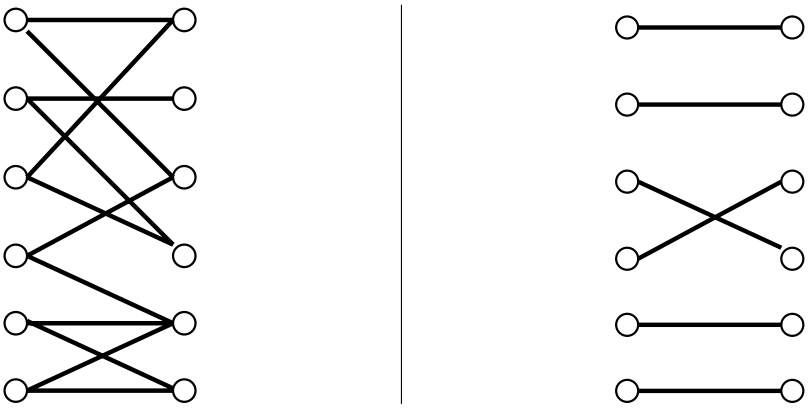

INPUT OUTPUT

15.6 Matching

Input description: A (weighted) graph $G = (V, E)$.

Problem description: Find the largest set of edges E' from E such that each vertex in V is incident to at most one edge of E'.

Discussion: Suppose we manage a group of workers, each of whom is capable of performing a subset of the tasks needed to complete a job. Construct a graph with vertices representing both the set of workers and the set of tasks. Edges link workers to the tasks they can perform. We must assign each task to a different worker so that no worker is overloaded. The desired assignment is the largest possible set of edges where no employee or job is repeated—i.e. , a matching.

Matching is a very powerful piece of algorithmic magic, so powerful that it is surprising that optimal matchings can be found efficiently. Applications arise often once you know to look for them.

Marrying off a set of boys to a set of girls such that each couple is happy is another bipartite matching problem, on a graph with an edge between any compatible boy and girl. For a synthetic biology application [MPC+06], I need to shuffle the characters in a string S to maximize the number of characters that move. For example, *aaabc* can be shuffled to *bcaaa* so that only one character stays fixed. This is yet another bipartite matching problem, where the boys represent the multiset of alphabet symbols and the girls are the positions in the string (1 to $|S|$). Edges link symbols to all the string positions that originally contained a different symbol.

This basic matching framework can be enhanced in several ways, while remaining essentially the same *assignment* problem:

- *Is your graph bipartite?* – Most matching problems involve bipartite graphs, as in the classic assignment problem of jobs to workers. This is fortunate because faster and simpler algorithms exist for bipartite matching.

- *What if certain employees can be given multiple jobs?* – Natural generalizations of the assignment problem include assigning certain employees more than one task to do, or (equivalently) seeking multiple workers for a given job. Here, we do not seek a matching so much as a covering with small "stars." Such desires can be modeled by replicating an employee vertex by as many times as we want her to be matched. Indeed, we employed this trick in the string permutation example above.

- *Is your graph weighted or unweighted?* – Many matching applications are based on unweighted graphs. Perhaps we seek to maximize the total number of tasks performed, where one task is as good as another. Here we seek a maximum *cardinality* matching—ideally a *perfect* matching where every vertex is matched to another in the matching.

  For other applications, however, we need to augment each edge with a weight, perhaps reflecting the suitability of an employee for a given task. The problem now becomes constructing a maximum *weight* matching—i.e. , the set of independent edges of maximum total cost.

Efficient algorithms for constructing matchings work by constructing *augmenting paths* in graphs. Given a (partial) matching M in a graph G, an augmenting path is a path of edges P that alternate (out-of-M, in-M, ..., out-of-M). We can enlarge the matching by one edge given such an augmenting path, replacing the even-numbered edges of P from M with the odd-numbered edges of P. Berge's theorem states that a matching is maximum if and only if it does not contain any augmenting path. Therefore, we can construct maximum-cardinality matchings by searching for augmenting paths and stopping when none exist.

General graphs prove trickier because it is possible to have augmenting paths that are odd-length cycles (i.e. , the first and last vertices are the same). Such cycles (or blossoms) are impossible in bipartite graphs, which by definition do not contain odd-length cycles.

The standard algorithms for bipartite matching are based on network flow, using a simple transformation to convert a bipartite graph into an equivalent flow graph. Indeed, an implementation of this is given in Section 6.5 (page 217).

Be warned that different approaches are needed to solve weighted matching problems, most notably the matrix-oriented "Hungarian algorithm."

Implementations: High-performance codes for both weighted and unweighted bipartite matching have been developed by Andrew Goldberg and his collaborators. `CSA` is a weighted bipartite matching code in C based on cost-scaling network flow, developed by Goldberg and Kennedy [GK95]. `BIM` is a faster unweighted bipartite matching code based on augmenting path methods, developed

by Cherkassky, et al. [CGM+98]. Both are available for noncommercial use from *http://www.avglab.com/andrew/soft.html*.

The First DIMACS Implementation Challenge [JM93] focused on network flows and matching. Several instance generators and implementations for maximum weight and maximum cardinality matching were collected, and can be obtained by anonymous FTP from *dimacs.rutgers.edu* in the directory *pub/netflow/matching*. These include

- A maximum-cardinality matching solver in Fortran 77 by R. Bruce Mattingly and Nathan P. Ritchey.
- A maximum-cardinality matching solver in C by Edward Rothberg, that implements Gabow's $O(n^3)$ algorithm.
- A maximum-weighted matching solver in C by Edward Rothberg. This is slower but more general than his unweighted solver just described.

GOBLIN (*http://www.math.uni-augsburg.de/~fremuth/goblin.html*) is an extensive C++ library dealing with all of the standard graph optimization problems, including weighted bipartite matching. LEDA (see Section 19.1.1 (page 658)) provides efficient implementations in C++ for both maximum-cardinality and maximum-weighted matching, on both bipartite and general graphs.

Blossum IV [CR99] is an efficient code in C for minimum-weight perfect matching available at *http://www2.isye.gatech.edu/~wcook/software.html*. An $O(mn\alpha(m,n))$ implementation of maximum-cardinality matching in general graphs (*http://www.cs.arizona.edu/~kece/Research/software.html*) is due to Kececioglu and Pecqueur [KP98].

The Stanford GraphBase (see Section 19.1.8 (page 660)) contains an implementation of the Hungarian algorithm for bipartite matching. To provide readily visualized weighted bipartite graphs, Knuth uses a digitized version of the Mona Lisa and seeks row/column disjoint pixels of maximum brightness. Matching is also used to construct clever, resampled "domino portraits".

Notes: Lovász and Plummer [LP86] is the definitive reference on matching theory and algorithms. Survey articles on matching algorithms include [Gal86]. Good expositions on network flow algorithms for bipartite matching include [CLRS01, Eve79a, Man89], and those on the Hungarian method include [Law76, PS98]. The best algorithm for maximum bipartite matching, due to Hopcroft and Karp [HK73], repeatedly finds the shortest augmenting paths instead of using network flow, and runs in $O(\sqrt{n}m)$. The Hungarian algorithm runs in $O(n(m + n \log n))$ time.

Edmond's algorithm [Edm65] for maximum-cardinality matching is of great historical interest for provoking questions on what problems can be solved in polynomial time. Expositions on Edmond's algorithm include [Law76, PS98, Tar83]. Gabow's [Gab76] implementation of Edmond's algorithm runs in $O(n^3)$ time. The best algorithm known for general matching runs in $O(\sqrt{n}m)$ [MV80].

Consider a matching of boys to girls containing edges (B_1, G_1) and (B_2, G_2), where B_1 and G_2 in fact prefer each other to their own spouses. In real life, these two would

run off with each other, breaking the marriages. A marriage without any such couples is said to be *stable*. The theory of stable matching is thoroughly treated in [GI89]. It is a surprising fact that no matter how the boys and girls rate each other, there is always at least one stable marriage. Further, such a marriage can be found in $O(n^2)$ time [GS62]. An important application of stable marriage occurs in the annual matching of medical residents to hospitals.

The maximum matching is equal in size to the minimum vertex cover in bipartite graphs. This implies that both the minimum vertex cover problem and maximum independent set problems can be solved in polynomial time on bipartite graphs.

Related Problems: Eulerian cycle (see page 502), network flow (see page 509).