Applications of Maximum Flow

We have studied the maximum flow problem, and discussed various algorithms for computing a maximum flow in a network. We have already discussed that many different situations can be modelled as a flow network, such as fluid in pipes, currents in a circuit, or traffic in a computer network to name a few. As it turns out, there are also a surprising number of seemingly unrelated problems, that can be **reduced** into a maximum flow problem. Thus, efficient max-flow algorithms can be used to solve these problems. We look at some of such problems in this module.

Min Cut Problem

Given a flow network $G = (V, E)$, we wish to find a partition of V, (S, T), having minimum capacity, i.e.

$$\min_{\text{cut } (S,T)} \sum_{\substack{u \in S, v \in T \\ (u,v) \in E}} c(u, v)$$

We have already discussed previously, that we can find a minimum cut by first finding a max-flow f of G, computing its residual graph G_f, and then finding the set $S = \{v \in V \mid v \text{ is reachable from } s \text{ in } G_f\}$ by doing a BFS from s. Then, $\{S, V \setminus S\}$ is a minimum cut.

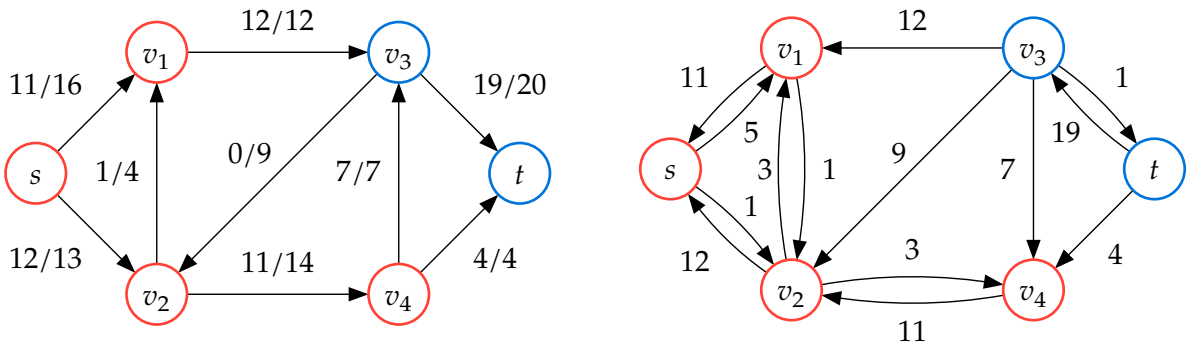


Figure 1: A flow network G with a max flow f. $S = \{s, v_1, v_2, v_4\}$. $(S, V \setminus S)$ is a min-cut of G.

Observe, that instead of giving an algorithm for the Min-Cut problem, we instead showed a procedure to solve a Min-Cut problem, provided that we have some way to solve an instance of the Max-Flow problem. In other words, we have shown that, if we have a subroutine to find a maximum flow in a flow network, we can use it to find a minimum cut in the flow network. We say, that **the minimum cut problem *reduces* in linear time to the maximum flow problem**.

Such a conversion of an instance of one problem X into an instance of another problem Y, such that the solution of the instance of problem Y can be used to get a solution of the instance of the problem X is called a **reduction**. In this case, since this conversion can be done in linear time, this is a **linear time reduction** (which is in fact a **polynomial time reduction**).

Maximum Bipartite Matching

Now we look at a completely different problem (seemingly unrelated to network flows), which is the bipartite matching problem. Before presenting the problem, let us define a few terms:

Given a graph $G = (V, E)$, a **matching** M in G is a subset of edges, such that no two edges share any common vertex.

A graph $G = (V, E)$ is said to be **bipartite**, if there exists a partition of V into A and B, $A \uplus B = V$ ($A \uplus B$ denotes a disjoint union, i.e. $A \cup B$ where A and B are disjoint sets.), such that all $e \in E$ are between $v_1 \in A$ and $v_2 \in B$. No edges connect vertices (v_1, v_2) such that $v_1, v_2 \in A$ or $v_1, v_2 \in B$.

The **maximum bipartite matching** problem: Given a bipartite graph $G = (V = A \uplus B, E)$ with $|A| = |B|$, find a maximum-cardinality matching in G.

A **perfect matching** in a graph G is a matching which covers every vertex of the graph. In a bipartite graph $G = (V = A \uplus B, E)$ where $|A| = |B| = n$, if a perfect matching exists, the cardinality of a perfect matching is n.

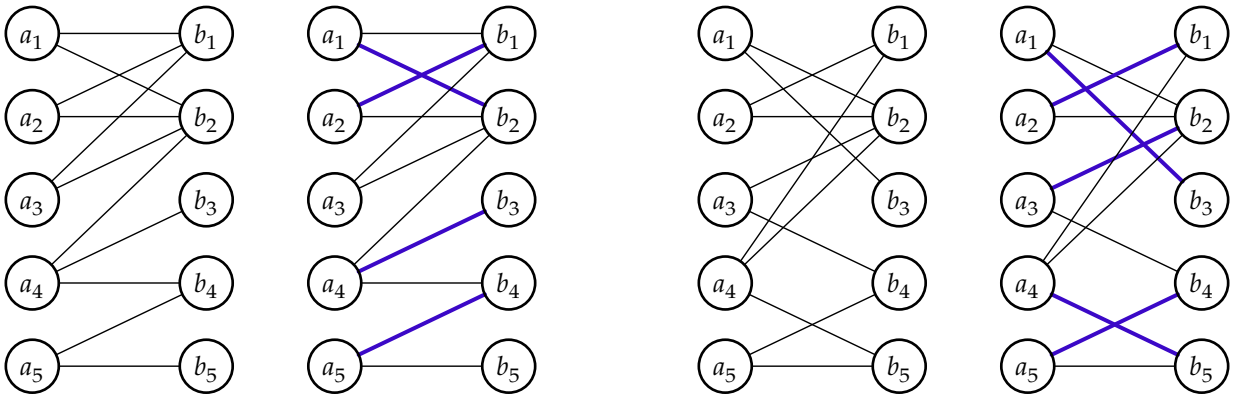


Figure 2: Bipartite graphs G_1 and G_2, and maximal matchings M_1 and M_2 in them. $|M_1| = 4$, $|M_2| = 5$. G_1 does not have any perfect matching. G_2 has a perfect matching. (M_2 is one.)

Reduction to Max-Flow

We will show that the maximum-cardinality matching problem on bipartite graphs *reduces* in linear time to the maximum flow problem on flow networks.

To find a maximum-cardinality matching on a bipartite graph $G = (V = A \uplus B, E)$, we first define a directed graph $G' = (V', E')$, $V' = V \cup \{s, t\}$, $E' = \{(u, v) \mid (u, v) \in E, u \in A, v \in B\} \cup \{(s, a) \mid a \in A\} \cup \{(b, t) \mid b \in B\}$. Essentially, we add a source and a sink node, direct all edges from A to B, and add edges from the source to vertices in A, and from vertices in B to the sink. We give capacity 1 to all edges connected to s or t, and capacity ∞ to all other edges (ones that are between A and B).

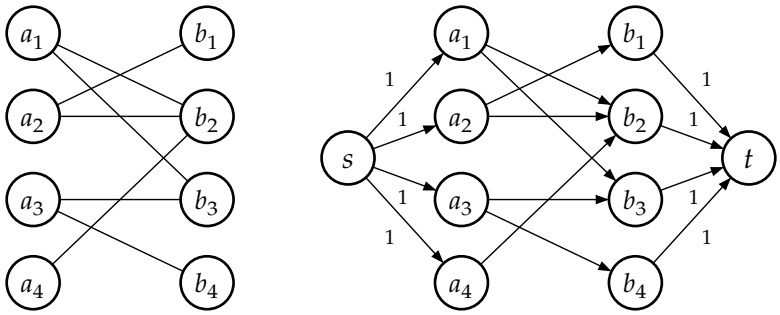


Figure 3: A bipartite graph G and its corresponding flow network G' as per our construction. Unlabelled edges have a capacity of ∞.

We find a integral maximum flow f on G'.[1] Once we have this flow, choose $M = \{(u,v) \in E \mid f(u,v) = 1\}$, i.e. select the edges from G whose corresponding edges in G' have positive flow in f. Then, M is a maximum-cardinality matching of G.

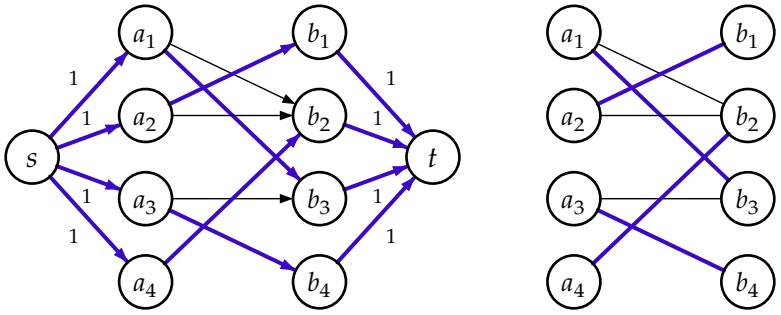


Figure 4: (a) Maximum flow of G'. Highlighted edges carry flow of 1. (b) A maximum cardinality matching M of G.

As always, we have to show that it works!

> **Maximum-cardinality bipartite matching reduces to Max-flow**
>
> Let $G = (A \uplus B, E)$ be a bipartite graph. Let $G' = (V', E')$ be the corresponding flow network constructed from G. The maximum cardinality of a matching in G equals the maximum value of an $s - t$ flow in G'.

Lemma 1: If G has a matching M of size k, then there exists a flow f in G' with $|f| = k$.

Proof: Let M be a matching of G, with $|M| = k$. Let $M = \{(u_1, v_1), ..., (u_k, v_k)\}$. We can construct a flow f in G':

- $f(s, u_i) = 1$ for all $i = 1, 2, ..., k$
- $f(v_i, t) = 1$ for all $i = 1, 2, ..., k$
- $f(u_i, v_i) = 1$ for all $i = 1, 2, ..., k$
- For all other edges $(u,v) \in E', f(u,v) = 0$

Clearly, capacity constraints are satisfied. We need to show flow conservation:

- For some node $a \in A$,
  - if $a = u_i$ for some i, then its only incoming edge is (s, u_i) which has flow 1. One of its outgoing edges is (u_i, v_i), which has flow 1. All other outgoing edges from a have 0 flow since they don't exist in M (by definition of a matching).
  - otherwise, its only incoming edge (s, a) has 0 flow. All its outgoing edges also have zero flow, since they don't exist in M.
- For some node $b \in B$,
  - if $b = v_i$ for some i, then its only outgoing edge is (v_i, t) which has flow 1. One of its incoming edges is (u_i, v_i) which has flow 1. All other incoming edges to b have 0 flow since they don't exist in M.
  - otherwise, its only outgoing edge (b, t) has 0 flow. All its incoming edges also have zero flow, since they don't exist in M.

[1] All the max-flow algorithms we have discussed will necessarily find an integral max flow on a network with integral edge capacities. (In an integral flow, flow values on all edges are integral, not just the net value of the flow.)

Hence, f is a valid flow.

Value of the flow is the total outflow from s:

$$|f| = f(s, V - s) = \sum_{a \in A} f(s, a) = \sum_{i=1}^{k} f(s, u_i) + \sum_{\substack{a \in A \\ a \neq u_i}} f(s, a) = k + 0 = k$$

Lemma 2: If G' has an integral flow f with value $|f| = k$, then there exists a matching M in G with cardinality k.

Proof: Let G' have an integral flow f with value $|f| = k$. Define a matching M as follows:

$$M = \{(u, v) \in E \mid u \in A, v \in B, f(u, v) = 1\}$$

First let us show that flow in any edge in f will be either 0 or 1. Suppose there is some edge $(u, v), f(u, v) \geq 2$. Clearly, $u \in A, v \in B$. (any other type of edge does not have capacity to hold flow ≥ 2.) Then, the outgoing flow from u is ≥ 2. The only incoming edge to u is (s, u) which has a capacity of 1, which is a contradiction, since flow cannot be conserved in this case. So, $f(u, v) \in \{0, 1\}$ for all $(u, v) \in E'$.

To show M is a matching, we need to show that no vertex appears in more than one edge $e \in M$.

- Let $a \in A$. Suppose there are two edges (a, b_1) and (a, b_2) in M. This means, in $f, f(a, b_1) = 1, f(a, b_2) = 1$. So, total flow exiting a is ≥ 2. The only incoming edge to a is (s, a), which has a capacity of 1, which is a contradiction, since flow on a isn't conserved.
- Let $b \in B$ appear in more than one edge in M. Similarly, we can arrive at a contradiction.

Thus, $M \subseteq E$ is a valid matching.

Consider the cut $(s \cup A, B \cup t)$ in G'. Clearly, the flow through this cut is $|f| = k$. Then,

$$\begin{aligned} f(s \cup A, B \cup t) &= |f| = k \\ \Rightarrow f(A, B) &= k \\ \Rightarrow \sum_{u \in A, v \in B} f(u, v) &= k \end{aligned}$$

Since the flow in any edge can only be 0 or 1, the value of the last summation is the count of the edges (u, v), $u \in A, v \in B$, for which $f(u, v) = 1$, Therefore, M has k edges $(|M| = k)$. ■

The Theorem follows from these two lemmas.

Perfect Matching

Given a bipartite graph $G = (A \uplus B, E)$, $|A| = |B| = n$, does G contain a perfect matching?

We can find this just by the above procedure, find the maximum cardinality matching in G, check if the cardinality of the matching is n, if so, then it is a perfect matching. If not, then no perfect matching exists. (Why?)

In other words, the decision problem "Does G have a perfect matching?", is equivalent to checking if the value of the max flow of G' is n.

Observe that in Figure 3/4, G has a perfect matching.

Hall's Theorem

Here, we look at a characterisation of the above decision problem, i.e. Does G have a perfect matching?

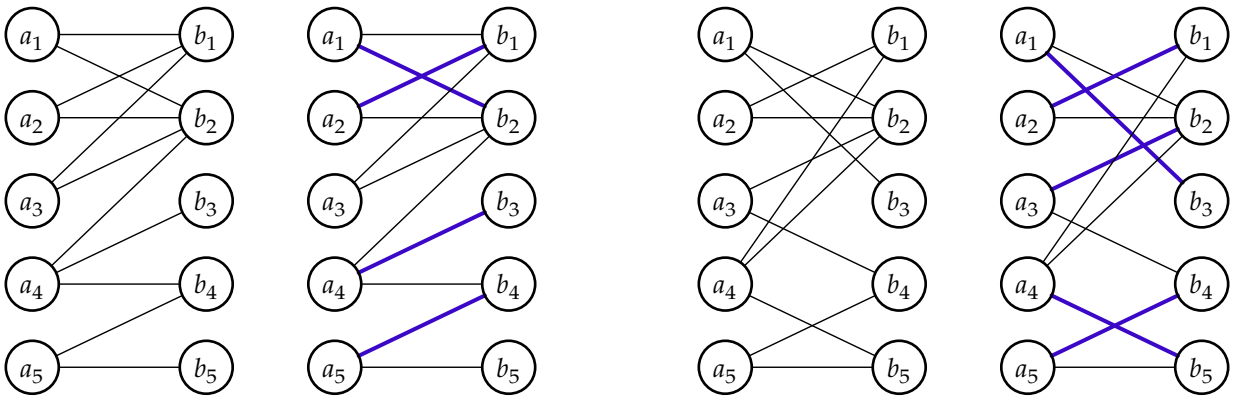


Figure 5: Graphs G_1 and G_2, and their maximal matchings. (Same as Figure 2, reproduced here for the sake of your PDF scrolling sanity.)

Take a look at G_1 and G_2 in Figure 5. G_2 has a perfect matching, but G_1 does not. Could we give a succinct argument for why G_1 does not have any perfect matching, other than computing it via max-flow?

In the matching M_1, a_3 is not matched with any vertex in B. If we did match a_3 with some vertex, say b_2, then a_1 would become unmatched. The only other vertex that a_1 has an edge to is b_1, say we match these. Now, a_2 becomes unmatched, and the only other vertex a_2 has an edge to is b_2, but we have already matched it with a_3. So it seems like we can never match all of $\{a_1, a_2, a_3\}$ in graph G_1, however we try, one of these vertices gets left out.

Can we formalise this argument?

For some subset of vertices $S \subseteq A$, define the **neighbour set** of S: $N(S) = \{b \in B \mid \exists a \in S \text{ s.t. } (a, b) \in E\}$. So, $N(S)$ contains all vertices in B that has some edge with a vertex in S.

In G_1, let $S = \{a_1, a_2, a_3\}$. Then, $N(S) = \{b_1, b_2\}$. Now, $|N(S)| < |S|$, which means the size of the neighbour set is smaller than the size of S itself. From this, it is very clear, that no perfect matching can exist in G. Because if it did, then the matched vertices of a_1, a_2, a_3 would be distinct, and they would be in $N(S)$, which clearly there aren't enough vertices in $N(S)$ for that to happen.

In general, in a graph G, if we find some $S \subseteq A$ such that $|N(S)| < |S|$, then G cannot have any perfect matching. In other words, for G to have a perfect matching, a necessary condition is: for all $S \subseteq A$, $|N(S)| \geq |S|$.

Hall's theorem states that this condition is a sufficient condition as well for existence of a perfect matching.

> **Hall's Theorem**
>
> A bipartite graph $G = (V = A \uplus B, E)$ with $|A| = |B| = n$ has a perfect matching if and only if $|S| \leq |N(S)|$ for every $S \subseteq A$.

Proof: [$\Rightarrow$]: Trivial.

[$\Leftarrow$]: Suppose the cardinality of a maximum matching in G is k. Construct the flow network G' from G by adding s and t, directing edges to go from A to B (all with capacity ∞), and adding

edges from s to $a \in A$, and from $b \in B$ to t (all with capacity 1). We know that the value of the max flow of G' is k.

By the max-flow min-cut theorem, there exists some cut (U, W) such that $c(U, W) = |f| = k$.

Let $A_1 = U \cap A, A_2 = A \setminus A_1$; $B_1 = U \cap B, B_2 = B \setminus B_1$. (Note that some of these sets may be empty.)

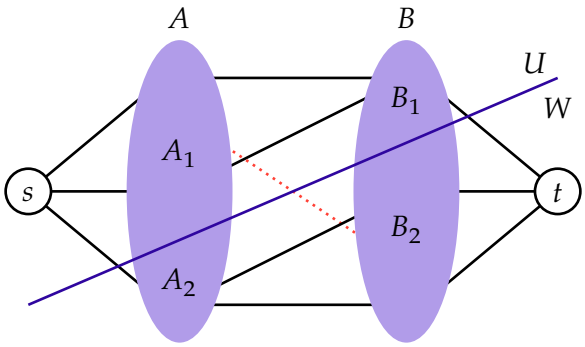


Figure 6: Defining A_1, A_2, B_1, B_2 by the min cut (U, W). The red dotted edge cannot exist.

Since this is a cut with finite capacity, no edge having ∞ capacity must cross the cut. This means, for all $a \in A_1$, if (a, b) is an edge, then $b \notin B_2, b \in B_1$. So, $N(A_1) \subseteq B_1$. (Otherwise there will be an ∞ edge crossing the cut.)

Now, calculate the capacity of the cut (U, W):

$$\begin{aligned} k = c(U, W) &= c(s \cup A_1 \cup B_1, A_2 \cup B_2 \cup t) \\ &= c(s, A_2) + c(A_1, B_2) + c(B_1, t) \\ &= |A_2| + 0 + |B_1| \\ &= n - |A_1| + |B_1| \\ &\geq n - |A_1| + |N(A_1)| && \text{since } N(A_1) \subseteq B_1 \\ &\geq n - |A_1| + |A_1| && \text{since } |A_1| \leq |N(A_1)| \\ &= n \end{aligned}$$

So, $k \geq n$, but we know that max-cardinality of a matching can't be $> n$, thus $k = n$. Hence, G has a perfect matching. ■

Kőnig's Theorem

This is another theorem that characterises the matching problem on bipartite graphs, into another problem, the **vertex cover problem**.

A **vertex cover** of a graph $G = (V, E)$ is a set of vertices $C \subseteq V$, such that for all edges $e \in E$, at least one end point of e must be in C. A **minimum vertex cover** of G is a minimum-cardinality vertex cover of G.

In a graph G, let M be a matching. Consider a vertex cover C of G. For each edge $e \in M$, at least one end point of that edge must be present in C (by definition of vertex cover). All endpoints of vertices in M are distinct. So, we have $|C| \geq |M|$, i.e. **the size of a maximum matching is a lower bound on the size of the minimum vertex cover of** G. Kőnig's theorem states that this lower bound is always achievable in bipartite graphs.

Kőnig's Theorem

The size of a minimum vertex cover is the same as the size of a maximum cardinality matching in every bipartite graph.

Proof: Let $G = (V = A \uplus B, E)$ be a bipartite graph. Let the cardinality of a maximum matching be k. Construct the flow network G' as usual, let (U, W) be its min cut, so $c(U, W) = k$.

Define $A_1 = U \cap A, A_2 = A \setminus A_1; B_1 = U \cap B, B_2 = B \setminus B_1$.

Again, since (U, W) is a cut with finite capacity, no edge in G must be from A_1 to B_2. So, $N(A_1) \subseteq B_1$.

Here, if we calculate the capacity of the cut (U, W), we have

$$\begin{aligned} k = c(U, W) &= c(s \cup A_1 \cup B_1, A_2 \cup B_2 \cup t) \\ &= c(s, A_2) + c(A_1, B_2) + c(B_1, t) \\ &= |A_2| + |B_1| \end{aligned}$$

So, $k = |A_2| + |B_1|$.

We claim, that $C = A_2 \cup B_1$ is a vertex cover of G.

Let us check all edges to see if one of their endpoints is in C.

For all $(u, v) \in E$,
- If $u \in A_1$, then $v \in N(A_1) \Rightarrow v \in B_1 \Rightarrow v \in C$.
- Else, $u \in A_2 \Rightarrow u \in C$.

So C is indeed a vertex cover of G. The size of this vertex cover is $|C| = |A_2| + |B_1| = k$, which is the size of the maximum cardinality matching of G. Since the size of a matching is a lower bound on the size of a vertex cover, C is a minimum vertex cover of G. ■

Summary

We have looked at some applications of the max-flow problem, how other problems can be reduced to the max flow problem, which gives us efficient algorithms to solve them. We have also seen some uses of flow networks to prove some theorems related to matching in bipartite graphs. Next up, we will look at the matching problem in general graphs.

Problems

1. Using max flow, prove Menger's theorem: If an undirected graph remains connected after removing any set of fewer than k edges, then the size of the minimum cut is at least k. A cut is a set of edges whose removal makes the graph disconnected.

Firstly, if an undirected graph remains connected after removing any set of fewer than k edges, then the minimum number of edges that need to be removed to disconnect the graph is at least k. There is nothing to prove here, it is self evident. Nor is this the statement of Menger's theorem.

(One version of) Menger's theorem states: **If an undirected graph remains connected after removing any set of fewer than k edges, then there exist at least k disjoint paths between any pair of vertices.**

- Two paths are disjoint if they do not share any edge.

Let G be an undirected graph, such that on removal of any set of fewer than k edges, G remains connected. Consider any two arbitrary vertices of G, call them s and t. Construct a flow network G' by making all edges directed (along both directions), and all edge capacities 1, taking s as the source and t as the sink. Consider the size (number of edges) of the minimum $s-t$ cut in G'. It cannot be less than k, since if there is a cut with less than k edges, by removing these edges from G we can disconnect it, which is a contradiction. So, the size of the minimum $s-t$ cut in G' must be $\geq k$.

By the max-flow min-cut theorem, the value of the max flow in G' should be at least k. Consider an integral max flow f in G'. WLOG, assume f is acyclic. We can decompose f into exactly $|f|$ path flows from s to t, each with value 1 (since no path flow can have a value more than 1). Also, no two of these path flows may share any edge, i.e. they must be disjoint. This is guaranteed by the fact that each edge has a capacity of 1, so one edge cannot be part of two path flows of value 1. Thus, this gives us at least k disjoint paths in G between s and t.

Since s and t were chosen arbitrarily, this means we can get at least k disjoint paths between any pair of vertices in G.

2. [KT book] Network flow issues come up in dealing with natural disasters and other crises, since major unexpected events often require the movement and evacuation of large numbers of people in a short amount of time.

   Consider the following scenario. Due to large-scale flooding in a region, paramedics have identified a set of n injured people distributed across the region who need to be rushed to hospitals. There are k hospitals in the region, and each of the n people needs to be brought to a hospital that is within a half-hour's driving time of their current location (so different people will have different options for hospitals, depending on where they are right now).

   At the same time, one doesn't want to overload any one of the hospitals by sending too many patients its way. The paramedics are in touch by cell phone, and they want to collectively work out whether they can choose a hospital for each of the injured people in such a way that the load on the hospitals is *balanced*: Each hospital receives at most $\lceil n/k \rceil$ people.

   Give a polynomial-time algorithm that takes the given information about the people's locations and determines whether this is possible.

First we abstract out the input information: We just need to know, for each of the n people, what are the hospitals that they can visit. Compute this information beforehand. We need to find out if there exists a pairing of each person with one hospital (that they can visit), such that the total number of people visiting a hospital is at most $\left\lceil \frac{n}{k} \right\rceil$.

Construct a bipartite graph $G = (V = (A \uplus B), E)$, with $A = \{a_1, ..., a_n\}$ representing n people, and $B = \{b_1, ..., b_k\}$ representing k hospitals. Add an edge between $\left(a_i, b_j\right)$ if person i can reach hospital j in time.

Construct a flow network G' by adding a source s, a sink t, and edges from (s, a_i) with capacity 1, and $\left(b_j, t\right)$ with capacity $\left\lceil \frac{n}{k} \right\rceil$. Direct all existing edges $\left(a_i, b_j\right)$ to be from A to B, with capacity ∞. Find the value of the max-flow on G' (by using any suitable polynomial time algorithm). If the value of the max-flow is n, then the answer is YES, otherwise NO.

Proof: If a integral max-flow of value n exists, then this means for each a_i an outgoing edge has flow $= 1$, which means each person is matched with one hospital (among the ones it can visit). Also, since outgoing capacity of each hospital is $\left\lceil \frac{n}{k} \right\rceil$, this means that each hospital needs to cater to at most $\left\lceil \frac{n}{k} \right\rceil$ people.

Alternatively: Let $p = \left\lceil \frac{n}{k} \right\rceil$. Construct a bipartite graph $G = (V = (A \uplus B), E)$ with $A = \{a_1, ..., a_n\}$ representing n people, and $B = \left\{b_{1,1}, b_{1,2}, ..., b_{1,p}, b_{2,1}, b_{2,2}, ..., b_{2,p}, b_{k,1}, b_{k,2}, ..., b_{k,p}\right\}$, i.e. p nodes for each of the k hospitals. An edge exists between $\left(a_i, b_{j,l}\right)$ if person i can reach hospital j in time (for all l). If a matching of size n exists in G, then the answer is YES, otherwise NO.

3. [KT book] Your friends have written a very fast piece of maximum-flow code based on repeatedly finding augmenting paths. However, after you've looked at a bit of output from it, you realize that it's not always finding a flow of *maximum* value. The bug turns out to be pretty easy to find; your friends hadn't really gotten into the whole backward-edge thing when writing the code, and so their implementation builds a variant of the residual graph that *only includes the forward edges*. In other words, it searches for $s - t$ paths in a graph G_f consisting only of edges e for which $f(e) < c(e)$, and it terminates when there is no augmenting path consisting entirely of such edges. We'll call this the Forward-Edge-Only Algorithm. (Note that we do not try to prescribe how this algorithm chooses its forward-edge paths; it may choose them in any fashion it wants, provided that it terminates only when there are no forward-edge paths.)

   It's hard to convince your friends they need to reimplement the code. In addition to its blazing speed, they claim, in fact, that it never returns a flow whose value is less than a fixed fraction of optimal. Do you believe this? The crux of their claim can be made precise in the following statement.

   There is an absolute constant $b > 1$ (independent of the particular input flow network), so that on every instance of the Maximum-Flow Problem, the Forward-Edge-Only Algorithm is guaranteed to find a flow of value at least $\frac{1}{b}$ times the maximum-flow value (regardless of how it chooses its forward-edge paths).

   Decide whether you think this statement is true or false, and give a proof of either the statement or its negation.

Discussion: Essentially, the Forward-Edge-Only algorithm finds a blocking flow f on the graph G. (Recall, that a blocking flow is one where every s to t path in G has at least one edge saturated. The Forward-Edge-Only algorithm continuously chooses some $s - t$ path and saturates its critical edge, until no $s - t$ path (with all non-saturated edges) exists. We do not add backward edges, so we are only concerned with $s - t$ paths that exist in G itself.)

Intuitively, it feels that the answer should be no, i.e. we should be able to construct some network by which we can make the ratio of the flow values to be as small as possible, i.e. some blocking flow exists whose value is very small as compared to the optimal max flow's value.

We may start with constructing a graph where the Forward-Edge-Only algorithm might reach a blocking flow which isn't a max-flow.

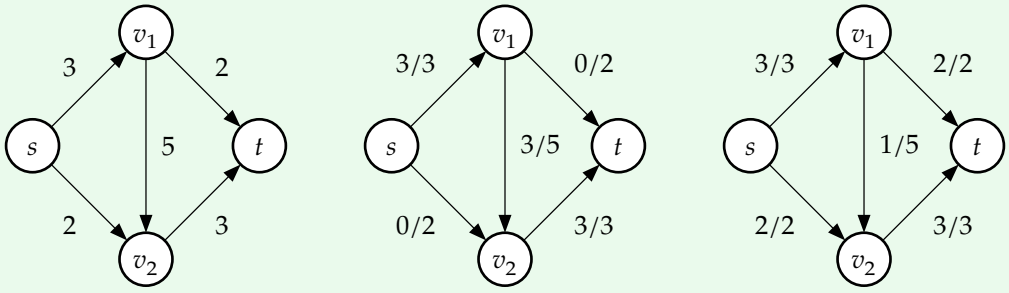
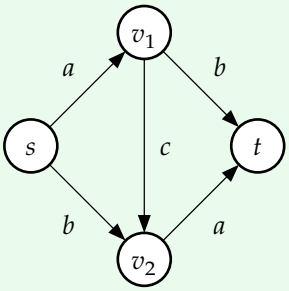


Figure 7: (a) A network G. (b) A possible flow f obtained from Forward-Edges-Only, with value 3. (c) A max flow f^* with value 5.

Consider the network G in Figure 7. Suppose it chooses the path $s \to v_1 \to v_2 \to t$ on its first iteration, then it augments its *residual graph* (without backward edges) by the residual flow $c_f = 3$. Now, this path flow is a blocking flow in G, and it cannot proceed further in finding any more $s - t$ paths with all edges having positive residual capacity. So we have gotten a flow f of value 3. (Observe, that if we maintained a correct residual graph which had backward edges, then we do have a $s - t$ path: $s \to v_2 \to v_1 \to t$, augmenting by which we would arrive at the max-flow f^*, with value 5).

The next thought is that just by tweaking the capacities of edges in this network, can we make the value of $|f|$ / $|f^*|$ arbitrarily small? If so, then we are done. Let's generalise the edge capacities here and analyse.



Let $a \geq b$. For the max flow of this network to be $a + b$ (which is the maximum possible), we need to route $a - b$ flow through the (v_1, v_2) edge. So we need

$$c \geq a - b$$

Also, lets say we want $s \to v_1 \to v_2 \to t$ to be a blocking flow, then it should block the edges (s, v_1), and (v_2, t). For that, we will need

$$c \geq a$$

So, we have, $|f^*| = a + b \leq 2a$, and $|f| = a$. Then,

$$\frac{|f|}{|f^*|} = \frac{a}{a+b} \geq \frac{1}{2}$$

So our approach doesn't work: On this graph, just by changing the edge capacities of the network, the Forward-Edges-Only algorithm will always output a flow whose value is at least $\frac{1}{2}|f^*|$ or better.

But this gives another idea. Of course we can make our graph however we want. Maybe repeating a structure like this arbitrarily can decrease the lower bound of the ratio arbitrarily. So the high level idea is: For some given k, construct a network G, which has one blocking flow with (ideally) constant flow value, and its max flow value is linear in k. This suffices to prove what we need.

Solution:

The statement is false.

Suppose for contradiction, there does exist some $b > 1$, such that for any flow network G, the Forward-Edges-Only algorithm is guaranteed to find a flow of value at least $\frac{1}{b}$ times the max-flow value.

Let $k \in \mathbb{Z}_{>0}$. Define a network $G_k = (V_k, E_k)$ as follows:

- $V_k = \{s, t\} \cup \{a_1, ..., a_k\} \cup \{b_1, ..., b_k\}$
- $E_k = \{(s, a_1), (b_1, a_2), (b_2, a_3), ..., (b_{k-1}, a_k), (b_k, t)\} \cup \{(a_1, b_1), (a_2, b_2), ..., (a_k, b_k)\} \cup \{(s, b_i) \mid 1 \leq i \leq k\} \cup \{(a_i, t) \mid 1 \leq i \leq k\}$, all edges have capacity 1.

(In order to make the flow linear in k we add $O(k)$ nodes, and allow flow to pass through each node from s to t to get a max flow. To get a constant value blocking flow we use a similar idea as in the previous example, and keep a path from s to t, upon saturation of which, no other $s - t$ path exists.)

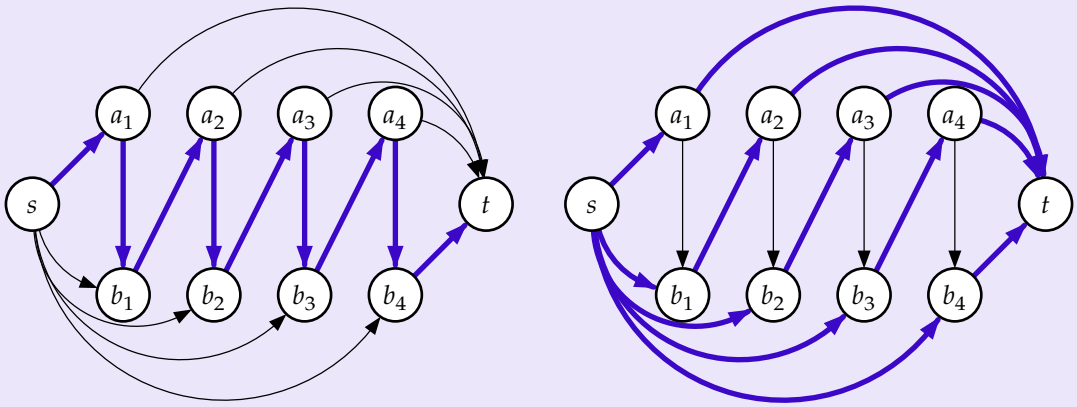


Figure 8: Construction of G_k for $k = 4$. All edge capacities are 1. (a) The highlighted edges form a blocking flow f, $|f| = 1$. (b) The highlighted edges form a max flow f^*, $|f^*| = 5$.

Suppose the Forward-Edges-Only algorithm pickes the path: $s \to a_1 \to b_1 \to ... \to a_k \to b_k \to t$. Then, since all edges in this path have capacity 1, all of them get saturated, so all these edges get removed from the residual graph. As no backward edges are added, no other $s - t$ path exists and the algorithm terminates here, with a flow f having value of $|f| = 1$.

We can clearly show that the value of the max flow is $k + 1$: By saturating these paths: $s \to a_1 \to t, s \to b_1 \to a_2 \to t$, ..., $s \to b_k \to t$, we can get a flow f^* with value $|f^*| = k + 1$. This is a max flow because the capacity of the $(s, V - s)$ cut is also $k + 1$.

Let $k = \lceil b \rceil$. Construct G_k as shown. Suppose the forward edges algorithm chooses the path $s \to a_1 \to b_1 \to ... \to a_k \to b_k \to t$. Then, once it augments by this path, there are no other $s - t$ path which only use forward residual edges, so the algorithm terminates with this flow f having value 1.

We have shown that G_k has a maximum flow value of $|f^*| = k + 1$.

By our hypothesis, we know that the ratio of the $|f|$ and $|f^*|$ must be at least $\frac{1}{b}$. So,

$$\frac{|f|}{|f^*|} \geq \frac{1}{b}$$
$$\Rightarrow \frac{1}{k+1} \geq \frac{1}{b}$$
$$\Rightarrow b \geq k + 1 > b$$

which is a contradiction. Hence, there does not exist any such absolute constant b.

4. [KT book] We define the *Escape Problem* as follows. We are given a directed graph $G = (V, E)$ (picture a network of roads). A certain collection of nodes $X \subseteq V$ are designated as *populated nodes*, and a certain other collection $S \subseteq V$ are designated as *safe nodes*. (Assume that X and S are disjoint.) In case of an emergency, we want evacuation routes from the populated nodes to the safe nodes. A set of evacuation routes is defined as a set of paths in G so that (i) each

node in X is the start of one path, (ii) the last node on each path lies in S, and (iii) the paths do not share any edges. Such a set of paths gives a way for the occupants of the populated nodes to "escape" to S, without overly congesting any edge in G.

(a) Given G, X, and S, show how to decide in polynomial time whether such a set of evacuation routes exists.

(b) Suppose we have exactly the same problem as in (a), but we want to enforce an even stronger version of the "no congestion" condition (iii). Thus we change (iii) to say "the paths do not share any nodes." With this new condition, show how to decide in polynomial time whether such a set of evacuation routes exists. Also, provide an example with the same G, X, and S, in which the answer is yes to the question in (a) but no to the question in (b).

(a) Construct a graph $G' = (V \cup \{s,t\}, E')$, $E' = E \cup \{(s,x) \mid x \in X\} \cup \{(v,t) \mid v \in S\}$. (Add a source s and sink t, add edges from s to all vertices in X, and from all vertices in S to t). The original edges in E have a capacity 1. Newly added edges from s have a capacity of 1, while newly added edges to t have a capacity ∞.

Run a polynomial time max-flow algorithm on this graph. If the max-flow value is equal to $|X|$, then such a set of evacuation routes exists, otherwise they don't.

Proof:

1. If such a set of escape routes exist in G, then the max flow value is $|X|$ in G'.

   Assume there is a valid set of $|X|$ escape routes, one starting at each vertex in X. We can construct a flow f by setting $f(e) = 1$ if e is used in any of these routes, $f(e) = 0$ otherwise, and set $x_i \in X, f(s, x_i) = 1$ and $s_j \in S, f\left(s_j, t\right) =$ sum of incoming flow to s_j. Clearly this flow does not violate capacity constraints, and also flow conservation, for every non S non X vertex they are intermediate vertices in the paths, so the number of incoming flow edges is same as the number of outgoing flow edges for these vertices (the paths are edge-disjoint). For X vertices, they each have exactly one outgoing flow edge, so their incoming flow is also 1 ($f(s, x_i) = 1$). For S vertices, flow conservation holds by the assignment of flow in $f\left(s_j, t\right)$. The value of this flow is $|X|$ since $f(s, V - s) = |X|$, and this is also a max-flow since $c(s, V - s) = |X|$.

2. If the value of the max flow of G' is $|X|$, then such a set of escape routes exist in G.

   Let f be a integral max flow of G', $|f| = |X|$. We can decompose this flow f into a set of $|f|$ path flows from $s \to t$, all of which have a value of 1. (See proof of problem 1). Since no edge $e \in E$ can exist in two of these paths (otherwise the value of flow in e will be > 1), thus we get $|X|$ edge-disjoint paths, (drop s and t from these paths), which start at a vertex in X and end at a vertex in S.

(b) Observe, that to ensure no two paths go through the same edge, we enforced edge capacities to be 1. Now, we are looking for vertex-disjoint paths, i.e. no two paths must use the same vertex. Recall in a previous module, we have shown that a network with vertex and edge capacities can be converted into an equivalent network having only edge capacities, such that their max-flow values are the same.

Construct a graph $G' = (V \cup \{s,t\}, E')$, $E' = E \cup \{(s,x) \mid x \in X\} \cup \{(v,t) \mid v \in S\}$ as usual. Set all edge capacities to ∞, all vertex capacities to 1 (other than s and t). Compute a max-flow in G' (by first converting into a edge-capacity network). If the value of the max-flow is $|X|$, then such a set of evacuating paths exist, otherwise they don't.

Proof:

1. If such a set of escape routes exist in G, then the max flow value is $|X|$ in G'.

Assume there is a valid set of $|X|$ escape routes, one starting at each vertex in X. Construct a flow f by setting $f(e) = 1$ if e is used in one of the paths. Note that being vertex disjoint implies that the paths are also edge-disjoint, since otherwise the same vertex would be used in multiple paths. Capacity conservation and flow conservation holds as before, and the value of this flow is $|X|$. This is a max flow because $c(s, V - s) = |X|$.

2. If the value of the max flow of G' is $|X|$, then such a set of escape routes exist in G.

   Let f be a integral max flow of G', $|f| = |X|$. We can decompose this flow f into a set of $|f|$ path flows from $s \to t$, all of which have a value of 1. No two of these paths share a vertex v, as otherwise the incoming flow to v would be > 1 which is not possible since the capacity of v is 1. So we get a vertex-disjoint set of paths from $|X|$ to $|S|$.

Example of a graph where (a) is true but (b) is false:

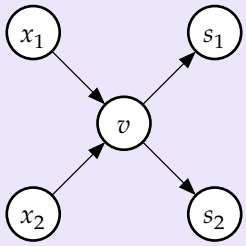


Figure 9: Example where there exists edge-disjoint set of evacuating paths, but no vertex-disjoint set of evacuating paths.

Let $G = (V, E)$, $V = X \cup S \cup \{v\}$, $X = \{x_1, x_2\}$, $S = \{s_1, s_2\}$. $E = \{(x_1, v), (x_2, v), (v, s_1), (v, s_2)\}$

For the above graph G, for part (a) we can find the following set of edge-disjoint evacuating paths: $x_1 \to v \to s_1$ and $x_2 \to v \to s_2$. But for part (b) there do not exist any vertex-disjoint evacuating paths for each $x \in X$ reaching some vertex in S.

5. [KT book] You've been called in to help some network administrators diagnose the extent of a failure in their network. The network is designed to carry traffic from a designated source node s to a designated target node t, so we will model the network as a directed graph G = (V, E), in which the capacity of each edge is 1 and in which each node lies on at least one path from s to t.

   Now, when everything is running smoothly in the network, the maximum $s - t$ flow in G has value k. However, the current situation (and the reason you're here) is that an attacker has destroyed some of the edges in the network, so that there is now no path from s to t using the remaining (surviving) edges. For reasons that we won't go into here, they believe the attacker has destroyed only k edges, the minimum number needed to separate s from t (i.e., the size of a minimum $s - t$ cut); and we'll assume they're correct in believing this.

   The network administrators are running a monitoring tool on node s, which has the following behavior. If you issue the command ping(v), for a given node v, it will tell you whether there is currently a path from s to v. (So ping(t) reports that no path currently exists; on the other hand, ping(s) always reports a path from s to itself.) Since it's not practical to go out and inspect every edge of the network, they'd like to determine the extent of the failure using this monitoring tool, through judicious use of the ping command.

   So here's the problem you face: Give an algorithm that issues a sequence of *ping* commands to various nodes in the network and then reports the *full* set of nodes that are not currently

reachable from s. You could do this by pinging every node in the network, of course, but you'd like to do it using many fewer pings (given the assumption that only k edges have been deleted). In issuing this sequence, your algorithm is allowed to decide which node to ping next based on the outcome of earlier *ping* operations.

Give an algorithm that accomplishes this task using only $O(k \log n)$ pings.

What does $k \log n$ operations suggest? Also look at problem 1...

Solution

Firstly, note that the k edges that are removed must be a min-cut. That is, there is some min-cut (S, T), and the edges that are removed are all the edges crossing this cut.

By Menger's theorem (Problem 1), the value of the max-flow of G is k. Thus, there exist at least k edge-disjoint paths from s to t in G. Since k edges are removed in total, and at least one edge must be removed from each of these paths (otherwise there will still remain a $s - t$ path in the graph). Therefore, from each of these paths exactly one edge must have been removed.

Consider one such path $p = \langle s = v_1, v_2, ..., v_m = t \rangle$. Suppose that the edge (v_i, v_{i+1}) was removed from this path. Then, what will be the outcome of ping(v) for each vertex in path p? Clearly, for $v_1, v_2, ..., v_i$, ping will report a path from s to v exists, whereas, for $v_{i+1}, ..., v_m$, ping reports that no path exists (why?). This is a monotonic function, which means we can binary search over these vertices! Say we want to know the highest index vertex v_i, such that v_i is reachable from s. Then, perform a standard binary search on the range $\{1, ..., l\}$:

- $l = 0, r = m$
- **while** $l < r$, let mid $= \lfloor (l + r + 1)/2 \rfloor$
  - ping(v_{mid}).
  - **if** reachable, then $l \leftarrow$ mid
  - **else**, $r \leftarrow$ mid $- 1$

Finally, l has the value of i such that v_i is the highest indexed vertex reachable from s. This takes $O(\log m) = O(\log n)$ ping operations to find out. Once we know this, we know that (v_i, v_{i+1}) was the edge that was removed.

Doing this for all k paths takes $O(k \log n)$ operations, after which we know exactly the k edges that were removed. Once we know this, we can exactly compute the reachable nodes from s after the removal of these k edges.

6. [KT book] Let M be an $n \times n$ matrix with each entry equal to either 0 or 1. Let m_{ij} denote the entry in row i and column j. A diagonal entry is one of the form m_{ii} for some i.

   Swapping rows i and j of the matrix M denotes the following action: we swap the values m_{ik} and m_{jk} for $k = 1, 2, ..., n$. Swapping two columns is defined analogously.

   We say that M is *rearrangeable* if it is possible to swap some of the pairs of rows and some of the pairs of columns (in any sequence) so that, after all the swapping, all the diagonal entries of M are equal to 1.

   (a) Give an example of a matrix M that is not rearrangeable, but for which at least one entry in each row and each column is equal to 1.

   (b) Give a polynomial-time algorithm that determines whether a matrix M with $0 - 1$ entries is rearrangeable.

First observation: The swapping operation is reversible. So if some matrix M is rearrangeable, that means that from a matrix with all ones in the diagonal entries, we can perform some swaps and reach M.

$$\begin{bmatrix} 1 & X & X & X \\ X & 1 & X & X \\ X & X & 1 & X \\ X & X & X & 1 \end{bmatrix} \xrightarrow{\text{some swaps}} M$$

Second observation: Track the positions of these n diagonal ones throughout these swaps. Since initially the ith and jth one are in different row and different column, they continue to remain in different row and column. (by the definition of the swapping operations). For e.g., one possible M which is rearrangeable can be of the form

$$M = \begin{bmatrix} X & 1 & X & X \\ 1 & X & X & X \\ X & X & X & 1 \\ X & X & 1 & X \end{bmatrix}$$

Then, we have a characterisation: If M is rearrangeable, there exists permutations $i_1, ..., i_n$ and $j_1, ..., j_n$, such that $m_{i_k, j_k} = 1$ for all k.

Sounds familiar?

(a) M has at least one 1 in each row and each column, but is not rearrangeable.

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

(b)

Construct a bipartite graph $G = (A \uplus B, E), A = \{a_1, a_2, ..., a_n\}, B = \{b_1, b_2, ..., b_n\}, E = \left\{\left(a_i, b_j\right) \mid m_{ij} = 1\right\}$. (Add edges between a_i and b_j if $m_{ij} = 1$.) If G has a perfect matching, then M is rearrangeable, otherwise it is not.

Proof:

Suppose G has a perfect matching. Then, let that matching have edges $\left(a_1, b_{\pi_1}\right), \left(a_2, b_{\pi_2}\right), ..., \left(a_n, b_{\pi_n}\right)$. This means, for all $i = 1, 2, ..., n$, we have $m_{i,\pi_i} = 1$, i.e. in the ith row, the π_ith entry is a 1. (All π_is are distinct.) Now, we need to rearrange it to get all these ones in the diagonal cells. We can perform a sequence of column swaps to reorder the columns according to the inverse of the permutation π. We swap columns until the original column π_i is moved to the i th position. (Any permutation can be decomposed into a sequence of swaps). After these swaps, an entry that was located at (i, π_i) is now located at (i, i). Thus, all diagonal elements of M' are 1, hence M is rearrangeable.

Suppose M is rearrangeable. This means there exist a sequence of row and column swaps, which result in a new matrix M', such that all its diagonal elements are 1. Then, there exists permutations $i_1, ..., i_n$ and $j_1, ..., j_n$ such that $m_{i_k, j_k} = 1$ for all $k = 1, 2, ..., n$. By construction of G, $(i_k, j_k) \in E$. Then, in G, consider $W = \left\{\left(a_{i_k}, b_{i_k}\right) \mid k = 1, 2, ..., n\right\}$. No two edges in this set share any vertices since i_k and j_k are permutations. Thus, W is a perfect matching in G.