FIT2004: Supplementary notes for lecture 11

Network Flow

DANIEL ANDERSON [1]

Network flow models describe the flow of material through a network with fixed capacities, with applications arising in several areas of combinatorial optimisation. On the surface, network flow can be used to model material flow through networks which has applications to telecommunication networks, transport networks, financial networks and more. Beyond its obvious applications however, network flow turns out to be useful and applicable to solving a large range of combinatorial problems, including graph matchings, scheduling, image segmentation, project management and many more.

Summary: Network Flow

In this lecture, we cover:

- Network flow and the maximum flow problem
- The Ford-Fulkerson Algorithm for maximum flow
- The minimum cut problem
- The min-cut max-flow theorem
- The bipartite matching problem and its solution using max flow

Recommended Resources: Network Flow

- CLRS, Introduction to Algorithms, Chapter 26
- Weiss, Data Structures and Algorithm Analysis, Section 9.4
- `https://visualgo.net/en/maxflow` - Visualisation of maximum flow
- `https://visualgo.net/en/matching` - Visualisation of bipartite matching
- `https://youtu.be/VYZGlgzr_As` - MIT 6.046 lecture on maximum flow

Network Flow Problems

Consider a road network consisting of towns connected by roads, along each of which a company can send up to a certain number of trucks per day.

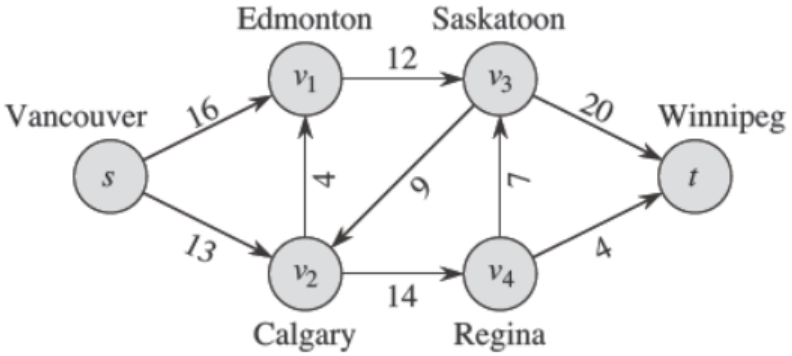

Figure 1: A road network of major cities in Canada. The edge weights represent the capacity of the roads, say the number of trucks that can be sent along that particular road per day. Image source: CLRS

[1] FACULTY OF INFORMATION TECHNOLOGY, MONASH UNIVERSITY: `daniel.anderson@monash.edu`. These notes are based on lecture slides by Arun Konagurthu, the textbook by CLRS, some video lectures from MIT OpenCourseware, and many discussions with students.

Our trucking company wants to know the maximum number of trucks that they can send per day from Vancouver to Winnipeg if they distribute them optimally. In this case, an optimal solution turns out to be to send:

- 12 trucks from Vancouver to Edmonton,
- 12 trucks from Edmonton to Saskatoon,
- 19 trucks from Saskatoon to Winnipeg,
- 11 trucks from Vancouver to Calgary,
- 11 trucks from Calgary to Regina,
- 7 trucks from Regina to Saskatoon,
- 4 trucks from Regina to Winnipeg,

which results in a total flow of 23 trucks per day.

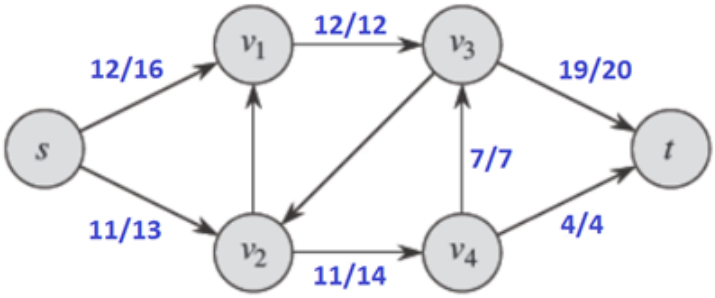

Figure 2: An optimal solution to the road network problem in Figure 1. Image source: Adapted from CLRS

A similar problem might be to consider a computer network, consisting of computers (nodes) that are connected via network. Each pair of connected computers can send messages to each other up to some fixed capacity, called their bandwidth. Given the bandwidths of each connection, we might want to find the total bandwidth between two computers given that we can send data along multiple different paths at the same time.

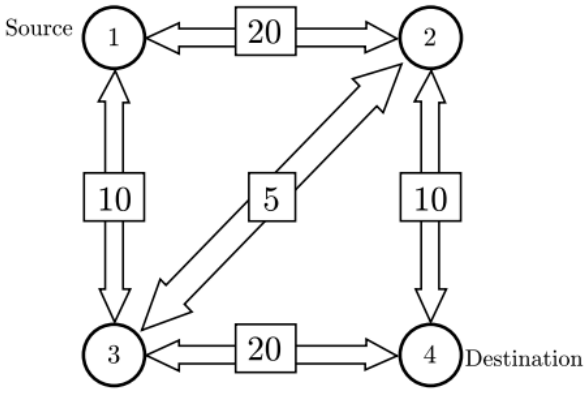

Figure 3: A computer network with edges labelled by the bandwidth of the connection between the two computers. The maximum possible bandwidth between the Source and the Destination is 30. Image source: ACM-ICPC World Finals 2000.

All of these are examples of **network flow** problems. In short, we have a network consisting of nodes, one of which is the "source," the location at which the flow is produced, and the "sink," the location at which the flow is consumed. The problem that we will consider is the **maximum network flow** problem, or max-flow for short, where we are interesting in maximising the total amount of flow that can be sent from the source node to the sink node.

Mathematical formulation as a linear program

The intuitive definition and examples above are more than enough to understand the maximum flow problem and the algorithms that we'll cover to solve it, but for completeness, here is the formal definition. It is not critical that you understand the formal definition, but it is very useful for proving things about the problem.

Problem Statement: Maximum Network Flow Problem

Let $G = (V, E)$ be a directed, edge-weighted network with two special vertices s and t, the source and sink respectively. Denote the capacity (edge weight) of an edge by $c(u, v)$. A flow f in G is a real-valued function

$$f : V \times V \to \mathbb{R}$$

that satisfies the following conditions:

- **Capacity constraint:** The flow along an edge must not exceed the capacity of that edge. Formally,

$$f(u, v) \leq c(u, v)$$

- **Flow conservation:** The amount of flow entering a node must be equal to the amount of flow leaving that node, with the exception of the source and sink. That is for all $u \in V \setminus \{s, t\}$,

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

  where $f(u, v) = 0$ if there is no edge (u, v), ie. if $(u, v) \notin E$.

The **value** of a flow $|f|$ is given by

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

The maximum flow problem then seeks to find a flow f that maximises the value:

$$\max_{\text{flow} f} |f| \quad \text{subject to the } \textbf{capacity} \text{ and } \textbf{conservation} \text{ constraints.}$$

The Ford-Fulkerson Algorithm

The Ford-Fulkerson Algorithm, often referred to as the Ford-Fulkerson Method since it encompasses a variety of potential implementations is one of the first and most intuitive algorithm for solving the maximum flow problem. Stated as briefly as possible, the algorithm is effectively:

While more flow can be pushed through the network, push more flow through the network.

Of course, we have to do some work to define what we really mean by pushing flow through a network, and to figure out how to find paths through which we are able to push flow. For example, simple naively pushing flow through the network until there is no longer a path from the source to the sink won't necessarily result in an optimal solution.

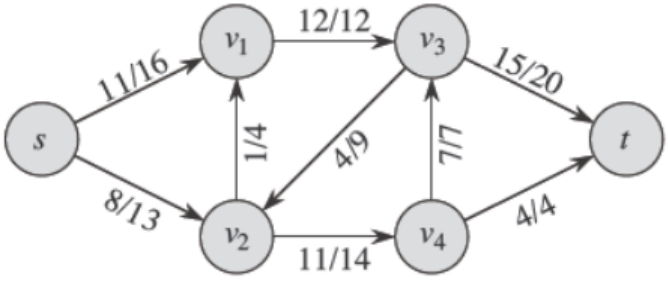

Figure 4: A flow on the road network from Figure 1. No more capacitated paths exist from the source to the sink, but the total flow of 19 is not optimal. In an optimal solution, we'd like to redirect the 4 flow that is going from v_3 to v_2 to go to t instead. In other words, we need a way to send suboptimal flow choices back and redirect them.
Image source: CLRS

To understand the Ford-Fulkerson method, we first need to define and explore the notion of the **residual network.**

Residual networks

Given a flow network $G = (V, E)$ and a flow f, the residual network G_f essentially represents how much more flow can be sent through each edge. For example, an edge with capacity 9 and flow 4 has residual capacity 5, as 5 additional units of flow can be pushed along it. The residual capacity of an edge (u, v) is given by

$$c_f(u, v) = c(u, v) - f(u, v).$$

Additionally, the residual network contains *back edges* or *reverse edges*, which are edges that flow in the opposite direction to those in E. The back edges are what allow the algorithm to "undo" flow by pushing flow back along the edge that it came from, and along a new direction instead. For each edge (u, v) with positive flow $f(u, v)$, the residual network contains an edge (v, u) with residual capacity

$$c_f(v, u) = f(u, v).$$

In other words, the back edge (v, u) has the ability to cancel out the flow that currently exists on the edge (u, v) by redirecting it down a different path.

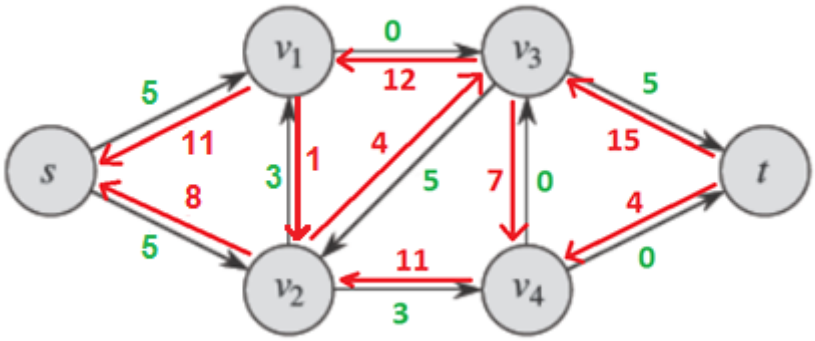

Figure 5: The residual network G_f corresponding to the network and flow in Figure 4. The green labelled edges are the forward edges from E with capacity $c(u, v) - f(u, v)$. The red edges are the back edges with capacity $f(v, u)$. Image source: Adapted from CLRS

Augmenting paths

The example of the suboptimal flow in Figure 4 demonstrates that we can not always improve a flow by seeking a capacitated path in G. In order to increase the flow in a network, we instead seek *Augmenting paths*, which are capacitated paths in the residual network G_f. An augmenting path may contain forward edges or back edges, which correspond to two situations conceptually:

- Augmenting along a forward edge simply corresponds to pushing more flow through that edge on the way from the source to the sink.
- Augmenting along a back edge corresponds to redirecting flow along that edge down a different path. The flow is reduced on the respective forward edge, and is moved to the remainder of the augmenting path.

Although the flow on the road network depicted in Figure 4 has no more capacitated paths from s to t, the residual network does indeed have a capacitated path given by $s \to v_2 \to v_3 \to t$. The total capacity of the path is the minimum capacity of the edges encountered along the path (that is the bottleneck of the path), as clearly we can send no more than this amount of additional flow along the path (or we would violate a capacity constraint.) Filling an edge to full capacity is often referred to as "saturating" the edge. Similarly, an edge is called saturated if the flow along it is at capacity.

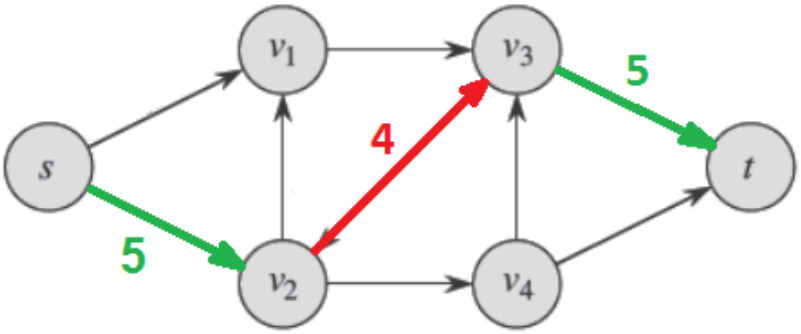

Figure 6: An augmenting path p the in residual network G_f in Figure 5. The total capacity of the augmenting path is 4, the bottleneck of the edge capacities in p. Image source: Adapted from CLRS

Augmenting 4 units of flow along the path p depicted in Figure 6 will result in a total flow of 23 units which is now optimal.

The key thing to understand here is that by augmenting along a back edge, we have redirected the flow that was once flowing along $v_3 \to v_2 \to v_4 \to t$ to flow directly from $v_3 \to t$ instead. As this flow is redirected away, the 4 units of flow that was being supplied to v_2 by v_3 is now supplied by s, so that the flow along $v_2 \to v_4 \to t$ is not reduced. Since any flow that is reduced by augmenting along a back edge always gets replaced, an augmentation always increases the total value of the flow.

The method

We now have everything we need to describe the Ford-Fulkerson method.

The Ford-Fulkerson Method

```
1: function MAX_FLOW(G = (V[1..N], E[1..N]), s, t)
2:     Set initial flow f to 0
3:     while there exists an augmenting path p in the residual network G_f do
4:         Augment the flow f along the augmenting path p
5:     end while
6:     return f
7: end function
```

Of course, the Ford-Fulkerson method is really just a skeleton. The method tells us what to do, but does not specifically tell us **how** to do it. There are many possible strategies for actually finding augmenting paths, each of which will result in a different time complexity.

An example implementation using depth-first search

The simplest way to find augmenting paths in the residual network is to simply run a depth-first search on G_f. Before we present the implementation, lets just look at a few tricks that we use to make the algorithm easier to write.

Use zero capacity edges to effectively represent back edges: In a naive implementation, we might try to write separate code for dealing with forward edges and back edges, but we can actually use a very simple trick to make them work the same, eliminating repeated code. All we need to do is define the capacity of the back edges to be 0, and to maintain that their flow value is always equal to the negative of the flow on the corresponding forward edge. This works because the formula for residual capacity on an edge

$$c_f(u,v) = c(u,v) - f(u,v),$$

when applied to a back edge will yield

$$c_f(v,u) = c(v,u) - f(v,u) = 0 - (-f(u,v)) = f(u,v),$$

which is the definition of the residual capacity on a back edge. In other words, this formulation allows us to treat forward and back edges the same since the formula for their residual capacities is now the same.

Augment flow in the same routine that finds an augmenting path: Once an augmenting path has been found, we know that we are going to augment along it, so we might as well make our depth-first search that finds augmenting paths also perform the augmentation at the same time. Otherwise we would simply have to write twice the length of code where we would have to locate an augmenting path and then separately traverse it again to add flow.

Edge structure: Let's assume that we have some structure representing an edge of the network, which contains the fields `capacity`, indicating the edge's capacity (remember that this will be zero for back edges), `flow`, which will indicate the current flow on that edge, and `rev` which contains a reference to the corresponding reverse edge (the back-edge of the forward edge or vice. versa).

Algorithm: Ford-Fulkerson using Depth-First Search

```
1:  function DFS(u, t, bottleneck)
2:      If u = t then return bottleneck
3:      visited[u] = True
4:      for each edge e = (u, v) leaving u do
5:          Set residual = e.capacity - e.flow
6:          if residual > 0 and not visited[v] then
7:              Set augment = dfs(v, t, min(bottleneck, residual))
8:              if augment > 0 then
9:                  e.flow += augment
10:                 e.rev.flow -= augment
11:                 return augment
12:             end if
13:         end if
14:     end for
15:     return 0
16: end function
17:
18: function MAX_FLOW(G = (V[1..N], E[1..M]), s, t)
19:     Set flow = 0, augment = 0
20:     do
21:         Set visited[1..N] = False
22:         augment = dfs(s, t, ∞)
23:         flow += augment
24:     loop while augment > 0
25:     return flow
26: end function
```

Time Complexity of the Ford-Fulkerson Algorithm

The time complexity of the Ford-Fulkerson algorithm depends on the strategy selected to find augmenting paths. Let's consider the simplest variant in which we use a depth-first search to find **any** augmenting path.

- Finding an augmenting path if one exists using a depth-first search from the source s takes $O(|E|)$ time.
- In the worst case, each augmentation only increases the amount of flow by one unit[2], and hence it could take up to $|f|$ augmentations, where $|f|$ is the value of a maximum flow.

Therefore, the total time complexity of the Ford-Fulkerson algorithm is $O(|E| \times |f|)$.

Better strategies for selecting augmenting paths

We can improve greatly on the time complexity bound of $O(|E| \times |f|)$ by selecting augmenting paths more carefully. We won't explore the details, but just summarise some of the common strategies that are employed by maximum flow algorithms.

- **Select shortest augmenting paths:** If we use a breadth-first search instead of a depth-first search to find augmenting paths, then we will always find an augmenting path with the fewest number of edges. This implementation is called the **Edmonds-Karp algorithm** and has a runtime complexity of $O(|V||E|^2)$.

[2]We have assumed for now that all capacities are integers. The maximum flow problem can in fact be generalised to consider non-integer capacities, although the Ford-Fulkerson algorithm is no longer guaranteed to work.

- **Select augmenting paths with the largest capacity:** Finding a path with the maximum possible capacity can be achieved by finding a maximum spanning tree of the residual graph G_f with Prim's algorithm and then augmenting along the resulting path from s to t. This is called the **fattest augmenting path algorithm** and results in a runtime of $O(|E|^2 \log(|V|) \log(\max c(u,v)))$.

The Minimum Cut Problem

Given a flow network $G = (V, E)$ with edge capacities and a designated source vertex s and sink vertex t, the minimum cut problem is the problem of finding a set of edges of minimum total capacity that if removed, would disconnect s from t.

Let's illustrate this with a realistic example, let's say you have a road network consisting of directed roads. Suppose that your tutor lives at vertex s and that Monash University is at vertex t. You have a test coming up in your next lab, so you want to sabotage your tutor's chances of getting to the lab on time to avoid taking the test! For each directed road, you know the dollar amount that it would cost to have that road temporarily shut down and blocked off. What is the minimum amount of money that you'll have to spend in order to ensure that there is no way for your tutor to get to the university?

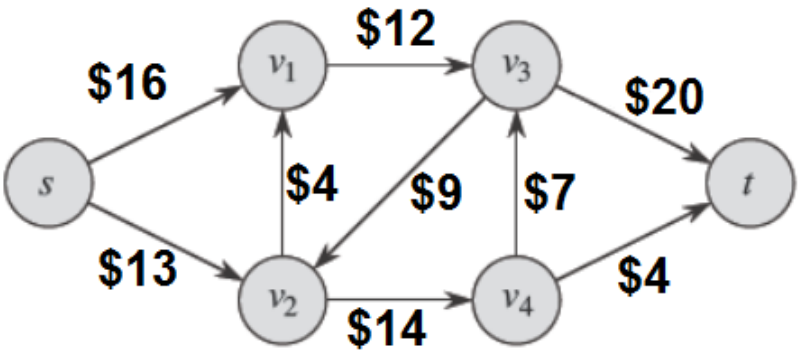

Figure 7: A road network, where edges are labelled by the cost of blocking off that particular road. Image source: Adapted from CLRS

This is precisely an example of a minimum cut problem. If we interpret the costs of blocking each road as the edges capacity, then the minimum cut is the cheapest way to disconnect s from t. By inspection, we can see that the minimum cut for the network above corresponds to blocking off the roads $v_1 \to v_3$, $v_4 \to v_3$ and $v_4 \to t$ for a total cost of $23.

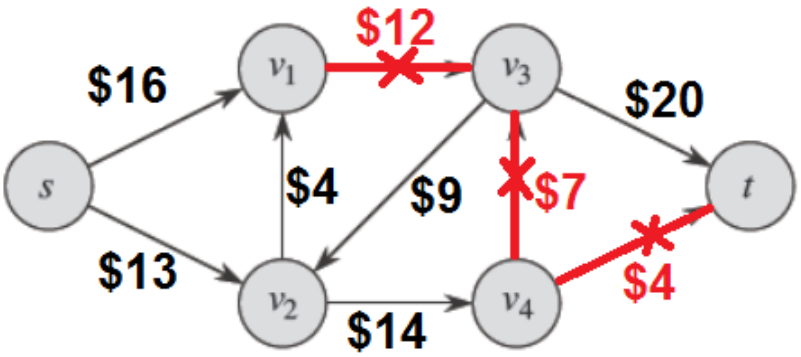

Figure 8: An optimal solution to the minimum cut problem. Image source: Adapted from CLRS

The minimum cut problem is intimately related to the maximum flow problem. Notice that this road network is actually the same network that we used as an example for maximum flow in the previous section, whose maximum flow turned out to be exactly 23 units. This is not a coincidence.

The Min-Cut Max-Flow Theorem

One of the most interesting theorems in network theory is the **Min-Cut Max-Flow Theorem.** Stated formally, the minimum cut problem is as follows.

Problem Statement: Minimum Cut Problem

Let $G = (V, E)$ be a directed, edge-weighted network with two special vertices s and t, the source and sink respectively. Denote the capacity (edge weight) of an edge by $c(u, v)$.

An **s-t cut** $C = (S, T)$ is a partition of the vertices V into two disjoint subsets S and T such that $s \in S$ and $t \in T$. The **capacity** of the cut C is defined to be the total capacity of all edges that begin in S and end in T, in other words, all edges that **cross** the cut. Formally,

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

where as usual, we take $c(u, v) = 0$ if there does not exist an edge (u, v). The **minimum cut** problem seeks an s-t cut $C = (S, T)$ such that $c(S, T)$ is as small as possible.

The minimum cut problem and the maximum flow problem are related by the Min-Cut Max-Flow theorem. Informally, in a given flow network, the minimum cut and maximum flow problems have the same solution.

Theorem: Min-Cut Max-Flow Theorem

Given a flow network $G = (V, E)$ with a source vertex s and sink vertex t, the value of a maximum s-t flow in G is equal to the minimum capacity s-t cut in G.

$$\max_f |f| = \min_{(S,T)} c(S, T)$$

Proof

For a formal proof of the theorem, see CLRS, Theorem 26.6. We will consider a more intuitive version of the proof, which follows roughly the same structure as the formal proof, but using intuitive arguments in place of formal definitions.

If we consider a flow network G, some flow f and some cut (S, T), the net amount of flow that crosses the cut, ie. moves through an edge from S into T must be equal to the value of the flow $|f|$. This has to be true because if $|f|$ total units of flow are moving from s (contained in S) to t (contained in T), then they must cross through the cut (S, T). Given this, the value of the maximum flow can not be greater than the capacity of any (S, T) cut, since it must travel through it, so in particular, the value of the maximum flow must be less than or equal to the capacity of the minimum cut.

Consider a maximum flow f and the particular s-t cut (S, T) such that S contains all vertices reachable from s in the residual graph G_f, and T contains all remaining vertices. (S, T) is a valid cut since t is not reachable from s, otherwise there would be an augmenting path, which would imply that the flow f was not in fact maximum. Suppose that the value of f was less than the capacity of (S, T), ie. that $|f| < c(S, T)$, then either there is an edge crossing (S, T) that is not saturated, or there is an edge bringing flow from T into S. If there is an unsaturated edge (u, v) such that $u \in S$ and $v \in T$, then there is a path from S to T in the residual graph, which is a contradiction. Similarly, if there is an edge bringing flow from T into S, then the corresponding back-edge from S into T has non-zero residual capacity, and hence there is a path from S to T in the residual graph, which is also a contradiction. Therefore, the value of the flow f must be equal to the capacity of the cut (S, T).

Since the value of the maximum flow f can not exceed the capacity of any cut, and we have found a cut such that $|f| = c(S, T)$, we can conclude that the value of the maximum flow is equal to the capacity of the minimum cut.

Not only does the proof above illuminate why the theorem is true, it also shows us how to construct a minimum cut if we have found a maximum flow. If $G = (V, E)$ is a network and f is the maximum flow, then the minimum cut (S, T) can be found by taking S equal to all of the vertices that are reachable from s via the residual graph G_f, and taking T as all remaining vertices. The proof shows that this will indeed be a minimum s-t cut in G.

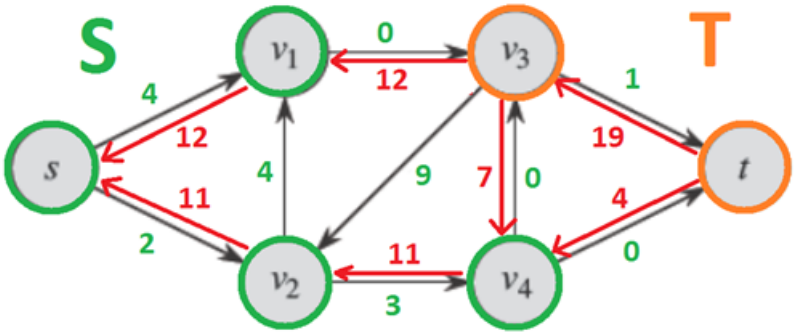

Figure 9: The residual network G_f for the maximum flow in the network in Figure 7. The vertices reachable from s are $\{s, v_1, v_2, v_4\}$, leaving $T = \{v_3, t\}$, across which, the capacity can be seen to equal to $12+7+4 = 23$, which agrees with the value of the maximum flow f. Image source: Adapted from CLRS

Corollary: Correctness of Ford-Fulkerson

The Min-Cut Max-Flow theorem also immediately shows us that the Ford-Fulkerson algorithm is correct. When there does not exist an augmenting path from s to t in the residual graph, the flow across the cut $(S = \{v : v \text{ reachable from } s \text{ in } G_f\},\ T = V \setminus S)$ is equal to the capacity of the cut. By the Min-Cut Max Flow theorem, this flow is therefore a maximum flow.

Applications of Max Flow: Bipartite Matching

Recall that a bipartite graph is one where the vertices can be separated into two disjoint subsets L and R such that every edge connects a vertex $u \in L$ to a vertex $v \in R$. In other words, there are no edges strictly inside L or strictly inside R.

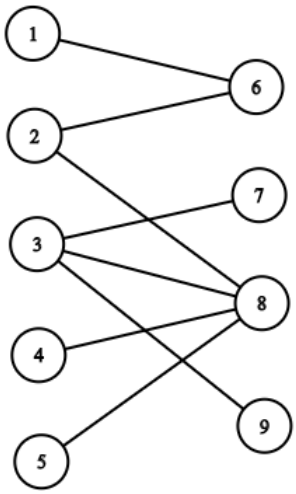

Figure 10: A bipartite graph made of the vertex sets $L = \{1,2,3,4,5\}$ and $R = \{6,7,8,9\}$.

We could interpret this bipartite graph as a set of applicants applying for jobs. Suppose that the set $L = \{1,2,3,4,5\}$ represents the job applications, and the set $R = \{6,7,8,9\}$ the available jobs that they have applied for. There is an edge between the applicant u and the job v if person u is qualified for job v. The bipartite matching problem asks us to find the maximum number of applicants that we can allocate to jobs such that no applicant gets multiple jobs, no job is taken by multiple people and that only qualified applicants can receive a job. For this example, the greatest number of applicants that can be successfully matched is three.

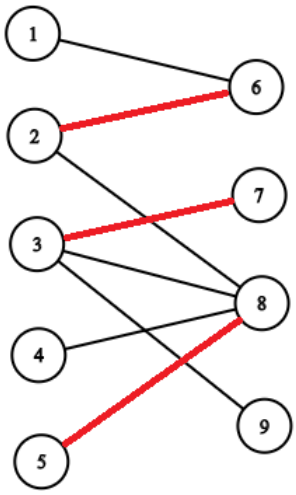

Figure 11: A maximum matching on the bipartite graph in Figure 10.

Problem Statement: Maximum Bipartite Matching

Given an undirected, bipartite graph $B = (V, E)$, a **matching** is a subset M of the edges E such that no vertex $v \in V$ is incident to multiple edges in M. The maximum bipartite matching problem seeks a matching M of the graph B with the largest possible number of edges.

Although the maximum bipartite matching problem does not sound very similar to the maximum flow problem, it can actually be solved by constructing a corresponding flow network and finding a maximum flow. The key idea is to use units of flow to represent matches. We construct the corresponding flow network by taking the bipartite graph $B = (V, E)$ and adding new sink (s) and source (t) vertices. We then connect the source vertex s to every vertex in the left subset L and connect every vertex in the right subset R to the sink vertex t, and direct all of the original edges in E to point from L to R. We then finally assign every edge a capacity of one.

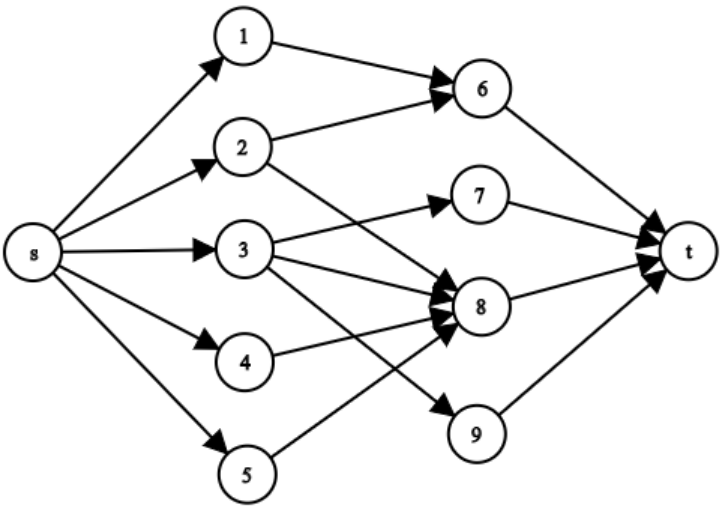

Figure 12: The corresponding flow network for the bipartite graph B in Figure 10. All edges have capacity 1.

Since every edge has a capacity of one, at most one unit of flow can be sent from s through any vertex $u \in L$ and at most one unit of flow can leave any $v \in R$ into t. This ensures that no vertex is used in multiple matches, as per the requirements of a matching. After finding a maximum flow, matches will correspond to edges from L to R that are saturated. The flow being a maximum flow ensures that the maximum possible number of matches are obtained.

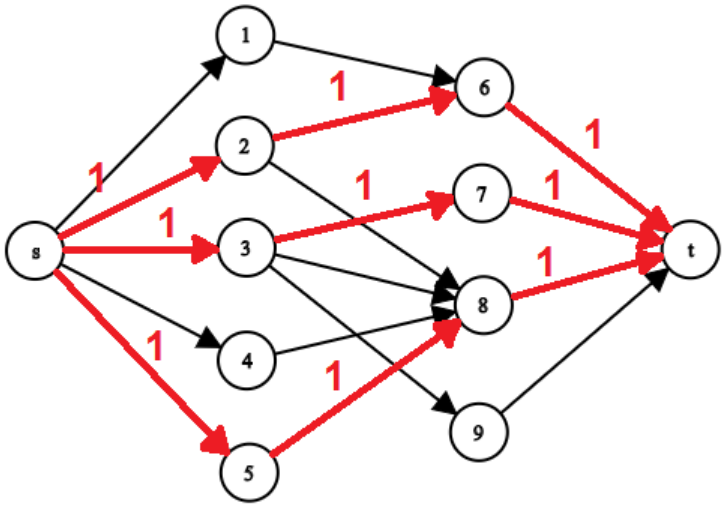

Figure 13: A maximum flow in the corresponding flow network. The saturated edges are a maximum matching with size 3.

Given a bipartite graph $B = (V, E)$, the maximum possible number of matchings can not be greater than the number of vertices, so the value of the flow in the corresponding flow network is bounded above by $|f| \leq |V|$. Using the Ford-Fulkerson algorithm to find the maximum flow therefore results in a runtime of $O(|V||E|)$ to solve the maximum bipartite matching problem.

This algorithm is sometimes called the **alternating paths algorithm** for bipartite matching, since the augmenting paths that will be found by Ford-Fulkerson will strictly alternate between the left and right parts of the graph using forward edges and back edges sequentially. Indeed, in practical implementations of the algorithm, constructing the flow network explicitly is not required, since as long as the current matches are kept track of, augmenting paths can be found by beginning at an arbitrary unmatched vertex and traversing the graph, alternating between unmatched vertices on the left and matched vertices on the right.