Notes 18 for CS 170

1 Network Flows

1.1 The problem

Suppose that we are given the network of Figure 1 (top), where the numbers indicate capacities, that is, the amount of flow that can go through the edge in unit time. We wish to find the maximum amount of flow that can go through this network, from s to t.

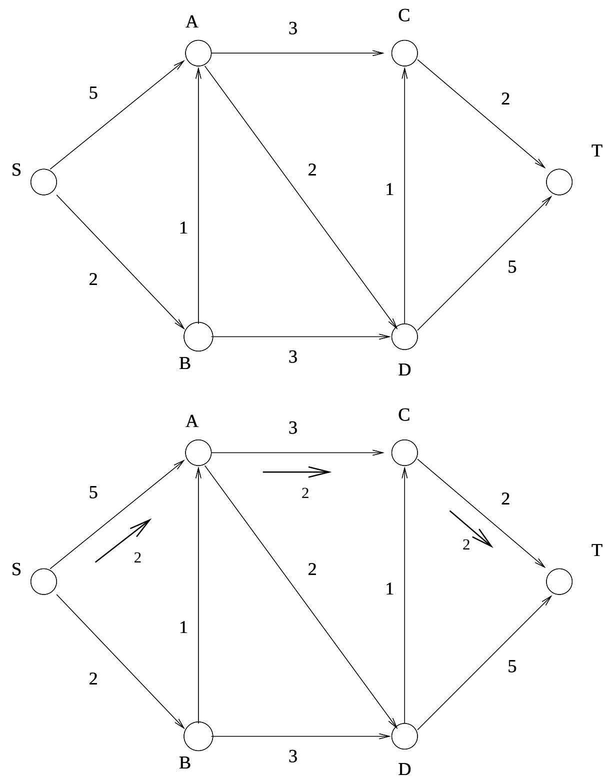


Figure 1: Max flow.

This problem can also be reduced to linear programming. We have a nonnegative variable for each edge, representing the flow through this edge. These variables are denoted $f_{s,a}, f_{s,b}, \ldots$ We have two kinds of constraints:

- *capacity* constraints such as $f_{s,a} \leq 5$ (a total of 9 such constraints, one for each edge), and
- *flow conservation* constraints (one for each node except s and t), such as $f_{a,d} + f_{b,d} = f_{d,c} + f_{d,t}$ (a total of 4 such constraints).

We wish to maximize $f_{s,a} + f_{s,b}$, the amount of flow that leaves s, subject to these constraints. It is easy to see that this linear program is equivalent to the max-flow problem. The simplex method would correctly solve it.

In general, we are given a "network" that is just a directed graph $G = (V, E)$ with two special vertices s, t, such that s has only outgoing edges and t has only incoming edges, and a capacity $c_{u,v}$ associated to each edge $(u, v) \in E$. Then the maximum flow in the network is the solution of the following linear program, having a variable $f_{u,v}$ for every edge.

$$
\begin{array}{lrcll}
\text{maximize} & \sum_{v:(s,v)\in E} f_{u,v} & & & \\
\text{subject to} & & & & \\
 & f_{u,v} & \leq & c_{u,v} & \text{for every } (u,v) \in E \\
 & \sum_{u:(u,v)\in E} f_{u,v} - \sum_{w:(v,w)\in E} f_{v,w} & = & 0 & \text{for every } v \in V - \{s,t\} \\
 & f_{u,v} & \geq & 0 & \text{for every } (u,v) \in E
\end{array}
$$

1.2 The Ford-Fulkerson Algorithm

If the simplex algorithm is applied to the linear program for the maximum flow problem, it will find the optimal flow in the following way: start from a vertex of the polytope, such as the vertex with $f_{u,v} = 0$ for all edges $(u, v) \in E$, and then proceed to improve the solution (by moving along edges of the polytope from one polytope vertex to another) until we reach the polytope vertex corresponding to the optimal flow.

Let us now try to come up directly with an efficient algorithm for max flow based on the idea of starting from an empty flow and progressively improving it.

How can we find a small improvement in the flow? We can find a path from s to t (say, by depth-first search), and move flow along this path of total value equal to the *minimum* capacity of an edge on the path (we can obviously do no better). This is the first iteration of our algorithm (see the bottom of Figure 1).

How to continue? We can look for another path from s to t. Since this time we already partially (or totally) use some of the edges, we should do depth-first search on the edges that have some *residual capacity*, above and beyond the flow they already carry. Thus, the edge (c, t) would be ignored, as if it were not there. The depth-first search would now find the path $s - a - d - t$, and augment the flow by two more units, as shown in the top of Figure 2.

Next, we would again try to find a path from s to t. The path is now $s - b - d - t$ (the edges $c - t$ and $a - d$ are full are are therefore ignored), and we augment the flow as shown in the bottom of Figure 2.

Next we would again try to find a path. But since edges $a - d$, $c - t$, and $s - b$ are full, they must be ignored, and therefore depth-first search would fail to find a path, after marking the nodes s, a, c as reachable from S. *We then return the flow shown, of value 6, as maximum.*

There is a complication that we have swept under the rug so far: When we do depth-first search looking for a path, we use not only the edges that are not completely full, but we must also traverse *in the opposite direction* all edges that already have some non-zero flow. This would have the effect of cancelling some flow; cancelling may be necessary to achieve optimality, see Figure 1.2. In this figure the only way to augment the current flow is via the

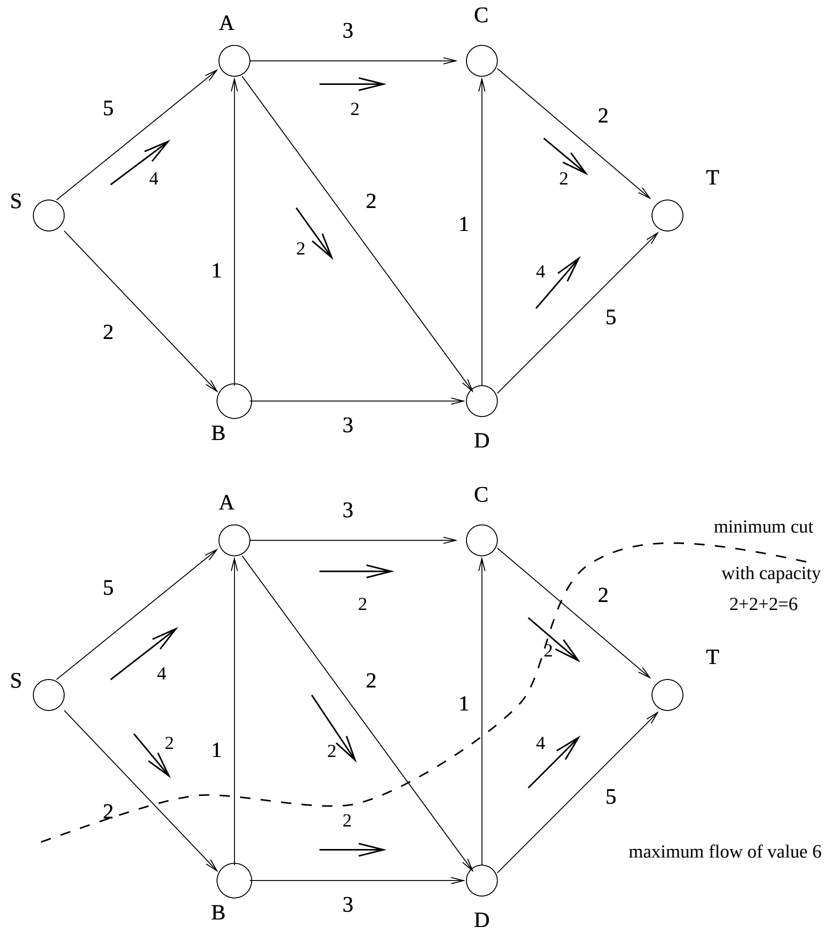
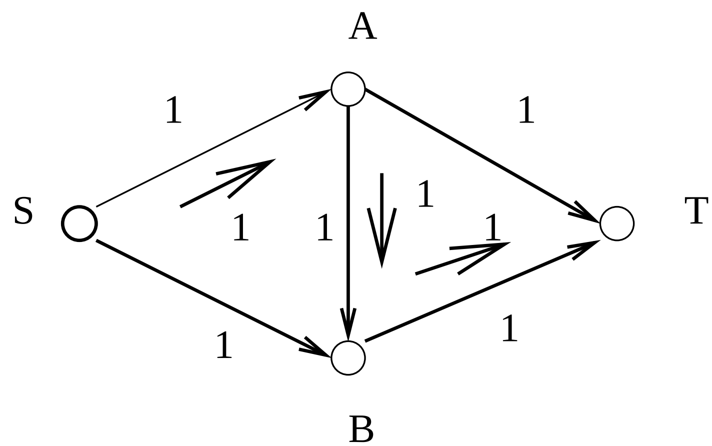


Figure 2: Max flow (continued).

path $s - b - a - t$, which traverses the edge $a - b$ in the reverse direction (a legal traversal, since $a - b$ is carrying non-zero flow).



Flows may have to be cancelled.

The algorithm that we just outlined is the Ford-Fulkerson algorithm for the maximum flow problem. The pseudocode for Ford-Fulkerson is as follows.

1. Given in input $G = (V, E)$, vertices s, t, and capacities $c_{u,v}$. Initialize $f_{u,v}$ to zero for all edges.
2. Use depth-first search to find a path from s to t. If no path is found, return the current flow.
3. Let c be the smallest capacity along the path.
4. For each edge (u, v) in the path, decrease $c_{u,v}$ by c, increase $f_{u,v}$ by c, and increase $c_{v,u}$ by c (if the edge (v, u) does not exist in G, create it). Delete edges with capacity zero.
5. Go to step 2

1.3 Analysis of the Ford-Fulkerson Algorithm

How can we be sure that the flow found by the Ford-Fulkerson algorithm is optimal?

Let us say that a set of vertices $S \subseteq V$ is a *cut* in a network if $s \in S$ and $t \notin S$. Let us define the *capacity* of a cut to be $\sum_{u \in S, v \notin S, (u,v) \in E} c_{u,v}$.

We first note that if we fix a flow f, and consider a cut S in the network, then the amount of flow that passes "through" the cut, is independent of S, and it is, indeed, the cost of the flow.

Lemma 1
Fix a flow f. For every cut S the quantity

$$\sum_{u \in S, v \notin S, (u,v) \in E} f_{u,v} - \sum_{u \in S, v \notin S, (v,u) \in E} f_{v,u}$$

is always the same, independently of S.

As a special case, we have that $\sum_u f_{s,u} = \sum_v f_{v,t}$, so that the flow coming out of s is exactly the flow getting into t.
PROOF: Assume $f_{u,v}$ is defined for every pair (u,v) and $f_{u,v} = 0$ if $(u,v) \notin E$.

$$
\begin{aligned}
& \sum_{u \in S, v \notin S} f_{u,v} - \sum_{u \notin S, v \in S} f_{u,v} \\
= & \sum_{u \in S, v \in V} f_{u,v} - \sum_{u \in S, v \in S} f_{u,v} - \sum_{u \notin S, v \in S} f_{u,v} \\
= & \sum_{u \in S, v \in V} f_{u,v} - \sum_{u \in V, v \in S} f_{u,v} \\
= & \sum_{v \in V} f_{s,v} + \sum_{u \in S-\{s\}, v \in V} f_{u,v} - \sum_{u \in V, v \in S-\{s\}} f_{u,v} \\
= & \sum_{v \in V} f_{s,v}
\end{aligned}
$$

and the last term is independent of S. $\Box$

A consequence of the previous result is that for every cut S and every flow f, the cost of the flow has to be smaller than or equal to the capacity of S. This is true because the cost of f is

$$\sum_{u \in S, v \notin S} f_{u,v} - \sum_{v \notin S, u \in S} f_{v,u} \leq \sum_{u \in S, v \notin S} f_{u,v} \leq \sum_{u \in S, v \notin S} c_{u,v}$$

and the right-hand side is exactly the capacity of S. Notice how this implies that if we find a cut S and a flow f such that the cost of f equals the capacity of S, then it must be the case that f is optimal. This is indeed the way in which we are going to prove the optimality of Ford-Fulkerson.

LEMMA 2
Let $G = (V, E)$ be a network with source s, sink t, and capacities $c_{u,v}$, and let f be the flow found by the Ford-Fulkerson algorithm. Let S be the set of vertices that are reachable from s in the residual network. *Then the capacity of S equals the cost of f, and so f is optimal.*

PROOF: First of all, S is a cut: s belongs to S by definition, and it is impossible that $t \in S$, because otherwise t would be reachable from s in the residual network of f, and f could not be the output of Ford-Fulkerson.

Now, the cost of the flow is $\sum_{u \in S, v \notin S} f_{u,v} - \sum_{v \notin S, u \in S} f_{v,u}$, while the capacity of the cut is $\sum_{u \in S, v \notin S} c_{u,v}$.

For every $u \in S$ and $v \notin S$, we must have $f_{u,v} = c_{u,v}$, otherwise v would be reachable from s in the residual network (which would contradict the definition of S). Also, for every $v \notin S$ and every $u \in S$, we must have $f_{v,u} = 0$, otherwise the residual network would have an edge with non-zero capacity going from u to v, and then v would be reachable from s which is impossible. So the cost of the flow is the same as the capacity of the cut. $\Box$

Notice that, along the way, we have proved the following important theorem.

THEOREM 3 (MAX FLOW / MIN CUT THEOREM)
In every network, the cost of the maximum flow equals the capacity of the minimum cut.

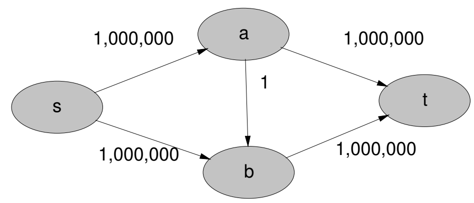


Figure 3: A bad instance for Ford-Fulkerson.

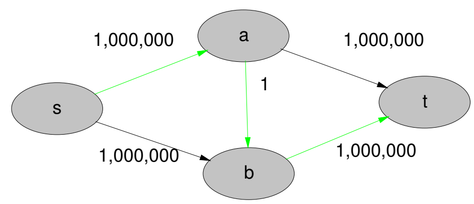


Figure 4: A bad instance for Ford-Fulkerson – Choice of path.

We have established the correctness of the Ford-Fulkerson algorithm. What about the running time? Each step of finding an augmenting path and updating the residual network can be implemented in $O(|V| + |E|)$ time. How many steps there can be in the worst case? This may depend on the values of the capacities, and the actual number of steps may be exponential in the size of the input. Consider the network in Figure 3.

If we choose the augmenting path with the edge of capacity 1, as shown in Figure 4, then after the first step we are left with the residual network of Figure 5. Now you see where this is going.

If step 2 of Ford-Fulkerson is implemented with breadth-first search, then each time we use an augmenting path with a minimal number of edges. This implementation of Ford-Fulkerson is called the Edmonds-Karp algorithm, and Edmonds and Karp showed that the number of steps is always at most $O(|V||E|)$, regardless of the values of the capacities, for a total running time of $O(|V||E|(|V| + |E|))$. The expression can be simplified by observing that we are only going to consider the portion of the network that is reachable from s, and that contains $O(E)$ vertices, so that an augmenting path can indeed be found in $O(E)$ time, and the total running time can be written as $O(|V||E|^2)$.

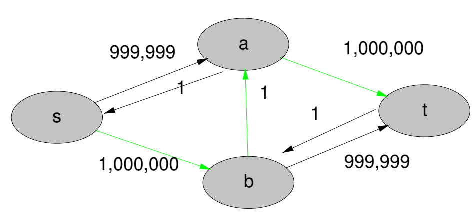


Figure 5: A bad instance for Ford-Fulkerson – Residual network.

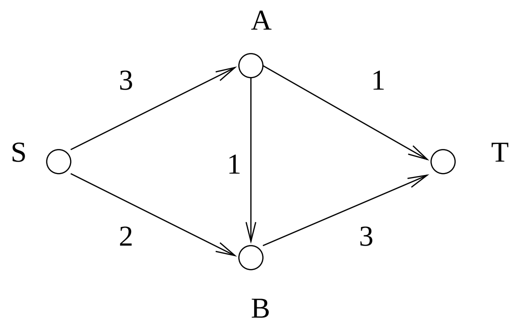


Figure 6: A simple max-flow problem.

2 Duality

As it turns out, the max-flow min-cut theorem is a special case of a more general phenomenon called *duality*. Basically, duality means that a maximization and a minimization problem have the property that any feasible solution of the min problem is greater than or equal any feasible solution of the max problem (see Figure 6). Furthermore, and more importantly, *they have the same optimum.*

Consider the network shown in Figure 6 below, and the corresponding max-flow problem. We know that it can be written as a linear program as follows ($f_{xy} \geq 0$ in the last line is shorthand for all 5 inequalities like $f_{s,a} \geq 0$, etc.):

$$P: \left\{ \begin{array}{llllllll}
\max f_{sa} & +f_{sb} & & & & & & \\
f_{sa} & & & & & \leq & 3 \\
 & f_{sb} & & & & \leq & 2 \\
 & & f_{ab} & & & \leq & 1 \\
 & & & f_{at} & & \leq & 1 \\
 & & & & f_{bt} & \leq & 3 \\
f_{sa} & & -f_{ab} & -f_{at} & & = & 0 \\
 & f_{sb} & +f_{ab} & & -f_{bt} & = & 0 \\
 & & & & f_{xy} & \geq & 0
\end{array} \right.$$

Consider now the following linear program (where again $y_{xy} \geq 0$ is shorthand for all inequalities of that form):

$$D: \left\{ \begin{array}{lllllllll}
\min 3y_{sa} & +2y_{sb} & +y_{ab} & +y_{at} & +3y_{bt} & & & & \\
y_{sa} & & & & & +u_a & & \geq & 1 \\
 & y_{sb} & & & & & +u_b & \geq & 1 \\
 & & y_{ab} & & & -u_a & +u_b & \geq & 0 \\
 & & & y_{at} & & -u_a & & \geq & 0 \\
 & & & & y_{bt} & & -u_b & \geq & 0 \\
 & & & & & & y_{xy} & \geq & 0
\end{array} \right.$$

This LP describes the min-cut problem! To see why, suppose that the u_a variable is meant to be 1 if A is in the cut with S, and 0 otherwise, and similarly for u_b (naturally, by the definition of a cut, S will always be with S in the cut, and T will never be with S). Each of the y variables is to be 1 if the corresponding edge contributes to the cut capacity, and 0 otherwise. Then the constraints make sure that these variables behave exactly as they should. For example, the first constraint states that *if A is not with S, then SA must be added to the cut.* The third one states that *if A is with S and B is not* (this is the only case in which the sum $-u_a + u_b$ becomes -1), *then AB must contribute to the cut.* And so on. Although the y and u's are free to take values larger than one, they will be "slammed" by the minimization down to 1 or 0 (we will not prove this here).

Let us now make a remarkable observation: These two programs have strikingly symmetric, *dual*, structure. Each variable of P corresponds to a constraint of D, and vice-versa. Equality constraints correspond to unrestricted variables (the u's), and inequality constraints to restricted variables. Minimization becomes maximization. The matrices are transposes of one another, and the roles of right-hand side and objective function are interchanged. Such LP's are called *dual* to each other.

By the max-flow min-cut theorem, the two LP's P and D above have the same optimum. *In fact, this is true for general dual LP's!* This is the *duality theorem*, which can be stated as follows.

Theorem 4 (Duality Theorem)
Consider a primal *LP problem written in the form* "maximize $c^T x$ subject to $A \cdot x \leq b$ and $x \geq 0$". *We define the* dual *problem to be* "minimize $b^T y$ subject to $A^T y \geq c$ and $y \geq 0$". *Suppose the primal LP has a finite solution. Then so does the dual problem, and the two optimal solutions have the same cost.*

The theorem has an easy part and a hard part. It is easy to prove that for every feasible x for the primal and every feasible y for the dual we have $c^T x \leq b^T y$. This implies that the optimum of the primal is at most the optimum of the dual (this property is called *weak duality.* To prove that the costs are equal for the optimum is the hard part.

Let us see the proof of weak duality. Let x be feasible for the primal, and y be feasible for the dual. We have:

$$c^T x \leq y^T A x \leq y^T b = b^T y,$$

where the first inequality follows from the dual feasibility constraint $A^T y \geq c$ (and the algebraic rule $(A^T y)^T = y^T A$), and the second inequality follows from the primal feasibility constraint $Ax \geq b$.

3 Matching

3.1 Definitions

A *bipartite graph* is a (typically undirected) graph $G = (V, E)$ where the set of vertices can be partitioned into subsets V_1 and V_2 such that each edge has an endpoint in V_1 and an endpoint in V_2.

Often bipartite graphs represent relationships between different entities: clients/servers, people/projects, printers/files to print, senders/receivers ...

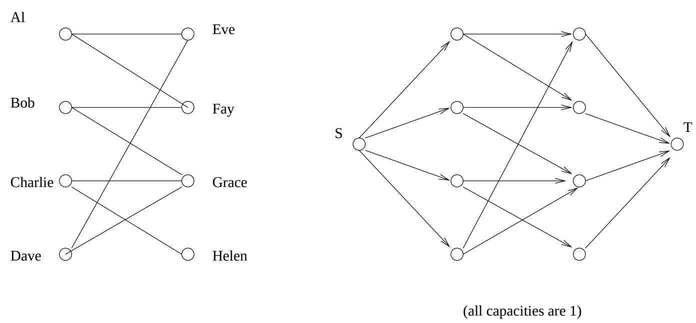


Figure 7: Reduction from matching to max-flow.

Often we will be given bipartite graphs in the form $G = (L, R, E)$, where L, R is already a partition of the vertices such that all edges go between L and R.

A *matching* in a graph is a set of edges that do not share any endpoint. In a bipartite graph a matching associates to some elements of L precisely one element of R (and vice versa). (So the matching can be seen as an *assignment*.)

We want to consider the following optimization problem: given a bipartite graph, find the matching with the largest number of edges. We will see how to solve this problem by reducing it to the maximum flow problem, and how to solve it directly.

3.2 Reduction to Maximum Flow

Let us first see the reduction to maximum flow on an example. Suppose that the *bipartite* graph shown in Figure 7 records the compatibility relation between four straight boys and four straight girls. We seek a maximum matching, that is, a set of edges that is as large as possible, and in which no two edges share a node. For example, in the figure below there is a *perfect* matching (a matching that involves all nodes).

To reduce this problem to max-flow we do this: We create a new source and a new sink, connect the source with all boys and all girls with the sinks, and direct all edges of the original bipartite graph from the boys to the girls. All edges have capacity one. It is easy to see that the maximum flow in this network corresponds to the maximum matching.

Well, the situation is slightly more complicated than was stated above: What is easy to see is that the optimum *integer-valued* flow corresponds to the optimum matching. We would be at a loss interpreting as a matching a flow that ships .7 units along the edge Al-Eve! Fortunately, what the algorithm in the previous section establishes is that *if the capacities are integers, then the maximum flow is integer.* This is because we only deal with integers throughout the algorithm. Hence *integrality comes for free in the max-flow problem.*

In general, given a bipartite graph $G = (L, R, E)$, we will create a network $G' = (V, E')$ where $V' = L \cup R \cup \{s, t\}$, and E' contains all directed edges of the form (s, u), for $u \in L$, (v, t), for $v \in R$, and (u, v), for $\{u, v\} \in E$. All edges have capacity one.

3.3 Direct Algorithm

Let us now describe a direct algorithm, that is essentially the composition of Ford-Fulkerson with the above reduction.

The algorithm proceeds in phases, starting with an empty matching. At each phase, it either finds a way to get a bigger matching, or it gets convinced that it has constructed the largest possible matching.

We first need some definitions. Let $G = (L, R, E)$ be a bipartite graph, $M \subseteq E$ be a matching. A vertex is *covered* by M if it is the endpoint of one of the edges in E.

An *alternating path* (in G, with respect to M) is a path of odd length that starts with a non-covered vertex, ends with a non-covered vertex, and alternates between using edges not in M and edges in M.

If M is our current matching, and we find an alternating path with respect to M, then we can increase the size of M by discarding all the edges in of M which are in the path, and taking all the others.

Starting with the empty matching, in each phase, the algorithm looks for an alternating path with respect to the current matching. If it finds an alternating path, it updates, and improves, the current matching as described above. If no alternating path is found, the current matching is output.

It remains to prove the following:

1. Given a bipartite graph G and a matching M, an alternating path can be found, if it exists, in $O(|V| + |E|)$ time using a variation of BFS.

2. If there is no alternating path, then M is a maximum matching.

We leave part (1) as an exercise and prove part(2).

Suppose M is not an optimal matching, that is, some other matching M^* has more edges. We prove that M must have an alternating path.

Let $G = (L, R, E')$ be the graph where E' contains the edges that are either in M or in M^* *but not in both* (i.e., $E' = M \oplus M^*$).

Every vertex of G has degree at most two. Furthermore if the degree is two, then one of the two incident edges is coming from M and the other from M^*.

Since the maximum degree is 2, G is made out of paths and cycles. Furthermore the cycles are of even length and contain each one an equal number of edges from M and form M^*.

But since $|M^*| > |M|$, M^* must contribute more edges than M to G, and so there must be some path in G where there are more edges of M^* than of M. This is an augmenting path for M.

This completes the description and proof of correctness of the algorithm. Regarding the running time, notice that no matching can contain more than $|V|/2$ edges, and this is an upper bound to the number of phases of the algorithm. Each phase takes $O(|E| + |V|) = O(|E|)$ time (we can ignore vertices that have degree zero, and so we can assume that $|V| = O(|E|)$), so the total running time is $O(|V||E|)$.