CS261: A Second Course in Algorithms
Lecture #6: Generalizations of Maximum Flow and Bipartite Matching*

Tim Roughgarden†

January 21, 2016

1 Fundamental Problems in Combinatorial Optimization

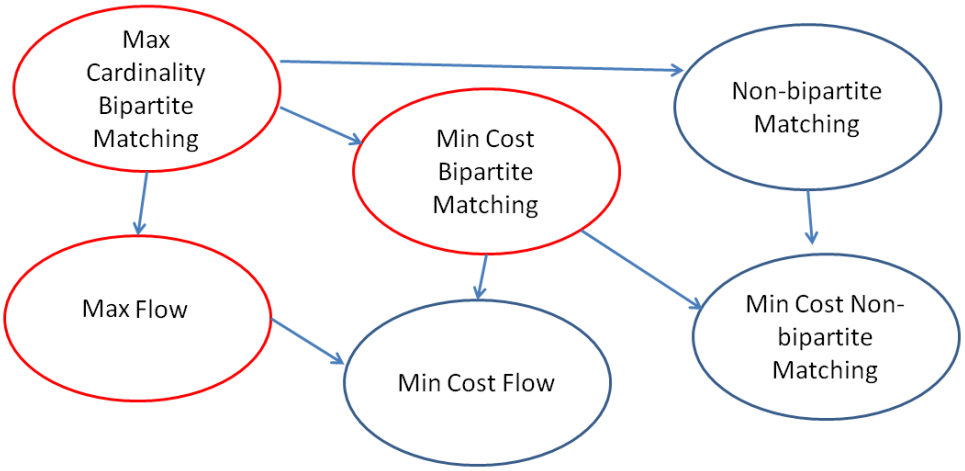

Figure 1: Web of six fundamental problems in combinatorial optimization. The ones covered thus far are in red. Each arrow points from a problem to a generalization of that problem.

We started the course by studying the maximum flow problem and the closely related s-t cut problem. We observed (Lecture #4) that the maximum-cardinality bipartite matching

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

problem can be viewed as a special case of the maximum flow problem (Figure 1). We then generalized the former problem to include edge costs, which seemed to give a problem incomparable to maximum flow.

The inquisitive student might be wondering the following:

1. Is there a natural common generalization of the maximum flow and minimum-cost bipartite matching problems?

2. What's up with graph matching in non-bipartite graphs?

The answer to the first question is "yes," and it's a problem known as minimum-cost flow (Figure 1). For the second question, there is a nice theory of graph matchings in non-bipartite graphs, both for the maximum-cardinality and minimum-cost cases, although the theory is more difficult and the algorithms are slower than in the bipartite case. This lecture introduces the three new problems in Figure 1 and some essential facts you should know about them. The six problems in Figure 1, along with the minimum spanning tree and shortest path problems that you already know well from CS161, arguably form the complete list of the most fundamental problems in *combinatorial optimization*, the study of efficiently optimizing over large collections of discrete structures.

The main take-ways from this lecture's high-level discussion are:

1. You should know about the existence of the minimum-cost flow and non-bipartite matching problems. They do come up in applications, if somewhat less frequently than the problems studied in the first five lectures.

2. There are reasonably efficient algorithms for all of these problems, if a bit slower than the state-of-the-art algorithms for the problems discussed previously. We won't discuss running times in any detail, but think of roughly $O(mn)$ or $O(n^3)$ as a typical time bound of a smart algorithm for these problems.

3. The algorithms and analysis for these problems follow exactly the same principles that you've been studying in previous lectures. They use optimality conditions, various progress measures, well-chosen invariants, and so on. So you're well-positioned to study deeply these problems and algorithms for them, in another course or on your own. Indeed, if CS261 were a semester-long course, we would cover this material in detail over the next 4-5 lectures. (Alas, it will be time to move on to linear programming.)

2 The Minimum Cost Flow Problem

An instance of the *minimum-cost flow problem* consists of the following ingredients:

- a directed graph $G = (V, E)$;
- a source $s \in V$ and sink $t \in V$;

- a target flow value d;
- a nonnegative capacity u_e for each edge $e \in E$;
- a real-valued cost c_e for each edge $e \in E$.

The goal is to compute a flow f with value d — that is, pushing d units of flow from s to t, subject to the usual conservation and capacity constraints — that minimizes the overall cost

$$\sum_{e \in E} c_e f_e. \tag{1}$$

Note that, for each edge e, we think of c_e as a "per-flow unit" cost, so with f_e units of flow the contribution of edge e to the overall cost is $c_e f_e$.[1]

There are two differences with the maximum flow problem. The important one is that now every edge has a cost. (In maximum flow, one can think of all the costs being 0.) The second difference, which is artificial, is that we specified a specific amount of flow d to send. There are multiple other equivalent formulations of the minimum-cost flow problem. For example, one can ask for the maximum flow with the minimum cost. Alternatively, instead of having a source s and sink t, one can ask for a "circulation" — meaning a flow that satisfies conservation constraints at every vertex of V — with the minimum cost (in the sense of (1)).[2]

Impressively, the minimum-cost flow problem captures three different problems that you've studied as special cases.

1. **Shortest paths.** Suppose you are given a "black box" that quickly does minimum-cost flow computations, and you want to compute the shortest path between some s and some t is a directed graph with edge costs. The black box is expecting a flow value d and edge capacities u_e (in addition to G, s, t, and the edge costs); we just set $d = 1$ and $u_e = 1$ (say) for every edge e. An integral minimum-cost flow in this network will be a shortest path from s to t (why?).

2. **Maximum flow.** Given an instances of the maximum flow problem, we need to define d and edge costs before feeding the input into our minimum-cost flow black box. The edge costs should presumably be set to 0. Then, to compute the maximum flow value, we can just use binary search to find the largest value of d for which the black box returns a feasible solution.

3. **Minimum-cost perfect bipartite matching.** The reduction here is the same as that from maximum-cardinality bipartite matching to maximum flow (Lecture #4) — the edge costs just carry over. The value d should be set to n, the number of vertices on each side of the bipartite graph (why?).

[1] If there is no flow of value d, then an algorithm should report this fact. Note this is easy to check with a single maximum flow computation.

[2] Of course if all edge costs are nonnegative, then the all-zero solution is optimal. But with negative cycles, this is a nontrivial problem.

Problem Set #2 explores various aspects of minimum-cost flows. Like the other problems we've studied, there are nice optimality conditions for minimum-cost flows. First, one extends the notion of a residual network to networks with costs — the only twist is that if an edge (w, v) of the residual network is the reverse edge corresponding to $(v, w) \in E$, then the cost of c_{wv} should be set to $-c_{vw}$. (Which makes sense given that reverse edges correspond to "undo" operations.) Then, a flow with value d is minimum-cost if and only if the corresponding residual network has no negative cycle. This then suggests a simple "cycle-canceling" algorithm, analogous to the Ford-Fulkerson algorithm. Polynomial-time algorithms can be designed using the same ideas we used for maximum flow in Lectures #2 and #3 and Problem Set #1 (blocking flows, push-relabel, scaling, etc.). There are algorithms along these lines with running time roughly $O(mn)$ that are also quite fast in practice. (Theoretically, it is also known how do a bit better.) In general, you should be happy if a problem that you care about reduces to the minimum-cost flow problem.

3 Non-Bipartite Matching

3.1 Maximum-Cardinality Non-Bipartite Matching

In the general (non-bipartite) matching problem, the input is an undirected graph $G = (V, E)$, not necessarily bipartite. The goal to compute a matching (as before, a subset $M \subseteq E$ with no shared endpoints) with the largest cardinality. Recall that the simplest non-bipartite graphs are odd cycles (Figure 2).

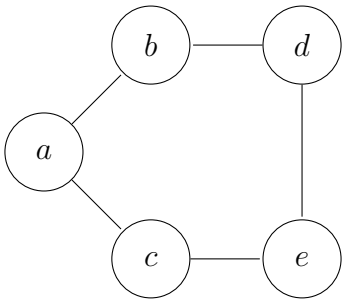

Figure 2: Example of non-bipartite graph: odd cycle.

A priori, it is far from obvious that the general graph matching problem is solvable in polynomial time (as opposed to being NP-hard). It appears to be significantly more difficult than the special case of bipartite matching. For example, there does not seem to be a natural reduction from non-bipartite matching to the maximum flow problem. Once again, we need to develop from scratch algorithms and strategies for correctness,

The non-bipartite matching problem admits some remarkable optimality conditions. For motivation, what is the maximum size of a matching in the graph in Figure 3? There are 16

vertices, so clearly a matching has at most 8 edges. It's easy to exhibit a matching of size 6 (Figure 3), but can we do better?

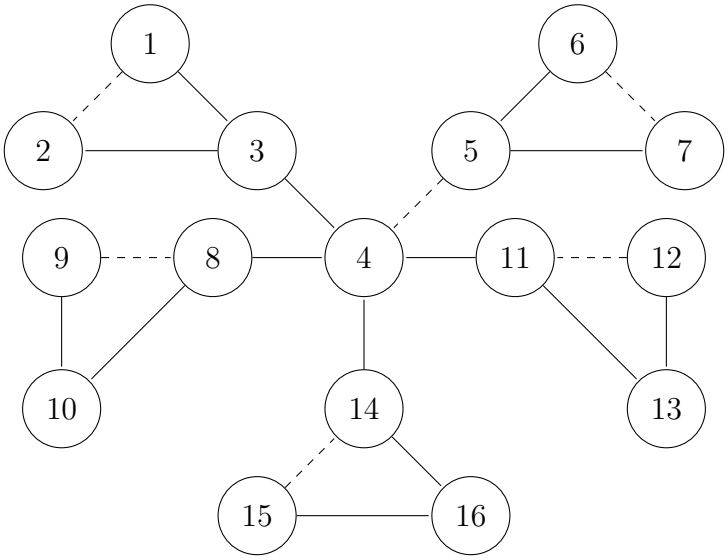

Figure 3: Example graph. A matching of size 6 is denoted by dashed edges.

Here's one way to argue that there is no better matching. In each of the 5 triangles, at most 2 of the 3 vertices can be matched to each other. This leaves at least five vertices, one from each triangle, that, if matched, can only be matched to the center vertex. The center vertex can only be matched to one of these five, so every matching leaves at least four vertices unmatched. This translates to matching at most 12 vertices, and hence containing at most 6 edges.

In general, we have the following.

Lemma 3.1 *In every graph $G = (V, E)$, the maximum cardinality of a matching is at most*

$$\min_{S \subseteq V} \frac{1}{2} \left[|V| - (\mathrm{oc}(S) - |S|) \right], \tag{2}$$

where $\mathrm{oc}(S)$ *denotes the number of odd-size connected components in the graph* $G \setminus S$.

Note that $G \setminus S$ consists of the pieces left over after ripping the vertices in S out of the graph G (Figure 4).

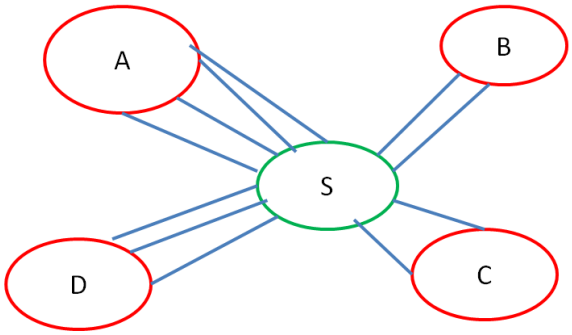

Figure 4: Suppose removing S results in 4 connected components, A, B, C and D. If 3 of them are odd-sized, then $oc(S) = 3$

For example, in the Figure 3, we effectively took S to be the center vertex, so $\text{oc}(S) = 5$ (since $G \setminus S$ is the five triangles) and (2) is $\frac{1}{2}(16-(5-1)) = 6$. The proof is a straightforward generalization of our earlier argument.

Proof of Lemma 3.1: Fix $S \subseteq V$. For every odd-size connected component C of $G \setminus S$, at least one vertex of C is not matched to some other vertex of C. These $\text{oc}(S)$ vertices can only be matched to vertices of S (if two vertices of C_1 and C_2 could be matched to each other, then C_1 and C_2 would not be separate connected components of $G \setminus S$). Thus, every matching leaves at least $\text{oc}(S) - |S|$ vertices unmatched, and hence matches at most $|V| - (\text{oc}(S) - |S|)$ vertices, and hence has at most $\frac{1}{2}(|V| - (\text{oc}(S) - |S|))$ edges. Ranging over all choices of $S \subseteq V$ yields the upper bound in (2). ■

Lemma 3.1 is an analog of the fact that a maximum flow is at most the value of a minimum s-t cut. We can think of (2) as the best upper bound that we can prove if we restrict ourselves to "obvious obstructions" to large matchings. Certainly, if we ever find a matching with size equal to (2), then no other matching could be bigger. But can there be a gap between the maximum size of a matching and the upper bound in (2)? Could there be obstructions to large matchings more subtle than the simple parity argument used to prove Lemma 3.1? One of the more beautiful theorems in combinatorics asserts that there can never be a gap.

Theorem 3.2 (Tutte-Berge Formula) *In Lemma 3.1, equality always holds:*

$$\textit{max matching size} = \min_{S \subseteq V} \frac{1}{2}\left[|V| - (\text{oc}(S) - |S|)\right].$$

The original proof of the Tutte-Berge formula is via induction, and does not seems to lead to an efficient algorithm.[3] In 1965, Edmonds gave the first polynomial-time algorithm for

[3]Tutte characterized the graphs with perfect matchings in the 1940s; in the 1950s, Berge extended this characterization to prove Theorem 3.2.

computing a maximum-cardinality matching.[4] Since the algorithm is guaranteed to produce a matching with cardinality equal to (2), Edmonds' algorithm provides an algorithmic proof of the Tutte-Berge formula.

A key challenge in non-bipartite matching is searching for a good path to use to increase the size of the current matching. Recall that in the Hungarian algorithm (Lecture #5), we used the bipartite assumption to argue that there's no way to encounter both endpoints of an edge in the current matching in the same level of the search tree. But this certainly *can* happen in non-bipartite graphs, even just in the triangle. Edmonds called these odd cycles "blossoms," and his algorithm is often called the "blossom algorithm." When a blossom is encountered, it's not clear how to proceed with the search. Edmonds' idea was to "shrink," meaning contract, a blossom when one is found. The blossom becomes a super-vertex in the new (smaller) graph, and the algorithm can continue. All blossoms are uncontracted in reverse order at the end of the algorithm.[5]

3.2 Minimum-Cost Non-Bipartite Matching

An algorithm designer is never satisfied, always wanting better and more general solutions to computational problems. So it's natural to consider the graph matching problem with both of the complications that we've studied so far: general (non-bipartite) graphs and edge costs.

The minimum-cost non-bipartite matching problem is again polynomial-time solvable, again first proved by Edmonds. From 30,000 feet, the idea to combine the blossom shrinking idea above (which handles non-bipartiteness) with the vertex prices we used in Lecture #5 for the Hungarian algorithm (which handle costs). This is not as easy as it sounds, however — it's not clear what prices should be given to super-vertices when they are created, and such super-vertices may need to be uncontracted mid-algorithm. With some care, however, this idea can be made to work and yields a polynomial-time algorithm.

While polynomial-time solvable, the minimum-cost matching problem is a relatively hard problem within the class P. State-of-the-art algorithms can handle graphs with 100s of vertices, but graphs with 1000s of vertices are already a challenge. From your other computer science courses, you know that in applications one often wants to handle graphs that are bigger than this by 1–6 orders of magnitude. This motivates the design of heuristics for matching that are very fast, even if not fully correct.[6]

For example, the following Kruskal-like greedy algorithm is a natural one to try. For convenience, we work with the equivalent maximum-weight version of the problem (each edge

[4] In this remarkable paper, titled "Paths, Trees, and Flowers," Edmonds defines the class of polynomial-time solvable problems and conjectures that the traveling salesman problem is not in the class (i.e., that $P \neq NP$). Keep in mind that NP-completeness wasn't defined (by Cook and Levin) until 1971.

[5] Your instructor covered this algorithm in last year's CS261, in honor of the algorithm's 50th anniversary. It takes two lectures, however, and has been cut this year in favor of other topics.

[6] In the last part of the course, we explore this idea in the context of approximation algorithms for NP-hard problems. It's worth remembering that for sufficiently large data sets, approximation is the most appropriate solution even for problems that are polynomial-time solvable.

has a weight w_e, the goal is to compute the matching with largest sum of weights).

Greedy Matching Algorithm

sort and rename the edges $E = \{1, 2, \ldots, m\}$ so that $w_1 \geq w_2 \geq \cdots w_m$
$M = \emptyset$
for $i = 1$ to m **do**
    if e_i shares no endpoint with edges in M **then**
        add e_i to M

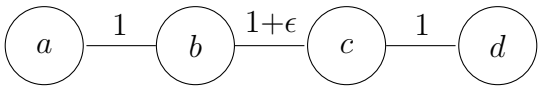

Figure 5: The greedy algorithm picks the edge (b, c), while the optimal matching consists of (a, b) and (c, d).

A simple example (Figure 5) shows that, at least for some graphs, the greedy algorithm can produce a matching with weight only 50% of the maximum possible. On Problem Set #2 you will prove that there are no worse examples — for every (non-bipartite) graph and edge weights, the matching output by the greedy algorithm has weight at least 50% of the maximum possible. Just over the past few years, new matching approximation algorithms have been developed, and it's now possible to get a $(1 - \epsilon)$-approximation in $O(m)$ time, for any constant $\epsilon > 0$ (the hidden constant in the "big-oh" depends on $\frac{1}{\epsilon}$) [**?**].