Notes on Reductions, Cuts and Matchings

In which we describe how to reduce other problems to the max flow problem, and how to analyze a randomized algorithm for finding the minimum cut in an undirected graph.

1 Variations of the Maximum Flow Problem

There are several variations of the maximum flow problem that are efficiently solvable given an algorithm for the standard maximum flow problem. We discuss an example.

In the max flow problem *with node capacities*, we are given a network in which, besides having a capacity $c_{u,v}$ for every edge (u, v), we are also given a capacity c_v for each node. A feasible flow must satisfy the edge capacity constraints and the conservation constraints as in the standard problem, and in addition it must also satisfy *node capacity* constraints that say that the total flow going into each node v (which will be equal to the total flow going out of the node v) must be at most c_v.

Given an instance of this problem, we can construct an equivalent instance of the standard max flow problem in the following way: for each node v other than s, t in the original instance, we create two nodes v_{in} and v_{out} in the new instance, with an edge (v_{in}, v_{out}) of capacity $c_{v_{in},v_{out}} = c_v$. For each edge (u, v) of the original instance, we create an edge (u_{out}, v_{in}) in the new network, of capacity $c_{u_{out},v_{in}} = c_{u,v}$.

Finally we claim that for every feasible flow of value F in the original network there is a feasible flow of the same value in the new network, and vice versa. To see how, if f is a feasible flow in the original network, we can create a feasible flow f' in the new network by letting $f'_{v_{in},v_{out}}$ be equal to the total flow through the node v according to f', and letting $f'_{u_{out},v_{in}} = f_{u,v}$ for every edge (u, v) of the original network. Conversely, if f is a feasible flow in the new network, we can get a feasible flow f' for the original network by letting $f'_{u,v} = f_{u_{out},v_{in}}$ for every edge (u, v) of the original network.

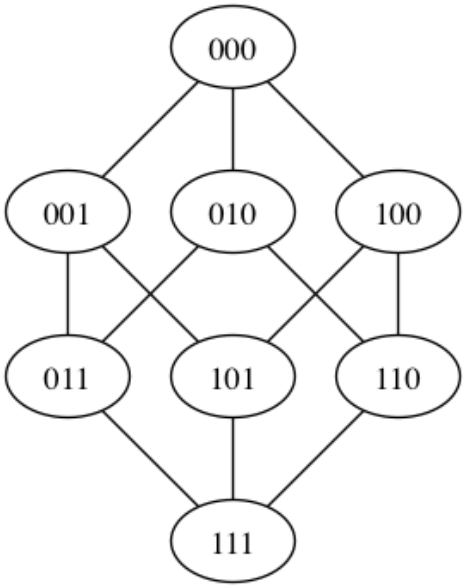
2 Maximum Matching in Bipartite Graphs

In this section we show applications of the theory of (and of algorithms for) the maximum flow problem to the design an algorithm for finding a maximum matching

in a given bipartite graph.

A bipartite graph is an undirected graph $G = (V, E)$ such that the set of vertices V can be partitioned into two subsets L and R such that every edge in E has one endpoint in L and one endpoint in R.

For example, the 3-cube is bipartite, as can be seen by putting in L all the vertices whose label has an even number of ones and in R all the vertices whose label has an odd number of ones.



There is a simple linear time algorithm that checks if a graph is bipartite and, if so, finds a partition of V into sets L and R such that all edges go between L and R: run DFS and find a spanning forest, that is, a spanning tree of the graph in each connected component. Construct sets L and R in the following way. In each tree, put the root in L, and then put in R all the vertices that, in the tree, have odd distance from the root; put in L all the vertices that, in the tree, have even distance from the root. If the resulting partition is not valid, that is, if there is some edge both whose endpoints are in L or both whose endpoints are in R, then there is some tree in which two vertices u, v are connected by an edge, even though they are both at even distance or both at odd distance from the root r; in such a case, the cycle that goes from r to u along the tree, then follows the edge (u, v) and then goes from v to r along the three is an odd-length cycle, and it is easy to prove that in a bipartite graph there is no odd cycle. Hence the algorithm either returns a valid bipartition or a certificate that the graph is not bipartite.

Several optimization problems become simpler in bipartite graphs. The problem of finding a *maximum matching* in a graph is solvable in polynomial time in general

graphs, but it has a very simple algorithm in bipartite graphs, that we shall see shortly. (The algorithm for general graphs is beautiful but rather complicated.)

In a undirected graph $G = (V, E)$, a matching is a subset $M \subseteq E$ of the edges such that every vertex of G is an endpoint of at most one edge in M. Said another way, every vertex of the graph (V, M) has degree zero or one. The maximum matching problem is the problem of finding a matching with the largest number of edges. We will describe an algorithm that is based on a reduction to the maximum flow problem. The reduction has other applications, because it makes the machinery of the max flow - min cut theorem applicable to reason about matchings. For example, it would be possible to give a very simple proof of Hall's theorem, a classical result in graph theorem, based on the max flow - min cut theorem.

The problem of finding a *maximum matching* in a graph, that is, a matching with the largest number of edges, often arises in assignment problems, in which tasks are assigned to agents, and almost always the underlying graph is bipartite, so it is of interest to have simpler and/or faster algorithms for maximum matchings for the special case in which the input graph is bipartite.

Consider the following algorithm.

- Input: undirected bipartite graph $G = (V, E)$, partition of V into sets L, R
- Construct a network $(G' = (V', E'), s, t, c)$ as follows:
  - the vertex set is $V' := V \cup \{s, t\}$, where s and t are two new vertices;
  - E' contains a directed edge (s, u) for every $u \in L$, a directed edge (u, v) for every edge $(u, v) \in E$, where $u \in L$ and $v \in R$, and a directed edge (v, t) for every $v \in R$;
  - each edge has capacity 1;
- find a maximum flow $f(\cdot, \cdot)$ in the network, making sure that all flows $f(u, v)$ are either zero or one
- return $M := \{(u, v) \in E \text{ such that } f(u, v) = 1\}$

The running time of the algorithm is the time needed to solve the maximum flow on the network (G', s, t, c) plus an extra $O(|E|)$ amount of work to construct the network and to extract the solution from the flow. In the constructed network, the maximum flow is at most $|V|$, and so, using the Ford-Fulkerson algorithm, we have running time $O(|E| \cdot |V|)$. The fastest algorithm for maximum matching in bipartite graphs, applies a max-flow algorithm called the *push-relabel algorithm* to the above network, and it has running time $O(|V| \cdot \sqrt{|E|})$. It is also possible to solve the problem in time $O(MM(|V|))$, where $MM(n)$ is the time that it takes to multiply two $n \times n$ matrices. (This approach does not use flows.) Using the currently best known matrix

multiplication algorithm, the running time is about $O(|V|^{2.37})$, which is better than $O(|V|\sqrt{|E|})$ in dense graphs. The algorithm based on push-relabel is always better in practice.

Remark 1 (Integral Flows) *It is important in the reduction that we find a flow in which all flows are either zero or one. In a network in which all capacities are zero or one, all the algorithms that we have seen in class will find an optimal solution in which all flows are either zero or one. More generally, on input a network with integer capacities, all the algorithms that we have seen in class will find a maximum flow in which all $f(u, v)$ are integers. It is important to keep in mind, however, that, even though in a network with zero/one capacities there always exists an optimal integral flow, there can also be optimal flows that are not integral.*

We want to show that the algorithm is correct that is that: (1) the algorithm outputs a matching and (2) that there cannot be any larger matching than the one found by the algorithm.

Claim 2 *The algorithm always outputs a matching, whose size is equal to the cost of the maximal flow of G'.*

PROOF: Consider the flow $f(\cdot, \cdot)$ found by the algorithm. For every vertex $u \in L$, the conservation constraint for u and the capacity constraint on the edge (s, u) imply:

$$\sum_{r:(u,r)\in E} f(u, r) = f(s, u) \leq 1$$

and so at most one of the edges of M can be incident on u.

Similarly, for every $v \in R$ we have

$$\sum_{\ell:(\ell,v)\in E} f(\ell, v) = f(v, t) \leq 1$$

and so at most one of the edges in M can be incident on v. □

Remark 3 *Note that the previous proof does not work if the flow is not integral*

Claim 4 *The size of the largest matching in G is at most the cost of the maximum flow in G'.*

PROOF: Let M^* be a largest matching in G. We can define a feasible flow in G' in the following way: for every edge $(u, v) \in M^*$, set $f(s, u) = f(u, v) = f(v, t) = 1$. Set all the other flows to zero. We have defined a feasible flow, because every flow is either zero or one, and it is one only on edges of G', so the capacity constraints are satisfied, and the conservation constraints are also satisfied, because for every vertex that is not matched in M^* there is zero incoming flow and zero outgoing flow, while for the matched vertices there is one unit of incoming flow and one unit of outgoing flow. The cost of the flow is the number of vertices in L that are matched, which is equal to $|M^*|$.

This means that there exists a feasible flow whose cost is equal to $|M^*|$, and so the cost of a maximum flow is greater than or equal to $|M^*|$. □

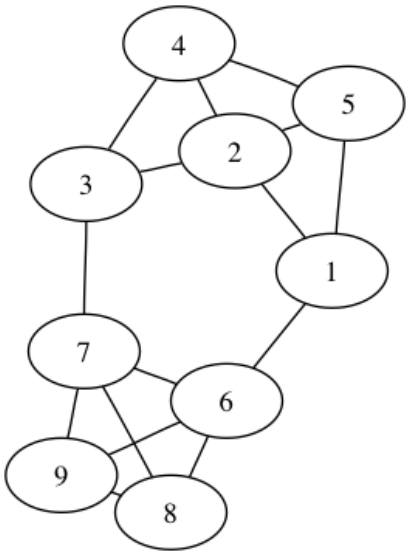
So we have established that our algorithm is correct and optimal.

3 Global Min-Cut and Edge-Connectivity

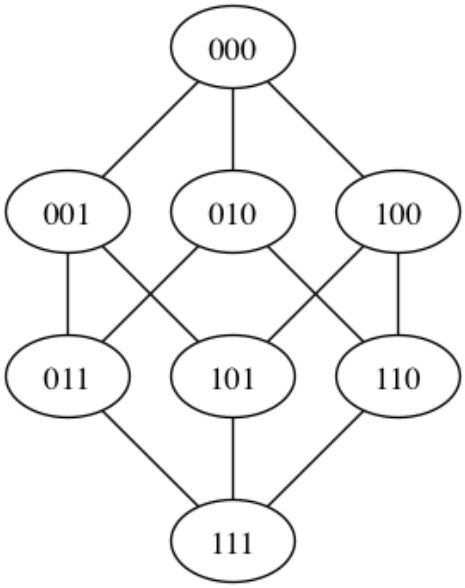
Definition 5 (Edge connectivity) *We say that an undirected graph is k-edge-connected if one needs to remove at least k edges in order to disconnect the graph. Equivalently, an undirected graph is k-edge-connected if the removal of any subset of $k - 1$ edges leaves the graph connected.*

Note that the definition is given in such a way that if a graph is, for example 3-edge-connected, then it is also 2-edge-connected and 1-edge connected. Being 1-edge-connected is the same as being connected.

For example, the graph below is connected and 2-edge connected, but it is not 3-edge connected, because removing the two edges $(3, 7)$ and $(1, 6)$ disconnects the graph.



As another example, consider the 3-cube:



The 3-cube is clearly not 4-edge-connected, because we can disconnect any vertex by removing the 3 edges incident on it. It is clearly connected, and it is easy to see that it is 2-edge-connected; for example we can see that it has a Hamiltonian cycle (a simple cycle that goes through all vertices), and so the removal of any edge still leaves a path that goes trough every vertex. Indeed the 3-cube is 3-connected, but at this point it is not clear how to argue it without going through some complicated case analysis.

The *edge-connectivity* of a graph is the largest k for which the graph is k-edge-connected, that is, the minimum k such that it is possible to disconnect the graph by removing k edges.

In graphs that represent communication or transportation networks, the edge-connectivity is an important measure of reliability.

Definition 6 (Global Min-Cut) *The global min-cut problem is the following: given in input an undirected graph $G = (V, E)$, we want to find the subset $A \subseteq V$ such that $A \neq \emptyset$, $A \neq V$, and the number of edges with one endpoint in A and one endpoint in $V - A$ is minimized.*

We will refer to a subset $A \subseteq V$ such that $A \neq \emptyset$ and $A \neq V$ as a *cut* in the graph, and we will call the number of edges with one endpoint in A and one endpoint in $V - A$ the *cost* of the cut. We refer to the edges with one endpoint in A and one endpoint in $V - A$ as the edges that *cross* the cut.

We can see that the Global Min Cut problem and the edge-connectivity problems are in fact the same problem:

- if there is a cut A of cost k, then the graph becomes disconnected (in particular, no vertex in A is connected to any vertex in $V - A$) if we remove the k edges that cross the cut, and so the edge-connectivity is at most k. This means that the edge-connectivity of a graph is at most the cost of its minimum cut;
- if there is a set of k edges whose removal disconnects the graph, then let A be the set of vertices in one of the resulting connected components. Then A is a cut, and its cost is at most k. This means that the cost of the minimum cut is at most the edge-connectivity.

We will discuss two algorithms for finding the edge-connectivity of a graph. One is a simple reduction to the maximum flow problem, and runs in time $O(|E| \cdot |V|^2)$. The other is a surprising simple randomized algorithm based on *edge-contractions* – the surprising part is the fact that it correctly solves the problem, because it seems to hardly be doing any work. We will discuss a simple $O(|V|^3)$ implementation of the edge-contraction algorithm, which is already better than the reduction to maximum flow. A more refined analysis and implementation gives a running time $O(|V|^2 \cdot (\log |V|)^{O(1)})$.

3.1 Reduction to Maximum Flow

Consider the following algorithm:

- Input: undirected graph $G = (V, E)$
- let s be a vertex in V of minimal degree
- define $c(u, v) = 1$ for every $(u, v) \in E$
- for each $t \in V - \{s\}$
  - solve the min cut problem in the network (G, s, t, c), and let A_t be the cut of minimum capacity
- output the cut A_t of minimum cost

The algorithm uses $|V| - 1$ minimum cut computations in networks, each of which can be solved by a maximum flow computation. Since each network can have a maximum flow of cost at most the degree of s, which is at most $2|E|/|V|$, and all capacities are integers, the Ford-Fulkerson algorithm finds each maximum flow in time $O(|E| \cdot opt) = O(|E|^2/|V|)$ and so the overall running time is $O(|E|^2)$.

To see that the algorithm finds the global min cut, let k be edge-connectivity of the graph, E^* be a set of k edges whose removal disconnects the graph, and let A^* be the connected component containing s in the disconnected graph resulting from the removal of the edges in E^*. So A^* is a global minimum cut of cost at most k (indeed, exactly k), and it contains s.

In at least one iteration, the algorithm constructs a network (G, s, t, c) in which $t \notin A^*$, which means that A^* is a valid cut, of capacity k, for the network, and so when the algorithm finds a minimum capacity cut in the network it must find a cut of capacity at most k (indeed, exactly k). This means that, for at least one t, the cut A_t is also an optimal global min-cut.

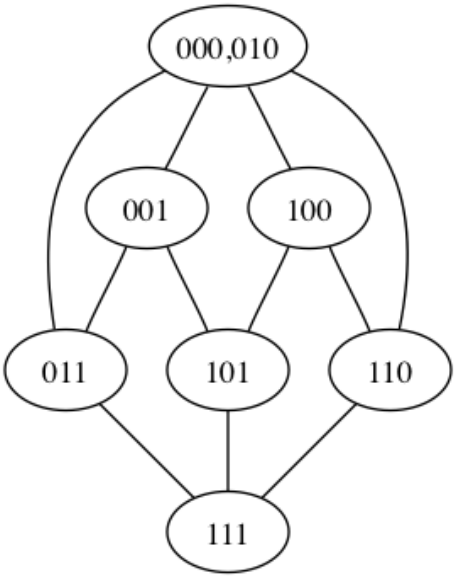
3.2 The Edge-Contraction Algorithm

Our next algorithm is due to David Karger, and it involves a rather surprising application of random choices.

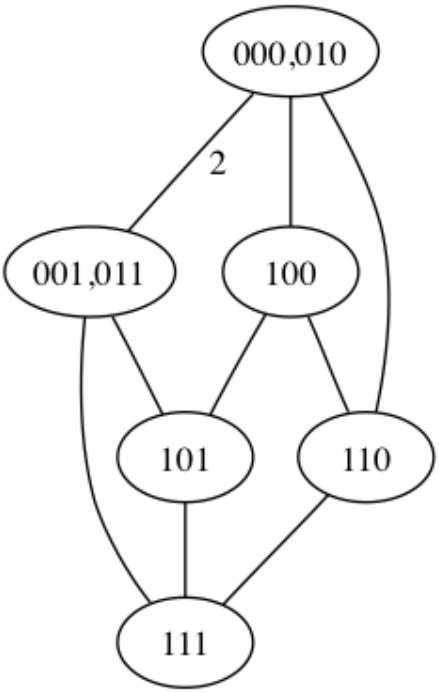
The algorithm uses the operation of *edge-contraction*, which is an operation defined over multi-graphs, that is graphs that can have multiple edges between a given pair of vertices or, equivalently, graphs whose edges have a positive integer weight.

If, in an undirected graph $G = (V, E)$ we *contract* an edge (u, v), the effect is that the edge (u, v) is deleted, and the vertices u and v are removed, and replaced by a new vertex, which we may call $[u, v]$; all other edges of the graph remain, and all the edges that were incident on u or v become incident on the new vertex $[u, v]$. If u had i edges connecting it to w, and v had j edges connecting it to w, then in the new graph there will be $i + j$ edges between w and $[u, v]$.

For example, if we contract the edge $(000, 010)$ in the 3-cube we have the following graph.



And if, in the resulting graph, we contract the edge $(001, 011)$, we have the following graph.



Note that, after the two contractions, we now have two edges between the "macro-vertices" $[000, 010]$ and $[001, 011]$.

The basic iteration of Karger's algorithm is the following:

- while there are ≥ 3 vertices in the graph
  - pick a random edge and contract it
- output the set A of vertices of the original graph that have been contracted into one of the two final macro-vertices.

One important point is that, in the randomized step, we sample uniformly at random among the edges of the multi-set of edges of the current *multi*-graph. So if there are 6 edges between the vertices (a, b) and 2 edges between the vertices (c, d), then a contraction of (a, b) is three times more likely than a contraction of (c, d).

The algorithm seems to pretty much pick a subset of the vertices at random. How can we hope to find an optimal cut with such a simple approach?

(In the analysis we will assume that the graph is connected. if the graph has two connected components, then the algorithm converges to the optimal min-cut of cost zero. If there are three or more connected components, the algorithm will discover them when it runs out of edges to sample, In the simplified pseudocode above we omitted the code to handle this exception.)

The first observation is that, if we fix for reference an optimal global min cut A^* of cost k, and if it so happens that there is never a step in which we contract one of the k edges that connect A^* with the rest of the graph then, at the last step, the two macro-vertices will indeed be A^* and $V - A^*$ and the algorithm will have correctly discovered an optimal solution.

But how likely is it that the k edges of the optimal solution are never chosen to be contracted at any iteration?

The key observation in the analysis is that if we are given in input a (multi-)graph whose edge-connectivity is k, then it must be the case that every vertex has degree $\geq k$, where the degree of a vertex in a graph or multigraph is the number of edges that have that vertex as an endpoint. This is because if we had a vertex of degree $\leq k - 1$ then we could disconnect the graph by removing all the edges incident on that vertex, and this would contradict the k-edge-connectivity of the graph.

But if every vertex has degree $\geq k$, then

$$|E| = \frac{1}{2} \sum_v \text{degree}(v) \geq \frac{k}{2} \cdot |V|$$

and, since each edge has probability $1/|E|$ of being sampled, the probability that, at the first step, we sample one of the k edges that cross the cut A^* is only

$$\frac{k}{|E|} \leq \frac{2}{|V|}$$

What about the second step, and the third step, and so on?

Suppose that we were lucky at the first step and that we did not select any of the k edges that cross A^*. Then, after the contraction of the first step we are left with a graph that has $|V| - 1$ vertices. The next observation is that this new graph *has still edge-connectivity* k because the cut defined by A^* is still well defined. If the edge-connectivity is still k, we can repeat the previous reasoning, and conclude that the probability that we select one of the k edges that cross A^* is at most

$$\frac{2}{|V| - 1}$$

And now we see how to reason in general. If we did not select any of the k edges that cross A^* at any of the first step $t - 1$ step, then the probability that we select one of those edges at step t is at most

$$\frac{2}{|V| - t + 1}$$

So what is the probability that we never select any of those edges at any step, those ending up with the optimal solution A^*? If we write E_t to denote the event that "at step t, the algorithm samples an edge which does not cross A^*," then

$$\mathbb{P}[E_1 \wedge E_2 \wedge \cdots \wedge E_{n-2}]$$

$$= \mathbb{P}[E_1] \cdot \mathbb{P}[E_2|E_1] \cdot \ldots \mathbb{P}[E_{n-2}|E_1 \wedge \cdots \wedge E_{n-3}]$$
$$\geq \left(1 - \frac{2}{|V|}\right) \cdot \left(1 - \frac{2}{|V| - 1}\right) \cdot \ldots \left(1 - \frac{2}{3}\right)$$

If we write $n := |V|$, the product in the last line is

$$\frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \cdot \ldots \frac{4}{6} \cdot \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3}$$

which simplifies to

$$\frac{2}{n \cdot (n-1)}$$

Now, suppose that we repeat the basic algorithm r times. Then the probability that it does not find a solution in any of the r attempts is at most

$$\left(1 - \frac{2}{n \cdot (n-1)}\right)^r$$

So, for example, if we repeat the basic iteration $50n \cdot (n-1)$ times, then the probability that we do not find an optimal solution is at most

$$\left(1 - \frac{2}{n \cdot (n-1)}\right)^{50n \cdot (n-1)} \leq e^{-100}$$

(where we used the fact that $1 - x \leq e^{-x}$), which is an extremely small probability.

One iteration of Karger's algorithm can be implemented in time $O(|V|)$, so overall we have an algorithm of running time $O(|V|^3)$ which has probability at least $1 - e^{-100}$ of finding an optimal solution.