Robert B. Gramacy
Math Senior Seminar
May 21, 2001

Combinatorial Optimization: Matchings

Marriage, Assignment, Augmenting Paths & Network Flows

Matchings:

Description of problem and definitions [4] [6] [1]

Definition 1 (Graphs). *A* graph G *is a pair of sets* $V = V(G) = \{v_1, v_2, \ldots, v_n\}$ *of vertices and* E *(or* $E(G)$*) of edges (written* $G = (V, E)$*) such that the elements of* E *are unordered pairs of elements of some subset of* V *(e.g* $(v_i, v_j) \in E$ *for* $v_i, v_j \in V$*).*

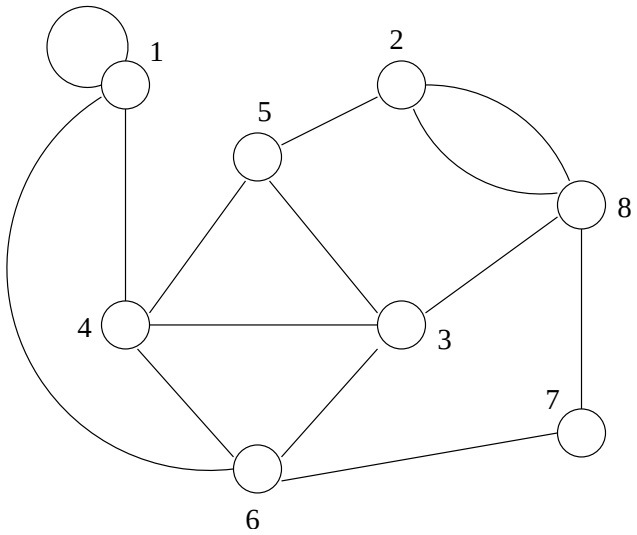
Edge e is said to *saturate* a vertex v if v is one of the vertices in the ordered pair of e.

Strictly speaking, a graph is just a pair of sets defined as above. However, it can be used as a representation of a pairing of like elements; a network of avenues or conduits (edges) on which a commodity can flow or travel between locations (vertices); etc. No restrictions are made as to repeated pairs (parallel edges), or loops (e.g (v_i, v_i)) being in E. A graph which has no edges of these types is called a *simple graph.*

Example 1. *Let* $G = (V, E)$ *be the graph such that the vertices and edges are*

$$V = \{1, 2, 3, 4, 5, 6, 7, 8\}$$
$$E = \{(1,1), (1,4), (1,5), (1,2), (2,5), (2,8), (3,5), (3,6), (3,8), (4,5), (4,6), (6,7), (7,8), (8,2)\}$$

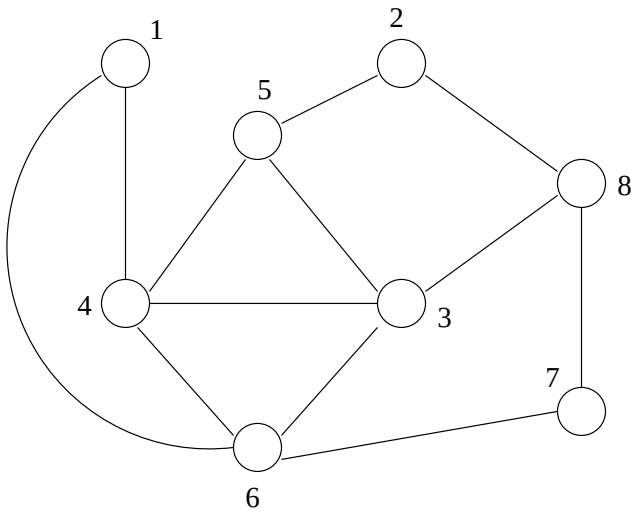
which has the following pictorial representation.



The underlying simple graph has edges

$$E(G_{simple}) = \{(1,4),(1,2),(2,5),(3,5),(3,6),(3,8),(4,5),(4,6),(6,7),(7,8)\}$$

which looks like:



Often a graph is referred to as a figure alone (like the one above), or simply as a listing of vertices and edges.

Definition 2 (Matchings). *A* matching M *(or $M(G)$ when we want to be explicit which graph we are referring to) is a subset of the edges E ($M \subseteq E$) such that no two edges of M (or a singe edge in the case of a loop) saturate the same vertex.*

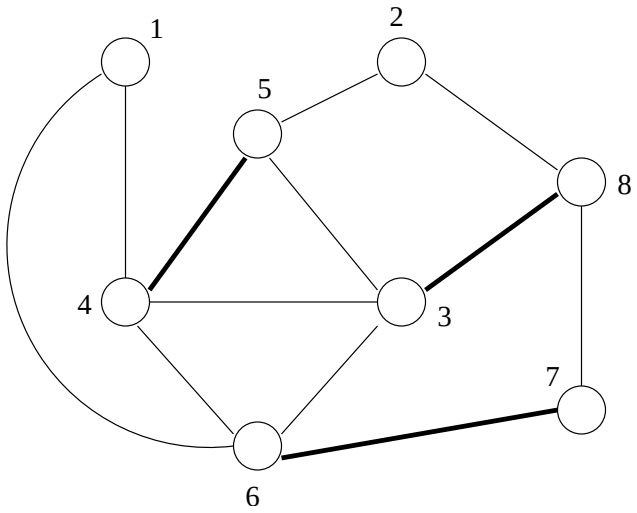
In the presence of a matching we overload the term "saturate" to mean that a particular vertex is (or isn't) saturated (unsaturated) by M. We will often be interested in the number of edges contained in a matching M. The cardinality of the matching is the number of edges contained in it; written $|M|$.

Matchings can be thought of as an exclusive pairing of the vertices as allowed by the edges of E. The definition of a matching does not allow a vertex to belong to more than one such pair. Typically we think of the graph G for which we want to describe a matching as being simple since a parallel edge or loop could never be an edge of a matching. Clearly, if the graph is not simple, dealing with the underlying simple graph is equivalent with respect to a matching.

Example 2. *For the underlying simple graph of G given in Example 1*

$$M(G) = \{(3,8),(4,5),(7,6)\}$$

is a matching of cardinality 3, which has the following pictorial representation:



Definition 3 (Maximal Matchings). *A matching M is called* maximal *if for all other matchings M' on the same graph we have that $|M'| \leq |M|$.*

We say that M is a *perfect* matching if $|M| = \lceil |V|/2 \rceil$ (i.e. the edges of M saturate all of the edges of V). A perfect matching is necessarily a maximal matching. The converse need not be true. Note that if $|V(G)|$ is odd, then G cannot admit a perfect matching. The graph of Example 2 does admit a perfect matching, namely–

$$M = \{(1,4),(2,5),(3,6),(7,8)\}.$$

There are at least two others. The problem of finding maximal matchings will get a great deal of attention in our following discussion. Consider the following practical application as motivation.

Problem 1 (Maximal Matchings). *During World War II, many airplane pilots from occupied countries fled to Britain to enlist in the Royal Air Force. In certain squadrons, each plane sent aloft by the RAF required a pilot with a navigator whose navigational skills and language skills were compatible. The RAF was interested in sending as many planes aloft at one time as possible. [4] [1]*

If we consider pilots and navigators as vertices, and edges exist between these vertices when the pilots/navigators speak the same language, then an optimal solution to the above problem is a maximal matching.

Clearly, if $|M| = \lceil |V|/2 \rceil$, then M is a perfect matching, and therefore maximal. But how do we know if a given matching M is maximal when $|M| \neq \lceil |V|/2 \rceil$? Or, equivalently, how do we know when there is a "better" (larger) matching? Even when the size of the vertex set is even the graph may not admit a perfect matching. Since our goal for much of this paper will be to to produce algorithms for finding maximal matchings, it would be helpful to have a criteria that is necessary and sufficient for knowing whether or not a given matching M is maximal. To establish such criteria we will need a few more definitions.

Definition 4 (Path). *A* path *P in a graph $G = (V, E)$ is a sequence of vertices and edges*

$$P = \langle v_1, e_1, v_2, e_2, v_3, \ldots, v_{k-1}, e_{k-1}, v_k \rangle$$

such that $v_i \in V,\ \forall i,\ e_i = v_i, v_{i+1} \in E$ *and* $v_i \neq v_j$ *when* $i \neq j$.

When the edges are implicit, as in the case of a simple graph, we usually omit these from the path P. This gives

$$P = \langle v_1, v_2, \ldots, v_k \rangle.$$

Alternatively, a path can also be represented solely by its edges. Often we speak of the vertices or edges along a path as $V(P)$ or $E(P)$ respectively. This notation comes from the fact that P is itself a graph (or subgraph).

A *cycle* is a path $P = \langle v_1, v_2, \ldots, v_k \rangle$, together with the edge (v_k, v_1), written

$$C = \langle v_1, v_2, \ldots, v_k, v_1 \rangle.$$

Definition 5 (Alternating Path). *An* alternating path *in a graph* $G = (V, E)$ *with respect to a matching* M *is a path whose edges are alternately in* M *and* $E - M$ *(not in* M*). [4]*

Definition 6 (Augmenting Path). *An* augmenting path *with respect to a matching* $M(G)$ *is an alternating path whose* origin*– first vertex in the path– and* terminus*– last vertex in the path– are both not saturated by* M*. [4]*

$P = \langle 2, 8, 3, 5, 4, 1 \rangle$ is an augmenting path with respect to the matching and the graph given in Example 2. Note that an augmenting path necessarily has an even number of vertices (and therefore odd number of edges). The following Lemma provides an answer to part of the question posed previously. Namely, how do we know if a given matching M is maximal?

Lemma 1 (Augmenting Paths for Larger Matchings). *Let* $E(P)$ *be the set of edges which make up the augmenting path* $P = \langle v_1, v_2, \ldots, v_{2k} \rangle$ *in a graph* $G = (V, E)$ *with respect to a matching* M*. Then* $M' = M \oplus E(P) = (M - E(P)) \cup (E(P) - M)$ *is a matching with respect to* G *where* $|M| = |M| + 1$*. [6]*

Note that the result of the symmetric difference operation above is simply a way of describing a set containing the edges of P which are not in M, along with those edges in M which are not in P. (*)

Proof. We first verify that no two edges of M' have any vertex in common (i.e. that it is indeed a matching). Suppose, for the sake of contradiction, that two edges of M', say e and e' share a vertex in common. We then have one of the following three cases:

i. both e and e' came from $M - E(P)$ (edges of the original matching not on the path P). But this cannot happen since we assumed that M is a matching.

ii. both e and e' came from $E(P) - M$ (the edges were on the path, but not in the matching). But then P is an alternating path, and so if neither e or e' was an element of M they cannot have a vertex in common.

iii. $e \in M - E(P)$ and $e' \in E(P - M)$. Therefore, e' is on the path P, but not an element of the matching M. This means that e' can be written like (v_{2i-1}, v_{2j}). Without loss of generality, let v_{2j} be the vertex that e' has in common with e. But since P is an augmenting path, v_{2j} must be adjacent to some edge $e'' \in M \cap E(P)$. But then $e \in M - E(P)$ implies that $e'' \neq e$, however they are adjacent the same node v_{2j}.

All three of the cases cases contradict that M is a matching. Therefore, we conclude that M' must be a matching.

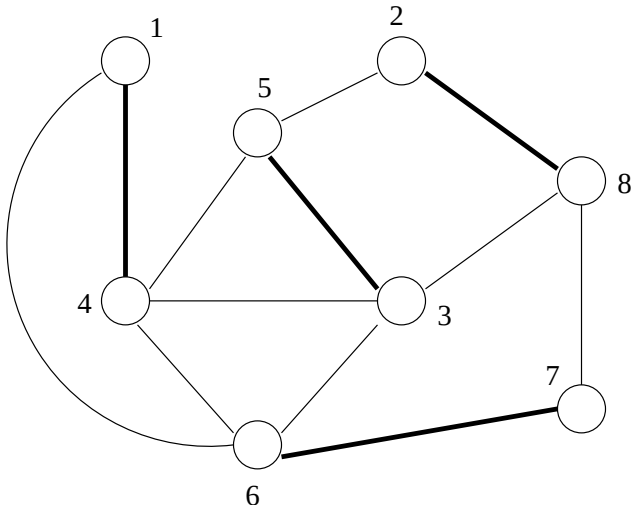
In light of (*), and the fact that since P is an augmenting path, both of its ends must be unsaturated. From this it follows that there is exactly one more edge on the path P which is not in M, than there are edges on the path which are in M. From this it follows that $|M'| = |M| + 1$. □

This lemma gives us leverage on generating a "better" matching than one we have already. That is, if we have a matching (for starters, a single edge will do), then Lemma 1 provides an algorithm for generating a larger matching, provided we can find an augmenting path.

Applying the above lemma to the augmenting path $P = \langle 2, 8, 3, 5, 4, 1 \rangle$ with respect to the matching and the graph given in Example 2 yields the (perfect) matching

$$M = \{(6, 7), (1, 4), (5, 3), (2, 8)\}$$

which corresponds to the following picture.



What happens if we can't find an augmenting path? As it turns out, it is equivalent to ask the following question instead: Although $|M| = \lceil |V|/2 \rceil$ is a sufficient condition for a maximal matching, how else do we know to stop looking for better matchings? The following theorem provides answers to both of these questions.

Theorem 1 (Matching Maximal ⇔ No Augmenting Path). *A matching M in a graph $G = (V, E)$ is maximum if and only if there are no augmenting paths in G with respect to M.*

Proof. The first direction ($\Rightarrow$) is exactly what we showed in Lemma 1. For the other direction ($\Leftarrow$) suppose, for the sake of contradiction, that there are no augmenting paths in G with respect to M, and that there is a matching $M'(G)$ such that $|M'| > |M|$. Consider the edges of $E' = M \oplus M' = (M - M') \cup (M' - M)$ which induce a subgraph $G'(V, E')$ of $G(V, E)$. **Note that this graph need not be connected.** Since two edges of a matching cannot be incident on the same vertex, all of the vertices of $V(G')$ have degree 2 or less. The only case when a vertex v can have degree 2 is when one of the edges incident with it came from M, and the other from M'. From this we have that each **connected component** of G' is either a path or a cycle, whose edges are alternately in M and M'. Note that a cycle whose edges are alternately in M and M' must have even length, and so must have as many edges from M as from M'. Therefore, if $|M'| > |M|$ one of the path-components P in G' must have more edges from M' than M. But then P must then be an augmenting path with respect to M in G, contradicting our assumption that there was no such path. The theorem follows. □

Now all we need is a routine which finds augmenting paths! Unfortunately, in general this can be quite difficult. Therefore, following we will first discuss methods for finding matchings in a special type of graphs, which can help solve a rather large class of interesting combinatorial optimization problems. We return to the general matching problem in the final section of the paper. After the following discussion– with maturity developed in the sections to come– we will have much of the artillery necessary to attack the general problem.

Bipartite Matchings:
Assignment [4], Augmenting Paths [6], Alternating Search Trees [1][6],
Di-Graph Reduction for Searching [6]

Consider the following problems as motivation:

Problem 2 (Marriage). *If we have a finite set of girls whom each have several boyfriends, under what conditions can each girl marry one of her boyfriends (assuming that no girl can marry more than one boy). [4]*

Problem 3 (Personnel / Task Assignment). *In a high school, n teachers $x_1, x_2, \ldots, x_n$ are available for n classes $y_1, y_2, \ldots, y_n$, each teacher being qualified to teach one or more of these classes. Can all of the teachers be assigned (one per class) to classes for which they are qualified? [4]*

In each of these problems there are two distinct groups V_1 (girls, teachers) and V_2 (boys, classes) between which there are "relationships" described by pairs in $V_1 \times V_2 = E$ (and it doesn't make sense to have relationships in $V_1 \times V_1$ (at least not without political action) or in $V_2 \times V_2$). Situations such as these can be modeled by a special kind of graph, where feasible solutions to these problems are matchings. Finding maximal solutions to matching problems formulated in this way will prove much easier than finding those associated with arbitrary graphs.

Definition 7 (Bipartite Graphs). *A* bipartite graph *is a graph $G = (V_1 \cup V_2, E)$ such that each edge in E has one end in V_1 and the other in V_2 (and $V_1 \cap V_2 = \emptyset$).*

Example 3 (A Bipartite Graph). *Let*

$$V_1 = \{1, 2, 3, 4\} \qquad V_2 = \{a, b, c, d, e\}$$

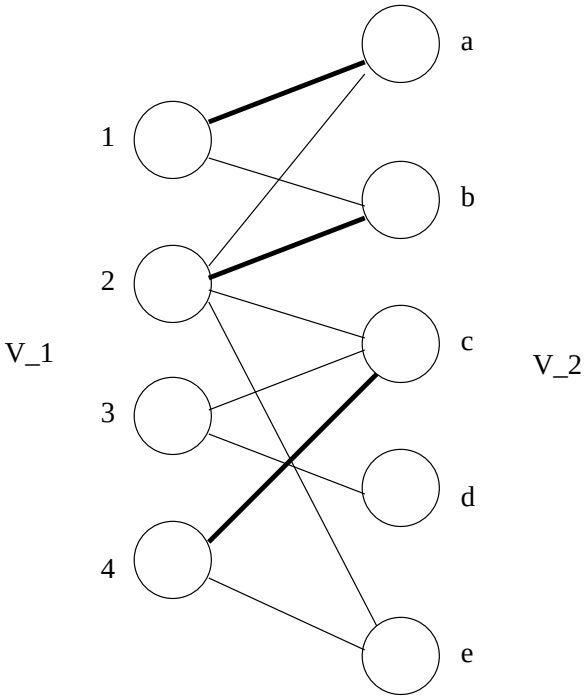
$$E = \{(1, a), (1, b), (2, a), (2, b), (2, c), (2, e), (3, c), (3, d), (4, c), (4, e)\}$$

which is depicted by the following picture.

A matching on a bipartite graph is defined no differently than for an arbitrary graph. The matching

$$M = \{(1, a), (2, b), (4, c)\}$$

is also illustrated in the following diagram associated with Example 3. Note that this example does not admit a perfect matching, and furthermore that M itself is not maximal.



We now produce several algorithmic techniques for finding maximal matchings on bipartite graphs– thereby producing strategies to find maximal solutions to assignment problems (like the ones above). As already mentioned, the problem essentially reduces to one of finding augmenting paths. Our first method will use this strategy. With some modification, we shall be able to generalize this strategy to find maximal matchings in arbitrary graphs. The second method will employ is a technique which involves *Network Flows*. This technique generalizes nicely to solve the *optimal* (weighted) bipartite matching problem.

Algorithm 1 (Search Trees for Bipartite Matchings). *Our first technique is a variation of a class of* label-setting *algorithms known as* breadth-first search. *We will choose a starting node $p \in V_1$ which is unsaturated by the current matching M (initially $M = \emptyset$) and attempt to find an augmenting path to some other node $q \in V_2$ by exploring all possible $p \rightsquigarrow q$ paths, $\forall q \in V_2$, by maintaining a* search tree. *Once an augmenting path is found, we proceed by augmenting M as in Lemma 1. If we find no such path, then this vertex cannot be covered by a larger matching, and so we try a different $p \in V_1$. If we are unsuccessful in finding an augmenting path from any $p \in V_1$ to any $q \in V_2$, the algorithm terminates. [1]*

Search Trees:
(Given a node p we explore "alternating" paths (creating an *alternating tree*) in search of an augmenting path.)

As we explore the graph we maintain two sets, or labelings: E (for even), and O (for odd); for nodes seen so far. We refer to the node p as the *root* of the tree. A (captured) node i in the tree is labeled *even* or *odd* depending on whether the number of arcs from p to this vertex i on the path $p \rightsquigarrow i$ in the alternating tree is even or odd. Thinking of all of the paths in the tree as the beginning of an augmenting path, we will also have that matched edges go from odd labeled vertices to even labeled vertices. Uncaptured nodes are unlabeled.

The algorithm maintains a list L of labeled (leaf) nodes from which we will explore next. Initially

$$E = L = \{p\} \qquad \text{and} \qquad O = \emptyset.$$

Definition 8 (Adjacency Lists). *For each $i \in V$ ($V = V_1 \cup V_2$ if the graph in bipartite) let the* adjacency list *for i be*

$$A(i) = \{j : (i, j) \in E\}.$$

Growing the Search Tree: (Algorithm 1 continued..)

- Choose any node $i \in L$ (initially $i = p$).
- If $(i \in E) \Rightarrow$ scan $A(i)$ and assign odd labels for those $j \in A(i)$ not already labeled ($j \notin E \cup O$). (By ignoring already labeled j's we miss nothing but redundant alternating paths.) If j is not saturated by M, then $p \rightsquigarrow j$ is an augmenting path, and we can (successfully) terminate the search procedure. Otherwise, we add each j to O and to L.
- else, if $(i \in O) \Rightarrow$ examine vertex i's unique matched arc $(i, j) \in M$ (j is necessarily unlabeled). Add j to O and to L. Do nothing if such an arc does not exist.
- We then remove i from L (since it is no longer a leaf). If $L = \emptyset$, (unsuccessfully) terminate this search. Otherwise, repeat.

If an augmenting path was not found, we choose another unsaturated $i \in V$ and repeat the above procedure, until no such untried i remain. □

The following result shows that if at any time during the above algorithm we fail to find an augmenting path from any node $i \in V$, we will never again find an augmenting path from i (with respect to later matchings).

Theorem 2 (Well-Ordered Augmenting Paths). *Suppose that there is no augmenting path starting from an unsaturated vertex $i \in N(G)$ in an arbitrary graph G with respect to a matching M. If P is an augmenting path whose endpoints are unsaturated $u, v \neq i \in V$, then there will be no augmenting path from i with respect to $M' = M \oplus E(P)$ either. [6]*

Proof. Consider i and P as above, and suppose (for the sake of contradiction) that there is an augmenting path Q from i with respect to $M \oplus E(P)$. We consider two cases:

i. Q has no node in common with P. In this case, $M' = M \oplus E(P)$ changes nothing as far as Q is concerned, which means that Q must have been an augmenting path with respect to M. This contradicts our assumption.

ii. Q has at least one node in common with P. Here, let $Q = \langle i = v_1, v_2, \ldots, v_k \rangle$ where i and v_k are unsaturated by M. Furthermore, let u_j be the first vertex which Q has in common with P. Therefore, $(v, u_j) \in M$. Note that any edge in Q, but not P, with an end as u_j must be unmatched, since one of the ends of u_j in P is saturated and P is an augmenting path with respect to M. One of the two sub-paths into which u_j divides P must end with an edge $(v, u_j) \notin M \oplus E(P)$. Let P_j denote this portion of P, and Q_j denote the portion of Q from i up to u_j. But then the combined path $Q_j \cdot P_j$ ($\cdot$ is the concatenation operation) is an augmenting path from i with respect to M (since both ends of P were unsaturated with respect to M). This is a contradiction.

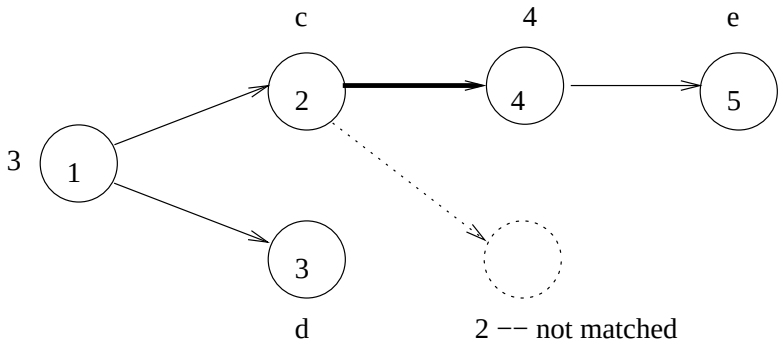
Thus we conclude that if there is no augmenting path from a vertex $i \in V(G)$ there can be no augmenting path from i in G with respect to $M \oplus E(P)$. □

Inductively [6], we have that if at some stage there is no augmenting path from a node $i \in N$, then there will never be an augmenting path from i. Notice that the above was shown for an arbitrary graph G, bipartite or not! We will take advantage of this result later when we consider finding augmenting paths in arbitrary graphs.

Example 4 (Algorithm 1 Illustration). *For this example we take the same graph G and matching M as in Example 3. The only unsaturated vertex in V_1 is $p = 3$. Below is the history of the sets L, E, O, along with a pictorial representation of the search tree.*

$E_1 = \{1\}$	$L_1 = \{3\}$	$O_1 = \emptyset$	
$E_2 = \{1\}$	$L_2 = \{c, d\}$	$O_2 = \{c, d\}$	
$E_3 = \{1, 4\}$	$L_3 = \{d, 4\}$	$O_3 = \{c, d\}$	
$E_4 = \{1, 4\}$	$L_4 = \{4\}$	$O_4 = \{c, d\}$	*(E, O unchanged)*
$E_5 = \{1, 4\}$	$L_5 = \{e\}$	$O_5 = \{c, d\}$	
$E_6 = \{1, 4\}$	$L_6 = \{4\}$	$O_6 = \{c, d, e\}$	*($4 \rightsquigarrow e$ is augmenting)*

The numbers i inside the vertices in the tree indicate the correspondence between the versions of the sets E_i, L_i, and O_i.



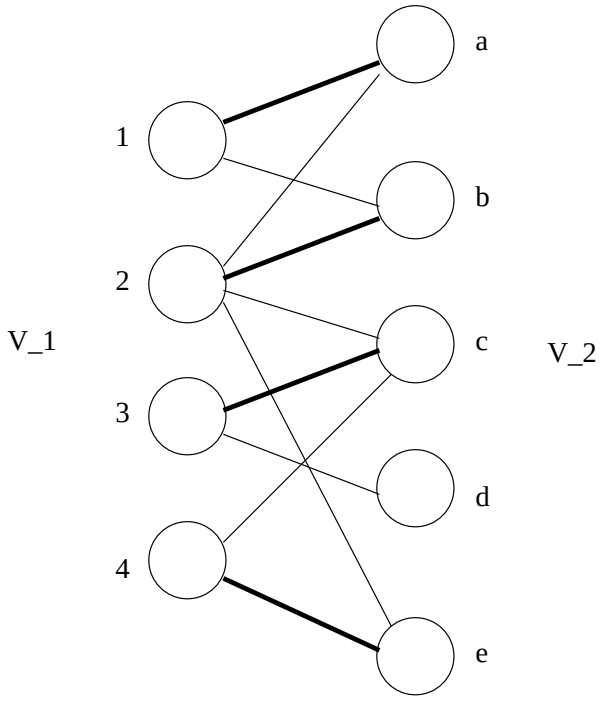
Performing the operation of Lemma 1 on the augmenting path

$$P = \langle 3, c, 4, e \rangle$$

produces the following maximal matching

$$M = \{(1, a), (2, b), (3, c), (4, e)\}$$

which is depicted below. This matching is maximal because there are no longer any unsaturated vertices in V_1. We prove this general rule below.



Theorem 3. *Algorithm 1 correctly solves the bipartite matching problem on graph* $G = (V_1 \cup V_2, E)$ *in* $O(|V_1| \cdot |E|)$ *time. [6]*

$O(|V_1| \cdot |E|)$ can be interpreted to mean that there exists a constant c such that the algorithm takes no more than $c|V_1| * |E|$ steps.

Proof. The algorithm terminates when it finds that there is no augmenting path from an unsaturated vertex in V_1 (to another node necessarily in V_2), with respect to a matching M. Therefore, by Theorem 1 we conclude that M is maximal.

In order to establish the time bound, note that a maximal matching can have no more than $|V_1|$ edges. For this reason the algorithm need only search for at most $|V_1|$ augmenting paths. Finding an augmenting path from a particular unsaturated vertex $p \in V_1$ might potentially examine every edge of $E(G)$. Therefore the search procedure, in total for all p, takes time in $O(|V_1| \cdot |E|)$. The operation of augmenting M (essentially performing the operation of symmetric difference) might also examine each edge of E, and similarly this could take time in $O(|V_1| \cdot |E|)$. The theorem follows. □

Note that a more clever version of our algorithm would first determine which of V_1 and V_2 was the larger, and pick p from this set. With this modification the algorithm would take time in $O(\min\{|V_1|, |V_2|\} \cdot |E|)$. [6]

An alternative approach to the technique described above for finding augmenting paths (of labelings odd, even, etc) involves a preprocessing step. The preprocessing step eliminates the need for three different labelings of vertices by converting bipartite G into a directed graph with respect to the current matching M.

Definition 9 (Directed Graphs). *A* directed graph *(or di-graph, written $D = (V, A)$) is a pair of sets $V = V(D) = \{v_1, v_2, \ldots, v_n\}$ of vertices and A (or $A(G)$) of arcs such that the elements of A are* ***ordered*** *pairs of elements of some subset of V (e.g $(v_i, v_j) \in A$ for $v_i, v_j \in V$ which describes an arc whose origin is v_i and terminus is v_j). The relationships between vertices described by the arc set are typically asymmetric.*

We construct the directed graph $D = (V, A)$ from bipartite $G = (V_1 \cup V_2, E)$ as follows. Let $V(D) = V_1$. For all pairs of nodes $v_1, w \in V_1$, let $(v_1, w) \in A(D)$ if and only if w can be reached from v_1 through intermediate node $v_2 \in V_2$ and $(v_2, w) \in M(G)$. Furthermore, keep track of those $v \in V(D)$ which are adjacent to unmatched vertices in the original $V(G)$.

After this construction, the search for augmenting paths reduces to performing a textbook [5] breadth-first search of D from unsaturated vertices in $V(D)$. If ever a path is found to reach $w \in V(D)$ such that w has an edge to an unsaturated vertex in $V(G)$ (that is, a vertex for which there an edge to $v_2 \in V_2(G)$, but there is no matched edge back to $V_2(G)$) we have found an augmenting path. Thus, the work of labeling, etc, performed in Algorithm 1 is shifted to the straightforward creation of the di-graph outlined above. This will be useful again later when we consider the task of finding augmenting paths with respect to matchings in arbitrary graphs.

We conclude this line of discussion to proceed with the description of a technique which yields the best known time bounds for producing solutions to the bipartite matching problem.

Network Flows for Bipartite Matchings:
Network Flows in General [1], Simplex Method [6][3], Ford & Fulkerson Algorithm [1][4],
Optimal (Weighted) Matchings & Successive Shortest Paths Algorithm [1]

Before showing how the bipartite matching problem reduces to a Network Flow problem, some definitions are in order.

Definition 10 (Network). *A* network $G = (N, A)$ *is a directed graph with nodes (or vertices)* N*, and directed arcs* A*, which both may have numerical a value or function associated with them (supply, demand, cost, capacity, etc).*

For now we will be interested in networks whose arcs have a capacity function associated with them, and whose nodes are associated a supply or demand. That is, each arc $(i, j) \in A$ is associated a nonnegative capacity $u(i, j)$. Capacity constraints can be thought of as a function $u : A \to \aleph$ (natural numbers). Every vertex $i \in N$ is assigned a number $b(i)$, which indicates whether it is a *supply*, *demand*, or neither, in which case it is referred to as an intermediate node. More precisely,

$$b(i) = \begin{cases} < 0 & \text{if } i \text{ is a source (a supplier)} \\ > 0 & \text{if } i \text{ is a sink (a demander)} \\ = 0 & \text{otherwise (an intermediate node)} \end{cases}$$

We consider two situations. In either case we require that in the formulation of a network that supply/demand nodes be designated a' priori. However, we have a choice as to whether the designation of how much a node will supply or demand is to be part of the network specification, or part of the solution. We will see examples of each.

A Network where nodes are a' priori given specific amounts to supply or demand is *well-defined* if the total supply equals the total demand. In other words, we must have that

$$\sum_{\forall i \in N} b(i) = 0. \tag{1}$$

This is sometimes referred to as *flow conservation.* After an example we will consider two similar problems involving Networks, and then show how solving one of them can help us solve the bipartite matching problem. Afterwards, we proceed with a general discussion of two techniques for solving these Network Flow problems.

Example 5. *[3] In this example there are no intermediate nodes.*

$$N = \{1, 2, 3, 4, 5, 6\}$$
$$A = \{(1, 2), (1, 3), (1, 5), (2, 3), (4, 2), (4, 3), (5, 3), (5, 4), (5, 6), (6, 3), (6, 4)\}$$

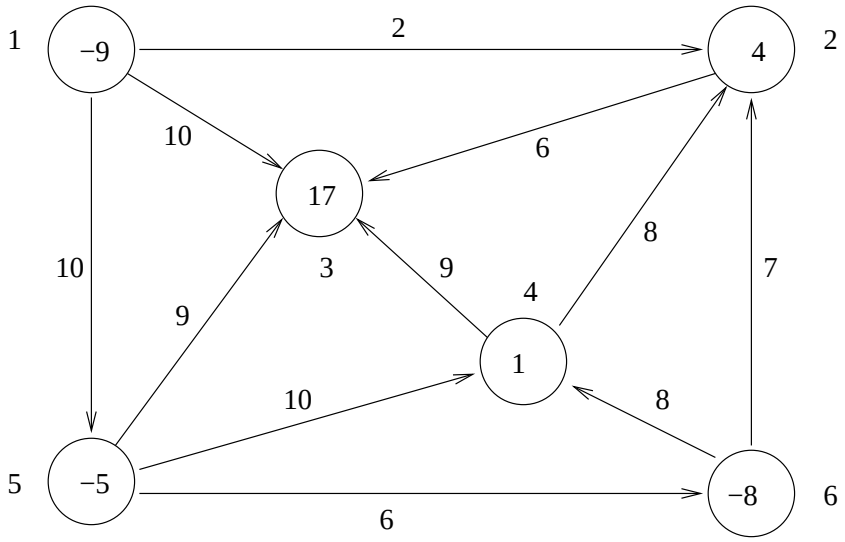
with a' priori supplies and demands

$$b(1) = -9 \qquad b(2) = 4 \qquad b(3) = 17 \qquad b(4) = 1 \qquad b(5) = -5 \qquad b(6) = -8$$

capacity constraints

$$u(1,2)=3 \quad u(1,3)=5 \quad u(1,5)=1 \quad u(2,3)=1 \quad u(4,2)=4 \quad u(4,3)=1$$
$$u(5,3)=6 \quad u(5,4)=1 \quad u(5,6)=1 \quad u(6,2)=1 \quad u(6,4)=1$$

which corresponds to the following picture. The names of the nodes in N can be found next to each node, and its supply/demand can be found within the node. Next to each edge is its capacity.



Our first problem involving network flows can be described directly in terms of the above example.

Problem 4 (Transshipment). *Is there a valid flow which satisfies the a' priori specified supplies and demand described by $b(\cdot)$ on a network $G=(N,A)$?*

A *flow* is the assignment of some amount of commodity along a subset of the arcs of $G=(N,A)$, subject to the capacity constraints, and a set of *mass-balance constraints*. That is, a flow x is said to be *valid* if it satisfies the capacity constraints

$$0 \le x(i,j) \le u(i,j) \qquad \forall (i,j) \in A \tag{2}$$

and

$$\sum_{j:(i,j)\in A} x(i,j) - \sum_{j:(j,i)\in A} x(i,j) = b(i) \qquad \forall i \in N \tag{3}$$

which are referred to as the mass-balance constraints. This simply means that if a node $i \in N$ is an intermediate node, then the inflow into that node must equal its outflow. Otherwise, if the node i is a supply, it can supply no more than $b(i)$. If it is a sink, the inflow can be no more then $b(i)$.

The following problem is similar to the Transshipment problem. However, it is an optimization problem, which will make it more useful to us for solving the bipartite matching problem.

Problem 5. *Given a set of sources $S \subset N$ and sinks $T \subset N$ such that $S \cap T = \emptyset$ what is the maximum amount of flow that can be sourced at nodes from S and sinked at nodes in T which satisfies the capacity and mass-balance constraints?*

In other words, supplies, demands and intermediate nodes are designated a' priori, but specific values are not assigned. We wish to learn what the maximum amount of commodity is that can be sent between sources and sinks which does not violate the capacity constraints. The designation of the resulting amount sourced and sinked at each node, and the actual flow along each arc, describes an optimal solution.

The combinatorial optimization problem can be phrased more precisely as is as follows.

maximize

$$\sum_{i \in S} b(i) \tag{4}$$

subject to

$$\sum_{i \in S} b(i) = -\sum_{i \in T} b(i) \tag{5}$$

$$b(i) = 0 \quad \forall i \in N - (S \cup T) \tag{6}$$

$$\sum_{j:(i,j) \in A}^{n} x(i,j) - \sum_{j:(i,j) \in A}^{n} x(j,i) = b(i) \quad \forall v_i \in N \tag{7}$$

$$0 \leq x(i,j) \leq u(i,j) \quad \forall (i,j) \in A. \tag{8}$$

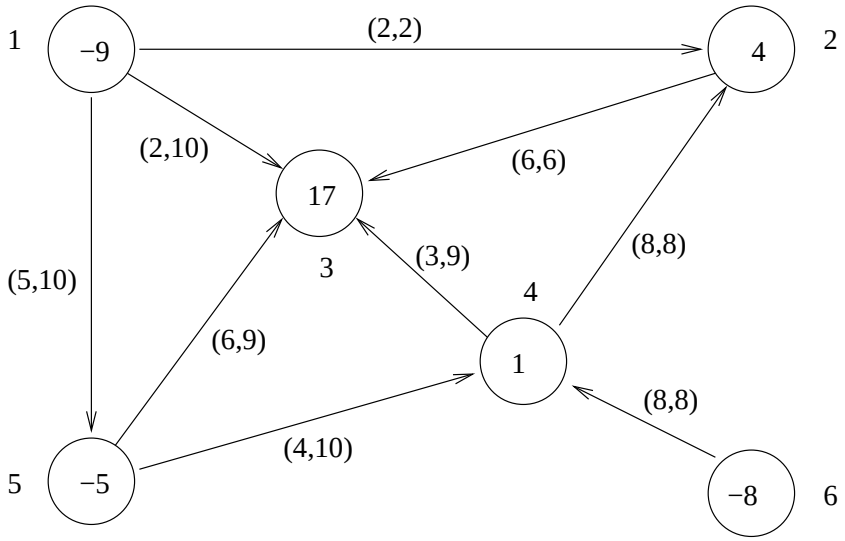
The constraints (5 - 8) can be summarized as follows: (5) the total inflow at the sources of S must equal the total outflow of the source at T; (6) for all other intermediate nodes the inflow must equal the outflow; (7) the flow itself must satisfy the mass-balance constraints; (8) and also must satisfy the capacity constraints. Flows which satisfy (5 - 8) are said to be *feasible*. Feasible flows which cause (4) to obtain a maximal value are said to be *optimal*. Sometimes we will write x or u in place of $x(i,j)$ and $u(i,j)$ for all $(i,j) \in A$ when referring to the flow on an entire network, or a set of capacity constraints.

Although it may seem cumbersome to phase an otherwise straight-forward graph/network optimization problem in this way, such a formulation can have great payoffs! The ability to phrase an arbitrary combinatorial optimization problem as a Linear programming (LP) problem allows us to take advantage of some nice results (e.g. techniques for solving them, like the Simplex Method). There are many existing LP solvers available commercially (`Maple`, `Mathematica`, and `MatLab` all have LP packages). There are even free ones (`Lindo`) available to students.

In the following graphs, arcs are left out which do not have any flow on them. For the arcs which are presented, each is associated a pair of numbers which should be read (flow, capacity). The actual input given to `Mathematica`, and the resulting output follows on the next two pages.

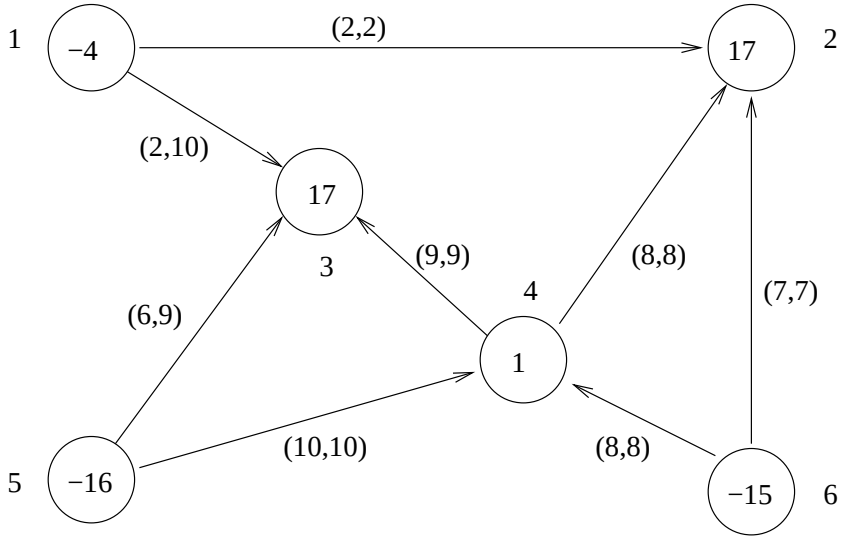
Transshipment Problem:

Note that both the mass-balance and capacity constraints are satisfied.



Maximum Flow Problem:

For this Maximum Flow Problem, the sources and sinks were taken to be the same as in the Transshipment problem (without constraint). Note that both the mass-balance and capacity constraints are satisfied. Note also that we were able to source and sink much more than in the Transshipment Problem, yet less arcs were used. The supply/demand within the nodes are as a result of the optimal flow. The *value* of a maximum flow is the total amount sourced or sinked (note that by flow conservation these quantities should be equal).



Problem 4, on Exercise 5: **Transshipment**

```
ConstrainedMax[
 e[12] + e[13] + e[15] - e[15] + e[53] + e[54] + e[56] - e[56] + e[64] + e[62],

 {(*subject to *)

  (* mass balance constraints *)
  -e[12] - e[15] - e[13] == -9,
  e[13] - e[23] + e[42] + e[62] == 4,
  e[13] + e[23] + e[43] + e[53] == 17,
  -e[43] - e[42] + e[64] + e[54] == 1,
  e[15] - e[53] - e[54] - e[56] == -5,
  e[56] - e[64] - e[62] == -8,

  (* capacity constraints *)
  e[12] >= 0, e[12] <= 2,
  e[13] >= 0, e[13] <= 10,
  e[15] >= 0, e[15] <= 10,
  e[23] >= 0, e[23] <= 6,
  e[42] >= 0, e[42] <= 8,
  e[43] >= 0, e[43] <= 9,
  e[53] >= 0, e[53] <= 6,
  e[54] >= 0, e[54] <= 10,
  e[56] >= 0, e[56] <= 6,
  e[64] >= 0, e[64] <= 8,
  e[62] >= 0, e[62] <= 7},

(* tell mathematica which variables *)
{e[12], e[13], e[15], e[23], e[42], e[43], e[53], e[54], e[56], e[64], e[62]}]
```

```
{22, {e[12] → 2, e[13] → 2, e[15] → 5, e[23] → 6, e[42] → 8, e[43] → 3, e[53] → 6,
  e[54] → 4, e[56] → 0, e[64] → 8, e[62] → 0}}
```

Problem 5, on Exercise 5: **Maximum Flow**

```
ConstrainedMax[
 e[12] + e[13] + e[15] - e[15] + e[53] + e[54] + e[56] - e[56] + e[64] + e[62],

 {(*subject to *)

  (* mass balance constraints *)
  -e[12] - e[15] - e[13] <= 0,
  e[13] - e[23] + e[42] + e[62] >= 0,
  e[13] + e[23] + e[43] + e[53] >= 0,
  -e[43] - e[42] + e[64] + e[54] >= 0,
  e[15] - e[53] - e[54] - e[56] <= 0,
  e[56] - e[64] - e[62] <= 0,

  (* Total Ouflow*)
  -e[12] - e[15] - e[13]
   + e[15] - e[53] - e[54]
   + e[56] - e[64] - e[62]
  (* Must Equal Total Inflow *)
   + e[13] - e[23] + e[42] + e[62]
   + e[13] + e[23] + e[43] + e[53]
   - e[43] - e[42] + e[64] + e[54]  == 0

  (* capacity constrints *)
  e[12] >= 0, e[12] <= 2,
  e[13] >= 0, e[13] <= 10,
  e[15] >= 0, e[15] <= 10,
  e[23] >= 0, e[23] <= 6,
  e[42] >= 0, e[42] <= 8,
  e[43] >= 0, e[43] <= 9,
  e[53] >= 0, e[53] <= 6,
  e[54] >= 0, e[54] <= 10,
  e[56] >= 0, e[56] <= 6,
  e[64] >= 0, e[64] <= 8,
  e[62] >= 0, e[62] <= 7},

(* tell mathematica which variables *)
{e[12], e[13], e[15], e[23], e[42], e[43], e[53], e[54], e[56], e[64], e[62]}]
```

```
{35, {e[12] → 2, e[13] → 2, e[15] → 0, e[23] → 0, e[42] → 8, e[43] → 9, e[53] → 6,
  e[54] → 10, e[56] → 0, e[64] → 8, e[62] → 7}}
```

From Bipartite Graphs To Network Flows

We now have the artillery to attack the question at hand. How can a bipartite matching problem be turned into a Network Flow problem? Following immediately, we will show how we can transform a bipartite graph $G = (V_1 \cup V_2, E)$ into a network $G' = (N, A)$ with a single source and a single sink. Then, once we solve a maximum flow problem on G' we show how to translate the results back to obtain the maximal matching in G.

The construction of the network $G = (N, A)$ from bipartite $G = (V_1 \cup V_2, E)$ proceeds as follows.

- Let $N(G) = V_1 \cup V_2 \cup \{s, t\}$.
- Let $A(G)$ contain arcs
  - i. (s, v_1) for all $v_1 \in V_1$ with unit capacity;
  - ii. (v_2, t) for all $v_2 \in V_2$ with unit capacity;
  - iii. (v_1, v_2) for all $v_1 \in V_1$ and $v_2 \in V_2$ such that $(v_1, v_2) \in E$ with infinite capacity.

Lets see what this means for the bipartite graph that we presented in Example 3.

Example 6 (Bipartite to Network Transformation). *Taking the bipartite graph* $G = (V_1 \cup V_2, E)$ *from Example 3 we produce*

$$N = \{1, 2, 3, 4\} \cup \{a, b, c, d, e\} \cup \{s, t\}$$

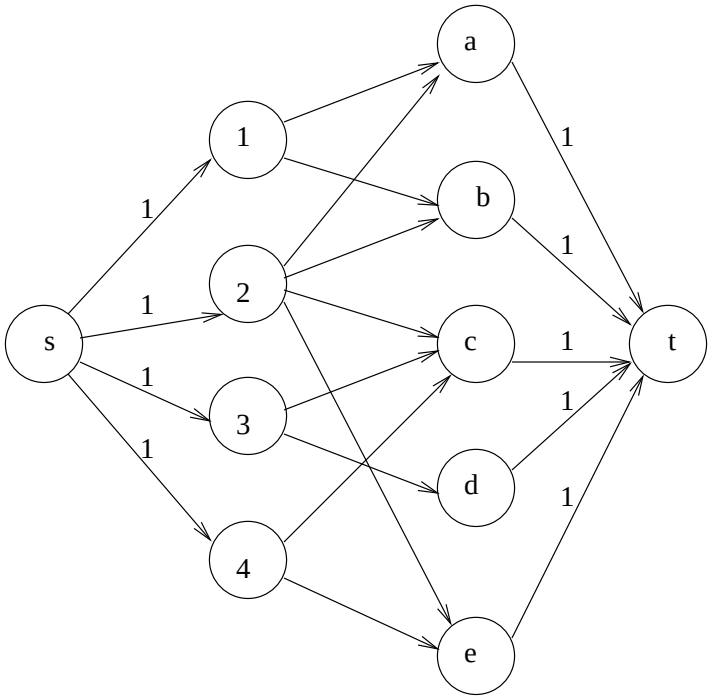
and

$$\begin{aligned} A = &\{(1, a), (1, b), (2, a), (2, b), (2, c), (2, e), (3, c), (3, d), (4, c), (4, e)\} \\ &\cup \{(s, 1), (s, 2), (s, 3), (s, 4)\} \\ &\cup \{(a, t), (b, t), (c, t), (d, t), (e, t)\} \end{aligned}$$

which results in the following picture: (arcs not labeled with a capacity are assumed to have infinite capacity). The node names have been moved inside the nodes so that there is less clutter. There are no a' priori mass balance constraints on this network. (See next page.)

We will take the new nodes s and t to be our source and sink for the Maximum Flow problem respectively. By our construction flow cannot leave a node of V_1 or enter a node of V_2 from more than one arc. Also, no more than one unit of flow can enter any of the nodes originally from V_1 or V_2. This is because of the capacity constraints of the arcs leaving the source s and entering the sink t. Furthermore we will **modify** the Maximum Flow problem by requiring that **flow on any arc must be integral** so that it is not possible to send a fractional amounts of flow out along two arcs.

Specifying that arcs from V_1 to V_2 can infinite capacity is not crucial. As long as their capacity is at least 1, we get the same results. We allow these arcs capacities to be infinite to point out that the capacity constraints on (s, v_1) and (v_2, t) are really what's important. The general result is formalized below.



Theorem 4 (Maximum Flow $\Leftrightarrow$ Maximal Matching). *The cardinality of a maximum matching in any bipartite graph $G = (V_1 \cup V_2, E)$ equals the value of the integer maximum flow in the constructed network $G(N, A)$.*

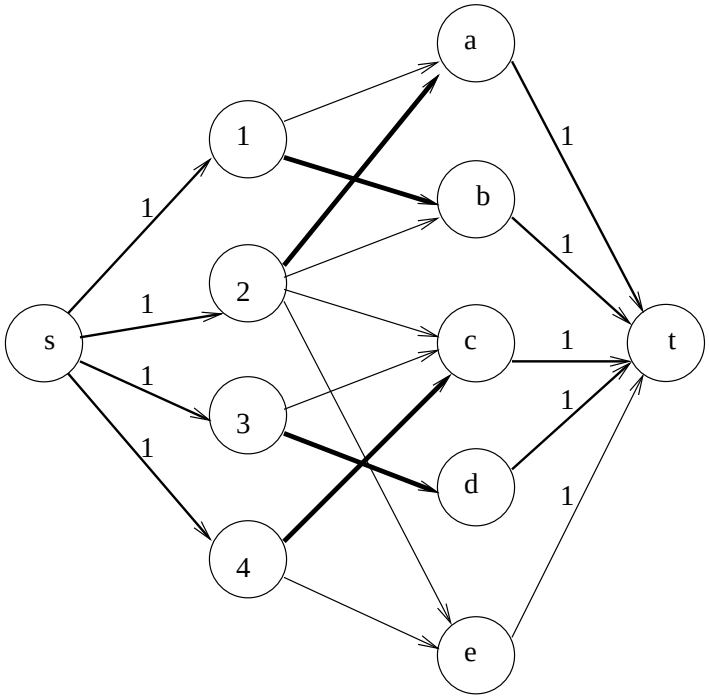
Proof. Given an matching M in $G = (V_1 \cup V_2, E)$ we will construct a feasible flow x in $G = (N, A)$ where the outflow of s and the inflow at t equals $|M|$, as follows. For each edge $(v_1, v_2) \in M$ such that $v_1 \in V_1$ and $v_2 \in V_2$ let there be unit flow through (s, v_1) and (v_2, t) and (v_1, v_2). Each edge in M is adjacent to one node of V_1 and one node of V_2. This, together with the fact that the capacity of arcs adjacent with either s or t are unit, and all arc flows are integral, we have that s and t must source and sink exactly $|M|$ units of flow. By construction x is feasible. The desired result follows.

Conversely, consider an integral maximum flow x in $G = (N, A)$. (*) By the construction of $G = (N, A)$ the outflow and inflow at s and t respectively, is equal to the flow on the arcs $(v_1, v_2) \in A$, where $v_1 \in V_1$ and $v_2 \in V_2$. Since the outflow and inflow from s and t are integral, and no more than 1, only one unit of flow can enter $v_1 \in V_1$ from s and exit $v_2 \in V_2$ to go out to t. From this we have that only one unit of flow can pass, necessarily along one arc, from a node of V_1 to a node of V_2. Therefore, if we let M be those edges (v_1, v_2) such that $f(v_1, v_2) = 1$, this gives a valid matching. By (*) $|M|$ is equal to the value of the maximum flow. $\square$

Although I have yet to figure out a way to get `Mathematica` to guarantee that its solutions to arbitrary LP problems are integral, as of yet I also have not been able to produce otherwise (in the case of Maximum Flow problems). However, one can show a theorem [6] which states that for any LP problem whose *basic* solution is integral, any algorithm using the simplex method (integral) will identify this solution. In spite of this, `Maple` does allow the user to specify integer variables, and `Lindo` can be forced to give integer solutions if

one exists. In general, the Integer Programming problem is more difficult than the general Linear Programming problem. Moreover, it has been shown to be *NP-Complete*, meaning that there is no known polynomial time algorithm for it, and that it is "just as difficult" as all other known non-deterministically polynomial problems, for where there are no known deterministic polynomial time algorithms.

Nonetheless, `Mathematica` returns an integral result in almost no time at all. The matching returned my `Mathematica` for the graph in Example 6 is given below. See the following page for the actual input and output corresponding to this solution. Note that the corresponding matching is different than the one provided by Algorithm 1.



For a long time it was not known whether or not the general LP problem was NP-Complete. Rather recently (1990's) a polynomial time algorithm was given, involving elliptical curves. This having been said, using the simplex method for which there are instances which cause a non-polynomial running time (see [3]), is not (theoretically) any better than using the search-tree algorithm provided earlier. However, many instances of LP problems can be solved by the Simplex Method in time linear in the number of variables and constraints! Better yet, there are algorithms which solve the *Simple Unit Capacity* Maximum Flow problem (which is exactly the one we needed to solve) in time $O(|A|\sqrt{|N|})$ (see [1] and [6]), which yields the best known time bound for solving the Bipartite Matching Problem. However, to develop this technique could be a paper in and of itself. The algorithm relies on a clever mixture of two different strategies for solving the Maximum Flow problem (See [1] or [6]). A *simple* network is one in which each node has exactly one incoming or outgoing arc. This should not be confused with the definition of a *simple graph*.

Example 6: **Maximum Flow for Bipartite Matching**

```
ConstrainedMax[
 e[s1] + e[s2] + e[s3] + e[s4],

 {(*subject to*)

  -e[s1] - e[s2] - e[s3] - e[s4] <= 0,
  e[at] + e[bt] + e[ct] + e[dt] + e[et] >= 0,

  (* mass balance constraints *)
  e[s1] - e[1 a] - e[1 b] == 0,
  e[s2] - e[2 a] - e[2 b] - e[2 c] == 0,
  e[s3] - e[3 c] - e[3 d] == 0,
  e[s4] - e[4 c] - e[4 e] == 0,
  e[1 a] + e[2 a] - e[at] == 0,
  e[1 b] + e[2 b] - e[bt] == 0,
  e[2 c] + e[3 c] + e[4 c] - e[ct] == 0,
  e[3 d] - e[dt] == 0,
  e[2 e] + e[4 e] - e[et] == 0,

  (* Total Outflow Must Equal Total Inflow *)
  -e[s1] - e[s2] - e[s3] - e[s4] + e[at] + e[bt] + e[ct] + e[dt] + e[et] == 0,

  (* capacity costraints, non-infinite *)
  e[s1] >= 0, e[s1] <= 1,
  e[s2] >= 0, e[s2] <= 1,
  e[s3] >= 0, e[s3] <= 1,
  e[s4] >= 0, e[s4] <= 1,
  e[at] >= 0, e[at] <= 1,
  e[bt] >= 0, e[bt] <= 1,
  e[ct] >= 0, e[ct] <= 1,
  e[dt] >= 0, e[dt] <= 1,
  e[et] >= 0, e[et] <= 1,

  (* infinite constraints *)
  e[1 a] >= 0,  e[1 b] >= 0, e[2 a] >= 0, s[2 b] >= 0,
  e[2 c] >= 0, e[2 e] >= 0, e[3 c] >= 0, e[3 d] >= 0, e[4 c] >= 0, e[4 e] >= 0},

 (* tell mathematica which variables *)
 {e[s1], e[s2], e[s3], e[s4], e[at], e[bt], e[ct], e[dt], e[et],
  e[1 a],  e[1 b], e[2 a], e[2 b], e[2 c], e[2 e], e[3 c], e[3 d], e[4 c], e[4 e]}]
```

```
{4, {e[s1] → 1, e[s2] → 1, e[s3] → 1, e[s4] → 1, e[at] → 1, e[bt] → 1, e[ct] → 1,
  e[dt] → 1, e[et] → 0, e[a] → 0, e[b] → 1, e[2 a] → 1, e[2 b] → 0, e[2 c] → 0,
  e[2 e] → 0, e[3 c] → 0, e[3 d] → 1, e[4 c] → 1, e[4 e] → 0}}
```

Following, we describe a classic algorithm for solving the Maximum Flow problem which is easy to understand, the Ford-Fulkerson algorithm. Each of the algorithms alluded to previously were developed (in part) from this technique, as were many of the algorithms for solving the Maximum Flow, and Minimum Cost Maximum Flow problem. Moreover, a generalization of the Ford-Fulkerson algorithm, and our LP formulation, can be useful for finding Optimal Solutions to the Weighted Bipartite Matching problem, to be discussed shortly.

Ford-Fulkerson Algorithm for Maximum Flows

The basic idea behind the version of the Ford-Fulkerson algorithm which we will discuss lies in finding paths from sources to sinks on which flow can be augmented. Since our transformation from the bipartite matching problem to the maximum flow problem only introduced a singe source and sink, we will deal only with this case here. A few definitions will help start us off.

Definition 11 (Arc Increments). *In a (directed) network $G = (N, A)$, with capacity constraints $u : A \to \aleph$, we define the* increment $i(\cdot)$ *for each pair of nodes $(a, b) \in N \times N$ as follows*

1. $i(a, b) = u(a, b) - x(a, b)$, *for $(a, b) \in A$– called a* forward arc *increment.*
2. $i(b, a) = x(a, b)$, *for $(a, b) \in A$– called a* reverse arc *increment, since it depends on the flow on the* forward arc (a, b).
3. $i(a, b) = 0$ *otherwise.*

One can think of the increment of an arc $a \in A$ as being equal to the amount of flow which can still be augmented along this (forward) arc $i(a)$, and the amount of flow which can be taken away from it, or augmented in the other (reverse) direction. If an arc (forward or reverse) does not exist between a pair $(a, b) \in N \times N$, then its increment is zero. It is important here to note that $i(b, a)$ in the definition above is not intended to represent an arc actually in A. However, we can think of the increment of an arc (a, b) as replacing each (forward arc) with two anti-parallel arcs whose capacities are $i(a, b) = u(a, b) - x(a, b)$ (in the forward direction) and $i(b, a) = x(a, b)$ (in the reverse). We have dummy zero increment arcs for pairs $(a, b) \in N \times N$ but $(a, b) \notin A$. Note also that our definition of increment still makes sense when there are already anti-parallel arcs in G. However, it doesn't make much sense to have flow on both arcs in an anti-parallel pair.

Definition 12 ($s \rightsquigarrow t$ Incrementing Paths). *A path $P = \langle s = v_1, v_2, \ldots, v_n = t \rangle$ is an $s \rightsquigarrow t$* incrementing path *if $i(v_i, v_{i+1}) > 0$ for all $i \in \{1, 2, \ldots, n-1\}$.*

In other words, an incrementing path is one which we can send flow along to relax some supply at the source, and a some demand at the sink. We can think of the task of finding incrementing paths as looking for paths in an auxiliary network $G = (N, i(A))$, where the

arcs A are replaced by pairs of incrementing anti-parallel arcs as described in Definition 11. However, note that $i(A)$ is really also a function of the flow x. Therefore increments would need to be recomputed whenever x changes.

If we find an incrementing path P, we can augment no more flow than the smallest arc increment in P. For any arc increment corresponding to a reverse arc in P augmenting flow along this arc really corresponds to reducing flow on the corresponding anti-parallel forward arc. Therefore, we define the increment of a path $i(P)$, where $P = \langle s = v_1, v_2, \ldots, v_n = t \rangle$, to be

$$i(P) = \min_{1 \leq i \leq n-1} i(v_i, v_{i+1}).$$

From the way we defined arc increments, if we augment $i(P)$ along P (adding flow to forward arcs of P and taking away flow from reverse arcs) the resulting augmented flow will satisfy the capacity constraints. As already mentioned, after the augmentation, the increment of the arcs along P will have to be updated to account for the new flow.

We now have the artillery necessary to describe the Ford-Fulkerson algorithm. In this description we assume that there are no anti-parallel arcs in $G = (N, A)$. This is mostly for convenience, and because our transformation from a bipartite graph will produce no anti-parallel arcs. The algorithm is easily generalized. See [4] and [6].

Algorithm 2 (Ford-Fulkerson). *Consider the network $G = (N, A)$, with capacity constraints $u : A \to \aleph$. The following technique computes a maximum flow x with value f in G.*

- *let $x = 0$. That is, set the flow on each arc $x(a, b) = 0$, for all $(a, b) \in A(G)$.*
- *let $i(a, b) = u(a, b)$ and $i(b, a) = 0$ for all $(a, b) \in A(G)$*
- *let $f = 0$, the total amount sourced at s or sinked at t.*
- *While there exists an $s \rightsquigarrow t$ incrementing path P*
  - *augment $i(P)$ units of flow along the arcs of P as follows*

$$\begin{array}{ll} x(a,b) := x(a,b) + i(a,b) & \text{if } (a,b) \text{ denotes a forward arc in } P \\ x(b,a) := x(b,a) - i(a,b) & \text{if } (a,b) \text{ denotes a reverse arc in } P \\ x(a,b) := x(a,b) & \text{otherwise} \end{array}$$

  - *recompute the increments for the node pairs $(a, b) \in P$.*
  - *$f := f + i(P)$.*
- *(end-while) Otherwise, stop. Return f.*

Intuitively, the flow f resulting from Algorithm 2 is maximum because at the end there were no more $s \rightsquigarrow t$ incrementing paths. In general, we have

$$f = \sum_{b \,:\, (s,b) \in A} x(s, b) = \sum_{a \,:\, (a,t) \in A} x(a, t) \tag{9}$$

where s is the source and t is the sink. This is due to flow conservation (1), and the way we define the value of the total flow, f.

In order to prove that Algorithm 2 yields a maximal flow we will need some more definitions.

Definition 13 ($s \rightsquigarrow t$ cuts). *Let $S \subset N$ such that the source $s \in S$ and the sink $t \notin S$. Let $T = N - S$ ($t \in T$). An $s \rightsquigarrow t$* cut *is a set of arcs $A(S,T)$ (or $A(T,S)$) of arcs (a,b) such that $a \in S$ and $b \in T$ ($a \in T$, $b \in S$). The capacity of a cut $u(S,T)$ is*

$$u(S,T) = \sum_{(a,b) \in A(S,T)} u(a,b).$$

Similarly, the flow on a cut $x(S,T)$ is

$$x(S,T) = \sum_{(a,b) \in A(S,T)} x(a,b).$$

In general, we can write $A(X,Y)$ to denote the set of arcs from nodes in X to nodes in Y, where $X, Y \subseteq N(G)$. The flow, and capacity functions over $A(X,Y)$ are defined similarly as above.

Lemma 2 (Relating Cuts to Flows). *Consider an arbitrary $s \rightsquigarrow t$ cut $A(S,T)$ and a flow x with value f in a network $G = (N,A)$, then [4]*

$$f = x(S,T) - x(T,S) \tag{10}$$

and

$$f \leq c(S,T). \tag{11}$$

In other words, the value of the flow f is equal to the total flow out of S, *minus* the total flow into S (out of T). Furthermore, f is always less than or equal to the total capacity of the arcs going from S to T, where $A(S,T)$ is an $s \rightsquigarrow t$ cut.

Proof. Since flow only exits the source s, and by (9)

$$x(\{s\}, N) = f \qquad \text{and} \qquad x(N, \{s\}) = 0$$

($N = N(G)$), the node set) while, by flow conservation (1), for vertex v which is not the source or the sink (the total inflow must equal total outflow)

$$x(\{v\}, N) = x(N, \{v\}).$$

Thus, for any $s \rightsquigarrow t$ cut $A(S,T)$ we have that

$$x(S,N) - x(N,S) = \sum_{v \in S} x(\{v\}, N) - x(N, \{v\}) = f \tag{12}$$

as the sum telescopes. Equivalently, we can write

$$x(S, N) = x(S, S \cup T) = x(S, S) + x(S, T)$$

and similarly

$$x(N, S) = x(N \cup T, S) = x(T, S) + x(S, S).$$

Combining this with (12) yields

$$\begin{aligned} f &= x(\{v\}, N) - x(N, \{v\}) \\ &= x(S, S) + x(S, T) - (x(T, S) + x(S, S)) \\ &= x(S, T) - x(T, S) \end{aligned}$$

thus showing (10). Now, since our capacity constraints tell us that for each $(a, b) \in A$, $x(a, b) \leq c(a, b)$ we must also have that $x(S, T) \leq u(S, T)$, which shows that

$$\begin{aligned} f &= x(S, T) - x(T, S) \\ &\leq x(S, T) \\ &\leq u(S, T) \end{aligned}$$

thus establishing (11). □

Since the cut $A(S, T)$ was arbitrary in the above lemma, (11) shows us that we must have that the flow value f is less than the capacity of any $s \rightsquigarrow t$ cut (in particular, the minimum capacity cut). Therefore, a *maximum flow* is a flow x with value f less than or equal to the minimum capacity. The following Theorem establishes the result that the maximum flow must have value equal to the capacity of a minimum cut. (This is where the stopping criteria of the Ford-Fulkerson Algorithm comes in.)

Theorem 5 (Max-Flow, Min-Cut). *Let $G = (N, A)$ be a network with source s and sink t in N, and whose arcs are associated integral capacity constraints $u : A \to \aleph$. Then there exists a maximum flow x with value f such that*

$$f = \min\{u(S, T) : A(S, T) \text{ is an } s \rightsquigarrow t \text{ cut}\} \tag{13}$$

or in other words, x is maximal. [4]

This result is attributed for Lester Ford and Ray Fulkerson, after whom Algorithm 2 is named. We deal only with integral capacities and flows here. However, the general result is shown similarly (although the Ford-Fulkerson algorithm need not converge). This proof uses Algorithm 2, and so also serves the purpose of showing that the flow resulting from its operations is indeed maximal.

Proof. Consider any valid flow x with value f. Let S be the set of nodes $z \in N$ such that either $z = s$, or there exists an incrementing path from s to z. Let us consider two cases:

i. The sink $t \in S$. In this case, this means that there must exist an incrementing path P from s to t with respect to the current flow x. Therefore $i(P) \geq 1$. In this case we can increase the value of f by augmenting flow from s to t along P as in Algorithm 2, producing a flow whose resulting value is $f + i(P)$. We can repeat this process (as is done in Algorithm 2) until there is no $s \rightsquigarrow t$ incrementing path. This brings us to our next case.

ii. The sink $t \in T = N - S$, and we have that $A(S,T)$ is an $s \rightsquigarrow t$ cut. Now, by the definition of S, for any $a \in S$ there is an incrementing path $s \rightsquigarrow a$. Consider then, some $b \in T$. If $(a,b) \in A$, then $x(a,b) = u(a,u)$. Otherwise, there would be an incrementing path $s \rightsquigarrow b$, contradicting that $b \notin S$. Alternatively, if $(b,a) \in A$, then $x(a,b) = 0$. Otherwise, $i(b,a) > 0$, and again there would be an incrementing $s \rightsquigarrow b$ path– again producing a contradiction. But this shows that

$$x(S,T) = c(S,T) \qquad \text{and} \qquad f(T,S) = 0$$

From Lemma 2 we have that

$$f = u(S,T) - 0 = u(S,T)$$

as desired.

Therefore, when Algorithm 2 cannot find an incrementing path, the value f of the current flow x must be equal to an $s \rightsquigarrow t$ cut, and is therefore maximum. □

We now have another way to solve the bipartite matching problem, using the Ford-Fulkerson algorithm for finding maximum flows. The running time of Algorithm 2 depends heavily on the choice of the routine used for finding shortest paths. Since we have assumed that the capacities are integral, each augmentation of flow along an incrementing path increases the flow value by at least 1. Therefore, the algorithm makes no more than f augmentations. Notice that if the arc capacities were allowed to be rational, and without specifying a lower bound on all flow augmentations, it is possible that the algorithm never finish! If we let $SP(G)$ denote the amount of time it takes to find an incrementing $s \rightsquigarrow t$ path in the network $G = (N,A)$, the Ford-Fulkerson algorithm runs in $O(f \cdot SP(G))$. See any of [5], [1], [4], [2], and many others for more information about path-finding routines. A modified *Breadth First Search* can be used in our case to get that the Ford-Fulkerson algorithm runs in $O(f \cdot |A|)$. This very close to the best we can do in general. Notice that f may not be polynomial in $|N|$ or $|A|$. However, when arc capacities are unit, in our case of solving the bipartite matching problem, it is.

Optimal Bipartite Matchings:

Economic Assignment [4][1], Minimum Cost Maximum Flows [1][6], Linear Programming Revisited [3][6], Successive Shortest Paths Algorithm[1]

In this section we consider the bipartite matching problem, where the edges of the graph have a cost or weight associated with them. Consider the following problems as motivation.

Problem 6 (Optimal Assignment). *A graduate school is prepared to enroll n college graduates which will fill n RA (Research Assistant) and TA (Teaching Assistant) positions. Based on grades, aptitude tests, and letters of recommendation, past experience, etc, the graduate school has assigned a* utility index *u_{ij} for assigning student i to job j. The objective is to identify an assignment that maximizes the total utility over all student/job assignments.[1]*

[4] Production planning in general requires finding solutions to weighted assignment problems, e.g. contracts are assigned to sub-contractors, individual workers are assigned tasks, machine tools assigned locations, goods are stored in assigned warehouse locations, etc. In general, problems of this type reduce to the problem of finding the assignments of least cost, or largest weight/utility.

These assignment problems can be easily written as bipartite matching problems on graphs $G = (V_1 \cup V_2, E)$, with a weight function $w : E \to \aleph$. Here we want to find the matching which maximizes the weight. Otherwise, we associate a cost $c : E \to \aleph$, where we are interested in finding a matching which minimizes the cost. The two situations are essentially identical. If W is greater than any $w(i,j)$ for all $(i,j) \in E$, then we can consider the cost $c(i,j) = W - w(i,j)$. In this case, minimizing the cost is the same as maximizing the weight. Alternatively we can consider $w(i,j) = C - c(i,j)$.

Definition 14 (Complete Bipartite Graphs). *A* complete bipartite graph *$G = (V_1 \cup V_2, E)$ is a bipartite graph in which we have that $E = V_1 \times V_2$. That is, for each $v_1 \in V_1$ and $v_2 \in V_2$, we have that $(v_1, v_2) \in E$.*

A key observation which will greatly simplify our discussion is that the underlying optimization problem is unchanged if we consider only *complete* bipartite graphs. For a bipartite graph $G = (V_1 \cup V_2, E)$ we can introduce edges into G between V_1 and V_2 which were not in G and set their weight to zero. Furthermore, if $|V_1| \neq |V_2|$, we can add nodes to one (or the other) connecting them to nodes in the other partition with with zero weight arcs, so that the partitions are equal in size. Clearly, the underlying optimization problem is unchanged. Therefore, for the remainder of this section we deal with only complete bipartite graphs, and consider finding a matching which maximizes the weight.

The next step might be to formulate the Optimal Assignment problem as a Network Flow problem: introduce new source and sink nodes, capacities and costs associated with its arcs, etc, and solve what is called a *Minimum Cost Maximum Flow* LP problem. While we will discuss this briefly later, the above assumption allows us to skip right to a simple algebraic formulation of the Optimal Assignment problem without introducing a network.

Consider $G = (V_1 \cup V_2, E)$, where $n = |V_1| = |V_2|$ and let $x(i,j)$ be a set of variables for $i = 1, \ldots, n$ and $j = 1, \ldots, n$. Here $x(i,j) = 1$ means that the edge $(v_{1_i}, v_{2_j}) = 1$ is included

in the matching, whereas if $x(i,j) = 0$ it is not. Clearly then, we wish to solve the following Integer Programming problem.

maximize

$$\sum_{\forall(i,j)} w(i,j) * x(i,j) \tag{14}$$

subject to

$$\sum_{j=1}^{n} x(i,j) = 1 \qquad \text{for } i = 1, \ldots, n \tag{15}$$

$$\sum_{i=1}^{n} x(i,j) = 1 \qquad \text{for } j = 1, \ldots, n \tag{16}$$

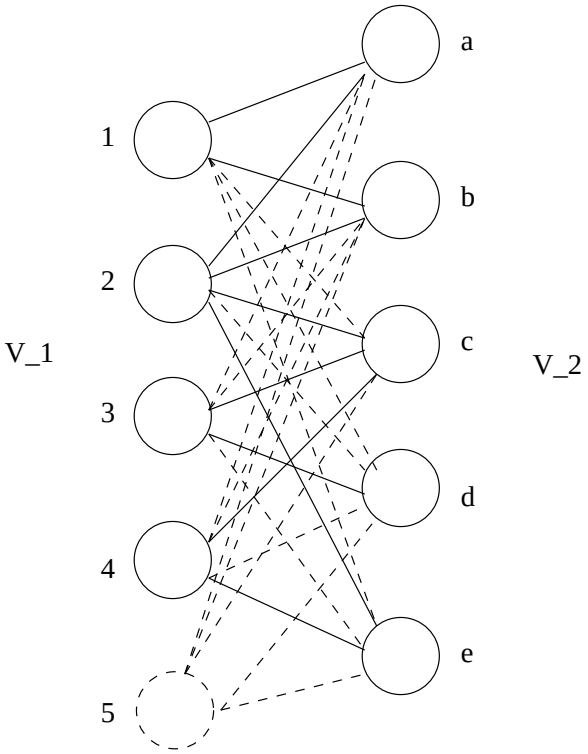
$$x(i,j) \geq 0 \tag{17}$$

This is actually a great deal simpler than the Maximum Flow formulation (4 - 8)! An example follows.

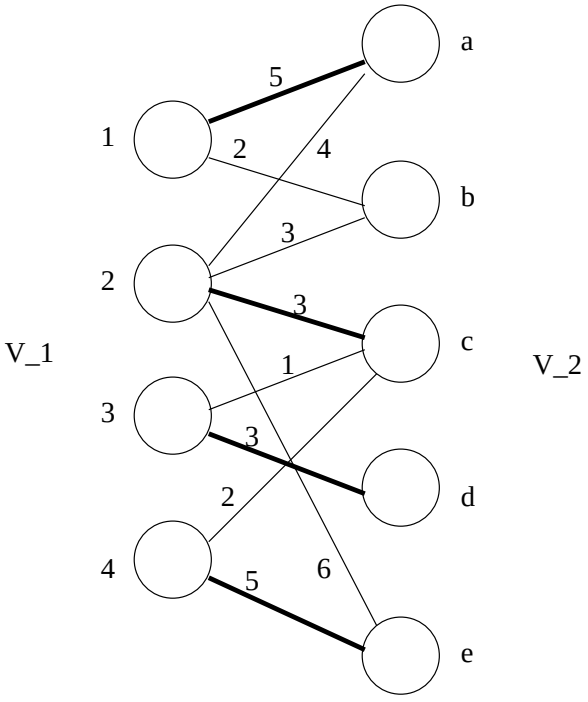
Example 7 (Optimal Assignment). *Again, we use the same bipartite graph as in Example 3, with the following weights assigned to its edges.*

$w(1,a) = 5$ $\quad w(1,b) = 2$ $\quad w(2,a) = 4$ $\quad w(2,b) = 3$ $\quad w(2,c) = 3$ $\quad w(2,e) = 6$
$w(3,c) = 1$ $\quad w(3,d) = 3$ $\quad w(4,c) = 2$ $\quad w(4,e) = 5$

Node 5 and the (auxiliary) dashed edges with zero weight were added to have a complete graph with $|V_1| = |V_2|$. In the following pictorial representation the weights are left off to reduce clutter. They will be included in solution diagram following.



We use `Mathematica` to get the optimal assignment that follows (leaving out the auxiliary edges and nodes). See the following page for the actual `Mathematica` input and output.



As already mentioned, we could create an auxiliary network, and solve a Minimum Cost Maximum Flow (MCMF) problem. As with the bipartite matching problem, this formulation

will allow us to use some very powerful algorithms for solving the MCMF problem to find an optimal matching[1]. One of these algorithms (also due to Ford and Fulkerson) is called the *Primal-Dual Method*. A variation on this algorithm, called *The Hungarian Method* can be used to solve the Optimal Assignment Problem directly. In many ways the Hungarian Method also resembles the search-tree technique we discussed earlier for solving the bipartite matching problem. Unfortunately, these techniques are beyond the scope of this paper. There are entire volumes of text dedicated to solving these and other similar problems– which are covered in just about all of our references. For more information about the Hungarian Method explicitly see [4] and [6].

Note also that the regular bipartite matching problem (sometimes called the *cardinality bipartite matching problem*) is the same as the Optimal Assignment problem where $w(i,j) = 1$ for all $(i,j) \in E(G)$. Therefore, we can also adapt the techniques described above to solve the Bipartite Matching problem as well.

[1].. for example, the Successive Shortest Paths (SSP) algorithm for solving a variation of the Transshipment Problem– which is very similar to the Ford-Fulkerson algorithm. It involves sending flow along shortest paths between the excess and deficit node pairs and maintains potentials at each node which produce "reduced arc costs" representing the utility of an arc with respect to a source, guaranteeing optimality– until the flow is feasible; the Out-Of-Kilter algorithm which contrary to the SSP algorithm maintains a feasible flow at every iteration, and strives to attain optimality by augmenting flows on shortest paths; as well as others like the Relaxation Algorithm which relies on some heuristics which make it efficient for most MCMF instances.

Exercise 7: **Optimal Assignment**

```
ConstrainedMax[
 5 * e[1 a] + 2 * e[1 b] + 4 * e[2 a] +
  3 * e[2 b] + 3 * e[2 c] + 6 * e[2 e] + 1 * e[3 c] + 3 * e[3 d] + 2 * e[4 c] + 5 * e[4 e],

 {(* subject to *)

  (* V_1 vertices can only have one edge matched *)
  e[1 a] + e[1 b] + e[1 c] + e[1 d] + e[1 e] == 1,
  e[2 a] + e[2 b] + e[2 c] + e[2 d] + e[2 e] == 1,
  e[3 a] + e[3 b] + e[3 c] + e[3 d] + e[3 e] == 1,
  e[4 a] + e[4 b] + e[4 c] + e[4 d] + e[4 e] == 1,
  e[5 a] + e[5 b] + e[5 c] + e[5 d] + e[5 e] == 1,

  (* V_2 vertices can only have one edge matched *)
  e[1 a] + e[2 a] + e[3 a] + e[4 a] + e[5 a] == 1,
  e[1 b] + e[2 b] + e[3 b] + e[4 b] + e[5 b] == 1,
  e[1 c] + e[2 c] + e[3 c] + e[4 c] + e[5 c] == 1,
  e[1 d] + e[2 d] + e[3 d] + e[4 d] + e[5 d] == 1,
  e[1 e] + e[2 e] + e[3 e] + e[4 e] + e[5 e] == 1,

  (* non-negrativity constraints *)
  e[1 a] >= 0, e[1 b] >= 0, e[1 c] >= 0, e[1 d] >= 0,
  e[1 e] >= 0, e[2 a] >= 0, e[2 b] >= 0, e[2 c] >= 0,
  e[2 d] >= 0, e[2 e] >= 0, e[3 a] >= 0, e[3 b] >= 0,
  e[3 c] >= 0, e[3 d] >= 0, e[3 e] >= 0, e[4 a] >= 0,
  e[4 b] >= 0, e[4 c] >= 0, e[4 d] >= 0, e[4 e] >= 0,
  e[5 a] >= 0, e[5 b] >= 0, e[5 c] >= 0, e[5 d] >= 0,
  e[5 e] >= 0},

 (* tell mathematica which variables *)
 {e[1 a], e[1 b], e[1 c], e[1 d], e[1 e], e[2 a],
  e[2 b], e[2 c], e[2 d], e[2 e], e[3 a], e[3 b], e[3 c], e[3 d], e[3 e],
  e[4 a], e[4 b], e[4 c], e[4 d], e[4 e], e[5 a], e[5 b], e[5 c], e[5 d], e[5 e]}]
```

```
{16, {e[a] → 1, e[b] → 0, e[c] → 0, e[d] → 0, e[e] → 0, e[2 a] → 0, e[2 b] → 0,
  e[2 c] → 1, e[2 d] → 0, e[2 e] → 0, e[3 a] → 0, e[3 b] → 0, e[3 c] → 0, e[3 d] → 1,
  e[3 e] → 0, e[4 a] → 0, e[4 b] → 0, e[4 c] → 0, e[4 d] → 0, e[4 e] → 1, e[5 a] → 0,
  e[5 b] → 1, e[5 c] → 0, e[5 d] → 0, e[5 e] → 0}}
```

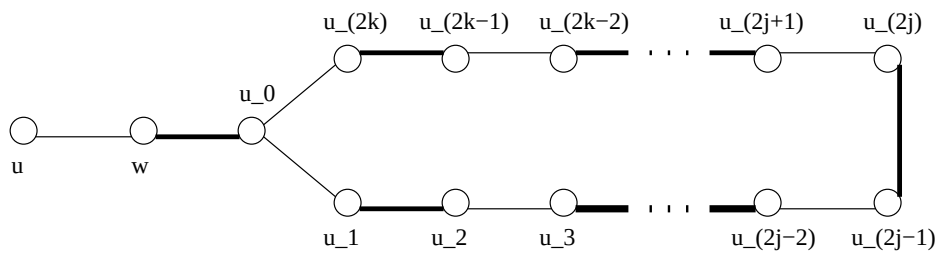
General Matchings:
Flowers & Blossoms [6][1], Finding Augmenting Paths & Contracting Blossoms [1][6], General Weighted Matchings [6][1]

In this section we will try to adapt our initial technique for solving the bipartite matching problem to solve the matching problem on arbitrary graphs. We have already given one example (Problem 1: RAF) of how a general bipartite matching problem can be used to solve a combinatorial optimization problem. Let us give on more example to motivate our final discussion. The problem is a fancy restatement of the general cardinality matching problem. Of course, as was the case with bipartite matchings, the most interesting problems will be optimization problems, however it will be easiest to consider the cardinality matching problem first. Once the technique for finding augmenting paths is generalized, we discuss general weighted matching problem.

Problem 7 (Hostel Pairing). *A hostel manager wants to assign pairs of roommates to rooms of her hostel. Nationality, religion, cultural background, and hobbies determine compatible pairs of roommates. The problem of finding the maximum number of compatible pairs is a maximum cardinality matching problem.[1]*

Algorithm 1, given earlier, *almost* works for the general matching problem. Recall that Algorithm 1 employs a search technique which essentially involves maintaining two sets, E and O of even an odd labeled vertices respectively. For the bipartite matching problem the even vertices were always from V_1 and the odd vertices were always from V_2. That is, we knew a' priori which vertices could possibly be odd, and which could be even. Moreover, we knew that because the graph was bipartite that no vertex could possibly be labeled both even and odd. The fact that this might not be the case in general is what trips up the algorithm for general matchings. It is possible that when searching for augmenting paths in the fashion of Algorithm 1 that we encounter situations where a vertex could have been labeled both odd and even. Furthermore, if we attempt to reduce the original input graph into a directed graph, as described earlier, it is possible for the algorithm to think it has found an augmenting path, when one does not exist! It turns out that we observe this phenomenon exactly when the reduction routine treats a vertex as both odd and even!

Furthermore, it turns out that it is only possible that a vertex could be labeled as odd or even when the graph contains a subgraph like the one depicted below. Recall that essentially the criteria for a vertex v to labeled even is that either v was the initial unsaturated vertex, or was discovered by traversing a matched edge. Alternatively, a vertex receives an odd label if it was discovered by traversing an unmatched edge.



A subgraph like the one shown above is called a *blossom*. Sometimes, the vertices v and w, and others which may occur along a chain of alternating matched an unmatched edges preceding v are referred to as the *stem*. Many texts refer the blossom and stem together as a *flower*. However, since the most interesting part of a flower is its blossom, we focus most on this part. Vertex u_0 in the above graph is called the *base* of the blossom. Notice that if we search from u_0 (which is odd) around the blossom like $\langle u_0, u_1, u_2, \ldots u_{2k-1}, u_{2k}\rangle$ the vertices $u_{2\ell-1}$ will have even labels, and $u_{2\ell}$ will have odd labels. On the other hand, if the search proceeds as $\langle u_0, u_{2k}, u_{2k-1}, \ldots, u_2, u_1\rangle$ then $u_{2\ell-1}$ will have odd labels, and $u_{2\ell}$ will be even. Therefore, every vertex in the blossom could be have been labeled odd or even, depending which way the blossom was traversed!

Clearly, whenever there exists a blossom as a subgraph in a larger graph, there exist vertices (in the blossom) which could be labeled two different ways. Furthermore, whenever there exist vertices which could possibly be labeled more than one way, there exists a blossom. Therefore, in order to to adapt the search procedure employed by Algorithm 1, we must be able to detect, and deal with blossoms.

Detecting blossoms is not too difficult. If at any time in the search procedure, the algorithm notices that it is trying to label an odd vertex even, or vice-versa, it has found a blossom! All it has to do then is backtrack from this vertex in both directions until it finds the base of the blossom!

One way to "deal with" with a blossom it simply to get rid of it! This can be done by replacing all of the vertices and edges of the blossom with a single vertex. This operation is referred to as *contracting* (or shrinking) a blossom. In particular, the operation of contacting a blossom B goes as follows: Let $G/B = (V/B, E, B)$ where $V/B = (V - V(B)) \cup \{v_b\}$. E/B is all of the edges of $E - E(B)$ together with edges like (u, v) where $v \in V(B)$ and $u \in V - V(B)$. In essence, the blossom B is replaced by a single vertex v_b. Where there were edges adjacent to vertices of the blossom B there exist edges now to v_b in G/B.

Next we must verify that the operation of contracting a blossom does not add or remove any augmenting paths. We show this through a sequence of lemmas and theorems verifying the following intuition. Clearly, when contracting a blossom it is necessary to keep the information about the blossom B somewhere in order to augment M (perhaps through a blossom), etc. If ever an augmenting path goes through a contracted blossom, we will need to expand the blossom to make the augmentation. We want that the operation of contracting a blossom does not hide any augmenting paths. We must also verify that the operation of contraction does not introduce any augmenting paths which were not there before. The following Lemma and Theorem also hi-light the essential modification which can be made to the search process of Algorithm 1 so that it can work with blossoms.

Lemma 3 (Paths Ending in Blossoms). *Suppose that while searching for an augmenting path from an unsaturated vertex i in an arbitrary graph G with respect to a matching M, we discovered a blossom B. Then there is an augmenting path from u to any vertex in $V(B)$ ending with a matched edge. [6]*

Proof. Since the blossom B was discovered from i, there must be an alternating path from u to u_0, the basis of B. We showed previously that any vertex in the blossom B can be labeled as both odd and even and so must be reachable two alternating paths from u_0. By

definition (as long as the path is not empty) the path giving an even labeling ends with a matched edge. Therefore there is an alternating path to every vertex in $V(B)$ ending with a matched edge, as desired. □

Theorem 6 (No Harm Done by Contracting Blossoms). *Suppose that while searching for an augmenting path from an unsaturated node $i \in V(G)$ with respect to a matching M, we discover a blossom B. In this case, there is an augmenting path from i in G if and only if there is one from i to in G/B with respect to M/B. [6]*

Before we get started, be wary that this is the most mathematically complicated discussion we have attempted thus far. (I am not sure that I understand everything.)

Proof. ($\Rightarrow$) Suppose that there is an augmenting path P from i in G/B with respect to M/B. (Assume that $i \neq v_b$). If P does not pass through v_b, then P is clearly also an augmenting path in G, and we are done. Otherwise, if P passes through v_b then P can be written like $P = \langle i = w_1, \ldots w_j, v_b, w_{j+1}, \ldots \rangle$. Since P is an alternating path, either $(w_j, v_b) \in M$ or $(v_b, w_{j+1}) \in M$. Consider the case where $(w_j, v_b) \in M$ (the other case is similar). In this case, since $(w_j, v_b) \in M$ we must have that $(w_j, u_0) \in M$ for the basis u_0 of B, and $(w_{j+1}, u_j) \in E - M$ for some other $u_j \in V(B)$. From this we construct the equivalent augmenting path in G by taking $P_{u_j} = \langle i = w_1, \ldots w_j, u_0 \rangle$ together by $P_{w_{j+1}}$ which is the path in B starting at the basis u_0 and going to w_j in a matched edge, followed by $P_{\ldots}$ which is the remainder of P starting a w_{j+1}. The resulting path can be written compactly as $P_{u_j} \cdot P_{w_{j+1}} \cdot P_{\ldots}$, where $(\cdot)$ is the concatenation operation. (When $i = v_b$ the argument is essentially the same, only $P_{u_j} = \emptyset$.)

($\Leftarrow$) Suppose now that there is an augmenting path P in G with respect to M. As before, if P does not pass through any vertex of the blossom B then we are done. Otherwise, consider the following two cases:

i. P enters B on a matched edge. Therefore, P must enter B through u_0. Let u_j be the last node on P which is still in blossom B. If it happens that $u_j = u_0$ then we are done because P is clearly an augmenting path in G/B where u_0 has been replaced by v_b. Otherwise, P contains at least two vertices of $V(B)$ (or equivalently one edge in $E(B)$). Since P entered B through u_0 matched, it must exit B unmatched (as every other edge of B is matched). When B is contracted, all of the edges (including the unsaturated edge, say (u_j, w_ℓ), P used to exit B, and the matched edge it came in on) will be adjacent to v_b. In this case if P looks like $\langle i = w_1, \ldots, u_0, \ldots, u_j, w_\ell, \ldots \rangle$, then $\langle i = w_1, \ldots, v_b, w_\ell, \ldots \rangle$ will be an augmenting path in G/B.

ii. P enters B on an unsaturated edge. (The hard case.) Therefore P cannot enter B through the basis u_0. If P leaves B through the basis u_0 on a matched edge, essentially the reverse of the above argument suffices. Otherwise, we have that P leaves B through an unmatched edge (in which case it does not leave through the basis). Thus, P can be written as $P_i = \langle i = w_1, \ldots u_j \rangle$ where P enters B at u_i, then by $P_B = \langle u_i, \ldots, u_j \rangle$ which is the part of P contained entirely in B, followed by $P_j = \langle u_j, \ldots \rangle$ finishing off P. That is, $P = P_i \cdot P_B \cdot P_j$. By Lemma 3, there is an alternating path from i through u_0 (the basis of B) ending in a matched edge at some $u_k \in V(B)$. Let Q be the initial portion of this path ending at the basis u_0. We must consider three possibilities:

1. If P_j and Q have no vertices in common, then $Q \cdot \langle v_b \rangle \cdot P_j$ is an augmenting path in G/B.
2. P_j and Q do intersect. Let Q be written as $Q = Q_i \cdot \langle x \rangle \cdot Q_{u_0}$ and $P_j = P_j^1 \cdot \langle x \rangle \cdot P_j^2$ where x is the last node that both P_j and Q have in common. In other words, P_j^2 has no node in common with Q. Then, if $Q_{u_0}^{-1}$ denotes the reverse of Q_{u_0}, we have that $P_i \cdot \langle v_b \rangle \cdot Q_{u_0}^{-1} \cdot P_j^2$ is an augmenting path in G/B.
3. The same as (2) above, but instead Q_{u_0} and P_i intersect. Let y be the last node of Q_{u_0} which is either on P_i or on P_j. If $Q_{u_0}^y$ denotes the part of Q_{u_0} after y, and y is on P_j, then $P_i \cdot \langle v_b \rangle \cdot Q_{u_0}^y \cdot P_j$ is an augmenting path in G/B. Otherwise, if y is on P_i, and we let P_i^y denote the initial part of P_j up to y, then $P_i^y \cdot Q \langle v_b \rangle \cdot P_j$ is an augmenting path in G/B.

And the theorem is (finally) proved. □

The above result essentially illustrates the sort of changes that need to be made to the search procedure in Algorithm 1. The process can be summed up simply, as follows:

Algorithm 1':

1. Search the graph G as in Algorithm 1.
   a. if a blossom B is found, contract B, thereby producing G/B.
   b. continue with (1) as if $G = G/B$ (however, it important to record somewhere the blossom which v_b represents so it can be recovered later).
2. If an augmenting path P is found, augment $M' = M \oplus E(P)$ as in Algorithm 1.
3. When P does not traverse through a contracted blossom, the augmentation is exactly as in Algorithm 1.
4. Otherwise (P traverses a blossom B)
   a. For each contracted blossom node $v_b \in V(P)$, expand B and continue the augmentation as indicated by the proof Theorem 6.
   b. Perform the above step (a) recursively (for all blossoms within blossoms).
   c. Proceed augmenting as usual once the path P exits the blossom.
   d. (do not re-contract blossoms)
5. Repeat (goto 1) until there are no more augmenting paths.
6. When the search is complete, expand all blossoms (recursively)

By Theorem's 1 and 6, when the above algorithm is complete, and the result is a maximal matching M for the graph G. For a more detail-oriented, pseudocode description of Algorithm 1' see [1] or [6]. Theorem 2 shows that we need only consider finding an augmenting path from each vertex once. Therefore, $|V(G)|$ is a bound on how many times Algorithm

1' will search for an augmenting path. Similarly, it should also be obvious that $|V|$ is also a bound on the total number of contractions. In fact, since a blossom has at least three nodes, which are reduced to one node by a contraction, one could argue that there can be at most $|V|/2$ contractions [1] total. Unfortunately, without explicitly outlining the operation of contracting a blossom, and the details of augmenting M through a contracted blossom, we do not altogether have enough information to state (or show) a full time bound on our algorithm. For the algorithm given in [1], which most closely resembles Algorithm 1', this text argues an $O(|V|^3)$ time bound. On the other hand, [6], whose algorithm uses the Directed Graph reduction discussed earlier, shows an $O(|V|^4)$ bound.

General Weighted Matchings

Since the methods for solving the general weighted matching problem are quite complicated, we avoid discussing it here, but refer the interested reader to [6]. While there does exist a rather straightforward LP formulation of the problem it turns out that the basic solutions to such a program need not be integral! Therefore, giving such a program to `Mathematica` would be in vain. The algorithm discussed in [6] does take advantage of some important properties of linear programming, and in particular the Revised Simplex Method– which would take another thirty pages to explain.

The general weighted matching problem can also be reduced to a Minimum Cost Maximum Flow (MCMF) problem defined on a auxiliary network. This is also briefly discussed in [6]. Some techniques for solving this problem were mentioned earlier. A nice reference for such problems in general is [1].

Surprisingly, it turns out that while the description of algorithms for the general matching problem is (conceptually) much more difficult than all of the algorithms discussed in this paper, their asymptotic (time) complexity is no worse than that of the algorithms for solving the general cardinality matching problem, discussed previously. Again, see [6].

That being said, we finish with a nice practical application general weighted matchings.

Problem 8 (Locating Objects In Space). *To identify an object in (three-dimensional) space, we could use two infrared sensors, located at geographically different sites. Each sensor provides an angle of sight of the object and hence the line on which the object must lie. The unique intersection of the two lines provided by the two sensors (provided that the two sensors and the object are not collinear) determines the unique location of the object in space.*

Consider now, the situation in which we wish to determine the locations of p objects using two sensors. The first sensor would provide us with a set of lines $L_1, L_2, \ldots, L_p$ for the p objects and the second sensor would provide us a different set of lines $L'_1, L'_2, \ldots, L'_p$. To identify the location of the objects– using the fact that if two lines correspond to the same object, the lines intersect one another– we need to match the lines from the first sensor to the lines from the second sensor. In practice, two difficulties limit the use of this approach. First, a line from a sensor might intersect more than one line from the other sensor, so the matching is not unique. Second, two lines corresponding to the same object might not intersect because the sensors make measurement errors in determining the angle of sight. We can overcome this difficulty in most situations by formulating the problem as an assignment problem.

In the assignment problem, we wish to match the p lines from the first sensor with the p lines from the second sensor. We define a cost c_{ij} of the assignment (i, j) as the minimum Euclidean distance between the line L_i and L_j. If the lines correspond to the same object, c_{ij} would be close to zero. An optimal solution of the assignment problem would provide an excellent matching of the lines. Simulation studies have found that in most circumstances, the matching produced by the assignment problem defines the correct location of the objects. [1]

Other applications include the Chinese Postman Problem (see [6]) and the determining of the atomic structure of arbitrary (real or fictional) molecules.

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Addison Wesley, 1993.

[2] Sara Baase and Allen van Gelder. *Computer Algorithms.* Addison Wesley, 3rd edition, 2000.

[3] Vašek Chvátal. *Linear Programming.* W.H. Freeman and Company, 15th edition, 2000.

[4] John Clark and Derek Alan Holton. *Graph Theory.* World Scientific, 2nd edition, 1998.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* McGraw Hill, 21st edition, 1998.

[6] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatroial Optimization: Algorithms and Complexity.* Dover, Dover edition, 1998.