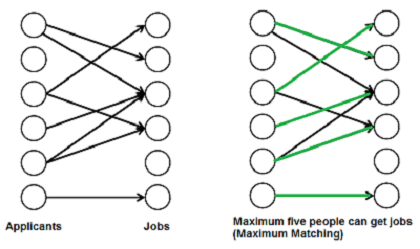
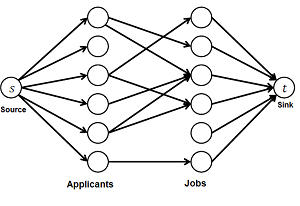
Maximum Bipartite Matching

* Difficulty Level : [Hard](https://www.geeksforgeeks.org/hard/)
* Last Updated : 16 Jun, 2021

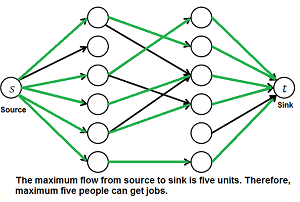
A matching in a [Bipartite Graph](https://www.geeksforgeeks.org/bipartite-graph) is a set of the edges chosen in such a way that no two edges share an endpoint. A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matchings for a given Bipartite Graph.   
**Why do we care?**   
There are many real world problems that can be formed as Bipartite Matching. For example, consider the following problem:   
*There are M job applicants and N jobs. Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. Find an assignment of jobs to applicants in such that as many applicants as possible get jobs.*



We strongly recommend to read the following post first.  
[Ford-Fulkerson Algorithm for Maximum Flow Problem](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/)  
**Maximum Bipartite Matching and Max Flow Problem**   
**M**aximum **B**ipartite **M**atching (**MBP**) problem can be solved by converting it into a flow network (See [this](http://www.youtube.com/watch?v=NlQqmEXuiC8)video to know how did we arrive this conclusion). Following are the steps.



***1) Build a Flow Network***   
There must be a source and sink in a flow network. So we add a source and add edges from source to all applicants. Similarly, add edges from all jobs to sink. The capacity of every edge is marked as 1 unit.



***2) Find the maximum flow.***   
We use [Ford-Fulkerson algorithm](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/) to find the maximum flow in the flow network built in step 1. The maximum flow is actually the MBP we are looking for.

**How to implement the above approach?**   
Let us first define input and output forms. Input is in the form of [Edmonds matrix](http://en.wikipedia.org/wiki/Edmonds_matrix) which is a 2D array ‘bpGraph[M][N]’ with M rows (for M job applicants) and N columns (for N jobs). The value bpGraph[i][j] is 1 if i’th applicant is interested in j’th job, otherwise 0.   
Output is number maximum number of people that can get jobs.   
A simple way to implement this is to create a matrix that represents [adjacency matrix representation](https://www.geeksforgeeks.org/graph-and-its-representations/)of a directed graph with M+N+2 vertices. Call the [fordFulkerson()](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/) for the matrix. This implementation requires O((M+N)\*(M+N)) extra space.   
Extra space can be reduced and code can be simplified using the fact that the graph is bipartite and capacity of every edge is either 0 or 1. The idea is to use DFS traversal to find a job for an applicant (similar to augmenting path in Ford-Fulkerson). We call bpm() for every applicant, bpm() is the DFS based function that tries all possibilities to assign a job to the applicant.  
In bpm(), we one by one try all jobs that an applicant ‘u’ is interested in until we find a job, or all jobs are tried without luck. For every job we try, we do following.   
If a job is not assigned to anybody, we simply assign it to the applicant and return true. If a job is assigned to somebody else say x, then we recursively check whether x can be assigned some other job. To make sure that x doesn’t get the same job again, we mark the job ‘v’ as seen before we make recursive call for x. If x can get other job, we change the applicant for job ‘v’ and return true. We use an array maxR[0..N-1] that stores the applicants assigned to different jobs.  
If bmp() returns true, then it means that there is an augmenting path in flow network and 1 unit of flow is added to the result in maxBPM(). 

[Recommended: Please solve it on “***PRACTICE***” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/maximum-bipartite-matching/1)

* C++
* Java
* Python
* C#
* PHP
* Javascript

|  |
| --- |
| # Python program to find  # maximal Bipartite matching.    class GFG:      def \_\_init\_\_(self,graph):            # residual graph          self.graph = graph          self.ppl = len(graph)          self.jobs = len(graph[0])        # A DFS based recursive function      # that returns true if a matching      # for vertex u is possible      def bpm(self, u, matchR, seen):            # Try every job one by one          for v in range(self.jobs):                # If applicant u is interested              # in job v and v is not seen              if self.graph[u][v] and seen[v] == False:                    # Mark v as visited                  seen[v] = True                    '''If job 'v' is not assigned to                     an applicant OR previously assigned                     applicant for job v (which is matchR[v])                     has an alternate job available.                     Since v is marked as visited in the                     above line, matchR[v]  in the following                     recursive call will not get job 'v' again'''                  if matchR[v] == -1 or self.bpm(matchR[v],                                                 matchR, seen):                      matchR[v] = u                      return True          return False        # Returns maximum number of matching      def maxBPM(self):          '''An array to keep track of the             applicants assigned to jobs.             The value of matchR[i] is the             applicant number assigned to job i,             the value -1 indicates nobody is assigned.'''          matchR = [-1] \* self.jobs            # Count of jobs assigned to applicants          result = 0          for i in range(self.ppl):                # Mark all jobs as not seen for next applicant.              seen = [False] \* self.jobs                # Find if the applicant 'u' can get a job              if self.bpm(i, matchR, seen):                  result += 1          return result      bpGraph =[[0, 1, 1, 0, 0, 0],            [1, 0, 0, 1, 0, 0],            [0, 0, 1, 0, 0, 0],            [0, 0, 1, 1, 0, 0],            [0, 0, 0, 0, 0, 0],            [0, 0, 0, 0, 0, 1]]    g = GFG(bpGraph)    print ("Maximum number of applicants that can get job is %d " % g.maxBPM())    # This code is contributed by Neelam Yadav |

**Output:**

Maximum number of applicants that can get job is 5

Bellman–Ford Algorithm | DP-23

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 05 Jun, 2021

Given a graph and a source vertex *src*in graph, find shortest paths from *src*to all vertices in the given graph. The graph may contain negative weight edges.   
We have discussed [Dijkstra’s algorithm](https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/) for this problem. Dijkstra’s algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). *Dijkstra doesn’t work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.* 

[Recommended: Please solve it on “***PRACTICE*** ” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/negative-weight-cycle/0)

**Algorithm**   
Following are the detailed steps.  
*Input:* Graph and a source vertex *src*   
*Output:* Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.  
**1)** This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.  
**2)** This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.   
…..**a)** Do following for each edge u-v   
………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]   
………………….dist[v] = dist[u] + weight of edge uv  
**3)** This step reports if there is a negative weight cycle in graph. Do following for each edge u-v   
……If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”   
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn’t contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle  
***How does this work?*** Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges (Proof is simple, you can refer [this](http://courses.csail.mit.edu/6.006/spring11/lectures/lec15.pdf) or [MIT Video Lecture](http://www.youtube.com/watch?v=Ttezuzs39nk))  
**Example**   
Let us understand the algorithm with following example graph. The images are taken from [this](http://www.cs.arizona.edu/classes/cs445/spring07/ShortestPath2.prn.pdf)source.  
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to the source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times.*

Example Graph

Let all edges are processed in the following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get the following distances when all edges are processed the first time. The first row shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed. 

https://media.geeksforgeeks.org/wp-content/uploads/bellmanford2.png

The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time (The last row shows final values). 

https://media.geeksforgeeks.org/wp-content/uploads/bellmanford3.png

The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don’t update the distances.  
**Implementation:** 

* C++
* Java
* Python3
* C#

|  |
| --- |
| # Python3 program for Bellman-Ford's single source  # shortest path algorithm.    # Class to represent a graph  class Graph:        def \_\_init\_\_(self, vertices):          self.V = vertices # No. of vertices          self.graph = []        # function to add an edge to graph      def addEdge(self, u, v, w):          self.graph.append([u, v, w])        # utility function used to print the solution      def printArr(self, dist):          print("Vertex Distance from Source")          for i in range(self.V):              print("{0}\t\t{1}".format(i, dist[i]))        # The main function that finds shortest distances from src to      # all other vertices using Bellman-Ford algorithm. The function      # also detects negative weight cycle      def BellmanFord(self, src):            # Step 1: Initialize distances from src to all other vertices          # as INFINITE          dist = [float("Inf")] \* self.V          dist[src] = 0              # Step 2: Relax all edges |V| - 1 times. A simple shortest          # path from src to any other vertex can have at-most |V| - 1          # edges          for \_ in range(self.V - 1):              # Update dist value and parent index of the adjacent vertices of              # the picked vertex. Consider only those vertices which are still in              # queue              for u, v, w in self.graph:                  if dist[u] != float("Inf") and dist[u] + w < dist[v]:                          dist[v] = dist[u] + w            # Step 3: check for negative-weight cycles. The above step          # guarantees shortest distances if graph doesn't contain          # negative weight cycle. If we get a shorter path, then there          # is a cycle.            for u, v, w in self.graph:                  if dist[u] != float("Inf") and dist[u] + w < dist[v]:                          print("Graph contains negative weight cycle")                          return            # print all distance          self.printArr(dist)    g = Graph(5)  g.addEdge(0, 1, -1)  g.addEdge(0, 2, 4)  g.addEdge(1, 2, 3)  g.addEdge(1, 3, 2)  g.addEdge(1, 4, 2)  g.addEdge(3, 2, 5)  g.addEdge(3, 1, 1)  g.addEdge(4, 3, -3)    # Print the solution  g.BellmanFord(0)    # Initially, Contributed by Neelam Yadav  # Later On, Edited by Himanshu Garg |

**Output:**

Vertex Distance from Source

0 0

1 -1

2 2

3 -2

4 1

**Notes**   
**1)**Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.  
**2)** Bellman-Ford works better (better than Dijkstra’s) for distributed systems. Unlike Dijkstra’s where we need to find the minimum value of all vertices, in Bellman-Ford, edges are considered one by one.                                                                    
**3)**Bellman-Ford does not work with undirected graph with negative edges as it will declared as negative cycle.  
**Exercise**   
**1)**The standard Bellman-Ford algorithm reports the shortest path only if there are no negative weight cycles. Modify it so that it reports minimum distances even if there is a negative weight cycle.  
**2)** Can we use Dijkstra’s algorithm for shortest paths for graphs with negative weights – one idea can be, calculate the minimum weight value, add a positive value (equal to absolute value of minimum weight value) to all weights and run the Dijkstra’s algorithm for the modified graph. Will this algorithm work?  
[**Bellman Ford Algorithm (Simple Implementation)**](https://www.geeksforgeeks.org/bellman-ford-algorithm-simple-implementation/)

Bellman Ford Algorithm (Simple Implementation)

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 21 Jun, 2021

We have introduced Bellman Ford and discussed on implementation [here](https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/).  
*Input:* Graph and a source vertex *src*   
*Output:* Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.  
**1)** This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.  
**2)** This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.   
…..**a)** Do following for each edge u-v   
………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]   
………………….dist[v] = dist[u] + weight of edge uv  
**3)** This step reports if there is a negative weight cycle in graph. Do following for each edge u-v   
……If dist[v] > dist[u] + weight of edge uv, then “Graph contains negative weight cycle”   
The idea of step 3 is, step 2 guarantees shortest distances if graph doesn’t contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle  
**Example**   
Let us understand the algorithm with following example graph. The images are taken from [this](http://www.cs.arizona.edu/classes/cs445/spring07/ShortestPath2.prn.pdf)source.  
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times.*

Example Graph

Let all edges are processed in following order: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). We get following distances when all edges are processed first time. The first row in shows initial distances. The second row shows distances when edges (B, E), (D, B), (B, D) and (A, B) are processed. The third row shows distances when (A, C) is processed. The fourth row shows when (D, C), (B, C) and (E, D) are processed. 

https://media.geeksforgeeks.org/wp-content/uploads/bellmanford2.png

The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get following distances when all edges are processed second time (The last row shows final values). 

https://media.geeksforgeeks.org/wp-content/uploads/bellmanford3.png

The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don’t update the distances. 

* C++
* Java
* Python3
* C#
* PHP
* Javascript

|  |
| --- |
| # Python3 program for Bellman-Ford's  # single source shortest path algorithm.  from sys import maxsize    # The main function that finds shortest  # distances from src to all other vertices  # using Bellman-Ford algorithm. The function  # also detects negative weight cycle  # The row graph[i] represents i-th edge with  # three values u, v and w.  def BellmanFord(graph, V, E, src):        # Initialize distance of all vertices as infinite.      dis = [maxsize] \* V        # initialize distance of source as 0      dis[src] = 0        # Relax all edges |V| - 1 times. A simple      # shortest path from src to any other      # vertex can have at-most |V| - 1 edges      for i in range(V - 1):          for j in range(E):              if dis[graph[j][0]] + \                     graph[j][2] < dis[graph[j][1]]:                  dis[graph[j][1]] = dis[graph[j][0]] + \                                         graph[j][2]        # check for negative-weight cycles.      # The above step guarantees shortest      # distances if graph doesn't contain      # negative weight cycle. If we get a      # shorter path, then there is a cycle.      for i in range(E):          x = graph[i][0]          y = graph[i][1]          weight = graph[i][2]          if dis[x] != maxsize and dis[x] + \                          weight < dis[y]:              print("Graph contains negative weight cycle")        print("Vertex Distance from Source")      for i in range(V):          print("%d\t\t%d" % (i, dis[i]))    # Driver Code  if \_\_name\_\_ == "\_\_main\_\_":      V = 5 # Number of vertices in graph      E = 8 # Number of edges in graph        # Every edge has three values (u, v, w) where      # the edge is from vertex u to v. And weight      # of the edge is w.      graph = [[0, 1, -1], [0, 2, 4], [1, 2, 3],               [1, 3, 2], [1, 4, 2], [3, 2, 5],               [3, 1, 1], [4, 3, -3]]      BellmanFord(graph, V, E, 0)    # This code is contributed by  # sanjeev2552 |

**Output:**

Vertex Distance from Source

0 0

1 -1

2 2

3 -2

4 1

**Time Complexity:** O(VE)  
This implementation is suggested by [**PrateekGupta10**](https://auth.geeksforgeeks.org/user/PrateekGupta10)

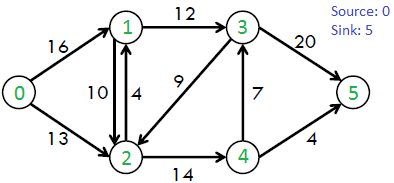
Attention reader! Don’t stop learning now. Get hold of all the important DSA concepts with the [**DSA Self Paced Course**](https://practice.geeksforgeeks.org/courses/dsa-self-paced?utm_source=geeksforgeeks&utm_medium=article&utm_campaign=gfg_article_dsa_content_bottom) at a student-friendly price and become industry ready.  To complete your preparation from learning a language to DS Algo and many more,  please refer [**Complete Interview Preparation Course**](https://practice.geeksforgeeks.org/courses/complete-interview-preparation?utm_source=GeeksforGeeks&utm_medium=Text&utm_campaign=GFG_Article_Bottom_Text_CIP)**.**

In case you wish to attend **live classes**with experts, please refer [**DSA Live Classes for Working Professionals**](https://practice.geeksforgeeks.org/courses/geeks-classes-live?utm_source=geeksforgeeks&utm_medium=article&utm_campaign=gfg_article_dsa_content_bottom)and [**Competitive Programming Live for Students**](https://practice.geeksforgeeks.org/courses/competitive-programming-live?utm_source=geeksforgeeks&utm_medium=article&utm_campaign=gfg_article_cp_content_bottom).

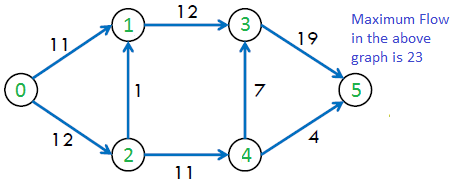
Ford-Fulkerson Algorithm for Maximum Flow Problem

* Difficulty Level : [Hard](https://www.geeksforgeeks.org/hard/)
* Last Updated : 24 Jun, 2021

Given a graph which represents a flow network where every edge has a capacity. Also given two vertices *source*‘s’ and *sink* ‘t’ in the graph, find the maximum possible flow from s to t with following constraints:  
**a)** Flow on an edge doesn’t exceed the given capacity of the edge.  
**b)** Incoming flow is equal to outgoing flow for every vertex except s and t.  
For example, consider the following graph from CLRS book.



The maximum possible flow in the above graph is 23. 



[Recommended: Please solve it on “***PRACTICE***” first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/find-the-maximum-flow/0)

Prerequisite : [**Max Flow Problem Introduction**](https://www.geeksforgeeks.org/max-flow-problem-introduction/)

**Ford-Fulkerson Algorithm**

The following is simple idea of Ford-Fulkerson algorithm:

**1)** Start with initial flow as 0.

**2)** While there is a augmenting path from source to sink.

Add this path-flow to flow.

**3)** Return flow.

**Time Complexity:** Time complexity of the above algorithm is O(max\_flow \* E). We run a loop while there is an augmenting path. In worst case, we may add 1 unit flow in every iteration. Therefore the time complexity becomes O(max\_flow \* E).  
**How to implement the above simple algorithm?**  
Let us first define the concept of Residual Graph which is needed for understanding the implementation.   
***Residual Graph*** of a flow network is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow. Every edge of a residual graph has a value called ***residual capacity*** which is equal to original capacity of the edge minus current flow. Residual capacity is basically the current capacity of the edge.   
Let us now talk about implementation details. Residual capacity is 0 if there is no edge between two vertices of residual graph. We can initialize the residual graph as original graph as there is no initial flow and initially residual capacity is equal to original capacity. To find an augmenting path, we can either do a BFS or DFS of the residual graph. We have used BFS in below implementation. Using BFS, we can find out if there is a path from source to sink. BFS also builds parent[] array. Using the parent[] array, we traverse through the found path and find possible flow through this path by finding minimum residual capacity along the path. We later add the found path flow to overall flow.   
The important thing is, we need to update residual capacities in the residual graph. We subtract path flow from all edges along the path and we add path flow along the reverse edges We need to add path flow along reverse edges because may later need to send flow in reverse direction (See following link for example).  
<https://www.geeksforgeeks.org/max-flow-problem-introduction/>  
Below is the implementation of Ford-Fulkerson algorithm. To keep things simple, graph is represented as a 2D matrix.

* C++
* Java
* Python
* C#

|  |
| --- |
| # Python program for implementation  # of Ford Fulkerson algorithm  from collections import defaultdict    # This class represents a directed graph  # using adjacency matrix representation  class Graph:        def \_\_init\_\_(self, graph):          self.graph = graph  # residual graph          self. ROW = len(graph)          # self.COL = len(gr[0])        '''Returns true if there is a path from source 's' to sink 't' in      residual graph. Also fills parent[] to store the path '''        def BFS(self, s, t, parent):            # Mark all the vertices as not visited          visited = [False]\*(self.ROW)            # Create a queue for BFS          queue = []            # Mark the source node as visited and enqueue it          queue.append(s)          visited[s] = True             # Standard BFS Loop          while queue:                # Dequeue a vertex from queue and print it              u = queue.pop(0)                # Get all adjacent vertices of the dequeued vertex u              # If a adjacent has not been visited, then mark it              # visited and enqueue it              for ind, val in enumerate(self.graph[u]):                  if visited[ind] == False and val > 0:                        # If we find a connection to the sink node,                      # then there is no point in BFS anymore                      # We just have to set its parent and can return true                      queue.append(ind)                      visited[ind] = True                      parent[ind] = u                      if ind == t:                          return True            # We didn't reach sink in BFS starting          # from source, so return false          return False          # Returns tne maximum flow from s to t in the given graph      def FordFulkerson(self, source, sink):            # This array is filled by BFS and to store path          parent = [-1]\*(self.ROW)            max\_flow = 0 # There is no flow initially            # Augment the flow while there is path from source to sink          while self.BFS(source, sink, parent) :                # Find minimum residual capacity of the edges along the              # path filled by BFS. Or we can say find the maximum flow              # through the path found.              path\_flow = float("Inf")              s = sink              while(s !=  source):                  path\_flow = min (path\_flow, self.graph[parent[s]][s])                  s = parent[s]                # Add path flow to overall flow              max\_flow +=  path\_flow                # update residual capacities of the edges and reverse edges              # along the path              v = sink              while(v !=  source):                  u = parent[v]                  self.graph[u][v] -= path\_flow                  self.graph[v][u] += path\_flow                  v = parent[v]            return max\_flow      # Create a graph given in the above diagram    graph = [[0, 16, 13, 0, 0, 0],          [0, 0, 10, 12, 0, 0],          [0, 4, 0, 0, 14, 0],          [0, 0, 9, 0, 0, 20],          [0, 0, 0, 7, 0, 4],          [0, 0, 0, 0, 0, 0]]    g = Graph(graph)    source = 0; sink = 5    print ("The maximum possible flow is %d " % g.FordFulkerson(source, sink))    # This code is contributed by Neelam Yadav |

**Output:**

The maximum possible flow is 23

The above implementation of Ford Fulkerson Algorithm is called [**Edmonds-Karp Algorithm**](http://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm). The idea of Edmonds-Karp is to use BFS in Ford Fulkerson implementation as BFS always picks a path with minimum number of edges. When BFS is used, the worst case time complexity can be reduced to O(VE2). The above implementation uses adjacency matrix representation though where BFS takes O(V2) time, the time complexity of the above implementation is O(EV3) (Refer [CLRS book](http://www.flipkart.com/introduction-algorithms-3rd/p/itmczynzhyhxv2gs?pid=9788120340077&affid=sandeepgfg) for proof of time complexity)  
This is an important problem as it arises in many practical situations. Examples include, maximizing the transportation with given traffic limits, maximizing packet flow in computer networks.  
[Dinc’s Algorithm for Max-Flow.](https://www.geeksforgeeks.org/dinics-algorithm-maximum-flow/)  
**Exercise:**   
Modify the above implementation so that it that runs in O(VE2) time.  
**References:**   
<http://www.stanford.edu/class/cs97si/08-network-flow-problems.pdf>   
[Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](http://www.flipkart.com/introduction-algorithms-3rd/p/itmczynzhyhxv2gs?pid=9788120340077&affid=sandeepgfg)  
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Attention reader! Don’t stop learning now. Get hold of all the important DSA concepts with the [**DSA Self Paced Course**](https://practice.geeksforgeeks.org/courses/dsa-self-paced?utm_source=geeksforgeeks&utm_medium=article&utm_campaign=gfg_article_dsa_content_bottom) at a student-friendly price and become industry ready.  To complete your preparation from learning a language to DS Algo and many more,  please refer [**Complete Interview Preparation Course**](https://practice.geeksforgeeks.org/courses/complete-interview-preparation?utm_source=GeeksforGeeks&utm_medium=Text&utm_campaign=GFG_Article_Bottom_Text_CIP)**.**

In case you wish to attend **live classes**with experts, please refer [**DSA Live Classes for Working Professionals**](https://practice.geeksforgeeks.org/courses/geeks-classes-live?utm_source=geeksforgeeks&utm_medium=article&utm_campaign=gfg_article_dsa_content_bottom)and [**Competitive Programming Live for Students**](https://practice.geeksforgeeks.org/courses/competitive-programming-live?utm_source=geeksforgeeks&utm_medium=article&utm_campaign=gfg_article_cp_content_bottom).