DESIGN & ANALYSIS OF ALGORITHMS (7)

techworldthink • March 10, 2022

18 Describe the Ford Fulkerson's procedure to compute the Max-Flow within a given Flow Network.

Ford-Fulkerson algorithm is a greedy approach for calculating the maximum possible flow in a network or a graph.

A term, flow network, is used to describe a network of vertices and edges with a source (S) and a sink (T). Each vertex, except S and T, can receive and send an equal amount of stuff through it. S can only send and T can only receive stuff.

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,The source vertex has all outward edge, no inward edge, and the sink will have all inward edge no outward edge.

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.

```
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.
```

Augmenting Path

It is the path available in a flow network.

Residual Graph

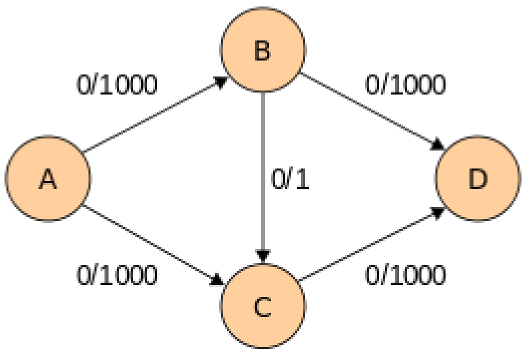
It represents the flow network that has additional possible flow.

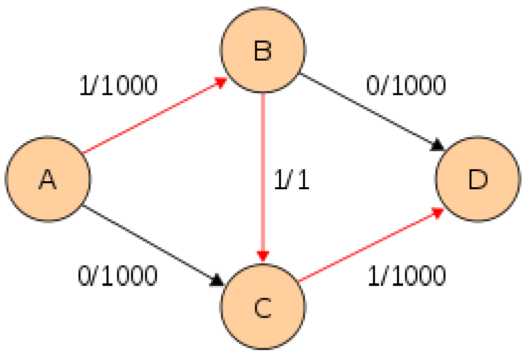
Residual Capacity

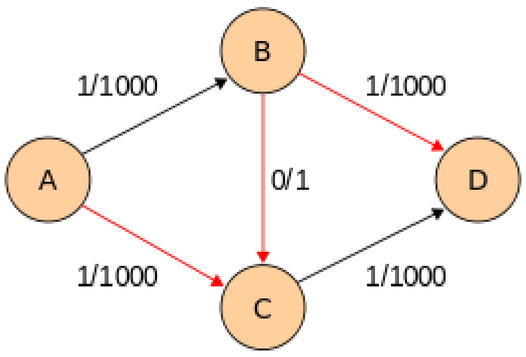
It is the capacity of the edge after subtracting the flow from the maximum capacity.

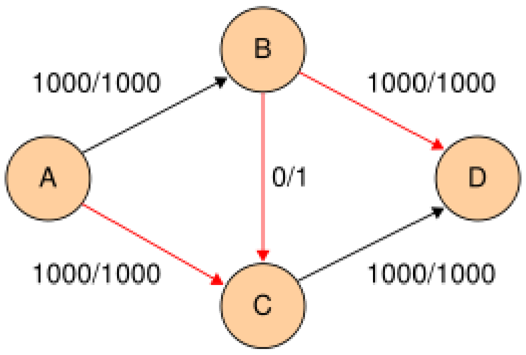
> *We can also consider reverse-path if required because if we do not consider them, we may never find a maximum flow.*

eg;









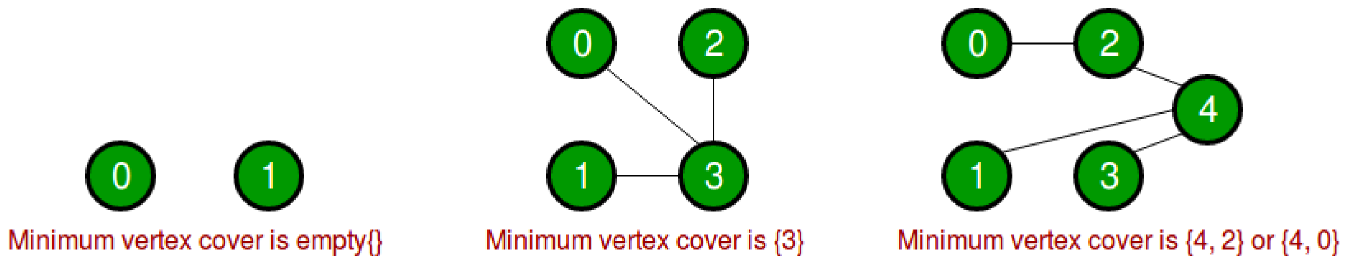
19 Explain the 2-approximation algorithm for Vertex Cover and justify its approximation ratio.

An approximation algorithm is a way of dealing with NP-completeness for an optimization problem. This technique does not guarantee the best solution. The goal of the approximation algorithm is to come close as much as possible to the optimal solution in polynomial time. Such algorithms are called approximation algorithm or heuristic algorithm.

In graph theory, a vertex cover (sometimes node cover) of a graph is **a set of vertices that includes at least one endpoint of every edge of the graph**. In computer science, the problem of finding a minimum vertex cover is a classical optimization problem.

A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either 'u' or 'v' is in the vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. ***Given an undirected graph, the vertex cover problem is to find minimum size vertex cover***.

The following are some examples.


0
1
Minimum vertex cover is empty{}
0
2
1
3
Minimum vertex cover is {3}
0
2
4
1
3
Minimum vertex cover is {4, 2} or {4, 0}

VertexCover

Vertex Cover Problem is a known NP Complete problem, i.e., there is no polynomial-time solution for this unless P = NP. There are approximate polynomial-time algorithms to solve the problem though.

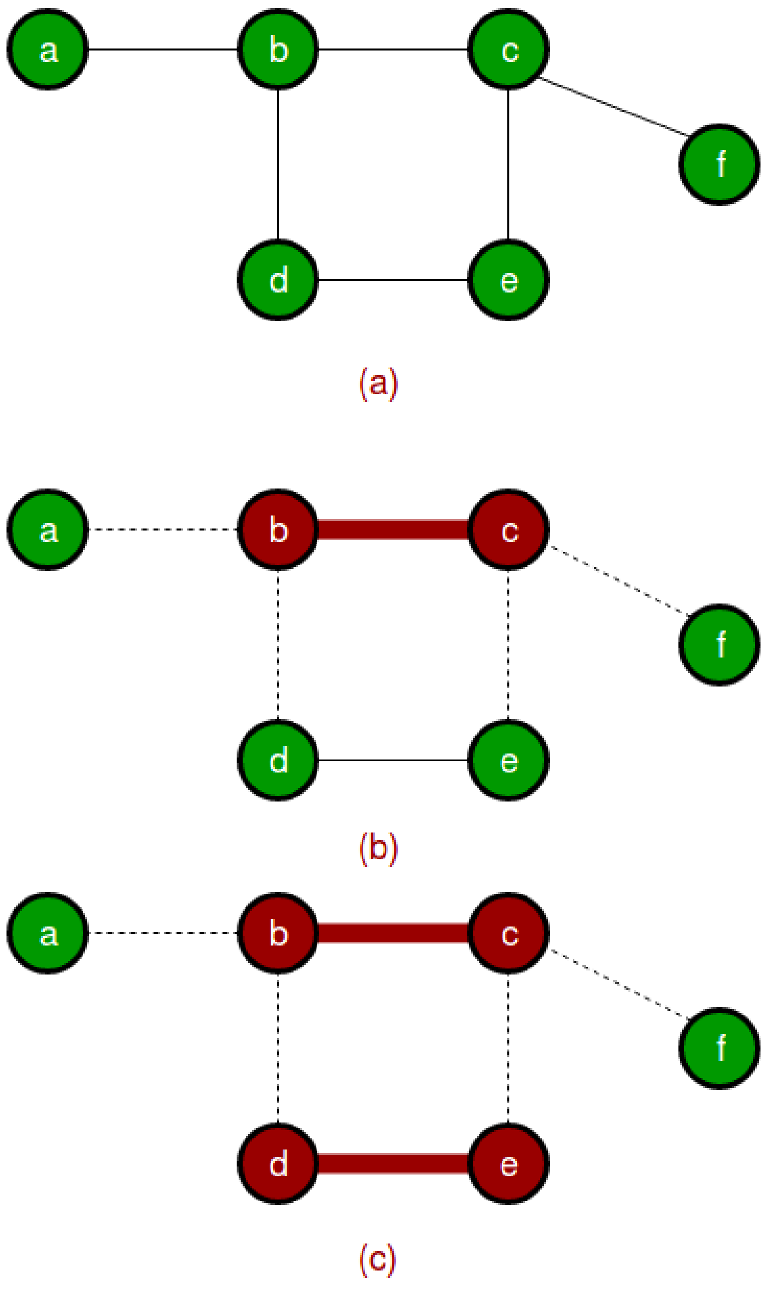
Naive Approach:

Consider all the subset of vertices one by one and find out whether it covers all edges of the graph. For eg. in a graph consisting only 3 vertices the set consisting of the combination of vertices are:{0,1,2,{0,1},{0,2},{1,2},{0,1,2}} . Using each element of this set check whether these vertices cover all all the edges of the graph. Hence update the optimal answer. And hence print the subset having minimum number of vertices which also covers all the edges of the graph.

Approximate Algorithm for Vertex Cover:

```
1) Initialize the result as {}
2) Consider a set of all edges in given graph.  Let the set be E.
3) Do following while E is not empty
...a) Pick an arbitrary edge (u, v) from set E and add 'u' and 'v' to
result
...b) Remove all edges from E which are either incident on u or v.
4) Return result
```

It can be proved that the above approximate algorithm never finds a vertex cover whose size is more than twice the size of the minimum possible vertex cover.



Minimum Vertex Cover is {b, c, d} or {b, c, e}

vertexCover

20 Describe Randomised Quick sort.

In QuickSort we first partition the array in place such that all elements to the left of the pivot element are smaller, while all elements to the right of the pivot are greater than the pivot. Then we recursively call the same procedure for left and right subarrays.

Unlike merge sort, we don't need to merge the two sorted arrays. Thus Quicksort requires lesser auxiliary space than Merge Sort, which is why it is often preferred to

Merge Sort. Using a randomly generated pivot we can further improve the time complexity of QuickSort.

```
QUICKSORT (array A, start, end)    {
          if (start < end)    {
                    p = partition(A, start, end)
                    QUICKSORT (A, start, p - 1)
                    QUICKSORT (A, p + 1, end)
          }
}


PARTITION (array, start, end){
    pivot = random(array,start,end);
    i = start; j = end;
    while(i<j){
        do { i++; } while(A[i]<=pivot);
        do { j++; } while(A[j]>pivot);
        if (i<j){ swap(A[i],A[j]); }
    }
    swap(A[start],A[j]);
   retutn j;
}
```