



中山大學
SUN YAT-SEN UNIVERSITY

Lecture 9

Shortest Path Algorithms

Algorithm Design and Analysis

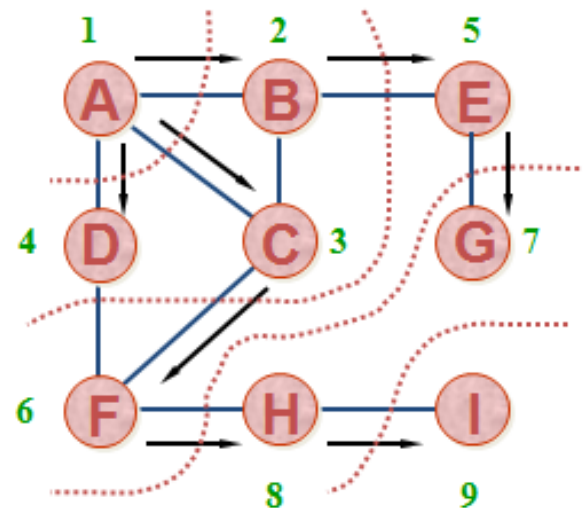
zhangzizhen@gmail.com

Shortest Path Algorithms

- Shortest Path Problem in Unweighted Graphs
 - Bread-First Search
- Single-Source Shortest Path Problem
 - Dijkstra algorithm
 - Bellman-Ford algorithm
- All-pairs Shortest Path Problem
 - Floyd-Warshall algorithm
- Application
 - A system of difference constraints

Breadth-first Traversal

- Breadth first traversal can be done by using a **queue**. The head is the vertex to be visited. When it is visited, all its adjacent vertices are put in the queue.
- Let v be the first vertex to be visited, and w_1, \dots, w_k be the adjacent vertices of v . We will first put v in the queue. Visit v and put w_1, \dots, w_k in the queue. Then visit w_1 and put the adjacent vertices of w_1 into the queue, ..., until all the vertices have been visited.

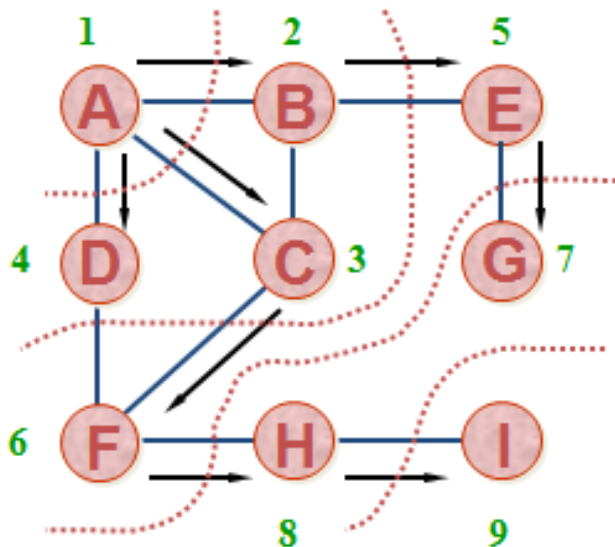


Shortest Paths in Unweighted Graph

- In BFS, vertices are discovered in the order of increasing distance from the root, so this tree has a very important property.
- The unique tree path from the root to any node $x \in V$ uses the smallest number of edges (or equivalently, intermediate nodes) possible on any root-to- x path in the graph.

Finding Paths

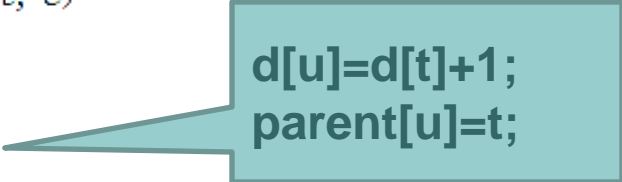
- The **parent array** set within BFS() is very useful for finding interesting paths through a graph.
- The vertex which **discovered** vertex i is defined as $\text{parent}[i]$.
- The parent relation defines a tree of discovery with the root of the tree.



| vertex | A | B | C | D | E | F | G | H | I |
|--------|----|---|---|---|---|---|---|---|---|
| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| parent | -1 | 1 | 1 | 1 | 2 | 3 | 5 | 6 | 8 |

Breadth-first Search

```
1  procedure BFS( $G, v$ ) is
2      create a queue  $Q$ 
3      create a set  $V$ 
4      add  $v$  to  $V$ 
5      enqueue  $v$  onto  $Q$ 
6      while  $Q$  is not empty loop
7           $t \leftarrow Q.dequeue()$ 
8          if  $t$  is what we are looking for then
9              return  $t$ 
10         end if
11         for all edges  $e$  in  $G.adjacentEdges(t)$  loop
12              $u \leftarrow G.adjacentVertex(t, e)$ 
13             if  $u$  is not in  $V$  then
14                 add  $u$  to  $V$ 
15                 enqueue  $u$  onto  $Q$ 
16             end if
17         end loop
18     end loop
19     return none
20 end BFS
```

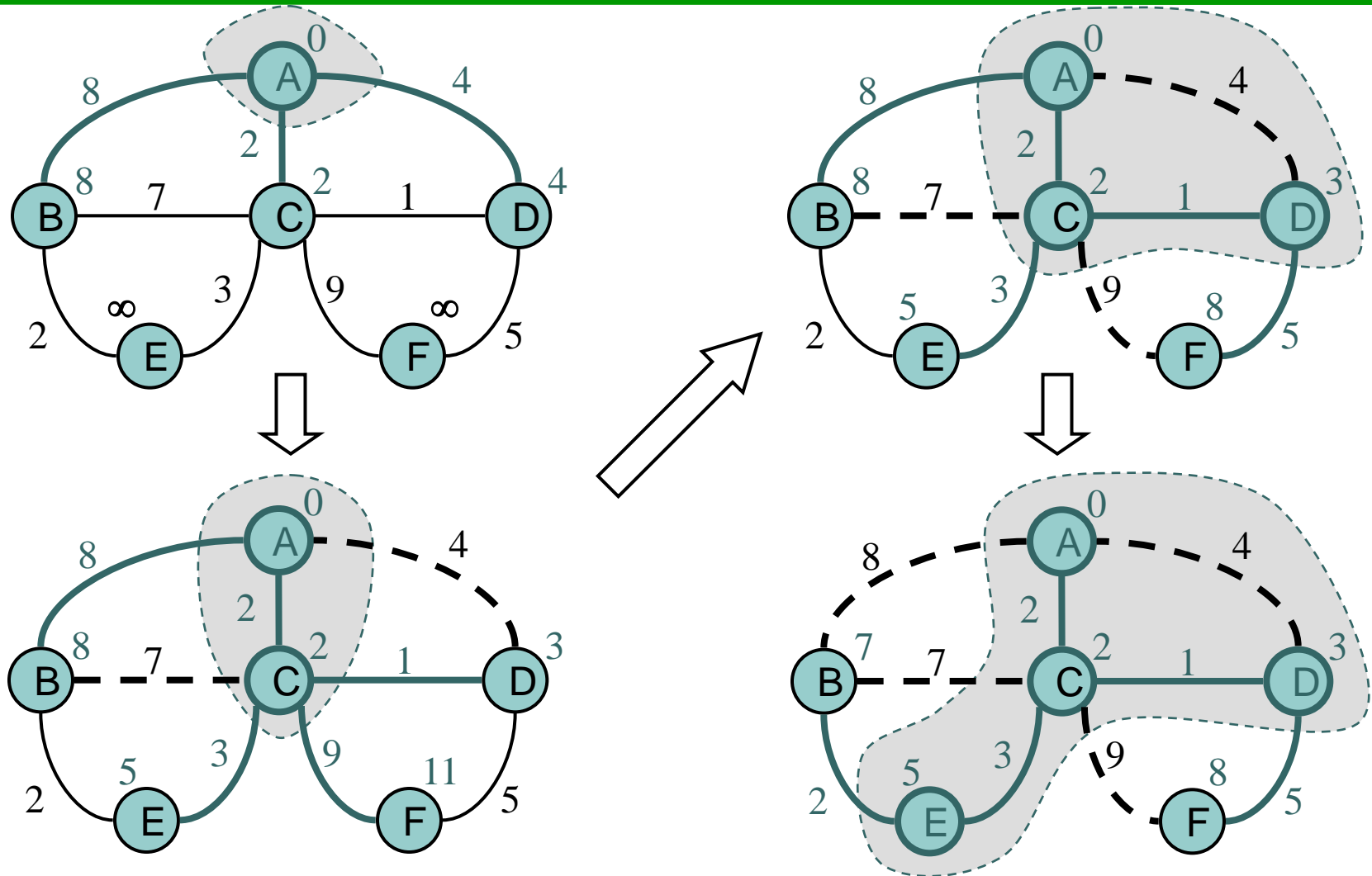


$d[u] = d[t] + 1;$
 $parent[u] = t;$

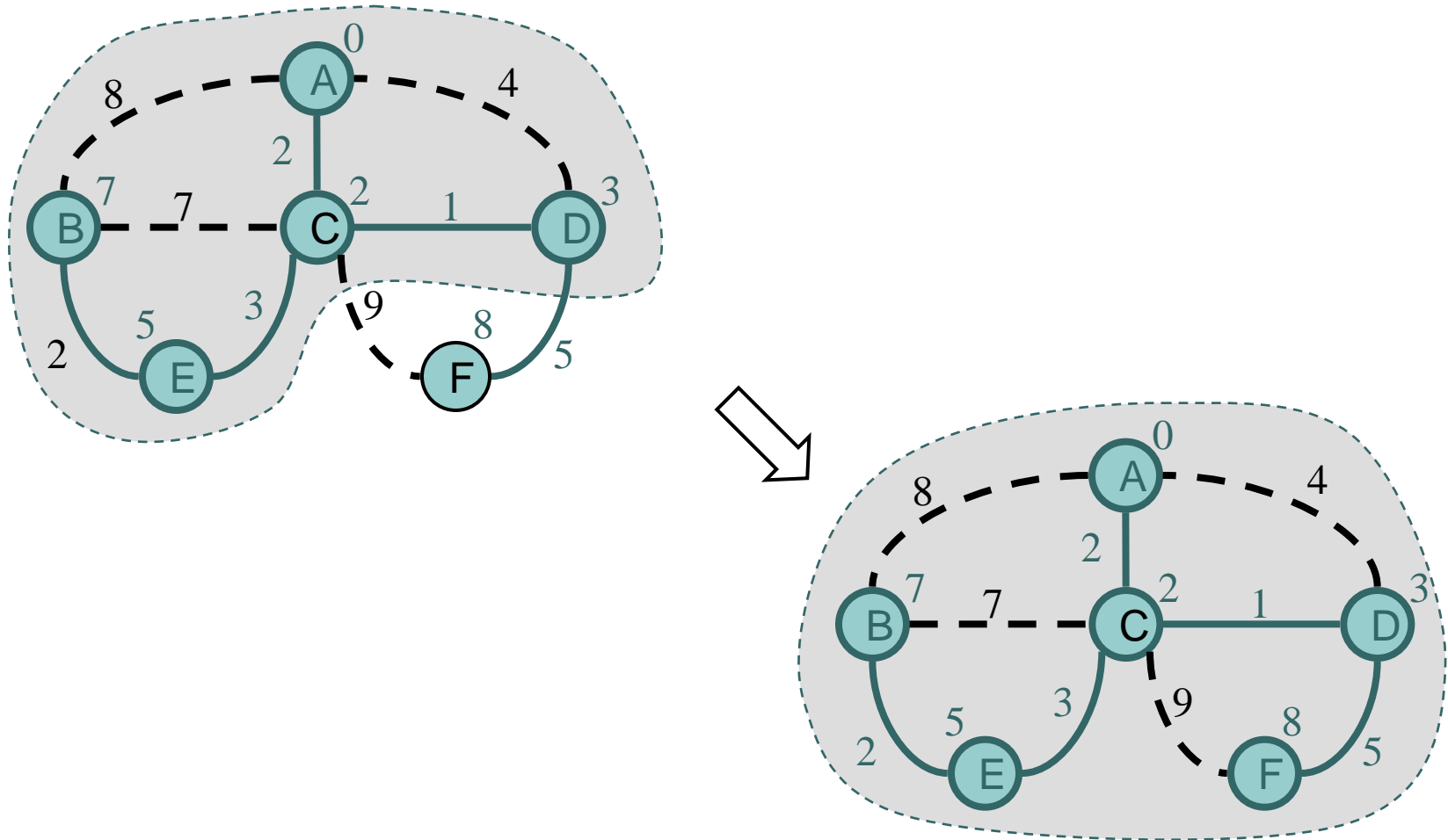
The Dijkstra's Algorithm

- Grow a “**cloud**” of vertices, beginning with s and eventually covering all the vertices.
- Store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices.
- At each step
 - Add to the cloud the vertex u outside the cloud with the **smallest distance** label, $d(u)$.
 - Update the labels of the vertices adjacent to u .

Example of the Dijkstra's Algorithm



Example of the Dijkstra's Algorithm

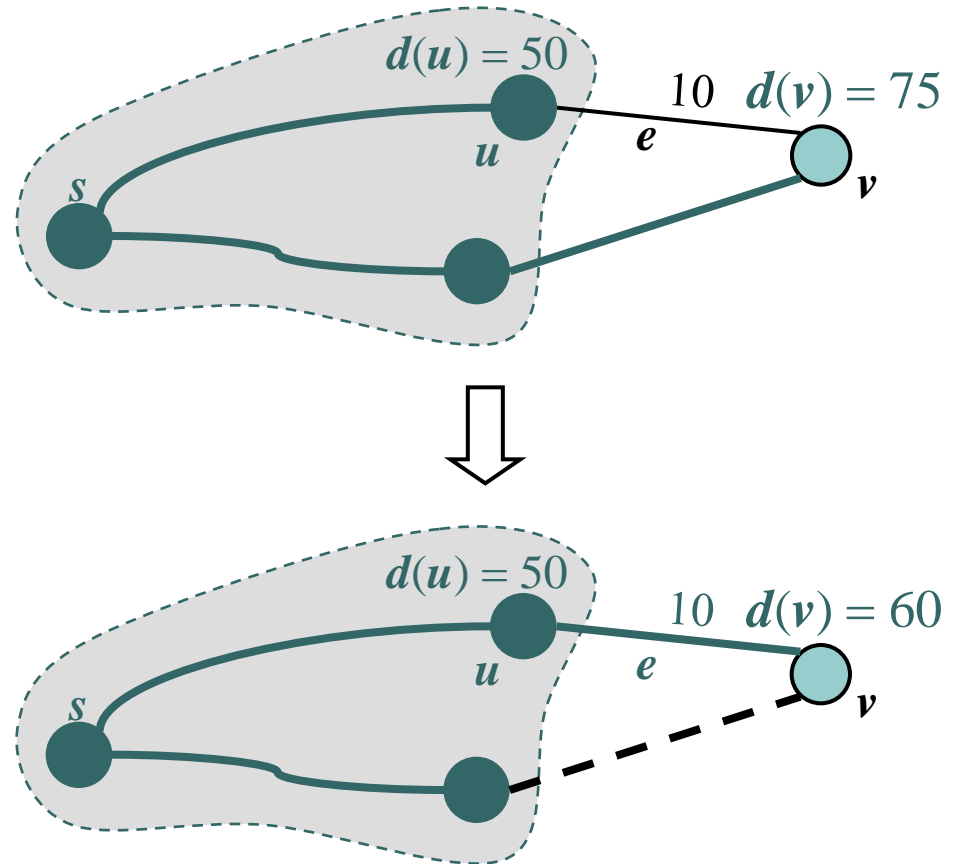


Relaxation

- Consider an edge $e = (u, v)$ such that
 - u is the vertex most recently added to the cloud
 - v is not in the cloud
- The relaxation of edge e updates distance $d(v)$ as follows:

Relax(u, v, w)

$$d(v) \leftarrow \min\{d(v), d(u) + w(e)\}$$



Implementation of Dijkstra's Algorithm

```

void Digraph<Weight, graph_size> :: set_distances(Vertex source,
                                                Weight distance[]) const
/* Post: The array distance gives the minimal path weight from vertex source to each
   vertex of the Digraph. */
{ Vertex v, w; bool found[graph_size]; // Vertices found in S
  for (v = 0; v < count; v++) {
    found[v] = false;
    distance[v] = adjacency[source][v];
  }
  found[source] = true; // Initialize with vertex source alone in the set S.
  distance[source] = 0;
  for (int i = 0; i < count; i++) { // Add one vertex v to S on each pass.
    Weight min = infinity;
    for (w = 0; w < count; w++) if (!found[w])
      if (distance[w] < min) {
        v = w;
        min = distance[w];
      }
    found[v] = true;
    for (w = 0; w < count; w++) if (!found[w])
      if (min + adjacency[v][w] < distance[w])
        distance[w] = min + adjacency[v][w];
  }
}

```



If there is no edge between (u,v),
then adjacency[u][v]= $+\infty$



Relaxation



Add “prev[w] = v;” here
for recovering the path

Time Complexity

A Generic Single-Source Shortest Path Algorithm:

```
Initialize(G, s);  
S :=  $\emptyset$ ;  
Q := V[G];  
while Q  $\neq \emptyset$  do  
    u := Extract-Min(Q);  
    S := S  $\cup$  {u};  
    for each v  $\in$  Adj[u] do  
        Relax(u, v, w)  
    od  
od
```

Complexity:

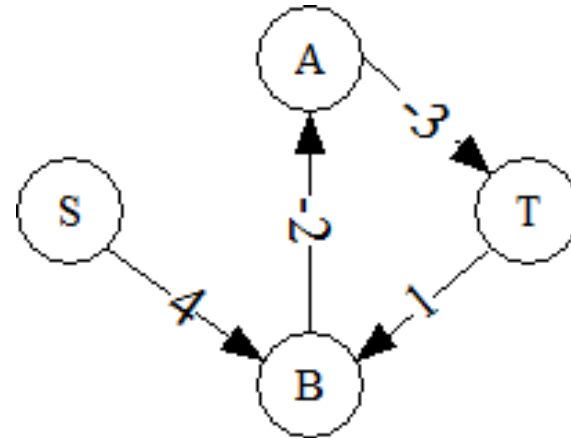
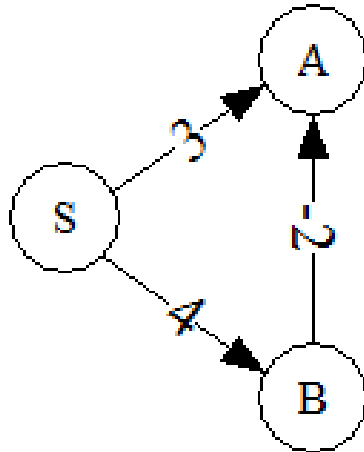
Naive implementation:
 $O(V^2)$

Using binary heaps:
 $O((V+E) \log V)$.

Using Fibonacci heaps:
 $O(E + V \log V)$.

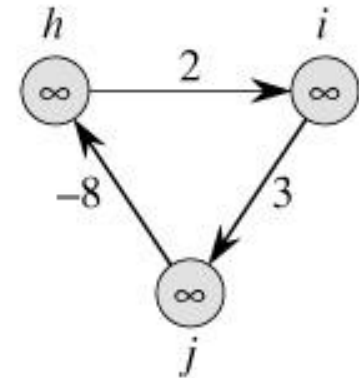
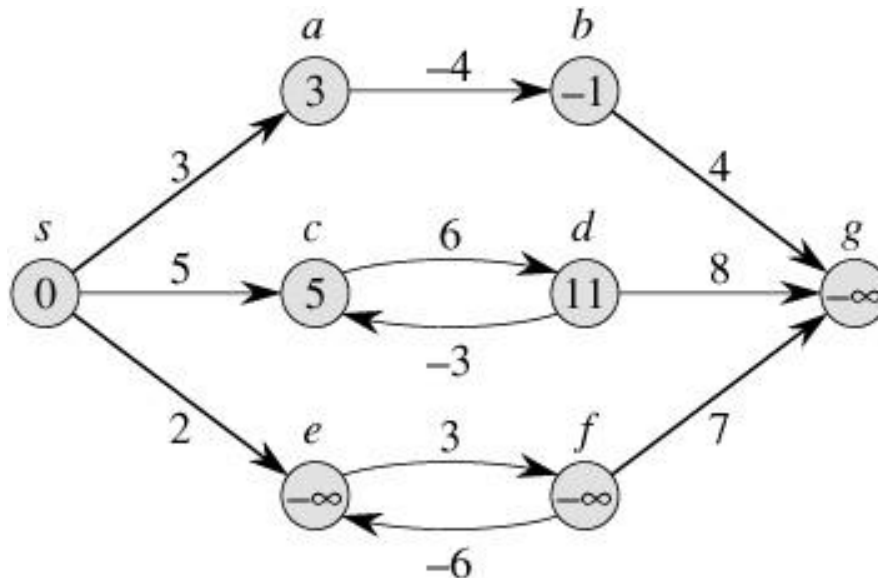
Negative weight edges and cycles

- If the graph contains negative weight edges, Dijkstra's algorithm cannot be directly used.
- Example:



Bellman–Ford algorithm

- The Bellman–Ford algorithm can be used on graphs with negative edge weights, as long as the graph contains no negative cycle reachable from the source vertex s . The presence of such cycles means there is no shortest path, since the total weight becomes lower each time the cycle is traversed.

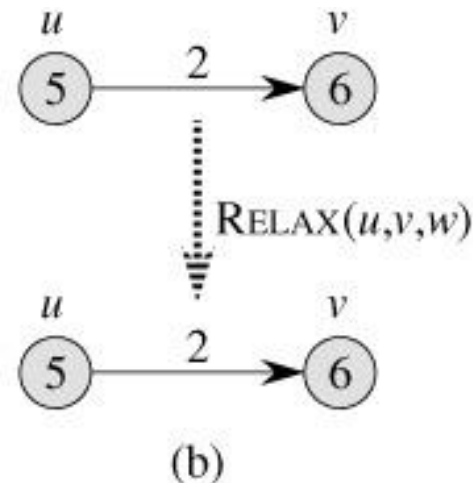
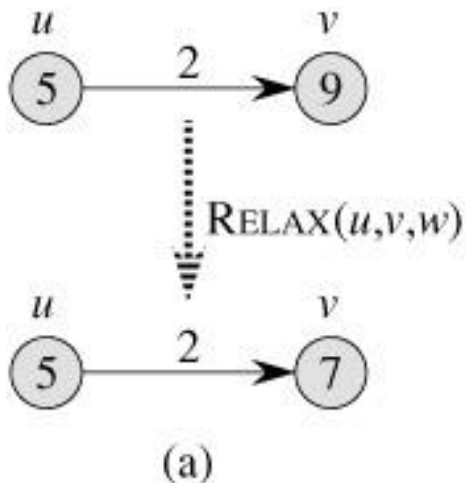


Revisit: relaxation

- $d[v]$ is an upper bound on the weight of a shortest path from source s to v .

RELAX(u, v, w)

- 1 if $d[v] > d[u] + w(u, v)$
- 2 then $d[v] \leftarrow d[u] + w(u, v)$



Important Lemma

- Path-relaxation property (Lemma 24.15):
- If $p = \langle v_0, v_1, \dots, v_k \rangle$ is a shortest path from $s = v_0$ to v_k , and the edges of p are relaxed in the order (v_0, v_1) , $(v_1, v_2), \dots, (v_{k-1}, v_k)$, then $d[v_k] = \delta(s, v_k)$.
- This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p .

Bellman–Ford algorithm

BELLMAN-FORD(G, w, s)

1 INITIALIZE-SINGLE-SOURCE(G, s)

2 for $i \leftarrow 1$ to $|V[G]| - 1$

3 do for each edge $(u, v) \in E[G]$

4 do RELAX(u, v, w)

5 for each edge $(u, v) \in E[G]$

6 do if $d[v] > d[u] + w(u, v)$

7 then return FALSE

8 return TRUE

- Complexity: $O(VE)$

All-Pairs Shortest Path

- Notice that finding the shortest path between a pair of vertices (s, t) in worst case requires first finding the shortest path from s to all other vertices in the graph.
- Many applications, such as finding the center or diameter of a graph, require finding the shortest path between all pairs of vertices.
- We can run Dijkstra's algorithm n times (once from each possible start vertex) to solve all-pairs shortest path problem in $O(n^3)$. Can we do better?

The Floyd-Warshall algorithm

- The Floyd-Warshall algorithm starts by numbering the vertices of the graph from 1 to n . Define $W[i,j]^k$ to be the length of the shortest path from i to j using only vertices numbered from $1, 2, \dots, k$ as possible intermediate vertices.
- When $k=0$, we are allowed no intermediate vertices, so the only allowed paths are the original edges. Thus the initial all-pairs shortest-path matrix consists of the initial adjacency matrix.

The Floyd-Warshall algorithm

- We will perform n iterations, where the k -th iteration allows only the first k vertices as possible intermediate steps on the path between each pair of vertices x and y .
- At each iteration, we allow a richer set of possible shortest path by adding a new vertex as a possible intermediary. Allowing the k th vertex as a stop helps only if there is a short path that goes through k , so

$$w[i, j]^k =$$

$$\min(w[i, j]^{k-1}, w[i, k]^{k-1} + w[k, j]^{k-1})$$

Implementation of the Floyd-Warshall algorithm

```

1 let dist be a  $|V| \times |V|$  array of minimum distances initialized to  $\infty$  (infinity)
2 for each vertex  $v$ 
3    $\text{dist}[v][v] \leftarrow 0$ 
4 for each edge  $(u, v)$ 
5    $\text{dist}[u][v] \leftarrow w(u, v)$  // the weight of the edge  $(u, v)$ 
6 for  $k$  from 1 to  $|V|$ 
7   for  $i$  from 1 to  $|V|$ 
8     for  $j$  from 1 to  $|V|$ 
9       if  $\text{dist}[i][j] > \text{dist}[i][k] + \text{dist}[k][j]$ 
10          $\text{dist}[i][j] \leftarrow \text{dist}[i][k] + \text{dist}[k][j]$ 
11       end if

```

- Time complexity: $O(V^3)$. The Floyd-Warshall algorithm is a good choice for computing paths between all pairs of vertices in **dense graphs**.

Difference constraints

- 问题： 设需要安排5项任务T1~T5， 有如下要求（为简单起见， 我们忽略完成每项任务的所需的时间）：
 1. 先做T1再做T2；
 2. 做T1后至少1小时后才能做T5；
 3. 先做T5再做T2， 且两者间隔时间不超过1小时；
 4. 先做T1再做T3， 且两者间隔时间不超过5小时；
 5. 先做T1再做T4， 且两者间隔时间不超过4小时.
- 如何安排任务？

Difference constraints

- A set of inequalities (difference constraints):
 - $x_1 - x_2 \leq 0$,
 - $x_1 - x_5 \leq -1$,
 - $x_2 - x_5 \leq 1$,
 - $x_3 - x_1 \leq 5$,
 - $x_4 - x_1 \leq 4$.

A system of difference constraints

- In a *system of difference constraints*, each row of the matrix A contains one 1 and one -1, and all other entries of A are 0. Thus, the constraints given by $Ax \leq b$ are a set of m *difference constraints* involving n unknowns, in which each constraint is a simple linear inequality of the form

$$x_j - x_i \leq b_k,$$

where

$$1 \leq i, j \leq n \text{ and } 1 \leq k \leq m.$$

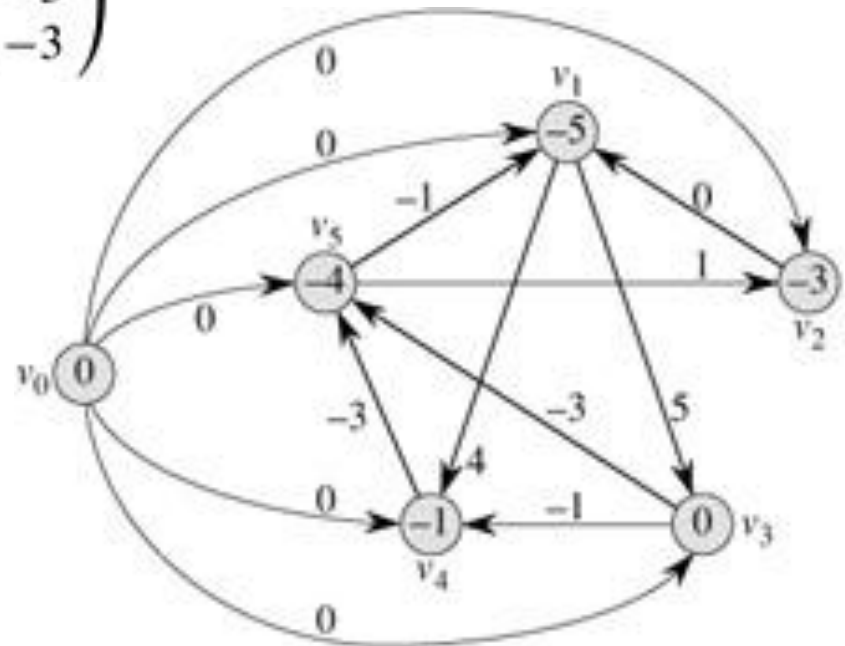
$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}.$$

Constraint graphs

- We can interpret systems of difference constraints from a graph-theoretic point of view.
- The vertex set V consists of a vertex v_i for each unknown x_i , plus an additional vertex v_0 .
- $E = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ is a constraint}\} \cup \{(v_0, v_1), (v_0, v_2), (v_0, v_3), \dots, (v_0, v_n)\}$.
- If $x_j - x_i \leq b_k$ is a difference constraint, then the weight of edge (v_i, v_j) is $w(v_i, v_j) = b_k$.
- The weight of each edge leaving v_0 is 0.

Constraint graphs

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \preceq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}.$$



Solving the system

- Theorem:** Given a system $Ax \leq b$ of difference constraints, let $G = (V, E)$ be the corresponding constraint graph. If G contains no negative-weight cycles, then

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \dots, \delta(v_0, v_n))$$
 is a feasible solution for the system. If G contains a negative-weight cycle, then there is no feasible solution for the system.
- Let $x = (x_1, x_2, \dots, x_n)$ be a solution to a system $Ax \leq b$ of difference constraints, and let d be any constant. Then

$$x + d = (x_1 + d, x_2 + d, \dots, x_n + d)$$
 is a solution to $Ax \leq b$ as well.
- Solve the system of difference constraints using the Bellman-ford algorithm.

Exercise

- soj.acmm.club
1014 1099 1119 1203 1214 1231 1240 1259 1500
- Choose **at least 2 problems** and write a report.
- Create a zip/rar file: ID_name_version.zip
- Submit it to 239o58336k.qicp.vip:55469
 - Account: login
 - Password: 123456
- Deadline: April 21.

Thank you!

