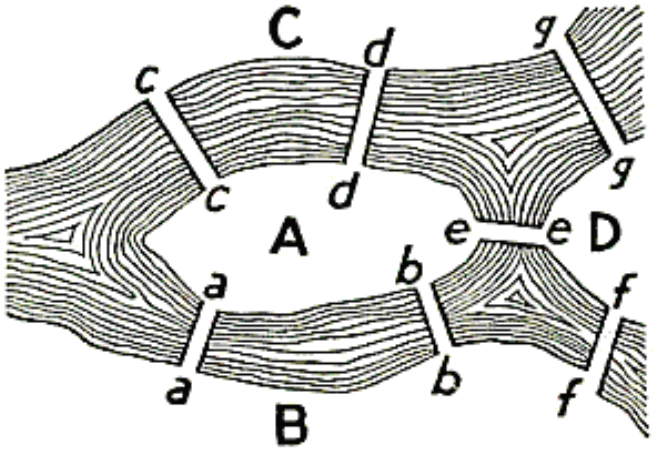
Classic Graph Theory Problems



Hiroki Sayama
sayama@binghamton.edu

The Origin

Königsberg bridge problem

Pregel River

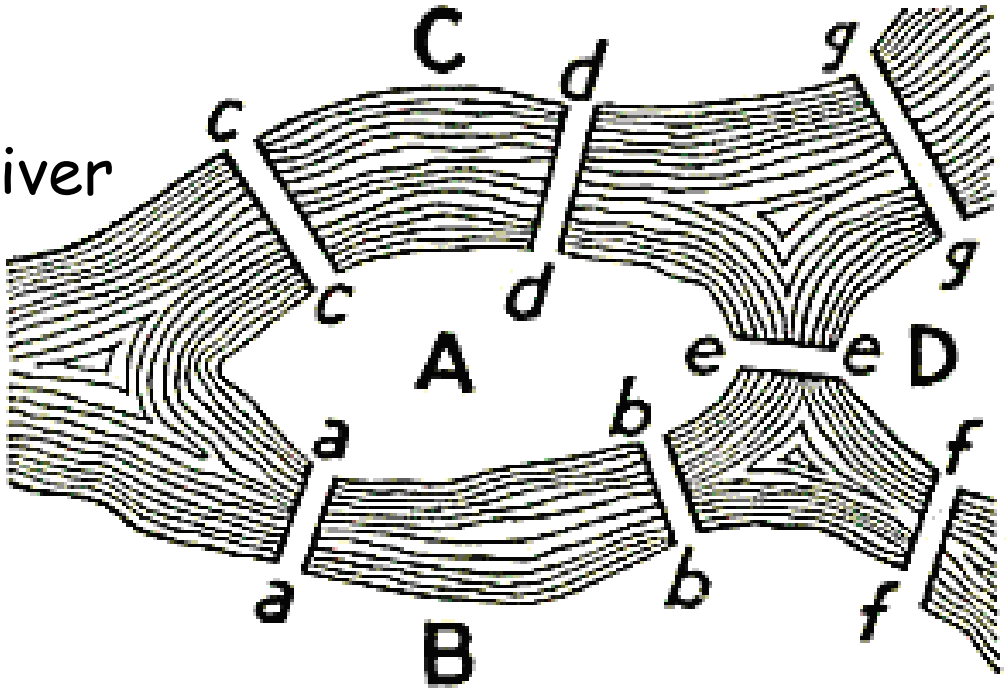
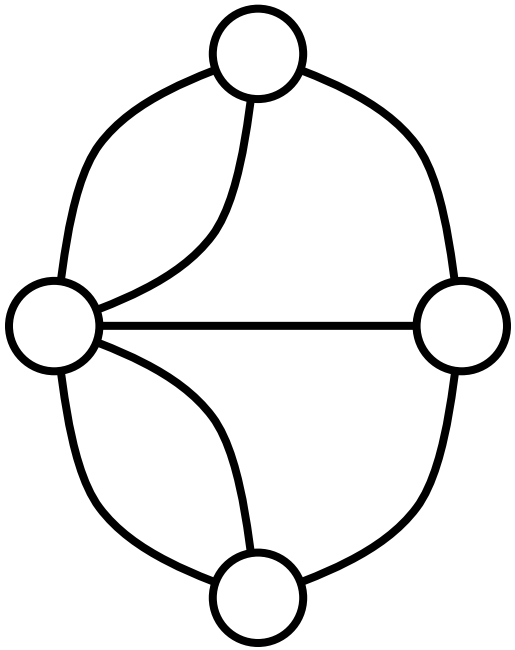


FIGURE 98. *Geographic Map: The Königsberg Bridges.*

(Solved negatively by Euler in 1736)

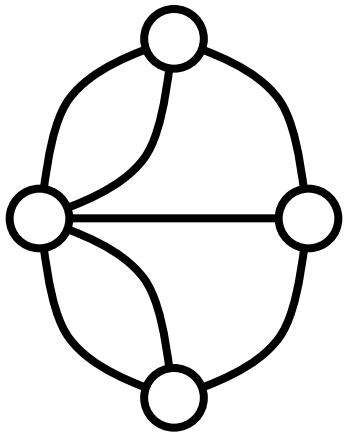
Representation in a graph



- Can all the seven edges be traversed in a single trail?

Theorem on the "Eulerian trail"

- An undirected connected graph has an "Eulerian trail" (a trail that includes every edge in a graph) if and only if it has at most two nodes of odd degree



This case has four nodes of odd degree

-> No Eulerian trail

Theorem on the "Eulerian trail"

- The condition is obviously necessary
- Why is it sufficient?
  - If all nodes have even degrees (called Eulerian graph), the graph must have a cyclic trail that includes every edge

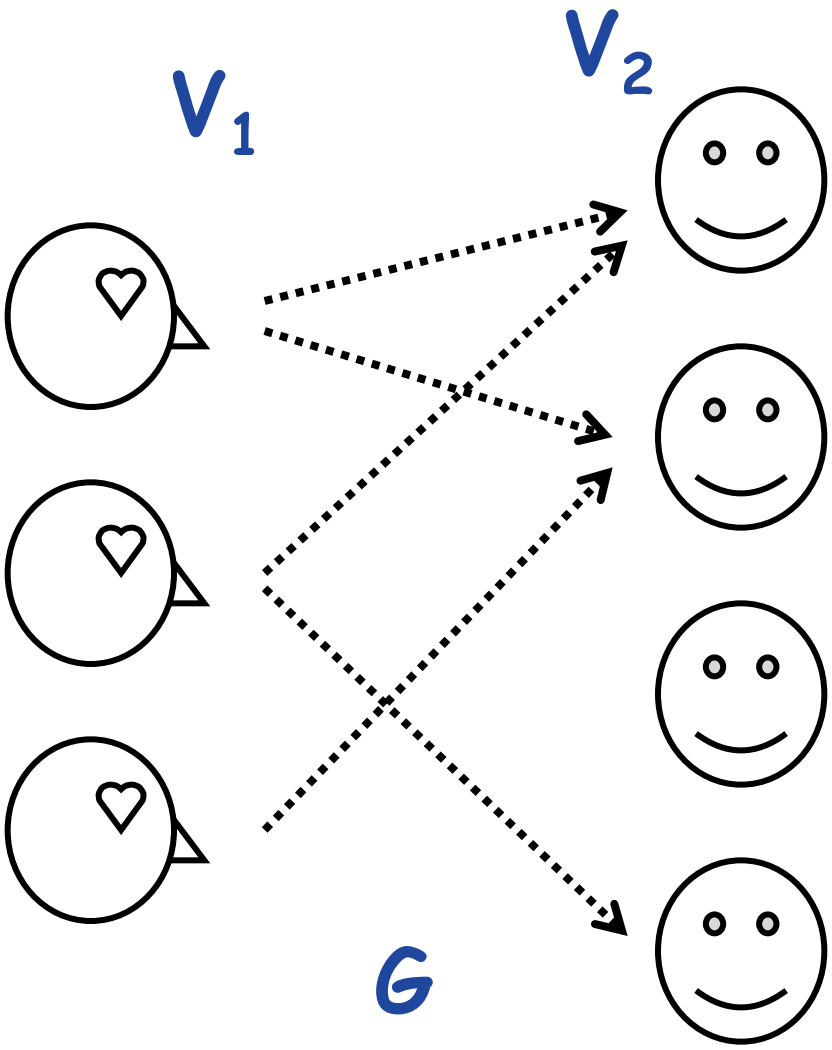
    (Easy to show that the longest trail must be cyclic and include every edge)
  - If there are two nodes with odd degrees (called semi-Eulerian graph), adding a new edge between them will reduce the problem to the above case

Implication

- Whether there exists an Eulerian trail in a graph can be determined by node degrees only
- Global property of a network is determined by local properties of nodes
- This is a highly exceptional case; doesn't happen in general

Matching Problem

Matching problem (bipartite)



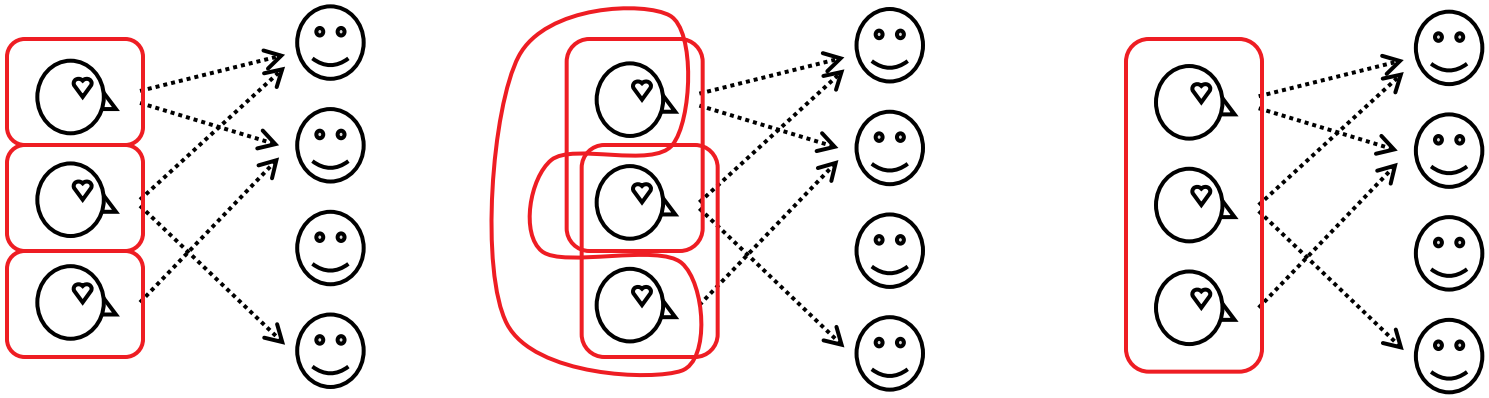
Does this bipartite graph G have a matching from V_1 to V_2?

Exercise

- A company announced five position openings (clerk, salesperson, programmer, system engineer, and designer); six job seekers applied to these positions
  - Applicant A looks for a clerk position; B for clerk or sales; C for sales, programming or SE; D for programming or design; and both E and F for design
- Can this company fill in all the positions?

A marriage theorem

- A bipartite graph G has a perfect matching from V_1 to V_2 if and only if every subset of V_1 is collectively connected to at least as many nodes in V_2 as itself

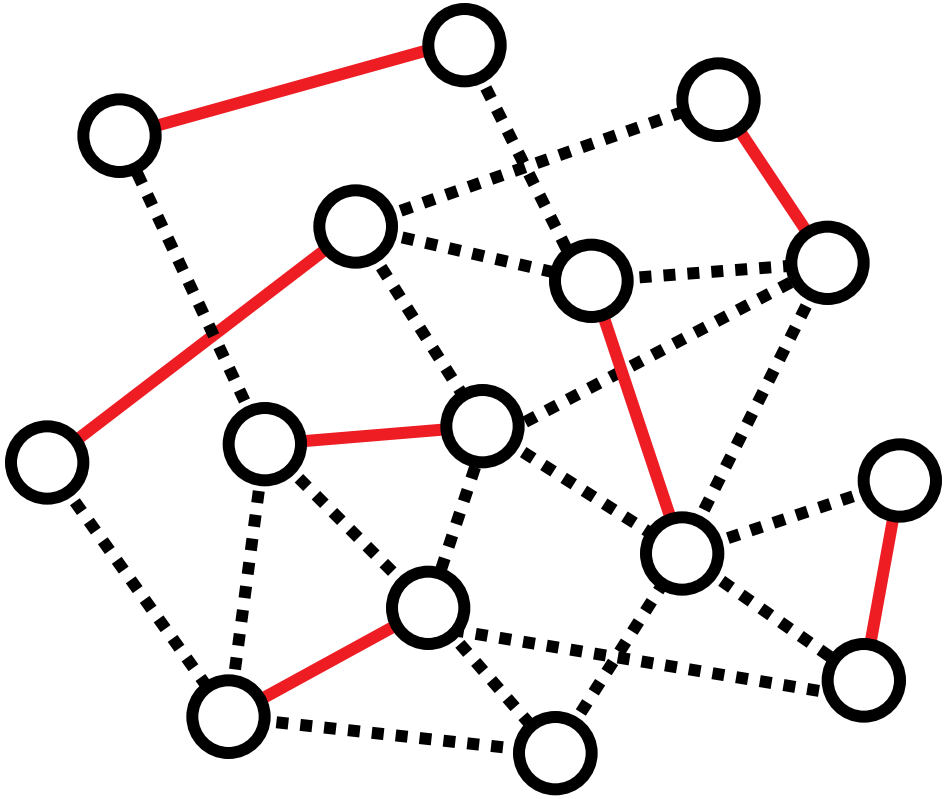


Exercise

- The condition is obviously necessary
- Why is it sufficient?
  (think about it yourself)

Matching problem (non-bipartite)

- **Matching:** Subset of edges that do not share any nodes



- What is the size of maximal matching?
- Is perfect matching possible?

Shortest Path Finding

Shortest path finding problem

- Given two nodes for start and goal, determine which path is the shortest one between these two nodes
  - Applications can be found in Google Maps, car navigation systems, train transit search systems, etc.

    (Weights may be distance, time, or fare)

Dijkstra's algorithm

- E. Dijkstra (1959)

- Finds the shortest paths from a given node to every other node in a graph
- Works with both undirected/directed, unweighted/weighted graphs
- Known to be the best among several shortest path finding algorithms

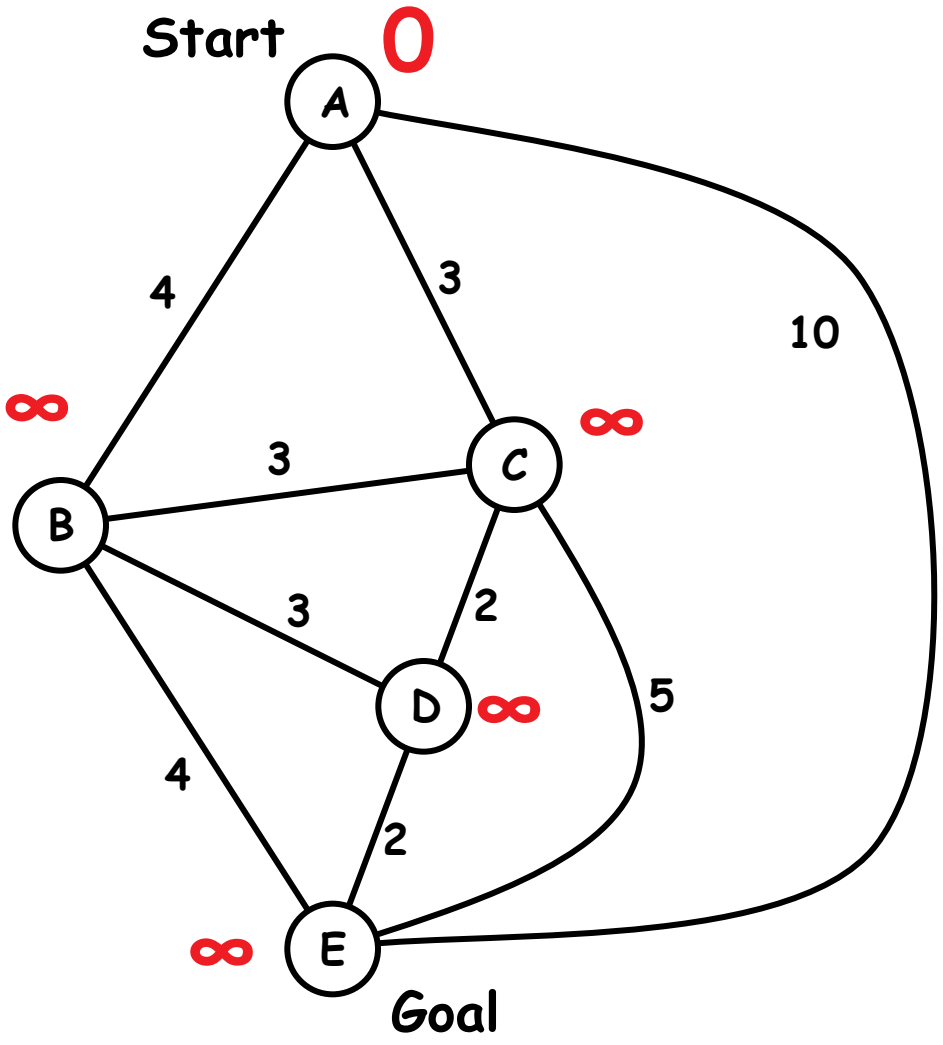
Dijkstra's algorithm (1)

- w(e): weight of edge e
- u_s : start node, u_g : goal node
- L(u): a label assigned to node u that represents the tentative shortest distance from u_s to u
  - Initially $L(u=u_s)=0$, $L(u\neq u_s)=\infty$
- T: a set of nodes for which the shortest paths are not yet determined
  - Initially T={ all nodes }

Dijkstra's algorithm (2)

1. Find a node with the smallest L in T (call it v hereafter)
2. If $v=u_g$, return L(v) as an answer
3. Otherwise, for every edge e=<v, x>: If x is in T and L(x) > L(v) + w(e), then let L(x) = L(v) + w(e)
4. Remove v out of T and go back to 1

Example



Edges:

A -> B, C, E

B -> A, C, D, E

C -> A, B, D, E

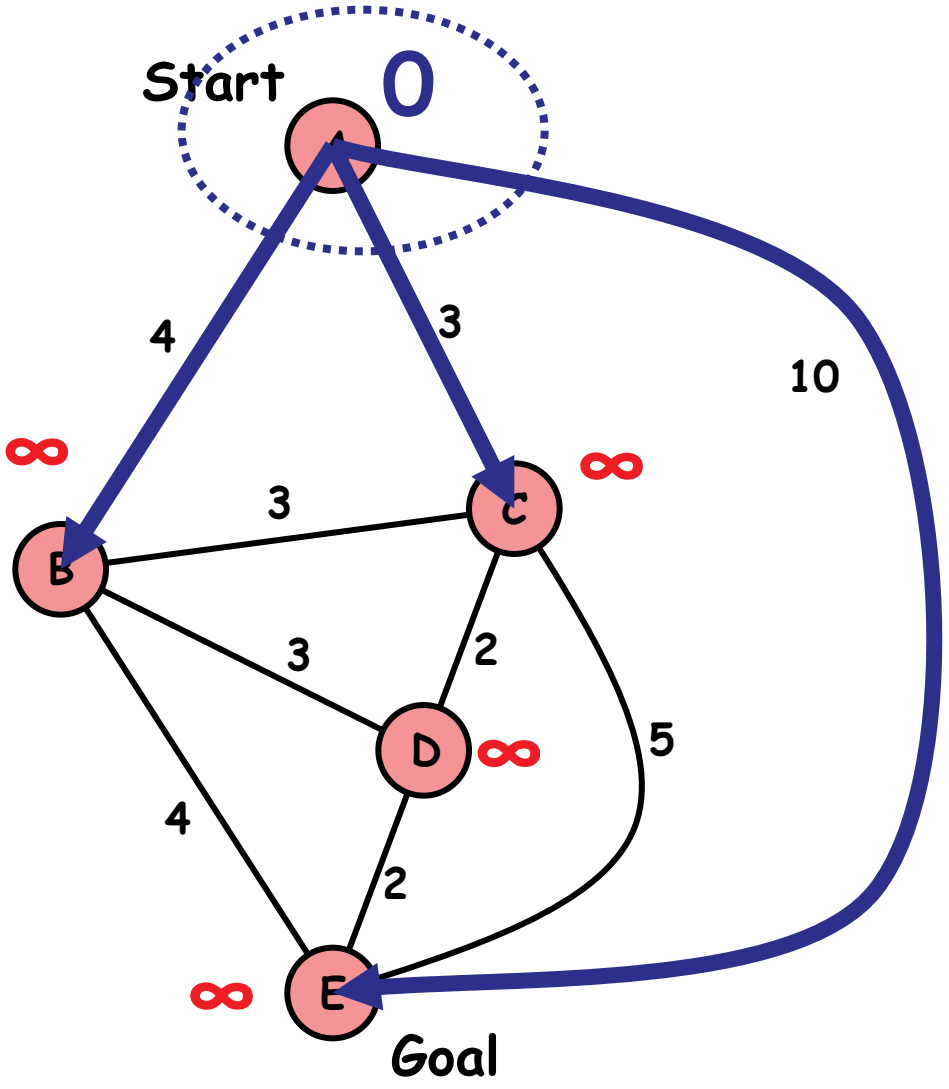
D -> B, C, E

E -> A, B, C, D

Contents of T:

{A, B, C, D, E}

Example



Edges:

A -> B, C, E

B -> A, C, D, E

C -> A, B, D, E

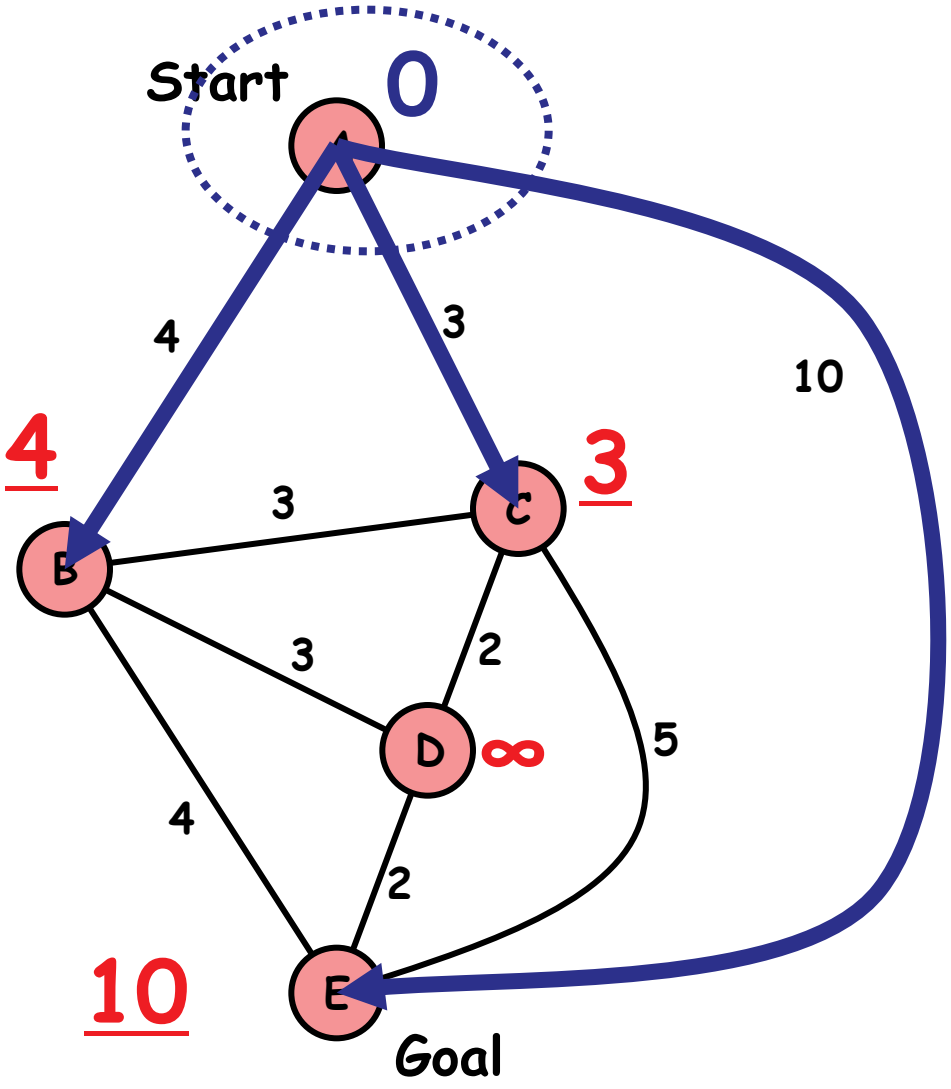
D -> B, C, E

E -> A, B, C, D

Contents of T:

{A, B, C, D, E}

Example



Edges:

A -> B, C, E

B -> A, C, D, E

C -> A, B, D, E

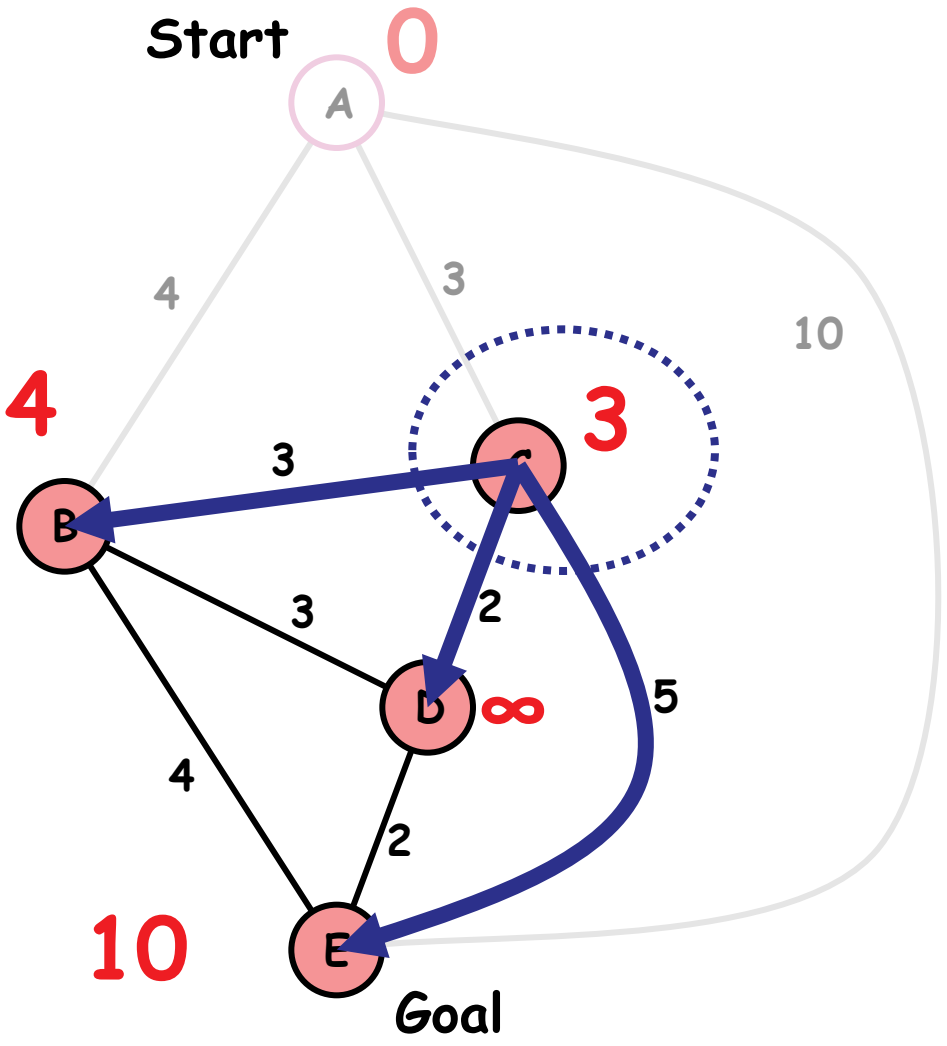
D -> B, C, E

E -> A, B, C, D

Contents of T:

{A, B, C, D, E}

Example



Edges:

A -> B, C, E

B -> A, C, D, E

C -> A, B, D, E

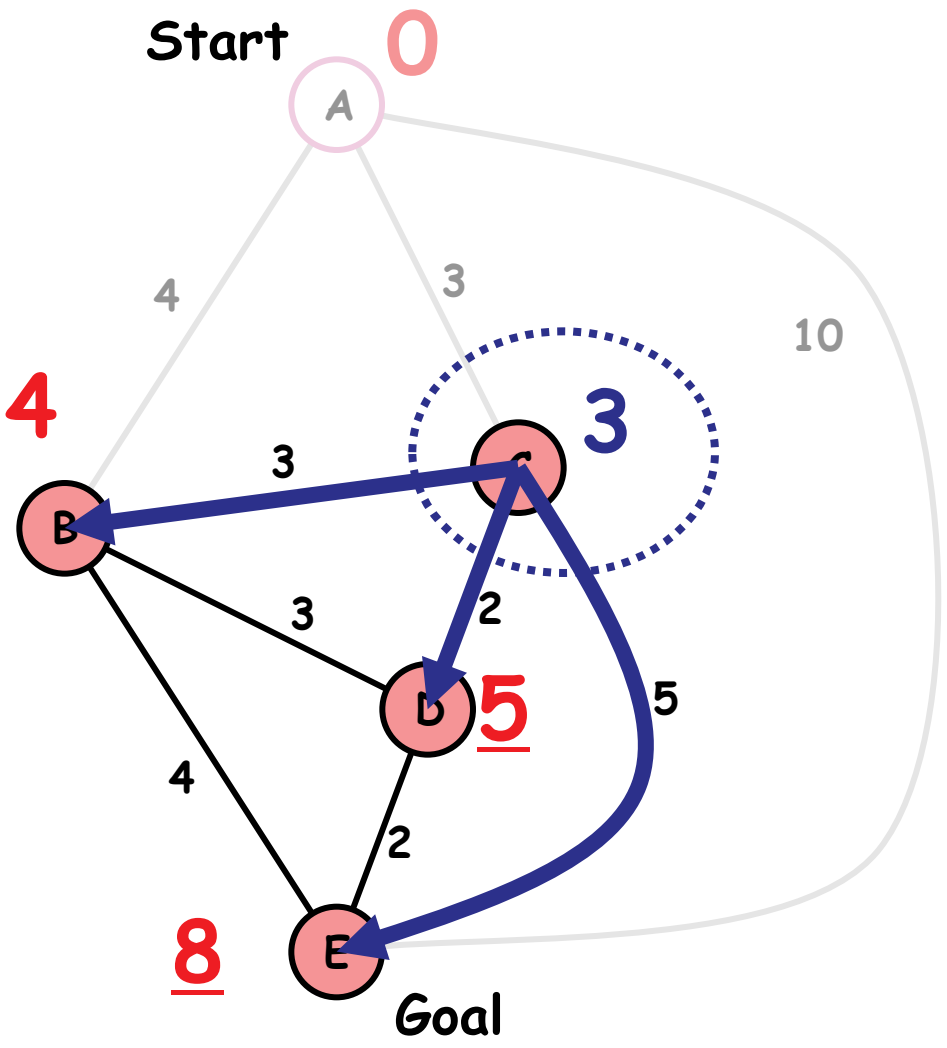
D -> B, C, E

E -> A, B, C, D

Contents of T:

{B, C, D, E}

Example



Edges:

A -> B, C, E

B -> A, C, D, E

C -> A, B, D, E

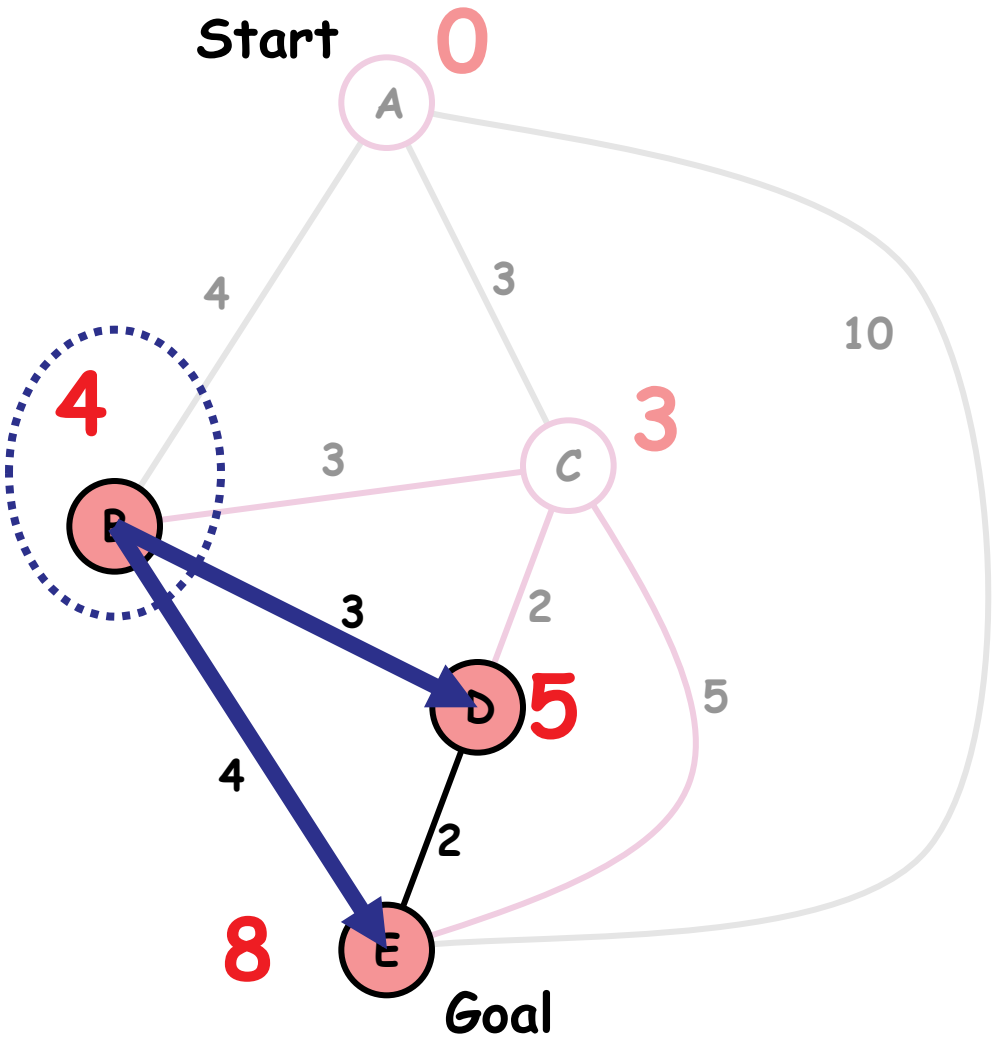
D -> B, C, E

E -> A, B, C, D

Contents of T:

{B, C, D, E}

Example



Edges:

A -> B, C, E

B -> A, C, D, E

C -> A, B, D, E

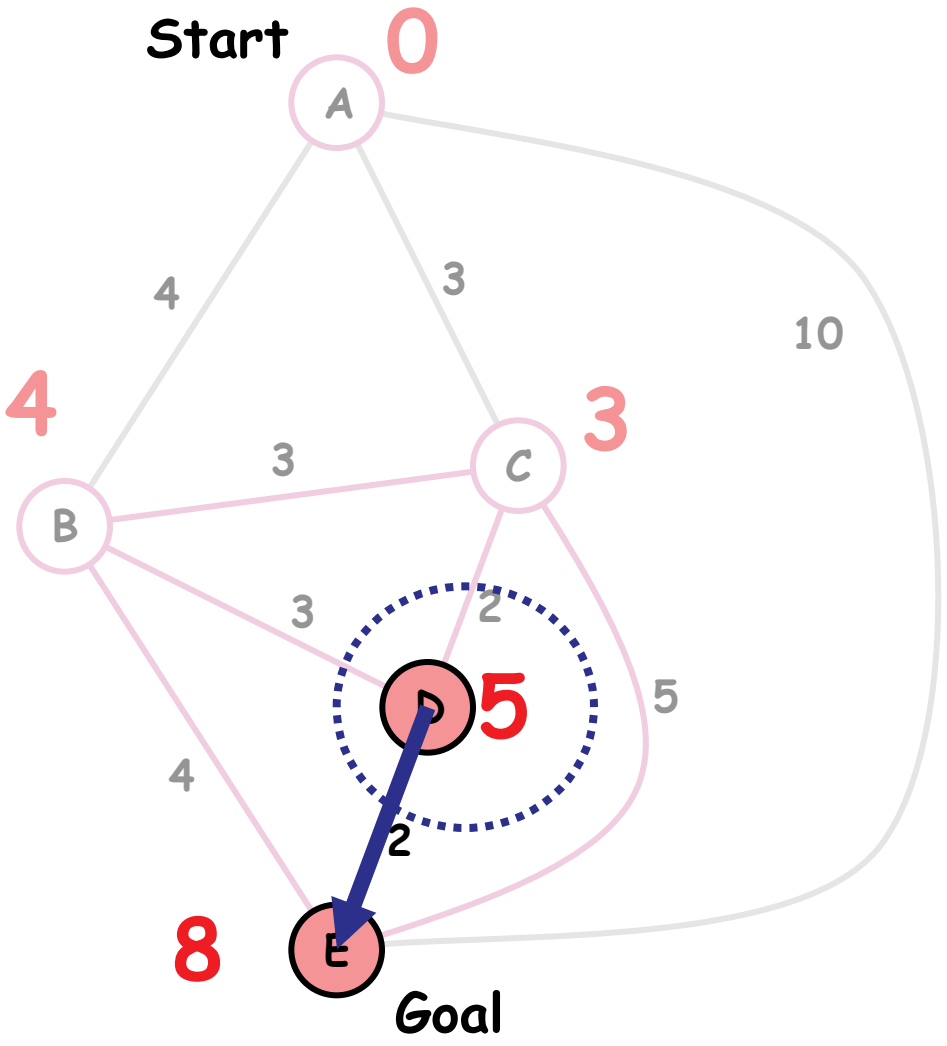
D -> B, C, E

E -> A, B, C, D

Contents of T:

{B, D, E}

Example



Edges:

A -> B, C, E

B -> A, C, D, E

C -> A, B, D, E

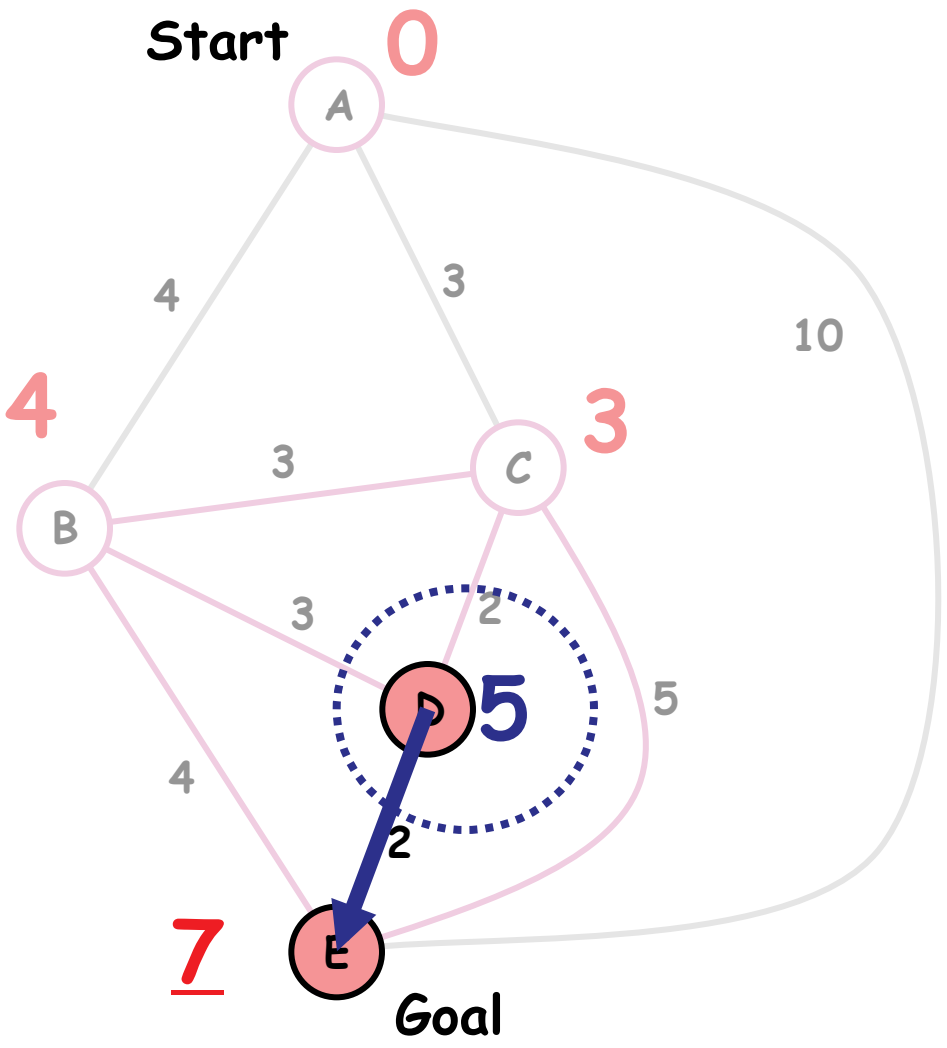
D -> B, C, E

E -> A, B, C, D

Contents of T:

{D, E}

Example



Edges:

A -> B, C, E

B -> A, C, D, E

C -> A, B, D, E

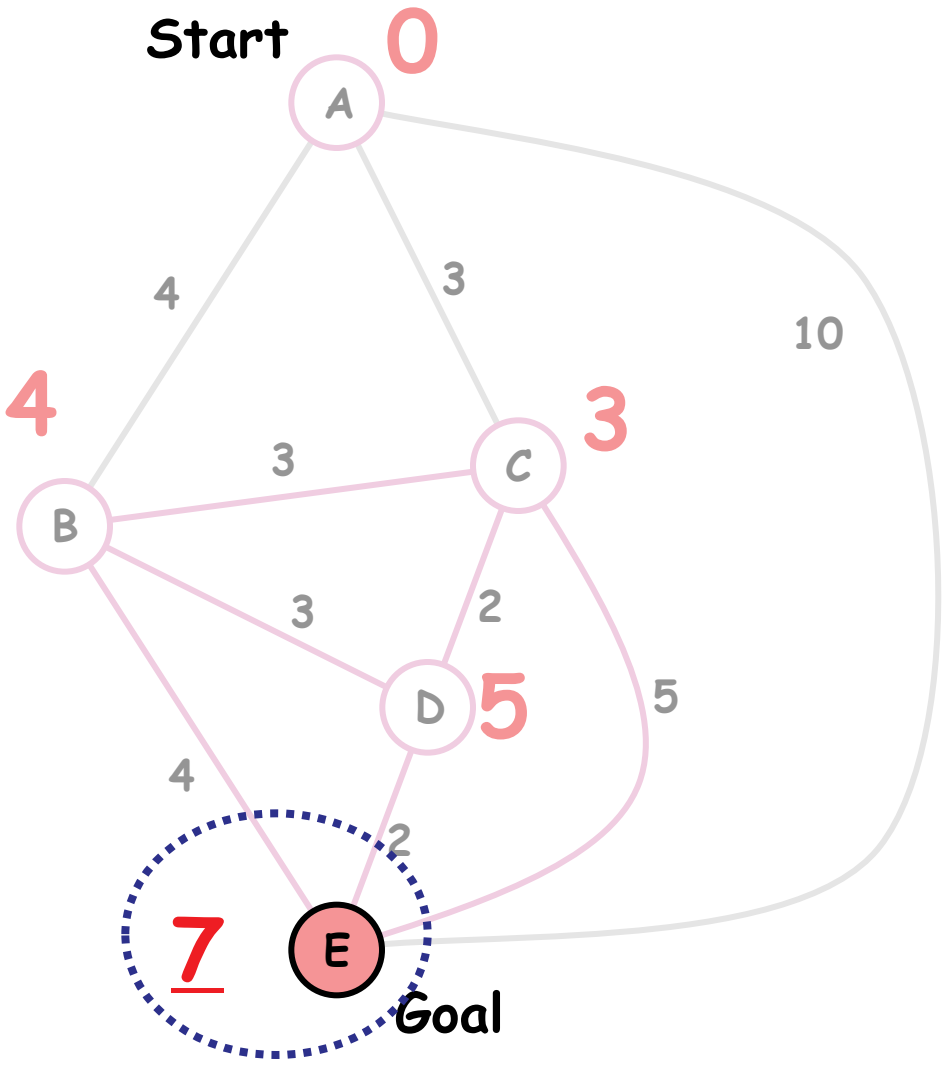
D -> B, C, E

E -> A, B, C, D

Contents of T:

{D, E}

Example



Edges:

A -> B, C, E

B -> A, C, D, E

C -> A, B, D, E

D -> B, C, E

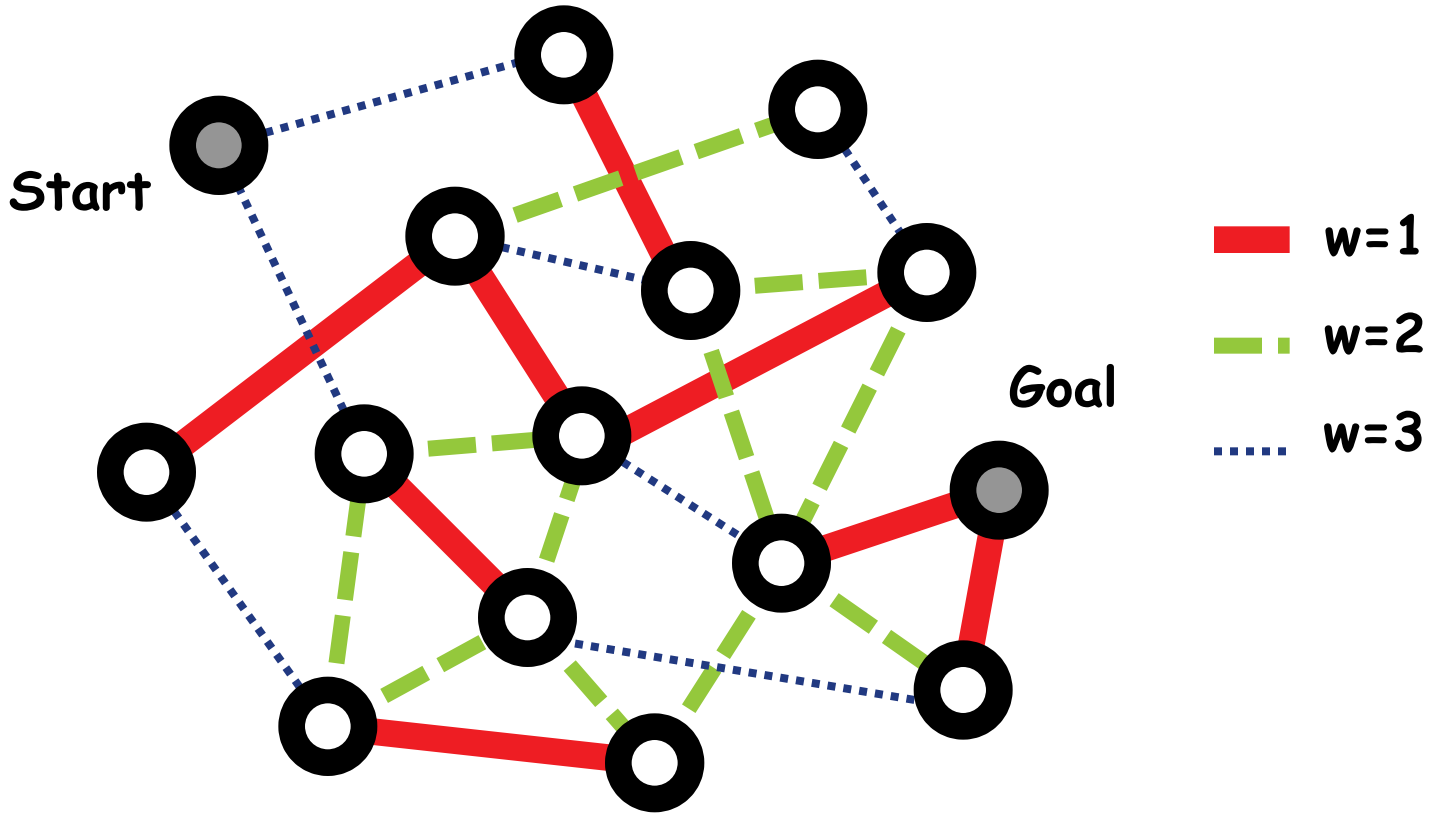
E -> A, B, C, D

Contents of T:

{E}

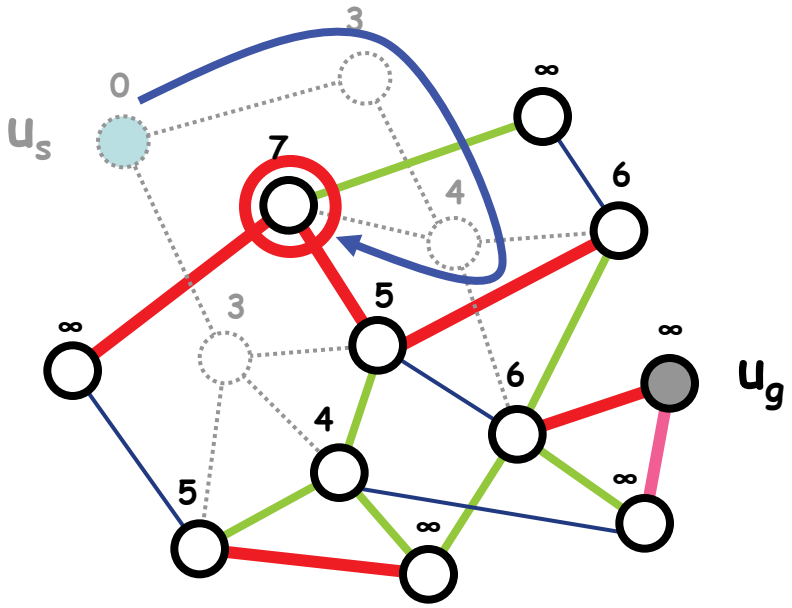
Exercise

- Find the shortest path and its length in the following graph using Dijkstra's algorithm



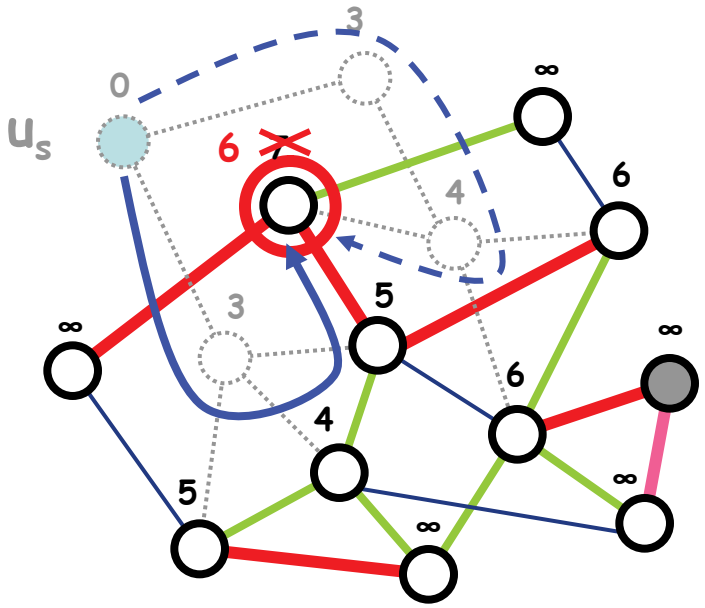
Why Dijkstra's algorithm works (1)

- For each node in T, the label L(u) always represents the length of the tentative shortest path from u_s to the node, without visiting any nodes in T



Why Dijkstra's algorithm works (2)

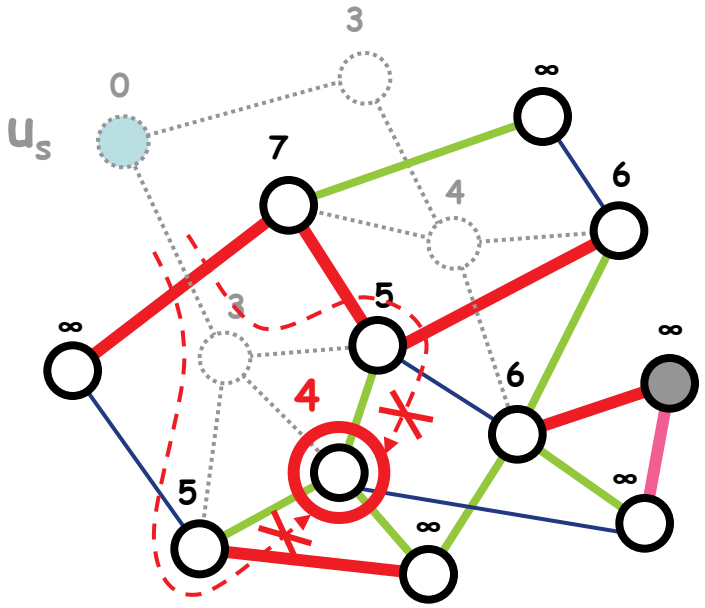
- There is still some possibility for each node in T that it may get smaller L(u) by way of other nodes in T



Why Dijkstra's algorithm works (3)

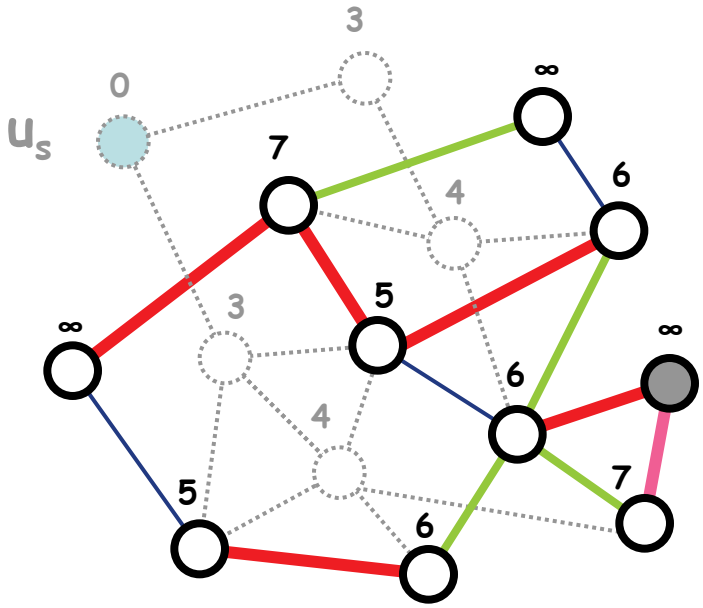
- However, the node with the smallest L(u) in T doesn't have such possibility

  -> For such a node, L(u) is the final value



Why Dijkstra's algorithm works (4)

- Hence that node can be removed from T, reducing the size of T
- Repeat this process until the goal node is finally removed from T



Exercise

- How to obtain the shortest path (i.e., sequence of edges) with the Dijkstra algorithm, not just its length?
  - Assign an additional label k(u) to each node to keep track of "where I came from"
  - k is updated every time L is updated
  - Once the shortest distance is obtained, reconstruct the actual path by following k(u) from the goal back to the start

Floyd-Warshall algorithm

- R. W. Floyd / S. Warshall (1962)

- Calculates shortest path lengths between ALL pairs of nodes
  - Works with undirected/directed, weighted/unweighted networks
  - Computational complexity: $O(n^3)$
  - Dynamic programming: Algorithm is simple and easy!

Floyd-Warshall algorithm (1)

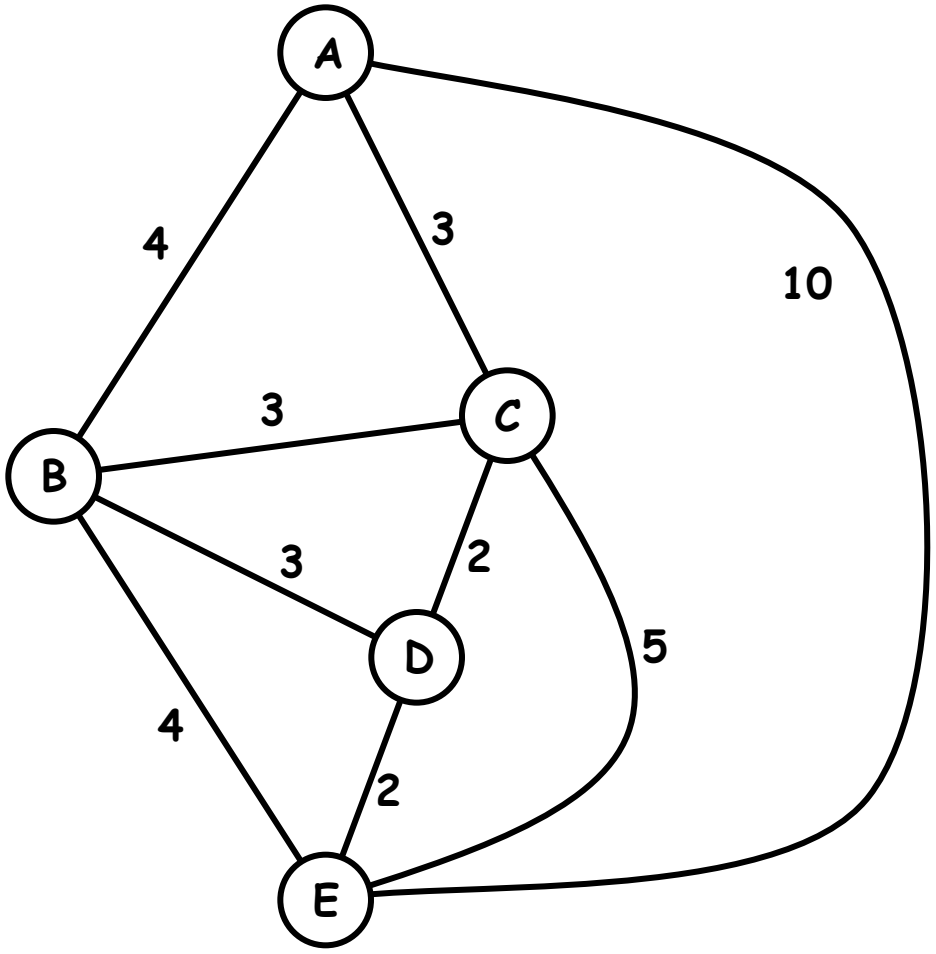
- $d^{(k)}_{ij}$: length of the shortest path between nodes i and j in which only nodes 1, 2, …, k are allowed to appear on the path

- $d^{(0)}_{ij}$: edge weight between nodes i & j (∞ if i and j are not connected)

Floyd-Warshall algorithm (2)

- Basic idea: calculate $d^{(k)}_{ij}$ using $d^{(k-1)}_{ij}$
- Update $d^{(k)}_{ij}$ only if it helps to go through node k
- $d^{(k)}_{ij} = \min \{ d^{(k-1)}_{ij} , d^{(k-1)}_{ik} + d^{(k-1)}_{kj} \}$
- Repeat this for k = 1 ~ n

Exercise



- Calculate shortest path lengths using the Floyd-Warshall algorithm
- Show how $d^{(k)}_{ij}$ develops over time

Other Route Search Problems on Networks

Chinese Postman Problem (CPP)

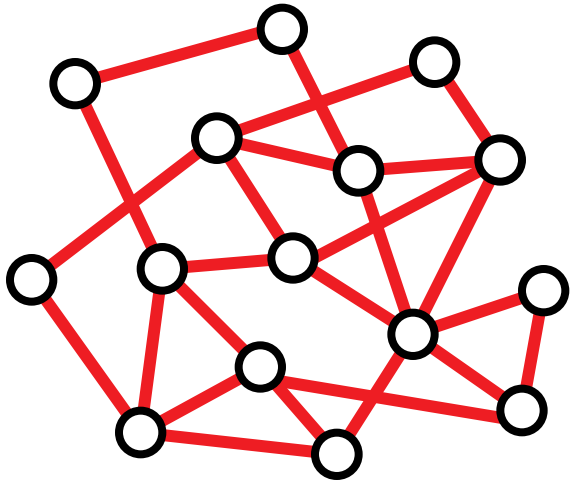
(Originally proposed by Kwan Mei-Ko in 1962)

Departing from a central post office in a city, a postman has to go through all the streets in the city, delivering mail to streetside houses, and then come back to the post office

Tell him the most efficient route to go through all the streets in the city

Chinese Postman Problem (CPP)

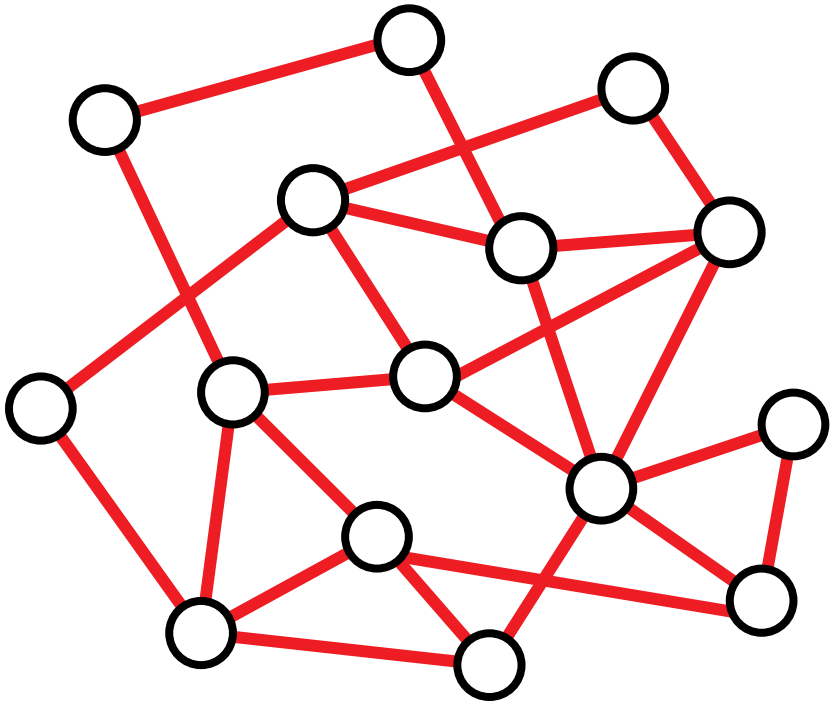
A problem to find the shortest cyclic walk that includes every edge in a graph



- Relatively easy to find for Eulerian or semi-Eulerian graphs
- Can be solved in general by finding which edges you should travel twice

Exercise

- Find the most efficient cyclic walk for a postman working in a city (graph below)
  - Assume every edge has length 1



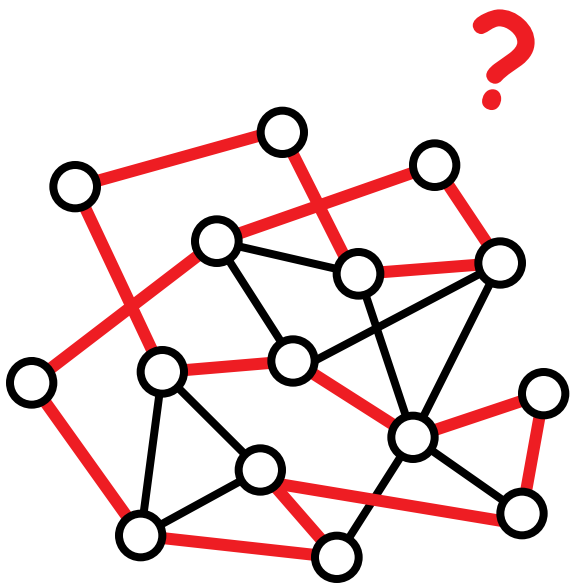
Traveling Salesman Problem (TSP)

- A salesman, who has just arrived at a central station of a city, has to visit all the houses in the city to sell stuff and then come back to the station
- He shouldn't visit the same house more than once

Tell him the most efficient route to visit every house in the city just once

Traveling Salesman Problem (TSP)

A problem to find the shortest cyclic path that includes every node in a graph



Very hard to solve in general

NP-hard problems

- Obtaining optimal solutions for TSP is known to be in the class called "NP-hard problems"
- Intuitively: Problems whose general solutions require computational complexity that is believed not to be bound by a polynomial of the problem size n (no rigorous proof given yet)

Minimum Spanning Tree

Minimum spanning tree

- *A subgraph of a connected network that is a tree (= graph with no cycles) and connects all the nodes with the smallest sum of link weights*
  - Works with *weighted* undirected graphs
  - Useful for finding and visualizing the "backbone" of a complex network
  - "Maximum spanning tree" could also be obtained by inverting link weights

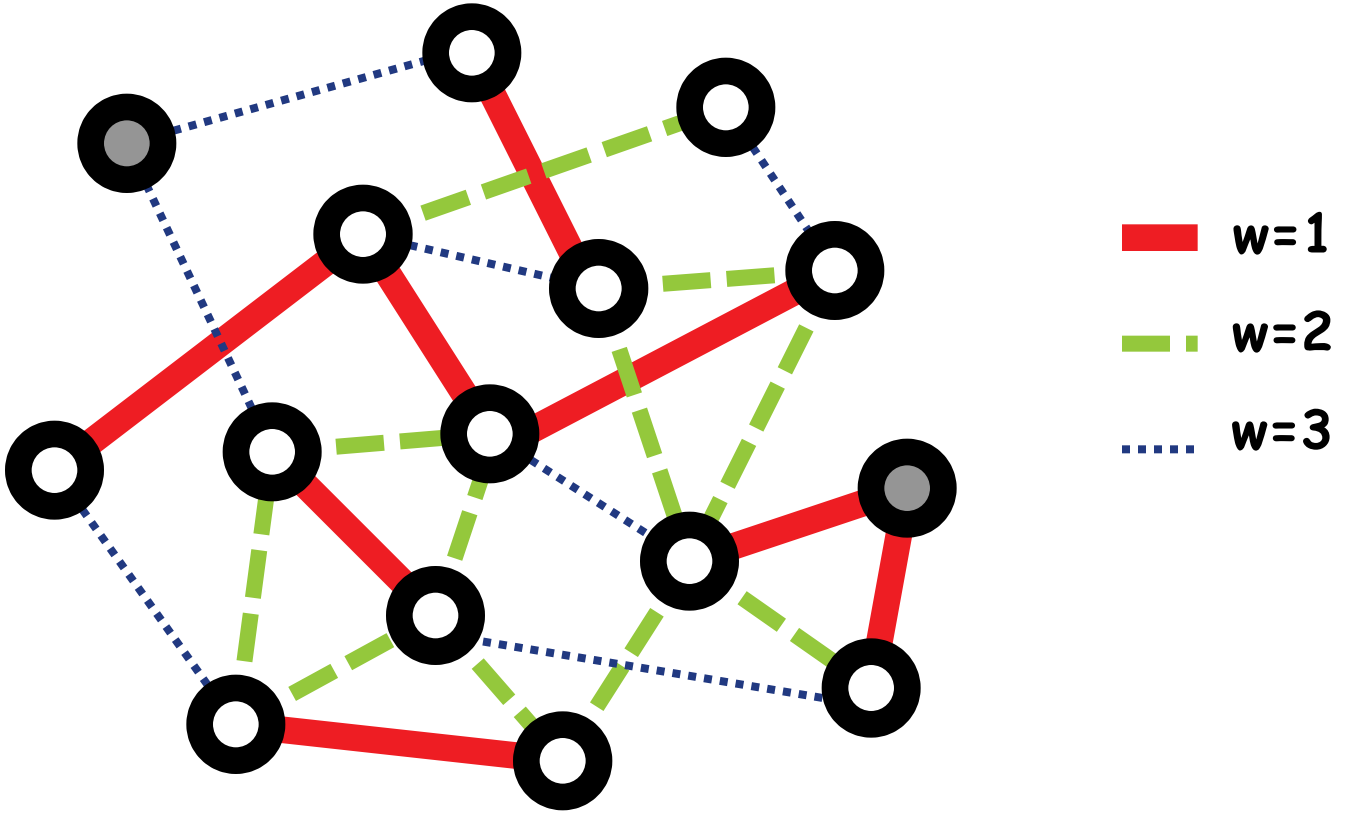
Kruskal's algorithm

- T: set of trees (initially just a set of nodes)
- L: set of links

1. Take a link with minimum weight from L
2. If the selected link connects two trees in T, connect them in T
   - Otherwise discard it
3. Repeat 1 & 2 until you get MST

Exercise

- Find the minimum spanning tree of the following network



Exercise

- Generate a random network with n = 100, p = 0.05
- Assign random weights to each link
- Visualize the original network & its minimum spanning tree using the same node positions

Max-Flow Min-Cut Theorem

"Thickness" of connection between nodes in unweighted graphs

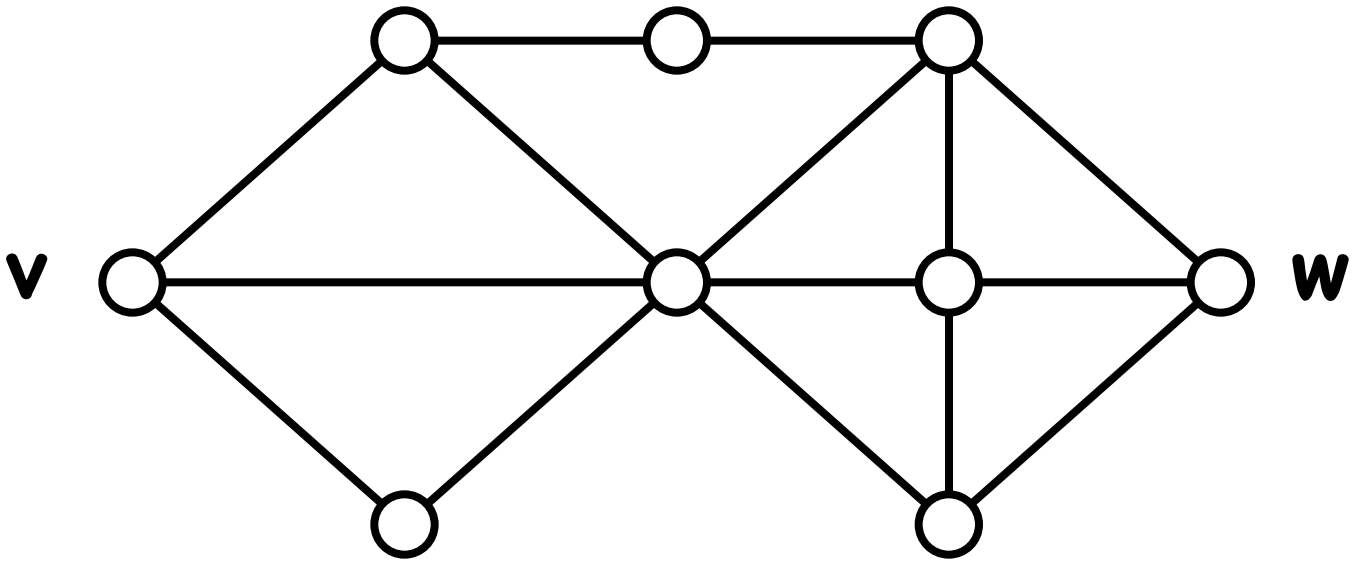
- Edge-disjoint path
- Disconnecting set
- Menger's theorem

Edge-disjoint paths and disconnecting sets

- Think about multiple paths that connect between nodes v and w
- If they have no common edges, they are called edge-disjoint paths
- If such paths always have to go through one of the edges in a selected set, then the set is called vw-disconnecting set

Exercise

- Give some examples of edge-disjoint paths between v and w and vw-disconnecting sets in the graph below



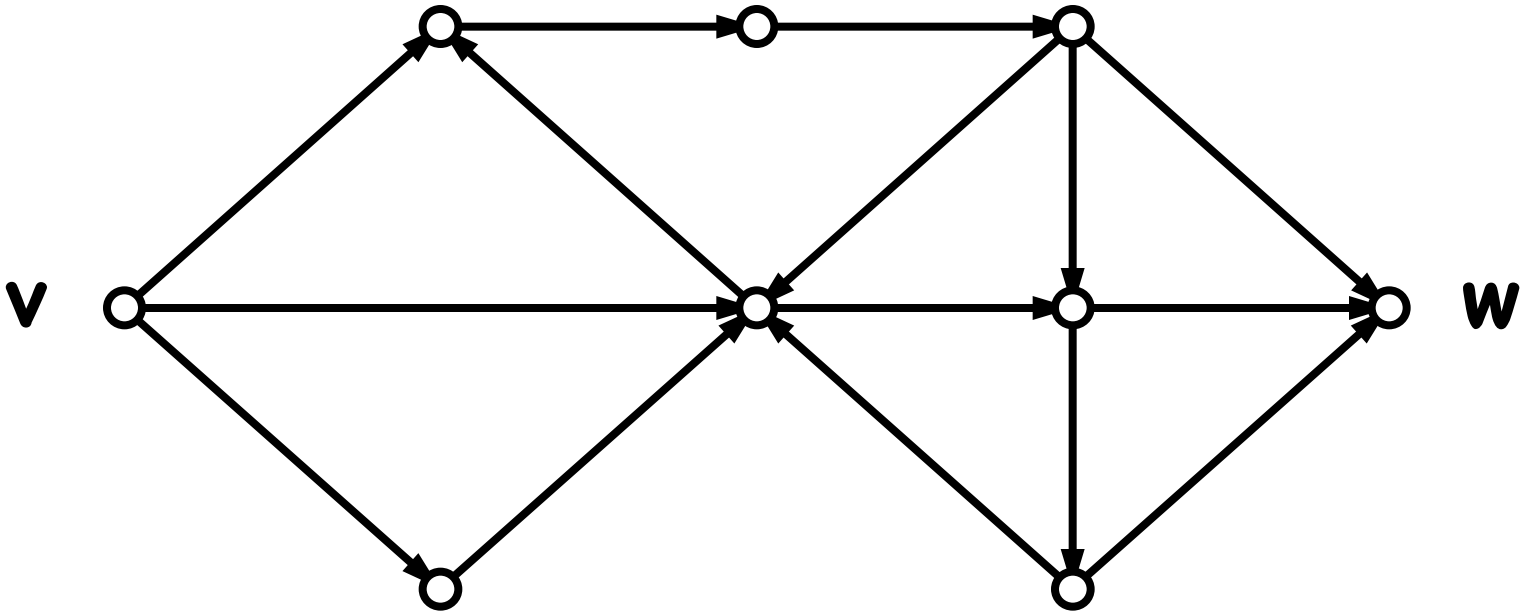
Menger's theorem (edge version)

- The maximal # of edge-disjoint paths between nodes v and w in a connected graph equals the minimum # of edges in the smallest vw-disconnecting set
  - Holds for both undirected and directed graphs
- Intuitive explanation:

  The "thickness" of connection between two nodes is determined by the capacity of the "narrowest" part

Exercise

- Confirm Menger's theorem by finding edge-disjoint paths between v and w and the smallest vw-disjointing set in the graph below



"Thickness" of connection between nodes in weighted directed graphs

- Flow
- Cut
- Maximum-flow minimum-cut theorem

Flow

- A flow from source v to sink w in a weighted directed graph is defined by the assignment of local current to each edge, f(e), that satisfy:

$f(e) \leq w(e)$

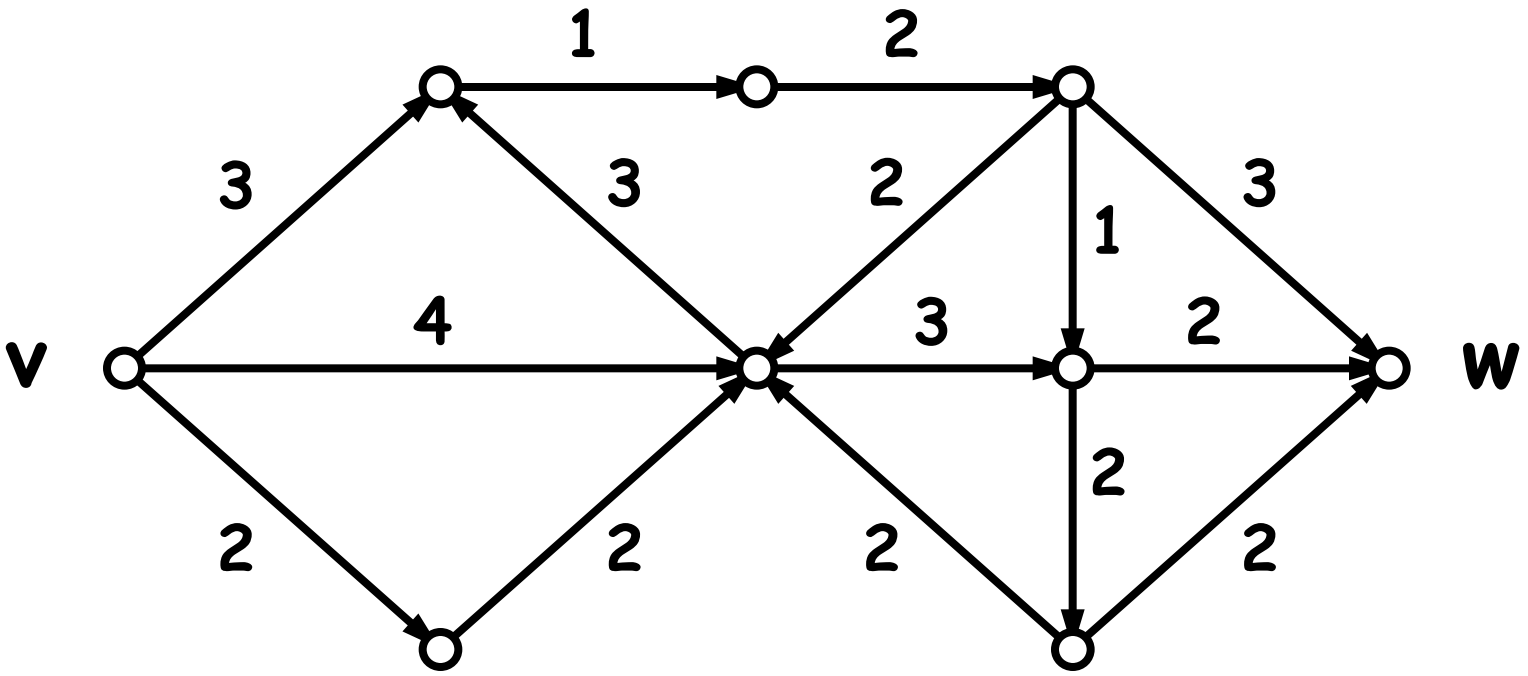
Local current must not exceed the weight (capacity) on every edge

$\Sigma_x f(\langle u, x \rangle) = \Sigma_y f(\langle y, u \rangle)$ for $u \neq v, w$

Incoming volume equals outgoing volume for all nodes except for source and sink

Exercise

- Make a sample flow with volume 3 from v to w on the graph below
  - Each number represents edge capacity



Maximum flow

- *A volume of a flow* is defined by the total outgoing volume at source v (= total incoming volume at sink w)

- *A maximum flow* is a flow that conveys the maximum volume from v to w on a graph

Cut, minimum cut

- A set of edges C is called a cut if every path between v and w has to go through one of the edges in C (the same notion as vw-disconnecting set)
- A capacity of a cut is the sum total of the weights of edges in the cut
  - Any flow b/w v and w can't exceed the capacity of any of their cuts
- A minimum cut is a cut that has minimum capacity

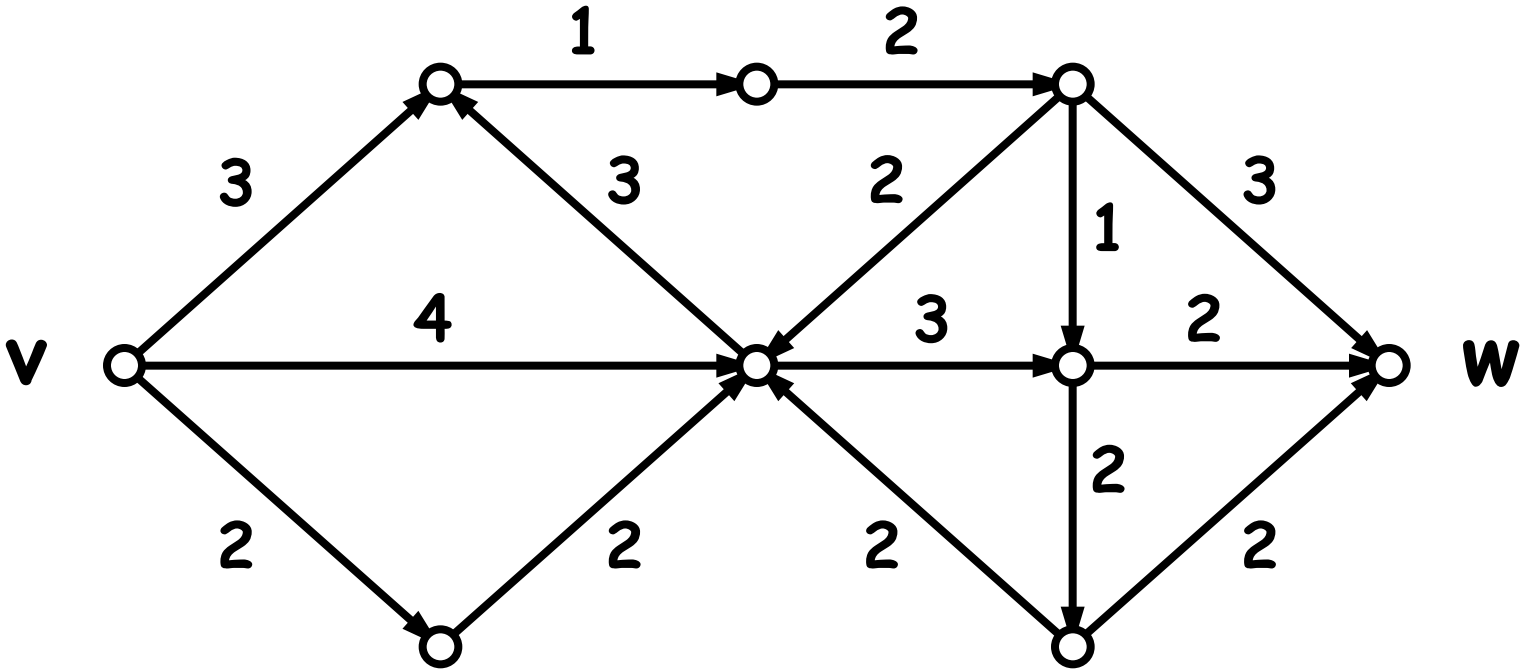
Maximum-flow minimum-cut theorem

- In any graph, the maximum flow between nodes v and w is exactly the capacity of the minimum cut between v and w

  (Assuming each weight as 1 will reduce this theorem to Menger's one)

Exercise

- Confirm the maximum-flow minimum-cut theorem on the graph below



Intuitive proof (1)

- Given a flow on a graph, try to pick up every unsaturated edge that does not use its capacity to the full extent [i.e. $f(e) < w(e)$]
- If the flow is maximum, then you cannot reach sink w from source v by using such unsaturated edges only
  - If you can, that means the flow is not maximum yet

Intuitive proof (2)

- Call the set of all nodes that can be reached by using only unsaturated edges S
- Call the set of all other nodes S'
- Since v is in S and w is in S', the set of edges bridging from S to S' forms a cut between v and w

Intuitive proof (3)

- All the edges from S to S' must be saturated
  - By the definition of S
- All the edges from S' back to S must be with 0 flow
  - Otherwise you could obtain a yet greater flow by canceling such a cyclic flow occurring in the middle of the graph

Intuitive proof (4)

- All the edges from S to S' must be saturated
- All the edges from S' to S must be with 0 flow

- I.e., the total volume of the flow from S to S' is all originated at source v, and is absorbed in sink w (= the total flow in the entire graph)

- This flow is exactly the sum total of the weights of all the edges from S to S' (= capacity of the cut)

Intuitive proof (5)

- Maximum flow = a capacity of a cut
- A minimum cut would have a capacity equal to or less than that of this cut; however, it must not be less than any flow by definition

  → This cut must be a minimum cut

  → It always equals the maximum flow

Exercise

- Create some non-trivial weighted network using NetworkX
- Choose one node arbitrarily
- Assess the strength of connection from the node to each of the rest of nodes by (1) distance and (2) max flow, and paint them using the results