Shortest Path*

Xiaofeng Gao

Department of Computer Science and Engineering
Shanghai Jiao Tong University, P.R.China

Algorithm Course: Shanghai Jiao Tong University

* Special thanks is given to *Prof. Kevin Wayne@Princeton*, *Prof. Charles E. Leiserson@MIT* for sharing their teaching materials, and also given to Mr. Mingding Liao from CS2013@SJTU for producing this lecture.

Outline

Outline

Paths in Graphs

Definition

Consider a digraph $G = (V, E)$ with edge-weight function $w : E \to R$, where $|V| = n$ and $|E| = m$. The weight of path $P = v_1 \to \ldots \to v_k$ is defined to be

$$w(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

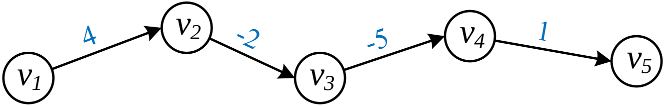
Paths in Graphs

Definition

Consider a digraph $G = (V, E)$ with edge-weight function $w : E \to R$, where $|V| = n$ and $|E| = m$. The weight of path $P = v_1 \to \ldots \to v_k$ is defined to be

$$w(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

Example: $w(P) = -2$



Shortest Path

Definition

A *shortest path* from u to v is a path of minimum weight from u to v. The *shortest path weight* from u to v is defined as

$$d(u,v) = \min\{w(P) \mid P \text{ is a path from } u \text{ to } v\}$$

Note: $d(u,v) = +\infty$ if no path from u to v exists.

Outline

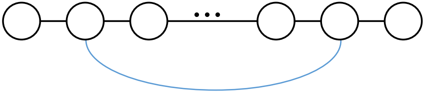
Properties of Shortest Path

Optimal Substructure. A subpath of a shortest path is a shortest path.

Properties of Shortest Path

Optimal Substructure. A subpath of a shortest path is a shortest path.

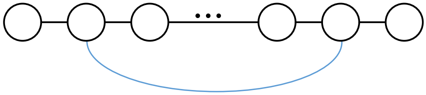
Proof. Proof by contradiction. □



Properties of Shortest Path

Optimal Substructure. A subpath of a shortest path is a shortest path.

Proof. Proof by contradiction. □

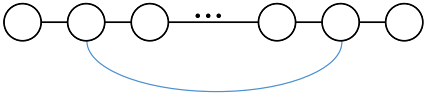


Triangle Inequality. $\forall v_1, v_2, v_3 \in V$, $d(v_1, v_2) \le d(v_1, v_3) + d(v_3, v_2)$.

Properties of Shortest Path

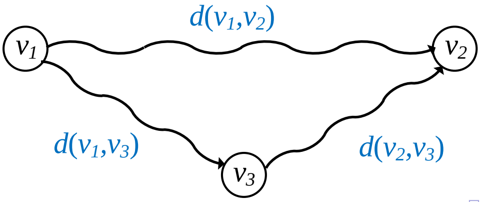
Optimal Substructure. A subpath of a shortest path is a shortest path.

Proof. Proof by contradiction. □



Triangle Inequality. $\forall v_1, v_2, v_3 \in V, d(v_1, v_2) \leq d(v_1, v_3) + d(v_3, v_2)$.

Proof. Proof by contradiction. □



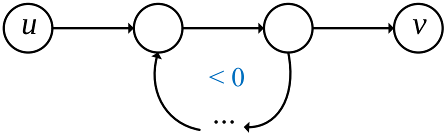
Well-Definedness of Shortest Paths

If a graph G contains a negative-weight cycle, then some shortest paths may not exist.

Well-Definedness of Shortest Paths

If a graph G contains a negative-weight cycle, then some shortest paths may not exist.

Example:



Outline

1. Introduction to Shortest Path
   - Definition
   - Property
   - Application
2. Single Source Shortest Paths
   - Problem Statement
   - Dijstra's Algorithm
   - Bellman-Ford Algorithm
3. All-Pair Shortest Paths
   - Matrix Multiplication
   - Floyd-Warshall Algorithm
   - Johnson's Algorithm

Shortest Path Applications

- PERT/CPM.
- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in LaTeX.
- Urban traffic planning.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Optimal truck routing through given traffic congestion pattern.

Ref.: Network Flows: Theory, Algorithms, and Applications, R.K. Ahuja, T.L. Magnanti, and J.B. Orlin , Prentice Hall, 1993

Outline

Single-Source Shortest Paths

Definition (Single-Source Shortest Paths Problem)

From a given source vertex $s \in V$, find the shortest-path weights $d(s, v)$ for all $v \in V$.

- If all edge weights $w(u, v)$ are nonnegative, all shortest-path weights must exist.
- If all edge weights $w(u, v)$ can be negative, the shortest-path weights may not exist because of negative circle.

Single-Source Shortest Paths

Definition (Single-Source Shortest Paths Problem)

From a given source vertex $s \in V$, find the shortest-path weights $d(s, v)$ for all $v \in V$.

- If all edge weights $w(u, v)$ are nonnegative, all shortest-path weights must exist.
- If all edge weights $w(u, v)$ can be negative, the shortest-path weights may not exist because of negative circle.
- Nonnegative weight $\Rightarrow$ Dijkstra's Algorithm
- Negative weight $\Rightarrow$ Bellman-Ford Algorithm

Outline

1. Introduction to Shortest Path
   - Definition
   - Property
   - Application
2. Single Source Shortest Paths
   - Problem Statement
   - Dijstra's Algorithm
   - Bellman-Ford Algorithm
3. All-Pair Shortest Paths
   - Matrix Multiplication
   - Floyd-Warshall Algorithm
   - Johnson's Algorithm

Dijkstra's Algorithm

IDEA: Greedy

- Maintain a set S of vertices whose shortest-path distances from s are known.
- At each step add to S the vertex $v \in V - S$ whose distance estimate from s is minimal.
- Update the distance estimates of vertices adjacent to v.

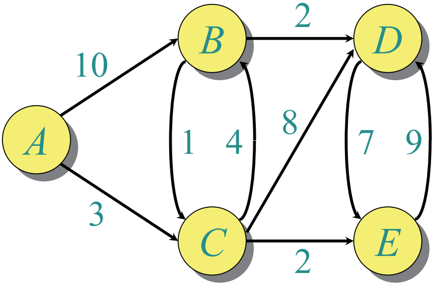
Dijkstra's Algorithm

Algorithm 1: Dijkstra's Algorithm

foreach $u \in V$ **do**
    INSERT(Q, u);
while $Q \neq \emptyset$ **do**
    $u \leftarrow$ EXTRACT-MIN(Q);
    $S \leftarrow S \cup \{u\}$;
    foreach $v \in Adj[u]$ **do**
        if $d[v] > d[u] + w(u, v)$ **then**
            $d[v] \leftarrow d[u] + w(u, v)$;   `/* Relaxation Step */`
            DECREASE-KEY(Q, v);

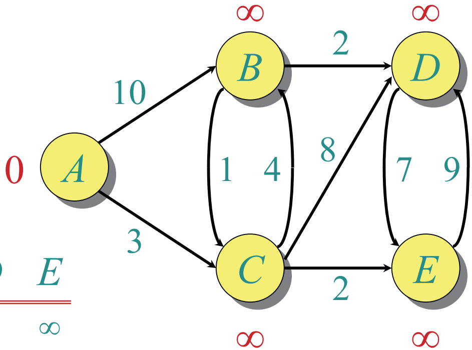
Example of Dijkstra's Algorithm

Graph with nonnegative edge weights:



Example of Dijkstra's Algorithm (cont.)

Initialize:



Q:

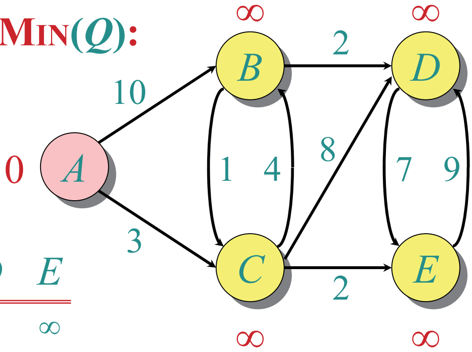
A	B	C	D	E
0	∞	∞	∞	∞

S: $\{\}$

Example of Dijkstra's Algorithm (cont.)

"A" $\leftarrow$ EXTRACT-MIN(Q):

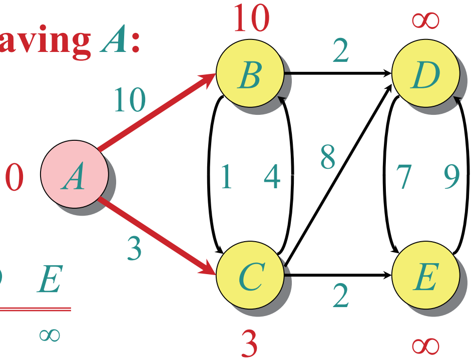
Q:	A	B	C	D	E
	0	∞	∞	∞	∞



S: { A }

Example of Dijkstra's Algorithm (cont.)

Relax all edges leaving *A*:

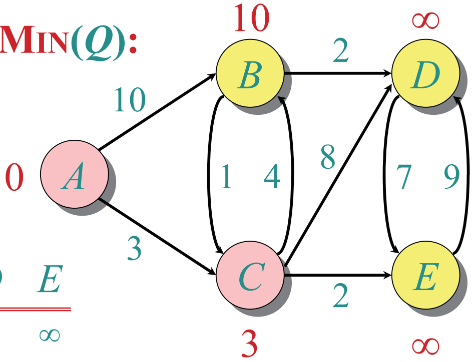


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–

S: $\{ A \}$

Example of Dijkstra's Algorithm (cont.)

"C" $\leftarrow$ EXTRACT-MIN(Q):

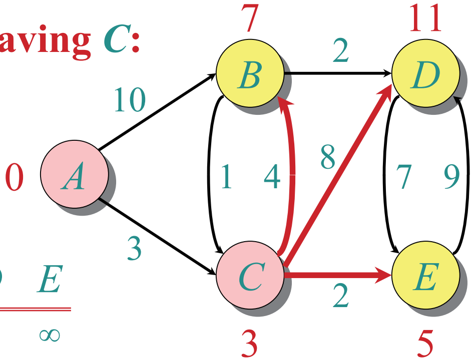


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–

S: { A, C }

Example of Dijkstra's Algorithm (cont.)

Relax all edges leaving *C*:

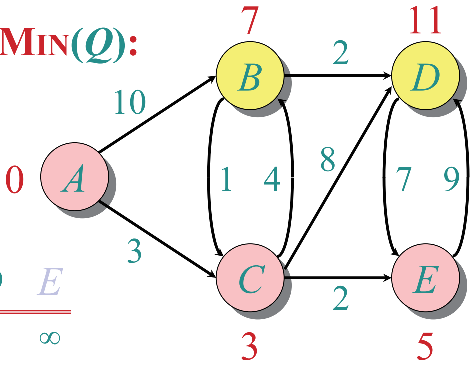


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–
		7		11	5

S: { A, C }

Example of Dijkstra's Algorithm (cont.)

"E" $\leftarrow$ EXTRACT-MIN(Q):

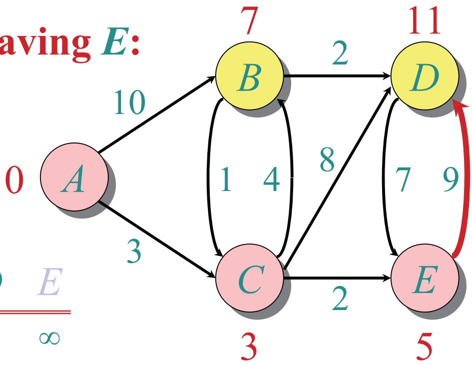


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–
		7		11	5

S: { A, C, E }

Example of Dijkstra's Algorithm (cont.)

Relax all edges leaving *E*:

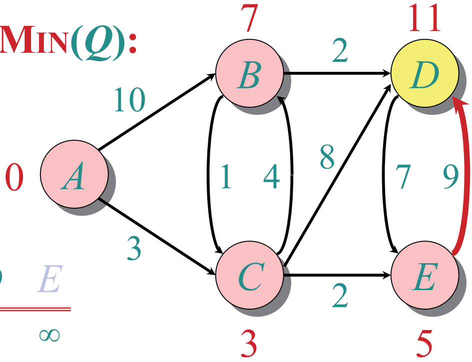


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

Example of Dijkstra's Algorithm (cont.)

"B" $\leftarrow$ EXTRACT-MIN(Q):

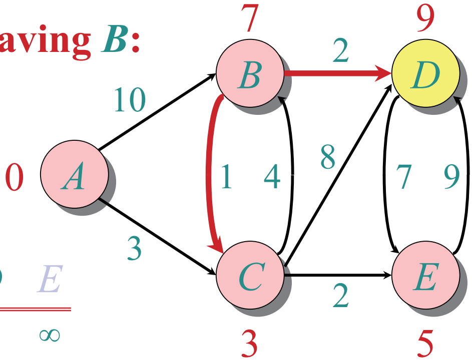


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

Example of Dijkstra's Algorithm (cont.)

Relax all edges leaving *B*:

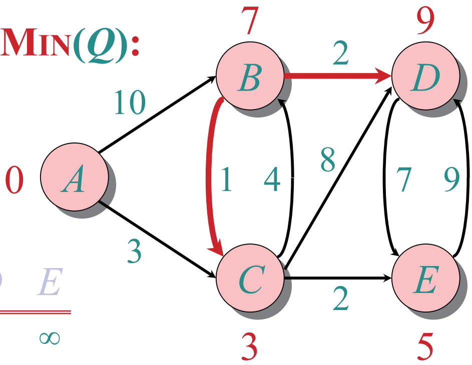


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

Example of Dijkstra's Algorithm (cont.)

"D" $\leftarrow$ EXTRACT-MIN(Q):



Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	$\boxed{5}$
		$\boxed{7}$		11	
				$\boxed{9}$	

S: { A, C, E, B, D }

Correctness of Dijkstra's Algorithm

Lemma. Initializing $d[s] \leftarrow 0$ and $d[v] \leftarrow +\infty$ for all $v \in V - \{s\}$ establishes $d[v] \geq d(s, v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps.

Correctness of Dijkstra's Algorithm

Lemma. Initializing $d[s] \leftarrow 0$ and $d[v] \leftarrow +\infty$ for all $v \in V - \{s\}$ establishes $d[v] \geq d(s,v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps.

Proof. Suppose not. Let v be the first vertex for which $d[v] < d(s,v)$, and let u be the vertex that caused $d[v]$ to change:

$$d[v] = d[u] + w(u,v).$$

Then,

$$\begin{aligned}
d[v] &< d(s,v) && \text{supposition} \\
&\leq d(s,u) + d(u,v) && \text{triangle inequality} \\
&\leq d(s,u) + w(u,v) && \text{sh. path} \leq \text{specific path} \\
&\leq d[u] + w(u,v) && v \text{ is first violation}
\end{aligned}$$

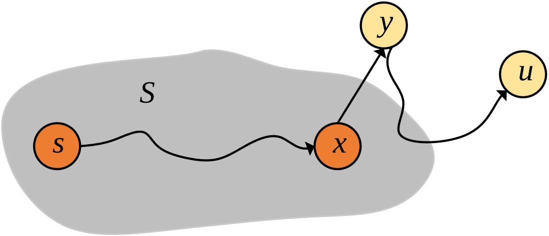
Correctness of Dijkstra's Algorithm (Cont.)

Theorem. Dijkstra's algorithm terminates with $d[v] = d(s, v)$ for all $v \in V$.

Correctness of Dijkstra's Algorithm (Cont.)

Theorem. Dijkstra's algorithm terminates with $d[v] = d(s, v)$ for all $v \in V$.

Proof. It suffices to show that $d[v] = d(s, v)$ for every $v \in V$ when v is added to S. Suppose u is the first vertex added to S for which $d[u] \neq d(s, u)$. Let y be the first vertex in $V - S$ along a shortest path from s to u, and let x be its predecessor.

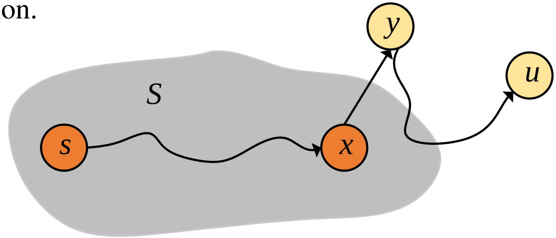


Correctness of Dijkstra's Algorithm (Cont.)

Proof. (cont.) Since u is the first vertex violating the claimed invariant, we have $d[x] = d(s,x)$. Since subpaths of shortest paths are shortest paths, it follows that $d[y]$ was set to $d(s,x) + w(x,y) = d(s,y)$ when (x,y) was relaxed just after x was added to S.

Consequently, we have $d[y] = d(s,y) \leq d(s,u) \leq d[u]$. However, $d[u] \leq d[y]$ by our choice of u in Dijkstra's Algorithm, so $d[y] = d(s,y) = d(s,u) = d[u]$.

Contradiction. □



Analysis of Dijkstra's Algorithm

Algorithm 1: Dijkstra's Algorithm

```
for u ∈ V do
    INSERT(Q, u);                                   /* |V| times */
while Q ≠ ∅ do
    u ← EXTRACT-MIN(Q) ;                            /* |V| times */
    S ← S ∪ {u};
    foreach v ∈ Adj[u] do
        if d[v] > d[u] + w(u, v) then
            d[v] ← d[u] + w(u, v);
            DECREASE-KEY(Q, v) ;                    /* degree(u) times */
```

Analysis of Dijkstra's Algorithm (Cont.)

Handshaking Lemma $\Rightarrow$ $O(E)$ implicit DECREASE-KEY.

Analysis of Dijkstra's Algorithm (Cont.)

Handshaking Lemma $\Rightarrow O(E)$ implicit DECREASE-KEY.

Performance:

- Array implementation optimal for dense graphs ($\Theta(n^2)$ edges).
- Binary heap much faster for sparse graphs ($\Theta(n)$ edges).
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but probably not worth implementing.

Implementation	EXTRACT−MIN	INSERT/ DECREASE−KEY	$\|V\| \times$ EXTRACT−MIN+ $(\|V\| + \|E\|) \times$ INS/DEC
Array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$
d-ary heap	$O(\frac{d \log \|V\|}{\log d})$	$O(\frac{\log \|V\|}{\log d})$	$O\left(\frac{(d\|V\|+\|E\|) \log \|V\|}{\log d}\right)$
Fibonacci heap	$O(\log \|V\|)^*$	$O(1)^*$	$O(\|V\| \log \|V\| + \|E\|)$

* Amortized Analysis

Unweighted Graph

Suppose $w(u, v) = 1$ for all $(u, v) \in E$. Can the code for Dijkstra be improved?

Unweighted Graph

Suppose $w(u, v) = 1$ for all $(u, v) \in E$. Can the code for Dijkstra be improved?

Use FIFO queue instead of priority queue $\Rightarrow$ breadth-first search

Time = $O(n + m)$.

Unweighted Graph

Suppose $w(u, v) = 1$ for all $(u, v) \in E$. Can the code for Dijkstra be improved?

Use FIFO queue instead of priority queue $\Rightarrow$ breadth-first search

Time = $O(n + m)$.

Correctness:

- The FIFO queue in breadth-first search mimics the priority queue in Dijkstra;
- Invariant: v comes after u in queue implies that $d[v] = d[u]$ or $d[v] = d[u] + 1$.

Outline

Shortest Paths

Definition

A *shortest path* from u to v is a path of minimum weight from u to v. The *shortest path weight* from u to v is defined as

$$d(u, v) = \min\{w(P) \mid P \text{ is the path from } u \text{ to } v\}$$

Note: negative weight is allowed.

Shortest Paths

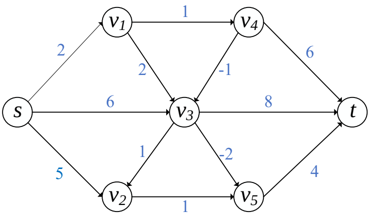
Definition

A *shortest path* from u to v is a path of minimum weight from u to v. The *shortest path weight* from u to v is defined as

$$d(u, v) = \min\{w(P) \mid P \text{ is the path from } u \text{ to } v\}$$

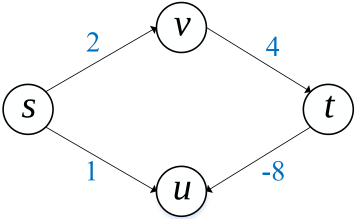
Note: negative weight is allowed.

Example: Nodes represent agents in a financial setting and $w(u, v)$ is cost of transaction in which we buy from agent u and sell to v.



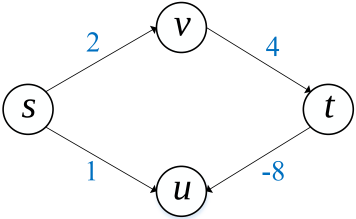
Shortest Path: Failed Attempt

Dijkstra:
Maybe fail if edge costs are negative.

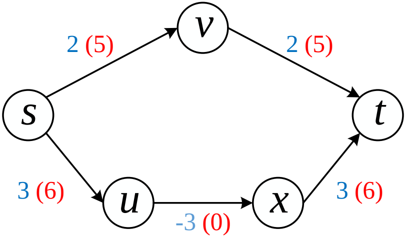


Shortest Path: Failed Attempt

Dijkstra:
Maybe fail if edge costs are negative.



Re-weighting:
Adding a constant to every edge weight can fail.



Dynamic Programming

Definition

$OPT(i, v)$ is the length of shortest s-v path P using at most i edges.

Case 1: P uses at most $i - 1$ edges.

- $OPT(i, v) = OPT(i - 1, v)$

Dynamic Programming

Definition

$OPT(i, v)$ is the length of shortest s-v path P using at most i edges.

Case 1: P uses at most $i - 1$ edges.

- ▷ $OPT(i, v) = OPT(i - 1, v)$

Case 2: P uses exactly i edges

- ▷ if (u, v) is the last edge, then OPT uses (u, v), and then selects best s-u path using at most $i - 1$ edges

Dynamic Programming

Definition

$OPT(i, v)$ is the length of shortest s-v path P using at most i edges.

Case 1: P uses at most $i-1$ edges.

- $OPT(i, v) = OPT(i-1, v)$

Case 2: P uses exactly i edges

- if (u, v) is the last edge, then OPT uses (u, v), and then selects best s-u path using at most $i-1$ edges

$$OPT(i, v) = \begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } v \neq s, i = 0 \\ \min\{OPT(i-1, v), \min\limits_{(u,v)\in E}\{OPT(i-1, u) + w(u, v)\}\} & \text{otherwise} \end{cases}$$

Shortest Paths: Implementation

Algorithm 2: Dynamic Programming

foreach *node* $u \in V$ **do**
    $M[0, u] \leftarrow \infty$;
$M[0, s] \leftarrow 0$;
for $i = 1$ *to* n **do**
    foreach *edge* $(u, v) \in E$ **do**
        $M[i, v] \leftarrow \min\{M[i-1, v], M[i-1, u] + w(u, v)\}$;

Algorithm Analysis: $O(mn)$ time, $O(n^2)$ space

Shortest Paths: Practical Improvements

Practical improvements.

- Maintain only one array $M[v]$ as shortest s-v path found so far;
- No need to check edges of the form (u, v) unless $M[u]$ changed.

Shortest Paths: Practical Improvements

Practical improvements.

- Maintain only one array $M[v]$ as shortest s-v path found so far;
- No need to check edges of the form (u, v) unless $M[u]$ changed.

Theorem. Throughout the algorithm, $M[v]$ is length of some s-v path, and after i rounds of updates, the value $M[v]$ is no larger than the length of shortest s-v path using $\leq i$ edges.

Shortest Paths: Practical Improvements

Practical improvements.

- Maintain only one array $M[v]$ as shortest s-v path found so far;
- No need to check edges of the form (u, v) unless $M[u]$ changed.

Theorem. Throughout the algorithm, $M[v]$ is length of some s-v path, and after i rounds of updates, the value $M[v]$ is no larger than the length of shortest s-v path using $\leq i$ edges.

Overall impact.

- Memory: $O(m + n)$;
- Running time: $O(mn)$ worst case, but substantially faster in practice.

Bellman-Ford: Efficient Implementation

Algorithm 3: Bellman-Ford Algorithm

```
foreach node u ∈ V do
    M[u] ← ∞;
    predecessor[u] ← ∅;
M[s] ← 0;
for i = 1 to n − 1 do
    foreach node u ∈ V do
        if M[u] has been updated in previous iteration then
            foreach edge (u, v) ∈ E do
                if M[v] > M[u] + w(u, v) then
                    M[v] ← M[u] + w(u, v);
                    predecessor[v] ← u;
    If no M[v] changed in this iteration, stop.
```

Detecting Negative Cycles

Lemma. If $OPT(n, v) = OPT(n-1, v)$ for all v, then no negative cycles.

Proof. Bellman-Ford Algorithm. □

Detecting Negative Cycles

Lemma. If $OPT(n, v) = OPT(n-1, v)$ for all v, then no negative cycles.
Proof. Bellman-Ford Algorithm. □

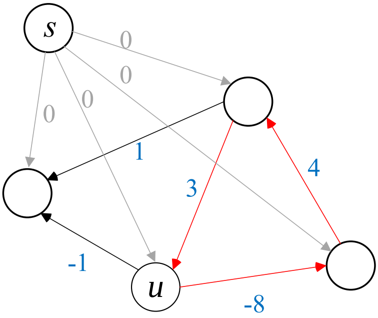
Lemma. If $OPT(n, v) < OPT(n-1, v)$ for some node u, then (any) shortest path from s to v contains a cycle W. Moreover W has negative cost.
Proof.

- $OPT(n, v) < OPT(n-1, v) \Rightarrow P$ has exactly n edges;
- By pigeonhole principle, P must contain a directed cycle W;
- Deleting W yields a s-v path with $< n$ edges $\Rightarrow W$ has negative cost. □

Detecting Negative Cycles

Theorem. Can detect negative cost cycle in $O(mn)$ time.

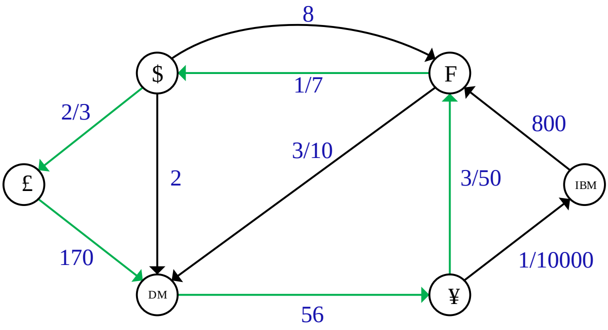
- Add new node s and connect it to all nodes with 0-cost edge.
- Check if $OPT(n, u) = OPT(n-1, u)$ for all nodes u. if no, then extract cycle from shortest path from s to u.



Detecting Negative Cycles: Application

Currency conversion. Given n currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?

Remark. Fastest algorithm very valuable!



Outline

Definition

Definition (All-Pair Shortest Paths Problem)

Given Digraph $G = (V, E)$, where $|V| = n$, with edge-weight function $w : E \to R$, find $n \times n$ matrix of shortest path lengths $d(i, j)$ for all $i, j \in V$.

Definition

Definition (All-Pair Shortest Paths Problem)

Given Digraph $G = (V, E)$, where $|V| = n$, with edge-weight function $w : E \to R$, find $n \times n$ matrix of shortest path lengths $d(i, j)$ for all $i, j \in V$.

IDEA #1:

- Run Bellman-Ford once from each vertex.
- Time $= O(n^2 m)$.
- Dense graph $\Rightarrow O(n^4)$ time.

Good first try!

Dynamic Programming

Consider the $n \times n$ adjacency matrix $A = (a_{ij})$ of the digraph, and define $d_{ij}^{(m)}$ as the weight of a shortest path from i to j that uses at most m edges.

Dynamic Programming

Consider the $n \times n$ adjacency matrix $A = (a_{ij})$ of the digraph, and define $d_{ij}^{(m)}$ as the weight of a shortest path from i to j that uses at most m edges.

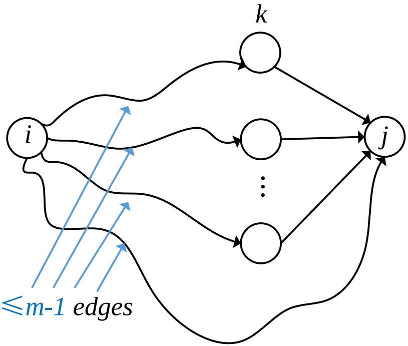
Claim: We have

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

and for $m = 1, 2, \ldots, n-1$

$$d_{ij}^{(m)} = \min_k \{d_{ik}^{(m-1)} + a_{kj}\}$$

Proof of Claim



Note: No negative-weight cycles implies

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

Matrix Multiplication

Compute $C = A \times B$, where C, A, and B are $n \times n$ matrices:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

Time = $\Theta(n^3)$ using the standard algorithm.

Matrix Multiplication

Compute $C = A \times B$, where C, A, and B are $n \times n$ matrices:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

Time = $\Theta(n^3)$ using the standard algorithm.

What if we map "$+$" $\to$ "min" and "$\times$" $\to$ "$+$"?

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

Matrix Multiplication

Compute $C = A \times B$, where C, A, and B are $n \times n$ matrices:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

Time = $\Theta(n^3)$ using the standard algorithm.

What if we map "$+$" $\to$ "min" and "$\times$" $\to$ "$+$"?

$$c_{ij} = \min_k \{a_{ik} + b_{kj}\}$$

Thus, $D^{(m)} = D^{(m-1)}$ " $\times$ " A.

Identity matrix $= I = D^{(0)} = (d_{ij}^{(0)}) = \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix}$

Matrix Multiplication (cont.)

The $(\min, +)$ multiplication is associative, and with the real numbers, it forms an algebraic structure called a closed semiring.

Matrix Multiplication (cont.)

The $(\min, +)$ multiplication is associative, and with the real numbers, it forms an algebraic structure called a closed semiring.

Consequently, we can compute

$$\begin{aligned} D^{(1)} &= D^{(0)} \times A &&= A^1 \\ D^{(2)} &= D^{(1)} \times A &&= A^2 \\ &\vdots && \vdots \\ D^{(n-1)} &= D^{(n-2)} \times A &&= A^{n-1} \end{aligned}$$

yielding $D^{(n-1)} = (d(i,j))$.

Matrix Multiplication (cont.)

The $(\min, +)$ multiplication is associative, and with the real numbers, it forms an algebraic structure called a closed semiring.

Consequently, we can compute

$$\begin{aligned} D^{(1)} &= D^{(0)} \times A &&= A^1 \\ D^{(2)} &= D^{(1)} \times A &&= A^2 \\ &\vdots && \vdots \\ D^{(n-1)} &= D^{(n-2)} \times A &&= A^{n-1} \end{aligned}$$

yielding $D^{(n-1)} = (d(i,j))$.

Time = $\Theta(n^4)$. No better than $n\times$ Bellman-Ford.

Improved Matrix Multiplication Algorithm

Repeated squaring: $A^{2k} = A^k \times A^k$.

Compute $A^2, A^4, A^8, \ldots, A^{2\lceil \log_2(n-1) \rceil}$ ($O(\log n)$ squarings).

Note: $A^{n-1} = A^n = A^{n+1} = \ldots$.

Improved Matrix Multiplication Algorithm

Repeated squaring: $A^{2k} = A^k \times A^k$.

Compute $A^2, A^4, A^8, \ldots, A^{2\lceil \log_2(n-1) \rceil}$ ($O(\log n)$ squarings).

Note: $A^{n-1} = A^n = A^{n+1} = \ldots$.

Time = $\Theta(n^3 \log n)$.

Improved Matrix Multiplication Algorithm

Repeated squaring: $A^{2k} = A^k \times A^k$.

Compute $A^2, A^4, A^8, \ldots, A^{2^{\lceil \log_2(n-1) \rceil}}$ ($O(\log n)$ squarings).

Note: $A^{n-1} = A^n = A^{n+1} = \ldots$.

Time = $\Theta(n^3 \log n)$.

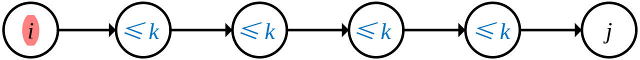
To detect negative-weight cycles, check the diagonal for negative values in $O(n)$ additional time.

Outline

Floyd-Warshall algorithm

Also dynamic programming, but faster!

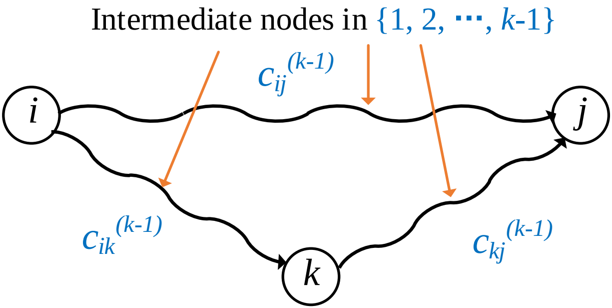
Define $c_{ij}^{(k)}$ as the weight of a shortest path from i to j with intermediate vertices belonging to the set $\{1, 2, \dots, k\}$.



Thus, $d(i,j) = c_{ij}^{(n)}$. Also, $c_{ij}^{(0)} = a_{ij}$.

Floyd-Warshall Recurrence

$$c_{ij}^{(k)} = \min_{k}\{c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\}$$



Pseudocode for Floyd-Warshall

Algorithm 4: Floyd-Warshall Algorithm

```
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            if c_ij > c_ik + c_kj then
                c_ij ← c_ik + c_kj;
```

Analysis:

- Okay to omit superscripts, since extra relaxations can't hurt.
- Runs in $\Theta(n^3)$ time.
- Simple to code and efficient in practice.

Transitive Closure of a Directed Graph

$$\text{Compute } t_{ij} = \begin{cases} 1 & \text{if there exists a path from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

Transitive Closure of a Directed Graph

Compute $t_{ij} = \begin{cases} 1 & \text{if there exists a path from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$

IDEA: Use Floyd-Warshall, but with $(\vee, \wedge)$ instead of $(\min, +)$:

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$$

Transitive Closure of a Directed Graph

Compute $t_{ij} = \begin{cases} 1 & \text{if there exists a path from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$

IDEA: Use Floyd-Warshall, but with $(\vee, \wedge)$ instead of $(\min, +)$:

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$$

Time $= \Theta(n^3)$.

Outline

1. Introduction to Shortest Path
   - Definition
   - Property
   - Application
2. Single Source Shortest Paths
   - Problem Statement
   - Dijstra's Algorithm
   - Bellman-Ford Algorithm
3. All-Pair Shortest Paths
   - Matrix Multiplication
   - Floyd-Warshall Algorithm
   - Johnson's Algorithm

Graph Reweighting

Theorem. Given a label $h(v)$ for each $v \in V$, reweight each edge $(u, v) \in E$ by $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$. Then, all paths between the same two vertices are reweighted by the same amount.

Graph Reweighting

Theorem. Given a label $h(v)$ for each $v \in V$, reweight each edge $(u, v) \in E$ by $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$. Then, all paths between the same two vertices are reweighted by the same amount.

Proof. Let $P = v_1 \to v_2 \to \ldots \to v_k$ be a path in the graph. We have:

$$\begin{aligned}
\hat{w}(P) &= \sum_{i=1}^{k-1} \hat{w}(v_i, v_{i+1}) \\
&= \sum_{i=1}^{k-1} (w(v_i, v_{i+1}) + h(v_i) - h(v_{i+1})) \\
&= \sum_{i=1}^{k-1} w(v_i, v_{i+1}) + h(v_1) - h(v_k) \\
&= w(P) + h(v_1) - h(v_k)
\end{aligned}$$

Johnson's Algorithm

① Find a vertex labeling h such that $\hat{w}(u,v) \geq 0$ for all $(u,v) \in E$ by using Bellman-Ford to solve the difference constraints

$$h(v) - h(u) \leq w(u,v)$$

or determine that a negative-weight cycle exists.

- ▷ Time = $O(mn)$

Johnson's Algorithm

① Find a vertex labeling h such that $\hat{w}(u,v) \geq 0$ for all $(u,v) \in E$ by using Bellman-Ford to solve the difference constraints

$$h(v) - h(u) \leq w(u,v)$$

or determine that a negative-weight cycle exists.

- ▷ Time = $O(mn)$

② Run Dijkstra's algorithm from each vertex using $\hat{w}$.

- ▷ Time = $O(mn + n^2 \log n)$

Johnson's Algorithm

① Find a vertex labeling h such that $\hat{w}(u,v) \geq 0$ for all $(u,v) \in E$ by using Bellman-Ford to solve the difference constraints

$$h(v) - h(u) \leq w(u,v)$$

or determine that a negative-weight cycle exists.

▷ Time = $O(mn)$

② Run Dijkstra's algorithm from each vertex using $\hat{w}$.

▷ Time = $O(mn + n^2 \log n)$

③ Reweight each shortest-path length $\hat{w}(P)$ to produce the shortest-path lengths $w(P)$ of the original graph.

▷ Time = $O(n^2)$

Total time = $O(mn + n^2 \log n)$.