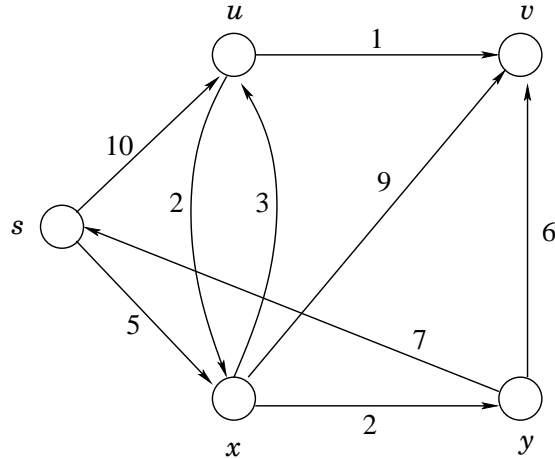

CS-230b
**Advanced Algorithms and
Applications**

Subhash Suri

**Computer Science Department
UC Santa Barbara**

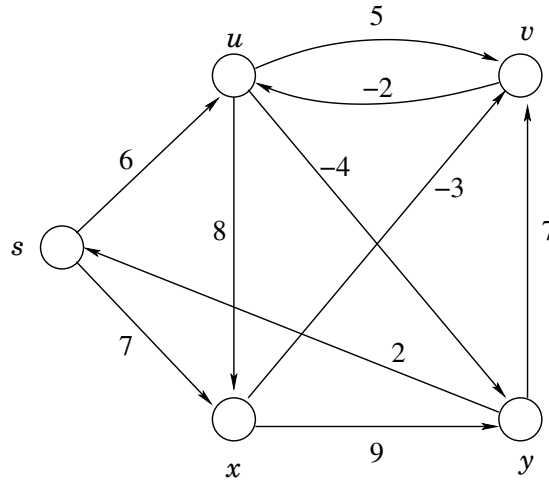
Fall Quarter 2004.

Shortest Paths



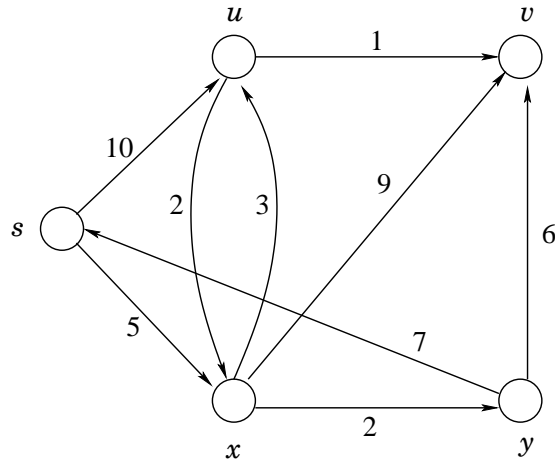
- Find shortest length path from s to v ?
- $s \rightarrow x \rightarrow u \rightarrow v$ has length $5 + 3 + 1 = 9$.
- Many applications. Discussed later.
- A network $G = (V, E)$.
- Vertices (nodes) $V = \{1, \dots, n\}$.
- Edges (links) $E = \{e_1, e_2, \dots, e_m\}$. Edge $e_{ij} = (i, j)$ is directed from i to j .
- Edge e_{ij} has cost (weight) c_{ij} . The costs can be positive or negative!

Negative Cost Shortest Paths



- What's the shortest path from s to y ?
- $s \rightarrow x \rightarrow v \rightarrow u \rightarrow y$ has length -2 .
- How can costs be negative?
- Examples later. (Arbitrage trading, scientific simulations, matching algorithms, min cost network flows).
- More general the formulation, the better.
- Any simple way to eliminate negative edges? Adding a constant to all edges?

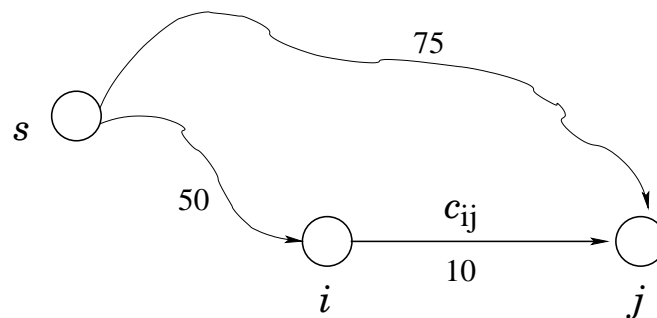
Getting Started



- Source node $s = v_0$.
- Compute SP distances from s to every node v_j .
- The paths themselves can be recovered from predecessors.
- Distance labels $d(j)$: best path length to j found so far.
- Initially, $d(0) = 0$, and $d(j) = \infty$ for others.
- The algorithm improves estimates for all $d(j)$ until SP distances become known.

Basic Idea

- How to improve the distance estimate?



- Suppose there is an edge (i, j) such that

$$d(j) > d(i) + c_{ij}$$

then we can improve the estimate of $d(j)$:

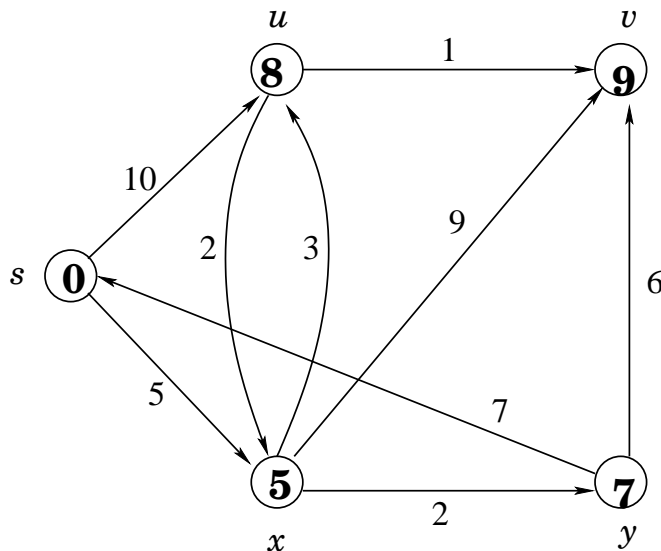
$$d(j) = d(i) + c_{ij}$$

- Previously, $d(i) = 50$ and $d(j) = 75$. The relabeling step finds a better path to j via i of cost 60.

Optimality Condition

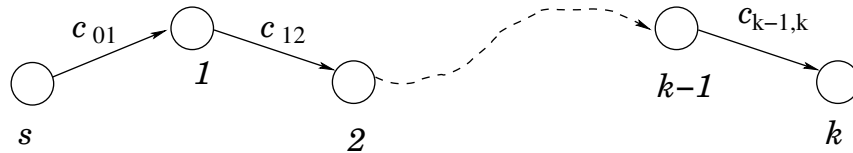
Theorem: Suppose each $d(j)$ is the length of some feasible path from s to j . Then, these $d()$ distances are shortest path distance *if and only if*

$$d(j) \leq d(i) + c_{ij}, \quad \text{for all } (i, j) \in E$$



Necessity Proof: Suppose \exists an edge (i, j) violating the condition. Then, $d(j) > d(i) + c_{ij}$. But then we can reach j via i at cost $d(i) + c_{ij}$, which is smaller than $d(j)$, contradicting the $d(j)$ is shortest path distance.

Sufficiency



1. Suppose $d(j) \leq d(i) + c_{ij}$ holds.
2. Let k be a node with incorrect distance.
3. $s \rightarrow 1 \rightarrow 2 \cdots k-1 \rightarrow k$ be actual SP.
4. So, $c_{01} + c_{12} + \cdots + c_{k-1,k} < d(k)$. (*)
5. By optimality condition we have

$$\begin{aligned}d(k) &\leq d(k-1) + c_{k-1,k} \\d(k-1) &\leq d(k-2) + c_{k-2,k-1} \\&\vdots \\d(1) &\leq d(0) + c_{01}\end{aligned}$$

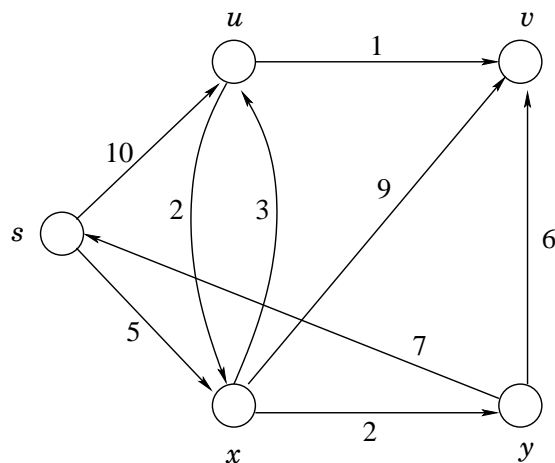
6. By summing, $d(k) \leq c_{01} + c_{12} + \cdots + c_{k-1,k}$ which contradicts (*)!!!

Generic SP Algorithm

algorithm Label-Correcting

1. $d(s) = 0$, $\text{pred}(s) = 0$;
2. $d(j) = \infty$, for $j = 1, 2, \dots, n$.
3. **while** $\exists(i, j)$ **with** $d(j) > d(i) + c_{ij}$ **do**
4. $d(j) = d(i) + c_{ij}$;
5. $\text{pred}(j) = i$;
6. **end**;

- Run the algorithm on example.



Analysis

- By optimality theorem, the output is correct IF the algorithm halts.
- If costs are integers, each distance update changes the value by ≥ 1 .
- If the largest edge cost is $|C|$, then SP length cannot drop below $-nC$. The largest feasible path length is $+nC$.
- Total number of relabeling is at most n^2C .
- So, the algorithm halts, although C can be very large. Algorithm not strongly polynomial.
- Alternatively, one can bound the run time by $O(2^n)$.

Self-Study

- Construct bad examples for generic labeling algorithm.
- Show termination even when costs are non-integer.
- Prove upper bound of $O(2^n)$.

Improved Label-Correcting

- The Generic Label-Correcting does not specify any order for selecting edge violations.
- To improve running time, we need a better order.
- Many variants of this algorithm, with different complexity and performance.
- Bellman-Ford Method: elegant, admits simple analysis, works very well in practice, and has the best theoretical running time.

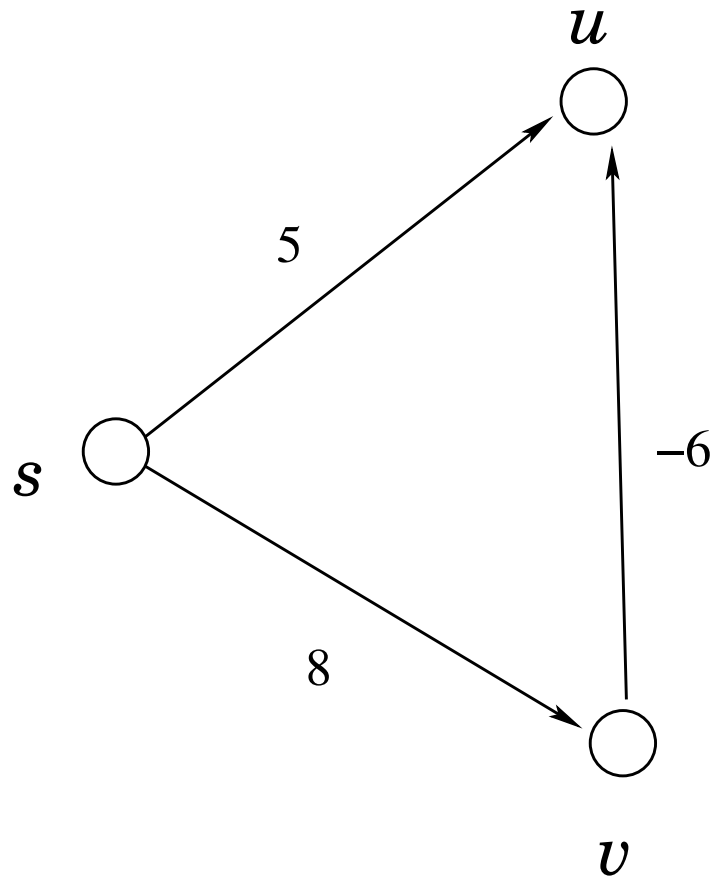
Bellman-Ford Algorithm

algorithm Bellman-Ford

Input: Graph $G = (V, E)$, with source node s .

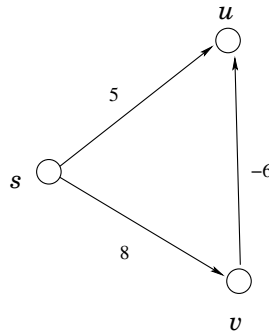
1. $d(s) = 0$, $\text{pred}(s) = 0$;
2. $d(j) = \infty$, for $j = 1, 2, \dots, n$.
3. for $k = 1$ to $|V| - 1$ do
4. for each edge (i, j) in E do
5. if $d(j) > d(i) + c_{ij}$;
6. $d(j) = d(i) + c_{ij}$;
7. $\text{pred}(j) = i$;
8. end;

Example



- Use edge order $(v, u), (s, u), (s, v)$.

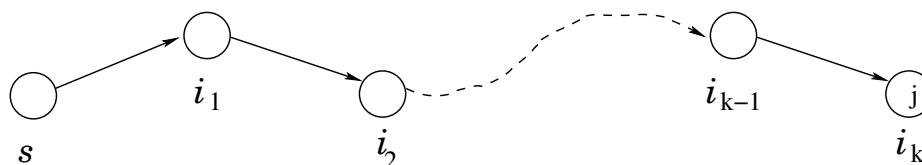
Bellman-Ford: Correctness



1. The algorithm terminates after $O(nm)$ steps, by construction. (Key: SP has $\leq n$ edges.)
2. Show that distances computed are correct.
3. By Induction. *If the SP to a node j consists of k edges, then after k iterations of the outer loop, $d(j)$ is correct.*
4. Basis of induction: $k = 1$. If the shortest path to j has only one edge, then E must contain this edge (s, j) . During the $k = 1$ iteration, the inner loop scans this edge at some point, and updates the distance $d(j)$ correctly.

Bellman-Ford: Correctness

General Case of Induction:



- Suppose, by IH, that after $k - 1$ iterations, nodes whose SP use at most $k - 1$ edges are correctly labeled.
- Consider a node j whose SP path has exactly k edges. Let $s \rightarrow i_1 \rightarrow i_2 \cdots i_{k-1} \rightarrow i_k = j$ be the SP to j .
- By induction, $d(i_{k-1})$ is correct.
- When the edge (i_{k-1}, j) is scanned, during the k th iteration of outer loop, we update $d(j) \leftarrow d(i_{k-1}) + c_{i_{k-1},j}$.
- So, $d(j)$ is correct after k iterations.

Negative Edges and Cycles

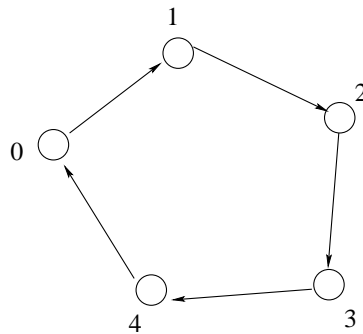
1. Effect of negative costs on Bellman-Ford?
2. The optimality condition is bullet-proof.
3. What about the termination condition?
4. Suppose G contains a cycle Z whose total cost is negative. By repeatedly going around Z we can continuously lower the path length!
5. Theorem: SP distances in G are well defined if and only if G does not contains a negative-cost cycle reachable from s . If SP are well-defined, then there is always a simple shortest path from s to each j .
6. Proof: If all reachable cycles are non-negative, then we can always cut them off.
7. Consequently, Bellman-Ford halts after $O(nm)$ steps IF there is no negative cycle reachable from s .

Detecting Negative Cycles

- How to tell if G has a negative cycle?
- [Idea 1:] We know that $-nC$ is a lower bound on any $d()$. So, if any label drops below $-nC$, we have a negative cycle. You can trace the cycle using pred pointers.
- [Idea 2:] Use the Bellman-Ford algorithm. At the end, make one more pass over all edges E . If you find ANY edge (i, j) for which $d(j) > d(i) + c_{ij}$, then G must have a negative cycle.
- Call it the Bellman-Ford Post Processing check.

Detecting Negative Cycles

Theorem: G has a negative cycle if and only if Bellman-Ford Post Processing check fails.



1. No negative cycle means all distances are correct. So, by optimality condition,
 $d(j) \leq d(i) + c_{ij}$.
2. Now, suppose there is a negative cycle.
Ex: $c_{0,1} + c_{1,2} + c_{2,3} + c_{3,4} + c_{4,0} < 0$.
3. Suppose the Post Processing check passes. Then, $d(j) \leq d(j-1) + c_{j-1,j}$, for all j .
4. Add them up, and cancel the common term. We get $c_{0,1} + c_{1,2} + c_{2,3} + c_{3,4} + c_{4,0} \geq 0$, which contradicts (2)!!!

Applications of Shortest Paths

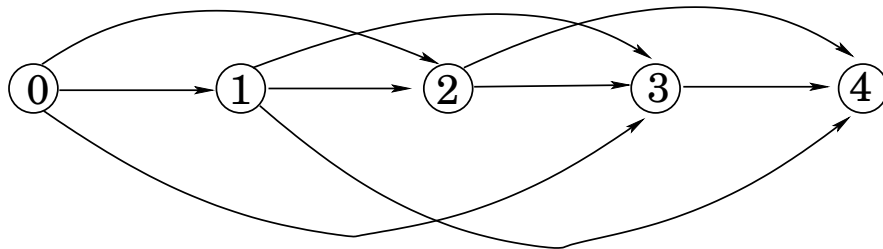
- Shortest path problems arise both as stand-alone models, and as subproblems in more complex settings.
- Obvious applications in telecom and transportation: sending messages or goods quickly and cheaply.
- Urban planning modelers, project management, inventory planning, DNA sequencing.
- Few examples of the use of shortest path useful in modeling subproblems in more complex settings:
 1. Inspection planning in a production line.
 2. Approximating piecewise linear functions.
 3. System of Difference Constraints.
 4. Others....

Production Line

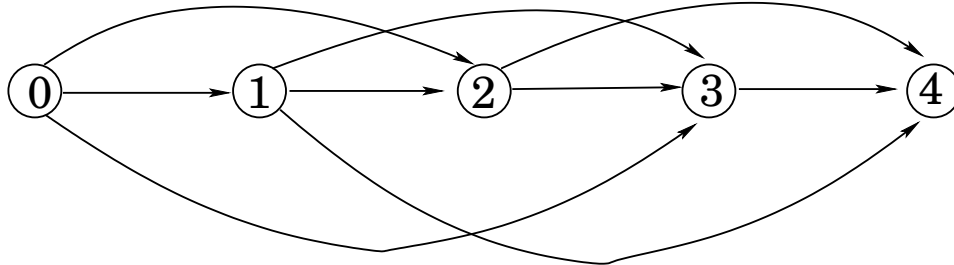
- n stations, each with manufacturing followed by possible inspection stage.
- Product enters stage 1 in B -size batches.
- Possible defects introduced at any stage. The prob. of producing a defect in stage i is α_i .
- All defects non-repairable, so defective item discarded.
- After any stage, we can either inspect all items, or none (no sampling).
- The n th stage must have inspection—can't afford to ship a defective item.
- Design an optimal inspection plan that minimizes the total cost of production and inspection.
- Fewer inspection decrease cost, but would increase production cost because defective items might continue down the production line.

Production Line

- Modeling parameters:
 1. p_i : manufac cost/item in stage i .
 2. f_{ij} : fixed inspection cost per batch after stage j , given the last inspection after stage i .
 3. g_{ij} : variable per unit cost for inspection after j , given last inspection after i .
 4. Inspection after j needs to look for defects introduced in stages $i + 1, i + 2, \dots, j$.
- Shortest path problem on nodes $0, 1, \dots, n$.
- Put an edge (i, j) between all i, j , with $i < j$.



Production Line



- Any path from node 0 to n is an inspection plan.
- E.g. path $0 \rightarrow 2 \rightarrow 4$ means inspection after stage 2 and 4.
- Cost of edge (i, j) :

$$c_{ij} = f_{ij} + B(i)g_{ij} + B(i) \sum_{k=i+1}^j p_k,$$

where $B(i) = B \prod_{k=1}^i (1 - \alpha_k)$ denotes the expected number of non-defective units at end of stage i .

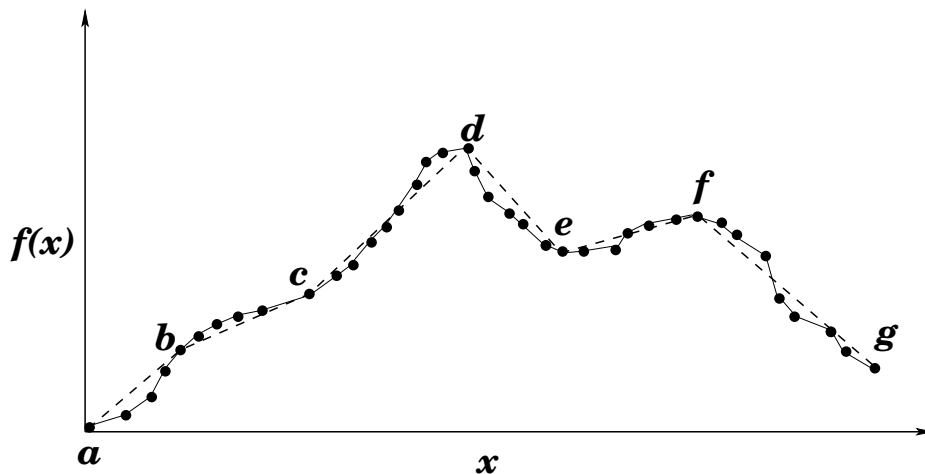
- Dijkstra's algorithm will solve the problem in $O(n^2)$ time.

Piecewise Function Approx

- Scientific data often use piecewise linear functions, contain large number of breakpoints.
- Large data difficult to store, transmit, manipulate, evaluate.
- Use a subset of data—approximation introduces inaccuracies.
- Optimal piecewise approximation? Tradeoff between cost and benefits.
- Model: $f(x)$ a piecewise linear function of scalar x .
- Think of tuples $(x_1, f(x_1)), (x_2, f(x_2)), \dots$ as points in the plane.
- Between two consecutive values, function varies linearly.

Piecewise Function Approx

- Assume n is very large, and we want to approximate f by another function z that passes through a smaller subset.

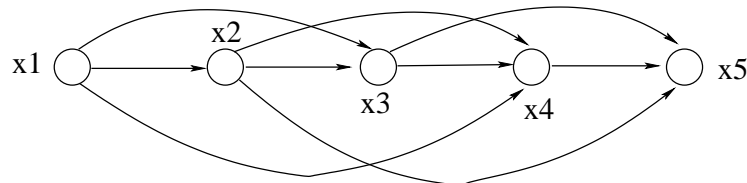


- Figure shows an example: input of 40 breakpoints (solid) approximated by 7 breakpoints (dashed).
- Fixed cost α per breakpoint in storage.
- Sum of Squared error: cost of dropping all points between x_i and x_j :

$$\beta \sum_{k=i+1}^{j-1} (f(x_k) - z(x_k))^2$$

Piecewise Function Approx

- Shortest path problem on n nodes, $1, 2, \dots, n$.



- Put an edge between each pair i, j , with $i < j$. Edge (i, j) signifies that we approximate the piecewise function between x_i and x_j by a single line segment joining x_i to x_j .
- The cost c_{ij} is

$$\alpha + \beta \sum_{k=i+1}^{j-1} (f(x_k) - z(x_k))^2$$

- Each directed path from 1 to n represents an approximating function z . The shortest path represents the optimal approximating function.

System of Difference Constraints

- **Input:** n variables $x()$, and m constraints of the form

$$x(i) - x(j) \leq b(k).$$

- Does there exist a feasible solution?
- **Example:**

$$x(3) - x(4) \leq 5$$

$$x(4) - x(1) \leq -10$$

$$x(1) - x(3) \leq 8$$

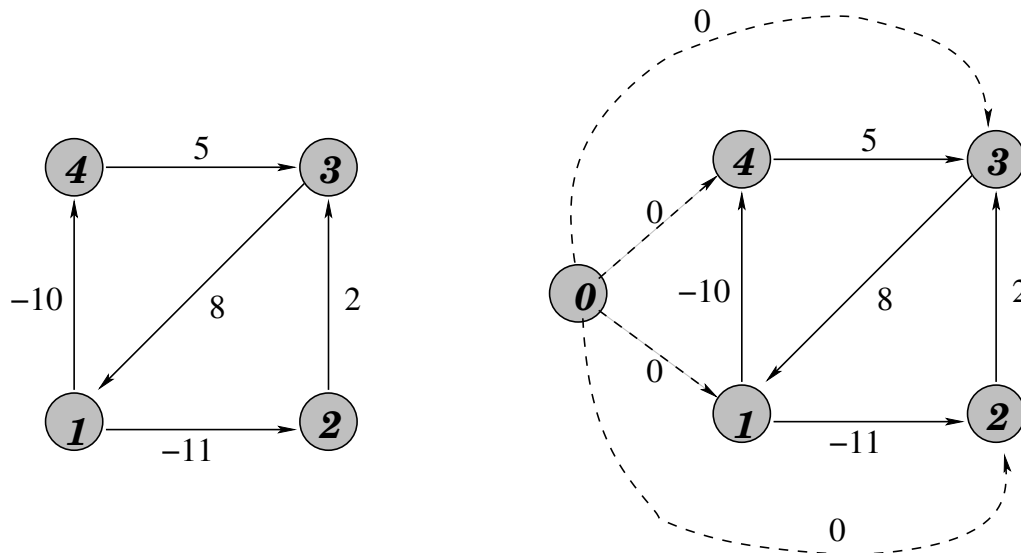
$$x(2) - x(1) \leq -11$$

$$x(3) - x(2) \leq 2$$

- Find real-valued $x()$ that satisfy all the constraints?
- **Example applications:** resource allocation, investment management.

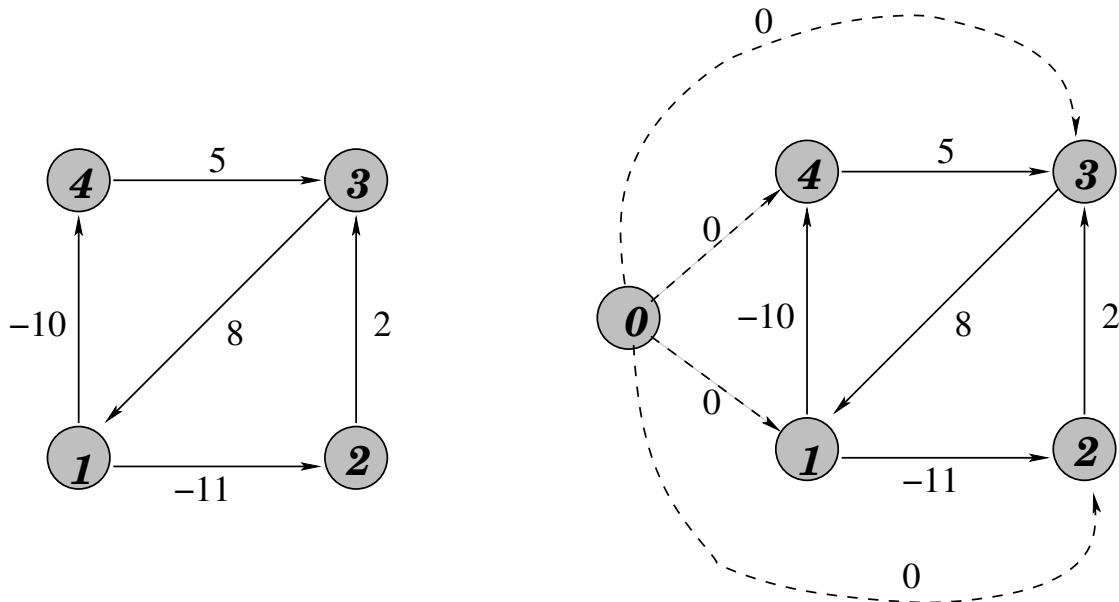
Difference Constraints (2)

- **Graph Model:** variable \equiv nodes, and constraint “ $x(i) - x(j)$ ” \equiv arc (j, i) .



- Augment the graph with node 0, with zero-cost edges to all other nodes.
- Think of $x()$ as shortest path distances.
- By SP Optimality: distance labels $d()$ are shortest path distances if and only if $d(j) - d(i) \leq c_{ij}$.
- Thus, difference constraints satisfied if and only if this graph *does not have a negative cycle* (and so shortest path distances exist!).

Difference Constraints (3)



- In this example, 1–2–3 is a negative cycle.
- Correspondingly, constraints $x(1) - x(3) \leq 8$, $x(2) - x(1) \leq -11$, and $x(3) - x(2) \leq 2$ are inconsistent.

Dijkstra: Label Setting

There are four versions of the SP problem.

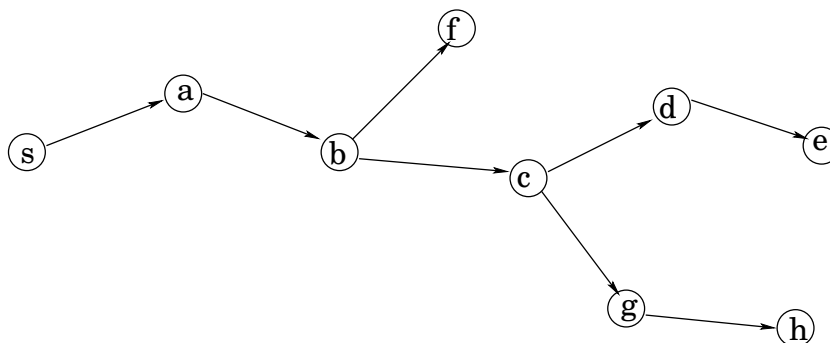
1. [Single Source-Destination Pair] Given a pair of node s and t , find a shortest path between them
2. [Single-Source] Given a source node s , find shortest paths to all other nodes.
3. [Single-Destination] Given a node t , find shortest paths from all other nodes to t .
4. [All-Pair] Find shortest paths between all pairs of nodes in G .

Observations:

- (2) = (3) with edge reversal.
- Methods for (1) also solve (2).
- For (4), run (1) from each node.
- (1) is the central problem.

Shortest Path Properties

1. **Shortest Path Tree:** union of shortest paths from s to all other nodes.
2. Explicitly storing $n - 1$ paths can take $\Omega(n^2)$ space.
3. Instead, we use the subpath optimality to store the tree.
4. If $s \rightarrow v_1 \rightarrow \dots \rightarrow v_i \rightarrow \dots \rightarrow v_j \rightarrow \dots \rightarrow v_k$ is a shortest path, then the subpath $v_i \rightarrow \dots \rightarrow v_j$ is a **SP** from v_i to v_j .

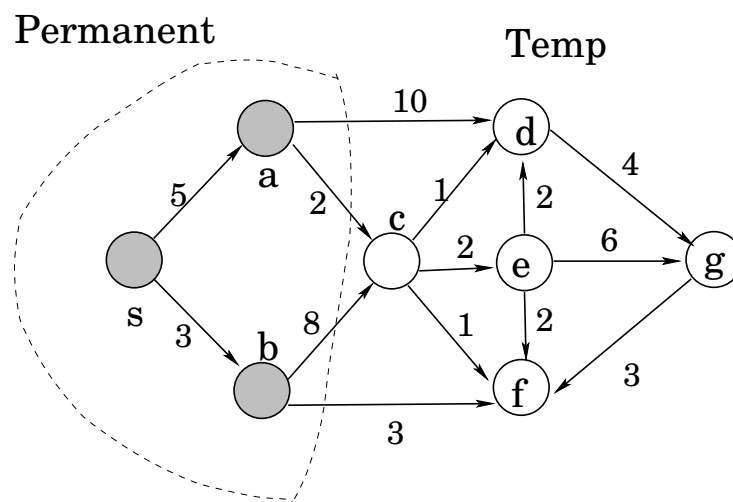


5. Implicitly storing the shortest path tree:

<i>Node</i>	s	a	b	c	d	e	f	g	h
<i>Pred</i>	ϕ	s	a	b	c	d	b	c	g

Dijkstra's SP Algorithm

1. Computes SP tree when the edge costs are non-negative.
2. Maintain labels $d(i)$ for each node i , current best estimate.
3. Some labels *temporary*, others *permanent*.
4. In each step, choose the node with *smallest temporary label*. Make it permanent, and scan its neighbors to update their $d()$.



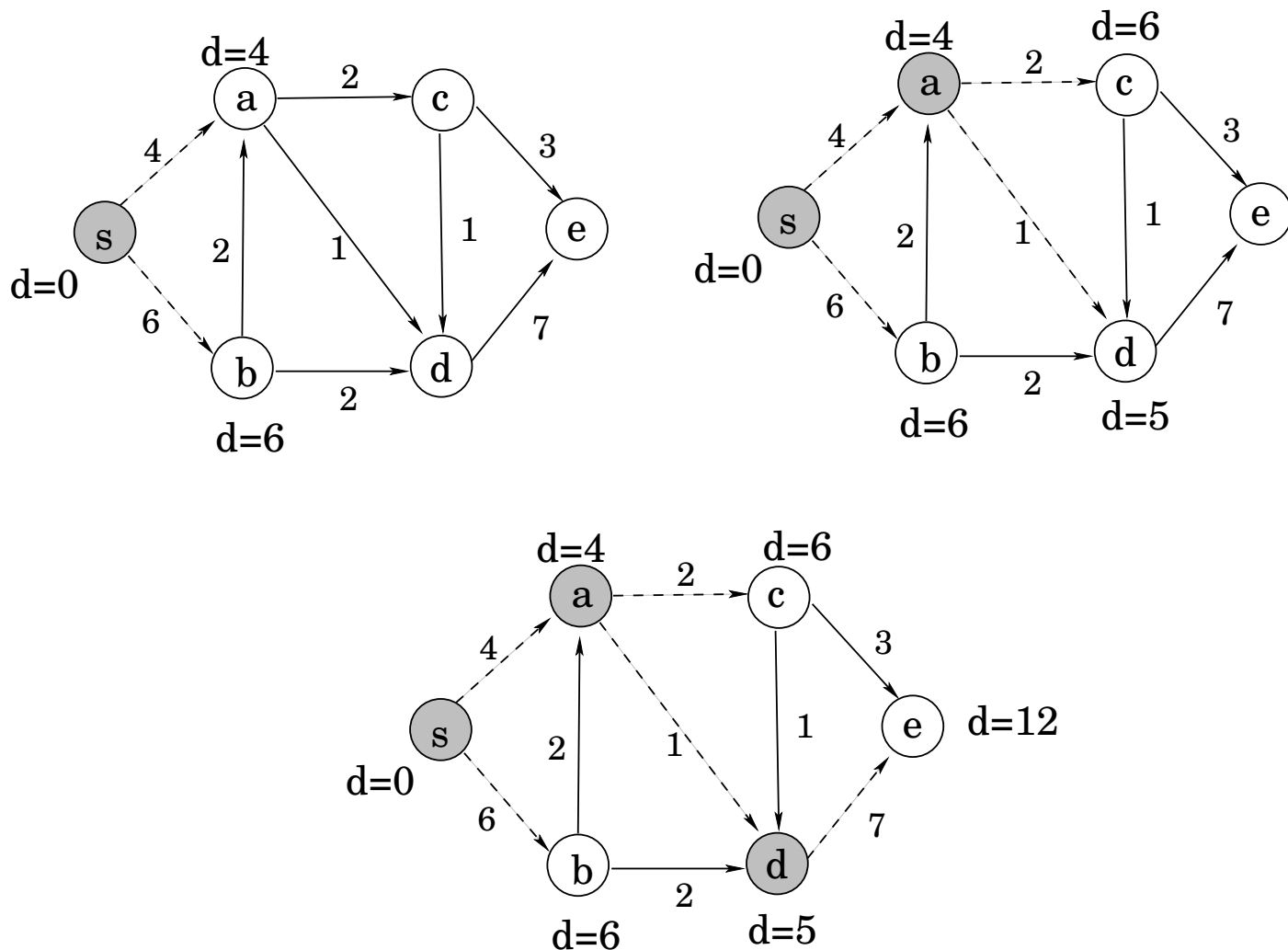
5. When done, the $d()$ labels are SP dist.
6. The predecessors define SP tree.

Dijkstra's SP Algorithm

algorithm Dijkstra

1. $S = \emptyset, \quad T = V;$
2. $d(i) = \infty$ for each $i \in T;$
3. $d(s) = 0, \quad \text{pred}(s) = 0;$
4. **while** $|S| < |V|$ **do**
5. **Choose** $i \in T$ **with** $d(i) = \min_{j \in T} \{d(j)\};$
6. $S = S \cup \{i\}$ **and** $T = T - \{i\};$
7. **for each** $(i, j) \in E$ **do**
8. **if** $d(j) > d(i) + c_{ij}$ **then**
9. $d(j) = d(i) + c_{ij}$ **and** $\text{pred}(j) = i;$
10. **end;**

Illustrating Dijkstra's Algorithm

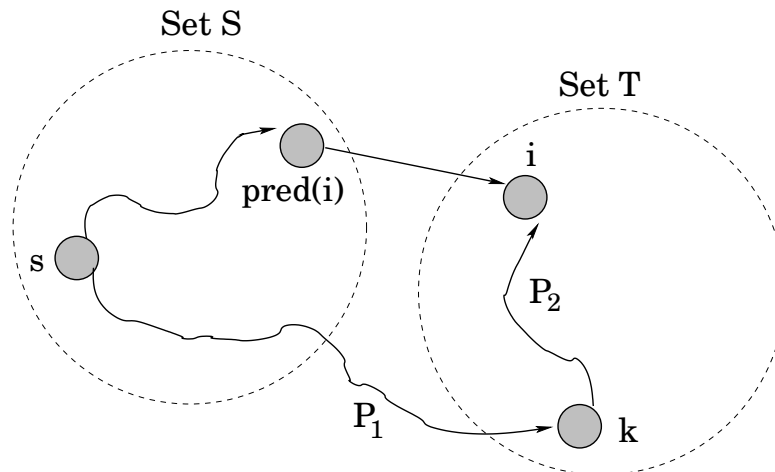


Correctness of Dijkstra

- [Invariant 1:] $d(i)$ for all $i \in S$ is optimal.
- [Invariant 2:] $d(j)$ for each $j \in T$ is the length of any shortest path whose internal nodes lie in S .
- Proof by induction on the cardinality of the set S .

Proof of Invariant 1

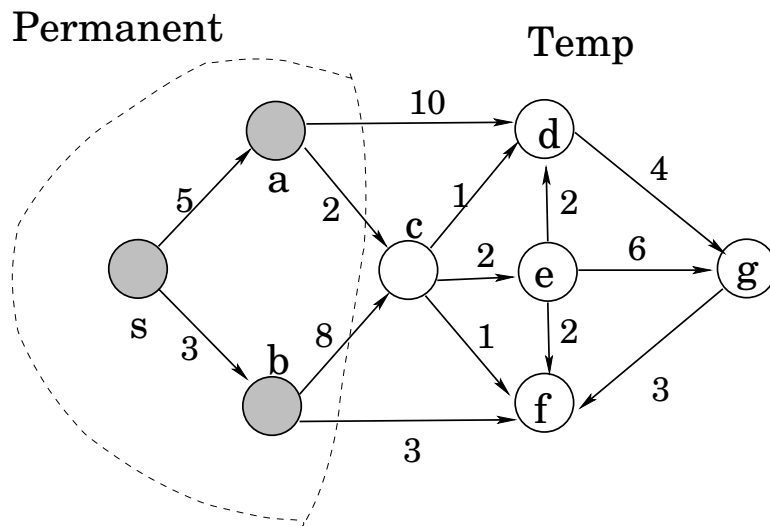
- $d(i)$ for all $i \in S$ is optimal.



1. A step transfers node i with smallest label from T to S .
2. $d(i)$ is SP length over paths not using T nodes. We show that paths using T nodes are at least as long.
3. Assume such a shortest path P , which can be decomposed into P_1 and P_2 , where P_1 has no internal T nodes.
4. i has smallest label: $len(P_1) \geq d(k) \geq d(i)$.
5. Positive edges costs: $len(P) \geq len(P_1)$.
6. Thus, $len(P) \geq d(i)$.

Proof of Invariant 2

- $d(j)$ for each $j \in T$ is the length of any shortest path whose internal nodes lie in S .
1. When a new node i is made permanent, $d()$ labels for some nodes in T might decrease. Why?
 2. Because i might be a new internal node.



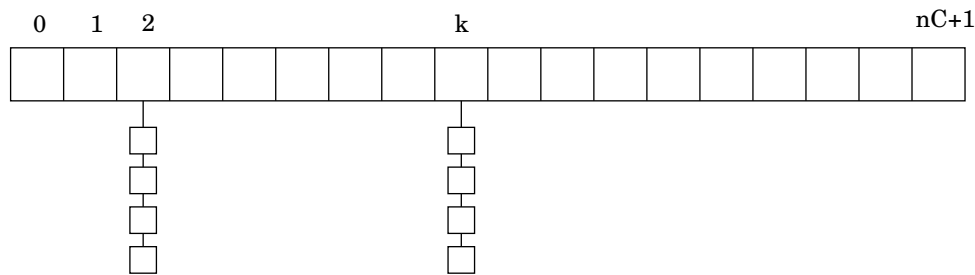
3. But Dijkstra scans all neighbors of i , and updates their distance.
4. Thus, $d(j)$ labels are correct for paths that use only S nodes as internal.

Time Complexity

- *Node Selection*: Each selection requires a scan of current temp nodes, and each selection makes one node permanent. So, the total work is $\sum_{i=1}^n i = O(n^2)$.
- *Label Update*: When making i permanent, algorithm scans all neighbors of i . Total work is $\sum_i |A(i)| = m$.
- Straightforward implementation of Dijkstra takes $O(n^2)$ time.
- Since the number of edges, m , is often much smaller than n^2 , can we improve the bound?
- One idea is to improve node selection by keeping d labels sorted.

Dial's Implementation

- *Monotone Property*: d labels made permanent by Dijkstra are *non-decreasing*.
- Suppose largest edge cost is C . Max distance label is nC .
- Dial keeps $nC + 1$ buckets, where bucket k stores all nodes with temp distance k , in doubly linked list.



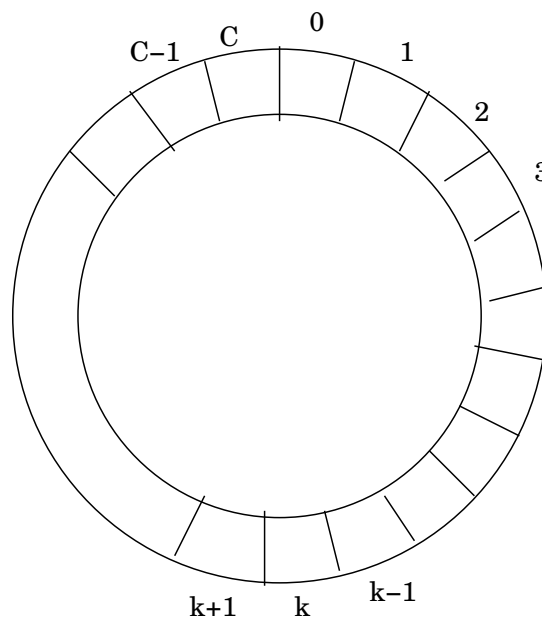
- Find first non-empty bucket, say, k . Each node in k 's list has smallest temp label. One by one, make them permanent, and scan their neighbors to update their distance.
- When a node's distance is updated from d_1 to d_2 , we move it from bucket d_1 to bucket d_2 .

Analysis of Dial

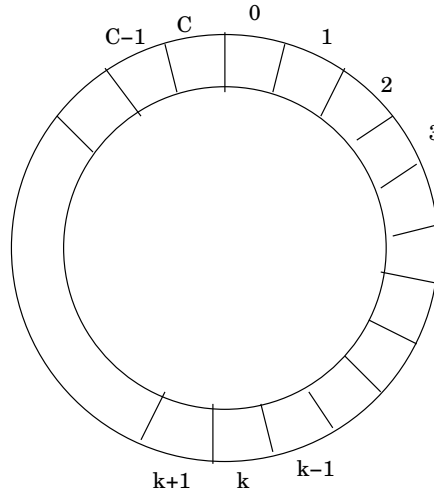
- By Monotone Property, if the last non-empty bucket was k , then the next one must after k .
- Total time spent in scanning buckets is $nC + 1$.
- Adding or removing from a bucket list takes $O(1)$.
- Thus, total time is $O(nC + m)$.
- When C is small, say, a constant, this is optimal $O(n + m)$.
- But if C is large, say, n^4 or 2^n , the bucket scanning can be very slow.
- The memory requirement is also large: $O(nC + m)$.

Improved Dial

- *Bounded Label Difference*: At any time, the difference between two (finite) temporary labels is at most C .
- **Proof.** Let $d(i)$ be the last label made permanent.
- For any other node $j \in T$, $d(j) = d(l) + c_{l,j}$, for some l .
- But $d(l) \leq d(i)$, and $d(i) \leq d(j)$.
- Thus, nodes in T have labels between $d(i)$ and $d(i) + C$.



Improved Dial



- Store node with temp label $d(j)$ in bucket $d(j) \bmod (C + 1)$.
- Over time, bucket k stores nodes with distance $k, k + C, k + 2C$ etc. But by **Bounded Label Difference**, at any time, the bucket holds nodes with the same distance.
- Examine buckets sequentially, with wraparound. Start next search from where the last one finished.
- Worst-case time $O(m + nC)$, memory $O(m + C)$.

Heap Implementation

- Heap data structure allows the following operations:
 1. CreateHeap (H)
 2. FindMin (i, H)
 3. Insert (i, H)
 4. DecreaseKey (newVal, i, H)
 5. DeleteMin (i, H)
- Maintain temporary labels in the Heap.
- Node Selection done via FindMin and DeleteMin.
- Updates handled via DecreaseKey.
- Remember Dijkstra's algorithm does n node selections and m updates.

Different Heaps

- *Binary Heap*: Standard heap requires $O(\log n)$ time to do insert, decreaseKey, and deleteMin. Other operations take $O(1)$ time.
- So, using Binary Heap, Dijkstra takes $O(m \log n)$ time.
- *d-Heap*: A d -ary heap requires $O(\log_d n)$ time to do insert and decreaseKey, but needs $O(d \log_d n)$ time for each deleteMin. Other operations take $O(1)$ time.
- So, using d -Heap, for any parameter $d \geq 2$, Dijkstra takes $O(m \log_d n + nd \log_d n)$ time.
- How do you pick d ?
- In Sparse graphs, $m = O(n)$, binary heap runs in $O(n \log n)$ time.
- In dense graphs, $m = \Omega(n^{1+\epsilon})$, $d = \lceil m/n \rceil$ makes the algorithm run in $O(m)$ time.

More Implementations

- *Fibonacci Heap*: All operations in $O(1)$ amortized, except `deleteMin`, which takes $O(\log n)$.
- Implementing Dijkstra using F-Heap takes worst-case time $O(m + n \log n)$ time. Best strongly polynomial algorithm.
- Empirically, researchers have found that Dial's implementation is the fastest Label-Setting shortest path algorithm for most networks.
- Experiments by Gallo-Pallottino suggest that some implementations of the Label-Correcting algorithms (Thresh1 and Thresh2) have the fastest running time both for positive and negative edge costs.

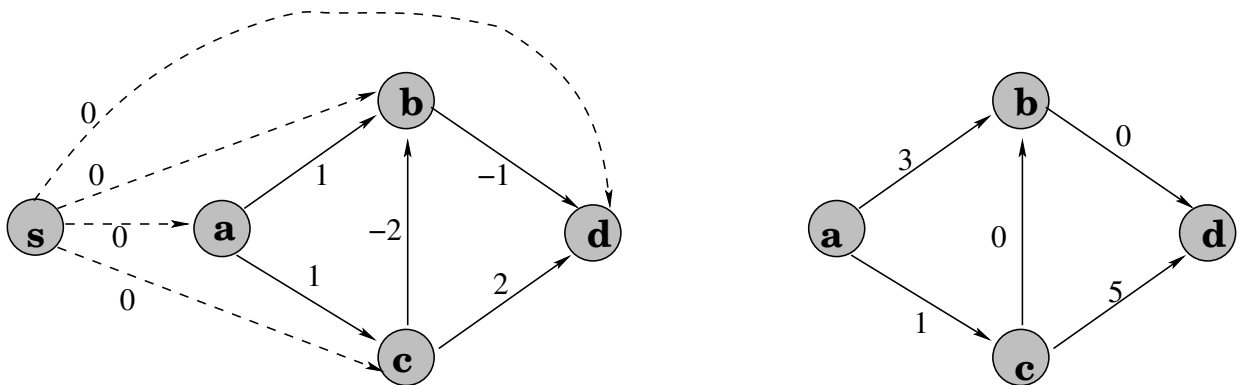
All-Pairs Shortest Paths

- Given $G = (V, E)$, compute shortest path distances between all pairs of nodes.
- Run single-source shortest path algorithm from each node as root. Total complexity is $O(nS(n, m))$, where $S(n, m)$ is the time for one shortest path iteration.
- If non-negative edges, use Dijkstra's algorithm: $O(m \log n)$ time per iteration.
- With negative edges, need to use Bellman-Ford algorithm: $O(nm)$ time per iteration.
- Edmonds-Karp introduced a trick, which allows one to use Dijkstra after one run of Bellman-Ford.

All-Pairs Shortest Paths (2)

- Add a new node s ; join it to all others with 0-cost edges.
- Compute shortest path distances from s to all other nodes, using Bellman-Ford.
- E.g. $d(a) = 0, d(b) = -2, d(c) = 0, d(d) = -3$.
- In the new graph, modify the weight of edge (u, v) as follows:

$$c'(u, v) = c(u, v) + d(u) - d(v) \quad (*)$$



All-Pairs Shortest Paths (3)

- New costs are non-negative (why?), so use Dijkstra's algorithm, n times, once per node.
- Non-negativity follows from Bellman-Ford condition: $d(v) \leq d(u) + c_{uv}$, which implies that $c'_{uv} = c_{uv} + d(u) - d(v) \geq 0$.
- Correctness of path computation: for any path p from x to y , we have $l'(p) = l(p) + d(x) - d(y)$.
- This follows from telescoping of terms. If p breaks into two subpaths, p_1 from x to z , and p_2 from z to y , then by induction:
$$l'(p) = l'(p_1) + l'(p_2) = (l(p_1) + d(x) - d(z)) + (l(p_2) + d(z) - d(y)) = l(p) + d(x) - d(y).$$
- Thus, total time is $O(nm + nm \log n)$.

All-Pairs Shortest Paths (4)

- In worst-case, for even positive-weight graphs, All-Pairs algorithm takes $\Theta(n^3 \log n)$ time.
- For negative-weight graphs, complexity is $O(n^4)$, but Edmonds-Karp heuristic improves it $O(n^3 \log n)$. But does require implementing both Dijkstra and Bellman-Ford.
- Floyd-Warshall is a simple algorithm, with no data structure at all, for computing all pair shortest paths.
- The algorithm is always $O(n^3)$.

Floyd-Warshall Algorithm

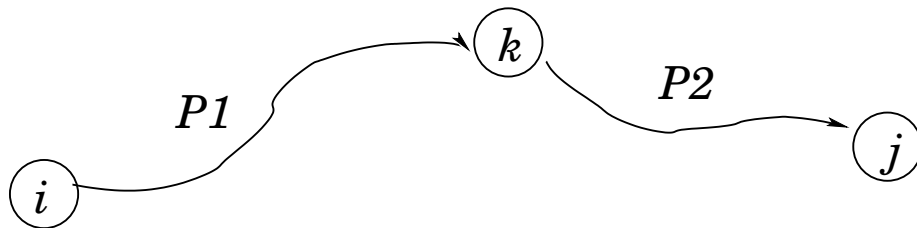
- $G = (V, E)$ has vertices $\{1, 2, \dots, n\}$.
- W is the cost adjacency matrix of graph G .
- Matrix D encodes the pair-wise distances. Assume all entries initialized to 0.

algorithm Floyd-Warshall

1. $D = W$;
2. for $k = 1$ to n
3. for $i = 1$ to n
4. for $j = 1$ to n
5. $d_{ij} = \min\{d_{ij}, d_{ik} + d_{kj}\}$
6. return D .

Correctness of Floyd-Warshall

- P_{ij}^k : shortest path whose intermediate nodes are in $\{1, 2, \dots, k\}$.
- Goal is to compute P_{ij}^n , for all i, j .

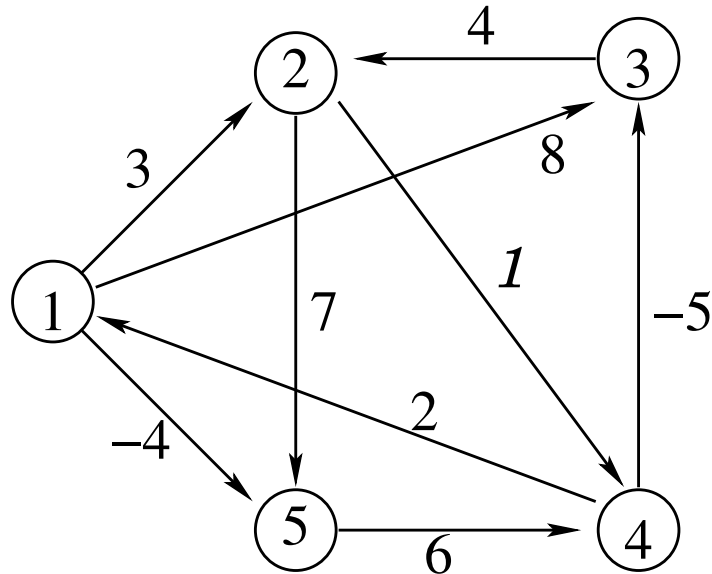


- Use Dynamic Programming. Two cases:
 1. Vertex k not on P_{ij}^k . Then, $P_{ij}^k = P_{ij}^{k-1}$.
 2. Vertex k is on P_{ij}^k . Then, neither P_1 nor P_2 uses k as an intermediate node. in its interior. (Simplicity of P_{ij}^k .) Thus,
$$P_{ij}^k = P_{ik}^{k-1} + P_{kj}^{k-1}$$

- Recursive formula for P_{ij}^k :

1. If $k = 0$, $P_{ij}^k = c_{ij}$.
2. If $k > 0$, $d_{ij}^k = \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$

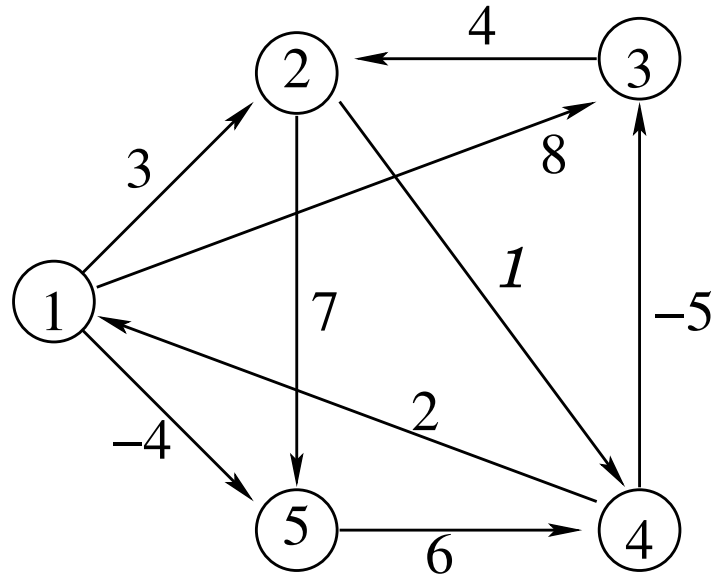
Example



- Matrices D_0 and D_1 :

$$\begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

Example



- Matrices D_2 and D_5 :

$$\begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$