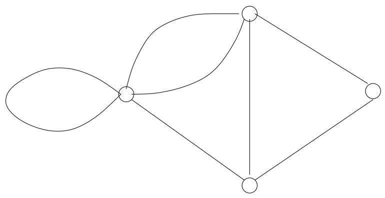
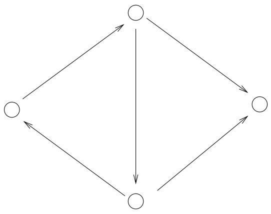
Graph Algorithms

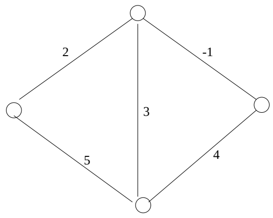
Definitions: A graph : $G(V, E)$

- V is a finite set of vertices
- E is a finite set of edges
- Each edge has two vertices $e = (u, v)$ where $e \in E$ and $u, v \in V$
- $e = (v, v)$ is possible (loop)
- More than 1 edge between 2 vertices is possible
- A simple graph has no multiple edges and has no loops

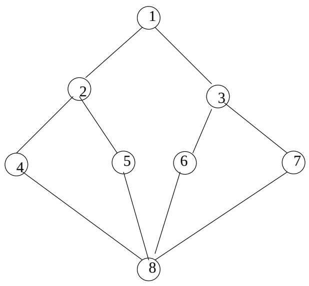


Directed Graphs and Weighted Graphs



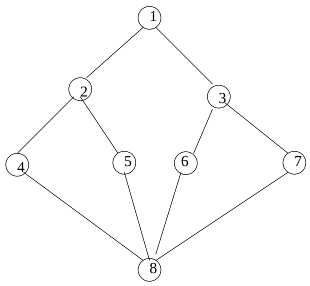


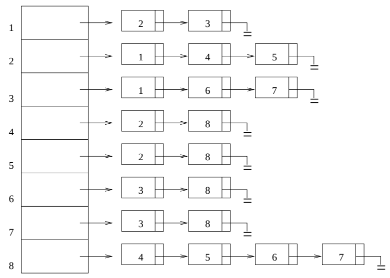
Storing Graphs : Adjacency Matrix



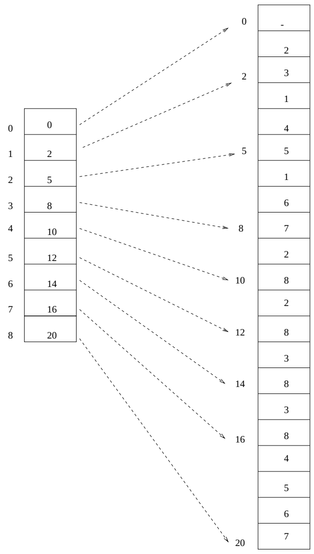
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	0	1	1	0	0	0
3	1	0	0	0	0	1	1	0
4	0	1	0	0	0	0	0	1
5	0	1	0	0	0	0	0	1
6	0	0	1	0	0	0	0	1
7	0	0	1	0	0	0	0	1
8	0	0	0	1	1	1	1	0

Storing Graphs : Adjacency List



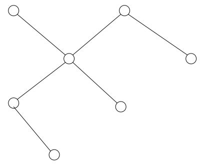


Storing Graphs : Compact Form of Adjacency List



Discussion
Pros and cons of the various methods?

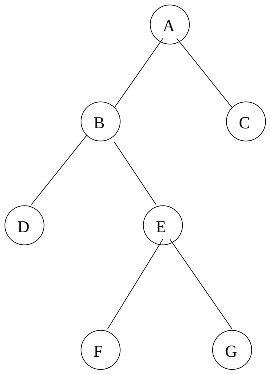
Trees



Let $G = (V, E)$ be an undirected graph. The following statements are equivalent:

- G is a tree
- Any two vertices in G are connected by a unique path
- G is connected, removing any edge will make it disconnected
- G is connected and has $|V| - 1$ edges
- G has no cycles and $|E| = |V| - 1$
- G has no cycles, but adding any edge adds a cycle

Basic Tree Traversal



3 Basic Traversal Techniques
- inorder
- preorder
- postorder

```
treenode = record
{
   Type data;
   treenode *left;
   treenode *right;
}
```


Tree Traversal Algorithms

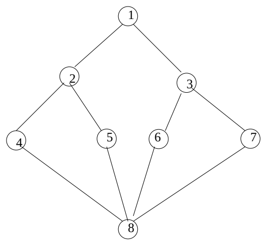
```
Inorder(t)
{
   if (t≠null) then
   {
      Inorder(t→left);
      visit(t);
      Inorder(t→right);
   }
}
Preorder(t)
{
   if (t≠null) then
   {
      visit(t);
      Preorder(t→left);
      Preorder(t→right);
   }
}
Postorder(t)
{
   if (t≠null) then
   {
      Postorder(t→left);
      Postorder(t→right);
      visit(t);
   }
}
```

Breadth First Search and Traversal

```
Algorithm BFS(v)
//A breadth first search of G is carried
//out beginning at vertex v.
//For any node i visited[i]=1 if i has already
//been visited.
//The graph G and array visited[]
//are global.
//visited[] is initialized to zero
{
   u:=v; // q is a queue of unexplored vertices
   visited[v]:=1;
   repeat
   {
      for all vertices w adjacent from u do
      {
         if (visited[w]=0) then
         {
            add w to q;//w is unexplored
            visited[w]:=1;
         }
      }
      if q is empty then return;
      //no unexplored vertex
      delete u from q;
      //Get first unexplored vertex
   }until(false)
}
```



```
Algorithm BFT(G,n)
//Breadth First Traversal of G
{
   for i:=1 to n do,
   //mark all vertices unvisited
      visited[i]:=0;
   for i:=1 to n do
      if (visited[i]=0) then BFS(i);
}
```



BFS sequence
(Using adjacency list)
1,2,3,4,5,6,7,8

Number of components in a graph?

Depth First Search

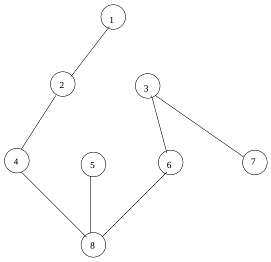
```
Algorithm DFS(v)
//Given an undirected (directed)
//graph G=(V,E) with n vertices
//and array visited[] initially set
//to zero, this algorithm visits all
//vertices reachable from v.
//G and visited[] are global
{
   visited[v]:=1;
   for each vertex w adjacent from v do
   {
      if (visited[w]=0)
         then
            DFS(w);
   }
}
```

DFS sequence of the earlier graph?

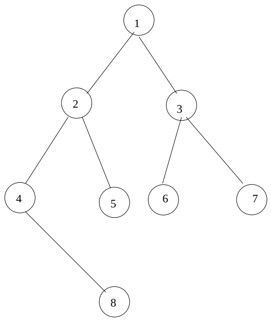
1,2,4,8,5,6,3,7

Connected Components and Spanning Trees

DFS Spanning Tree



BFS Spanning Tree



Exercise

1. Given an algorithm to check if an undirected graph is

   (a) A tree?

   (b) Bipartite?

   (c) Has a cycle of length x?

2. Given a directed graph $G(V, E)$ and a node $v \in V$. Write an efficient algorithm to decide whether there is a directed path from v to every other node in the graph

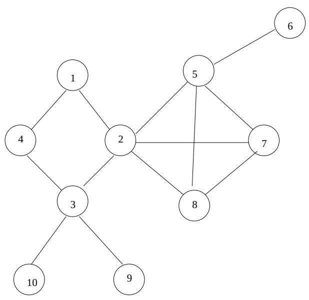
Biconnected Components and DFS

- undirected graph

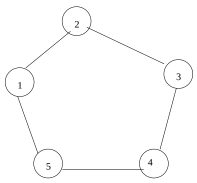
- a vertex v in a connected graph G is an **articulation point** iff the deletion of vertex v together with all the edges incident to v disconnects the graph into two or more non-empty components.

- a graph is biconnected iff it contains no articulation points.

An Example



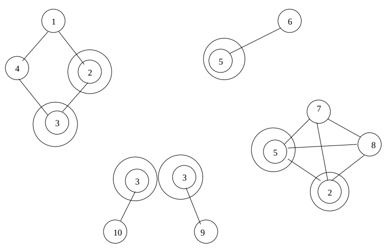
Identify all articulation points?



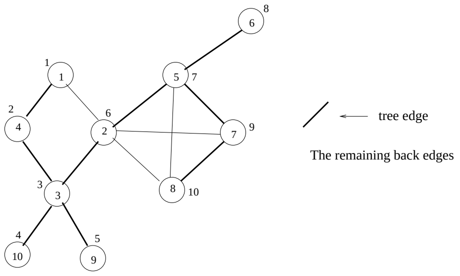
A Biconnected Graph

Scheme to Construct a Biconnected Graph

for each articulation point a do
{
Let $B_1, B_2, ..., B_k$ be the biconnected;
components containing vertex a;
Let $v_i, v_i \neq a$, be a vertex in $B_i, 1 \leq i \leq k_i$;
Add to G the edges (v_i, v_{i+1}) ,$1 \leq i < k$;
}



A Depth First Spanning Tree



dfn[i] is the depth first number of vertex i

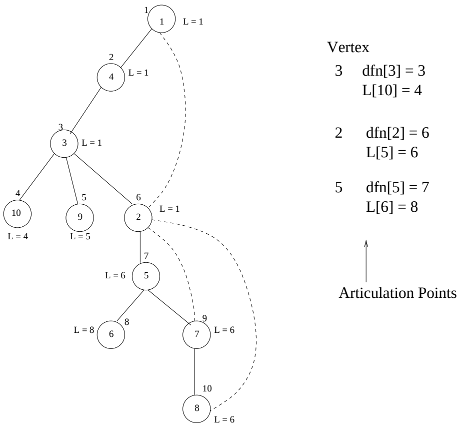
dfn[1] = 1
dfn[10]= 4
dfn[7] = 9
etc...

We define for each vertex u:

$L[u] = \min\{dfn[u],$
$\quad \min\{L[w] | w$ is a child of $u\},$
$\quad \min\{dfn[w] | (u, w)$ is a back edge$\}\}$

$L[u]$ is the lowest depth first number that can be reached from u using a path of descendents followed by at most one back edge

If u is not the root, then u is an articulation point iff u has a child w such that $L[w] \geq dfn[u]$



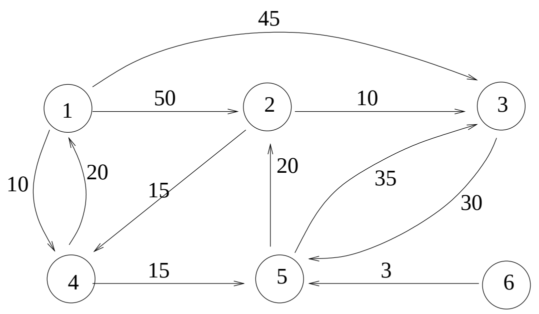
$L(u)$ can be easily computed if the vertices of the depth first spanning tree are visited in postorder

L[1:10] = {1,1,1,1,6,8,6,6,5,4}

Algorithm $Art(u, v)$
//u is a start vertex for the dfs.
//v is its parent, if any, in the depth first
//spanning tree. It is assumed that the
//global array dfn is initialized to zero
//and that the global variable num is initialized
//to 1. n is the number of vertices in G.

```
{
   dfn[u] := num;
   L[u] := num;
   num := num + 1;
   for each vertex w adjacent from u do
   {
      if (dfn[w] = 0) then
      {
         Art(w, u); // w is unvisited
         L[u] := min (L[u], L[w]);
      }
      else if (w ≠ v) then L[u]:=min(L[u], dfn[w]);
   }
}
```

Single-Source Shortest Paths



Path	Length
$1 \rightarrow 4$	10
$1 \rightarrow 4 \rightarrow 5$	25
$1 \rightarrow 4 \rightarrow 5 \rightarrow 2$	45
$1 \rightarrow 3$	45
$1 \rightarrow 6$	Cannot be reached


```
Algorithm ShortestPath(v, cost, dist, n)
//dist[j], 1 ≤ j ≤ n, is set to the length of
//the shortest path from vertex v to vertex j
//in a digraph G with n vertices
//G is represented by its cost adjacency matrix
//cost[1 : n, 1 : n]
{
   for i := 1 to n do
   { //initialize S
      S[i] := false; dist[i] := cost[v, i];
   }
   S[v] := true; dist[v] := 0.0 //put v in S
   for num := 2 to n do
   {
      choose u from among those vertices
      not in S such that dist[u] is minimum;
      S[u] := true; //put u in S
      for (each w adjacent to u with
         S[w] = false) do
      //update distances
      if (dist[w] > dist[u] + cost[u, w])
         then dist[w] := dist[u] + cost[u, w];
   }
}
```


All Pairs Shortest Paths

We can reuse Dijkstra's greedy algorithm (all edges must be non-negative)

Another method?

Assume no negative cycle

Let us examine a shortest path from i to $j, i \neq j$

1. May have some (or no) intermediate vertices
2. No cycle
3. If k is an intermediate vertex on this shortest path, then subpaths from i to k and k to j must be shortest paths from i to k and k to j respectively
4. If k is the vertex with the highest index, then i to k is the shortest path going through no vertex with index greater then $k-1$. Same from k to j.

Let $A^k(i,j)$ represent the length of a shortest path from i to j going through no vertex of index greater then k

$A(i,j) = \min\{\min_{1 \le k \le n}\{A^{k-1}(i,k) + A^{k-1}(k,j)\}, cost(i,j)\}$

Let
$A^0(i,j) = cost(i,j)$
$A^k(i,j) = \min\{A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\}$

*Not true for graphs with negative cycles.

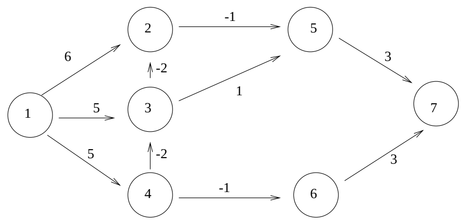
When neither i nor j equals to k, $A^{k-1}(i,j)$ is used only in the computation of $A^k(i,j)$. So $A^k(i,j)$ can overwrite $A^{k-1}(i,j)$.

When i equals to k, $A^{k-1}(i,j)$ equals $A^k(i,j)$. $A^k(k,j) = \min\{A^{k-1}(k,j), A^{k-1}(k,k) + A^{k-1}(k,j)\} = A^{k-1}(k,j)$
So when i equals to k, $A^k(i,j)$ can overwrite $A^{k-1}(i,j)$.
Similarly when j equals to k, $A^k(i,j)$ can overwrite $A^{k-1}(i,j)$.
So, in all cases $A^k(i,j)$ can overwrite $A^{k-1}(i,j)$.

Hence, superscript is not needed.

Algorithm AllPaths($cost, A, n$)
//$cost[1:n, 1:n]$ is the cost adjacency matrix of
//a graph with n vertices;
//$A[i,j]$ is the cost of the shortest path from
//vertex i to vertex j. $cost[i,i] = 0.0$
//for $1 \le i \le n$.
{
  for $i := 1$ to n do
    for $j := 1$to n do
      $A[i.j] := cost[i,j]$;
  for $k := 1$ to n do
    for $i := 1$ to n do
      for $j := 1$ to n do
        $A[i,j] = min(A[i,j], A[i,k] + A[k,j])$;
}

Single-Source Shortest Paths with General Weights



Algorithm BellmanFord($v, cost, dist, n$)
//Single source/all destinations shortest
//paths with negative edge costs
{
  for $i := 1$ to n do
    $dist[i] := cost[v,i]$;
  for $k := 2$ to n do
    for each u such that $u \ne v$ and
      u has at least an incoming edge do
      for each $< i, u >$ in the graph do
        if $dist[u] > dist[i] + cost[i,u]$
          then $dist[u] := dist[i] + cost[i,u]$;
}

Exercise

1. Using the graph on the previous page, manually go through the BellmanFord algorithm.

2. Arbitrage is the trading of one currency for another with the hopes of taking advantage of small differences in conversion rates among several currencies in order to achieve a profit. For example, if \$1.00 in U.S. currency buys 0.7 British pounds currency, 1 pound in British currency buys 9.5 French francs, and 1 French franc buys 0.16 in U.S. dollars, then an arbitrage trader can start with \$1.00 and earn $1 \times 0.7 \times 9.5 \times 0.16 = 1.064$ dollars thus earning a profit of 6.4 percent.

   You will write a program that determines whether a sequence of currency exchanges can yield a profit as described above.

   To result in successful arbitrage, a sequence of exchanges must begin and end with the same currency, but any starting currency may be considered.

Longest Path Problem

Consider the project of building a garage. The separate activities involved, and their estimated durations, are given in the table below:

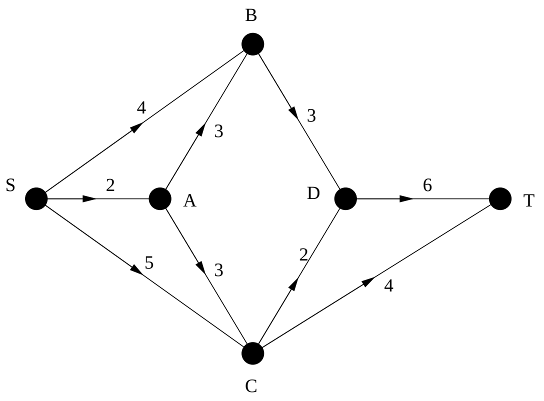
Activity	Duration	Preceding Activities
A	7	-
B	2	-
C	15	-
D	8	E
E	10	A,B
F	2	D,G
G	5	E
H	8	G
I	2	C,F
J	3	I

1. What is the earliest completion time for the project?

2. Where are the critical paths?

Maximum Flow

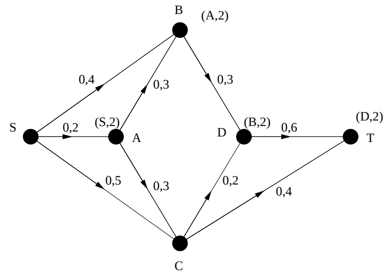
Find a maximum flow and a minimum cut in the following basic network



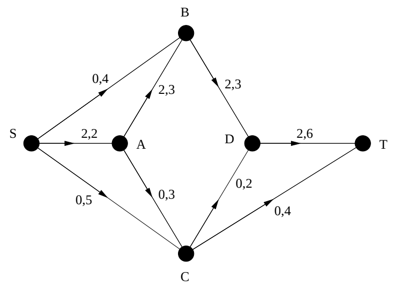
Maximum Flow

First iteration

Start with zero flow

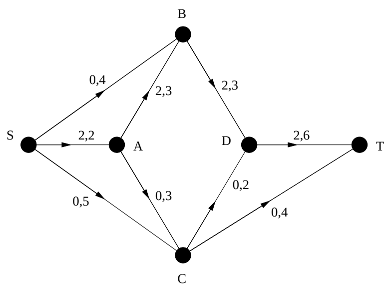


Find a flow augmenting path

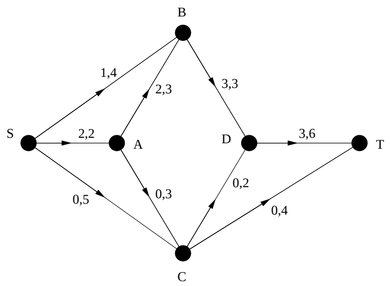


Maximum Flow

Second iteration

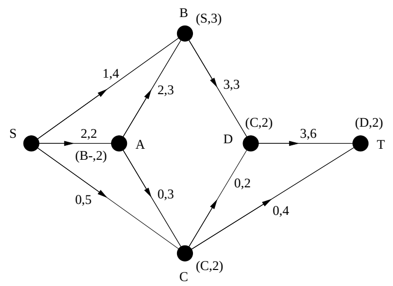


Increase flow along SBDT by 1

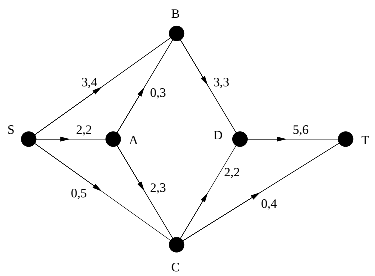


Maximum Flow

Third iteration

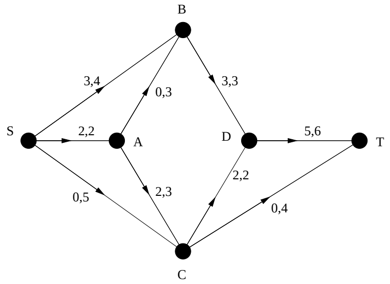


Increase flow along SBACDT by 2

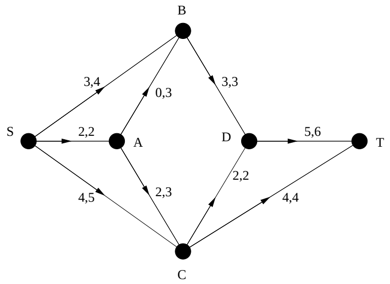


Maximum Flow

Fourth iteration

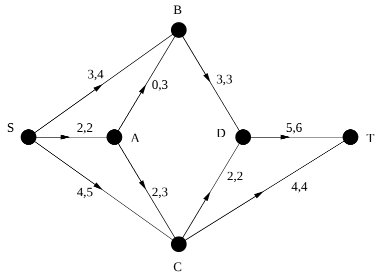


Increase flow along SCT by 4



Maximum Flow

Fifth iteration

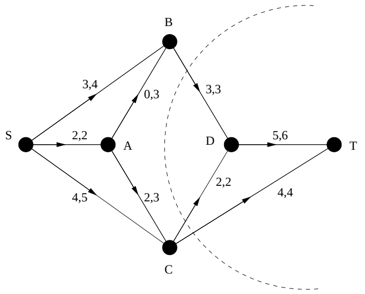


No more flow-augmenting paths

Maximum flow : 5 + 4 = 9 = 2 + 1 + 2 + 4

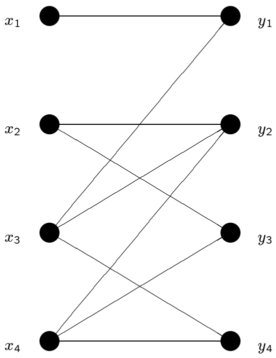
Minimum cut {S,A,B,C},{D,T}

has capacity 3 + 2 + 4 = 9

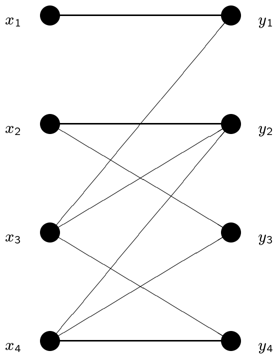


The Maximum Matching Problem

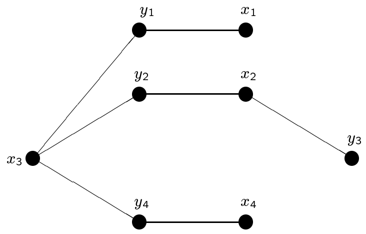
Find a maximum matching in the following bipartite graph:



Step 1: Perform a Greedy Assignment

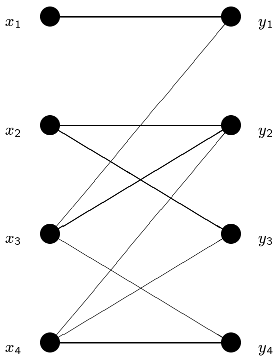


Step 2 (Matching Improvement) : Find Alternating Paths



- An alternating path starts with an "un-matched" vertex. It will alternate between using an unmatched edge and matched edge along the path.
- If an alternating path ends with a "un-matched" vertex, we can improve the matching by exchanging the matched and unmatched edges in the path.
- x_3 is an unmatched vertex.
- $x_3y_2x_2y_3$ is an alternating path that ends with an unmatched vertex y_3.
- Matching can be increased by making (x_3, y_2) and (x_2, y_3) into matched edges and (y_2, x_2) into an un-match edge.

Revised Matching



Since the revised matching has 4 edges, it is the maximum matching.

The Maximum Matching Algorithm

1. Find an initial matching using any method. A greedy based method will be sufficient.
2. Find an unmatch vertex. Using this unmatched vertex, find an alternating path that ends with another unmatched vertex. If possible, revised the matching.
3. Repeat Step 2 until no improvement can be done.

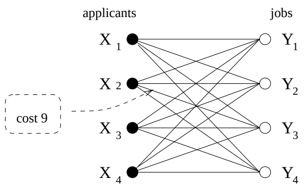
Hungarian Algorithm for the Assignment Problem

Find the optimum assignment in the following case

jobs

applicants		Y_1	Y_2	Y_3	Y_4
	X_1	6	12	15	15
	X_2	4	8	9	11
	X_3	10	5	7	8
	X_4	12	10	6	9

cost matrix



bipartite graph K 4,4

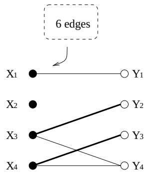
Construct initial partial graph

weights		Y_1	Y_2	Y_3	Y_4
6	X_1	0	6	9	9
4	X_2	0	4	5	7
5	X_3	5	0	2	3
6	X_4	6	4	0	3

weights		Y_1	Y_2	Y_3	Y_4
6	X_1	0	6	9	6
4	X_2	0	4	5	4
5	X_3	5	0	2	0
6	X_4	6	4	0	0

6 zeros

first revised cost matrix

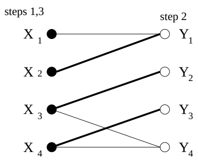


first partial graph

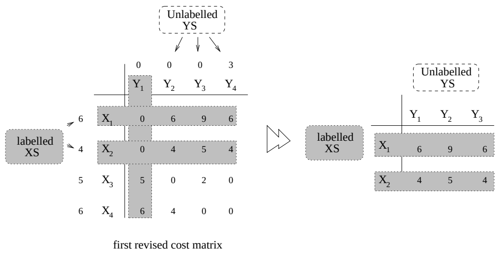
Label vertices

	0	0	0	3
	Y_1	Y_2	Y_3	Y_4
X_1	0	6	9	6
X_2	0	4	5	4
X_3	5	0	2	0
X_4	6	4	0	0

first revised cost matrix

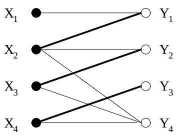


labelled partial graph



		4	0	0	3
		Y_1	Y_2	Y_3	Y_4
10	X_1	0	2	5	2
8	X_2	0	0	1	0
5	X_3	9	0	2	0
6	X_4	10	4	0	0

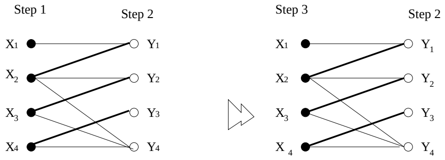
second revised cost matrix



second partial graph

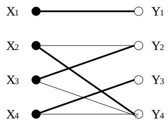
second partial graph

labelled partial graph



An alternating path is $Y_4 X_2 Y_1 X_1$

Step 5 : revised matching is:



New matching $X_3 Y_2$, $X_4 Y_3$, $X_1 Y_1$, $X_2 Y_4$

	Y_1	Y_2	Y_3	Y_4
X_1	6	12	15	15
X_2	4	8	9	11
X_3	10	5	7	8
X_4	12	10	6	9

original cost matrix

	-4	0	0	3
	Y_1	Y_2	Y_3	Y_4
X_1	0	2	5	2
X_2	0	0	1	0
X_3	9	0	2	0
X_4	10	4	0	0

final revised cost matrix

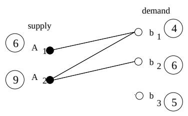
optimum assignment : X_1Y_1, X_2Y_4 , X_3Y_2 , X_4Y_3

TOTAL COST: 6 + 11 + 5 + 6 = 28 = 10 + 8 + 5 + 6 - 4+ 0 + 0 + 3

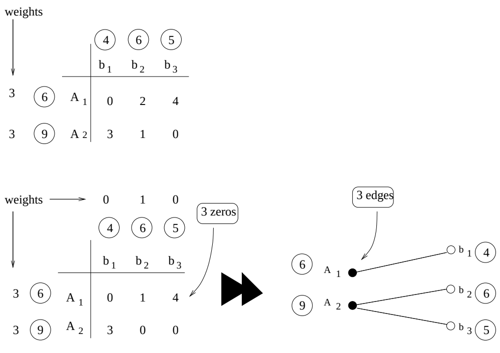
Find a minimum-cost solution to the transportation problem for the following situation

			demand	
		4	6	5
supply		b_1	b_2	b_3
6	A_1	3	5	7
9	A_1	6	4	3

cost matrix

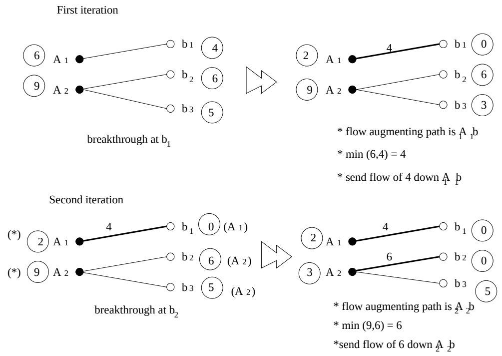


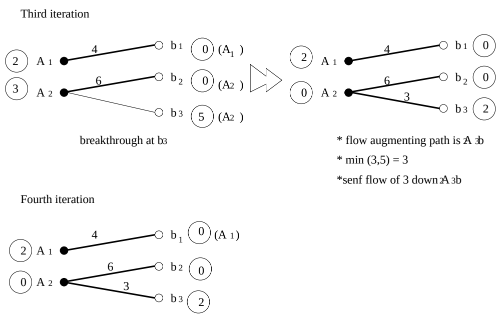
bipartite graph

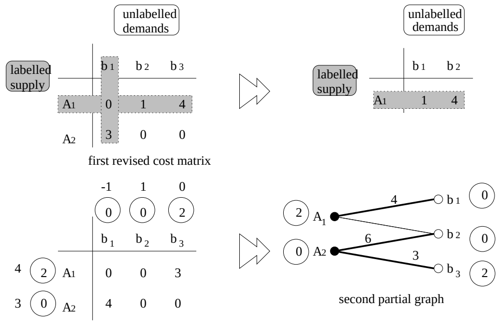


first revised cost matrix

first partial graph



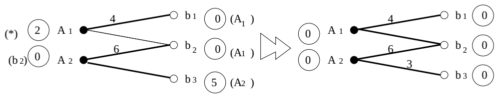




first revised cost matrix

second partial graph

second revised cost matrix



breakthrough at b_3

* flow augmenting path is $b_3 b_2 A_2 b_1 A$
* min (6,2,2) = 2
* increase flow by 2 on $A_1 b_2$, $A_2 b_3$
*decrease flow by 2 on $A_2 b_1$

maximum backward
flow along A b is 6

Final flow is	4 along A_1 b_1	with cost	(4 X 3)
	2 along A_1 b_2		+ (2 X 5)
	4 along A_2 b_2		+ (4 X 4)
	5 along A_2 b_3		+ (5 X 3) = 53