GOVERNMENT COLLEGE UNIVERSITY LAHORE

COMPUTER SCIENCE DEPARTMENT

EXAM SHEDULING PROBLEM USING MAXIMUM FLOW ALGORITHM (DINIC's MAX FLOW ALGO)

Prepared By:

MUHAMMAD JAWAD AMIN

0175-BSCS-2020

Section (A)

Title: Using Dinitz's Maximum Flow Algorithm To Determine That How Many Exams Can Conduct In An Educational Institute.

ABSTRACT

Every educational institute on the earth take the exams of the students which are studying in this institute to check their abilities and to promote them to the next class. There is a particular time of the year in which every educational institute conduct exams for all the classes which are studying in the institute at the same time. At that time, they must schedule the exam conduction process on the daily bases depending on the no of classes which are going to take part in the exam. Some time there is a lot of classes but limited number of rooms, timeslots, proctors, and many other things related to exam conduction. These limitations cause many problems in the exam scheduling in the institute. They are unable to conduct the exams in an efficient way in which every source is used efficiently.

This scheduling problem in the educational institutes can be represented as a **Maximum Flow Problem** or as a Network Graph. So, this problem can be solved using the Maximum Flow Algorithms. There are many Maximum Flow Algorithms, but I will use the **Dinic's Maximum Flow Algorithm** to solve this problem. The reason behind that the network graph representation of this problem is look like a bipartite graph and **Dinic's Maximum Flow Algorithm** runs very efficiently on the bipartite graphs with the time complexity of $\mathbf{O(EV^{1/2})}$.

Here I am modifying the Dinic's Max Flow Algo to solve this problem by just considering only four limitations which no of classes which take the exam, no of rooms available, no of time slots available for the exam conduction, and on the no of proctors which will oversee the exams. On the other hand there are also some other limitation such as the relation between the seating capacity of rooms and the no of students in each class and how many exams a proctor can oversee.

PROBLEM STATEMENT:

This algorithm is designed or modified to solve the problem of Exam Scheduling in the educational institutions depending on the number of classes which must give the exam, no of rooms in which exam will be conducted, no of time slots available for the exam conduction and on the number of proctors which will oversee the exam. Basically, this algorithm will tell us how many exams can be conducted in the institute according to the given requirement. Such type of scheduling problems is very common in the real world which can also be solved by using this algo.

This algorithm will solve this problem by providing a comprehensive solution working at a $O(EV^2)$ or $O(EV^{1/2})$ time complexity, being a practical solution to solve these problems.

FOR EXAMPLE:

The Government College University wants to conduct the final exams of some semester and they have limited number of rooms, classes of students, time slots and proctors but they don't know how to schedule the exam efficiently. So, it is a max flow or scheduling problem which can also be solved by using the **Dinitz's Algorithm for Maximum Flow**.

BACKGROUND:

The maximum flow problem was first observed in 1930 by A.N Tolstoy to solve the Transportation Problem for a Railway System for Soviet Union to find that how many trains can arrive at a particular railway station and how many trains can leave a railway station at a time. Basically, in this problem the no of trains is calculated which can leave the source and reach to their destination. This maximum flow problem was first formulated by the T.E Harris and F. S. Ross as a simplified model of Soviet railway traffic flow. In order to solve this problem Lester R. Ford and Delbert R. Fulkerson created the first well known algorithm named as **Ford-Fulkerson Algorithm** time complexity of **O(E*f*)** where ***f*** is the max flow**.** This algorithm at that time was very helpful for solving many max flow problems and it is implemented in different ways by making changes in by many different algorithm writers therefore, now it is considered as a method to solve the max flow problem not as algorithm. The commonly known implementation of this algorithm is **Edmonds Karp's Algorithm** for Max Flow problems with time complexity of **$O(VE^2)$.**

Yefim Dinitz invented the **Dinitz's Max Flow Algorithm** in 1969 in response to a pre-class exercise in **Adelson-Velskys** algorithms class which was published in 1970 in the journal **Doklady Akademii Nauk SSSR**. At that time, he was not aware of the basic facts regarding the **Ford–Fulkerson algorithm**. It was a strongly polynomial algorithm which find the maximum flow through a Network Graph with a time complexity of **$O(EV^2)$** and if the Network Graph is a bipartite graph with edges of capacity one only then its time is reduced to **$O(EV^{1/2})$.** Dinitz's algorithm and the Edmonds–Karp algorithm both have independently showed that in the Ford–Fulkerson algorithm, if each augmenting path is the shortest one, then the length of the augmenting paths is non-decreasing and the algorithm always terminates. This is because there was a problem in the Ford-Fulkerson Algorithm which is that when it terminates then it is not the guarantee that it has found the max flow through the Network Graph

which has to be check using **the Min Cut Max Flow Theorem** or by finding the **Min Cut** in the Network Graph.

SOME TERMINOLOGIES:

There are some terminologies which one should have to know to understand the Maximum Flow problem. Because these terminologies are the basis of the Network Graph or Max Flow problem. Some of the important definitions are given below:

(Definition 1: A **flow network** is a directed graph G = (V, E) with two distinguished vertices: a source s and a sink t. Each edge (u, v) ∈ E has a nonnegative capacity c (u, v). If (u, v) ∉ E, then c (u, v) = 0.**)**

(Definition 2: A **flow** on G is a function *f*: V × V → R satisfying the following:

- **Capacity constraint**: For all u, v ∈ V, f (u, v) ≤ c (u, v).
- **Flow conservation**: For all u ∈ V – {s, t}, ∑ f (u, v) = 0. v ∈ V
- **Skew symmetry**: For all u, v ∈ V, f (u, v) = –f (v, u).**)**

(Definition 3: Let *f* be a flow on G = (V, E). The **residual network** Gf (V, E_f) is the graph with strictly positive residual capacities c_f (u, v) = c (u, v) – f (u, v) > 0. Edges in Ef admit more flow if (v, u) ∉ E, c (v, u) = 0, but f (v, u) = – f (u, v). $|E_f| \leq 2 |E|$.**)**

(Definition 4: A graph **G_f** that indicate the additional possible path for flow or help to find the augmenting paths for flow in a network is known as **Residual Graph G_f.)**

(Definition 5: The capacity of edges in the residual graph is known as **Residual Capacity c_f**. It can be calculated by subtracting the flow through the edge from its original capacity which is as follows: **c_f** = c (u, v) – f (u, v).**)**

(Definition 6: Any path from s to t in G_f is an **augmenting path** in G with respect to *f*. The flow value can be increased along an augmenting path p by c f (p) = min {c f (u, v)}.**)**

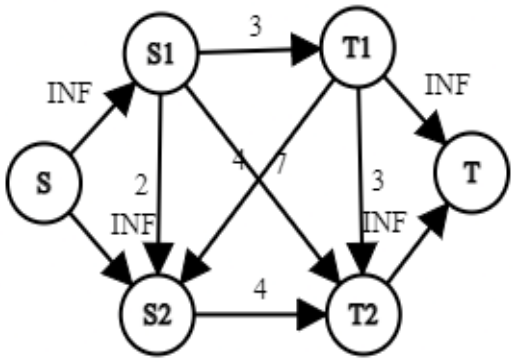
(Definition 7: Minimum capacity edge in augmenting path, which decides the maximum flow from source **S** to the sink **T** through that Augmenting path is called the **Bottleneck Edge**.**)**

These were the definitions which are necessary for the of study of maximum flow problem. There are also some **assumptions** which should also have to be applied on the problem which are as follows:

- If edge (u, v) ∈ E exists, then (v, u) ∉ E.
- No self-loop edges (u, u) exist and there is no cycle between two consecutive edges.

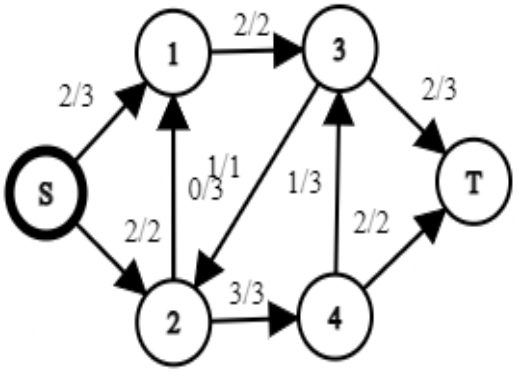
There are also two more important concepts which should be known and used in this problem which are Multi Source and Multi Sink concepts. The **Multi Source** means that your network graph has more than one sources (S1, S2, S3, ……., Sn) to do the flow through the network and similarly **Multi Sink** means your network graph has multiple sinks (T1, T2, T3 ……., Tn) to consume the flow which is again problem. Due to this problem, you don't know from which source to start the flow, and to which sink consume the flow. To solve this problem, we convert the multiple sources into a single source by

joining all the sources to a new vertex called **Super Source** (**S)** with edges having capacity equal to infinity similarly we convert the multiple sinks into a single sink by joining all the sinks to a new vertex called **Super Sink** (**T)** with edges having capacity infinity. For example:



A NETWORK GRAPH:

Example of a network graphs **G** with some flow ***f*** as follows:



This is a network graph in which circles represent the vertices and the arrow between two vertices represent an edge with some capacity and flow. Each edge has some capacity and flow, the value written to the left of slash is the **flow *f*** and the value written right to the slash is **capacity *c***. Where vertices with name **S** and **T** are **Source** and **Sink** respectively. It can be seen from the above graph that the **S** don't have incoming edges and **T** don't have outgoing edges and one more thing the graph has only one source **S** and one sink **T**.

DINIC's ALGORITHM:

After having a detail view of network flow here comes the Dinic's Max Flow Algorithm to find the maximum flow from source S to the sink T. Dinitz's Algorithm improves the Edmonds-Karp Algorithm by discovering a blocking flow, which is in some sense a maximal set of shortest augmenting paths that can be used simultaneously to update the current flow without violating capacity constraints.

Algorithm 01:

Dinic'sAlgo (G, S, T)

1: $f \leftarrow 0$; $G_f \leftarrow G$

2: WHILE Gf contains an $s - t$ path P **DO**

3: Let h be a blocking flow in Gf

4: $f \leftarrow f + h$

5: Update G_f

6: END WHILE

7: RETURN *f*

This is the Dinic's Algorithm for maximum flow which will calculate the maximum flow through a network. But we have used a term in the algorithm which is Blocking Flow. If **G** is a flow network, f is a flow, and h is a flow in the residual graph $\mathbf{G_f}$, then **h** is called a **blocking flow** if every shortest augmenting path in $\mathbf{G_f}$ contains at least one edge that is saturated by **h**, and every edge **e** with $\mathbf{h_e > 0}$ belongs to a shortest augmenting path. Basically, here calculation of the blocking flow means finding the edge in the level graph having the minimum capacity. The blocking flow in Dinic's

Algorithm is calculated using the separate algorithm which is as follows:

Algorithm 02:

BlockingFlow (G_f, S, T)

1: $h \leftarrow 0$

2: Let G' be the level graph composed of advancing edges in G_f.

3: Initialize c'(e) = r(e) (residual capacity of e) for each edge e in G'.

4: Initialize stack with (S).

5: REPEAT

6: Let u be the top vertex on the stack.

7: **IF** u = t **THEN**

8: Let P be the path defined by the current stack.

9: Let $\delta(P) = \min \{c'(e) \mid e \in P\}$.

10: $h \leftarrow h + \delta(P) 1_P$.

11: $c'(e) \leftarrow c'(e) - \delta(P)$ for all $e \in P$.

12: Delete edges with c'(e) = 0 from G'.

13: Let (u, v) be the newly deleted edge that occurs earliest in P.

14: Truncate the stack by popping all vertices above u.

15: **ELSE IF** G' contains an edge (u, v) **THEN**

16: Push v onto the stack.

17: **ELSE**

18: Delete u and all its incoming edges from G'.

19: Pop u from the stack.

20: **END IF**

21: UNTIL stack is empty

22: RETURN h

This is the algorithm which finds the blocking flow in the level graph which is built using the **BFS O(V+E)** in the Dinic's Algo. This algo plays an important role in the efficiency of the Dinic's Algo because it uses the **DFS O(V+E)** to find the blocking flow through the level graph which is a very efficient method because the level graph only contains the paths equals to the edges coming out of the Source **S**.

Once the blocking flow or bottleneck edge is found it returns that blocking flow to the Dinic,s Algorithm which then updates the flow which is being calculated and on the ither hand it augments the blocking flow through the residual graph $\mathbf{G_f}$. Here augmenting the flow means updating the residual graph or updating the flow values along the augmented path in the residual graph which is find using as level graph.

APPLICATION:

There are many applications of the maximum flow problem in which we can use this algorithm to solve these problems. Some of these problems are as follows:

- Airline Scheduling
- Exam Scheduling
- Max water flow through pipes
- Max current flow
- Max packet flow through a network
- Max traffic flow through a map
- Bipartite Matching
- Tuple Selection
- Owl and Mice Problem

PROOF OF CORRECTNESS:

Loop Invariant:

If an augmenting path exists in the residual network $\mathbf{G_f}$ find the blocking flow and augment the flow ***f*** through the residual network $\mathbf{G_f}$. Repeat this until there is no augmenting path left in the residual graph G_f.

Step 01:

Initialization Stage:

First, we should know that Dinic's Algo uses the **BFS** to find the augmenting path in the residual network or to generate a level graph and it uses the **DFS** to find the blocking flow or minimum capacity edge in the augmenting path or level graph. So, before the algorithm starts, we create a network graph **G** with some capacities **c** as positive integers on its edges and flow ***f*** on its edges initially to zero and **Max Flow Variable** to zero. As the flow along the edges is initially then during the first iteration of the while loop j=1, **BFS** will find an augmenting path and return true, after that **DFS** will find the blocking flow in the level graph and update the flow across the residual network and add the blocking flow to the **Max Flow Variable**. So, the loop invariant is true for 1^{st} iteration or initialization stage.

Step 02:

Maintenance Stage:

Assuming that the loop invariant is true for the j^{th} iteration now we will prove the loop invariant for j=j+1 iteration. As we know that **BFS** has found the augmenting path in the j=1 iteration and change the graph **G** to $\mathbf{G_f}$ by augmenting the flow across the network graph. Now the graph contains the residual edges which have some positive capacity. During j=j+1 iteration BFS will again find the augmenting path in the $\mathbf{G_f}$ and return true. After that **DFS** will again find the blocking flow and algo will again augment the flow across the network and add the blocking flow to the **Max Flow Variable.** Hence the loop invariant is true for the j=j+1 iteration or maintenance stage.

Step 03:

Termination Stage:

Assuming that the loop invariant is true for the E=E-1 iteration we will prove that the loop invariant is true for E^{th} iteration. As we know that **BFS** is finding the level graph or augmenting path in the $\mathbf{G_f}$. We assumed that **BFS** has found the augmenting path for the Eth iteration and **DFS** has found the blocking flow or edge with minimum capacity in the augmenting path and this blocking flow has been augmented across the $\mathbf{G_f}$ and **Max Flow Variable** is Updated. But after the end of Eth iteration for which loop invariant is true the E+1 iteration will start but this time **BFS** will not find any augmenting path in the $\mathbf{G_f}$ therefore it returns false to the while loop due to which while loop will terminate, and algorithm will return the max flow which is now stored in the **Max Flow Variable**. In this way loop invariant is true for the termination stage or E^{th} iteration.

Step 04:

Conclusion:

Algorithm runs if there exists an augmenting path in the $\mathbf{G_f}$ or **BFS** can find the level graph in the $\mathbf{G_f}$. The augmenting paths in the $\mathbf{G_f}$ depends on the no of edges and the capacities of the edges.

TIME COMPLEXITY:

Here the time complexity calculation of Dinic's Algo is done on a coded algo not on the pseudo code which is as folows:

Algorithm:

Dinic'sAlgo (G, S, T)

1: Max Flow = 0

2: G_f = G

3: G_f = UpdateGraph(G_f)

4: G_L = -1

5: bool AugPath = false

6: WHILE (1) [(AugPath = BFS ()) == True**] DO**

7: Temp Flow = DFS (G_L)

8: **WHILE (2) [**Temp Flow != 0**] DO**

9: Max Flow += Temp Flow

10: Temp Flow = Temp Flow + DFS (Level Graph)

11: **END WHILE (2)**

12: END WHILE (1)

13: RETURN Max Flow

Calculation Of Time Complexity:

Line No 01: Steps → O (1)

Line No 02: Steps → O (VE)

Line No 03: Steps → O(CR)

//Where C=No of Classes & R=No of Rooms

Line No 04: Steps → O(V)

Line No 05: Steps → O (1)

Line No 06: Steps → $O(VE+E^2)$

Line No 07: Steps → O(V)

Line No 08: Steps → $O(EV^2)$

Line No 09: Steps → O(EV)

Line No 10: Steps → $O(EV^2)$

Line No 11: Steps→ O (1)

Line No 12: Steps→ O (1)

Line No 13: Steps→ O (1)

Adding The Dominant Terms:

T(N) = $O(VE) + O(CR) + O(V) + O(VE+E^2) + O(EV^2) + O(EV) + O(EV^2) + O(V)$

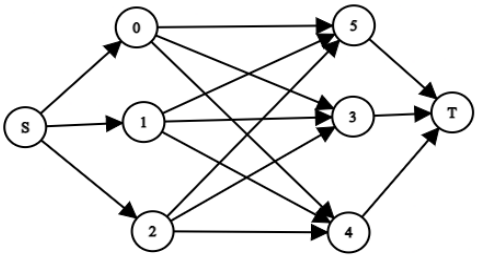
As from the above equation the biggest term or dominant term among these are **$O(EV^2)$.** Hence the time complexity of the Dinic's Algorithm is:

T(N) = $O(EV^2)$

TIME COMPLEXITY CASES:

Best Case:

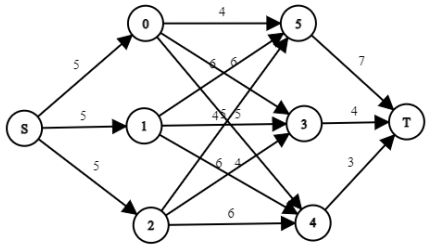
The best case for the Dinic's Algo is when it is used with a bipartite graph having capacity on each edge equal to one (1) which is equal to **$O(EV^{1/2})$.** A bipartite graph with implicit capacity equal to one is given below:



Average Case:

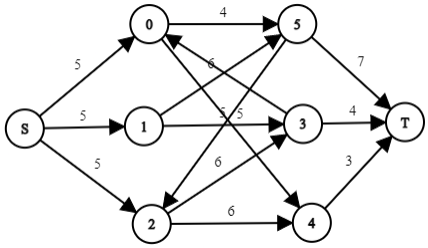
The average case for the Dinic's Algorithm is when the graph on

which it is operated is a bipartite graph but don't have edges with capacity equal to one (1) which is equal to **$O(EV^2)$.** A bipartite graph with capacities greater than one is given below:



Worst Case:

The worst case of Dinic's Algorithm is when the graph on which it is operated is not a bipartite graph and don't have capacities equal to one (1) which is equal to O**(EV^2).** A graph which is not a bipartite and don't have capacities equal to one is given below:



Space Complexity:

The space complexity of the Dinic's Algorithm is **O(E+V)** because it must only store the vertices along with their connected edges. An adjacency list is given below:

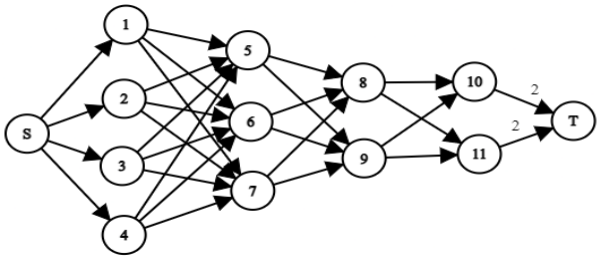
S (Source)	S → 1, S → 2, S → 0
0	0 → 1, 0 → 2, 0 → T
1	1 → 0, 1→2, 1→ T
2	2→ 0, 2→1, 2→ T
T (Sink)	It does not contain any outgoing edge.

DRY RUN OF THE ALGORITHM:

Now we will dry run the Dinic's Algorithm on the example which is explained in the problem statement. The example is that GCU wants to conduct the final exam of some university classes. The data for the problem is as follows:

No of Classes	No of Rooms	No of Times Slots	No of Proctors	Proctors can Oversee Exams
4	3	2	2	2

According to above data the graph which will be generated is as follows:

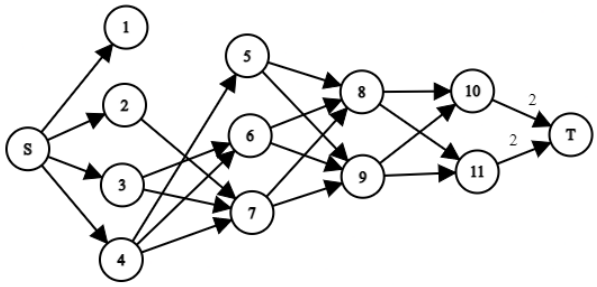


In the above graph those edges which don't have capacities their capacity is one (1) only. The vertices which are one length far from the source are classes, the vertices which are two lengths far from the source are room, the vertices which are three length far from the source are the timeslots and the vertices which are four length far from the source are proctors which are connected to the sink with the capacity of two (2) which means that they can oversee the maximum two exams. Now the data about the no of class student and the seating capacities of the rooms is as follows:

Classes	1	2	3	4
No of Students	50	45	40	35

Rooms	5	6	7
Seating Capacity	35	40	45

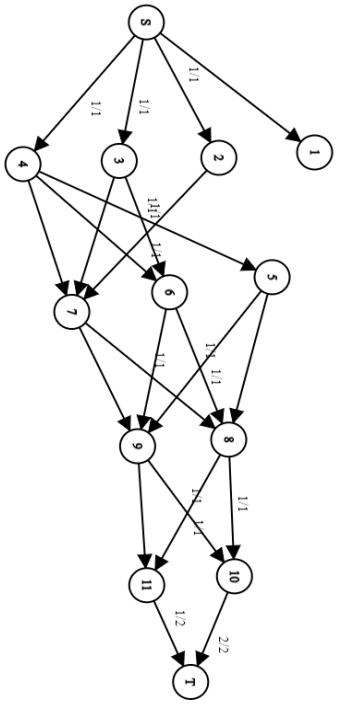
When the algo starts running it will make a bipartite graph like structure of them as shown above in a figure. But algo will remove some edges from the graph depending on the no of students in the classes and the no of seats in the rooms because a class can give exam only in that room whose seating capacity matches with the no students in the class. The modified graph is as follows:



The edges which are removed from the graph depending on the capacity and no of students' relation of classes and rooms are as follows:

Classes	Removed Edges
1	1 → 5
	1 → 6
	1 → 7
2	2→5
	2→6
3	3→5
4	No edge is removed.

After the removal of edges the Dinic's Max Flow Algorithm is run on the residual network which gives the maximum flow as three (3). This max flow value tells us that how many exams we can scheduled under these circumstances which are equal to three (3). The final max flow graph which is generated after the max flow algorithm runs on it is as follows:



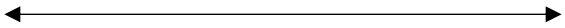
The table for this graph in which all the data is represented such as flow of edges, capacity of edges and whether the edge is residual or not is as follows:

Edges	Capacity	Flow	Residual
S→2	1	1	No
S→3	1	1	No
S→4	1	1	No
2→7	1	1	No
3→6	1	1	No
4→5	1	1	No
5→8	1	1	No
6→8	1	1	No
7→8	1	1	No
8→10	1	1	No
8→11	1	1	No
9→10	1	1	No
10→T	2	2	No
11→T	1	2	No

From the above table we can get the all sort of information that which class can give the final exam, in which room the exam will be conducted, which time slot will be selected and which proctor oversee the exam on the other hands it also tells that how many exams a proctor can oversee.

CONCLUSION:

By using this algorithm on this problem, we have concluded that we can use this algorithm for any kind of scheduling problem which we can represent in the form of network graph. On the other hand, some other type of problems can also be solved using this algorithm some of its applications are explained above. One benefit of this algorithm is that it is a **strongly polynomial** algorithm.



REFERENCES:

- Introduction To Algorithm: By Thomas H. Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein.
- Some of information from: http://ocw.mit.edu/terms
- Graph used above created by myself using the resource: https://csacademy.com/app/graph_editor/
- Some historical information from this site: https://blogs.cornell.edu/info4220/2015/03/10/the-origin-of-the-study-of-network-flow/