In this lecture, we will discuss the Ford-Fulkerson algorithm for finding the max flow in flow networks. We will then show how to use this algorithm to find a maximum matching in bipartite graphs.

1 Flows and flow networks

A flow network $G = (V, E)$ is a directed graph such that

1. Every edge has a capacity $c(e) \geq 0$. (*We assume $c(e)$ is integer.*)
2. There is a single source $s \in V$. (*We assume no edge enters s.*)
3. There is a single sink $t \in V$. (*We assume no edge leaves t.*)

We also assume that

- *If $(u, v) \in E$ then $(v, u) \notin E$.*

  This assumption is just for the purposes of the analysis: it is easy to transform a network with antiparallel edges (that is, a graph where $(u, v) \in E$ and $(v, u) \in E$) into a flow network (*how?*).

- *There is at least one edge incident to every node.* Therefore G has $m \geq \frac{n}{2}$ edges.

An example of a flow network appears in Figure 1(a).

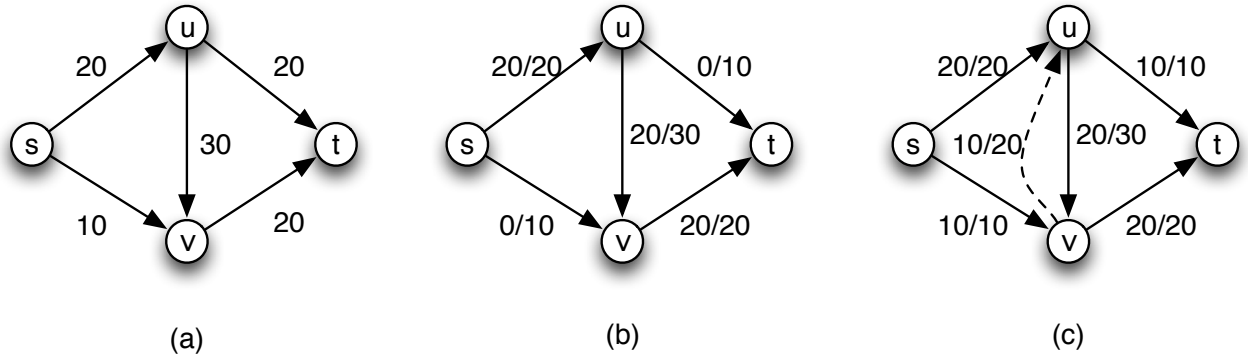


Figure 1: An example flow network, a flow of value 20 and a maximum flow of value 30.

Given a flow network G, an s-t flow in G is a function that maps every edge to a non-negative real number, that is

$$f : E \to R^+.$$

Intuitively, $f(e)$ represents the amount of flow that e carries. A flow satisfies two kinds of constraints:

1. **Capacity constraints:** $\forall e \in E$, $0 \leq f(e) \leq c(e)$.

2. **Flow conservation:** $\forall v \in V - \{s,t\}$, $\sum_{(u,v)\in E} f(u,v) = \sum_{(v,w)\in E} f(v,w)$, i.e.,

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e).$$

The flow conservation constraints basically state that the flow **into** vertex v equals the flow **out of** vertex v. To keep the notation concise and intuitive, we define for a vertex v:

(a) $f^{\text{out}}(v) = \sum_{e \text{ out of } v} f(e)$,
(b) $f^{\text{in}}(v) = \sum_{e \text{ into } v} f(e)$,

(This notation extends to sets of vertices.) We may now rewrite the flow conservation constraints as: for all $v \in V - \{s,t\}$, $f^{\text{in}}(v) = f^{\text{out}}(v)$.

Since the source is allowed to have outgoing edges (but no incoming), it may carry flow on these edges.

Definition 1 *The value of a flow f, denoted by $|f|$, is*

$$|f| = \sum_{v\in V} f(s,v) = \sum_{e \text{ out of } s} f(e) = f^{out}(s)$$

Definition 2 (Max flow) *Given a flow network G, find a flow of maximum possible value.*

We will first examine whether there is some natural upper bound for the maximum value a flow may take.

Definition 3 *An s-t cut (S,T) in G is a partition of V into two sets S and T, such that $s \in S$ and $t \in T$.*

To go from source s to sink t, f must cross (S,T) at some point. So it must use some (and at most all) of the capacity of the edges crossing this cut. Then the value of the flow cannot exceed $\sum_{\text{e out of } S} c(e)$.

Definition 4 *The capacity $c(S,T)$ of s-t cut (S,T) is given by*

$$c(S,T) = \sum_{e \text{ out of } S} c(e).$$

Intuitively, the above suggests that the max flow is upper bounded by the capacity of every cut in the flow network, that is

$$\max_f |f| \leq \min_{(S,T) \text{ cut in } G} c(S,T). \tag{1}$$

In the following section we will discuss an algorithm that efficiently computes the max flow in G, establishes that $\max_f |f| = \min_{(S,T) \text{ cut in } G} c(S,T)$ and efficiently computes such a cut of minimum capacity in G.

2 Residual graph and augmenting paths

Suppose we want to find the maximum flow in the graph of Figure 1(a). We could follow a greedy approach (see Figure 1(b)) and first push 20 units of flow over (s, u) (we do not push anything on (s, v) to keep flow conservation straightforward). Even though by inspection we see that max flow is 30 units, now we are stuck: we can no longer push any flow without exceeding some capacity!

What we would need to do at this point is **push back** flow so that we can send more flow in a different direction. E.g., suppose we push back 10 units of flow along (v, u), as denoted by the dashed lines in Figure 1(c). Then we could send 10 more units of flow from s to t along edges $(s, v), (v, u), (u, t)$, as shown in Figure 1(c) to obtain a valid flow in G with value 30.

Note that what we basically created by pushing flow back on (v, u) is an s-t path on which we are pushing flow

- **Forward**, on edges with leftover capacity (e.g, (s, v))
- **Backward**, on edges that are already carrying flow so as to divert it to a different direction.

We formalize these ideas by using the notion of the residual graph G_f.

Definition 5 *Given flow network G and flow f, the residual graph G_f consists of*

- *the same vertices as G;*
- *for every edge $e = (u, v) \in E$ such that $f(e) < c(e)$, an edge $e = (u, v)$ with capacity $c_f(e) = c(e) - f(e)$ (**forward** edges);*
- *for every edge $e = (u, v) \in E$ such that $f(e) > 0$, an edge $e = (v, u)$ with capacity $c_f(e) = f(e)$ (**backward** edges).*

So G_f has $\leq 2m$ edges.

The residual graph G_f provides a roadmap for augmenting f as follows. Let P be a simple s-t path in G_f. We will augment f by pushing extra flow on P (for this reason, we call P **augmenting path**). Clearly the maximum amount of flow we can push on **every** edge e in P without violating capacity constraints in G_f is $\min_{e \in P} c_f(e)$. We define this as the capacity $c(P)$ of path P. So $c(P)$ is the minimum residual capacity of any edge of P and it corresponds to the maximum amount of flow we can safely push on an augmenting path P.

Given an augmenting path P, we define operation Augment() in G_f as follows.

```
Augment(f, P)
  for each edge (u, v) ∈ P do
    if e = (u, v) is a forward edge then
      f'(e) = f(e) + c(P)
    else// that is, e = (u, v) is a backward edge, so (v, u) ∈ E
      f'(v, u) = f(v, u) − c(P)
    end if
  end for
  Return f'
```

So the result of Augment() is a new flow f' (as Fact 1 will guarantee) obtained by increasing the flow values on the edges of P in G_f, which corresponds to increasing and decreasing flow values on edges of G.

Fact 1 *f' is a flow.*

Proof. We need to show that f' satisfies capacity and flow conservation constraints.

- Capacity constraints: We need to show that for all $e \in E$, $0 < f'(e) < c(e)$. Now f' only differs from f in the edges on P so we just need check capacity constraints on these edges. There are two cases:

  1. If e is a forward edge, then $f'(e) = f(e) + c(P)$. So
     - $f'(e) \geq f(e) \geq 0$
     - $f'(e) = f(e) + c(P) \leq f(e) + c_f(e) = f(e) + c(e) - f(e) = c(e)$, since $c(P)$ is the minimum over all residual capacities.
  2. If $e = (u, v)$ is a backward edge, then we decrease the flow on edge (v, u) in G, thus $f'(v, u) = f(v, u) - c(P)$. So we need to check how the flow on edge (v, u) is updated.
     - $f'(v, u) \leq f(v, u) \leq c(v, u)$.
     - Since $e = (u, v) \in P$, $c(P) \leq c_f(u, v)$. For backward edges, we defined $c_f(u, v) = f(v, u)$. Hence $c(P) \leq f(v, u)$. So we have $f'(v, u) = f(v, u) - c(P) \geq 0$.

- Conservation constraints: again, we only need check those on the nodes v of P. We know that f satisfies flow conservation constraints and we need verify that the change in flow into v equals the change in flow out of v.

  There are four cases for edges $(u, v), (v, w) \in P$:

  1. If edges (u, v), (v, w) are both forward edges then $f'(u, v) = f(u, v) + c(P)$ and
  $$f'^{\text{in}}(v) = f^{\text{in}}(v) + c(P), f'^{\text{out}}(v) = f^{\text{out}}(v) + c(P),$$
  2. If edge (u, v) is forward, edge (v, w) is backward then $f'(w, v) = f(w, v) - c(P)$ and
  $$f'^{\text{in}}(v) = f^{\text{in}}(v) + c(P) - c(P), f'^{\text{out}}(v) = f^{\text{out}}(v),$$
  3. If edge (u, v) is backward, edge (v, w) is forward
  $$f'^{\text{in}}(v) = f^{\text{in}}(v), f'^{\text{out}}(v) = f^{\text{out}}(v) + c(P) - c(P),$$
  4. If edges (u, v), (v, w) are both backward edges then
  $$f'^{\text{in}}(v) = f^{\text{in}}(v) - c(P), f'^{\text{out}}(v) = f^{\text{out}}(v) - c(P),$$

□

3 The Ford – Fulkerson algorithm

In this section we show that the following algorithm computes the maximum flow in a flow network and analyze its running time.

Ford-Fulkerson($G = (V, E, c), s, t$)
  for all $e \in E$ **do** $f(e) = 0$

end for
while there is an s-t path in G_f **do**
  Let P be a simple s-t path in G_f
  $f' = \text{Augment}(f, P)$
  Update $f = f'$
  Update $G_f = G_{f'}$
end while
Return f'

3.1 Running time analysis

We will first examine the running time of this algorithm. Note that if every iteration of the while loop returns a flow increased by an integer amount, and there is a finite upper bound to the flow, then the algorithm terminates. Fortunately, integrality comes for free in the Max Flow problem when edge capacities are integers.

Fact 2 *During execution of the Ford-Fulkerson algorithm, the flow values $\{f(e)\}$ and the residual capacities in G_f are all integers.*

The proof is by induction on the number of iterations (*exercise*).

Fact 3 *Let f be a flow in G and P a simple s-t path in G_f with residual capacity $c(P) > 0$. Then after Augment()*

$$|f'| = |f| + c(P) \geq |f| + 1.$$

Proof. Recall that $|f| = f^{\text{out}}(s)$. Since P is an s-t path, it contains an edge out of s, say (s, u). Since P is simple, it does not contain any edge entering s (P is in G_f, where there are edges entering s!): otherwise, s would be visited again. Since no edge enters s in G, (s, u) is a forward edge in G_f, thus the flow on this edge is updated to $f(s, u) + c(P) \geq f(s, u) + 1$. Since no other edge going out of s is updated, it follows that the value of f' is $|f'| = |f| + c(P) \geq |f| + 1$. □

A trivial upper bound for $|f|$ is $\sum_{e \text{ out of } s} c(e)$; call this number C. Then Fact 3 guarantees that the Ford-Fulkerson algorithm executes at most C iterations. The running time of each iteration is bounded as follows:

- $O(m + n)$ to create G_f using adjacency list representation.
- $O(m + n)$ to run BFS or DFS to find the augmenting path (P is just an s-t path in G_f).
- $O(n)$ to perform Augment() since P has at most $n - 1$ edges.

Therefore, one iteration requires $O(m)$ time and the overall running time of Ford-Fulkerson is $O(mC)$.

Note that this is a pseudo-polynomial time algorithm since it is not polynomial in the description of the input number C: to describe C, we use $\log_2 C$ bits.

However, there is an easy way to obtain polynomial algorithm for Max Flow. Essentially, we don't want to use DFS to find the augmenting path. Instead, suppose we always augment the flow along the shortest path from s to t, that is, the path with the fewest edges. We can find this path P using BFS. This algorithm, called the Edmonds-Karp algorithm, yields a running time of $O(nm^2)$ (see your textbook for a detailed analysis of the running time if you are interested).

3.2 Optimality of Ford-Fulkerson

We will now show that the flow computed by the Ford-Fulkerson algorithm upon termination is a maximum flow. We first observe that the algorithm terminates when there is no augmenting s-t path in G_f. So consider the residual graph G_f upon termination of the algorithm and the cut (S^*, T^*), where S^* contains the set of nodes reachable from the source s and T^* every other node.

First, we claim that (S^*, T^*) is an s-t cut, that is, $s \in S^*$ and $t \in T^*$: this is true since if $t \in S^*$, then there is an s-t path in G_f, which contradicts termination of the algorithm. So $t \in T^*$. Next, we claim that there is no edge from any node in S^* to any node in T^* in G_f. For suppose there is an edge (u, v) with $u \in S^*$ and $v \in T^*$: then v is reachable from s, so v should be in S^*.

Now consider any edge $e = (u, v)$ in G with $u \in S^*$ and $v \in T^*$ (see Figure 2). Then it must be that $f(e) = c(e)$. Otherwise, e would appear in G_f as a forward edge with a residual capacity of $c(e) - f(e)$. Similarly, consider any edge $e = (u, v)$ in G with $u \in T^*$ and $v \in S^*$. If e had any

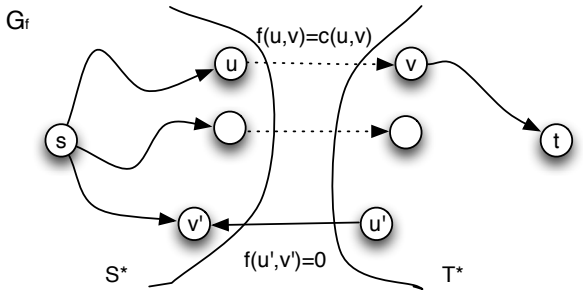


Figure 2: The cut (S^*, T^*) in G_f upon termination of the Ford-Fulkerson algorithm. Dotted lines denote edges that exist in G but not in G_f.

flow on it, then we would add $(u, v) \in G_f$ with flow $f(u, v) = f(e)$ to allow to "undo" the flow on e in G. Hence it must be that $f(e) = 0$.

Let's collect some further information about the cut (S^*, T^*). We define the **net flow** across an s-t cut (S, T) as the amount of flow leaving the cut minus the amount of flow entering the cut, that is

$$f^{\text{out}}(S) - f^{\text{in}}(S). \tag{2}$$

Then the net flow across the cut (S^*, T^*) is given by

$$f^{\text{out}}(S^*) - f^{\text{in}}(S^*) = \sum_{e \text{ out of } S^*} f(e) - \sum_{e \text{ into } S^*} f(e) = \sum_{e \text{ out of } S^*} c(e) - 0 = c(S^*, T^*) \tag{3}$$

So now we have exhibited an s-t cut with net flow equal to its capacity. If we could relate the value of the flow $|f|$ (that is, the flow out of s) to the net flow of this cut, and in particular, if we could show them equal, then f would be a flow with value equal to the capacity of some cut in the graph. Using our intuition from the beginning of Section 1 about how the value of the maximum flow and the capacity of any cut relate, we would then have strong evidence that f is indeed maximum since $|f|$ cannot exceed the capacity of any cut in G.

We will now show that the net flow across any s-t cut indeed equals the value of the flow in the flow network.

Lemma 1 *Let f be any s-t flow, and (S, T) any s-t cut. Then*

$$|f| = f^{out}(S) - f^{in}(S).$$

Proof. First, we rewrite

$$|f| \;=\; f^{\text{out}}(s) = \sum_{v \in S} \Big(f^{\text{out}}(v) - f^{\text{in}}(v) \Big) \tag{4}$$

since for every $v \in S - \{s\}$ the terms in the right-hand side of (4) cancel out because of flow conservation constraints and $f^{\text{in}}(s) = 0$.

Next we rewrite the right-hand side of the equation 4 in terms of the edges that participate to these sums. There are three types of edges:

1. Edges with both endpoints in S: such edges appear once in the first sum in equation 4 and once in the second, hence their flows cancel out.
2. Edges with the tail in S and head in T: such edges contribute to the first sum in equation 4 so they appear with a $+$.
3. Edges with the head in S and tail in T: such edges contribute to the second sum in equation 4 so they appear with a $-$.

In effect, we may rewrite the right-hand side of equation 4 as

$$\sum_{e \text{ out of } S} f(e) - \sum_{e \text{ into } S} f(e).$$

The lemma follows. □

We may now obtain a very simple, formal proof for equation 1.

Corollary 1 *Let f be any s-t flow and (S, T) any s-t cut. Then*

$$|f| \leq c(S, T).$$

Proof.

$$|f| = f^{\text{out}}(S) - f^{\text{in}}(S) \leq f^{\text{out}}(S) \leq c(S, T).$$

□

We may now conclude the proof of optimality of the Ford-Fulkerson algorithm as follows. By Corollary 1, $|f|$ is at most $c(S, T)$ for every cut (S, T); in particular,

$$|f| \leq c(S^*, T^*).$$

By Lemma 1, $|f|$ is equal to the net flow of any (S, T) cut; in particular,

$$|f| = f^{\text{out}}(S^*) - f^{\text{in}}(S^*).$$

By Equation 3, the net flow of f is exactly $c(S^*, T^*)$. Hence the above becomes

$$|f| = f^{\text{out}}(S^*) - f^{\text{in}}(S^*) = c(S^*, T^*).$$

Then the flow computed by Ford-Fulkerson is a max flow since it cannot be increased anymore (otherwise, it would exceed the capacity of (S^*, T^*)). So we proved the following theorem.

Theorem 1 *If f is an s-t flow such that there is no s-t path in G_f, then there is an s-t cut (S^*, T^*) in G such that $|f| = c(S^*, T^*)$. Therefore, f is a max flow and (S^*, T^*) is a cut of min capacity.*

As a by-product of our analysis, we also obtain that (S^*, T^*) is a cut of minimum capacity, which can be computed by BFS (or DFS) in G_f upon termination of the algorithm in order to obtain all nodes reachable from s. So (S^*, T^*) is computable in $O(m)$.

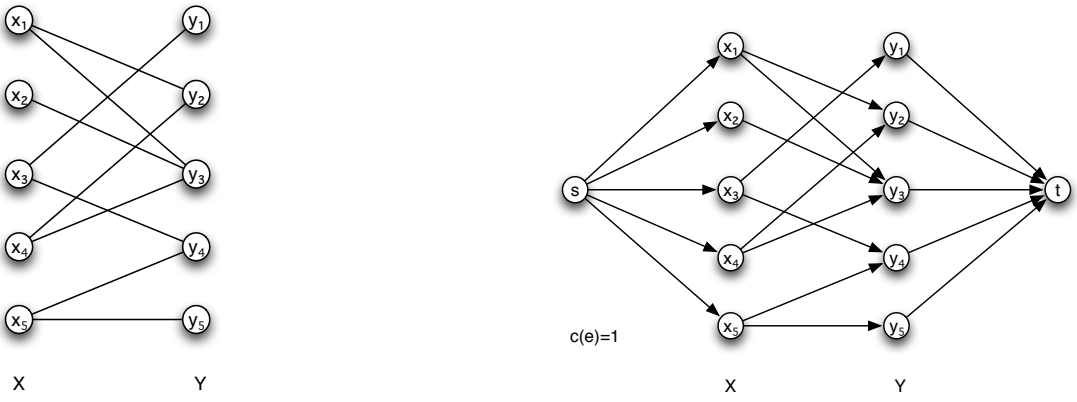
Theorem 2 (Max-flow Min-cut) *In every flow network, the maximum value of an s-t flow equals the minimum capacity of an s-t cut.*

Another important by-product of our analysis follows from Theorem 1 and Fact 2.

Theorem 3 (Integrality theorem) *If all capacities in a flow network are integers, then there is a maximum flow for which every flow value $f(e)$ is an integer.*

4 Application: finding maximum matching in bipartite graphs

In the first lecture on graph algorithms, we talked about bipartite graphs, that is, graphs where we can partition the set of vertices into two disjoint sets X, Y such that all edges in the graph have one endpoint in one set and one endpoint in the other. The problem we considered was to find a one-to-one matching of people with jobs (see Figure 3(a)). That is, we want to find a subset $M \subseteq E$ such that every person x_i is matched to a job y_j, and every node in X, Y appears exactly once in M.



(a) A bipartite graph G

(b) The derived flow network G'

Figure 3: Transforming a bipartite graph G into a flow network G'.

This kind of matching is called a **perfect matching**. A perfect matching might not always be possible (e.g., if $|X| \neq |Y|$). Since matchings model situations where one entity is assigned to another (e.g., jobs to people, jobs to machines, etc.), we might still want to find a matching of maximum size. Note that, if we had an algorithm to find maximum matchings then we could also find perfect matchings (*why?*).

Definition 6 *A matching M is a subset of edges where every vertex in $X \cup Y$ appears at most once.*

Finding maximum matchings in bipartite graphs is one of the oldest combinatorial problems. We will show how to use the Ford-Fulkerson algorithm to find maximum (and perfect) matchings in bipartite graphs.

Given a bipartite graph $G = (X \cup Y, E)$, we construct a flow network G' as follows.

- Add a source s.
- Add a sink t.
- Add (s, x) edges for all $x \in X$.
- Add (y, t) edges for all $y \in Y$.
- Add capacity 1 to every edge.
- Direct all $e \in E$ from X to Y.

For example, Figure 3(b) shows the flow network derived from the bipartite graph in Figure 3(a).

We now compute a max s-t flow in G'. We claim that the value of the max flow in G' equals the size of the max matching in G.

Claim 1 *The size of the max matching in G equals the value of the max flow in G'. The edges of the matching are the edges that carry flow from X to Y in G'.*

Proof. We will show how, given any matching in G, we may construct a flow in G' with value equal to the size of the matching, and the reverse.

- First, let M be a matching with k edges. Then we may send one unit of flow along each of the k edge-disjoint s-t paths that use these k edges, and obtain an s-t flow of value k.
- Next suppose we have a flow f' in G' with $|f| = k$. We need to show how to construct a matching M' from f. To this end, we need show two things: that M' is indeed a matching (each vertex in $X \cup Y$ appears at most once in M') and that it has size k.

  By the integrality theorem, there is an integer-valued flow f of value k. Then for every edge e, $f(e) = 0$ or $f(e) = 1$ (*why?*). So consider the set of edges of the form (x, y) on which the flow value is 1: these will form our matching M'.

  We want to show that $|M'| = k$. Consider the cut (S, T) where $S = \{s\} \cup X$, $T = \{t\} \cup Y$. By Lemma 1 we know that the net flow across (S, T) equals $|f|$, so the net flow across (S, T) equals k. We can also compute the net flow of (S, T) using the definition in equation 2: it equals $|M'|$, since the flow out of S equals $|M'|$ (these are the only edges that carry flow), while the flow into S equals 0 (no edges enter S). Thus $|M'| = k$.

  Now observe that each node in X is the tail of at most one edge in M': otherwise, if two edges in M' leave the same $x \in X$, they would each carry 1 unit of flow. But there is only 1 unit of flow coming into x, on edge (s, x), so conservation of flow would be violated. A similar argument shows that each node in Y is the head of at most one edge in M'.

The claim follows. □

The simple implementation of the Ford-Fulkerson algorithm yields a running time of $O(mn)$ for finding a maximum matching in a bipartite graph (*why?*).

4.1 Hall's theorem

Suppose that we are given a bipartite graph $G = (X \cup Y, E)$ such that $|X| = |Y| = n$. Obviously, it is not the case that every such bipartite graph has a perfect matching (e.g., see Figure 3(a)).

In particular, if we construct G' as in the previous section, run Ford-Fulkerson on G' and compute a max flow with value less than n, then we know that G has no perfect matching.

We are now concerned with the following question: we are looking for an easy way to be convinced that a bipartite graph has no perfect matching **after** we've run the algorithm. In other words, we are looking for a **succinct certificate** of non-existence of a perfect matching. Such a certificate should be short and, given the certificate, we should be quickly convinced that no perfect matching exists in G without having to review the entire execution of the algorithm.

Let A be a subset of X and $N(A)$ the neighbors of A in Y. A necessary condition for a perfect matching to exist is that for all $A \subseteq X$, $|N(A)| \geq |A|$ (*why?*). Then such a set A with $|N(A)| < |A|$ is a succinct certificate of non-existence of a perfect matching.

We will now show that the condition above is necessary and sufficient for a bipartite graph to have a perfect matching. To this end, we will provide an algorithm that, on input a bipartite graph with $|X| = |Y| = n$, either finds a perfect matching, or fails because it identified a set $A \subseteq X$ such that $|N(A)| < |A|$. This is summarized in the following theorem.

Theorem 4 (Hall's theorem) *Let $G = (X \cup Y, E)$ with $|X| = |Y| = n$. Then G either has a perfect matching or there is a subset $A \subseteq X$ such that $|N(A)| < |A|$. A perfect matching or such a subset may be found in $O(mn)$ time.*

Proof. First we construct G' as before and run Ford-Fulkerson in G'.

1. If $|f| = n$, then output edges $e = (x, y)$ such that $f(e) = 1$: these edges form the perfect matching.
2. If $|f| < n$, for (S^*, T^*) defined as in Section 3.2, output $A = S^* \cap X$; this set of nodes satisfies $|A| > |N(A)|$.

Correctness of the first case follows from our analysis in the previous section. We will now show that, if $|f| < n$, then the set A defined as above indeed satisfies $|A| > |N(A)|$.

By the max flow min cut theorem, G' has an s-t cut of capacity at most $n-1$ (this is its minimum capacity cut), call is (S^*, T^*). Since (S^*, T^*) is an s-t cut, S^* will contain s and potentially nodes from both X and Y (*why?*), while T^* will contain t (and potentially nodes from both X and Y as well).

Now consider the set A of nodes in X that also belong to S^*, that is $A = X \cap S^*$. (This set will be our certificate of non-existence of a perfect matching.) First, update S^* to include all neighbors of the set A in Y. We claim that the capacity of the updated cut cannot increase by doing so (and we do so because it will give us the means to relate A and $N(A)$). Our reasoning is as follows. Let y be a neighbor of some node in A such that y is not in S^*. If we add y to S^*, then we add a new outgoing edge from S^* to t but since y has at least one incoming edge from A, the capacity of the updated cut cannot increase (see Figure 4). (*What about nodes in X that are not in S^* and have outgoing edges to y? Do these affect the capacity of the updated (S^*, T^*)?*)

So now S^* contains the set A and all its neighbors $N(A)$ in Y, and its capacity is still at most $n - 1$. We will now compute the capacity of the updated cut using the definition of the capacity of a cut, that is, $c(S, T) = \sum_{e \text{ out of } S} c(e)$. The only edges out of the updated S^* are:

- Edges from the source s to $X - A$. There are $n - |A|$ such edges.
- Edges from the nodes in Y that belong to S^* (that is, the nodes in $Y \cap S^*$) to sink t. These nodes contain the set $N(A)$ so, say, there are $\alpha \geq |N(A)|$ such edges.

So now we have

$$c(S^*, T^*) \leq n - 1 \Rightarrow n - |A| + \alpha < n \Rightarrow \alpha < |A| \Rightarrow |N(A)| \leq \alpha < |A|.$$

Finally, A may be computed in $O(mn)$ time (*why?*).

□

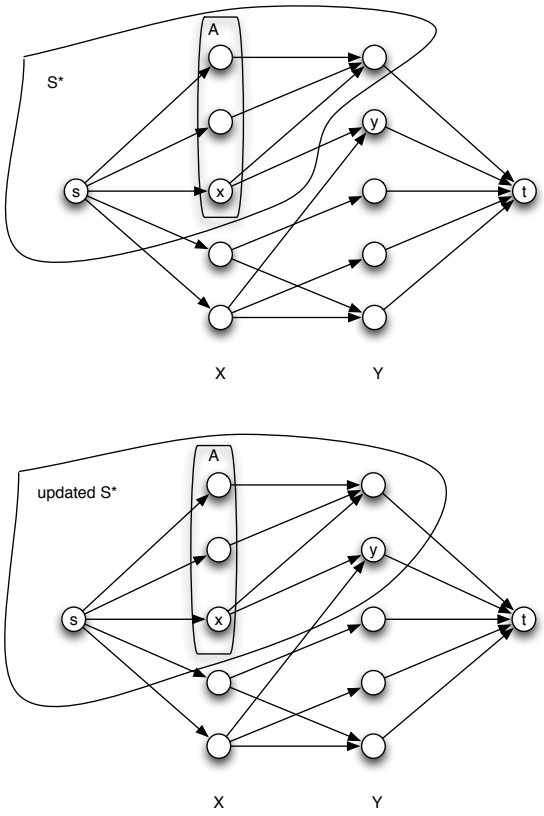


Figure 4: The updated set S^* contains all neighbors of A.