Outline

Single Source Shortest Path (SSSP) Problem

Single Source Shortest Path Problem

Input: A directed graph $G = (V, E)$; an edge weight function $w : E \to R$, and a start vertex $s \in V$.
Find: for each vertex $u \in V$, $\delta(s, u) =$ the length of the shortest path from s to u, and the shortest $s \to u$ path.

Single Source Shortest Path (SSSP) Problem

Single Source Shortest Path Problem

Input: A directed graph $G = (V, E)$; an edge weight function $w : E \rightarrow R$, and a start vertex $s \in V$.
Find: for each vertex $u \in V$, $\delta(s, u) =$ the length of the shortest path from s to u, and the shortest $s \rightarrow u$ path.

There are several different versions:

- G can be directed or undirected.

Single Source Shortest Path (SSSP) Problem

Single Source Shortest Path Problem

Input: A directed graph $G = (V, E)$; an edge weight function $w : E \to R$, and a start vertex $s \in V$.
Find: for each vertex $u \in V$, $\delta(s, u) =$ the length of the shortest path from s to u, and the shortest $s \to u$ path.

There are several different versions:

- G can be directed or undirected.
- All edge weights are 1.

Single Source Shortest Path Problem

Input: A directed graph $G = (V, E)$; an edge weight function $w : E \to R$, and a start vertex $s \in V$.
Find: for each vertex $u \in V$, $\delta(s, u) =$ the length of the shortest path from s to u, and the shortest $s \to u$ path.

There are several different versions:

- G can be directed or undirected.
- All edge weights are 1.
- All edge weights are positive.

Single Source Shortest Path (SSSP) Problem

Single Source Shortest Path Problem

Input: A directed graph $G = (V, E)$; an edge weight function $w : E \to R$, and a start vertex $s \in V$.
Find: for each vertex $u \in V$, $\delta(s, u) =$ the length of the shortest path from s to u, and the shortest $s \to u$ path.

There are several different versions:

- G can be directed or undirected.
- All edge weights are 1.
- All edge weights are positive.
- Edge weights can be positive or negative, but there are no cycles with negative total weight.

SSSP Problem

- Note 1: There are natural applications where edge weights can be negative.

SSSP Problem

- Note 1: There are natural applications where edge weights can be negative.
- Note 2: If G has a cycle C with negative total weight, then we can just go around C to decrease the $\delta(s, *)$ indefinitely.

- Note 1: There are natural applications where edge weights can be negative.
- Note 2: If G has a cycle C with negative total weight, then we can just go around C to decrease the $\delta(s, *)$ indefinitely.
- Then the problem is not well defined. We will need an algorithm to detect such condition.

SSSP Problem

- Note 1: There are natural applications where edge weights can be negative.
- Note 2: If G has a cycle C with negative total weight, then we can just go around C to decrease the $\delta(s, *)$ indefinitely.
- Then the problem is not well defined. We will need an algorithm to detect such condition.
- For the case when $w(e) = 1$ for all edges, we have shown that the problem can be solved by BFS in $\Theta(n + m)$ time.

SSSP Problem

- Note 1: There are natural applications where edge weights can be negative.
- Note 2: If G has a cycle C with negative total weight, then we can just go around C to decrease the $\delta(s,*)$ indefinitely.
- Then the problem is not well defined. We will need an algorithm to detect such condition.
- For the case when $w(e) = 1$ for all edges, we have shown that the problem can be solved by BFS in $\Theta(n+m)$ time.
- We next discuss algorithms for more general cases.

SSSP Problem: Positive Edge Weight

We consider the case where G is directed and $w(e) \geq 0$ for all $e \in E$. If G is undirected, the algorithm is almost identical.

SSSP Problem: Positive Edge Weight

We consider the case where G is directed and $w(e) \geq 0$ for all $e \in E$. If G is undirected, the algorithm is almost identical.

General Description:

- Each vertex $u \in V$ has a variable $d[u]$, which is an upper bound of $\delta(s, u)$.

SSSP Problem: Positive Edge Weight

We consider the case where G is directed and $w(e) \geq 0$ for all $e \in E$. If G is undirected, the algorithm is almost identical.

General Description:

- Each vertex $u \in V$ has a variable $d[u]$, which is an upper bound of $\delta(s, u)$.
- During the execution, we keep a set $S \subseteq V$.

SSSP Problem: Positive Edge Weight

We consider the case where G is directed and $w(e) \geq 0$ for all $e \in E$. If G is undirected, the algorithm is almost identical.

General Description:

- Each vertex $u \in V$ has a variable $d[u]$, which is an upper bound of $\delta(s, u)$.
- During the execution, we keep a set $S \subseteq V$.
- For each $u \in S$, $d[u] = \delta(s, u)$ has been computed. Initially S contains s only and $d[s] = \delta(s, s) = 0$.

SSSP Problem: Positive Edge Weight

We consider the case where G is directed and $w(e) \geq 0$ for all $e \in E$. If G is undirected, the algorithm is almost identical.

General Description:

- Each vertex $u \in V$ has a variable $d[u]$, which is an upper bound of $\delta(s, u)$.
- During the execution, we keep a set $S \subseteq V$.
- For each $u \in S$, $d[u] = \delta(s, u)$ has been computed. Initially S contains s only and $d[s] = \delta(s, s) = 0$.
- The vertices in $V - S$ are stored in a priority queue Q. $d[u]$ is the key value for Q.

SSSP Problem: Positive Edge Weight

We consider the case where G is directed and $w(e) \geq 0$ for all $e \in E$. If G is undirected, the algorithm is almost identical.

General Description:

- Each vertex $u \in V$ has a variable $d[u]$, which is an upper bound of $\delta(s, u)$.
- During the execution, we keep a set $S \subseteq V$.
- For each $u \in S$, $d[u] = \delta(s, u)$ has been computed. Initially S contains s only and $d[s] = \delta(s, s) = 0$.
- The vertices in $V - S$ are stored in a priority queue Q. $d[u]$ is the key value for Q.
- In each iteration, the vertex in Q with min $d[u]$ value is included into S.

SSSP Problem: Positive Edge Weight

We consider the case where G is directed and $w(e) \geq 0$ for all $e \in E$. If G is undirected, the algorithm is almost identical.

General Description:

- Each vertex $u \in V$ has a variable $d[u]$, which is an upper bound of $\delta(s, u)$.
- During the execution, we keep a set $S \subseteq V$.
- For each $u \in S$, $d[u] = \delta(s, u)$ has been computed. Initially S contains s only and $d[s] = \delta(s, s) = 0$.
- The vertices in $V - S$ are stored in a priority queue Q. $d[u]$ is the key value for Q.
- In each iteration, the vertex in Q with min $d[u]$ value is included into S.
- For vertex $v \in Q$ where $u \to v \in E$, $d[v]$ is updated.

SSSP Problem: Positive Edge Weight

We consider the case where G is directed and $w(e) \geq 0$ for all $e \in E$. If G is undirected, the algorithm is almost identical.

General Description:

- Each vertex $u \in V$ has a variable $d[u]$, which is an upper bound of $\delta(s, u)$.
- During the execution, we keep a set $S \subseteq V$.
- For each $u \in S$, $d[u] = \delta(s, u)$ has been computed. Initially S contains s only and $d[s] = \delta(s, s) = 0$.
- The vertices in $V - S$ are stored in a priority queue Q. $d[u]$ is the key value for Q.
- In each iteration, the vertex in Q with min $d[u]$ value is included into S.
- For vertex $v \in Q$ where $u \to v \in E$, $d[v]$ is updated.
- When Q is empty, the algorithm stops.

Priority Queue

To implement the algorithm, we need a data structure.

Priority Queue

A Priority Queue is a data structure Q. It consists of a set of items. Each item has a key. The data structure supports the following operations.

Priority Queue

To implement the algorithm, we need a data structure.

Priority Queue

A Priority Queue is a data structure Q. It consists of a set of items. Each item has a key. The data structure supports the following operations.

- Insert(Q, x): insert an item x into Q.
- Extract-Min(Q): remove and return the item with minimum key value.
- Min(Q): return the item with minimum key value.
- Decrease-Key(Q, x, k): decrease the key value of an item x to k.

Priority Queue

To implement the algorithm, we need a data structure.

Priority Queue

A Priority Queue is a data structure Q. It consists of a set of items. Each item has a key. The data structure supports the following operations.

- Insert(Q, x): insert an item x into Q.
- Extract-Min(Q): remove and return the item with minimum key value.
- Min(Q): return the item with minimum key value.
- Decrease-Key(Q, x, k): decrease the key value of an item x to k.

By using a Heap data structure, priority queue can be implemented so that:

- Min(Q) takes $O(1)$ time.
- All other three operations take $O(\log n)$ time (n is the number of items in Q.)

Outline

Dijkstra's Algorithm

Main Data structures:

- G: Adjacency List Representation.

Dijkstra's Algorithm

Main Data structures:

- G: Adjacency List Representation.
- For Each vertex $u \in V$:
  - Adj$[u]$: the adjacency list for u
  - $d[u]$: An upper bound for $\delta(s, u)$
  - $\pi[u]$: indicates the shortest $s \to u$ path
- S: A set that holds the finished vertices.

Dijkstra's Algorithm

Main Data structures:

- G: Adjacency List Representation.
- For Each vertex $u \in V$:
    - Adj$[u]$: the adjacency list for u
    - $d[u]$: An upper bound for $\delta(s, u)$
    - $\pi[u]$: indicates the shortest $s \to u$ path
- S: A set that holds the finished vertices.
- Q: A priority queue that holds the vertices not in S.

Dijkstra's Algorithm

Main Data structures:

- G: Adjacency List Representation.
- For Each vertex $u \in V$:
  - Adj$[u]$: the adjacency list for u
  - $d[u]$: An upper bound for $\delta(s, u)$
  - $\pi[u]$: indicates the shortest $s \rightarrow u$ path
- S: A set that holds the finished vertices.
- Q: A priority queue that holds the vertices not in S.

Initialize(G, s)

1. **for** each $u \in V$ **do**
2. $\quad d[u] = \infty$; $\pi[u] =$ NIL;
3. $d[s] = 0$

Dijkstra's Algorithm

Relax$(u, v, w(*))$

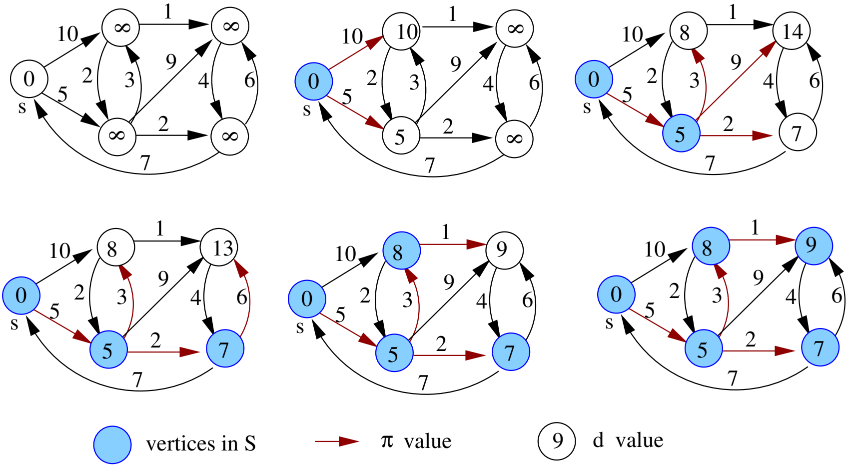
1. **if** $d[v] > d[u] + w(u \to v)$ **do**
2. $\quad d[v] = d[u] + w(u \to v)$
3. $\quad \pi[v] = u$

Dijkstra's Algorithm

Dijkstra$(G, s, w(*))$

Initialize(G, s)
$S \leftarrow \emptyset$
$Q \leftarrow V$
while $Q \neq \emptyset$ **do**
    $u \leftarrow$ Extract-Min(Q)
    $S \leftarrow S \cup \{u\}$
    for each $v \in Adj[u]$ **do**
        Relax$(u, v, w(*))$
    end for
end while

Dijkstra's Algorithm: Example



Dijkstra's Algorithm: Analysis

- **Initialize**: $\Theta(n)$
- **Relax** This is actually the decrease-key operation of the priority queue, which takes $O(\log n)$ time.
- Line 1: $\Theta(n)$
- Line 2: Initialize an empty set takes $O(1)$ time.
- Line 3: Insert n items into Q, $\Theta(n \log n)$ time.
- Line 4: While loop (not counting the time for the for loop, lines 7-9):
  - The loop iterates n times. (Q has n items in it initially. Each iteration removes one item from Q. Nothing is added into it. The loop stops when Q is empty.)
  - In the loop body, Extract-Min takes $O(\log n)$ time. The line 6 takes $O(1)$ time.
  - Thus the total run time of the while loop (not counting lines 7-9) is $O(n \log n)$.

Dijkstra's Algorithm: Analysis

The total runtime of the lines 7-9:

- Each entry in $Adj[u]$ is processed once.
- When it is processed, we call Relax once.
- Thus the processing of each entry takes $O(\log n)$ time.
- There are a total of $\Theta(m)$ entries in all $Adj[u]$'s (m is the number of edges in G).
- So the the total run time for lines 7-9 is: $O(m \log n)$ time.

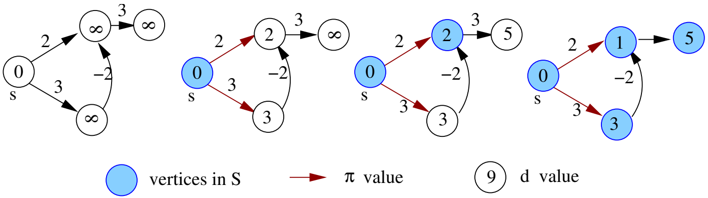
Dijkstra's Algorithm: Analysis

The total runtime of the lines 7-9:

- Each entry in $Adj[u]$ is processed once.
- When it is processed, we call Relax once.
- Thus the processing of each entry takes $O(\log n)$ time.
- There are a total of $\Theta(m)$ entries in all $Adj[u]$'s (m is the number of edges in G).
- So the the total run time for lines 7-9 is: $O(m \log n)$ time.

Since this term ($O(m \log n)$) dominates all other terms, the whole algorithm takes $O(m \log n)$ time.

If the edge weight of G can be negative, Dijkstra's algorithm doesn't work:



Outline

Bellman-Ford Algorithm

Bellman-Ford$(G, s, w(*))$

1. **Initialize**(G, s)
2. **for** $i = 1$ **to** n **do**
3.   **for** each $e = (u, v) \in E$ **do**
4.   **Relax**$(u, v, w(*))$
5. **for** each $e = (u, v) \in E$ **do**
6.   **if** $d[v] > d[u] + w(u \to v)$ **output** "G has a negative cycle"
7. $d[u]$ is the length of the shortest $s \to u$ path for each $u \in V$

Bellman-Ford Algorithm

Bellman-Ford$(G, s, w(*))$

1. **Initialize**(G, s)
2. **for** $i = 1$ **to** n **do**
3. for each $e = (u, v) \in E$ **do**
4. **Relax**$(u, v, w(*))$
5. **for** each $e = (u, v) \in E$ **do**
6. **if** $d[v] > d[u] + w(u \to v)$ **output** "G has a negative cycle"
7. $d[u]$ is the length of the shortest $s \to u$ path for each $u \in V$

Analysis:

- This time, we don't need Extract-Min operation. So we don't need priority queue anymore. **Relax** now takes $O(1)$ time.

Bellman-Ford Algorithm

Bellman-Ford$(G, s, w(*))$

1. **Initialize**(G, s)
2. **for** $i = 1$ **to** n **do**
3.   **for** each $e = (u, v) \in E$ **do**
4.   **Relax**$(u, v, w(*))$
5. **for** each $e = (u, v) \in E$ **do**
6.   **if** $d[v] > d[u] + w(u \to v)$ **output** "G has a negative cycle"
7. $d[u]$ is the length of the shortest $s \to u$ path for each $u \in V$

Analysis:

- This time, we don't need Extract-Min operation. So we don't need priority queue anymore. **Relax** now takes $O(1)$ time.
- The loop iterates $n \cdot m$ times. The loop body takes $O(1)$ time. Thus the algorithm takes $\Theta(nm)$ time.

Bellman-Ford Algorithm: Correctness Proof

Why Bellman-Ford algorithm works?

Path-Relaxation Property

Let $G = (V, E)$ be a directed graph with edge weight function $w(*)$ and the starting vertex s. Consider any shortest path $P = \langle v_0, v_1, \ldots, v_k \rangle$ from $s = v_0$ to a vertex v_k. If G is initialized by Initialize(G, s) and then a sequence of relaxation steps occurs that includes, in order, relaxations of the edges $v_0 \to v_1$, $v_1 \to v_2, \ldots, v_{k-1} \to v_k$, then $d[v_k] = \delta(s, v_k)$ after these relaxations and at all times afterward. This property holds no matter what other edge relaxations occur.

Bellman-Ford Algorithm: Correctness Proof

Why Bellman-Ford algorithm works?

Path-Relaxation Property

Let $G = (V, E)$ be a directed graph with edge weight function $w(*)$ and the starting vertex s. Consider any shortest path $P = \langle v_0, v_1, \dots, v_k \rangle$ from $s = v_0$ to a vertex v_k. If G is initialized by Initialize(G, s) and then a sequence of relaxation steps occurs that includes, in order, relaxations of the edges $v_0 \to v_1$, $v_1 \to v_2, \dots, v_{k-1} \to v_k$, then $d[v_k] = \delta(s, v_k)$ after these relaxations and at all times afterward. This property holds no matter what other edge relaxations occur.

Proof: We show by induction that after the ith edge of path P is relaxed, we have $d[v_i] = \delta(s, v_i)$.

Bellman-Ford Algorithm: Correctness Proof

Why Bellman-Ford algorithm works?

Path-Relaxation Property

Let $G = (V, E)$ be a directed graph with edge weight function $w(*)$ and the starting vertex s. Consider any shortest path $P = \langle v_0, v_1, \ldots, v_k \rangle$ from $s = v_0$ to a vertex v_k. If G is initialized by Initialize(G, s) and then a sequence of relaxation steps occurs that includes, in order, relaxations of the edges $v_0 \to v_1, v_1 \to v_2, \ldots, v_{k-1} \to v_k$, then $d[v_k] = \delta(s, v_k)$ after these relaxations and at all times afterward. This property holds no matter what other edge relaxations occur.

Proof: We show by induction that after the ith edge of path P is relaxed, we have $d[v_i] = \delta(s, v_i)$.

Base case $i = 0$: Before any edges of P have been relaxed, from the Initialization, we have $d[v_0] = d[s] = 0 = \delta(s, s)$. Because the relaxation never increases the $d[*]$ value, $d[s] = 0$ always holds. So the statement is true for the base case.

Bellman-Ford Algorithm: Correctness Proof

Proof (continued):

Induction step: Assume $d[v_{i-1}] = \delta(s, v_{i-1})$, and we examine the relaxation of the edge $v_{i-1} \to v_i$. Because P is the shortest $s \to v_i$ path, after the relaxation of the edge $v_{i-1} \to v_i$, we will have $d[v_i] = \delta(s, v_i)$. Again, because relaxation never increases $d[*]$ value, $d[v_i] = \delta(s, v_i)$ remains valid afterward.

Bellman-Ford Algorithm: Correctness Proof

Lemma 24.2

Let $G = (V, E)$ be an directed graph with edge weight function $w(*)$ and the starting vertex s. Assuming G has no negative-weight cycles. Then after $|V| - 1$ iterations of the for loop of Bellman-Ford algorithm, we have $d[v] = \delta(s, v)$ for all vertices in v that are reachable from s.

Bellman-Ford Algorithm: Correctness Proof

Lemma 24.2

Let $G = (V, E)$ be an directed graph with edge weight function $w(*)$ and the starting vertex s. Assuming G has no negative-weight cycles. Then after $|V| - 1$ iterations of the for loop of Bellman-Ford algorithm, we have $d[v] = \delta(s, v)$ for all vertices in v that are reachable from s.

Proof: Consider any vertex v that is reachable from s. Let $P = \langle v_0, v_1, \ldots, v_k \rangle$ be the shortest path from $s = v_0$ to $v = v_k$. Because G has no negative-weight cycles, P contains no cycles. Thus P has at most $|V| - 1$ edges, namely $k \leq |V| - 1$.

Bellman-Ford Algorithm: Correctness Proof

Lemma 24.2

Let $G = (V, E)$ be an directed graph with edge weight function $w(*)$ and the starting vertex s. Assuming G has no negative-weight cycles. Then after $|V| - 1$ iterations of the for loop of Bellman-Ford algorithm, we have $d[v] = \delta(s, v)$ for all vertices in v that are reachable from s.

Proof: Consider any vertex v that is reachable from s. Let $P = \langle v_0, v_1, \ldots, v_k \rangle$ be the shortest path from $s = v_0$ to $v = v_k$. Because G has no negative-weight cycles, P contains no cycles. Thus P has at most $|V| - 1$ edges, namely $k \leq |V| - 1$.

Each of the $|V| - 1$ iterations of the for loop relaxes all $|E|$ edges. Among the edges relaxed in the ith iteration is the edge $v_{i-1} \rightarrow v_i$. According to the Path-Relaxation Property, $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$.

Bellman-Ford Algorithm: Correctness Proof

Correctness of Bellman-Ford Algorithm:

- If G contains no negative cycle, then by Lemma 24.2, the algorithm computes $\delta(s, v)$ for all v reachable from s.

Correctness of Bellman-Ford Algorithm:

- If G contains no negative cycle, then by Lemma 24.2, the algorithm computes $\delta(s, v)$ for all v reachable from s.
- If G has a negative-weight cycle C that is reachable from s, then for any vertex v on C, $\delta(s, v) = -\infty$. So the condition of the **if** statement at line 6 will be true for such vertex v. The algorithm will correctly **output** "G has a negative cycle".

Outline

All Pairs Shortest Path (APSP) Problem

All Pairs Shortest Path (APSP) Problem

Input: A directed graph $G = (V, E)$ and a weight function $w : E \to R$.
Output: for each pair $u, v \in V$, find $\delta(u, v) =$ the length of the shortest path from u to v, and the shortest $u \to v$ path.

All Pairs Shortest Path (APSP) Problem

All Pairs Shortest Path (APSP) Problem

Input: A directed graph $G = (V, E)$ and a weight function $w : E \to R$.
Output: for each pair $u, v \in V$, find $\delta(u, v) =$ the length of the shortest path from u to v, and the shortest $u \to v$ path.

- If $w(e) = 1$ for all $e \in E$:
  - Call BFS n times, once for each vertex u.
  - Total runtime: $\Theta(n(n + m))$.

All Pairs Shortest Path (APSP) Problem

All Pairs Shortest Path (APSP) Problem

Input: A directed graph $G = (V, E)$ and a weight function $w : E \to R$.
Output: for each pair $u, v \in V$, find $\delta(u, v) =$ the length of the shortest path from u to v, and the shortest $u \to v$ path.

- If $w(e) = 1$ for all $e \in E$:
  - Call BFS n times, once for each vertex u.
  - Total runtime: $\Theta(n(n+m))$.
- If $w(e) \geq 0$ for all $e \in E$:
  - Call Dijkstra's algorithm n times, once for each vertex u.
  - Total runtime: $\Theta(nm \log n)$.

All Pairs Shortest Path (APSP) Problem

Input: A directed graph $G = (V, E)$ and a weight function $w : E \to R$.
Output: for each pair $u, v \in V$, find $\delta(u, v) =$ the length of the shortest path from u to v, and the shortest $u \to v$ path.

- If $w(e) = 1$ for all $e \in E$:
  - Call BFS n times, once for each vertex u.
  - Total runtime: $\Theta(n(n+m))$.
- If $w(e) \geq 0$ for all $e \in E$:
  - Call Dijkstra's algorithm n times, once for each vertex u.
  - Total runtime: $\Theta(nm \log n)$.
- If $w(e)$ can be negative:
  - Call Bellman-Ford algorithm n times, once for each vertex u.
  - Total runtime: $\Theta(n^2 m)$. Since $m = \Theta(n^2)$ in the worst case, the runtime can be $\Theta(n^4)$.

APSP Problem: Negative Edge weight

- We will try to improve the algorithm for the last case.

APSP Problem: Negative Edge weight

- We will try to improve the algorithm for the last case.
- Since we need to compute $\delta(u, v)$ for all $u, v \in V$, we will use adjacency matrix representation for G.

Let $w[1..n, 1..n]$ be a 2D array:

$$w[i,j] = w_{ij} = \begin{cases} 0 & \text{if } i = j \\ w[i,j] & \text{if } i \neq j \text{ and } i \to j \in E \\ \infty & \text{if } i \neq j \text{ and } i \to j \notin E \end{cases}$$

APSP Problem: Negative Edge weight

- We will try to improve the algorithm for the last case.
- Since we need to compute $\delta(u, v)$ for all $u, v \in V$, we will use adjacency matrix representation for G.

Let $w[1..n, 1..n]$ be a 2D array:

$$w[i,j] = w_{ij} = \begin{cases} 0 & \text{if } i = j \\ w[i,j] & \text{if } i \neq j \text{ and } i \to j \in E \\ \infty & \text{if } i \neq j \text{ and } i \to j \notin E \end{cases}$$

We want to compute an array $D[1..n, 1..n]$ such that:

$$D[i,j] = d_{ij} = \delta(i,j)$$

APSP Problem: Negative Edge weight

We assume G contains no negative cycles for now.

APSP Problem: Negative Edge weight

We assume G contains no negative cycles for now.
Define: $d_{ij}^{(t)}$ = the length of the shortest $i \to j$ path that contains at most t edges.

APSP Problem: Negative Edge weight

We assume G contains no negative cycles for now.
Define: $d_{ij}^{(t)}$ = the length of the shortest $i \to j$ path that contains at most t edges.

Then:

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

APSP Problem: Negative Edge weight

We assume G contains no negative cycles for now.
Define: $d_{ij}^{(t)}$ = the length of the shortest $i \to j$ path that contains at most t edges.

Then:

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

$$d_{ij}^{(1)} = w[i,j] = \begin{cases} 0 & \text{if } i = j \\ w[i,j] & \text{if } i \neq j \text{ and } i \to j \in E \\ \infty & \text{if } i \neq j \text{ and } i \to j \notin E \end{cases}$$

APSP Problem: Negative Edge weight

Since G contains no negative cycles, we have:

$$\delta(i,j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} \ldots$$

APSP Problem: Negative Edge weight

Since G contains no negative cycles, we have:

$$\delta(i,j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} \dots$$

This is because:

- If the shortest $i \to j$ path P contains $\geq n$ edges, it must contains a cycle C (since G has only n vertices).
- Since G has no negative cycles, we can delete C from P, without increasing the length, to get another $i \to j$ path P' with fewer edges.

APSP Problem: Negative Edge weight

Since G contains no negative cycles, we have:

$$\delta(i,j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} \ldots$$

This is because:

- If the shortest $i \to j$ path P contains $\geq n$ edges, it must contains a cycle C (since G has only n vertices).
- Since G has no negative cycles, we can delete C from P, without increasing the length, to get another $i \to j$ path P' with fewer edges.
- So the shortest path contains at most $n - 1$ edges.

APSP Problem: Negative Edge weight

- All we have to do is to compute $d_{ij}^{(n-1)}$.

APSP Problem: Negative Edge weight

- All we have to do is to compute $d_{ij}^{(n-1)}$.
- We need to find a recursive formula for $d_{ij}^{(n-1)}$:

APSP Problem: Negative Edge weight

- All we have to do is to compute $d_{ij}^{(n-1)}$.
- We need to find a recursive formula for $d_{ij}^{(n-1)}$:

$$d_{ij}^{(t)} = \min\{\underbrace{d_{ij}^{(t-1)}}_{(1)}, \underbrace{\min\{\ d_{ik}^{(t-1)} + W[k,j] \mid 1 \le k \le n\}}_{(2)}\}$$

APSP Problem: Negative Edge weight

- All we have to do is to compute $d_{ij}^{(n-1)}$.
- We need to find a recursive formula for $d_{ij}^{(n-1)}$:

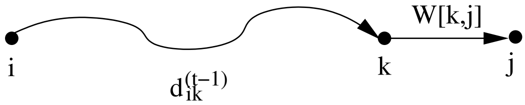
$$d_{ij}^{(t)} = \min\{\underbrace{d_{ij}^{(t-1)}}_{(1)}, \underbrace{\min\{\ d_{ik}^{(t-1)} + W[k,j] \mid 1 \le k \le n\}}_{(2)}\}$$

Case (1) The shortest $i \to j$ path actually only contains $t-1$ edges, so its length is $d_{ij}^{(t-1)}$.

Case (2) The shortest $i \to j$ path P contains t edges, Let k be the vertex on P right before reaching j.

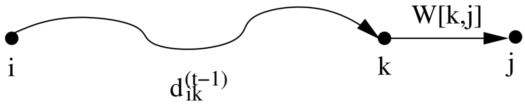
The weight of the last edge is $W[k,j]$. The portion P' of P from i to k is the shortest $i \to k$ path containing at most $t-1$ edges. The length of P' is $d_{ik}^{(t-1)}$.

(Do you realize that this is the Optical Substructure Property for this problem? We are using dynamic programming!)



APSP Problem: Negative Edge weight

Case (2) The shortest $i \to j$ path P contains t edges, Let k be the vertex on P right before reaching j.
The weight of the last edge is $W[k,j]$. The portion P' of P from i to k is the shortest $i \to k$ path containing at most $t-1$ edges. The length of P' is $d_{ik}^{(t-1)}$.
(Do you realize that this is the Optical Substructure Property for this problem? We are using dynamic programming!)



The term (1) can be re-written as $d_{ij}^{(t-1)} + 0 = d_{ij}^{(t-1)} + W[j,j]$. It can be included into the term (2). Thus:

$$d_{ij}^{(t)} = \min_{1 \le k \le n} \{ d_{ik}^{(t-1)} + W[k,j] \}$$

APSP Problem: Negative Edge weight

For $t = 1, 2, \ldots$ define:

$$D^{(t)} = (d_{ij}^{(t)})_{1 \leq i,j \leq n}$$

Then $D^{(1)} = (d_{ij}^{(1)})_{1 \leq i,j \leq n} = W[1..n, 1..n]$ = the input adjacency matrix.

APSP Problem: Negative Edge weight

For $t = 1, 2, \ldots$ define:

$$D^{(t)} = (d_{ij}^{(t)})_{1 \le i,j \le n}$$

Then $D^{(1)} = (d_{ij}^{(1)})_{1 \le i,j \le n} = W[1..n, 1..n]$ = the input adjacency matrix.

Matrix Operator $\otimes$

Let $A = (a_{ij})$ and $B = (b_{ij})$ be two $n \times n$ matrices. Define:

$$C = (c_{ij}) = A \otimes B$$

where

$$c_{ij} = \min_{1 \le k \le n} \{a_{ik} + b_{kj}\}$$

APSP Problem: Negative Edge weight

For $t = 1, 2, \ldots$ define:

$$D^{(t)} = (d_{ij}^{(t)})_{1 \le i,j \le n}$$

Then $D^{(1)} = (d_{ij}^{(1)})_{1 \le i,j \le n} = W[1..n, 1..n]$ = the input adjacency matrix.

Matrix Operator $\otimes$

Let $A = (a_{ij})$ and $B = (b_{ij})$ be two $n \times n$ matrices. Define:

$$C = (c_{ij}) = A \otimes B$$

where

$$c_{ij} = \min_{1 \le k \le n} \{a_{ik} + b_{kj}\}$$

It is easy to see:

$$D^{(t)} = D^{(t-1)} \otimes W$$

APSP Problem: Negative Edge weight

Observations

- If G has no negative cycles, then $D^{(n-1)} = D^{(n)} = D^{(n+1)} \ldots$

APSP Problem: Negative Edge weight

Observations

- If G has no negative cycles, then $D^{(n-1)} = D^{(n)} = D^{(n+1)} \dots$
- If $D^{(n-1)} = D^{(n)}$, then $D^{(n+1)} = D^{(n)} \otimes W = D^{(n-1)} \otimes W = D^{(n)}$. Thus: $D^{(n-1)} = D^{(n)} = D^{(n+1)} \dots$

APSP Problem: Negative Edge weight

Observations

- If G has no negative cycles, then $D^{(n-1)} = D^{(n)} = D^{(n+1)} \dots$
- If $D^{(n-1)} = D^{(n)}$, then $D^{(n+1)} = D^{(n)} \otimes W = D^{(n-1)} \otimes W = D^{(n)}$. Thus: $D^{(n-1)} = D^{(n)} = D^{(n+1)} \dots$
- If G has a negative cycle C, let i, j be two vertices in on C. Then $\delta(i,j) = -\infty$. Therefore, $D_{ij}^{(t)}$ will go to $-\infty$ when $t \to \infty$. Thus, in this case $D^{(n-1)} \neq D^{(n)}$.

Observations

- If G has no negative cycles, then $D^{(n-1)} = D^{(n)} = D^{(n+1)} \ldots$
- If $D^{(n-1)} = D^{(n)}$, then $D^{(n+1)} = D^{(n)} \otimes W = D^{(n-1)} \otimes W = D^{(n)}$. Thus: $D^{(n-1)} = D^{(n)} = D^{(n+1)} \ldots$
- If G has a negative cycle C, let i, j be two vertices in on C. Then $\delta(i,j) = -\infty$. Therefore, $D_{ij}^{(t)}$ will go to $-\infty$ when $t \to \infty$. Thus, in this case $D^{(n-1)} \neq D^{(n)}$.
- Hence G contains a negative cycle if and only if $D^{(n-1)} \neq D^{(n)}$.

APSP Problem: Negative Edge weight

SimpleAPSA(W) (W is the input adjacency matrix.)

1. $D^{(1)} = W$
2. **for** $t = 2$ **to** n **do**
3. compute $D^{(t)} = D^{(t-1)} \otimes W$
4. **if** $D^{(n-1)} = D^{(n)}$ **output** solution matrix $D^{(n-1)}$
5. **else output** "G contains negative cycles"

APSP Problem: Negative Edge weight

SimpleAPSA(W) (W is the input adjacency matrix.)

1. $D^{(1)} = W$
2. **for** $t = 2$ **to** n **do**
3. compute $D^{(t)} = D^{(t-1)} \otimes W$
4. **if** $D^{(n-1)} = D^{(n)}$ **output** solution matrix $D^{(n-1)}$
5. **else output** "G contains negative cycles"

Analysis:

- $\otimes$ takes $\Theta(n^3)$ time.

APSP Problem: Negative Edge weight

SimpleAPSA(W) (W is the input adjacency matrix.)

1. $D^{(1)} = W$
2. **for** $t = 2$ **to** n **do**
3. compute $D^{(t)} = D^{(t-1)} \otimes W$
4. **if** $D^{(n-1)} = D^{(n)}$ **output** solution matrix $D^{(n-1)}$
5. **else output** "G contains negative cycles"

Analysis:

- $\otimes$ takes $\Theta(n^3)$ time.
- The loop iterates n times. So total runtime is $\Theta(n^4)$.

APSP Problem: Negative Edge weight

We can do better than this. It can be shown $\otimes$ is associative. Namely:

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C$$

APSP Problem: Negative Edge weight

We can do better than this. It can be shown $\otimes$ is associative. Namely:

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C$$

So we can compute $D^{(t)}$ by repeated squaring: $D^{(2)} = D^{(1)} \otimes D^{(1)}$, $D^{(4)} = D^{(2)} \otimes D^{(2)}$, $D^{(8)} = D^{(4)} \otimes D^{(4)}$ ...

APSP Problem: Negative Edge weight

We can do better than this. It can be shown $\otimes$ is associative. Namely:

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C$$

So we can compute $D^{(t)}$ by repeated squaring: $D^{(2)} = D^{(1)} \otimes D^{(1)}$, $D^{(4)} = D^{(2)} \otimes D^{(2)}$, $D^{(8)} = D^{(4)} \otimes D^{(4)} \ldots$

FasterAPSA(W)

1. $k = \lceil \log_2(n-1) \rceil$ (k is the smallest integer such that $2^k \geq (n-1)$.)
2. Compute $D^{(2)}, D^{(4)}, D^{(8)} \ldots D^{(2^k)}, D^{(2^{k+1})}$ by repeated squaring.
3. **if** $D^{(2^k)} = D^{(2^{k+1})}$ **output** solution matrix $D^{(2^k)}$
4. **else output** "G contains negative cycles"

APSP Problem: Negative Edge weight

Analysis:

- $D^{(2^k)} = D^{(2^{k+1})}$ implies $D^{(n-1)} = D^{(n)} = \cdots = D^{(2^k)} = \cdots = D^{(2^{k+1})}$. So in this case $D^{(2^k)} = D^{(n-1)}$ is the solution matrix.

APSP Problem: Negative Edge weight

Analysis:

- $D^{(2^k)} = D^{(2^{k+1})}$ implies $D^{(n-1)} = D^{(n)} = \cdots = D^{(2^k)} = \cdots = D^{(2^{k+1})}$. So in this case $D^{(2^k)} = D^{(n-1)}$ is the solution matrix.
- $D^{(2^k)} \neq D^{(2^{k+1})}$ implies $D^{(n-1)} \neq D^{(n)}$ and hence G has negative cycles.

APSP Problem: Negative Edge weight

Analysis:

- $D^{(2^k)} = D^{(2^{k+1})}$ implies $D^{(n-1)} = D^{(n)} = \cdots = D^{(2^k)} = \cdots = D^{(2^{k+1})}$. So in this case $D^{(2^k)} = D^{(n-1)}$ is the solution matrix.
- $D^{(2^k)} \neq D^{(2^{k+1})}$ implies $D^{(n-1)} \neq D^{(n)}$ and hence G has negative cycles.
- We call $\otimes$ $k = \log_2 n$ times. So this algorithm takes $\Theta(n^3 \log n)$ time.

APSP Problem: Negative Edge weight

Analysis:

- $D^{(2^k)} = D^{(2^{k+1})}$ implies $D^{(n-1)} = D^{(n)} = \cdots = D^{(2^k)} = \cdots = D^{(2^{k+1})}$. So in this case $D^{(2^k)} = D^{(n-1)}$ is the solution matrix.
- $D^{(2^k)} \neq D^{(2^{k+1})}$ implies $D^{(n-1)} \neq D^{(n)}$ and hence G has negative cycles.
- We call $\otimes$ $k = \log_2 n$ times. So this algorithm takes $\Theta(n^3 \log n)$ time.

Can we do better?

APSP Problem: Negative Edge weight

Observations

The matrix operator $\otimes$ is very similar to the matrix multiplication operator $\times$:

- replace the scalar multiplication in MM by +
- replace the + operator in MM by min

APSP Problem: Negative Edge weight

Observations

The matrix operator $\otimes$ is very similar to the matrix multiplication operator $\times$:

- replace the scalar multiplication in MM by $+$
- replace the $+$ operator in MM by min

Question

Since $\otimes$ is so similar to the matrix multiplication, can we use Strassen's algorithm to compute $\otimes$, and improve the above algorithm?

APSP Problem: Negative Edge weight

Observations

The matrix operator $\otimes$ is very similar to the matrix multiplication operator $\times$:

- replace the scalar multiplication in MM by +
- replace the + operator in MM by min

Question

Since $\otimes$ is so similar to the matrix multiplication, can we use Strassen's algorithm to compute $\otimes$, and improve the above algorithm?

Unfortunately, NO

- The MM is defined by the scalar multiplication and $+$. However, for Strassen's algorithm to work, we need an inverse operator $-$ of $+$.

APSP Problem: Negative Edge weight

Observations

The matrix operator $\otimes$ is very similar to the matrix multiplication operator $\times$:

- replace the scalar multiplication in MM by +
- replace the + operator in MM by min

Question

Since $\otimes$ is so similar to the matrix multiplication, can we use Strassen's algorithm to compute $\otimes$, and improve the above algorithm?

Unfortunately, NO

- The MM is defined by the scalar multiplication and $+$. However, for Strassen's algorithm to work, we need an inverse operator $-$ of $+$.
- For $\otimes$, the operator that corresponds to $+$ is min. There is no inverse operator for min.

APSP Problem: Negative Edge weight

Observations

The matrix operator $\otimes$ is very similar to the matrix multiplication operator $\times$:

- replace the scalar multiplication in MM by +
- replace the + operator in MM by min

Question

Since $\otimes$ is so similar to the matrix multiplication, can we use Strassen's algorithm to compute $\otimes$, and improve the above algorithm?

Unfortunately, NO

- The MM is defined by the scalar multiplication and +. However, for Strassen's algorithm to work, we need an inverse operator $-$ of +.
- For $\otimes$, the operator that corresponds to + is min. There is no inverse operator for min.
- Strassen's algorithm does not work for $\otimes$.

Outline

APSP Problem: Floyd-Warshall Algorithm

We can improve by other ideas. We redefine:

$$d_{ij}^{(t)} = \text{the length of the shortest } i \to j \text{ path with all intermediate vertices in } \{1, 2, \ldots, t\}$$

APSP Problem: Floyd-Warshall Algorithm

We can improve by other ideas. We redefine:

$$d_{ij}^{(t)} = \text{the length of the shortest } i \to j \text{ path with all intermediate vertices in } \{1, 2, \ldots, t\}$$

Then

$$d_{ij}^{(0)} = W[i,j]$$

APSP Problem: Floyd-Warshall Algorithm

We can improve by other ideas. We redefine:

$$d_{ij}^{(t)} = \text{the length of the shortest } i \to j \text{ path with all intermediate vertices in } \{1, 2, \ldots, t\}$$

Then

$$d_{ij}^{(0)} = W[i,j]$$

- $d_{ij}^{(0)}$ is the length of the shortest $i \to j$ path with all intermediate vertices in $\{1, 2, \ldots, 0\} = \emptyset$. So such path has no any intermediate vertices, and must be the edge $i \to j$ (if it exists.)

APSP Problem: Floyd-Warshall Algorithm

We can improve by other ideas. We redefine:

$$d_{ij}^{(t)} = \begin{array}{l}\text{the length of the shortest } i \to j \text{ path with} \\ \text{all intermediate vertices in } \{1, 2, \ldots, t\}\end{array}$$

Then

$$d_{ij}^{(0)} = W[i, j]$$

- $d_{ij}^{(0)}$ is the length of the shortest $i \to j$ path with all intermediate vertices in $\{1, 2, \ldots, 0\} = \emptyset$. So such path has no any intermediate vertices, and must be the edge $i \to j$ (if it exists.)

As before, we need to derive a recursive formula.

APSP Problem: Floyd-Warshall Algorithm

$$d_{ij}^{(t)} = \min\{\underbrace{d_{ij}^{(t-1)}}_{(1)}, \underbrace{d_{it}^{(t-1)} + d_{tj}^{(t-1)}}_{(2)}\}$$

APSP Problem: Floyd-Warshall Algorithm

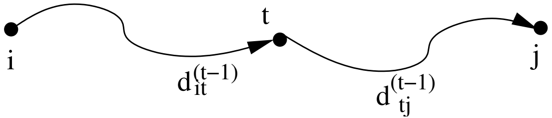
$$d_{ij}^{(t)} = \min\{\underbrace{d_{ij}^{(t-1)}}_{(1)}, \underbrace{d_{it}^{(t-1)} + d_{tj}^{(t-1)}}_{(2)}\}$$

- Case (1): The shortest $i \to j$ path P with all intermediate vertices in $\{1, 2, \ldots, t\}$ does not pass the vertex t. So all its intermediate vertices are in $\{1, 2, \ldots (t-1)\}$. Thus the length of P is $d_{ij}^{(t-1)}$

APSP Problem: Floyd-Warshall Algorithm

$$d_{ij}^{(t)} = \min\{\underbrace{d_{ij}^{(t-1)}}_{(1)}, \underbrace{d_{it}^{(t-1)} + d_{tj}^{(t-1)}}_{(2)}\}$$

- Case (1): The shortest $i \to j$ path P with all intermediate vertices in $\{1, 2, \ldots, t\}$ does not pass the vertex t. So all its intermediate vertices are in $\{1, 2, \ldots (t-1)\}$. Thus the length of P is $d_{ij}^{(t-1)}$
- Case (2): The shortest $i \to j$ path P with all intermediate vertices in $\{1, 2, \ldots, t\}$ does pass the vertex t. The first part of P is the shortest $i \to t$ path with all intermediate vertices in $\{1, 2, \ldots, (t-1)\}$. The length is $d_{it}^{(t-1)}$. Similarly the length of the second part of P is $d_{tj}^{(t-1)}$.



APSP Problem: Floyd-Warshall Algorithm

Floyd-Warshall(W)

1. $D^{(0)} = W$
2. **for** $t = 1$ to n **do**
3. Compute $D^{(t)}$ from $D^{(t-1)}$ by using the above formula.
4. **return** $D^{(n)}$

APSP Problem: Floyd-Warshall Algorithm

Floyd-Warshall(W)

1. $D^{(0)} = W$
2. **for** $t = 1$ to n **do**
3. Compute $D^{(t)}$ from $D^{(t-1)}$ by using the above formula.
4. **return** $D^{(n)}$

Analysis:

- By definition, $d_{ij}^{(n)}$ is the length of the shortest $i \to j$ path with all intermediate vertices in $\{1, 2, \ldots, n\}$. This is really not a restriction. So $d_{ij}^{(n)}$ is the length of the shortest $i \to j$ path.

APSP Problem: Floyd-Warshall Algorithm

Floyd-Warshall(W)

1. $D^{(0)} = W$
2. **for** $t = 1$ to n **do**
3. Compute $D^{(t)}$ from $D^{(t-1)}$ by using the above formula.
4. **return** $D^{(n)}$

Analysis:

- By definition, $d_{ij}^{(n)}$ is the length of the shortest $i \to j$ path with all intermediate vertices in $\{1, 2, \ldots, n\}$. This is really not a restriction. So $d_{ij}^{(n)}$ is the length of the shortest $i \to j$ path.
- Thus $D^{(n)}$ is the solution matrix.

APSP Problem: Floyd-Warshall Algorithm

Floyd-Warshall(W)

1. $D^{(0)} = W$
2. **for** $t = 1$ to n **do**
3. Compute $D^{(t)}$ from $D^{(t-1)}$ by using the above formula.
4. **return** $D^{(n)}$

Analysis:

- By definition, $d_{ij}^{(n)}$ is the length of the shortest $i \to j$ path with all intermediate vertices in $\{1, 2, \ldots, n\}$. This is really not a restriction. So $d_{ij}^{(n)}$ is the length of the shortest $i \to j$ path.
- Thus $D^{(n)}$ is the solution matrix.
- $D^{(t)}$ has n^2 entries in it. Each entry is min of two terms. So each entry of $D^{(t)}$ takes $O(1)$ time.

APSP Problem: Floyd-Warshall Algorithm

Floyd-Warshall(W)

1. $D^{(0)} = W$
2. **for** $t = 1$ to n **do**
3. Compute $D^{(t)}$ from $D^{(t-1)}$ by using the above formula.
4. **return** $D^{(n)}$

Analysis:

- By definition, $d_{ij}^{(n)}$ is the length of the shortest $i \to j$ path with all intermediate vertices in $\{1, 2, \ldots, n\}$. This is really not a restriction. So $d_{ij}^{(n)}$ is the length of the shortest $i \to j$ path.
- Thus $D^{(n)}$ is the solution matrix.
- $D^{(t)}$ has n^2 entries in it. Each entry is min of two terms. So each entry of $D^{(t)}$ takes $O(1)$ time.
- $D^{(t)}$ can be computed from $D^{(t-1)}$ in $\Theta(n^2)$ time.

APSP Problem: Floyd-Warshall Algorithm

Floyd-Warshall(W)

1. $D^{(0)} = W$
2. **for** $t = 1$ to n **do**
3. Compute $D^{(t)}$ from $D^{(t-1)}$ by using the above formula.
4. **return** $D^{(n)}$

Analysis:

- By definition, $d_{ij}^{(n)}$ is the length of the shortest $i \to j$ path with all intermediate vertices in $\{1, 2, \ldots, n\}$. This is really not a restriction. So $d_{ij}^{(n)}$ is the length of the shortest $i \to j$ path.
- Thus $D^{(n)}$ is the solution matrix.
- $D^{(t)}$ has n^2 entries in it. Each entry is min of two terms. So each entry of $D^{(t)}$ takes $O(1)$ time.
- $D^{(t)}$ can be computed from $D^{(t-1)}$ in $\Theta(n^2)$ time.
- The whole algorithm takes $\Theta(n^3)$ time.