5. Graphs

Part 2

Objectives

- Spanning Trees
  - Prim algorithm
  - Kruskal algorithm
- Eulerian Graphs

Minimum Spanning Tree (MTS)

Introduction

Suppose we wish to connect all the computers in a new office building using the least amount of cable. We can model this problem using an undirected, weighted graph *G* whose vertices represent the computers, and whose edges represent all the possible pairs (*u*,*v*) of computers, where the weight *w*(*u*,*v*) of edge (*u*,*v*) is equal to the amount of cable needed to connect computer *u* to computer *v*. Rather than computing a shortest-path tree from some particular vertex *v*, we are interested instead in finding a tree *T* that contains all the vertices of *G* and has the minimum total weight over all such trees. Algorithms for finding such a tree are the focus of this section.

MTS - Problem Definition

Given an undirected, weighted graph *G*, we are interested in finding a tree *T* that contains all the vertices in *G* and minimizes the sum

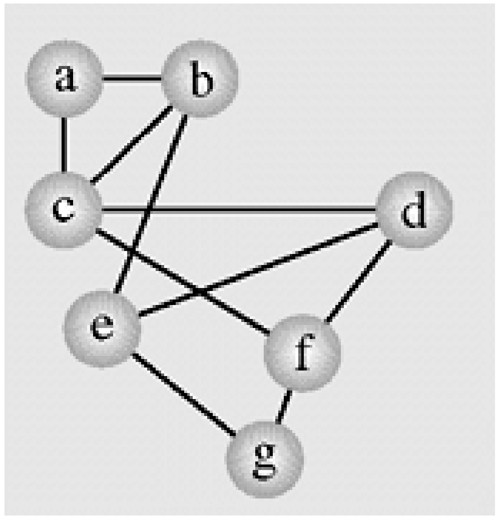
$$w(T) = \sum_{(u,v) \text{ in } T} w(u,v).$$

A tree, such as this, that contains every vertex of a connected graph *G* is said to be a ***spanning tree***, and the problem of computing a spanning tree *T* with smallest total weight is known as the ***minimum spanning tree*** (or ***MST***) problem.

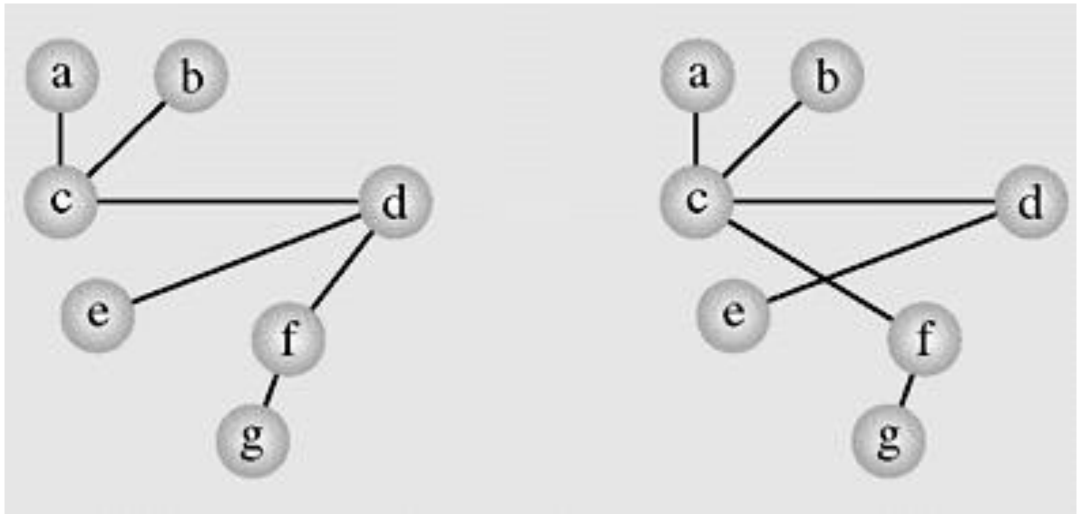
Spanning Tree example

In graph theory, a **tree** is an undirected, connected graph without simple cycles**.** A **spanning tree** is a tree that includes all vertices of the original graph

A graph representing the airline connections between seven cities



Two spanning tree of the graph



Spanning Tree Application

- Minimum Spanning Trees (MST) are useful in many applications, for example, finding the shortest total connections for a set of edges.
- If we are running cable to the nodes, representing cities, and <u>we wish to minimize cable cost</u>, the MST would be a viable option.
- An algorithms will be presented: Prim-Jarnık MST algorithm and Kruskal’s Algorithm.

MTS algorithms

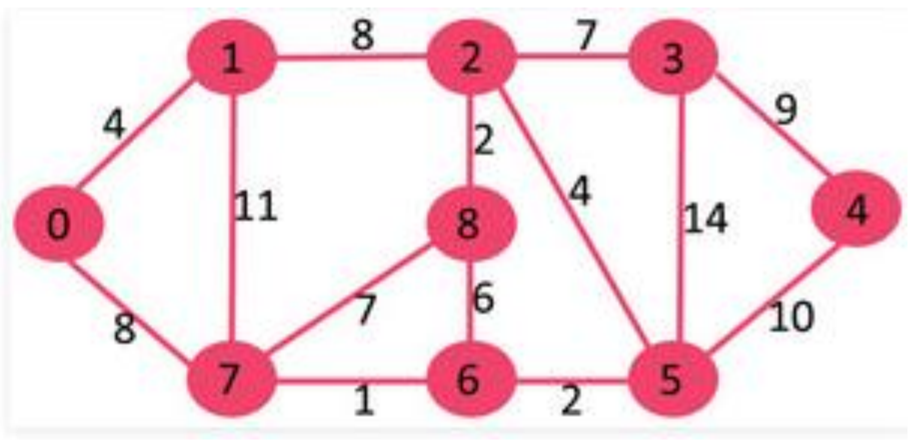
In this section, we discuss two classic algorithms for solving the MST problem. These algorithms are both applications of the ***greedy method***, which is based on choosing objects to join a growing collection by iteratively picking an object that minimizes some cost function. The first algorithm we discuss is the Prim-Jarn´ık algorithm, which grows the MST from a single root vertex, much in the same way as Dijkstra's shortest-path algorithm. The second algorithm we discuss is Kruskal's algorithm, which "grows" the MST in clusters by considering edges in ondecreasing order of their weights.

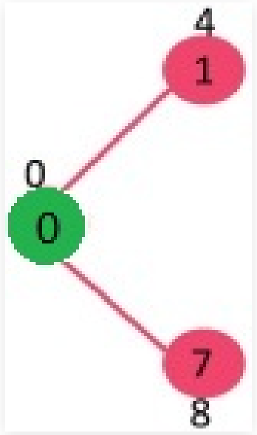
In order to simplify the description of the algorithms, we assume, in the following, that the input graph *G* is undirected (that is, all its edges are undirected) and simple (that is, it has no self-loops and no parallel edges). Hence, we denote the edges of *G* as unordered vertex pairs (*u*,*v*).

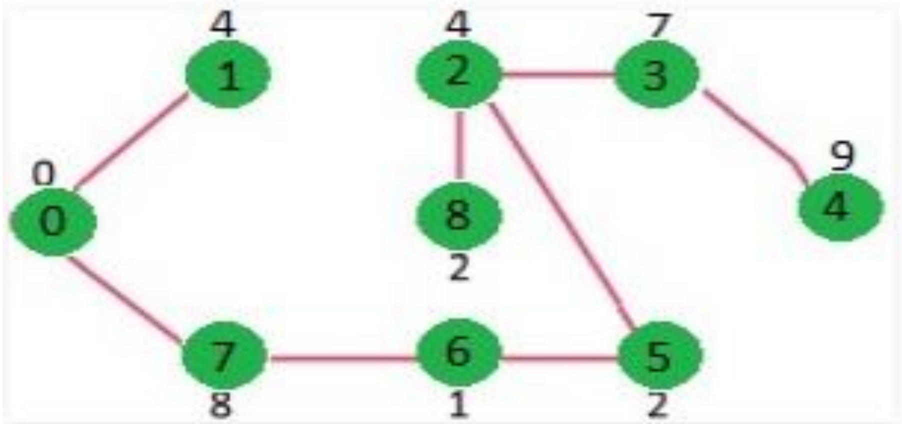
MTS Prim-Jarnik Algorithm

- Initialize a tree with a single vertex, chosen arbitrarily from the graph.
- Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
- Repeat step 2 (until all vertices are in the tree).

Prim-Jarnik Algorithm demo







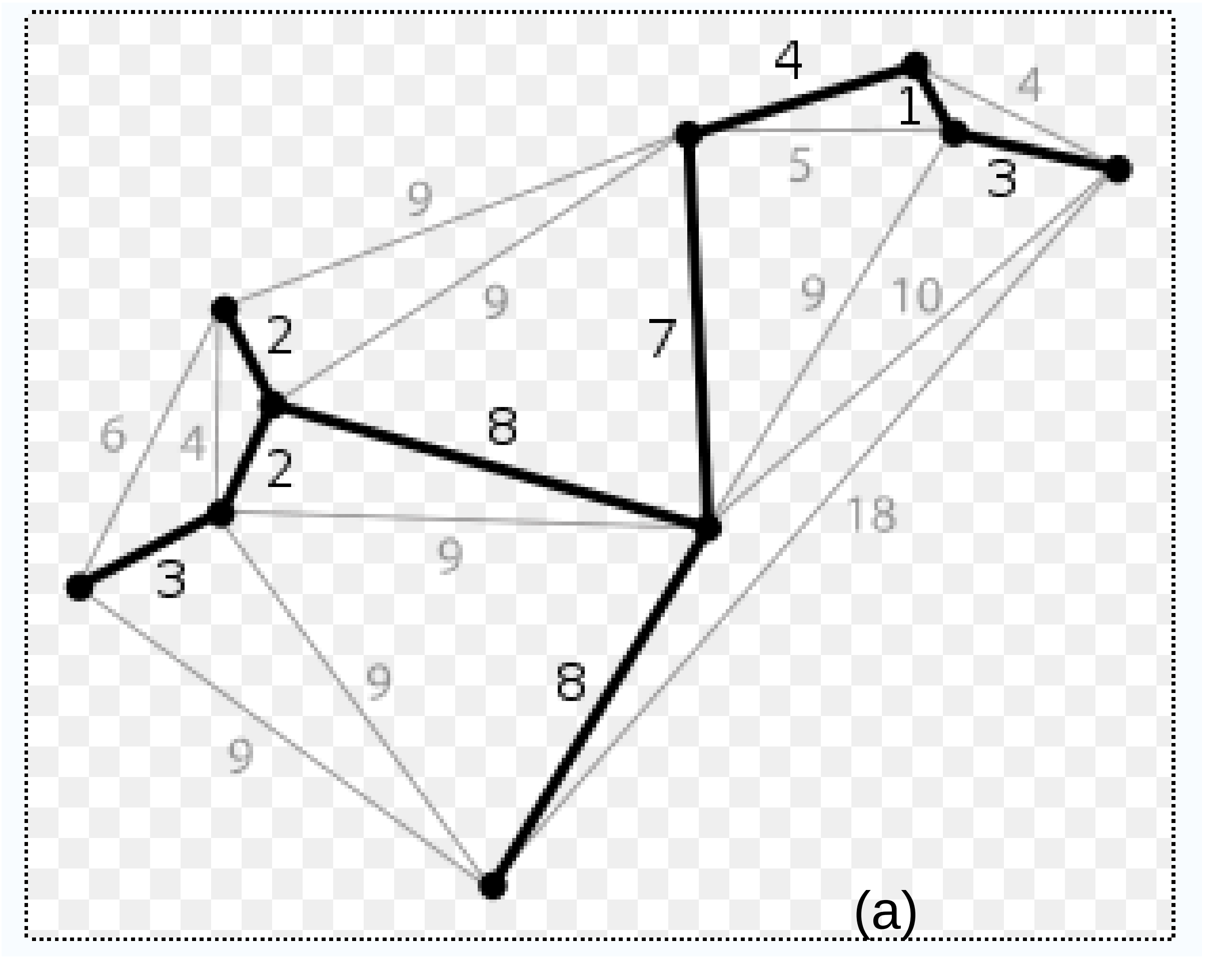
A spanning tree of graph (a) found with Prim's algorithm starting with the vertex 0. The vertices are selected in the following order: 0, 1, 7, 6, 5, 2, 8, 3, 9

Kruskal Algorithm

(A popular algorithm, all edges are ordered by weight, each edge is checked to see whether it can be considered part of the tree. It is added to the tree if no cycle arises after its inclusion)

```
KruskalAlgorithm(weighted connected undirected graph)
tree = null;
edges = sequence of all edges of graph sorted by weight
for (i = 1; i<=|E| and |tree|<|V| - 1; i++)
        if e_i from edges does not form a cycle with edges in tree
                add e_i to tree
```

Kruskal Algorithm – demo

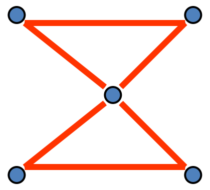


A spanning tree of graph (a) found with Kruskal's algorithm. The vertices are selected in the following order: 1, 2, 2, 3, 3, 4, 7, 8, 8

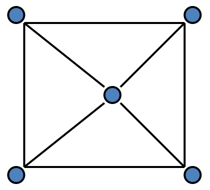
Euler cycle and paths

Euler path: a path traversing all the edges of the graph exactly once

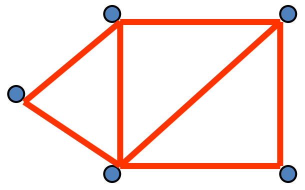
Euler cycle: a cycle traversing all the edges of the graph exactly once



Has Euler cycle

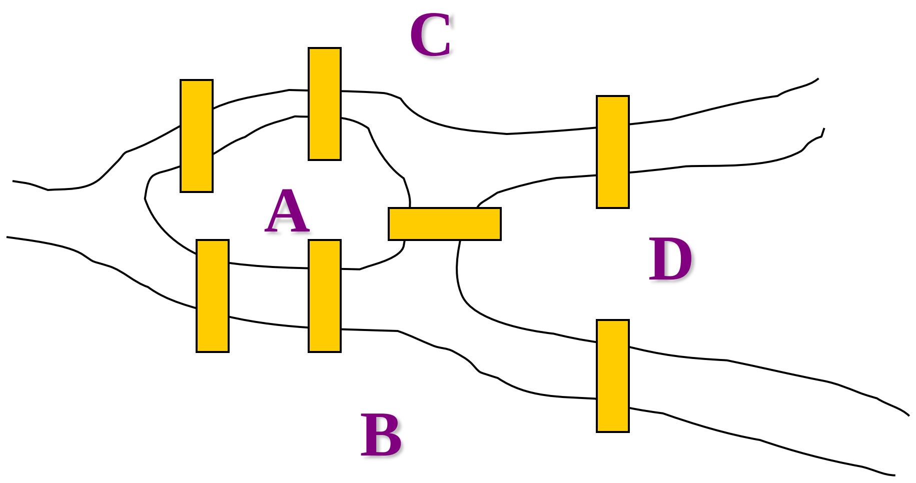


No Euler cycle
no Euler path



Has Euler path,
but no Euler cycle

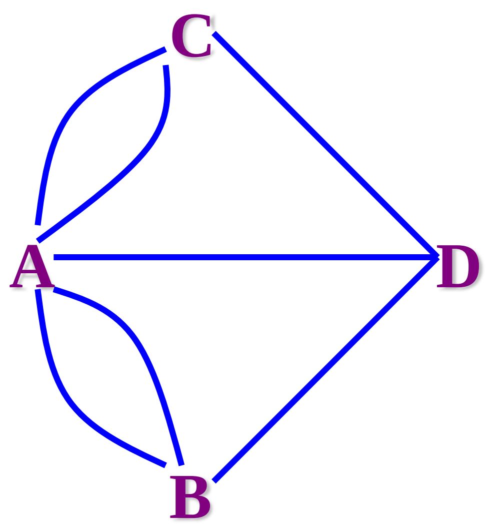
The Bridges of Königsberg



Is it possible to start at some location, travel across all the bridges without crossing any bridge twice, and return to the same starting point?

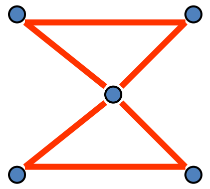
Kneiphof island on Pregel river and 7 bridges built in 18-th century in Königsberg town (Kaliningrad).

Rephrasing in terms of Euler cycles:

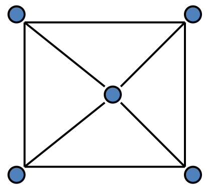


Necessary and sufficient conditions for Euler cycles

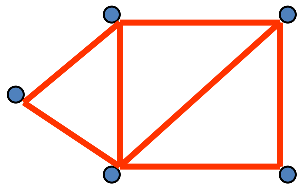
Theorem 1: A connected multigraph has an Euler cycle if and only if each of its vertices has even degree.



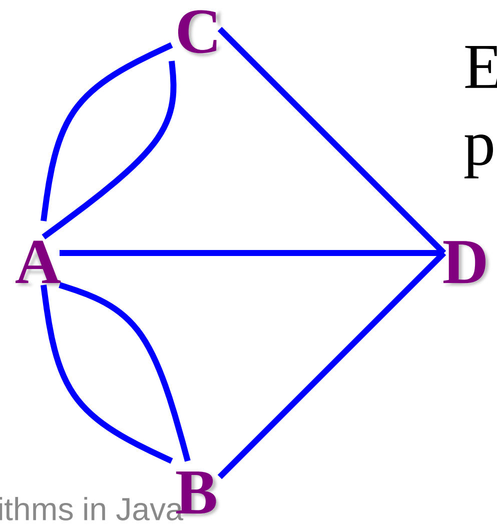
Has Euler cycle



No Euler cycle
no Euler path



Has Euler path,
but no Euler cycle



Euler cycle???
path???

Necessary condition for Euler cycle

Theorem 1: A connected multigraph has an Euler cycle if and **only if** each of its vertices has even degree.

Proof Sketch, PART 1:

- Assume the graph has an Euler cycle.
- Observe that every time the cycle passes through a vertex, it contributes 2 to the vertex's degree

(since the cycle enters via an edge incident with this vertex and leaves via another such edge)

Sufficient condition for Euler cycle

Theorem 1: A connected multigraph has an Euler cycle **if** and only if each of its vertices has even degree.

Proof Sketch, PART 2: Demonstrate an algorithm for finding the Euler cycle in a graph where all vertices have even degree.

- Assume every vertex in a multigraph ***G*** has even degree. Start at an arbitrary non-isolated vertex v_0, choose an arbitrary edge (v0,v1), then choose an arbitrary unused edge from v1 and so on. Then after a finite number of steps the process will arrive at the starting vertex v0, yielding a cycle with distinct edges.
- If the cycle includes all edges of G, this will be an Euler cycle; if not, begin the procedure again from a vertex contained in this cycle and splice the two cycles together; continue until all edges are used.

Note

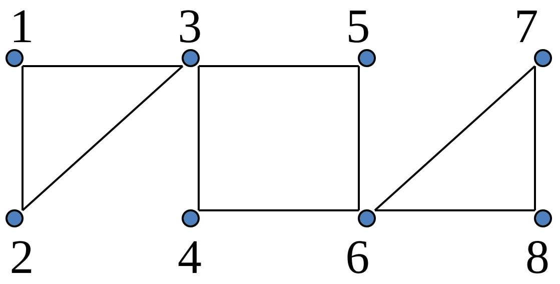
In the above procedure, once you entered a vertex v, there will always be another unused edge to exit v because v has an even degree and only an even number of the edges incident with it had been used before you entered it.
The only edge from which you may not be able to exit after entering it is v_0 (because an odd number of edges incident with v_0 have been used as you didn't enter it at the beginning) , but if you have reached v_0, then you have already constructed a required cycle.

A procedure for constructing an Euler cycle

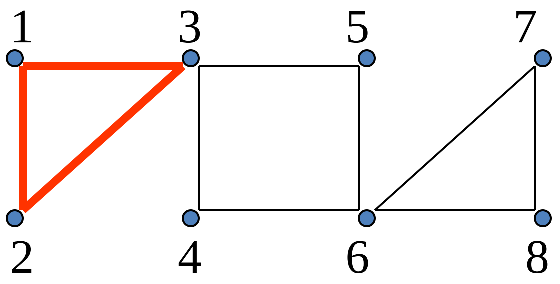
```
**Algorithm** Euler(G)
//Input: Connected graph G with all vertices having even degrees
//Output: Euler cycle

Construct a cycle in G
Remove all the edges of cycle from G to get subgraph H
while H has edges
    find a non-isolated vertex v that is both in cycle and in H
    //the existence of such a vertex is guaranteed by G's connectivity
    construct subcycle in H
    splice subcycle into cycle at v
    remove all the edges of subcycle from H
return cycle
```

Example

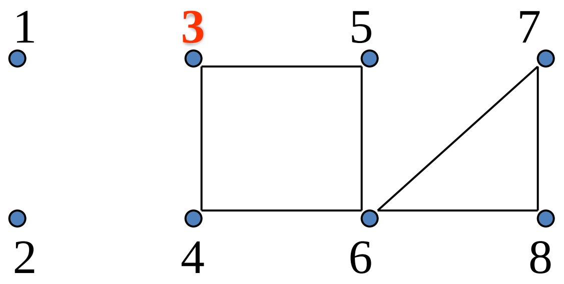
G



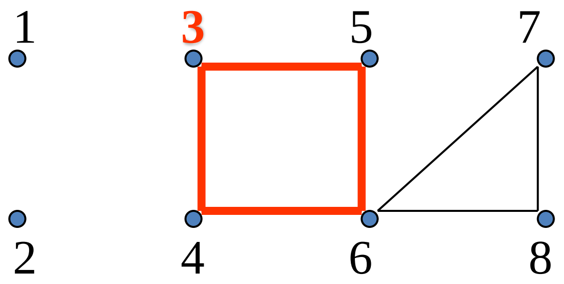
cycle
1231



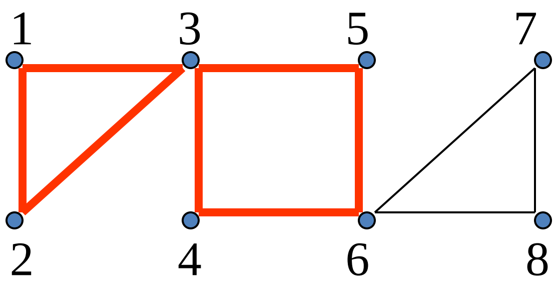
H



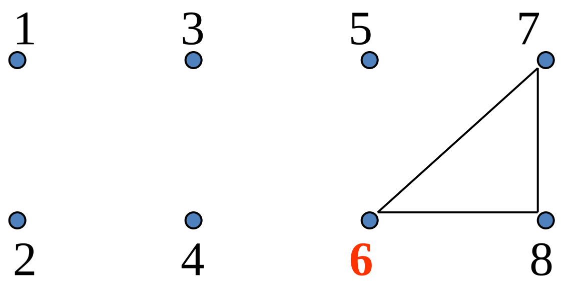
subcycle
34653



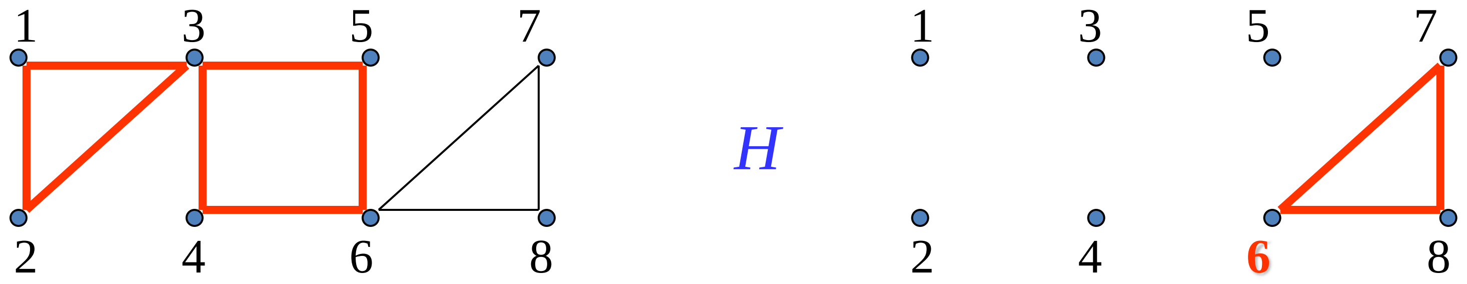
splicing
34653
into
1231



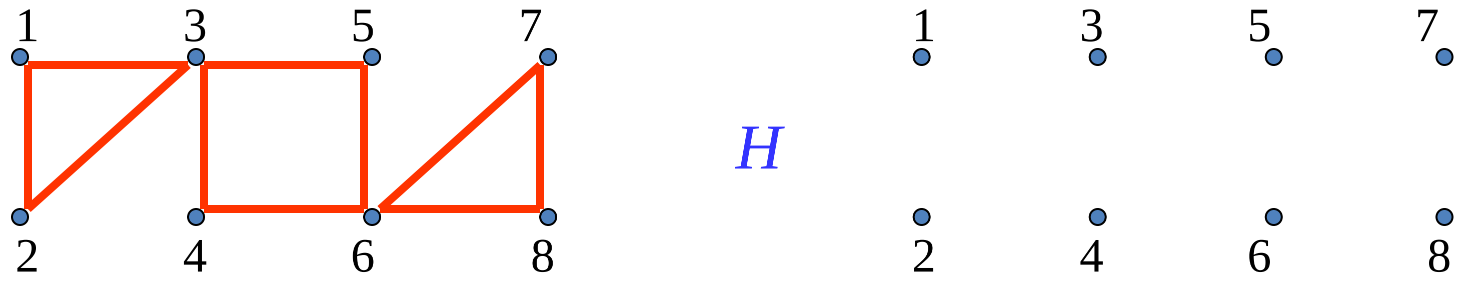
H



Example (cont.)



splicing
6786 into
12346531



Euler cycle obtained
12346786531

Algorithm for finding an Euler cycle from the vertex X using stack

Algorithm *Euler*(*G*)
//Input: Connected graph *G* with all vertices having even degrees
//Output: Euler cycle

```
declare a stack S of characters
declare empty array E (which will contain Euler cycle)
push the vertex X to S
while(S is not empty)
 {ch = top element of the stack S
  if ch is isolated then remove it from the stack and put it to E
   else
   select the first vertex Y (by alphabet order), which is adjacent
   to ch,push  Y  to S and remove the edge (ch,Y) from the graph
 }
 the last array E obtained is an Euler cycle of the graph
```

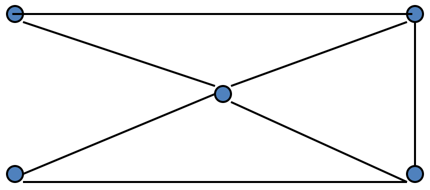
Necessary and sufficient conditions for Euler paths

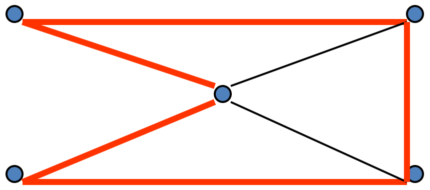
Theorem 2. A connected multigraph has an Euler path but not an Euler cycle if and only if it has exactly two vertices of odd degree.

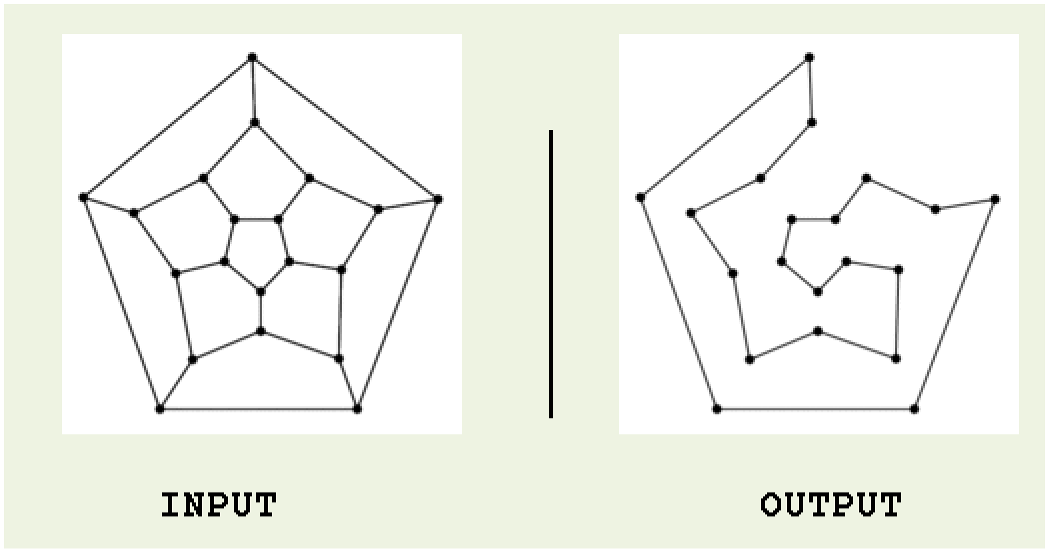
Hamilton paths and cycles - 1

Hamilton cycle: visits every vertex of the graph exactly once before returning, as the last step, to the starting vertex.

Examples:





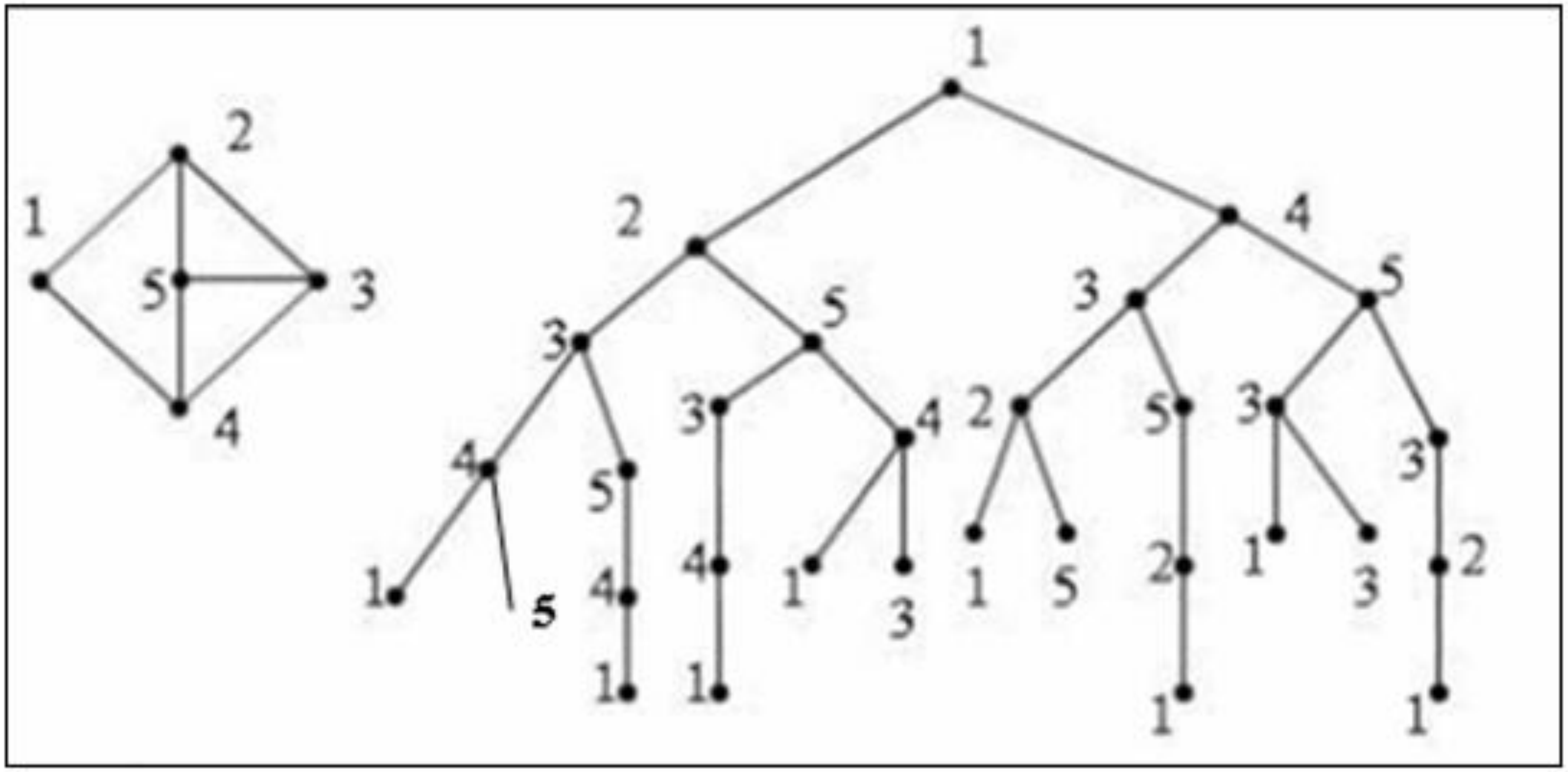


Hamilton path: visits every vertex of the graph exactly once.

Finding Hamilton’s cycles using Backtracking

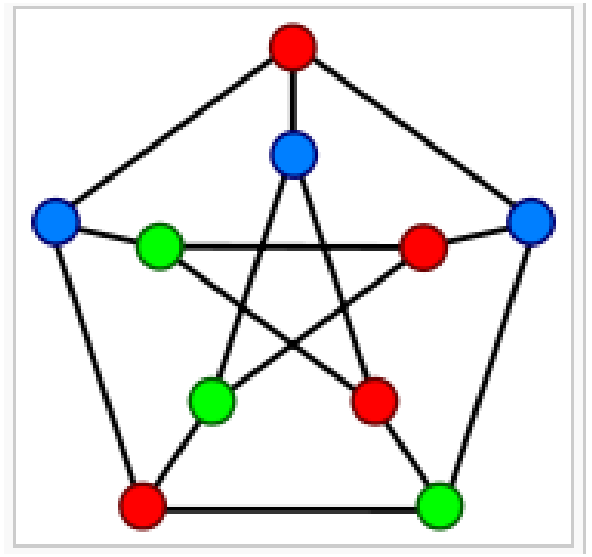
Given the graph G = (V,E) and X is a vertex of G. Suppose there exists at least one Hamilton Cycle for the graph. The following is a backtracking algorithm for finding one Hamilton cycle from the vertex X:
declare an empty array H (which will contain Hamilton cycle)
(1) Put the vertex X to H
(2) Check if H is a Hamilton cycle then stop, else go to (3)
(3) Consider the last vertex Y in H, if there is/are vertex(es) adjacent to Y, select an adjacent vertex Z and put it to H. If there no adjacent vertex, remove Y from H and denote it as a bad selection (so you do not select it in the same way again). Go to (2).

List all Hamilton’s cycles using Backtracking



Graph coloring - 1

- In graph theory, graph coloring is a way of coloring the vertices of a graph such that no two adjacent vertices share the same color.
- The chromatic number of a graph is the minimum number of colors one can use to color the vertices of the graph so that no two adjacent vertices are the same color.



- If the chromatic number of the graph G is denoted by χ(G). A graph for which k = χ(G) is called k-colorable. For a complete graph $\chi(K_n)=n$, $\chi(C_{2n})=2$, $\chi(C_{2n+1})=3$; and for bipartite graph $\chi(G) \leq 2$.
- Determining a chromatic number of a graph G is an NP-complete problem.

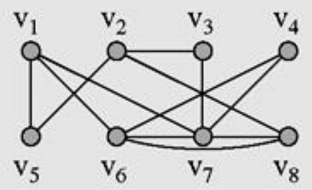
Graph coloring - 2

- **Sequential coloring** establishes the sequence of vertices and a sequence of colors before coloring them, and then color the next vertex with the lowest number possible

```
sequentialColoringAlgorithm(graph = (V, E))
    put vertices in a certain order $v_{P1}, v_{P2}, . . . , v_{Pv}$ ;
    put colors in a certain order $c_1, c_2, . . . , c_k$;
    for i = 1 to |V|
        j=the smallest index of color that does not appear in any neighbor of $v_{Pi}$;
        color($v_{Pi}$) = $c_j$;
```

The complexity of this algorithm is $O(|V|^2)$

Graph Coloring - 3



(a)

v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
c_1	c_1	c_2	c_1	c_2	c_2	c_3	c_4

(b)

v_7	v_6	v_1	v_2	v_8	v_3	v_4	v_5
c_1	c_2	c_3	c_1	c_3	c_2	c_3	c_2

(c)

v_7	v_6	v_1	v_8	v_4	v_2	v_5	v_3
c_1	c_2	c_3	c_3	c_3	c_1	c_2	c_2

(d)

(a) A graph used for coloring; (b) colors assigned to vertices with the sequential coloring algorithm that orders vertices by index number; (c) vertices are put in the largest first sequence; (d) graph coloring obtained with the Brélaz algorithm

Summary

- Spanning Trees
  - Prim algorithm
  - Kruskal algorithm
- Eulerian and Hamilton Graphs
- Graph coloring

Reading at home

Text book: Data Structures and Algorithms in Java

- 14.7 Minimum Spanning Trees - 662
- 14.7.1 Prim-Jarn´ık Algorithm - 664
- 14.7.2 Kruskal’s Algorithm - 667
- (Euler's tour and Euler's cycle, exercise C.14.5.2)