a b c d e f g h I j k l m n o p q r s t u v x y z

Two vertices u and v are said to be **adjacent**=**connected** or neighbors if there is an edge {u,v} that connects the two vertices  
An edge is said to be **incident** on u if u is one of its endpoints

graph G1(V1,E1) is **ISOMORPHIC** with graph G2(V2,E2) if there exists a one-to-one correspondence f: V1  V2 such that {u,v} is an edge of G1 if and only if {f(u),f(v)} is an edge of G2

To prove that graphs are not **isomorphic**:  
Prove that the number of vertices is not equal or  
Prove that the number of edges is not equal or  
Prove that there is a difference in degrees for the vertices or

A path P is called a **simple path** if all vertices in P are distinct  
A path P is called a **trail** if all edges are distinct

A path P = (v0, v1, v2, ….. , vn) is called **a closed path** if v0 is equal to vn  
A path P is called **a cycle** if it’s a closed path of length 3 or larger in which all vertices are distinct except v0 = vn

An edge e={b,e} in a connected graph G is called a **BRIDGE** when removing e graph G becomes disconnected  
A vertex v in a connected graph G is called a **CUTPOINT** when removing v and all edges incident on v results in a disconnected graph

Now actually what has to be proven is if it is possible to create a trail that starts and ends in two different vertices of G and uses all edges in this graph exactly once.   
Such a trail is called a **TRAVERSABLE TRAIL**.

A finite connected graph **with two odd vertices** is traversable. A traversable trail may begin at either odd vertex and will end at the other odd vertex.  
  
Any finite connected graph is **Eulerian** if and only if each vertex has an even degree

|  |  |
| --- | --- |
| **Fleury’s algorithm** Pick vertex u as the starting point Pick an unused edge to traverse, pick {u,1} Check if {u,1} is a bridge | To apply Fleury’s algorithm to a traversable graph, convert the traversable graph to an Eulerian graph by adding an extra vertex and connect that vertex to the two vertices with odd degree. |

In a **weighted graph** all edges are assigned a non-negative integer value that is called the edge's weight or length

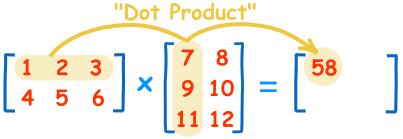
**The length of a path** in a weighted graph is equal to the sum of  
 the *weights* of all the edges in the path.

A indeg(A) = 2  
 outdeg(A)=1

*indeg(a)* = 0 🡺 *a* is a *source*  
*outdeg(f)* = 0 🡺 *f* is a *sink*

A directed graph only contains an Eulerian trail if and only if for each vertex u: indeg(u) = outdeg(u)

A graph is said to be **bipartite** if its vertices can be divided into two subsets M and N so that each edge of G is incident on a vertex of M and a vertex of N. Such a graph is denoted as Km,n where m and n represent the number of vertices in M and N (m≤n)

If M is the adjacency matrix of graph G   
the matrix Mn gives the number of paths with length n between 2 vertices of G





A subgraph T of a connected graph G is called a **spanning tree** of G if T is a tree and T includes all vertices of G  
We can use a kind of DFS-method and of BFS-method to find a spanning tree of a connected graph

|  |  |
| --- | --- |
| **KRUSKAL**  1. Arrange the edges of G in order of increasing weight:  7, 7, 9, 9, 10, 16, 24, 38, 70 2. Start with only the vertices of G 3. Process the list of edges sequentially, add all edges that don’t result in a cycle | **PRIM**  1. Choose a vertex v. This is your temporary tree T 2. Repeat until all vertices are connected:  3. Select an edge with minimum weight that has one vertex in T and the other vertex not in T 4. Add this edge and its destination vertex to T, which gives you a new temporary tree T |
| **Dijkstra**  Initially set l = *infinite* and p = *zero* for every vertex, except the starting vertex u which gets l = 0 and p = u  Now repeat until the end-vertex w is part of the temporary spanning tree T;  for all vertices in T consider all possible expansions  Expand T with that edge that has the smallest l of all possible expansions |  |

A **ROOTED TREE** is a tree with a designated vertex that is called the **ROOT**  
Every vertex that is no leaf (**All vertices with outdegree 0)** is called an **internal vertex**.  
In a rooted tree with root f, a branch Tf,(f,c) is the rooted tree that consists of f together with all vertices reachable from f using a path that starts with edge (f,c). The maximum vertex level is called the depth of the tree

|  |  |
| --- | --- |
| The **WEIGHT** of a branch Tv,e is equal to the number of edges in that branch. The ***CENTROID*** is the set of vertices of which the maximum weight of all branches is as small as possible | The **LEVEL** of a branch Tv,e is equal to the maximum distance from v to any of the leaves in the rooted tree  The ***CENTER*** is the set of vertices of which the maximum level of all branches is as small as possible. |
|  |  |

NB: domino problem:  
**The values on the domino stones are vertices (0 .. 4) and the domino stones themself represent edges between two vertices.**   
Finding a cycle of all 15 domino stones now is the same as finding the *Eulerian trail in the graph*.  
  
A **BINARY TREE**  is a rooted (directed) tree where out-deg(v)<=2 for every vertex v in the tree  
Each level has maximal 2level nodes

**A binary tree is complete if:**all levels, except possibly the last one, have the max number of nodes   
on the last level the nodes appear as far left as possible

There are three standard ways of traversing a binary tree T with root R. These three algorithms, called

preorder, - first time you are there you name it   
inorder, - second time you are there you name it  
and postorder, - last time you are there you name it  
are as follows:

|  |  |  |
| --- | --- | --- |
| **Preorder**: (1) Process !label! the root R.  (2) Traverse the left subtree of R in preorder.  (3) Traverse the right subtree of R in preorder. | **Inorder**: (1) Traverse the left subtree of R in inorder.  (2) Process the root R.  (3) Traverse the right subtree of R in inorder. | **Postorder**: (1) Traverse the left subtree of R in postorder.  (2) Traverse the right subtree of R in postorder.  (3) Process the root R. |

Polish notation of the   
mathematical expression Or it’s called the prefix form of the expression  
-- infix(inorder)

-- prefos(preorder)

 -- postfix(postorder)

**Huffman's algorithm**

Repeat:

Arrange the weights in increasing order

Remove the two smallest weights,  
Add them together  
Replace this sum at the right place

Go back to step 2  
  
At the end build the weighted tree

**Networks**

The flow over the cut = the flow from SRC to SNK minus the flow from SNK to SRC.

The capacity of the cut = the sum of all capacities of the arrows from SRC to SNK  
  
**Ford-Fulkerson algorithm:**

Initialize the flow of each edge to 0 : f(i,j) = 0

Repeat while there is still an augmenting path in N

Find such an augmenting path from src to snk.

Determine the flow increment value v:

If ‘bottleneck’ edge is in same direction,

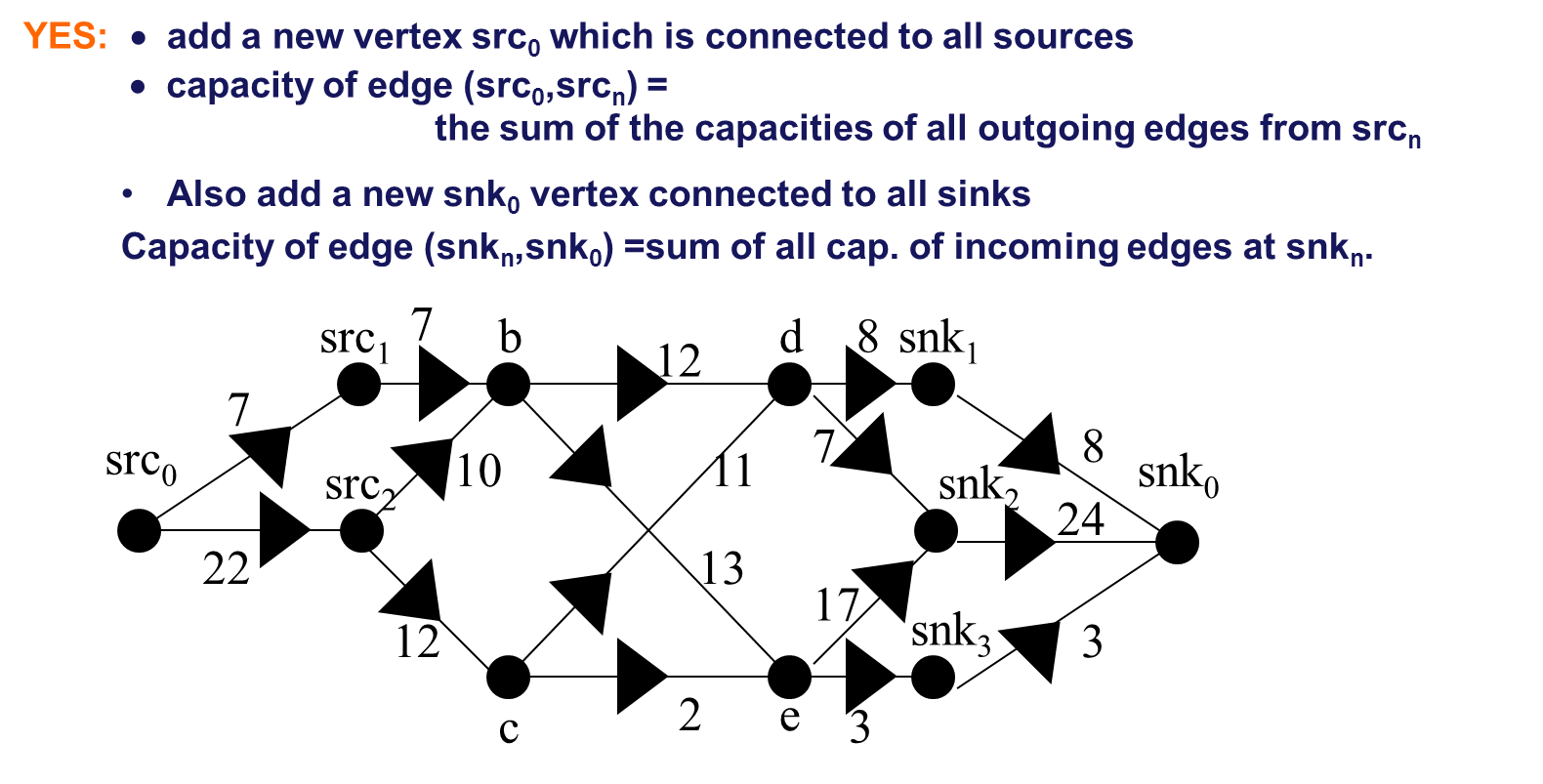
v = c(bottleneck edge) – f(bottleneck edge)

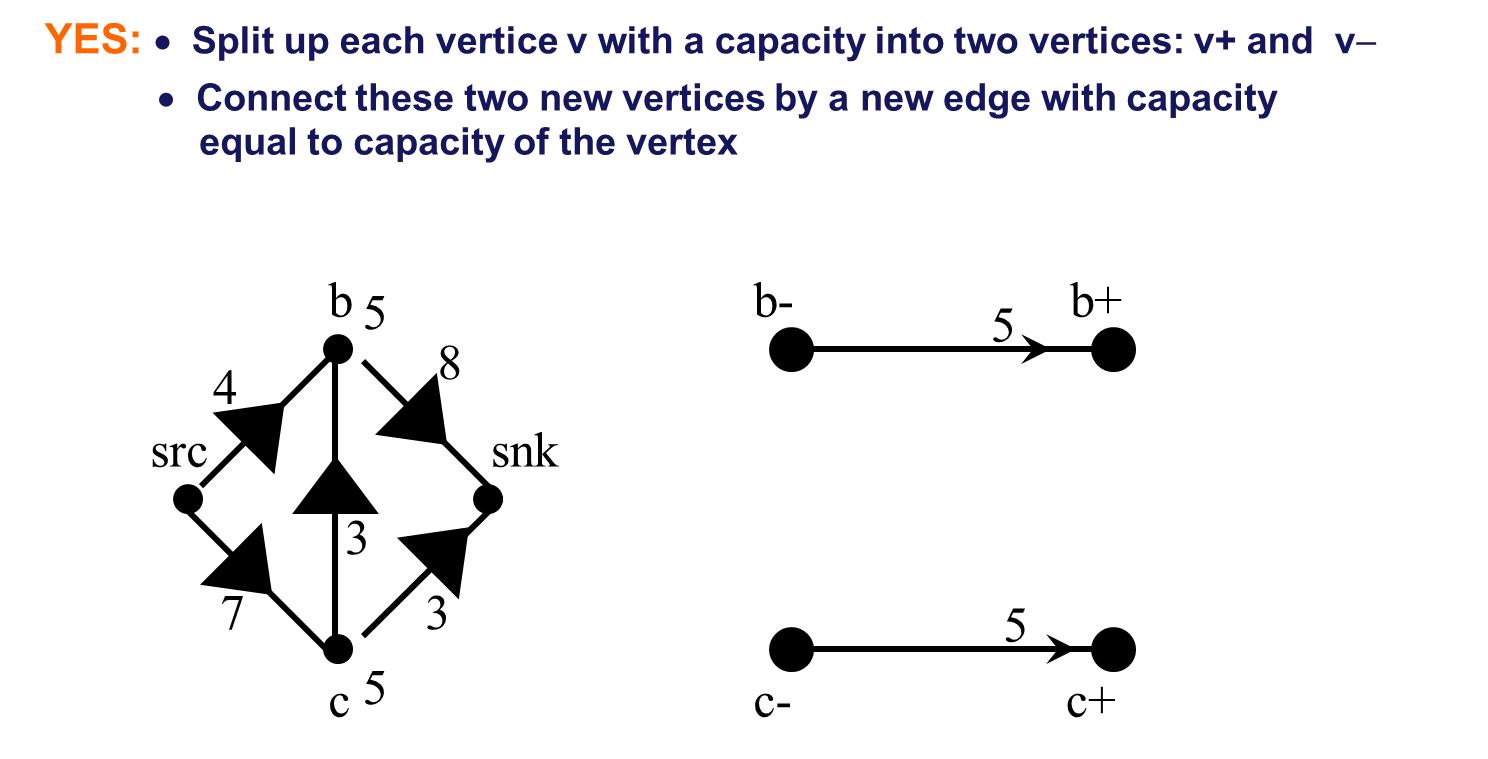
If ‘bottleneck’ edge is in the opposite direction

v = f(bottleneck edge) – 0

Increase the flow of all edges in the same direction in the augmenting path with v

Decrease the flow of all edges in opposite direction in the augmenting path with v





What is the maximum flow of this network?

Give SRC = {…, …., … } and SNK = {…, …., … } of the minimal cut. Use DFS.