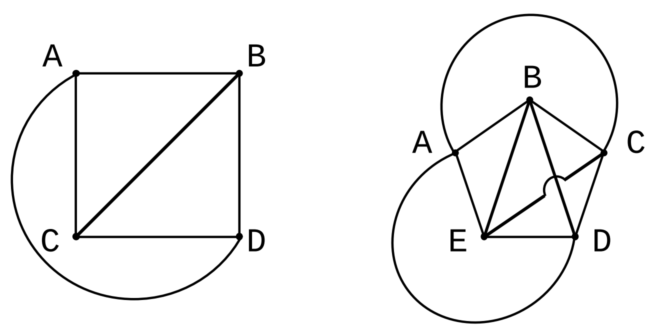
GRAPHS

Hanan Samet

Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
e-mail: hjs@umiacs.umd.edu

GRAPH (G)

- Generalization of a tree
  1. no longer a distinguished node called the root
     - implies no need to distinguish between leaf and nonleaf nodes
  2. two nodes can be linked by more than one sequence of edges
- Formally: set of vertices (V) and edges (E) joining them, with at most one edge joining any pair of vertices
- $(V_0, V_1, \ldots, V_n)$: path of length *n* from V_0 to V_n (chain)
- Simple Path: distinct vertices (elementary chain)
- Connected: path between any two vertices of G
- Cycle: simple path of length ≥ 3 from V_0 to V_0 (length in terms of edges)
- Planar: curves intersect only at points of graph
- Degree: number of edges intersecting at the node
- Isomorphic: if there is a one-to-one correspondence between nodes and edges of two graphs



SAMPLE GRAPH PROBLEM

- Given *n* people at a party who shake hands, show that at the party's end, an even number of people have shaken hands with an odd number of people

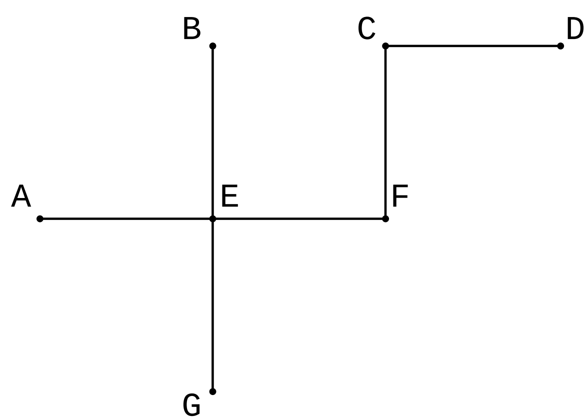
- Theorem: For any graph G an even number of nodes have an odd degree

- Proof:

  1. each edge joins 2 nodes
  2. each edge contributes 2 to the sum of degrees
  3. sum of degrees is even
  4. thus an even number of nodes with odd degree

FREE TREES

- Connected graph with no cycles
- Given G as a free tree with n vertices
  1. Connected, but not so if any edge is removed
  2. One simple path from V to V´ (V ≠ V´)
  3. No cycles and $n - 1$ edges
  4. G is connected with $n - 1$ edges

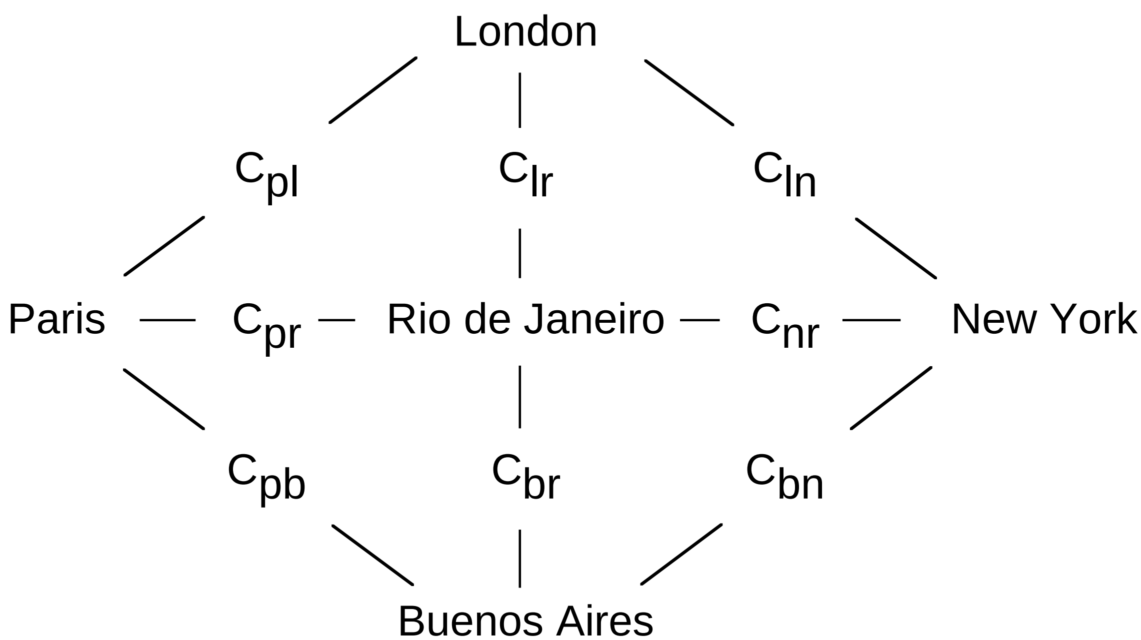


- Differences from regular trees:
  1. No identification of root
  2. No distinction between terminal and branch nodes

FREE SUBTREES

- Definition: set of edges such that all the vertices of the graph are connected to form a free tree

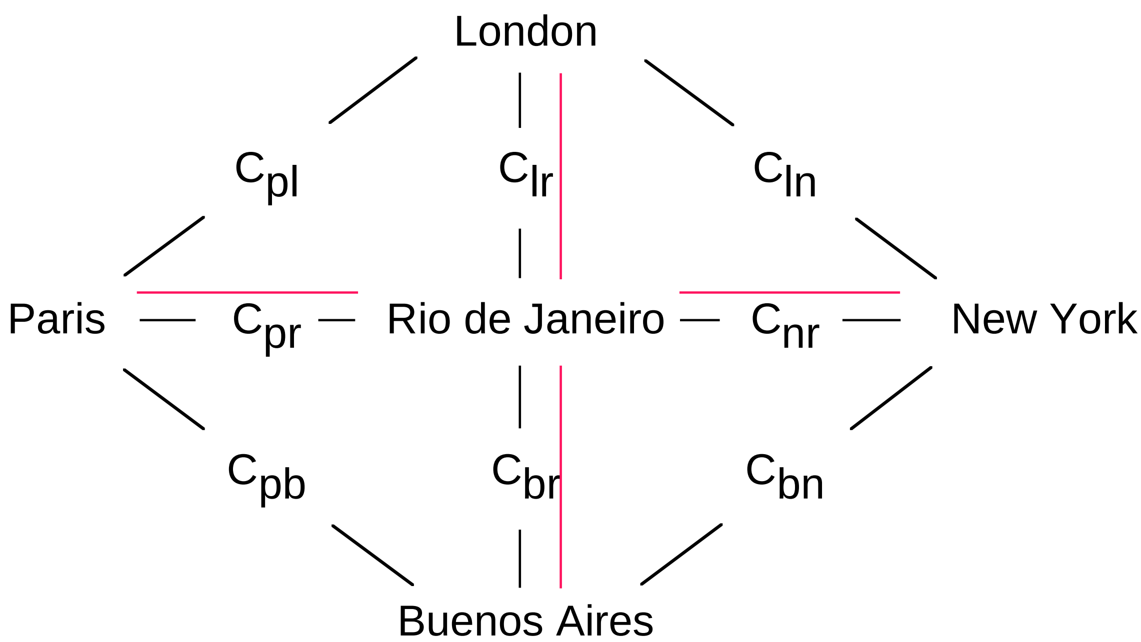
- Ex: distribution of telephone networks



FREE SUBTREES

- Definition: set of edges such that all the vertices of the graph are connected to form a free tree

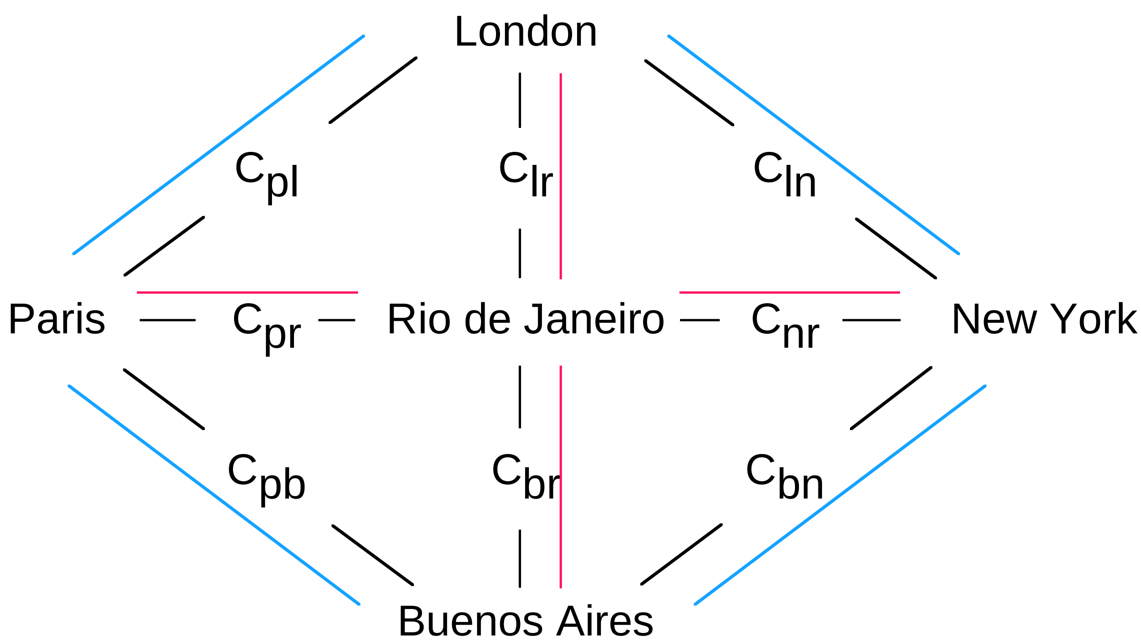
- Ex: distribution of telephone networks



- Free subtree

FREE SUBTREES

- Definition: set of edges such that all the vertices of the graph are connected to form a free tree
- Ex: distribution of telephone networks

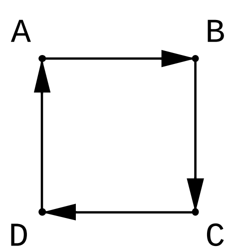


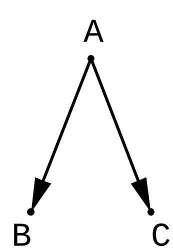
- Free subtree
- Given: connected graph G
  n nodes (5)
  m edges (8)
- Cyclomatic Number = number of edges that must be deleted to yield a free tree (= $m - n + 1$)

DIRECTED GRAPH

- Definition: graph with direction attached to the edges
- $(V_0, V_1, \ldots, V_n)$: path of length n from V_0 to V_n
- Elementary path: all vertices are distinct
- Circuit: cycle (but can have length 1 or 2)
- Elementary Circuit: all vertices are distinct
- Indegree:
- Outdegree:
- Strongly Connected: path from any V to any V´
- Rooted: at least one V with paths to all V´ ≠ V

- Note: strongly connected implies rooted but not vice versa





DATA STRUCTURES FOR GRAPHS

- Must decide what information is to be accessible and with what ease
- Most important information conveyed by a graph is connectivity which is indicated by its edges
- Two choices

  1. vertex-based
     - keeps track of nodes connected to each node
     - can implement as array A of lists

       a. one entry for each vertex p

       b. $A[p]$ is a list of all vertices P that are connected to p by virtue of the existence of an edge between p and q where $q \in P$ (also known as an *adjacency list*)

  2. edge-based
     - keeps track of edges
     - usually represented as a list of pairs of form $(p\ q)$ where there is an edge between vertices p and q
     - drawback: need to search entire set to determine edges connected to a particular vertex

DATA STRUCTURES FOR GRAPHS

- Must decide what information is to be accessible and with what ease
- Most important information conveyed by a graph is connectivity which is indicated by its edges
- Two choices
  1. vertex-based
     - keeps track of nodes connected to each node
     - can implement as array *A* of lists
       a. one entry for each vertex *p*
       b. *A*[*p*] is a list of all vertices *P* that are connected to *p* by virtue of the existence of an edge between *p* and *q* where $q \in P$ (also known as an *adjacency list*)
  2. edge-based
     - keeps track of edges
     - usually represented as a list of pairs of form (*p q*) where there is an edge between vertices *p* and *q*
     - drawback: need to search entire set to determine edges connected to a particular vertex

ADJACENCY MATRIX

- Hybrid approach
- Good for representing a directed graph

$A_{ij} = 1$	if an edge exists from i to j
$A_{ij} = 0$	otherwise
$A \cdot A = A^2$	adjacency matrix of distance 2

- Somewhat wasteful of space as there is an entry for every possible edge even though the array is usually sparse
  1. in such cases, a vertex-based representation such as an adjacency list is more economical
  2. adjacency matrix is useful if want to detect if an edge exists between two vertices
     - cumbersome when using a list as need to search
- Useful if want to keep track of all vertices reachable from every vertex
- Ex: Leftmost derivations

  $A \rightarrow bC$

  $A \rightarrow Bd$

  $B \rightarrow c$

	A	B	C	b	c	d
A	0	1	0	1	0	0
B	0	0	0	0	1	0
C	0	0	0	0	0	0
b	0	0	0	0	0	0
c	0	0	0	0	0	0
d	0	0	0	0	0	0

- Boolean matrices

$1 + 1 = 1 \qquad 1 \cdot 1 = 1$

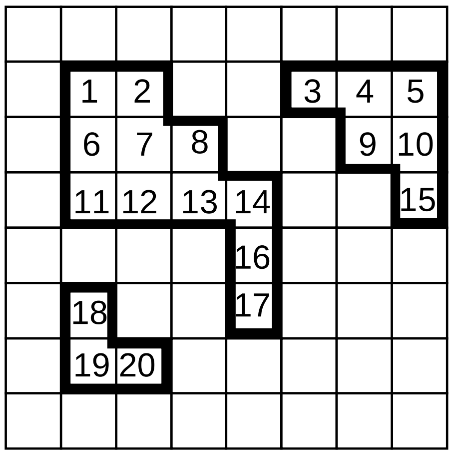
$1 + 0 = 1 \qquad 1 \cdot 0 = 0 \cdot 1 = 0 \cdot 0 = 0$

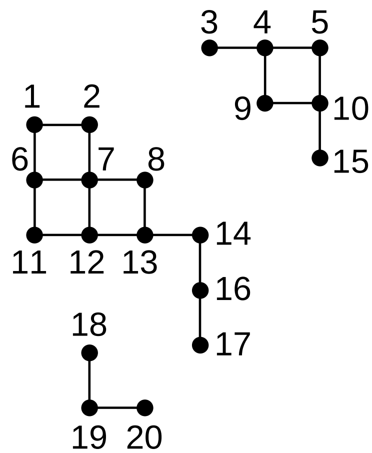
$0 + 0 = 0$

- Cycle of length n $\quad A^n_{ii} = 1$

CONNECTED GRAPH

- Def: there exists path between any two vertices of the graph
- Ex: binary image





1. image graph
   - image elements are vertices
   - horizontal and vertical adjacencies between image elements are edges
2. connected component labeling: determine separate regions of binary image
   - image graph is stored implicitly
   - easy to access adjacent vertices given location of a vertex
   - neither a vertex-based or edge-based representation; instead algorithms are based on them
     1. vertex-based implies need to follow connectivity
        - depth-first or seed-filling approach
        - many page faults if disk-resident data
     2. edge-based determines edges by examining image row-by-row
        - only need to access two rows simultaneously
        - good for disk-resident data
     3. both take ≈ O(number of image elements) time

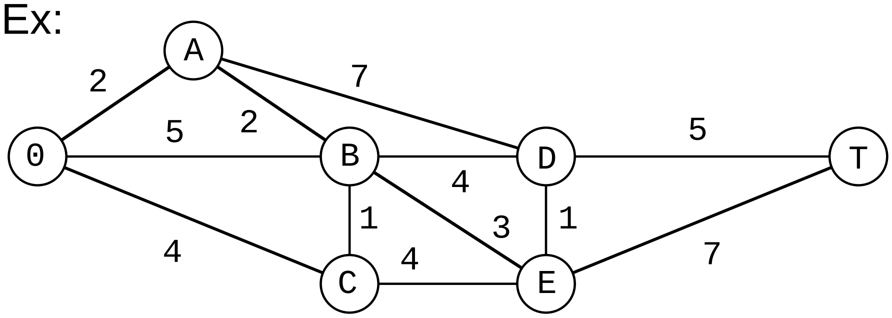
MINIMUM SPANNING TREE

- Cost C_{ij} associated with each edge from i to j
- Find the free subtree of G with minimum cost
- Solution:

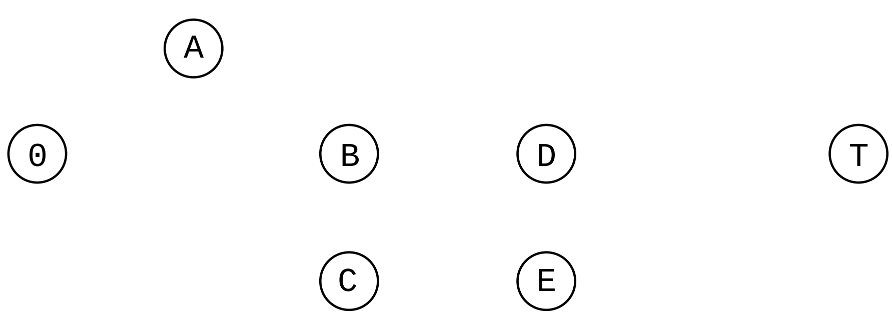
C = connected nodes: initially { }
U = unconnected nodes: initially { all nodes }

1. choose arbitrary node and place it in C
2. select node in U that is closest to a node in C and add edge; move node from U to C; repeat until U is empty

Ex:



Start with node 0
C is built by choosing:



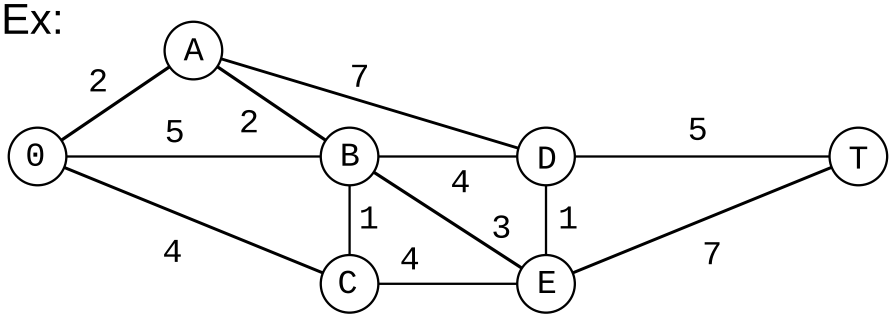
MINIMUM SPANNING TREE

- Cost C_{ij} associated with each edge from i to j
- Find the free subtree of G with minimum cost
- Solution:

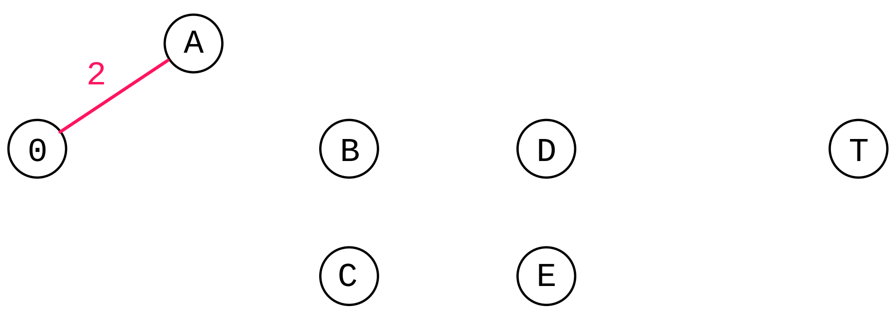
C = connected nodes: initially { }
U = unconnected nodes: initially { all nodes }

1. choose arbitrary node and place it in C
2. select node in U that is closest to a node in C and add edge; move node from U to C; repeat until U is empty

Ex:



Start with node 0
C is built by choosing: A



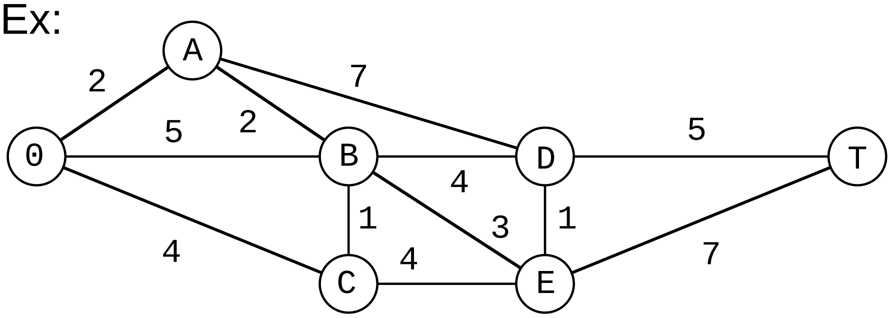
MINIMUM SPANNING TREE

- Cost C_{ij} associated with each edge from i to j
- Find the free subtree of G with minimum cost
- Solution:

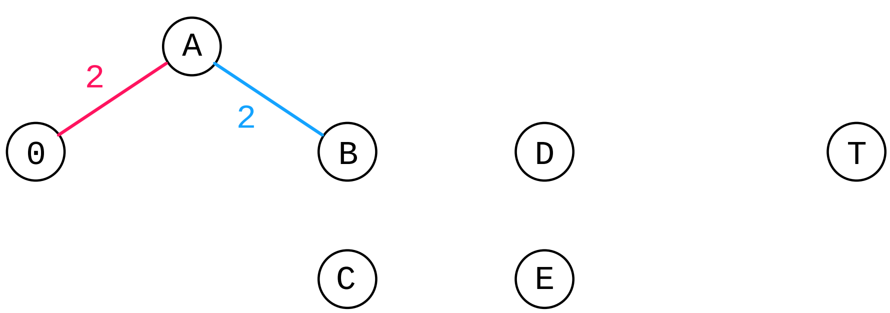
C = connected nodes: initially { }
U = unconnected nodes: initially { all nodes }

1. choose arbitrary node and place it in C
2. select node in U that is closest to a node in C and add edge; move node from U to C; repeat until U is empty

Ex:



Start with node 0
C is built by choosing: A B



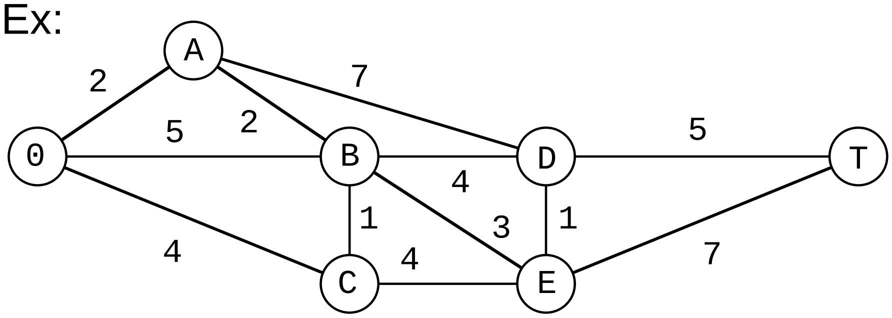
MINIMUM SPANNING TREE

- Cost C_{ij} associated with each edge from i to j
- Find the free subtree of G with minimum cost
- Solution:

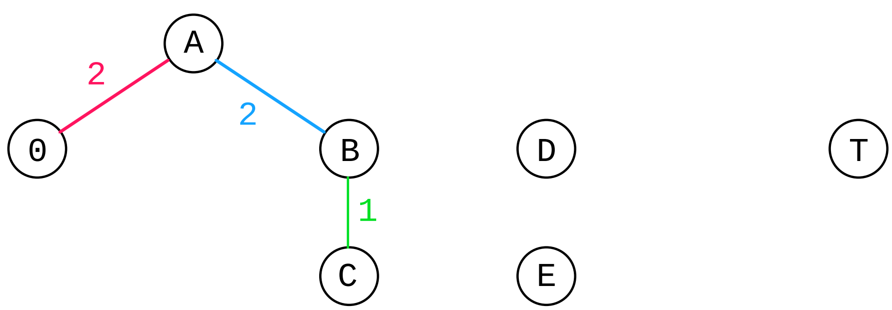
C = connected nodes: initially { }
U = unconnected nodes: initially { all nodes }

1. choose arbitrary node and place it in C
2. select node in U that is closest to a node in C and add edge; move node from U to C; repeat until U is empty

Ex:



Start with node 0
C is built by choosing: A B C



MINIMUM SPANNING TREE

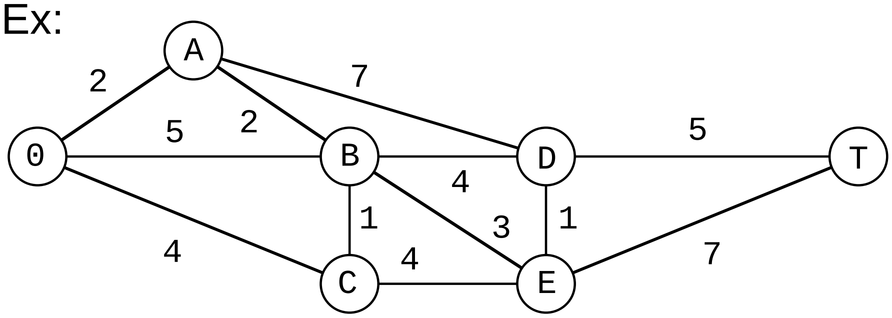
- Cost C_{ij} associated with each edge from i to j
- Find the free subtree of G with minimum cost
- Solution:

C = connected nodes: initially { }

U = unconnected nodes: initially { all nodes }

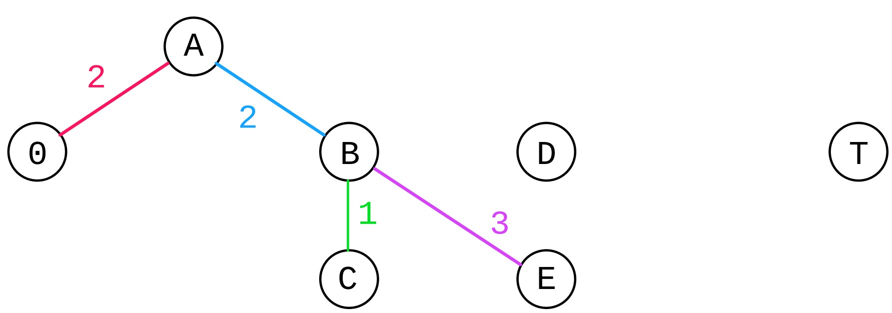
1. choose arbitrary node and place it in C
2. select node in U that is closest to a node in C and add edge; move node from U to C; repeat until U is empty

Ex:



Start with node 0

C is built by choosing: A B C E



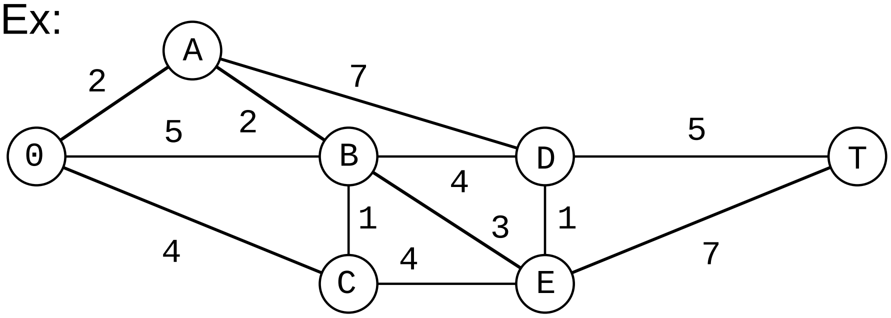
MINIMUM SPANNING TREE

- Cost C_{ij} associated with each edge from i to j
- Find the free subtree of G with minimum cost
- Solution:

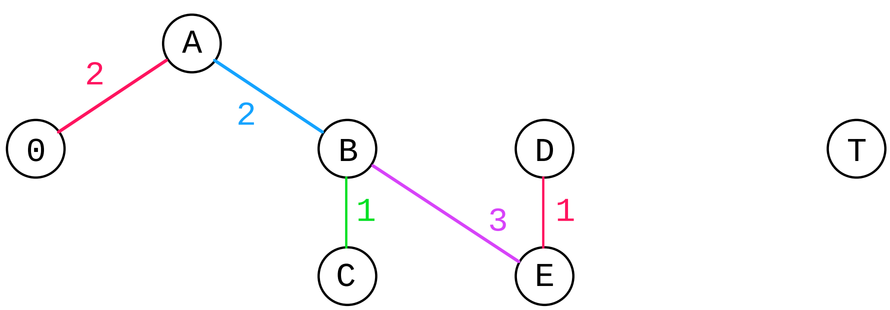
C = connected nodes: initially { }
U = unconnected nodes: initially { all nodes }

1. choose arbitrary node and place it in C
2. select node in U that is closest to a node in C and add edge; move node from U to C; repeat until U is empty

Ex:



Start with node `0`
C is built by choosing: `A B C E D`



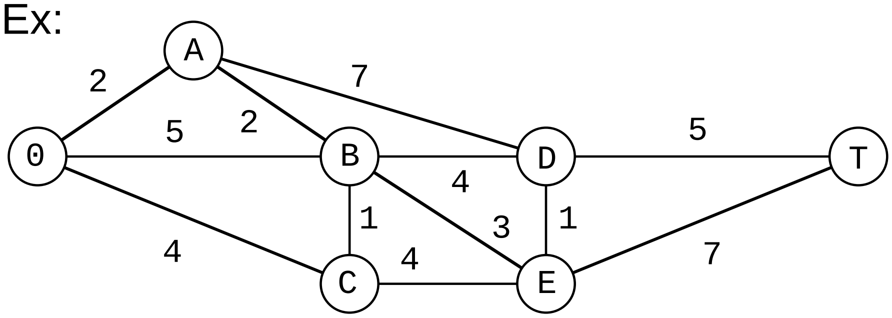
MINIMUM SPANNING TREE

- Cost C_{ij} associated with each edge from i to j
- Find the free subtree of G with minimum cost
- Solution:

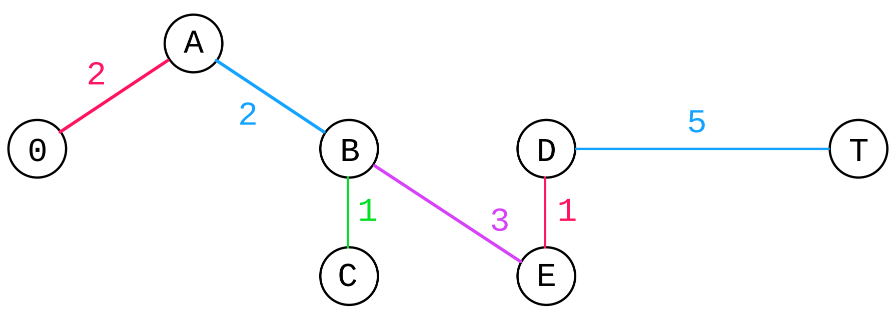
C = connected nodes: initially { }
U = unconnected nodes: initially { all nodes }

1. choose arbitrary node and place it in C
2. select node in U that is closest to a node in C and add edge; move node from U to C; repeat until U is empty

Ex:



Start with node 0
C is built by choosing: A B C E D T



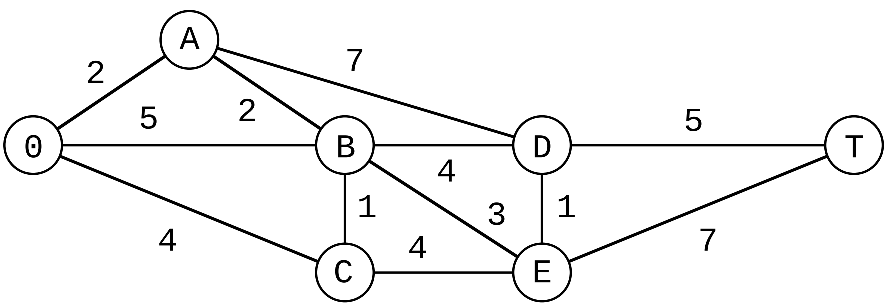
SHORTEST ELEMENTARY CHAIN

- Given node X0 in G find the shortest (cheapest) chain joining X0 with all the nodes of G

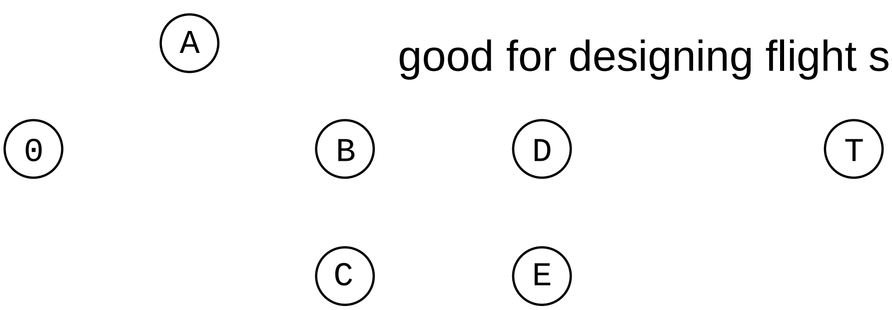
- Solution:
  C = connected nodes: initially X0
  U = unconnected nodes: initially all but X0
  E = set of edges: initially empty

  1. find the closest node in U to X0 (say X1)
  2. move X1 from U to C
  3. add (X0, X1) to E
  4. for each Xi in C find Yi in U that is closest; choose Ym such that cost from X0 to Ym is a minimum and add (Xm, Ym) to E; repeat until U is empty

Ex: start at node 0



Result:



good for designing flight schedules

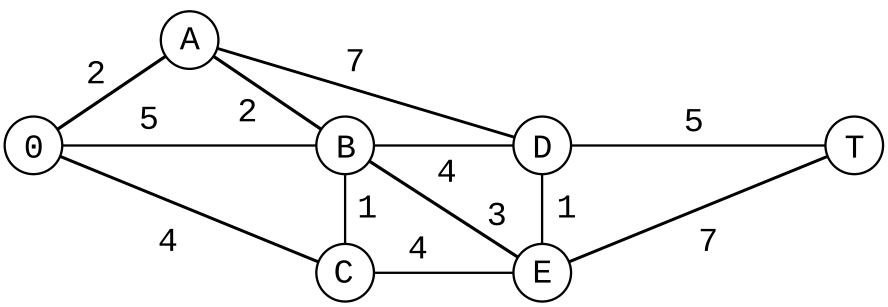
SHORTEST ELEMENTARY CHAIN

- Given node X0 in G find the shortest (cheapest) chain joining X0 with all the nodes of G

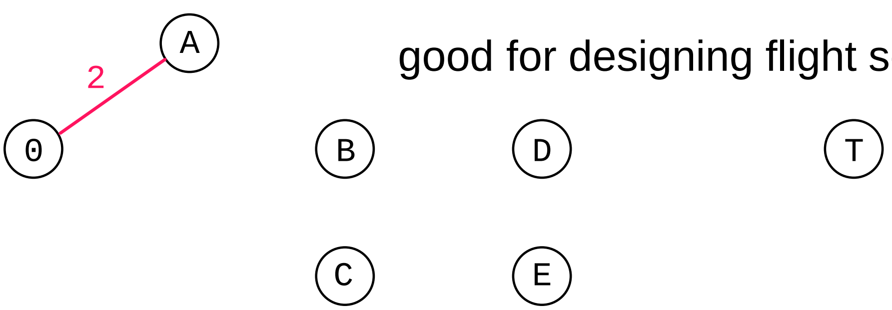
- Solution:
  C = connected nodes: initially X0
  U = unconnected nodes: initially all but X0
  E = set of edges: initially empty

  1. find the closest node in U to X0 (say X1)
  2. move X1 from U to C
  3. add (X0, X1) to E
  4. for each Xi in C find Yi in U that is closest; choose Ym such that cost from X0 to Ym is a minimum and add (Xm, Ym) to E; repeat until U is empty

Ex: start at node 0



Result: $0_{(0,A)}$



good for designing flight schedules

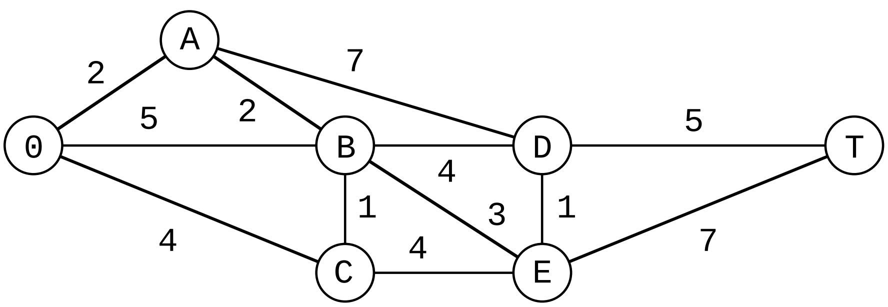
SHORTEST ELEMENTARY CHAIN

- Given node X0 in G find the shortest (cheapest) chain joining X0 with all the nodes of G

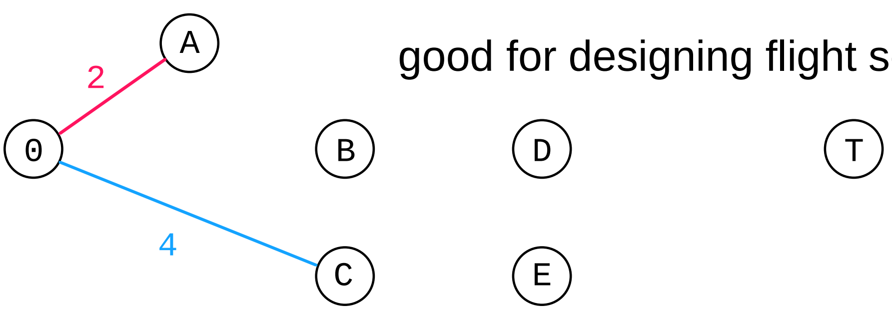
- Solution:
  C = connected nodes: initially X0
  U = unconnected nodes: initially all but X0
  E = set of edges: initially empty

  1. find the closest node in U to X0 (say X1)
  2. move X1 from U to C
  3. add (X0, X1) to E
  4. for each Xi in C find Yi in U that is closest; choose Ym such that cost from X0 to Ym is a minimum and add (Xm, Ym) to E; repeat until U is empty

Ex: start at node 0



Result: 0(0,A) A(0,C) or (A,B)



good for designing flight schedules

SHORTEST ELEMENTARY CHAIN

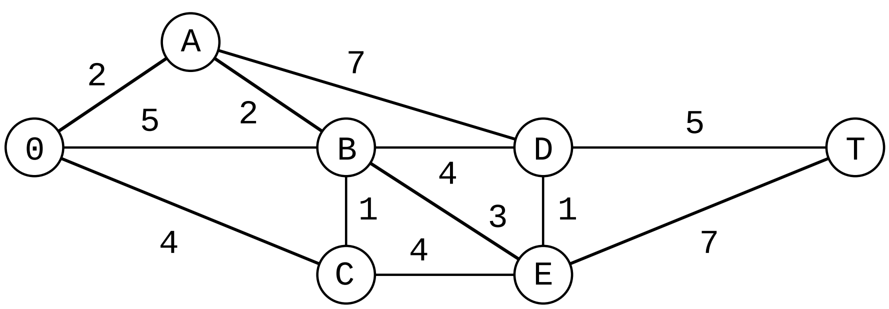
- Given node X0 in G find the shortest (cheapest) chain joining X0 with all the nodes of G

- Solution:

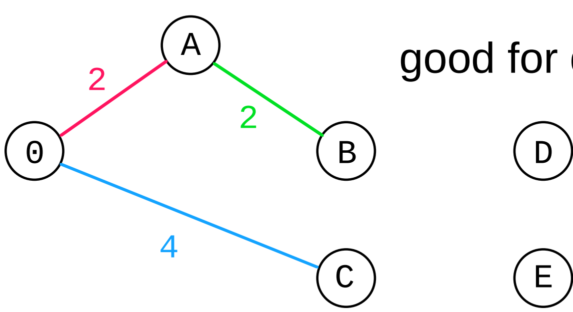
  C = connected nodes: initially X0
  U = unconnected nodes: initially all but X0
  E = set of edges: initially empty

  1. find the closest node in U to X0 (say X1)
  2. move X1 from U to C
  3. add (X0, X1) to E
  4. for each Xi in C find Yi in U that is closest; choose Ym such that cost from X0 to Ym is a minimum and add (Xm, Ym) to E; repeat until U is empty

Ex: start at node `0`



Result: `0` `(0,A)` `A` `(0,C)` or `(A,B)` `C` `(A,B)`



good for designing flight schedules

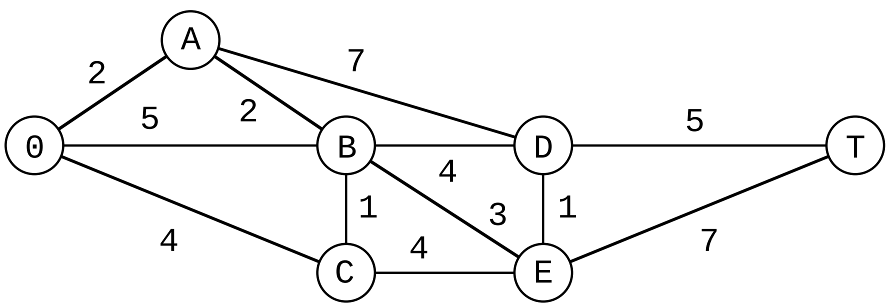
SHORTEST ELEMENTARY CHAIN

- Given node X0 in G find the shortest (cheapest) chain joining X0 with all the nodes of G

- Solution:
  C = connected nodes: initially X0
  U = unconnected nodes: initially all but X0
  E = set of edges: initially empty

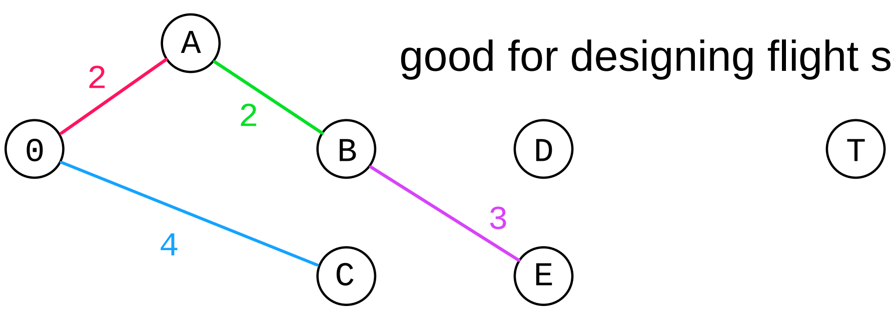
  1. find the closest node in U to X0 (say X1)
  2. move X1 from U to C
  3. add (X0, X1) to E
  4. for each Xi in C find Yi in U that is closest; choose Ym such that cost from X0 to Ym is a minimum and add (Xm, Ym) to E; repeat until U is empty

Ex: start at node 0



Result: $0_{(0,A)}$ $A_{(0,C)\text{ or }(A,B)}$ $C_{(A,B)}$ $B_{(B,E)}$

good for designing flight schedules



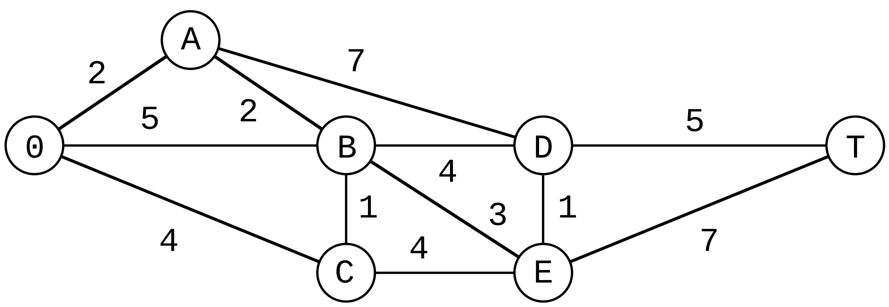
SHORTEST ELEMENTARY CHAIN

- Given node X0 in G find the shortest (cheapest) chain joining X0 with all the nodes of G

- Solution:
  C = connected nodes: initially X0
  U = unconnected nodes: initially all but X0
  E = set of edges: initially empty

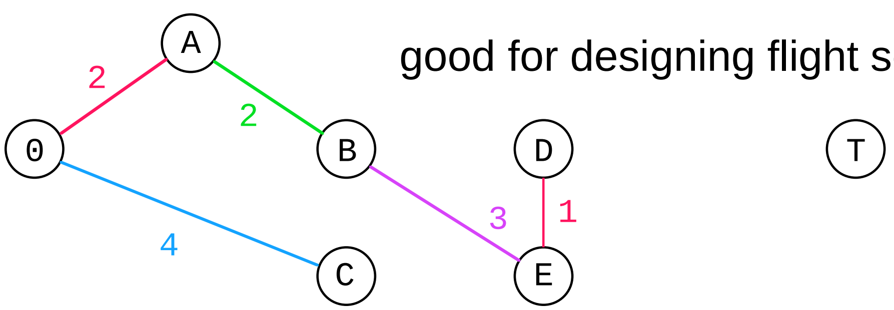
  1. find the closest node in U to X0 (say X1)
  2. move X1 from U to C
  3. add (X0, X1) to E
  4. for each Xi in C find Yi in U that is closest; choose Ym such that cost from X0 to Ym is a minimum and add (Xm, Ym) to E; repeat until U is empty

Ex: start at node `0`



Result: $0_{(0,A)}$ $A_{(0,C)}$ or $_{(A,B)}$ $C_{(A,B)}$ $B_{(B,E)}$ $E_{(D,E)}$ or $_{(B,D)}$

good for designing flight schedules



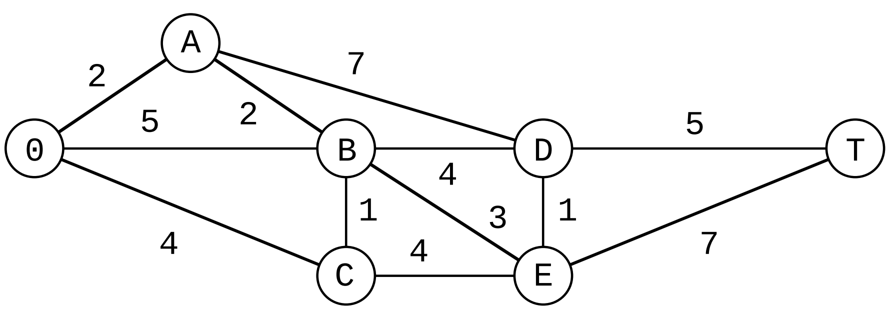
SHORTEST ELEMENTARY CHAIN

- Given node X0 in G find the shortest (cheapest) chain joining X0 with all the nodes of G

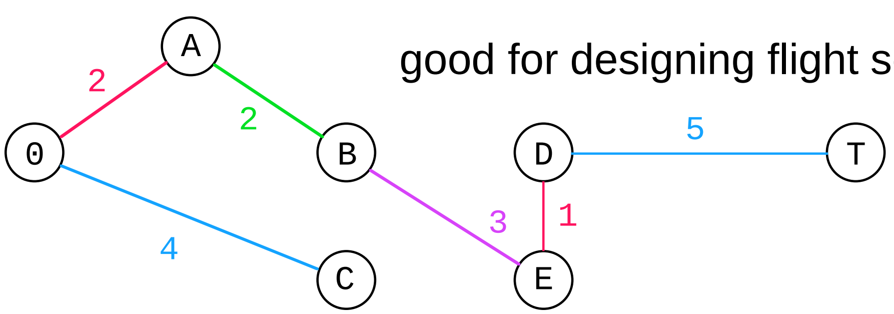
- Solution:
  C = connected nodes: initially X0
  U = unconnected nodes: initially all but X0
  E = set of edges: initially empty

  1. find the closest node in U to X0 (say X1)
  2. move X1 from U to C
  3. add (X0, X1) to E
  4. for each Xi in C find Yi in U that is closest; choose Ym such that cost from X0 to Ym is a minimum and add (Xm, Ym) to E; repeat until U is empty

Ex: start at node `0`



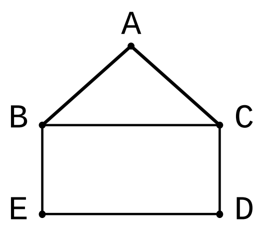
Result: `0(0,A)` `A(0,C)` or `(A,B)` `C(A,B)` `B(B,E)` `E(D,E)` or `(B,D)` `D(D,T)`

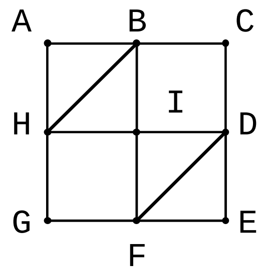


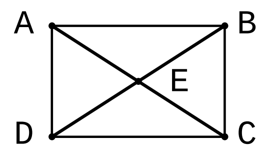
good for designing flight schedules

EULERIAN CHAINS AND CYCLES

- When is it possible to trace a planar graph without tracing any edge more than once so that the pencil is never removed from the paper?







- Eulerian cycle = edges are all the edges of G (end up at point where started)

- Theorem: an Eulerian cycle exists for a connected graph G whenever all nodes have an even degree and vice versa

- Proof: one direction: if an Eulerian cycle exists, then each time we enter a node by one edge we leave by another edge

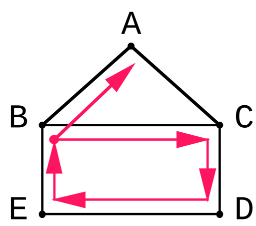
  other direction: more complex

- Eulerian chain = joins nodes X and Y such that its edges are all the edges of G (end up at point different from starting point)

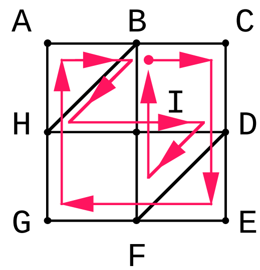
- Theorem: an Eulerian chain between nodes X and Y for a connected graph G exists if and only if nodes X and Y have odd degree and the remaining nodes have even degree

EULERIAN CHAINS AND CYCLES

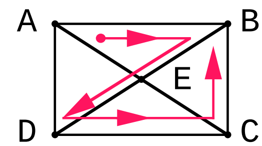
- When is it possible to trace a planar graph without tracing any edge more than once so that the pencil is never removed from the paper?



Eulerian chain



Eulerian cycle



Neither

- Eulerian cycle = edges are all the edges of G (end up at point where started)

- Theorem: an Eulerian cycle exists for a connected graph G whenever all nodes have an even degree and vice versa

- Proof: one direction: if an Eulerian cycle exists, then each time we enter a node by one edge we leave by another edge

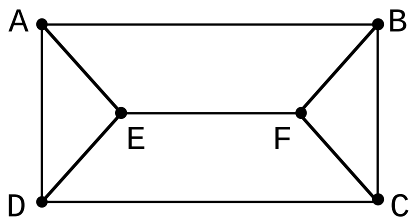
  other direction: more complex

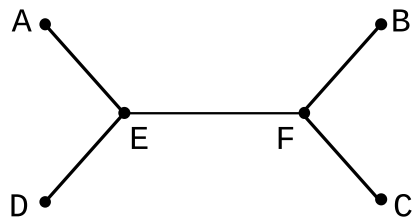
- Eulerian chain = joins nodes X and Y such that its edges are all the edges of G (end up at point different from starting point)

- Theorem: an Eulerian chain between nodes X and Y for a connected graph G exists if and only if nodes X and Y have odd degree and the remaining nodes have even degree

HAMILTONIAN CHAINS AND CYCLES

- When is it possible for a salesman based in city X to cover his territory in such a way that he never visits a city more than once, where not every city is connected directly to another city?



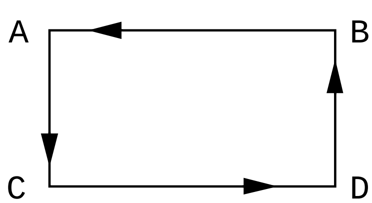


Hamiltonian cycle = cycle where each vertex appears once (salesman ends up at home!)
Hamiltonian chain = chain where each vertex appears once (salesman need not end up at home!)

- Unlike Eulerian chains and cycles, no necessary and sufficient conditions exist for a graph G to have a Hamiltonian chain or cycle
- Sufficient condition:

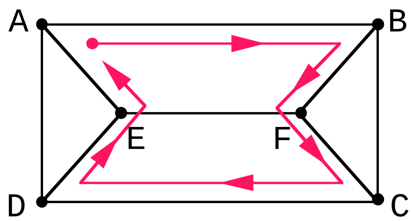
Theorem: A simple graph with $n \geq 3$ nodes such that for any distinct nodes X and Y not joined by an edge and degree (X) + degree (Y) $\geq n$, then G has a Hamiltonian cycle

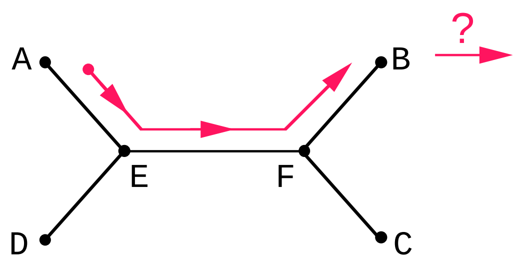
Ex:



HAMILTONIAN CHAINS AND CYCLES

- When is it possible for a salesman based in city X to cover his territory in such a way that he never visits a city more than once, where not every city is connected directly to another city?





Hamiltonian cycle exists
Hamiltonian chain exists
Hamiltonian cycle = A B F C D E A

No Hamiltonian chain or cycle
(only one way from ADE to BCF)

Hamiltonian cycle = cycle where each vertex appears once (salesman ends up at home!)

Hamiltonian chain = chain where each vertex appears once (salesman need not end up at home!)

- Unlike Eulerian chains and cycles, no necessary and sufficient conditions exist for a graph G to have a Hamiltonian chain or cycle

- Sufficient condition:
  Theorem: A simple graph with $n \geq 3$ nodes such that for any distinct nodes X and Y not joined by an edge and degree (X) + degree (Y) $\geq n$, then G has a Hamiltonian cycle

Ex: