Graph Theory – Tractable Problems

Adapted from slides by B. Prabhakaran, L. Barrera, G. Bebis, J. Reif and W. T. Trotter

Topics Covered

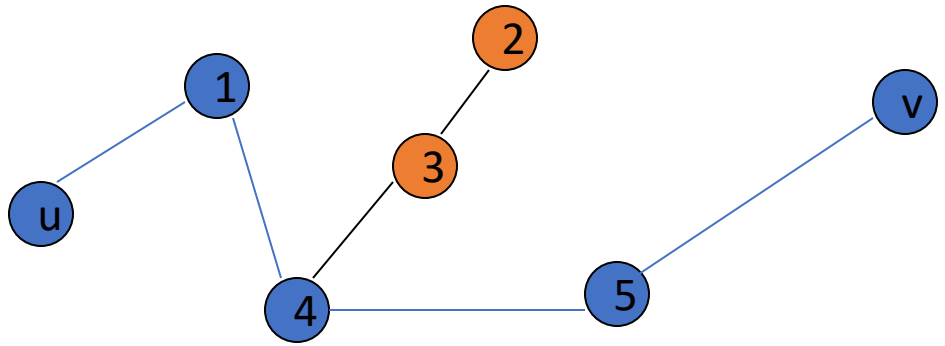
- Shortest path
- Minimum spanning tree
- Bipartite graph matching
- Euler Circuit

Shortest Path

Connectivity – Path

A **Path** is a sequence of edges that begins at a vertex of a graph and travels along edges of the graph, always connecting pairs of adjacent vertices.

Representation example: G = (V, E), Path P represented, from u to v is {{u, 1}, {1, 4}, {4, 5}, {5, v}}



Shortest Path

- Generalize distance to weighted setting
- Given graph $G = (V,E)$ with weight function $W: E \rightarrow R$ (assigning real values to edges)
- Weight of path $p = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is

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

- Shortest path = a path of the minimum weight
- Applications
  - static/dynamic network routing
  - robot motion planning
  - map/route generation in traffic

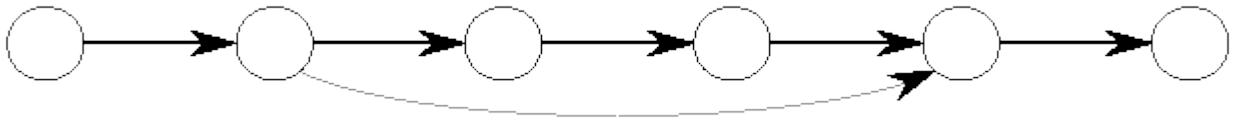
Shortest-Path Problems

- Shortest-Path problems
  - **Single-source (single-destination).** Find a shortest path from a given source (vertex *s*) to each of the vertices. The topic of this lecture.
  - **Single-pair.** Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
  - **All-pairs.** Find shortest-paths for every pair of vertices.

- Unweighted shortest-paths – BFS.

Optimal Substructure

- An optimization problem is one in which you want to find, not just *a* solution, but the *best (optimal)* solution
- Optimal substructure: An optimal solution can be constructed efficiently from optimal solutions of its subproblems.
- *Theorem*: subpaths of shortest paths are shortest paths
- Proof (”cut and paste”)
  - if some subpath were not the shortest path, one could substitute the shorter subpath and create a shorter total path



Greedy Algorithms

- A “greedy algorithm” sometimes works well for optimization problems
- A greedy algorithm works in phases. At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum
- For problems with optimal substructure, greedy algorithms can sometimes obtain global optimum (not always, alt. using Dynamic Programming)

Dijkstra's algorithm

- Dijkstra's algorithm is a solution to the single-source shortest path problem in graph theory.
- Works on both directed and undirected graphs. However, all edges must have nonnegative weights.
- Approach: Greedy
- Dijkstra's greedy strategy produces a globally optimum solution (need to prove)

Dijkstra's algorithm

- Basic idea: find the shortest path for each vertex one by one. Each time when it finds the shortest path for a vertex, add that vertex to list and use that to update other vertices' shortest paths.

- The reason it is a greedy strategy: it treats the current shortest path to the nearest unvisited vertex as the shortest path to it. Once we claim the shortest path to a vertex, we treat it as final and will not update it in the future.

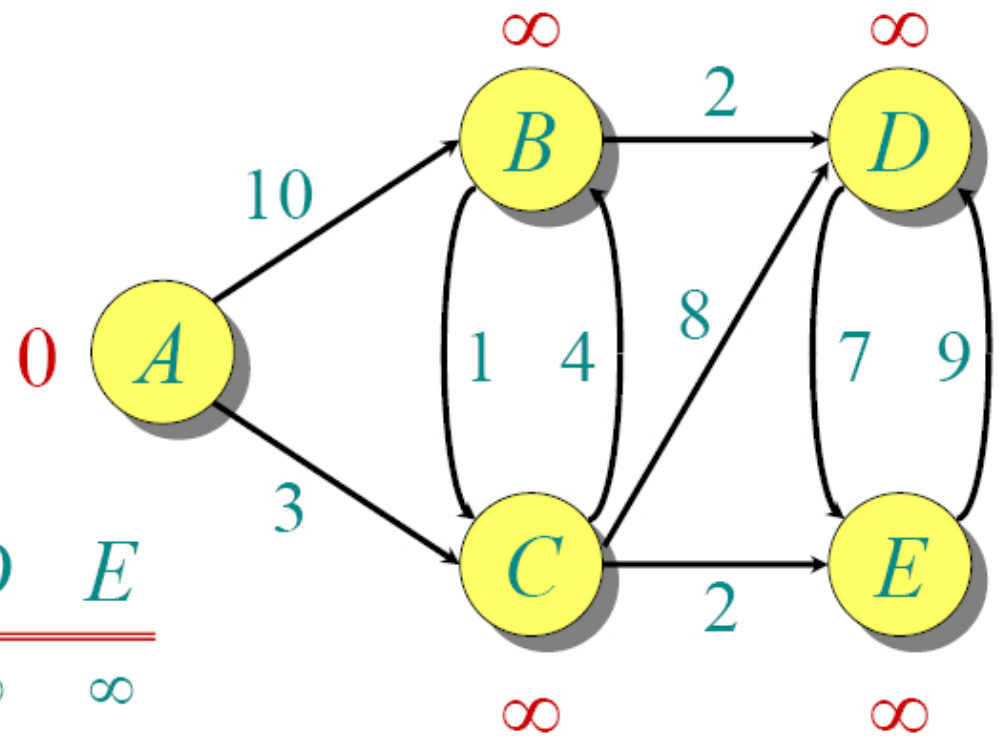
Dijkstra's algorithm

```
dist[s] ←0                                  (distance to source vertex is zero)
for  all v ∈ V–{s}
      do  dist[v] ←∞                        (set all other distances to infinity)
S←∅                                         (S, the set of visited vertices is initially empty)
Q←V                                         (Q, the queue initially contains all vertices)
while Q ≠∅                                  (while the queue is not empty)
     u ← mindistance(Q,dist)                (select the element of Q with the min. distance)
     S←SU{u}                                (add u to list of visited vertices)
     for all v ∈ neighbors[u] ∩ Q
           do  if   dist[v] > dist[u] + w(u, v)          (if new shortest path found)
                    then     dist[v] ←dist[u] + w(u, v)  (set new value of shortest path)
                             path[v] ←u                  (add traceback code)

return dist, path
```

Dijkstra Animated Example

Initialize:

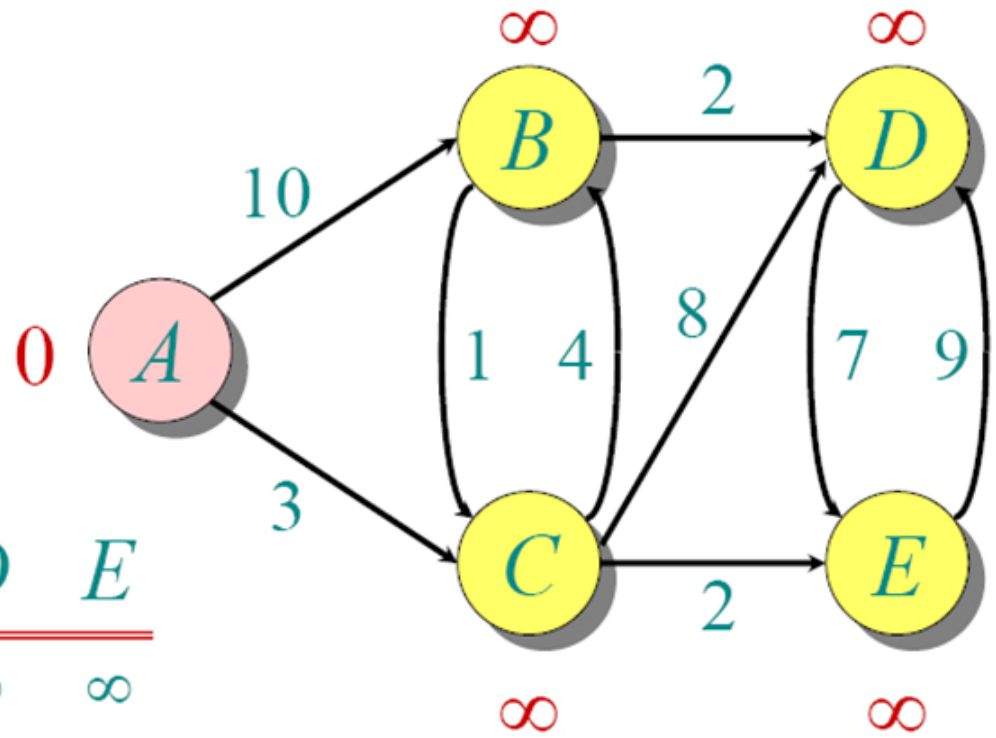


path:

Q:	A	B	C	D	E
dist:	0	∞	∞	∞	∞

S: {}

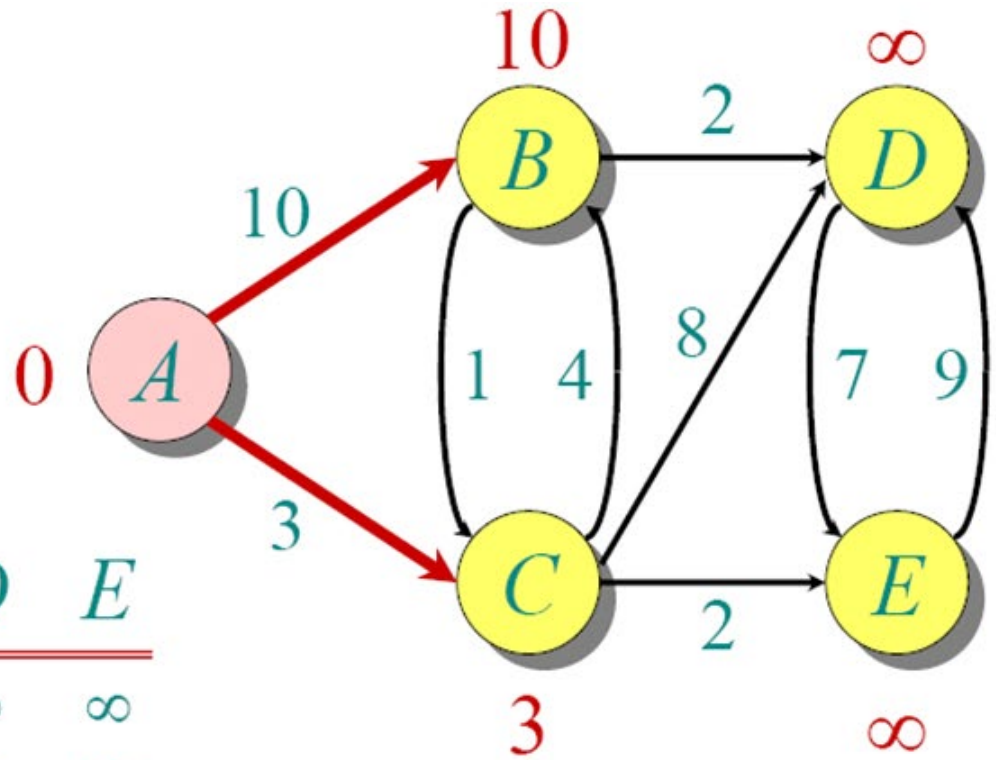
Dijkstra Animated Example



path:

Q:	A	B	C	D	E
dist:	0	∞	∞	∞	∞

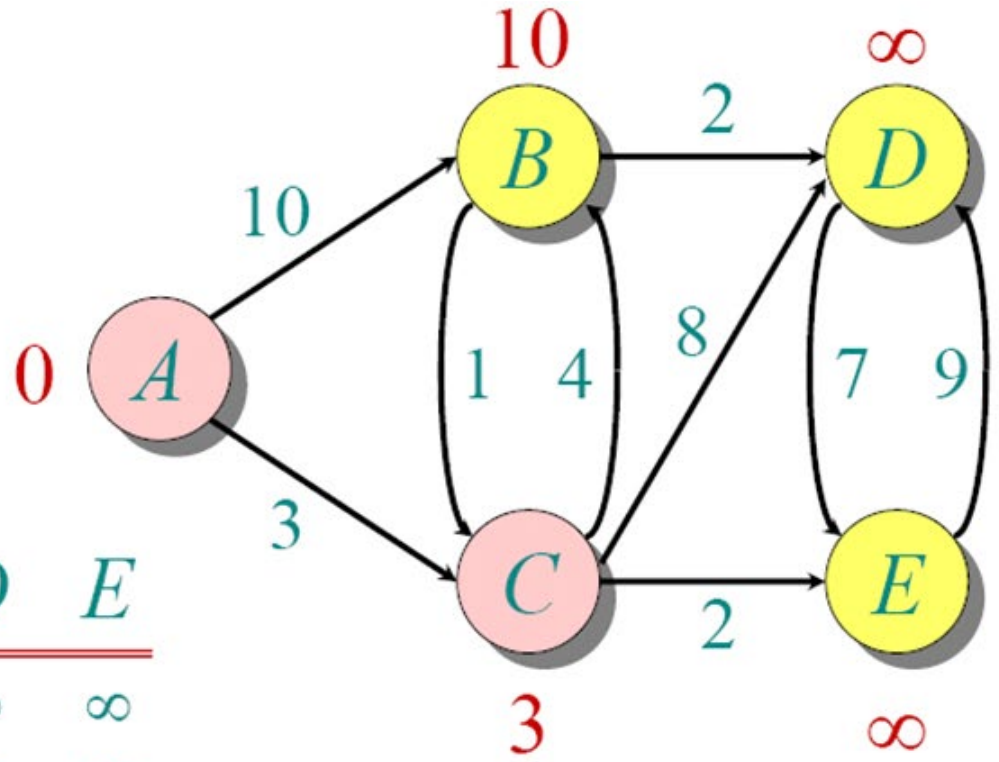
Dijkstra Animated Example



path:		A	A		
Q:	A	B	C	D	E
dist:	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

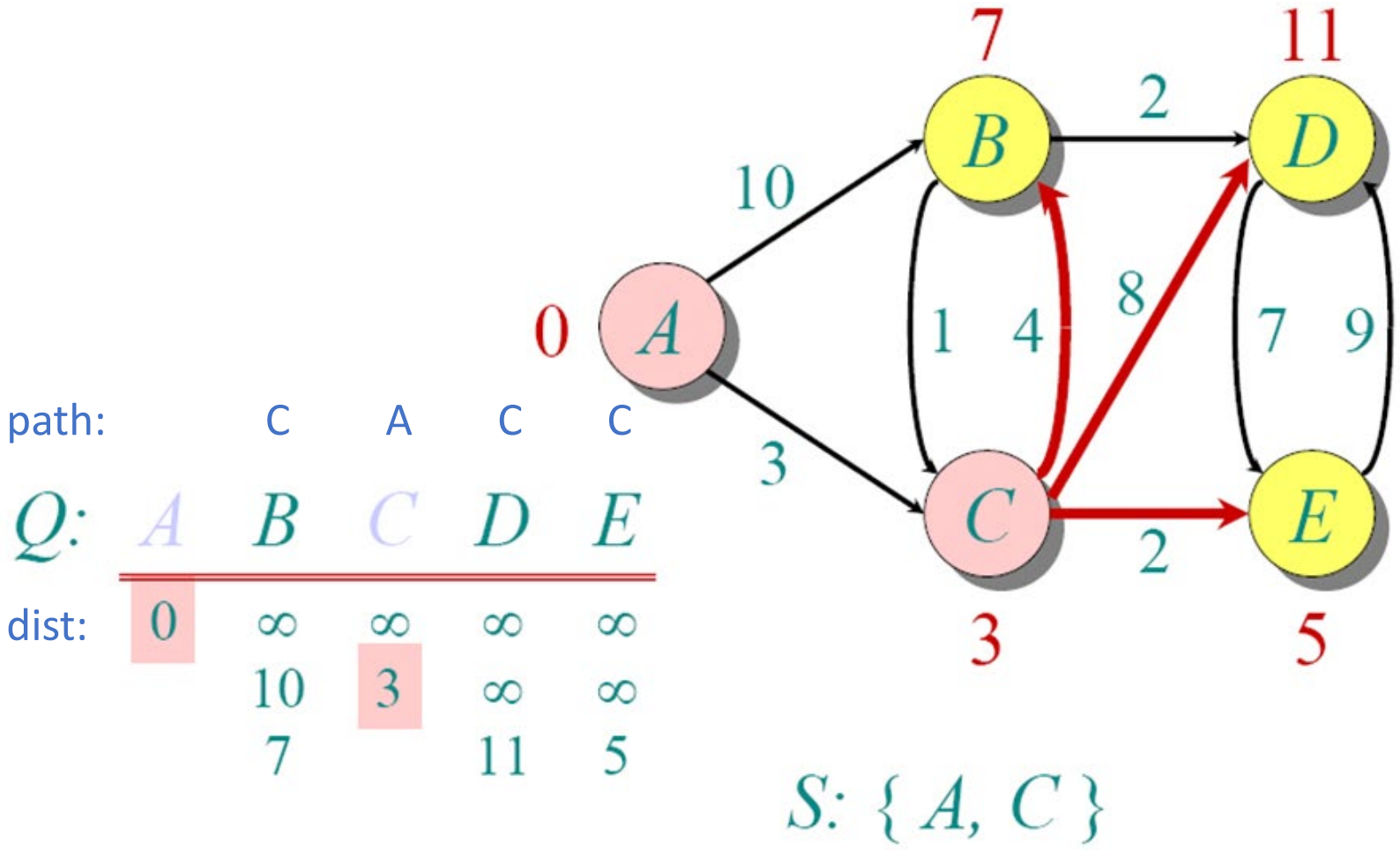
Dijkstra Animated Example



path:		A	A		
Q:	A	B	C	D	E
dist:	0	∞	∞	∞	∞
		10	3	∞	∞

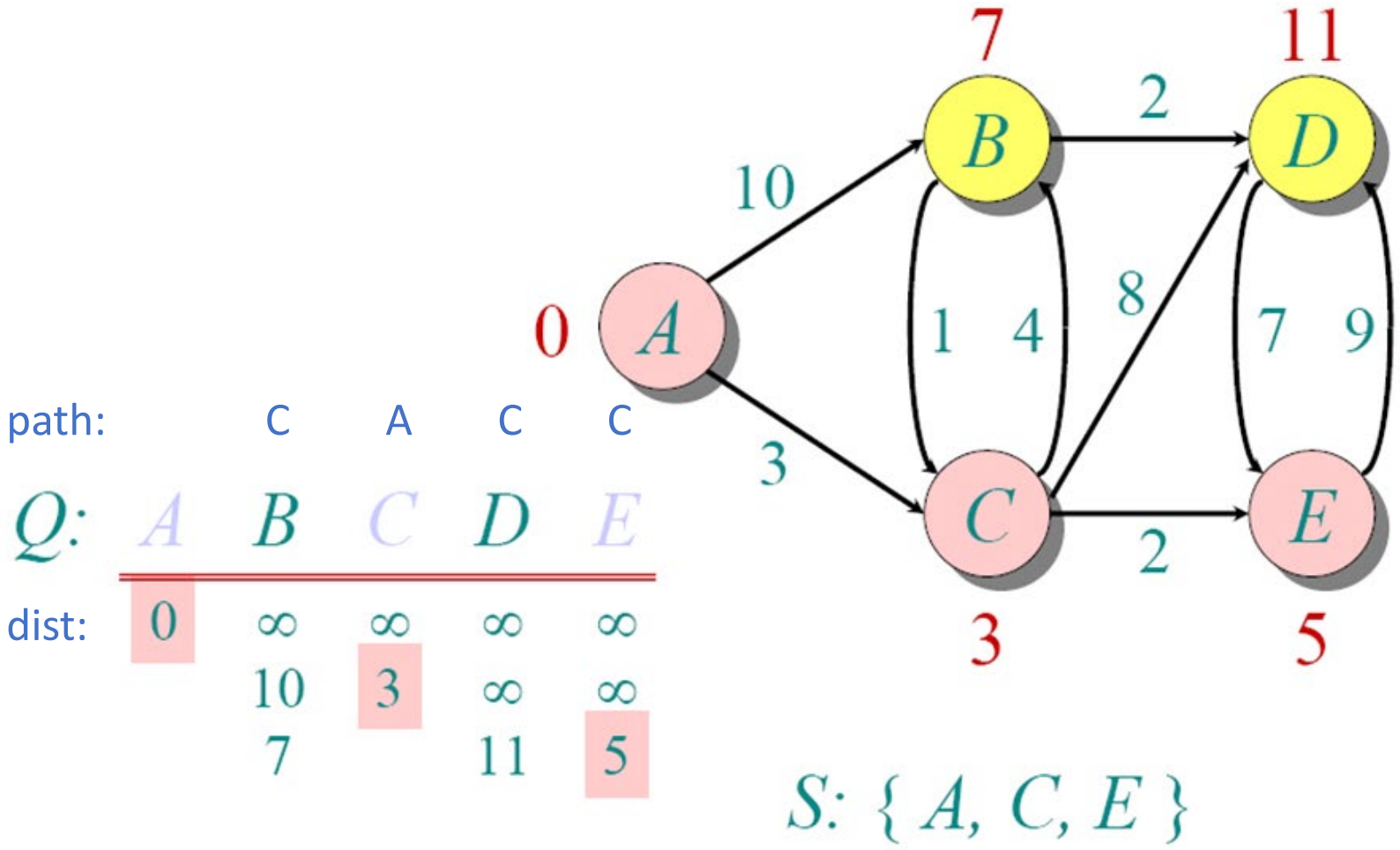
S: { A, C }

Dijkstra Animated Example

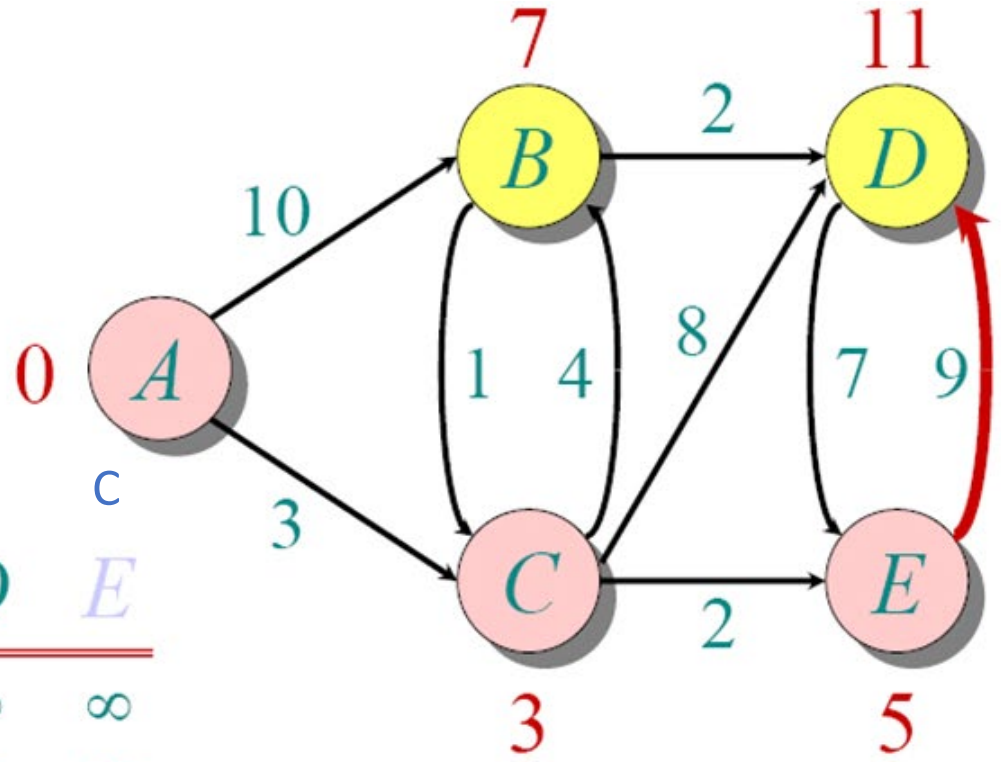


path:		C	A	C	C
Q:	A	B	C	D	E
dist:	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

Dijkstra Animated Example



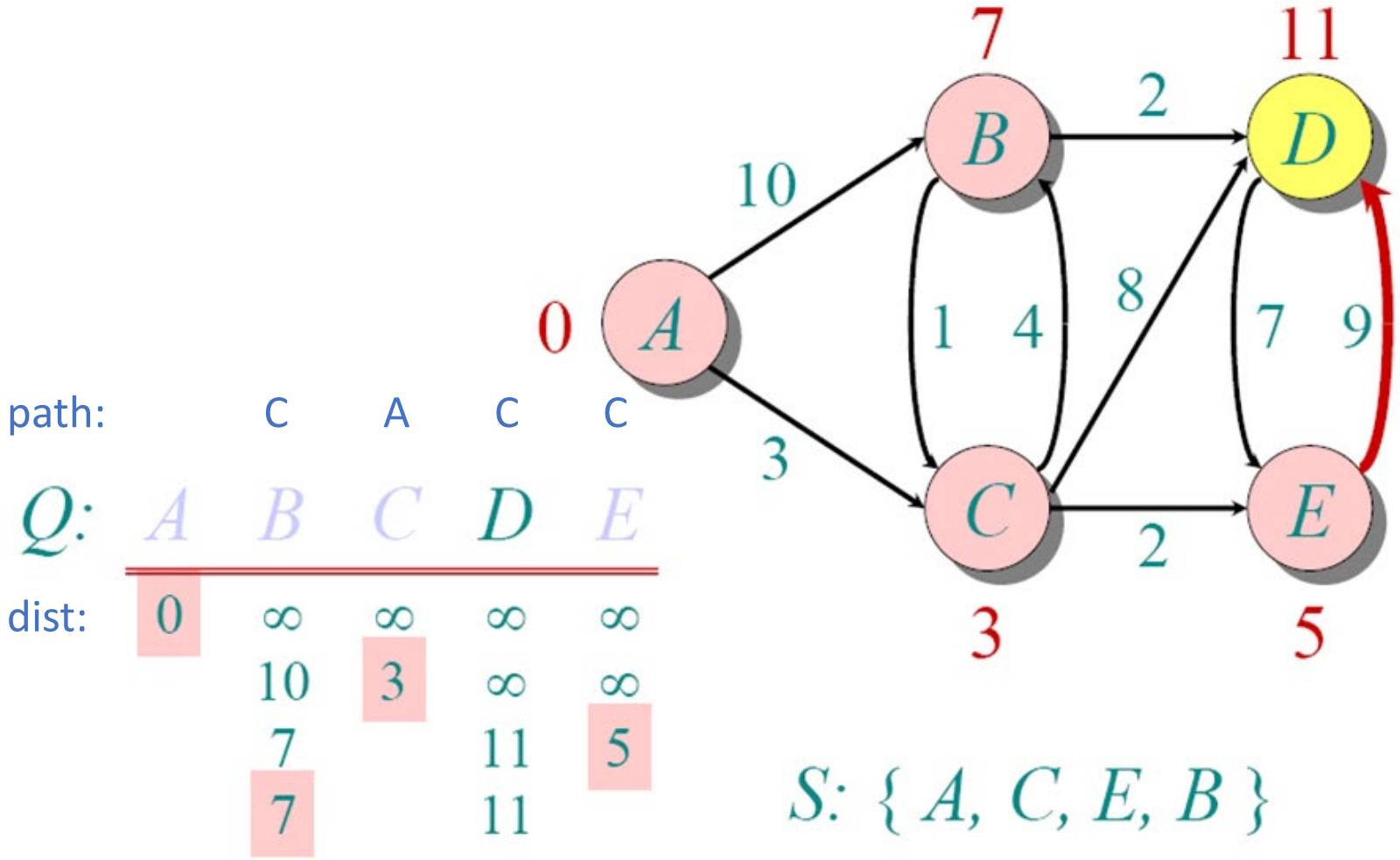
Dijkstra Animated Example



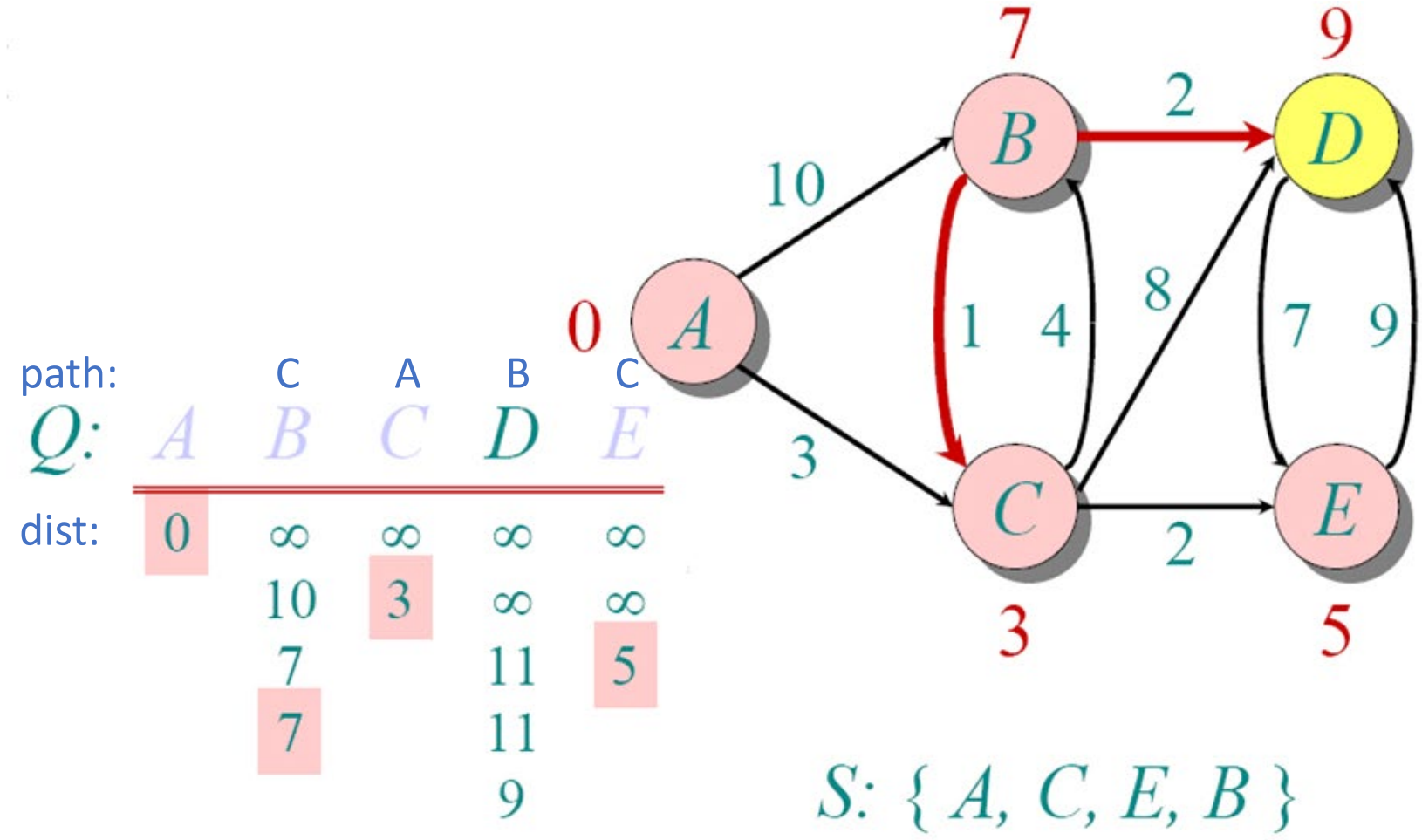
path:		C	A	C	C
Q:	A	B	C	D	E
dist:	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

$S: \{ A, C, E \}$

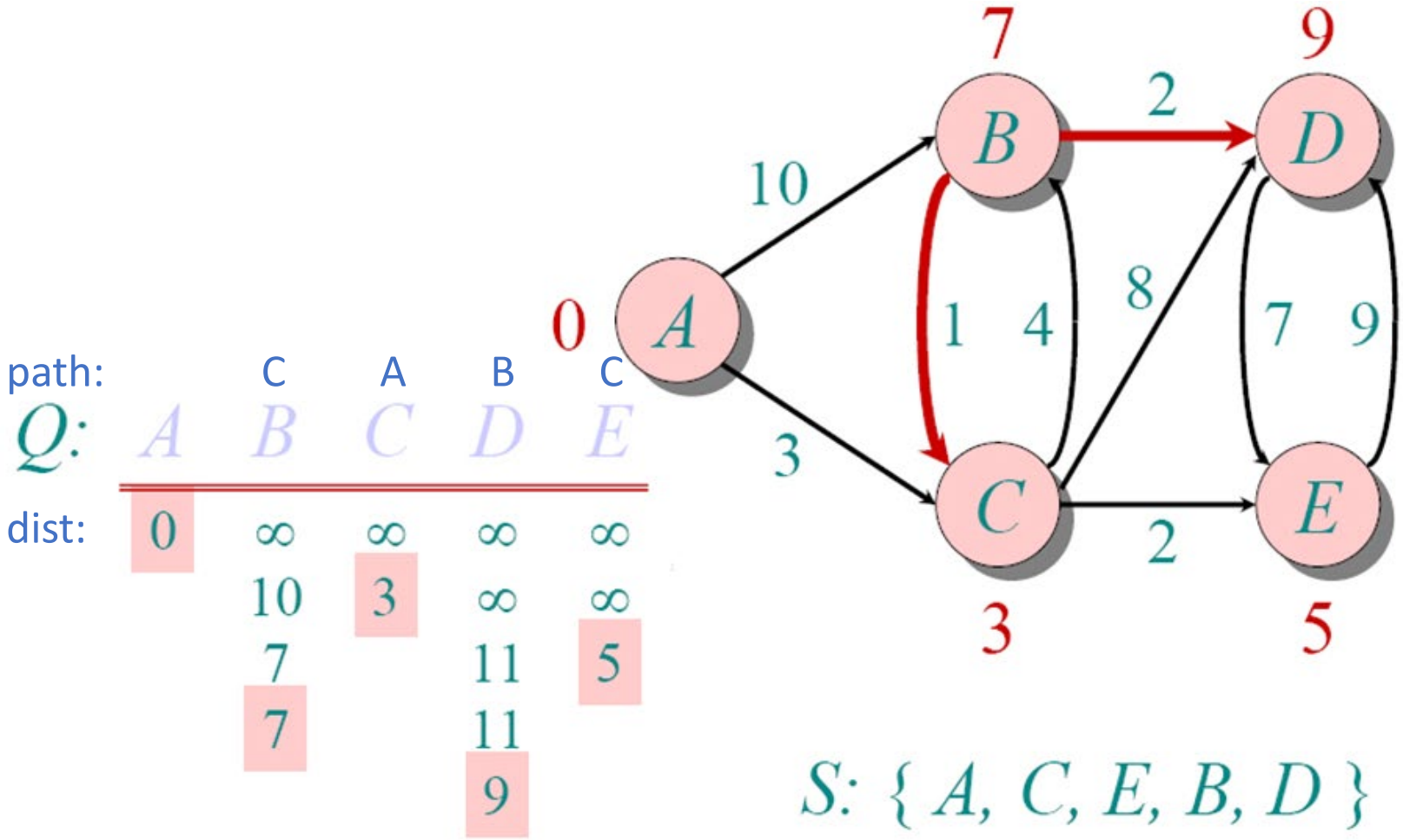
Dijkstra Animated Example



Dijkstra Animated Example



Dijkstra Animated Example



Dijkstra’s Algorithm Analysis

- Time Complexity of Dijkstra’s Algorithm is $O(V^2)$ but with min-priority queue it drops down to $O((V+E)\log V)$.

Dijkstra's Algorithm

```
dist[s] ←0
for  all v ∈ V–{s}
      do  dist[v] ←∞
S←∅
Q←V
while Q ≠∅
     u ← mindistance(Q,dist)
     S←S∪{u}
     for all v ∈ neighbors[u]
           do  if   dist[v] > dist[u] + w(u, v)
                    then     dist[v] ←dist[u] + w(u, v)
                             path[v] ←u
return dist, path
```

$O((V + E) \log V)$

$O(V)$

Executed V times

$O(\log V)$

$O(V \log V)$

$O(E)$ in total

$O(\log V)$

$O(E \log V)$

Proof of Dijkstra's Algorithm

- Prove by mathematical induction
  - Source node s, visited vertices set S
  - dist[v]: distance from s to v computed by Dijkstra's Algorithm
  - D[v]: the true shortest distance from s to v
  - Want to prove dist[v]=D[v]

Proof of Dijkstra's Algorithm

1. Base case: When S={s}, dist[s]=D[s]=0
2. Inductive hypothesis: For every vertex v in S, dist[v]=D[v]
3. Let u be the next vertex selected by Dijkstra's Algorithm to be added to S, we claim that dist[u]=D[u]
   - Otherwise, let $\pi(s,u)$ be a path from s to u that is shorter than dist[u]
   - Let x be the last node in $\pi(s,u)$ that is in S, y be the next node after x in $\pi(s,u)$ (so y is not in S)
     - Then $\pi(s,u) = \pi(s,x) + w(x,y) + \pi(y,u) < dist[u]$

$\pi(s,x) + w(x,y)$: V_I, $dist[x]$

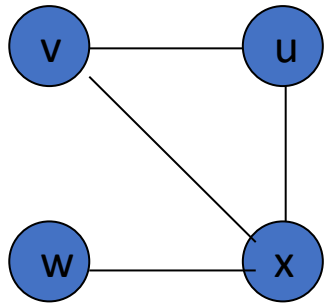
$\pi(y,u)$: V_I, $dist[y]$

Contradict to dist[u] is the minimum distance computed by Dijkstra's Algorithm for any node not in S

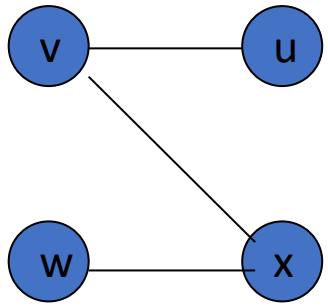
Minimum Spanning Tree

Spanning Tree

- Graph H=(V',E') is a spanning subgraph of G=(V,E) if V'=V.



graph



spanning tree

- Graph H is a spanning tree if
  - Is a spanning subgraph, and
  - Is a tree

Minimum Spanning Tree

- Edges are weighted
- Find minimum cost spanning tree
- Applications
  - Find cheapest way to wire your house
  - Find minimum cost to send a message on the Internet

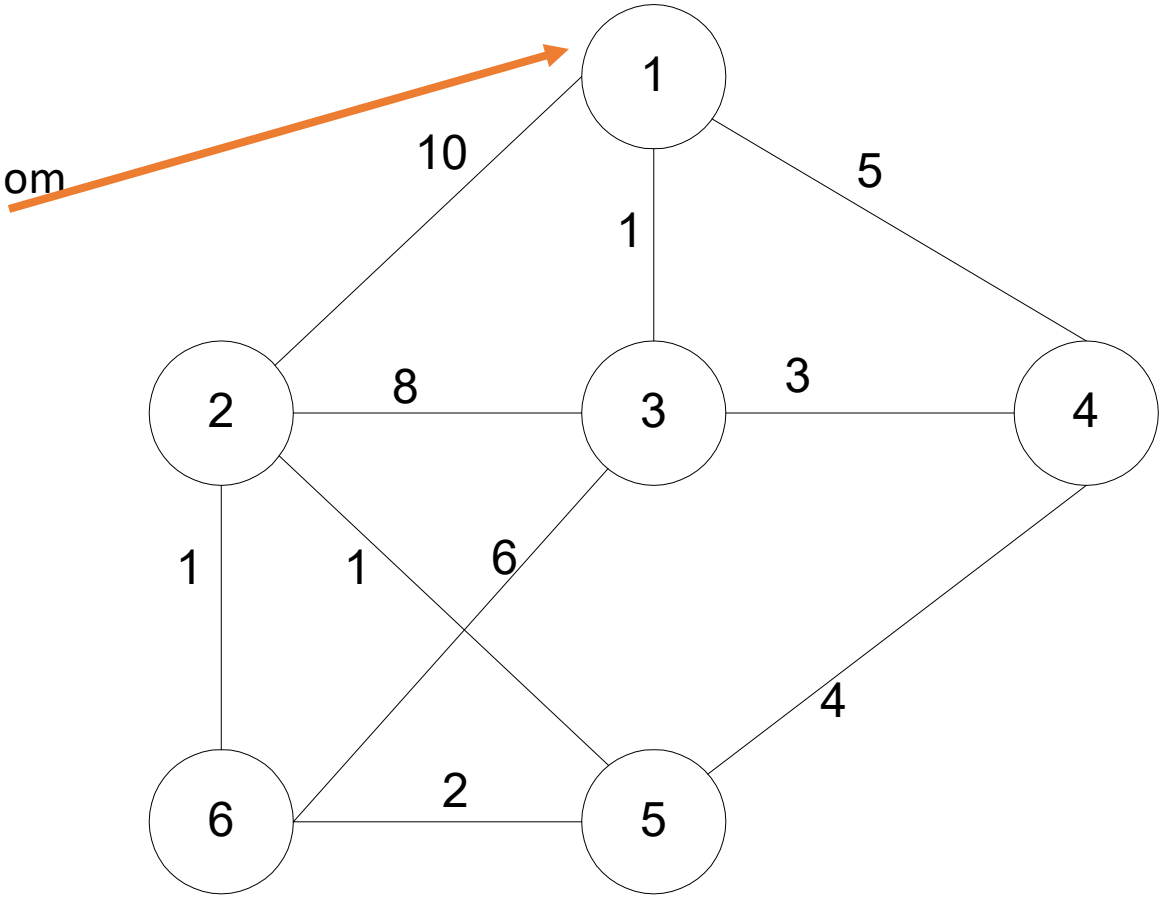
Prim’s Algorithm

- Approach: Greedy
- Build tree incrementally
  - Pick lowest cost edge connected to known (incomplete) spanning tree that does not create a cycle and expand to include it in the tree
- Prim’s greedy strategy produces a globally optimum solution (need to prove)

Prim's Algorithm

Starting from empty T,
choose a vertex at random
and initialize

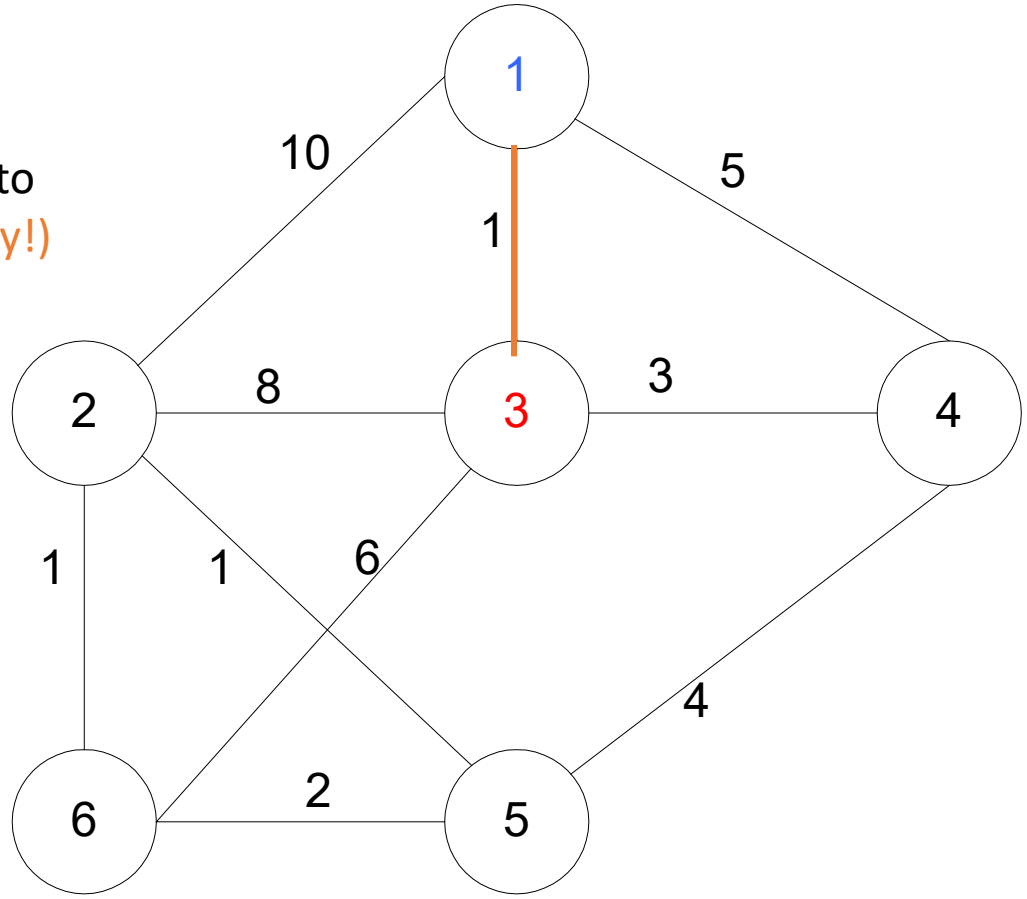
V = {1), E' ={}



Prim's Algorithm

Choose the vertex u not in V
such that edge weight from u to
a vertex in V is minimal (greedy!)

V={1,3} E'= {(1,3) }



Prim's Algorithm

Repeat until all vertices have been chosen

Choose the vertex u not in V such that edge weight from v to a vertex in V is minimal (greedy!)

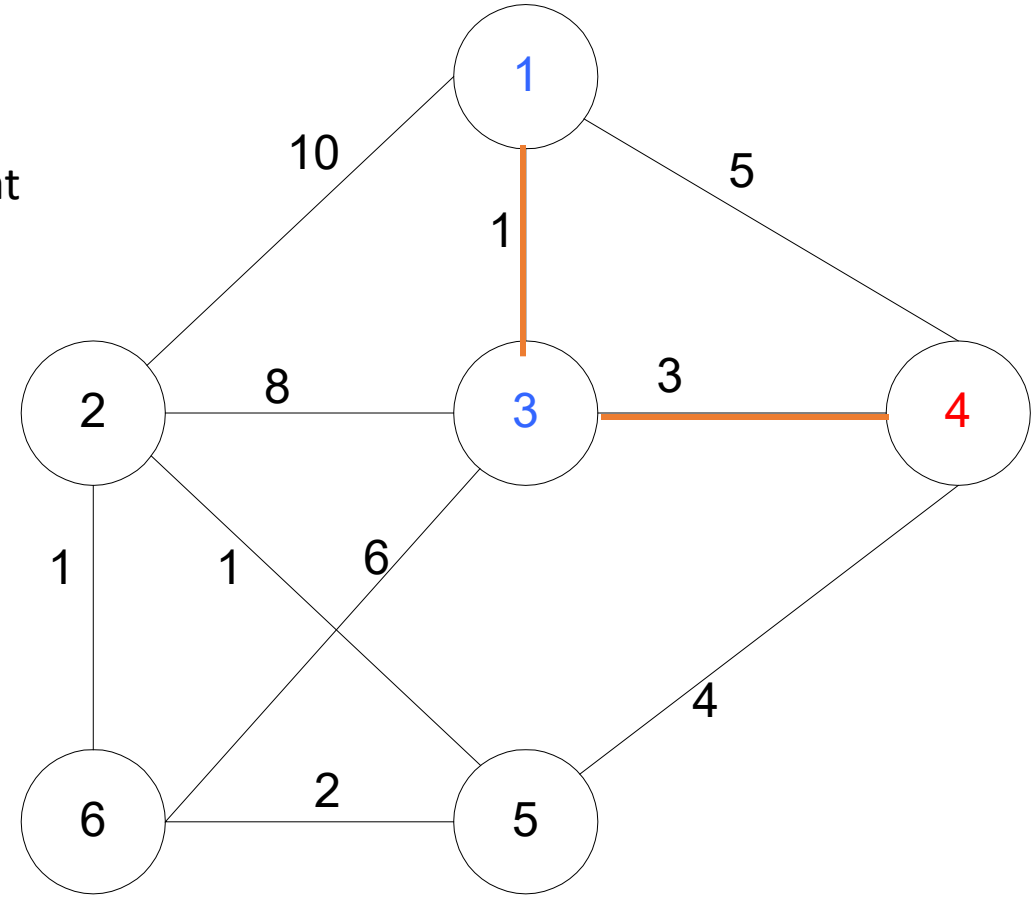
V= {1,3,4} E'= {(1,3),(3,4)}

V={1,3,4,5} E'={(1,3),(3,4),(4,5)}

....

V={1,3,4,5,2,6}

E'={(1,3),(3,4),(4,5),(5,2),(2,6)}



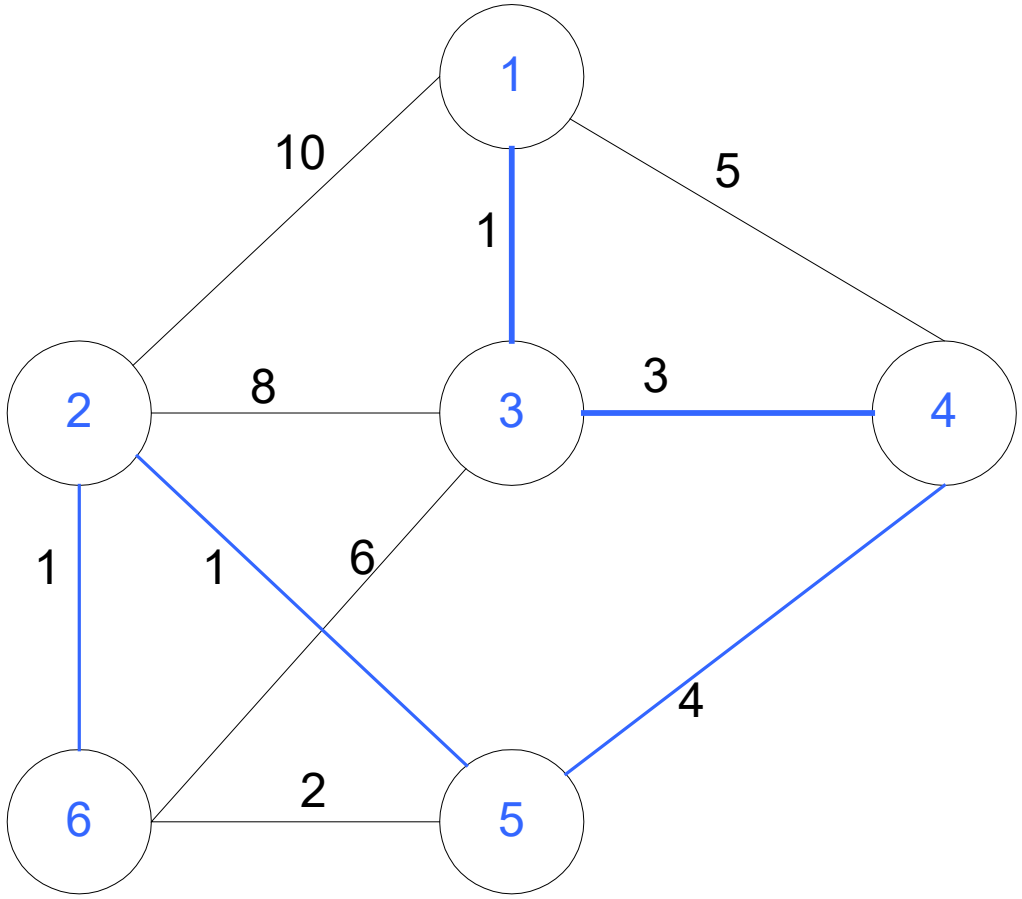
Prim's Algorithm

Repeat until all vertices have been chosen

V={1,3,4,5,2,6}

E'={(1,3),(3,4),(4,5),(5,2),(2,6)}

Final Cost: 1 + 3 + 4 + 1 + 1 = 10



Prim’s Algorithm

- If the “Select the unmarked node u with minimum cost” is done with min heap then O((V+E)logV)

PRIM(V, E, w, r)

```
1.  Q ← ∅
2.  for each u ∈ V
3.      do key[u] ← ∞
4.         π[u] ← NIL
5.         INSERT(Q, u)
6.  DECREASE-KEY(Q, r, 0)        ▶ key[r] ← 0
7.  while Q ≠ ∅
8.      do u ← EXTRACT-MIN(Q)
9.         for each v ∈ Adj[u]
10.            do if v ∈ Q and w(u, v) < key[v]
11.               then π[v] ← u
12.                    DECREASE-KEY(Q, v, w(u, v))
```

$O(V)$ if Q is implemented as a min-heap

Total time: $O(V\log V + E\log V) = O((V+E)\log V)$

$O(\log V)$

Executed $|V|$ times

Takes $O(\log V)$

Min-heap operations: $O(V\log V)$

Executed O(E) times total

Constant

Takes $O(\log V)$

$O(E\log V)$

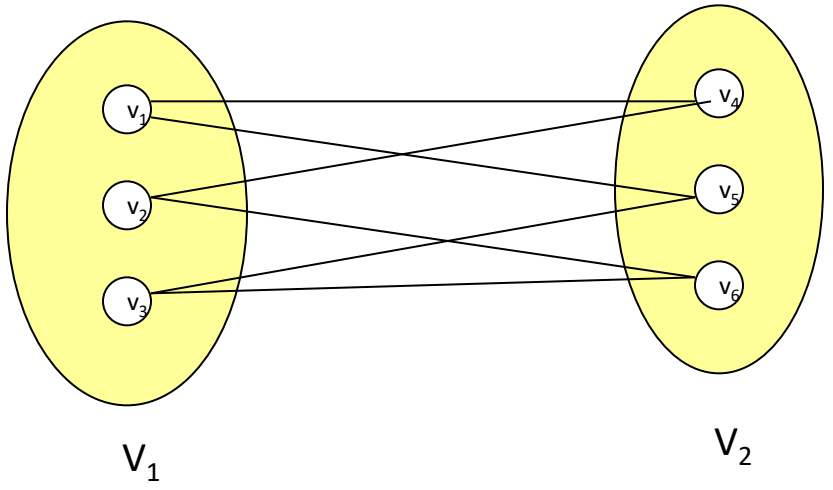
Bipartite Graph Matching

Bipartite Graphs

- In a simple graph G, if V can be partitioned into two disjoint sets V_1 and V_2 such that every edge in the graph connects a vertex in V_1 and a vertex V_2 (so that no edge in G connects either two vertices in V_1 or two vertices in V_2)

  Application example: Representing Relations

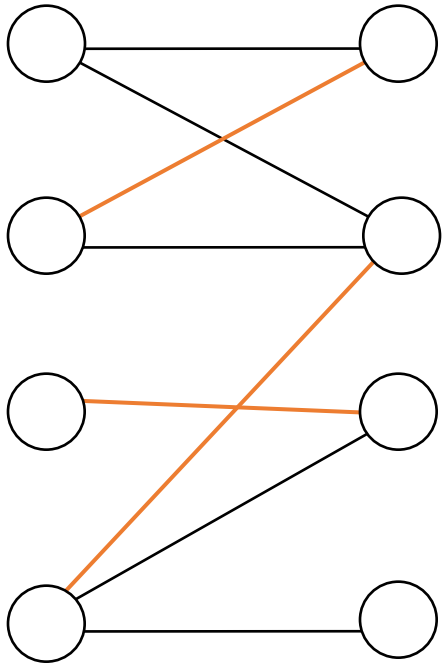
  Representation example: $V_1 = \{v_1, v_2, v_3\}$ and $V_2 = \{v_4, v_5, v_6\}$,



Graph Matching

- Graph G = (V,E)
- Graph *Matching* M is a subset of E
  - *if* e_1, e_2 distinct edges in M
  - *Then* they have no vertex in common

example



Vertex v is matched if v is in an edge of M

Matchings in Bipartite Graph

- Let G be a bipartite graph with parts *X* and *Y*.
- Complete matching from X to *Y*: Every vertex in *X* is incident with an edge of *M*.

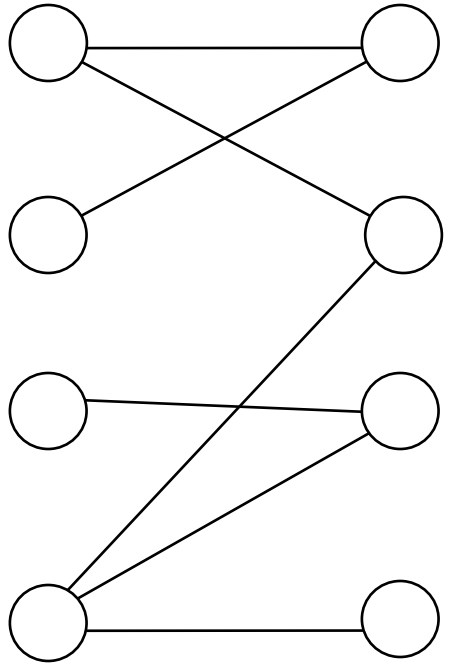
  (the direction is important, and |X|<|Y|)
- Perfect matching: *M* is complete from *X* to *Y* and *Y* to *X*. (|X|=|Y|)
- What applications can you think of?
  - Seat assignment to ensure harmony
  - Job assignment to maximize efficiency

Graph Matching Problem

- Maximum matching: M is a maximum matching for G if M has largest cardinality among all possible matchings.
- Maximal matching: A matching M that cannot be enlarged by adding additional edges, i.e. no larger M’ containing M.
- The goal here is to find a maximum matching

Greedy Strategy May Fail

- Greedy strategy: match any unmatched vertex in X with the next unmatched vertex in Y



Augmenting Path Given Graph Matching

- An *augmenting path* P = (e_1, e_2, …, e_k)

$$\text{require} \begin{cases} \text{begins and ends at} \\ \quad \text{unmatched vertices} \\ e_1, e_3, e_5, \ldots, e_k \in \text{E-M} \\ e_2, e_4, \ldots, e_{k-1} \in \text{M} \end{cases}$$

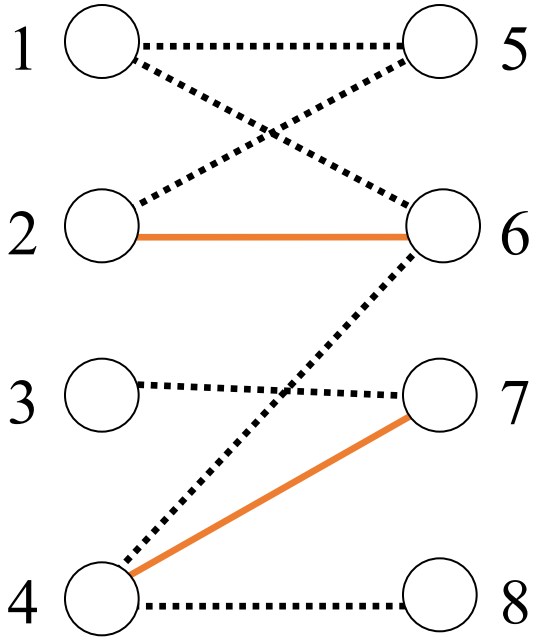
Augmentation

- Given an augmenting path, change its unmatched edges to matched and vice-versa, increasing the size of the matching by one

$$M' = P \oplus M = (P \cup M) - (P \cap M)$$

Graph Matching Example

- Initial matching M in G



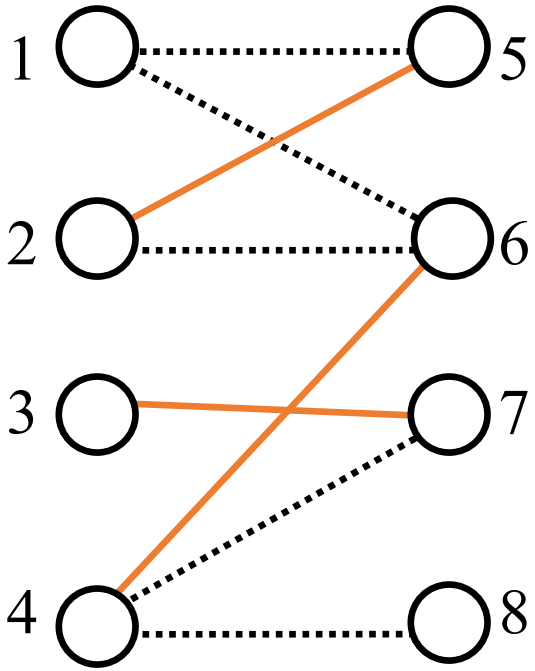
|M| = 2

- *Augmenting path*

  P = ((5,2), (2,6), (6,4), (4,7), (7,3))

Graph Matching Example

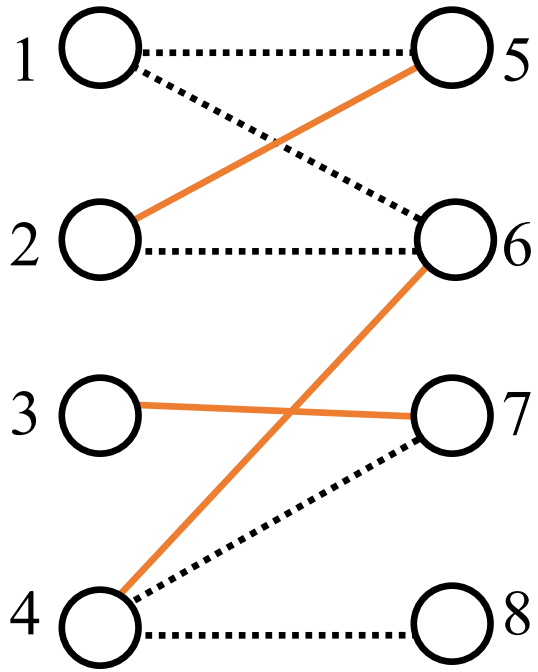
- *Augmenting path*
  $P = ((5,2), (2,6), (6,4), (4,7), (7,3))$
- New matching $M' = P \oplus M = (P \cup M) - (P \cap M)$



$|P \oplus M| = 3$

Graph Matching Example

- New matching M'



| M' | = 3

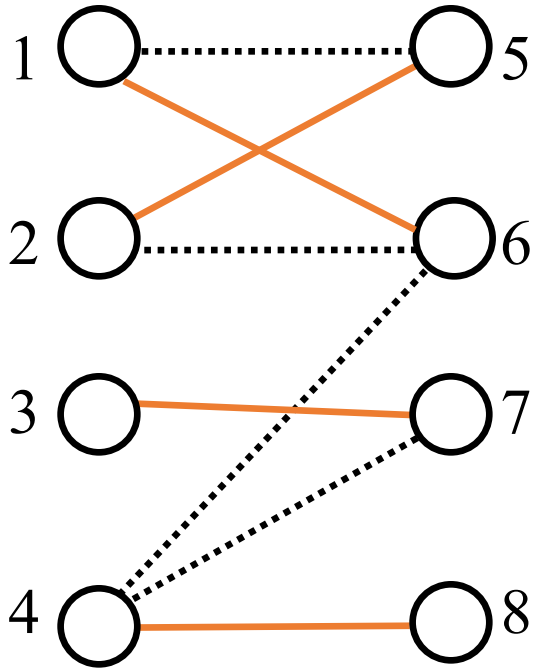
- *Augmenting path*

p = ((1,6), (6,4), (4,8))

Graph Matching Example

- *Augmenting path* p = ((1,6), (6,4), (4,8))

- Max matching M'' = P $\oplus$ M' = (P ∪ M') – (P ∩ M')



| M'' | = 4

Characterization of a Maximum Graph Matching via Lack of Augmented Path

- *Theorem:* M is *maximum* matching iff there is *no* augmenting path relative to M

- Proof: “iff”=“if and only if ” demands the proofs for both directions.

Proof of Characterization of Maximum Graph Matching

- *Proof*

  (-->) If M is maximum matching and p is an augmenting path for M, then M $\oplus$ P is a matching size > |M|

  (<--) (Prove by contrapositive)

  If M, M' are matchings with |M| < |M'|

  then there is an augmenting path.

Proof of Characterization of Maximum Graph Matching

- (<--) Construct graph G’ = (V, M $\oplus$ M’)
  - No vertex G’ is incident with more than 2 edges.
  - Has only paths with edges alternating between M and M’.
  - Since |M| < |M’|, should exist an alternating path with edges in M’ more than edges in M
    - An augmenting path

Maximum Matching Algorithm

- *Algorithm*

input graph $G = (V,E)$

[1] $M \leftarrow \{\}$

[2] *while* there exists an augmenting path p relative to M

do $M \leftarrow M \oplus P$

[3] *output* maximum matching M

Finding Augmented Paths

Remaining problem:

Find augmenting path

=> Use modified Breadth First Search

Breadth-First Search Algorithm for Augmented Path

- *Assume* G is *bipartite graph* with matching M.
- Use Breadth-First Search:

LEVEL(0) = some unmatched vertex r

For *odd L* > 0,

LEVEL(L) = {u | {v,u} ∈ E – M

when v ∈ LEVEL(L -1) and u in no lower level}

For *even L* > 0,

LEVEL(L) = {u | {v,u} ∈ M

when v ∈ LEVEL(L -1) and u in no lower level}

Breadth-First Search Algorithm for Augmented Path

- *Cases*

  (1) If for some odd L >0,

  LEVEL(L) contains an unmatched vertex u

  then the Breadth First Search tree T has an augmenting path from r to u

  (2) Otherwise no augmenting path exists, so M is maximum.

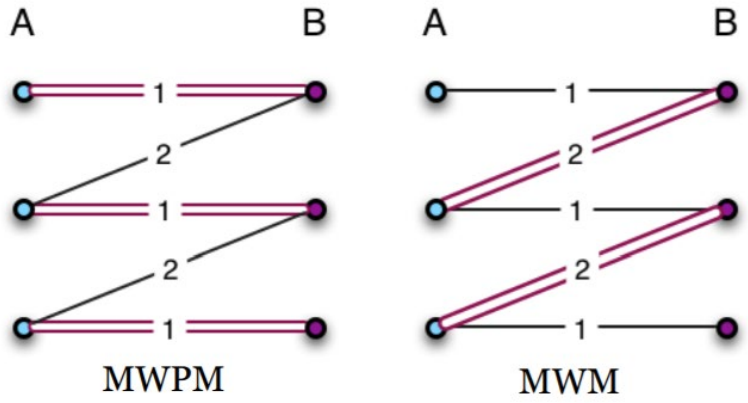
Time Complexity

- If G = (V,E) is a bipartite graph, then the maximum matching can be constructed in O(V(V+E)) time.

- Each stage requires O(V+E) time for Breadth First Search construction of augmenting path.

Weighted Bipartite Matching

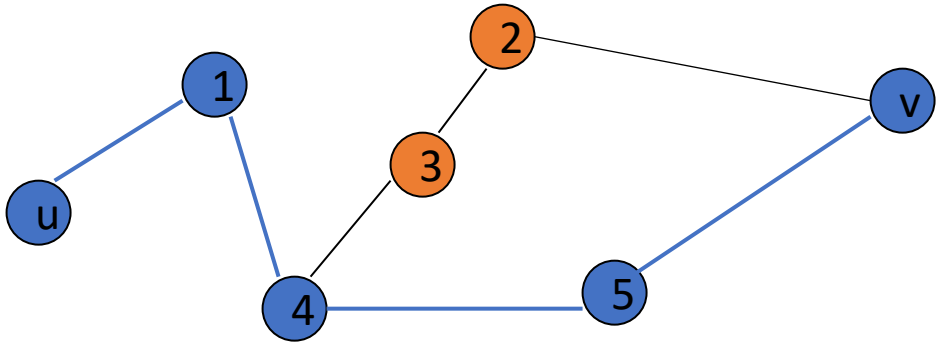
- Generalization: each edge has a weight
- Variants:
- Maximum (Minimum) weighted perfect matching
  - The maximum or minimum among all perfect matchings
- Maximum weighted matching
  - Not necessarily perfect
- Hungarian Algorithm



Euler Circuit

Connectivity – Path

A **Path** is a sequence of edges that begins at a vertex of a graph and travels along edges of the graph, always connecting pairs of adjacent vertices.



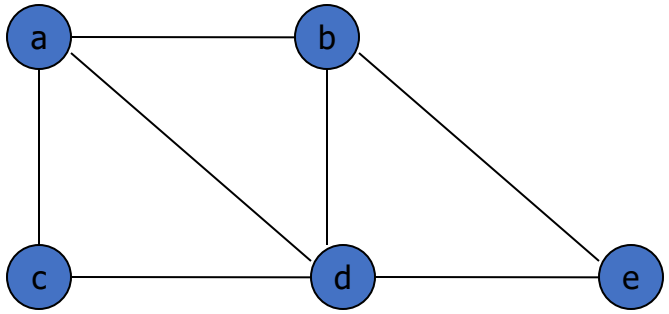
Circuit: u = v, length of path > 0

Euler - definitions

- An **Eulerian path** (**Eulerian trail**, **Euler walk**) in a graph is a path that uses each edge precisely once. If such a path exists, the graph is called **traversable**.

- An **Eulerian circuit** (**Eulerian cycle**, **Euler tour**) in a graph is a cycle that uses each edge precisely once. If such a cycle exists, the graph is called **Eulerian** (also **unicursal**).

- Representation example: G1 has Euler path a, c, d, e, b, d, a, b

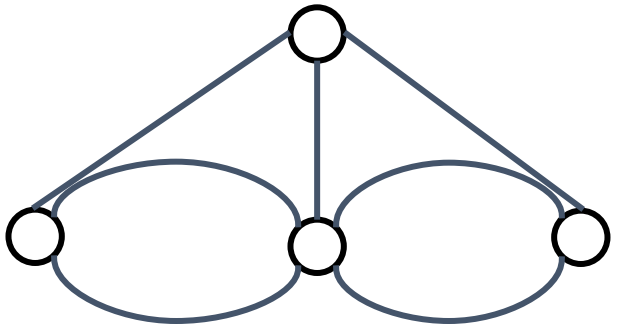


The Seven Bridges of Königsberg, Germany

- The residents of Königsberg, Germany, wondered if it was possible to take a walking tour of the town that crossed each of the seven bridges over the Presel river exactly once.
- Is it possible to start at some node and take a walk that uses each edge exactly once, and ends at the starting node?

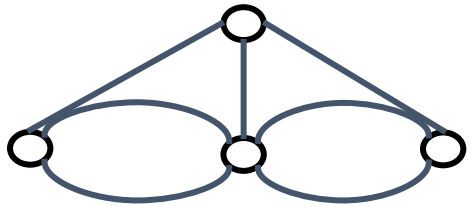
The Seven Bridges of Königsberg, Germany

Euler:



- Has no tour that uses each edge exactly once.
- (Even if we allow the walk to start and finish in different places.)
- Can you see why?

The problem in our language:

Show that  contains no Euler circuit.

Euler - theorems

- A connected graph G has an Euler circuit if and only if G has no vertices of odd degree

- A connected graph G is has an Euler path from vertex a to some other vertex b if and only if $a \neq b$ are the only two vertices of odd degree

Proof of Euler theorems

- Core idea: At every vertex other than the starting and ending point, we come into the vertex along one edge and go out along another; this can happen more than once, but since we cannot use edges more than once, the number of edges incident at such a vertex must be even.

1. If $a = b$, then a also has even degree. → Euler circuit

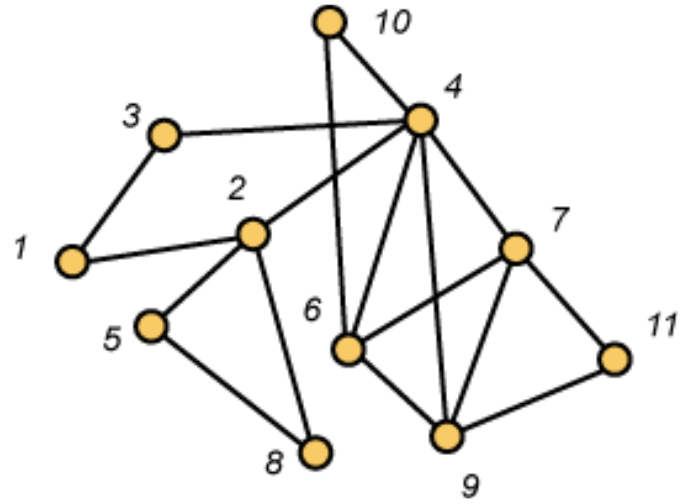
2. If $a \neq b$, then a and b both have odd degree. → Euler path

Algorithm for Euler Circuits

- Choose a root vertex r and start with the trivial partial circuit (r).
- Given a partial circuit $(r = x_0, x_1, \ldots, x_t = r)$ that traverses some but not all of the edges of G containing r, remove these edges from G. Let i be the least integer for which x_i is incident with one of the remaining edges. Form a greedy partial circuit among the remaining edges of the form $(x_i = y_0, y_1, \ldots, y_s = x_i)$.
- Expand the original circuit:

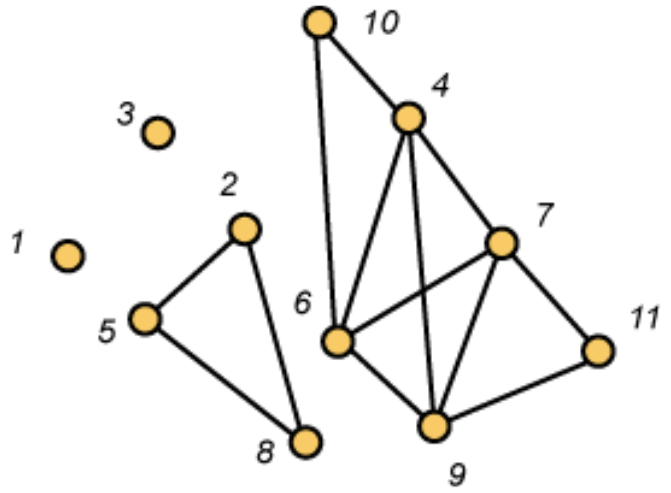
  $r = (x_0, x_1, \ldots, x_{i-1}, x_i = y_0, y_1, \ldots, y_s = x_i, x_{i+1}, \ldots, x_t = r)$

An Example



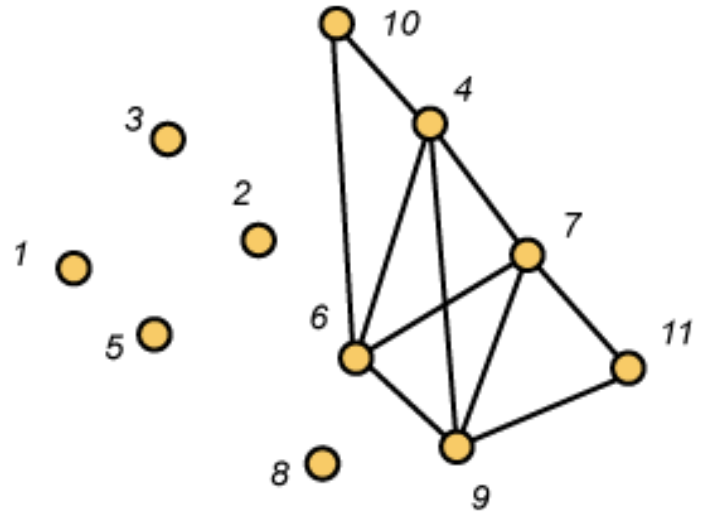
Start with the trivial circuit (1). Then the greedy algorithm yields the partial circuit (1,2,4,3,1).

Remove Edges and Continue



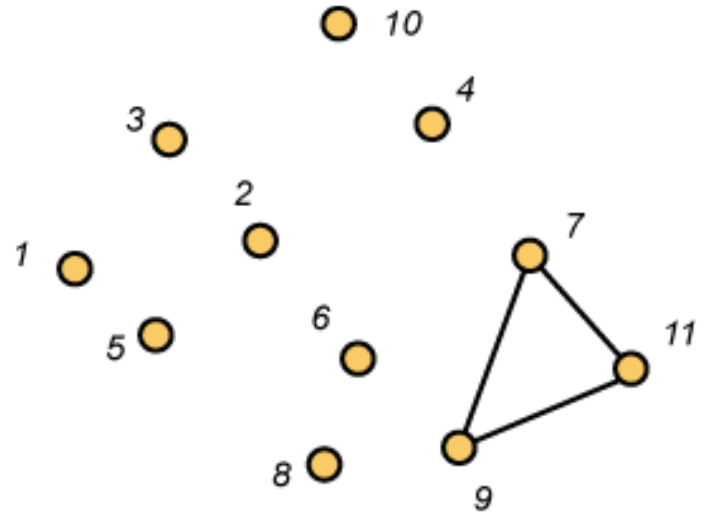
Start with the partial circuit (1,2,4,3,1). First vertex incident with an edge remaining is 2. A greedy approach yields (2,5,8,2). Expanding, we get the new partial circuit (1,2,5,8,2,4,3,1)

Remove Edges and Continue



Start with the partial circuit (1,2,5,8,2,4,3,1). First vertex incident with an edge remaining is 4. A greedy approach yields (4,6,7,4,9,6,10,4). Expanding, we get the new partial circuit (1,2,5,8,2,4,6,7,4,9,6,10,4,3,1)

Remove Edges and Continue



Start with the partial circuit (1,2,5,8,2,4,6,7,4,9,6,10,4,3,1) First vertex incident with an edge remaining is 7. A greedy approach yields (7,9,11,7). Expanding, we get the new partial circuit (1,2,5,8,2,4,6,7,9,11,7,4,9,6,10,4,3,1). This exhausts the edges and we have an Euler circuit.

Time Complexity

- The Euler circuit can be found in O(V+E) time.

Hamiltonian Cycle

- A Hamiltonian cycle/circuit is a cycle that visits each vertex exactly once (except for the starting vertex, which is visited once at the start and once again at the end).
- The *Hamiltonian cycle problem* is to find a Hamiltonian cycle in a given graph.

Hamiltonian Cycle