

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

1. **extractMin, insert, delete, meld** in \( O(\log n) \) time

2. **decreaseKey** in \( O(1) \) amortized time: \( \ell \) decreaseKey operations for \( \ell \geq n \) take together \( O(\ell) \) time

3. Relaxed Heaps: **decreaseKey** in \( O(1) \) worst case time but at the expense of **meld** (not necessary for Dijkstra’s algorithm)

Dijkstra’s algorithm can be implemented in \( O(n \log n + m) \) time.
Fibonacci Heaps

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

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

Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps (European Symposium on Algorithms, September 2009!)
Key takeaways of Dijkstra

1. Non-negative edges: In order to get to \( t \), only need nodes whose shortest distance is smaller than \( t \).
Key takeaways of Dijkstra

1. Non-negative edges: In order to get to \( t \), only need nodes whose shortest distance is smaller than \( t \).
   - The intermediate set \( X \) keeps the \( i - 1 \) closest nodes.
Key takeaways of Dijkstra

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

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

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

2. How to recognize the \( i \)-th closest node?
   \[
   d'(s, u) = \min\left(d'(s, u), \ dist(s, v) + \ell(v, u)\right)
   \]
Key takeaways of Dijkstra

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

2. How to recognize the $i$-th closest node?
   $d'(s, u) = \min\left(d'(s, u), \text{dist}(s, v) + \ell(v, u)\right)$
   - $d'(s, u) \geq d(s, u)$
Key takeaways of Dijkstra

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

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

   \[ d'(s, u) = \min \left( d'(s, u), \text{dist}(s, v) + \ell(v, u) \right) \]
   - $d'(s, u) \geq d(s, u)$
   - $d'(s, v) = \min_{u \in V - X} d'(s, u)$ is the $i$-th closest node, and $d'(s, v) = d(s, v)$
Part II

Negative Edges: Bellman-Ford
What are the distances computed by Dijkstra’s algorithm?

The distance as computed by Dijkstra algorithm starting from \( s \):

- **A** \( s = 0, \ x = 5, \ y = 1, \ z = 0 \).
- **B** \( s = 0, \ x = 1, \ y = 2, \ z = 5 \).
- **C** \( s = 0, \ x = 5, \ y = 1, \ z = 2 \).
- **D** IDK.
With negative length edges, Dijkstra’s algorithm can fail.
With negative length edges, Dijkstra’s algorithm can fail.
With negative length edges, Dijkstra’s algorithm can fail
With negative length edges, Dijkstra’s algorithm can fail.

![Graph Diagram]

Dijkstra’s Algorithm and Negative Lengths
With negative length edges, Dijkstra’s algorithm can fail
With negative length edges, Dijkstra’s algorithm can fail
Dijkstra’s Algorithm and Negative Lengths

With negative length edges, Dijkstra’s algorithm can fail.

![Diagram showing the failure of Dijkstra's algorithm with negative edge lengths.](image)
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.
Anything we can learn from Dijkstra?

\[ d'(s, u) = \min \left( d'(s, u), \ dist(s, v) + \ell(v, u) \right) \]

- \( d'(s, u) \geq d(s, u) \) still true.
Anything we can learn from Dijkstra?

\[ d'(s, u) = \min\left( d'(s, u), \ \text{dist}(s, v) + \ell(v, u) \right) \]

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

if \( s = v_0 \rightarrow v_1 \rightarrow v_2 \ldots \rightarrow v_k \) is a shortest path from \( s \) to \( v_k \)

- for \( 1 \leq i < k \): \( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i \) is a shortest path from \( s \) to \( v_i \), i.e. subpath of a shortest path is still a shortest path.

- Not true: \( \text{dist}(s, v_i) \leq \text{dist}(s, v_{i+1}) \), the intermediate set is no longer \( X \); in fact, it can be anything.
Anything we can learn from Dijkstra?

\[ d'(s, u) = \min\left( d'(s, u), \ dist(s, v) + \ell(v, u) \right) \]

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

if \( s = v_0 \rightarrow v_1 \rightarrow v_2 \ldots \rightarrow v_k \) is a shortest path from \( s \) to \( v_k \)

- for \( 1 \leq i < k \): \( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i \) is a shortest path from \( s \) to \( v_i \), i.e. subpath of a shortest path is still a shortest path.

- Not true: \( dist(s, v_i) \leq dist(s, v_{i+1}) \), the intermediate set is no longer \( X \); in fact, it can be anything

Solution: Update all edges \(|V| - 1\) times!
Bellman-Ford Algorithm

\begin{algorithm}
\For{each $u \in V$}{
  $d(u) \leftarrow \infty$
}
\[d(s) \leftarrow 0\]

\For{$k = 1$ to $n-1$}{
  \For{each $v \in V$}{
    \For{each edge $(u, v) \in \text{ln}(v)$}{
      \[d(v) = \min\{d(v), d(u) + \ell(u, v)\}\]
    }
  }

\For{each $v \in V$}{
  \[\text{dist}(s, v) \leftarrow d(v)\]
}
\end{algorithm}

Running time: $O(mn)$
Part III

Bellman-Ford and DP
Compute the shortest path distance from $s$ to $t$ recursively?

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 $1 \leq i < k$:

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

Sub-problem idea: paths of fewer hops/edges.
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 $v_i$.
1. Compute the shortest path distance from \( s \) to \( t \) recursively?

2. What are the smaller sub-problems?

Lemma

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

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

Sub-problem idea: paths of fewer hops/edges
Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source $s$.
\[d(v, k)\]: shortest path length from $s$ to $v$ using at most $k$ edges.

Note:
\[d(s, v) = d(v, n-1)\].

Recursion for $d(v, k)$:
\[d(v, k) = \min \{ \min_{u \in \text{In}(V)} (d(u, k-1) + \ell(u, v)), d(v, k-1) \} \].

Base case:
\[d(s, 0) = 0\] and $d(v, 0) = \infty$ for all $v \neq s$. 
Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source $s$.

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

Note: $\text{dist}(s, v) = d(v, n - 1)$. 
Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source $s$.

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

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

Recursion for $d(v, k)$:
Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source $s$.

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

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

Recursion for $d(v, k)$:

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

Base case: $d(s, 0) = 0$ and $d(v, 0) = \infty$ for all $v \neq s$. 
Example
Bellman-Ford Algorithm

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

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

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

Running time: $O(mn)$
Space: $O(n^2)$
Space can be reduced to $O(n)$. 
Bellman-Ford Algorithm

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

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

Running time:

$$O(mn)$$

Space:

$$O(n^2)$$

Space can be reduced to

$$O(n)$$

(UIUC) CS/ECE 374 17 April 1, 2021 17 / 36
Bellman-Ford Algorithm

for each \( u \in V \) do
\[
d(u, 0) \leftarrow \infty
\]
\[
d(s, 0) \leftarrow 0
\]

for \( k = 1 \) to \( n - 1 \) do
  for each \( v \in V \) do
    \[
d(v, k) \leftarrow d(v, k - 1)
    \]
    for each edge \( (u, v) \in \text{In}(v) \) do
      \[
d(v, k) = \min\{d(v, k), d(u, k - 1) + \ell(u, v)\}
      \]
  for each \( v \in V \) do
    \[
dist(s, v) \leftarrow d(v, n - 1)
    \]

Running time: \( O(mn) \)
Bellman-Ford Algorithm

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

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

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

Running time: $O(mn)$  Space:
Bellman-Ford Algorithm

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

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

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

Running time: $O(mn)$  Space: $O(n^2)$
Bellman-Ford Algorithm

for each \( u \in V \) do
\[
d(u, 0) \leftarrow \infty
\]
\[
d(s, 0) \leftarrow 0
\]

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

for each \( v \in V \) do
  \[
dist(s, v) \leftarrow d(v, n - 1)
\]

Running time: \( O(mn) \)  Space: \( O(n^2) \)
Space can be reduced to \( O(n) \).
Bellman-Ford Algorithm

```plaintext
for each \( u \in V \) do
    \( d(u) \leftarrow \infty \)

\( d(s) \leftarrow 0 \)

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

for each \( v \in V \) do
    \( \text{dist}(s, v) \leftarrow d(v) \)

Running time: \( O(mn) \) Space: \( O(n) \)
```
Negative Length Cycles

Definition

A cycle $C$ is a negative length cycle if the sum of the edge lengths of $C$ is negative.
A cycle $C$ is a negative length cycle if the sum of the edge lengths of $C$ is negative.
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 $t$?
Given \( G = (V, E) \) with edge lengths and \( s, t \). Suppose

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

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

\(-\infty\)
Bellman-Ford: Negative Cycle Detection

Check if distances change in iteration $n$.

```plaintext
for each $u \in V$ do
    $d(u) \leftarrow \infty$
    $d(s) \leftarrow 0$

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

(* One more iteration to check if distances change *)
for each $v \in V$ do
    for each edge $(u, v) \in in(v)$ do
        if $(d(v) > d(u) + \ell(u, v))$
            Output ‘‘Negative Cycle’’

for each $v \in V$ do
    $\text{dist}(s, v) \leftarrow d(v)$
```
Negative Cycle Detection

Given directed graph $G$ with arbitrary edge lengths, does it have a negative length cycle?
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 be negative cycles not reachable from $s$.

2. Run Bellman-Ford $|V|$ times, once from each node $u$?
Negative Cycle Detection

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

2. Negative cycle detection can be done with one Bellman-Ford invocation.