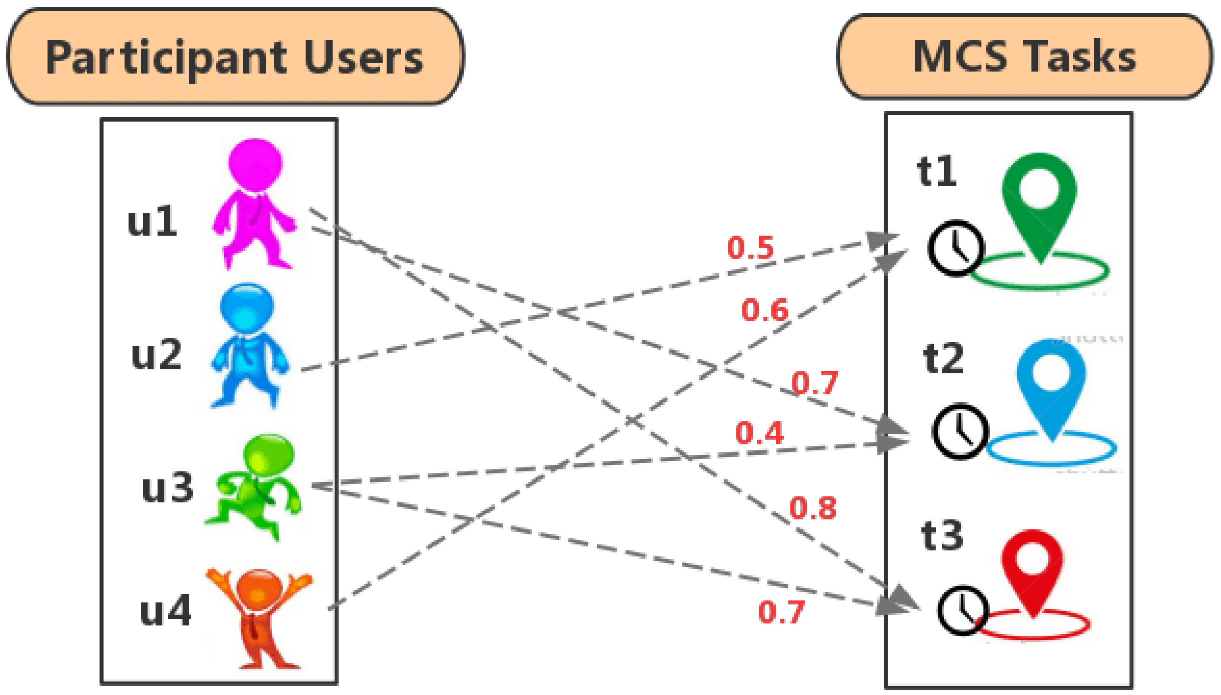
TASK ALLOCATION using Maximum Bipartite Matching



TEAM MEMBERS:

NAME	ADMISSION NO
KRINA PATEL	U19CS008
BRIJESH ROHIT	U19CS009
DHRUV GANDHI	U19CS015
SUMIT SHETTY	U19CS038

EXECUTIVE SUMMARY

- There are **N** tasks and **M** machines.
- Each machine can only accomplish **one task** and each task can be assigned to only **one machine**.
- Not all tasks can be completed by all machines, which is decided by range of machine up to which it can perform and this is the input/property value of machines.
- A task is also measured by an input/property value, which can be less than or equal to machine's range to which it is allocated.
- ***We have to find an allocation in which maximum tasks are completed and maximum utilization of machines is found.***

Content:

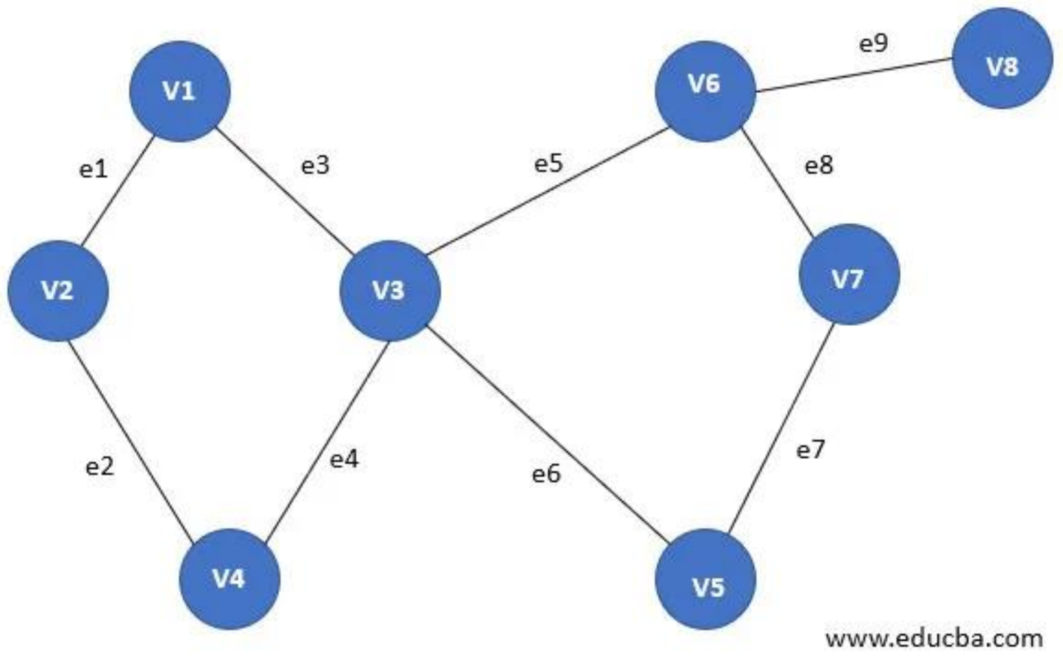
Introduction to graph.

A **graph (V, E)** is a set of **vertices V1, V2…Vn** and set of edges **E = E1, E2,….En**.

Here, each distinct edge can identify using the unordered pair of vertices (Vi, Vj).

2 vertices Vi and Vj are said to be adjacent if there is an edge whose endpoints are Vi and Vj. Thus E is said to be a connect of Vi and Vj.

In the example below, circles represent vertices, while blue lines connecting the circles represent edges.



- **Order of the graph** = The number of vertices in the graph.
- **Size of the graph** = The number of edges in the graph.
- **Degree of a vertex of a graph** = Number of edges incident to the vertex.

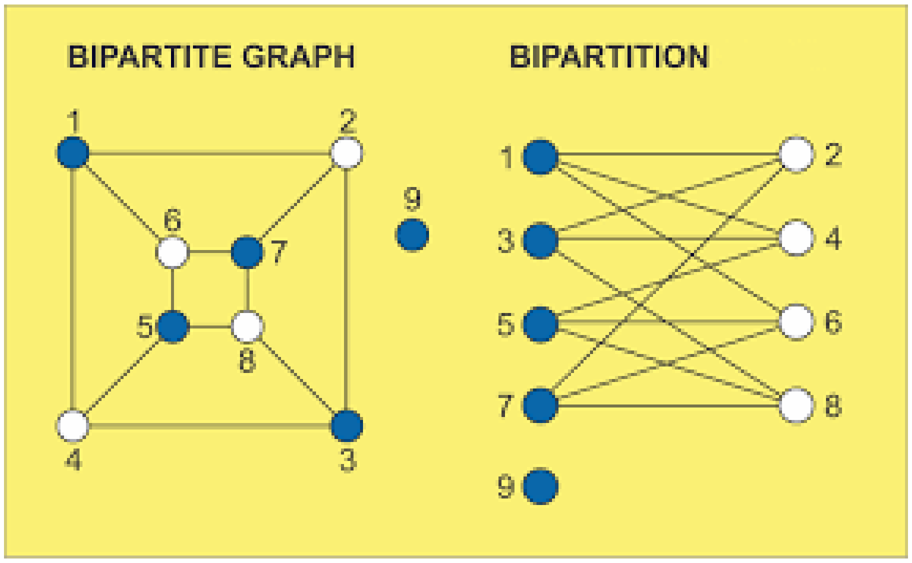
There are many types of graphs categorized based on number of nodes, number of edges, degree of vertices, type of connection/edge(loop), direction of edges, labelled edges etc.

We in particular are interested in Bipartite Graphs, which holds our solution to Task Allocation problem easily.

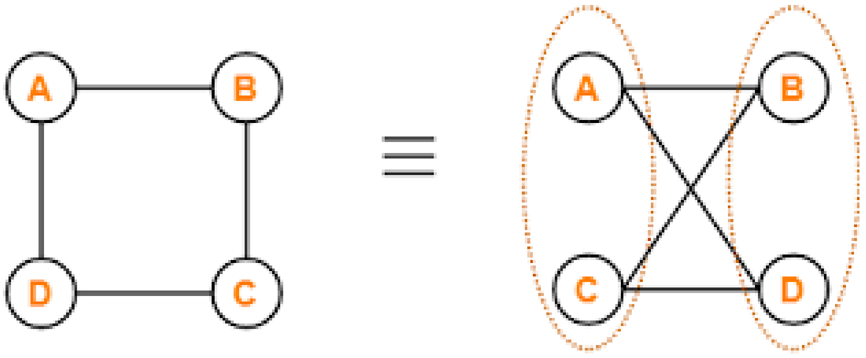
BIPARTITE GRAPH

If the vertex-set of a graph G can be split into two disjoint sets, V_1 and V_2, in such a way that each edge in the graph joins a vertex in V_1 to a vertex in V_2, and there are no edges in G that connect two vertices in V_1 or two vertices in V_2, then the graph G is called a ***BIPARTITE GRAPH.***

Example 1.



Example 2.

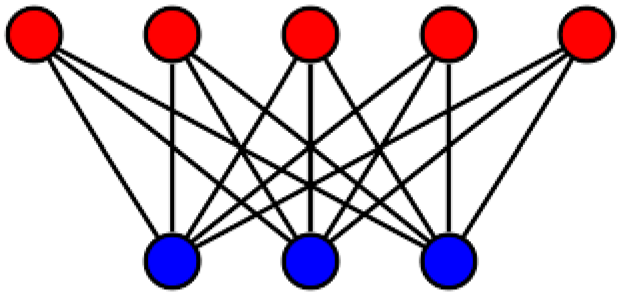


Example of Bipartite Graph

COMPLETE BIPARTITE GRAPH:

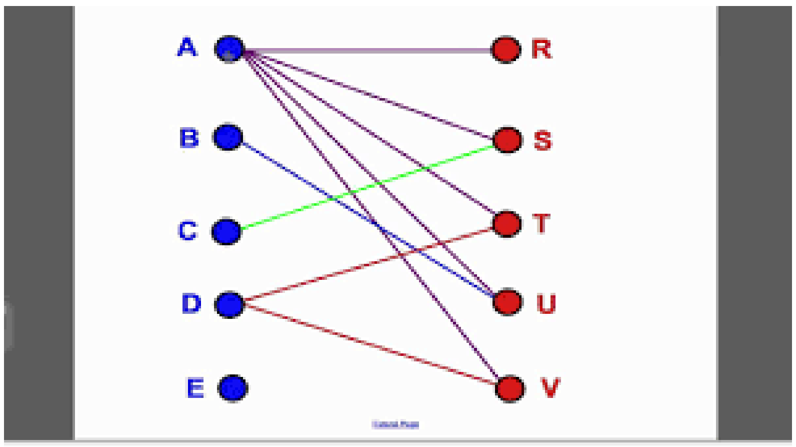
A complete bipartite graph is a bipartite graph in which each vertex in the first set is joined to every single vertex in the second set. The complete bipartite graph is denoted by Kx,y where the graph G contains x vertices in the first set and y vertices in the second set.

Example 1.



A complete bipartite graph can have an isolated point as condition is in splitting, but condition like no edges between two sets are possible.

Example 2.



Input/Output and Matching

- Inputs are integral arrays, one specifying the value/property of task, one for machine capacity and one specifying compatibility property from set of tasks to the set of machines compatible to process the task. Output is the number specifying maximum number of tasks which can be processed after allocation in single go at that instance of time.

- A **matching** M is a set of pairwise non-adjacent edges of a graph (in other words, no more than one edge from the set should be incident to any vertex of the graph M). The **cardinality** of a matching is the number of edges in it. The maximum (or largest) matching is a matching whose cardinality is maximum among all possible matchings in a given graph. All those vertices that have an adjacent edge from the matching (i.e., which have degree exactly one in the subgraph formed by M) are called **saturated** by this matching.

- An **alternating path** is a path in which the edges alternately belong / do not belong to the matching.

- An **augmenting path** is an alternating path whose initial and final vertices are unsaturated, i.e., they do not belong in the matching.

- A maximal matching is a matching to which no more edges can be added without increasing the degree of one of the nodes to two; it is a local maximum. A maximum matching is a matching with the largest possible number of edges; it is globally optimal (This is the solution we want).
- A matching M is maximum if there is no augmenting path relative to the matching M.

Explanation of Solution

We initially take an empty matching. Then, the algorithm finds an augmenting path, and we update the matching by alternating it along this path and repeat the process of finding the augmenting path. As soon as it is not possible to find such a path, process ends by matching final vertex if found else/and the current/final matching found is the maximum one.

Implemented algorithm searches for any of these paths using depth-first traversal. The algorithm looks through all the vertices of the graph in turn, starting each traversal from it, trying to find an augmenting path starting at this vertex.

The algorithm looks at all the vertices T of the first part of the graph *tasks*: $T = 1 \ldots n_1$. If the current task vertex T is already saturated with the current matching (i.e., some edge adjacent to it has already been selected for the machine we are approaching), then skip already marked task vertex. Otherwise, the algorithm tries to saturate this task vertex, for which it starts a search for an augmenting path starting from this task vertex.

The search for an augmenting path is carried out using depth-first traversal. Initially, the depth-first traversal is at the current unsaturated task vertex t of the first part/set task, which looks through all edges from this task vertex. Let the current edge be an edge (t,m), connecting task t and machine m. If the machine vertex m is not yet saturated with matching, then we have succeeded in finding an augmenting path: it consists of a single edge (t,m); in this case, we simply include this edge in the matching and add the number of tasks completed in and stop searching for the augmenting path from the vertex t. Otherwise, if machine vertex m is already saturated with some edge (m,p), then it will go along this edge: thus, we will try to find an augmenting path passing through the edges $(t,m),(m,p),\ldots$. It will saturate m with p and then start augmenting path with t again until found. To do this, it will simply go to the vertex t in our traversal - now we try to find an augmenting path from this vertex again.

So, this traversal, launched from the vertex t, will either find an augmenting path, and thereby saturate the machine vertex m, or it will not find such an augmenting path (and, therefore, this task vertex t cannot be saturated).

After all the vertices $t=1 \ldots n_1$ have been scanned, the current matching will be maximum.

ALGORITHM AND ANALYSIS

TIME COMPLEXITY ANALYSIS:

We are using two functions:

1. bipartiteMatch : this function returns true if matching is found or saturation can be done to current task vertex sent by maxMatch function and can call itself if the machine vertex is already visited and assigned at the same time. In the latter, it will try to find a new augmenting path starting from adjacent vertex. If the matching is not found in the augmenting function than condition becomes false and also if v is visited than we return false.
2. maxMatch: this function traverse through all the vertices in the graph and also keeps track of counter which records total tasks allocated. While traversing all the vertices we call bipartiteMatch function for current unsaturated vertex and try to allocate matching in bipartiteMatch function.

PART-1:

ALGORITHM (bipartiteMatch)	COST
Begin	
for all vertex v, which are adjacent with u, do	
if v is not visited, then	E * O(1)
mark v as visited	E * O(1)
If v is not assigned or bipartiteMatch(assign[v], visited, assign) is true, then	E * O(1)*T`(n-1)
assign[v] = u	E * O(1)
return true	E * O(1)
done	
return false	E * O(1)
End	
TOTAL COST = E*O(1)*5+ E*O(1)*T(n-1) = a*T`(n-1) + b	

Here E is number of edges incident on vertex u which will be the number of vertices adjacent to u.

T`(E) = E*O(1)*4+ E*O(1)*T(n-1)

= a*T`(n-1) + b

T`(n) = θ(n)

If we write accurately, in dfs traversal as explained in video

T` = O(V+E), V = number of vertex, and E = number of edges.

V = N+M, and E = N*M

Where N = number of machines or processor

and   M = number of tasks,

So T` = O(N+M + N*M)

= O(N*M), as asymptotically N*M >>> N+M

So overall we get T` = O(N*M)

PART-2:

ALGORITHM (bipartiteMatch)	COST
Begin initially no vertex is assigned count = 0 for all applicant u in M, do make all node as unvisited if bipartiteMatch(u, visited, assign) is true increase count by 1 done End	1. O(1) 2. M*O(1)+1 3. M*O(1) 4. M*T` 5. M*O(1)
TOTAL COST	T = **θ(N*M*M)**

M = number of vertices in tasks.

So T   = O(1) + M* O(1)*3 + 1 + M*T`

= M*O(N*M) + O(M)

= O(N*M*M), as asymptotically N*M*M >>>> M

So over all Time complexity = θ(N*M*M)

SPACE COMPLEXITY ANALYSIS:

Since we are not constructing any graph and using array visited, array assigned, both of which contains members as many as number of vertices, so overall space complexity is

S(V) = O(N), which are number of tasks or

S(V) = O(V), asymptotically both will be same.

REFERENCES:

1. **Wikipedia**
2. **Educba.com**
3. **Educative.io**
4. **Www3.nd.edu**
5. **Www2.cs.duke.edu**
6. **www.toptal.com**