Chapter 26

Michelle Bodnar, Andrew Lohr

May 5, 2017

Exercise 26.1-1

To see that the networks have the same maximum flow, we will show that every flow through one of the networks corresponds to a flow through the other. First, suppose that we have some flow through the network before applying the splitting procedure to the anti-symmetric edges. Since we are only changing one of any pair of anti-symmetric edges, for any edge that is unchanged by the splitting, we just have an identical flow going through those edges. Suppose that there was some edge (u, v) that was split because it had an anti-symmetric edge, and we had some flow, $f(u, v)$ in the original graph. Since the capacity of both of the two edges that are introduced by the splitting of that edge have the same capacity, we can set $f'(u, v) = f'(u, x) = f'(x, v)$. By constructing the new flow in this manor, we have an identical total flow, and we also still have a valid flow.

Similarly, suppose that we had some flow f' on the graph with split edges, then, for any triple of vertices u, x, v that correspond to a split edge, we must have that $f'(u, x) = f'(x, v)$ because the only edge into x is (u, x) and the only edge out of x is (x, v), and the net flow into and out of each vertex must be zero. We can then just set the flow on the unsplit edge equal to the common value that the flows on (u, x) and (x, v) have. Again, since we handle this on an edge by edge basis, and each substitution of edges maintains the fact that it is a flow of the same total, we have that the end result is also a valid flow of the same total value as the original.

Since we have shown that any flow in one digraph can be translated into a flow of the same value in the other, we can translate the maximum value flow for one of them to get that it's max value flow is $\leq$ to that of the other, and do it in the reverse direction as well to achieve equality.

Exercise 26.1-2

The capacity constraint remains the same. We modify flow conservation so that each s_i and t_i must satisfy the "flow in equals flow out" constraint, and we only exempt s and t. We define the value of a flow in the multiple-source, multiple-sink problem to be $\left|\sum_{i=1}^{m} \sum_{v \in V} f(s_i, v) - \sum_{v \in V} f(v, s_i)\right|$. Let $f_i = \sum_{v \in V} f(s_i, v) - \sum_{v \in V} f(v, s_i)$. In the single-source flow network, set

$f(s, s_i) = f_i$. This satisfies the capacity constraint and flow conservation, so it is a valid assignment. The flow for the multiple-source network in this case is $|f_1 + f_2 + \cdots + f_m|$. In the single-source case, since there are no edges coming into s, the flow is $\sum_{i=1}^{m} f(s, s_i)$. Since $f(s, s_i)$ is positive and equal to f_i, they are equivalent.

Exercise 26.1-3

Suppose that we are in the situation posed by the question, that is, that there is some vertex u that lies on no path from s to t. Then, suppose that we have for some vertex v, either $f(v, u)$ or $f(u, v)$ is nonzero. Since flow must be conserved at u, having any positive flow either leaving or entering u, there is both flow leaving and entering. Since u doesn't lie on a path from s to t, we have that there are two cases, either there is no path from s to u or(possibly and) there is no path from u to t. If we are in the second case, we construct a path with $c_0 = u$, and c_{i+1} is an successor of c_i that has $f(c_i, c_{i+1})$ being positive. Since the only vertex that is allowed to have a larger flow in than flow out is t, we have that this path could only ever terminate if it were to reach t, since each vertex in the path has some positive flow in. However, we could never reach t because we are in the case that there is no path from u to t. If we are in the former case that there is no path from s to u, then we similarly define $c_0 = u$, however, we let c_{i+1} be any vertex so that $f(c_{i+1}, c_i)$ is nonzero. Again, this sequence of vertices cannot terminate since we could never arrive at having s as one of the vertices in the sequence.

Since in both cases, we an always keep extending the sequence of vertices, we have that it must repeat itself at some point. Once we have some cycle of vertices, we can decrease the total flow around the cycle by an amount equal to the minimum amount of flow that is going along it without changing the value of the flow from s to t since neither of those two vertices show up in the cycle. However, by decreasing the flow like this, we decrease the total number of edges that have a positive flow. If there is still any flow passing though u, we can continue to repeat this procedure, decreasing the number of edges with a positive flow by at least one. Since there are only finitely many vertices, at some point we need to have that there is no flow passing through u. The flow obtained after all of these steps is the desired maximum flow that the problem asks for.

Exercise 26.1-4

Since f_1 and f_2 are flows, they satisfy the capacity constraint, so we have $0 \leq \alpha f_1(u, v) + (1-\alpha) f_2(u, v) \leq \alpha c(u, v) + (1-\alpha) c(u, v) = c(u, v)$, so the new flow satisfies the capacity constraint. Further, f_1 and f_2 satisfy flow conservation,

so for all $u \in V - \{s, t\}$ we have

$$\begin{aligned}\sum_{v \in V} \alpha f_1(v,u) + (1-\alpha) f_2(v,u) &= \alpha \sum_{v \in V} f_1(v,u) + (1-\alpha) \sum_{v \in V} f_2(v,u) \\ &= \alpha \sum_{v \in V} f_1(u,v) + (1-\alpha) \sum_{v \in V} f_2(u,v) \\ &= \sum_{v \in V} \alpha f_1(u,v) + (1-\alpha) f_2(u,v).\end{aligned}$$

Therefore the flows form a convex set.

Exercise 26.1-5

A linear programming problem consists of a set of variables, a linear function of those variables that needs to be maximized, and a a set of constraints. Our variables x_e will be the amount of flow across each edge e. The function to maximize is $\sum_e$ leaving $_s\, x_e - \sum_e$ entering $_s\, x_e$. The sum of these flows is exactly equal to the value of the flow from s to t. Now, we consider constraints. There are two types of constraints, capacity constraints and flow constraints. The capacity constraints are just $x_e \leq c(e)$ where c_e is the capacity of edge e. The flow constraints are that $\sum_e$ leaving $_v\, x_e - \sum_e$ entering $_v\, x_e = 0$ for all vertices $v \neq s, t$. Since this linear program captures all the same constraints, and wants to maximize the same thing, it is equivalent to the max flow problem.

Exercise 26.1-6

Use the map to create a graph where vertices represent street intersections and edges represent streets. Define $c(u, v) = 1$ for all edges (u, v). Since a street can be traversed, start off by creating a directed edge in each direction, then make the transformation to a flow problem with no antiparallel edges as described in the section. Make the home the source and the school the sink. If there exist at least two distinct paths from source to sink then the flow will be at least 2 because we could assign $f(u, v) = 1$ for each of those edges. However, if there is at most one distinct path from source to sink then there must exist a bridge edge (u, v) whose removal would disconnect s from t. Since $c(u, v) = 1$, the flow into u is at most 1. We may assume there are no edges into s or out from t, since it doesn't make sense to return home or leave school. By flow conservation, this implies that $f = \sum_{v \in V} f(s, v) \leq 1$. Thus, determining the maximum flow tells the Professor whether or not his children can go to the same school.

Exercise 26.1-7

We can capture the vertex constraints by splitting out each vertex into two, where the edge between the two vertices is the vertex capacity. More formally,

our new flow network will have vertices $\{0,1\} \times V$. It has an edge between $1 \times v$ and $0 \times u$ if there is an edge (v,u) in the original graph, the capacity of such an edge is just $c(v,u)$. The edges of the second kind that the new flow network will have are from $0 \times v$ to $1 \times v$ for every v with capacity $l(v)$. This new flow network will have $2|V|$ vertices and have $|V|+|E|$ edges. Lastly, we can see that this network does capture the idea that the vertices have capacities $l(v)$. This is because any flow that goes through v in the original graph must go through the edge $(0 \times v, 1 \times v)$ in the new graph, in order to get from the edges going into v to the edges going out of v.

Exercise 26.2-1

To see that equation (26.6) equals (26.7), we will show that the terms that we are throwing into the sums are all zero. That is, we will show that if $v \in V \setminus (V_1 \cup V_2)$, then $f'(s,v) = f'(v,s) = 0$. Since $v \notin V_1$, then there is no edge from s to v, similarly, since $v \notin V_2$, there is no edge from v to s. This means that there is no edge connecting s and v in any way. Since flow can only pass along edges, we know that there can be no flow passing directly between s and v.

Exercise 26.2-2

The flow across the cut is $11 + 1 + 7 + 4 - 4 = 19$. The capacity of the cut is $16 + 4 + 7 + 4 = 31$.

Exercise 26.2-3
If we perform a breadth first search where we consider the neighbors of a vertex as they appear in the ordering $\{s, v_1, v_2, v_3, v_4, t\}$, the first path that we will find is s, v_1, v_3, t. The min capacity of this augmenting path is 12, so we send 12 units along it. We perform a BFS on the resulting residual network. This gets us the path s, v_2, v_4, t. The min capacity along this path is 4, so we send 4 units along it. Then, the only path remaining in the residual network is $\{s, v_2, v_4, v_3\}$ which has a min capacity of 7, since that's all that's left, we find it in our BFS. Putting it all together, the total flow that we have found has a value of 23.
Exercise 26.2-4

A minimum cut corresponding to the maximum flow is $S = \{s, v_1, v_2, v_4\}$ and $T = \{v_3, t\}$. The augmenting path in part (c) cancels flow on edge (v_3, v_2).

Exercise 26.2-5

Since the only edges that have infinite value are those going from the supersource or to the supersink, as long as we pick a cut that has the supersource and all the original sources on one side, and the other side has the supersink as well as all the original sinks, then it will only cut through edges of finite capacity. Then, by Corollary 26.5, we have that the value of the flow is bounded above

by the value of any of these types of cuts, which is finite.

Exercise 26.2-6

Begin by making the modification from multi-source to single-source as done in section 26.1. Next, create an extra vertex $\hat{s_i}$ for each i and place it between s and s_i. Explicitly, remove the edge from s to s_i and add edges $(s, \hat{s_i})$ and $(\hat{s_i}, s_i)$. Similarly, create an extra vertex $\hat{t_i}$ for each vertex t_i and place it between t and t_i. Remove the edges (t_i, t) and add edges $(t_i, \hat{t_i})$ and $(\hat{t_i}, t)$. Assign $c(\hat{s_i}, s_i) = p_i$ and $c(t_i, \hat{t_i}) = q_i$. If a flow which satisfies the constraints exists, it will assign $f(\hat{s_i}, s_i) = p_i$. By flow conservation, this implies that $\sum_{v \in V} f(s_i, v) = p_i$. Similarly, we must have $f(t_i, \hat{t_i}) = q_i$, so by flow conservation this implies that $\sum_{v \in V} f(v, t_i) = q_i$.

Exercise 26.2-7

To check that f_p is a flow, we make sure that it satisfies both the capacity constraints and the flow constraints. First, the capacity constraints. To see this, we recall our definition of $c_f(p)$, that is, it is the smallest residual capacity of any of the edges along the path p. Since we have that the residual capacity is always less than or equal to the initial capacity, we have that each value of the flow is less than the capacity. Second, we check the flow constraints, Since the only edges that are given any flow are along a path, we have that at each vertex interior to the path, the flow in from one edge is immediately canceled by the flow out to the next vertex in the path. Lastly, we can check that its value is equal to $c_f(p)$ because, while s may show up at spots later on in the path, it will be canceled out as it leaves to go to the next vertex. So, the only net flow from s is the initial edge along the path, since it (along with all the other edges) is given flow $c_f(p)$, that is the value of the flow f_p.

Exercise 26.2-8

Paths chosen by the while loop of line 3 go from s to t and are simple because capacities are always nonnegative. Thus, no edge into s will ever appear on an augmenting path, so such edges may as well never have existed.

Exercise 26.2-9

The augmented flow does satisfy the flow conservation property, since the sum of flow into a vertex and out of a vertex can be split into two sums each, one running over flow in f and the other running over flow in f', since we have the parts are equal separately, their sums are also equal.

The capacity constraint is not satisfied by this arbitrary augmentation of flows. To see this, suppose we only have the vertices s and t, and have a single edge from s to t of capacity 1. Then we could have a flow of value 1 from s to

t, however, augmenting this flow with itself ends up putting two units along the edge from s to t, which is greater than the capacity we can send.

Exercise 26.2-10

Suppose we already have a maximum flow f. Consider a new graph G where we set the capacity of edge (u, v) to $f(u, v)$. Run Ford-Fulkerson, with the modification that we remove an edge if its flow reaches its capacity. In other words, if $f(u, v) = c(u, v)$ then there should be no reverse edge appearing in residual network. This will still produce correct output in our case because we never exceed the actual maximum flow through an edge, so it is never advantageous to cancel flow. The augmenting paths chosen in this modified version of Ford-Fulkerson are precisely the ones we want. There are at most $|E|$ because every augmenting path produces at least one edge whose flow is equal to its capacity, which we set to be the actual flow for the edge in a maximum flow, and our modification prevents us from ever destroying this progress.

Exercise 26.2-11

To test edge connectivity, we will take our graph as is, pick an arbitrary s to be our source for the flow network, and then, we will consider every possible other selection of our sink t. For each of these flow networks, we will replace each (undirected) edge in the original graph with a pair of anti-symmetric edges, each of capacity 1.

We claim that the minimum value of all of these different considered flow networks' maximum flows is indeed the edge connectivity of the original graph. Consider one particular flow network, that is, a particular choice for t. Then, the value of the maximum flow is the same as the value of the minimum cut separating s and t. Since each of the edges have a unit capacity, the value of any cut is the same as the number of edges in the original graph that are cut by that particular cut. So, for this particular choice of t, we have that the maximum flow was the number of edges needed to be removed so that s and t are in different components. Since our end goal is to remove edges so that the graph becomes disconnected, this is why we need to consider all $n - 1$ flow networks. That is, it may be much harder for some choices of t than others to make s and t end up in different components. However, we know that there is some vertex that has to be in a different component than s after removing the smallest number of edges required to disconnect the graph. So, this value for the number of edges is considered when we have t be that vertex.

Exercise 26.2-12

Since every vertex lies on some path starting from s there must exist a cycle which contains the edge (v, s). Use a depth first search to find such a cycle with no edges of zero flow. Such a cycle must exist since f satisfies conservation of flow. Since the graph is connected this takes $O(E)$. Then decrement the flow

of every edge on the cycle by 1. This preserves the value of the flow so it is still maximal. It won't violate the capacity constraint because $f > 0$ on every edge of the cycle prior to decrementing. Finally, flow conservation isn't violated because we decrement both an incoming and outgoing edge for each vertex on the cycle by the same amount.

Exercise 26.2-13

Suppose that your given flow network contains $|E|$ edges, then, we were to modify all of the capacities of the edges by taking any edge that has a postive capacity and increasing its capacity by $\frac{1}{|E|+1}$. Doing this modification can't get us a set of edges for a min cut that isn't also a min cut for the unmodified graph because the difference between the value of the min cut and the next lowest cut vale was at least one because all edge weights were integers. This means that the new min cut value is going to be at most the original plus $\frac{|E|}{|E|+1}$. Since this value is more than the second smallest valued cut in the original flow network, we know that the choice of cuts we make in the new flow network is also a minimum cut in the original. Lastly, since we added a small constant amount to the value of each edge, our minimum cut would have the smallest possible number of edges, otherwise one with fewer would have a smaller value.

Exercise 26.3-1
First, we pick an augmenting path that passes through vertices 1 and 6. Then, we pick the path going through 2 and 8. Then, we pick the path going through 3 and 7. Then, the resulting residual graph has no path from s to t. So, we know that we are done, and that we are pairing up vertices $(1, 6), (2, 8)$, and $(3, 7)$. This number of unit augmenting paths agrees with the value of the cut where you cut the edges $(s, 3), (6, t)$, and $(7, t)$.

Exercise 26.3-2

We proceed by induction on the number of iterations of the while loop of Ford-Fulkerson. After the first iteration, since c only takes on integer values and $(u, v).f$ is set to 0, c_f only takes on integer values. Thus, lines 7 and 8 of Ford-Fulkerson only assign integer values to $(u, v).f$. Assume that $(u, v).f \in \mathbb{Z}$ for all (u, v) after the n^{th} iteration. On the $(n+1)^{st}$ iteration $c_f(p)$ is set to the minimum of $c_f(u, v)$ which is an integer by the induction hypothesis. Lines 7 and 8 compute $(u, v).f$ or $(v, u).f$. Either way, these the the sum or difference of integers by assumption, so after the $(n + 1)^{st}$ iteration we have that $(u, v).f$ is an integer for all $(u, v) \in E$. Since the value of the flow is a sum of flows of edges, we must have $|f| \in \mathbb{Z}$ as well.

Exercise 26.3-3

The length of an augmenting path can be at most $2 \min\{|L|, |R|\} + 1$. To

see that this is the case, we can construct an example which has an augmenting path of that length.

Suppose that the vertices of L are $\{\ell_1, \ell_2, \ldots \ell_{|L|}\}$, and of R are $\{r_1, r_2, \ldots, r_{|R|}\}$. For convenience, we will call $m = \min\{|L|, |R|\}$. Then, we will place the edges

$$\{(\ell_m, r_m - 1), (\ell_1, r_1), (\ell_1, r_m)\} \cup \left(\cup_{i=2}^{i=m-1}\{(\ell_i, r_i), (\ell_i, r_{i-1})\}\right)$$

Then, after augmenting with the shortest length path $\min\{|L|, |R|\}-1$ times, we could end up having sent a unit flow along $\{(\ell_i, r_i)\}_{i=1,\ldots,m-1}$. At this point, there is only a single augmenting path, namely, $\{s, \ell_m, r_{m-1}, \ell_{m-1}, r_{m-2}, \ldots, \ell_2, r_1, \ell_1, r_m, t\}$. This path has the length $2m + 1$.

It is clear that any simple path must have length at most $2m + 1$, since the path must start at s, then alternate back and forth between L and R, and then end at t. Since augmenting paths must be simple, it is clear that our bound given for the longest augmenting path is tight.

Exercise 26.3-4

First suppose there exists a perfect matching in G. Then for any subset $A \subseteq L$, each vertex of A is matched with a neighbor in R, and since it is a matching, no two such vertices are matched with the same vertex in R. Thus, there are at least $|A|$ vertices in the neighborhood of A. Now suppose that $|A| \leq |N(A)|$ for all $A \subseteq L$. Run Ford-Fulkerson on the corresponding flow network. The flow is increased by 1 each time an augmenting path is found, so it will suffice to show that this happens $|L|$ times. Suppose the while loop has run fewer than L times, but there is no augmenting path. Then fewer than L edges from L to R have flow 1. Let $v_1 \in L$ be such that no edge from v_1 to a vertex in R has nonzero flow. By assumption, v_1 has at least one neighbor $v_1' \in R$. If any of v_1's neighbors are connected to t in G_f then there is a path, so assume this is not the case. Thus, there must be some edge (v_2, v_1) with flow 1. By assumption, $N(\{v_1, v_2\}) \geq 2$, so there must exist $v_2' \neq v_1'$ such that $v_2' \in N(\{v_1, v_2\})$. If (v_2', t) is an edge in the residual network we're done since v_2' must be a neighbor of v_2, so $s, v_1, v_1', v_2, v_2', t$ is a path in G_f. Otherwise v_2' must have a neighbor $v_3 \in L$ such that (v_3, v_2') is in G_f. Specifically, $v_3 \neq v_1$ since (v_3, v_2') has flow 1, and $v_3 \neq v_2$ since (v_2, v_1') has flow 1, so no more flow can leave v_2 without violating conservation of flow. Again by our hypothesis, $N(\{v_1, v_2, v_3\}) \geq 3$, so there is another neighbor $v_3' \in R$.

Continuing in this fashion, we keep building up the neighborhood v_i', expanding each time we find that (v_i', t) is not an edge in G_f. This cannot happen L times, since we have run the Ford-Fulkerson while-loop fewer than $|L|$ times, so there still exist edges into t in G_f. Thus, the process must stop at some vertex v_k', and we obtain an augmenting path $s, v_1, v_1', v_2, v_2', v_3, \ldots, v_k, v_k', t$, contradicting our assumption that there was no such path. Therefore the while loop runs at least $|L|$ times. By Corollary 26.3 the flow strictly increases each time by f_p. By Theorem 26.10 f_p is an integer. In particular, it is equal to 1. This implies that $|f| \geq |L|$. It is clear that $|f| \leq |L|$, so we must have $|f| = |L|$. By

Corollary 26.11 this is the cardinality of a maximum matching. Since $|L| = |R|$, any maximum matching must be a perfect matching.

Exercise 26.3-5

We convert the bipartite graph into a flow problem by making a new vertex for the source which has an edge of unit capacity going to each of the vertices in L, and a new vertex for the sink that has an edge from each of the vertices in R, each with unit capacity. We want to show that the number of edge between the two parts of the cut is at least L, this would get us by the max-flow-min-cut theorem that there is a flow of value at least $|L|$. The, we can apply the integrality theorem that all of the flow values are integers, meaning that we are selecting $|L|$ disjoint edges between L and R.

To see that every cut must have capacity at lest $|L|$, let S_1 be the side of the cut containing the source and let S_2 be the side of the cut containing the sink. Then, look at $L \cap S_1$. The source has an edge going to each of $L \cap (S_1)^c$, and there is an edge from $R \cap S_1$ to the sink that will be cut. This means that we need that there are at least $|L \cap S_1| - |R \cap S_1|$ many edges going from $L \cap S_1|$ to $R \cap S_2$. If we look at the set of all neighbors of $L \cap S_1$, we get that there must be at least the same number of neighbors in R, because otherwise we could sum up the degrees going from $L \cap S_1$ to R on both sides, and get that some of the vertices in R would need to have a degree higher than d. This means that the number of neighbors of $L \cap S_1$ is at least $L \cap S_1$, but we have that they are in S_1, but there are only $|R \cap S_1|$ of those, so, we have that the size of the set of neighbors of $L \cap S_1$ that are in S_2 is at least $|L \cap S_1| - |R \cap S_1|$. Since each of these neighbors has an edge crossing the cut, we have that the total number of edges that the cut breaks is at least $(|L| - |L \cap S_1|) + (|L \cap S_1| - |R \cap S_1|) + |R \cap S_1| = |L|$. Since each of these edges is unit valued, the value of the cut is at least $|L|$.

Exercise 26.4-1

When we run INITIALIZE-PREFLOW(G,s), $s.e$ is zero prior to the for loop on line 7. Then, for each of the vertices that s has an edge to, we decrease the value of $s.e$ by the capacity of that edge. This means that at the end, $s.e$ is equal to the negative of the sum of all the capacities coming out of s. This is then equal to the negative of the cut value of the cut that puts s on one side, and all the other vertices on the other. The negative of the value of the min cut is larger or equal to the negative of the value of this cut. Since the value of the max flow is the value of the min cut, we have that the negative of the value of the max flow is larger or equal to $s.e$.

Exercise 26.4-2

We must select an appropriate data structure to store all the information which will allow us to select a valid operation in constant time. To do this, we will need to maintain a list of overflowing vertices. By Lemma 26.14, a push or

a relabel operation always applies to an overflowing vertex. To determine which operation to perform, we need to determine whether $u.h = v.h + 1$ for some $v \in N(u)$. We'll do this by maintaining a list $u.high$ of all neighbors of u in G_f which have height greater than or equal to u. We'll update these attributes in the PUSH and RELABEL functions. It is clear from the pseudocode given for PUSH that we can execute it in constant time, provided we have maintain the attributes $\delta_f(u, v)$, $u.e$, $c_f(u, v)$, $(u, v).f$, and $u.h$. Each time we call PUSH(u, v) the result is that u is no longer overflowing, so we must remove it from the list. Maintain a pointer $u.overflow$ to u's position in the overflow list. If a vertex u is not overflowing, set $u.overflow = NIL$. Next, check if v became overflowing. If so, set $v.overflow$ equal to the head of the overflow list. Since we can update the pointer in constant time and delete from a linked list given a pointer to the element to be deleted in constant time, we can maintain the list in $O(1)$. The RELABEL operation takes $O(V)$ because we need to compute the minimum $v.h$ from among all $(u, v) \in E_f$, and there could be $|V| - 1$ many such v. We will also need to update $u.high$ during RELABEL. When RELABEL(u) is called, set $u.high$ equal to the empty list and for each vertex v which is adjacent to u, if $v.h = u.h + 1$, add u to the list $v.high$. Since this takes constant time per adjacent vertex we can maintain the attribute in $O(V)$ per call to relabel.

Exercise 26.4-3

To run RELABEL(u), we need to take the min a number of things equal to the out degree of u (and so taking this min will take time proportional to the out degree). This means that since each vertex will only be relabeled at most $O(|V|)$ many times, the total amount of work is on the order of $|V| \sum_{v \in V} outdeg(v)$. But the sum of all the out degrees is equal to the number of edges, so, we have the previous expression is on the order of $|V||E|$.

Exercise 26.4-4

In the proof of $2 \implies 3$ in Theorem 26.6 we obtain a minimum cut by letting $S = \{v \in V |$ there exists a path from s to v in $G_f\}$ and $T = V - S$. Given a flow, we can form the residual graph in $O(E)$. Then we just need to perform a depth first search to find the vertices reachable from s. This can be done in $O(V + E)$, and since $|E| \geq |V| - 1$ the whole procedure can be carried out in $O(E)$.

Exercise 26.4-5

First, construct the flow network for the bipartite graph as in the previous section. Then, we relabel everything in L. Then, we push from every vertex in L to a vertex in R, so long as it is possible. keeping track of those that vertices of L that are still overflowing can be done by a simple bit vector. Then, we relabel everything in R and push to the last vertex. Once these operations have been done, The only possible valid operations are to relabel the vertices of L

that weren't able to find an edge that they could push their flow along, so could possibly have to get a push back from R to L. This continues until there are no more operations to do. This takes time of $O(V(E+V))$.

Exercise 26.4-6

The number of relabel operations and saturating pushes is the same as before. An edge can handle at most k nonsaturating pushes before it becomes saturated, so the number of nonsaturating pushes is at most $2k|V||E|$. Thus, the total number of basic operations is at most $2|V|^2 + 2|V||E| + 2k|V||E| = O(kVE)$.

Exercise 26.4-7

This won't affect the asymptotic performance, in fact it will improve the bound obtained in lemma 16.20 to be that no vertex will ever have a height more than $2|V| - 3$. Since this lemma was the source of all the bounds later, they carry through, and are actually a little bit (not asymptotically) better (lower).

To see that it won't affect correctness of the algorithm. We notice that the reason that we needed the height to be as high as it was was so that we could consider all the simple paths from s to t. However, when we are done initializing, we have that the only overflowing vertices are the ones for which there is an edge to them from s. Then, we only need to consider all the simple paths from them to t, the longest such one involves $|V| - 1$ vertices, and, so, only $|V| - 2$ different edges, and so it only requires that there are $|V| - 2$ differences in heights, since the set $\{0, 1, \ldots, |V| - 3\}$ has $|V| - 2$ different values, this is possible.

Exercise 26.4-8

We'll prove the claim by induction on the number of push and relabel operations. Initially, we have $u.h = |V|$ if $u = s$ and 0 otherwise. We have $s.h - |V| = 0 \leq \delta_f(s, s) = 0$ and $u.h = 0 \leq \delta_f(u, t)$ for all $u \neq s$, so the claim holds prior to the first iteration of the while loop on line 2 of the GENERIC-PUSH-RELABEL algorithm. Suppose that the properties have been maintained thus far. If the next iteration is a nonsaturating push then the properties are maintained because the heights and existence of edges in the residual network are preserved. If it is a saturating push then edge (u, v) is removed from the residual network, which increases both $\delta_f(u, t)$ and $\delta_f(u, s)$, so the properties are maintained regardless of the height of u. Now suppose that the next iteration causes a relabel of vertex u. For all v such that $(u, v) \in E_f$ we must have $u.h \leq v.h$. Let $v' = \min\{v.h|(u, v) \in E_f\}$. There are two cases to consider. First, suppose that $v'.h < |V|$. Then after the relabeling we have $u.h = 1 + v'.h \leq 1 + \min_{(u,v)\in E_f} \delta_f(v, t) = \delta_f(u, t)$. Second, suppose that $v'.h \geq |V|$. Then after relabeling we have $u.h = 1+v'.h \leq 1+|V|+\min_{(u,v)\in E_f} \delta_f(v, s) = \delta_f(u, s)+|V|$ which implies that $u.h - |V| \leq \delta)f(u, s)$. Therefore the GENERIC-PUSH-

RELABEL procedure maintains the desired properties.

Exercise 26.4-9

What we should do is to, for successive backwards neighborhoods of t, relabel everything in that neighborhood. This will only take at most $O(VE)$ time (see 26.4-3). This also has the upshot of making it so that once we are done with it, every vertex's height is equal to the quantity $\delta_f(u,t)$. Then, since we begin with equality, after doing this, the inductive step we had in the solution to the previous exercise shows that this equality is preserved.

Exercise 26.4-10

Each vertex has maximum height $2|V|-1$. Since heights don't decrease, and there are $|V|-2$ vertices which can be overflowing, the maximum contribution of relabels to Φ over all vertices is $(2|V|-1)(|V|-2)$. A saturating push from u to v increases Φ by at most $v.h \leq 2|V|-1$, and there are at most $2|V||E|$ saturating pushes, so the total contribution over all saturating pushes to Φ is at most $(2|V|-1)(2|V||E|)$. Since each nonsaturating push decrements Φ by at least on and Φ must equal zero upon termination, we must have that the number of nonsaturating pushes is at most

$$(2|V|-1)(|V|-2)+(2|V|-1)(2|V||E|) = 4|V|^2|E|+2|V|^2-5|V|+3-2|V||E|.$$

Using the fact that $|E| \geq |V|-1$ and $|V| \geq 4$ we can bound the number of saturating pushes by $4|V|^2|E|$.

Exercise 26.5-1

When we initialize the preflow, we have 26 units of flow leaving s. Then, we consider v_1 since it is the first element in the L list. When we discharge it, we increase it's height to 1 so that it can dump 12 of it's excess along its edge to vertex v_3, to discharge the rest of it, it has to increase it's height to $|V|+1$ to discharge it back to s. It was already at the front, so, we consider v_2. We increase its height to 1. Then, we send all of its excess along its edge to v_4. We move it to the front, which means we next consider v_1, and do nothing because it is not overflowing. Up next is vertex v_3. After increasing its height to 1, it can send all of its excess to t. This puts v_3 at the front, and we consider the non-overflowing vertices v_2 and v_1. Then, we consider v_4, it increases its height to 1, then sends 4 units to t. Since it still has an excess of 10 units, it increases its height once again. Then it becomes valid for it to send flow back to v_2 or to v_3. It considers v_4 first because of the ordering of its neighbor list. This means that 10 units of flow are pushed back to v_2. Since $v_4.h$ increased, it moves to the front of the list Then, we consider v_2 since it is the only still overflowing vertex. We increase its height to 3. Then, it is overflowing by 10 so it increases its height to 3 to send 6 units to v_4. It's height increased so it goes to the front

of the list. Then, we consider v_4, which is overflowing. it increases its height to 3, then it sends 6 units to v_3. Again, it goes to the front of the list. Up next is v_2 which is not overflowing, v_3 which is, so it increases it's height by 1 to send 4 units of flow to t. Then sends 2 units to v_4 after increasing in height. The excess flow keeps bobbing around the four vertices, each time requiring them to increase their height a bit to discharge to a neighbor only to have that neighbor increase to discharge it back until v_2 has increased in height enough to send all of it's excess back to s, this completes and gives us a maximum flow of 23.

Exercise 26.5-2

Initially, the vertices adjacent to s are the only ones which are overflowing. The implementation is as follows:

Algorithm 1 PUSH-RELABEL-QUEUE(G,s)

```
1: INITIALIZE-PREFLOW(G, s)
2: Initialize an empty queue q
3: for v ∈ G.Adj[s] do
4:     q.push(v)
5: end for
6: while q.head ≠ NIL do
7:     DISCHARGE(q.head)
8:     q.pop()
9: end while
```

Note that we need to modify the DISCHARGE algorithm to push vertices v onto the queue if v was not overflowing before a discharge but is overflowing after one. Between lines 7 and 8 of DISCHARGE(u), add the line "**if** $v.e > 0$, $q.push(v)$." This is an implementation of the generic push-relabel algorithm, so we know it is correct. The analysis of runtime is almost identical to that of Theorem 26.30. We just need to verify that there are at most $|V|$ calls to DISCHARGE between two consecutive relabel operations. Observe that after calling PUSH(u, v), Corollary 26.28 tells us that no admissible edges are entering v. Thus, once v is put into the queue because of the push, it can't be added again until it has been relabeled. Thus, at most $|V|$ vertices are added to the queue between relabel operations.

Exercise 26.5-3

If we change relabel to just increment the value of u, we will not be ruining the correctness of the Algorithm. This is because since it only applies when $u.h \leq v.h$, we won't be every creating a graph where h ceases to be a height function, since $u.h$ will only ever be increasing by exactly one whenever relabel is called, ensuring that $u.h + 1 \leq v.h$. This means that Lemmatae 26.15 and 26.16 will still hold. Even Corollary 26.21 holds since all it counts on is that

relabel causes some vertex's h value to increase by at least one, it will still work when we have all of the operations causing it to increase by exactly one. However, Lemma 26.28 will no longer hold. That is, it may require more than a single relabel operation to cause an admissible edge to appear, if for example, $u.h$ was strictly less than the h values of all its neighbors. However, this lemma is not used in the proof of Exercise 26.4-3, which bounds the number of relabel operations. Since the number of relabel operations still have the same bound, and we know that we can simulate the old relabel operation by doing (possibly many) of these new relabel operations, we have the same bound as in the original algorithm with this different relabel operation.

Exercise 26.5-4

We'll keep track of the heights of the overflowing vertices using an array and a series of doubly linked lists. In particular, let A be an array of size $|V|$, and let $A[i]$ store a list of the elements of height i. Now we create another list L, which is a list of lists. The head points to the list containing the vertices of highest height. The next pointer of this list points to the next nonempty list stored in A, and so on. This allows for constant time insertion of a vertex into A, and also constant time access to an element of largest height, and because all lists are doubly linked, we can add and delete elements in constant time. Essentially, we are implementing the algorithm of Exercise 26.5-2, but with the queue replaced by a priority queue with constant time operations. As before, it will suffice to show that there are at most $|V|$ calls to discharge between consecutive relabel operations.

Consider what happens when a vertex v is put into the priority queue. There must exist a vertex u for which we have called PUSH(u, v). After this, no admissible edge is entering v, so it can't be added to the priority queue again until after a relabel operation has occurred on v. Moreover, every call to DISCHARGE terminates with a PUSH, so for every call to DISCHARGE there is another vertex which can't be added until a relabel operation occurs. After $|V|$ DISCHARGE operations and no relabel operations, there are no remaining valid PUSH operations, so either the algorithm terminates, or there is a valid relabel operation which is performed. Thus, there are $O(V^3)$ calls to DISCHARGE. By carrying out the rest of the analysis of Theorem 26.30, we conclude that the runtime is $O(V^3)$.

Exercise 26.5-5

Suppose to try and obtain a contradiction that there were some minimum cut for which a vertex that had $v.h > k$ were on the sink side of that cut. For that minimum cut, there is a residual flow network for which that cut is saturated. Then, if there were any vertices that were also on the sink side of the cut which had an edge going to v in this residual flow network, since it's h value cannot be equal to k, we know that it must be greater than k since it could be only at most one less than v. We can continue in this way to let S be the set

of vertices on the sink side of the graph which have an h value greater than k. Suppose that there were some simple path from a vertex in S to s. Then, at each of these steps, the height could only decrease by at most 1, since it cannot get from above k to 0 withour going through k, we know that there is no path in the residual flow network going from a vertex in S to s. Since a minimal cut corresponds to disconnected parts of the residual graph for a maximum flow, and we know there is no path from S to s, there is a minimum cut for which S lies entirely on the source side of the cut. This was a contradiction to how we selected v,and so have shown the first claim.

Now we show that after updating the h values as suggested, we are still left with a height function. Suppose we had an edge (u, v) in the residual graph. We knew from before that $u, h \leq v.h + 1$. However, this means that if $u.h > k$, so must be $v.h$. So, if both were above k, we would be making them equal, causing the inequality to still hold. Also, if just $v.k$ were above k, then we have not decreased it's h value, meaning that the inequality also still must hold. Since we have not changed the value of $s.h$, and $t.h$, we have all the required properties to have a height function after modifying the h values as described.

Problem 26-1

a. This problem is identical to exercise 26.1-7.

b. Construct a vertex constrained flow network from the instance of the escape problem by letting our flow network have a vertex (each with unit capacity) for each intersection of grid lines, and have a bidirectional edge with unit capacity for each pair of vertices that are adjacent in the grid. Then, we will put a unit capacity edge going from s to each of the distinguished vertices, and a unit capacity edge going from each vertex on the sides of the grid to t. Then, we know that a solution to this problem will correspond to a solution to the escape problem because all of the augmenting paths will be a unit flow, because every edge has unit capacity. This means that the flows through the grid will be the paths taken. This gets us the escaping paths if the total flow is equal to m (we know it cannot be greater than m by looking at the cut which has s by itself). And, if the max flow is less than m, we know that the escape problem is not solvable, because otherwise we could construct a flow with value m from the list of disjoint paths that the people escaped along.

Problem 26-2

a. Set up the graph G' as defined in the problem, give each edge capacity 1, and run a maximum-flow algorithm. I claim that if (x_i, y_j) has flow 1 in the maximum flow and we set (i, j) to be an edge in our path cover, then the result is a minimum path cover. First observe that no vertex appears twice in the same path. If it did, then we would have $f(x_i, y_j) = f(x_k, y_j)$ for some $i \neq k \neq j$. However, this contradicts the conservation of flow, since the

capacity leaving y_j is only 1. Moreover, since the capacity from s to x_i is 1, we can never have two edges of the form (x_i, y_j) and (x_i, y_k) for $k \neq j$. We can ensure every vertex is included in some path by asserting that if there is no edge (x_i, y_j) or $(x_j, y_i$ for some j, then j will be on a path by itself. Thus, we are guaranteed to obtain a path cover. If there are k paths in a cover of n vertices, then they will consist of $n - k$ edges in total. Given a path cover, we can recover a flow by assigning edge (x_i, y_j) flow 1 if and only if (i, j) is an edge in one of the paths in the cover. Suppose that the maximum flow algorithm yields a cover with k paths, and hence flow $n - k$, but a minimum path cover uses strictly fewer than k paths. Then it must use strictly more than $n - k$ edges, so we can recover a flow which is larger than the one previously found, contradicting the fact that the previous flow was maximal. Thus, we find a minimum path cover. Since the maximum flow in the graph corresponds to finding a maximum matching in the bipartite graph obtained by considering the induced subgraph of G' on $\{1, 2, \ldots, n\}$, section 26.3 tells us that we can find a maximum flow in $O(VE)$.

b. This doesn't work for directed graphs which contain cycles. To see this, consider the graph on $\{1, 2, 3, 4\}$ which contains edges $(1, 2), (2, 3), (3, 1)$, and $(4, 3)$. The desired output would be a single path 4,3,1,2 but flow which assigns edges (x_1, y_2), (x_2, y_3), and (x_3, y_1) flow 1 is maximal.

Problem 26-3

a. Suppose to a contradiction that there were some $J_i \in T$, and some $A_k \in R_i$ so that $A_k \notin T$. However, by the definition of the flow network, there is an edge of infinite capacity going from A_k to J_i because $A_k \in R_i$. This means that there is an edge of infinite capacity that is going across the given cut. This means that the capacity of the cut is infinite, a contradiction to the given fact that the cut was finite capacity.

b. Though tempting, it doesn't suffice to just look at the experts that are on the s side of the cut. To see why this doesn't work, imagine there's one specialized skill areal, such as "Computer power switch operator', that is required for every job. Then, any finite cut that would include any job getting done would requiring that this expert be hired. However, since there is an infinite capacity edge coming from him to every other job, then all of the experts need for all the other jobs would also need to be hired. So, if we have this obiquitously required employee, any minimum cut would have to be all or nothing, but it is trivial to find a counterexample to this being optimal.

   In order for this problem to be solvable, one must assume that for every expert you've hired, you do all of the jobs that he is required for. If this is the case, then let $S_k \subseteq [n]$ be the indices of the experts that lie on the source side of the cut, and let $S_i \subseteq [m]$ be the indices of jobs that lie on the source side of the cut, then the net revenue is just

$$\sum_{S_i} p_i - \sum_{S_k} c_k$$

To see this is minimum, transferring over some set of experts and tasks from the sink side to the source side causes the capacity to go down by the cost of those experts and go up by the revenue of those jobs. If the cut was minimal than this must be a positive change, so the revenue isn't enough to justify the hire, meaning that those jobs that were on the source side in the minimal cut are exactly the jobs to attempt.

c. Again, to get a solution, we must make the assumption that for every expert that is hired, all jobs that that expert is required for must be completed. Basically just run either the $O(V^3)$ relabel-to-front algorithm described in section 26.5 on the flow network, and hire the experts that are on the source side of the cut. By the previous part, we know that this gets us the best outcome. The number of edges in the flow network is m+n+r, and the number of vertices is 2+m+n, so the runtime is just $O((2+m+n)^3)$, so it's cubic in $max(m,n)$. There is no dependence on R using this algorithm, but this is reasonable since we have the inherent bound that $r < mn$, which is a lower order term.

Without this unstated assumption, I suspect that there isn't an efficient solution possible, but cannot think of what NP-complete problem you would use for the reduction.

(Of course if he just needed experts in topics contained in this book, he could of just hired either Michelle or me)

Problem 26-4

a. If there exists a minimum cut on which (u, v) doesn't lie then the maximum flow can't be increased, so there will exist no augmenting path in the residual network. Otherwise it does cross a minimum cut, and we can possibly increase the flow by 1. Perform one iteration of Ford-Fulkerson. If there exists an augmenting path, it will be found and increased on this iteration. Since the edge capacities are integers, the flow values are all integral. Since flow strictly increases, and by an integral amount each time, a single iteration of the while loop of line 3 of Ford-Fulkerson will increase the flow by 1, which we know to be maximal. To find an augmenting path we use a BFS, which runs in $O(V + E') = O(V + E)$.

b. If the edge's flow was already at least 1 below capacity then nothing changes. Otherwise, find a path from s to t which contains (u, v) using BFS in $O(V + E)$. Decrease the flow of every edge on that path by 1. This decreases total flow by 1. Then run one iteration of the while loop of Ford-Fulkerson in $O(V + E)$. By the argument given in part a, everything is integer valued and flow strictly increases, so we will either find no augmenting path, or will increase the flow by 1 and then terminate.

Problem 26-5

a. Since the capacity of a cut is the sum of the capacity of the edges going from a vertex on one side to a vertex on the other, it is less than or equal to the sum of the capacities of all of the edges. Since each of the edges has a capacity that is $\leq C$, if we were to replace the capacity of each edge with C, we would only be potentially increasing the sum of the capacities of all the edges. After so changing the capacities of the edges, the sum of the capacities of all the edges is equal to $C|E|$, potentially an overestimate of the original capacity of any cut, and so of the minimum cut.

b. Since the capacity of a path is equal to the minimum of the capacities of each of the edges along that path, we know that any edges in the residual network that have a capacity less than K cannot be used in such an augmenting path. Similarly, so long as all the edges have a capacity of at least K, then the capacity of the augmenting path, if it is found, will be of capacity at least K. This means that all that needs be done is remove from the residual network those edges whose capacity is less than K and then run BFS.

c. Since K starts out as a power of 2, and through each iteration of the while loop on line 4, it decreases by a factor of two until it is less than 1. There will be some iteration of that loop when $K = 1$. During this iteration, we will be using any augmenting paths of capacity at least 1 when running the loop on line 5. Since the original capacities are all integers, the augmenting paths at each step will be integers, which means that no augmenting path will have a capacity of less than 1. So, once the algorithm terminates, there will be no more augmenting paths since there will be no more augmenting paths of capacity at least 1.

d. Each time line 4 is executed we know that there is no augmenting path of capacity at least $2K$. To see this fact on the initial time that line 4 is executed we just note that $2K = 2 \cdot 2^{\lfloor \lg(C) \rfloor} > 2 \cdot 2^{\lg(C)-1} = 2^{\lg(C)} = C$. Then, since an augmenting path is limited by the capacity of the smallest edge it contains, and all the edges have a capacity at most C, no augmenting path will have a capacity greater than that. On subsequent times executing line 4, the loop of line 5 during the previous execution of the outer loop will of already used up and capacious augmenting paths, and would only end once there are no more.

   Since any augmenting path must have a capacity of less than $2K$, we can look at each augmenting path p, and assign to it an edge e_p which is any edge whose capacity is tied for smallest among all the edges along the path. Then, removing all of the edges e_p would disconnect the residual network since every possible augmenting path goes through one of those edge. We know that there are at most $|E|$ of them since they are a subset of the edges. We also know that each of them has capacity at most $2K$ since that was the

value of the augmenting path they were selected to be tied for cheapest in. So, the total cost of this cut is $2K|E|$.

e. Each time that the inner while loop runs, we know that it adds an amount of flow that is at least K, since that's the value of the augmenting path. We also know that before we start that while loop, there is a cut of cost $\leq 2K|E|$. This means that the most flow we could possibly add is $2K|E|$. Combining these two facts, we get that the most cuts possible is $\frac{2K|E|}{K} = 2|E| \in O(|E|)$.

f. We only execute the outermost for loop $\lg(C)$ many times since $\lg(2^{\lfloor \lg(C) \rfloor}) \leq \lg(C)$. The inner while loop only runs $O(|E|)$ many times by the previous part. Finally, every time the inner for loop runs, the operation it does can be done in time $O(|E|)$ by part b. Putting it all together, the runtime is $O(|E|^2 \lg(C))$.

Problem 26-6

a. Suppose M is a matching and P is an augmenting path with respect to M. Then P consists of k edges in M, and $k+1$ edges not in M. This is because the first edge of P touches an unmatched vertex in L, so it cannot be in M. Similarly, the last edge in P touches an unmatched vertex in R, so the last edge cannot be in M. Since the edges alternate being in or not in M, there must be exactly one more edge not in M than in M. This implies that $|M \oplus P| = |M| + |P| - 2k = |M| + 2k + 1 - 2k = |M| + 1$ since we must remove each edge of M which is in P from both M and P. Now suppose $P_1, P_2, \ldots, P_k$ are vertex-disjoint augmenting paths with respect to M. Let k_i be the number of edges in P_i which are in M, so that $|P_i| = 2k + i + 1$. Then we have

$$M \oplus (P_1 \cup P_2 \cup \ldots \cup P_k) = |M| + |P_1| + \ldots + |P_k| - 2k_1 - 2k_2 - \ldots - 2k_k = |M| + k.$$

To see that we in fact get a matching, suppose that there was some vertex v which had at least 2 incident edges e and e'. They cannot both come from M, since M is a matching. They cannot both come from P since P is simple and every other edge of P is removed. Thus, $e \in M$ and $e' \in P \backslash M$. However, if $e \in M$ then $e \in P$, so $e \notin M \oplus P$, a contradiction. A similar argument gives the case of $M \oplus (P_1 \cup \ldots \cup P_k)$.

b. Suppose some vertex in G' has degree at least 3. Since the edges of G' come from $M \oplus M^*$, at least 2 of these edges come from the same matching. However, a matching never contains two edges with the same endpoint, so this is impossible. Thus every vertex has degree at most 2, so G' is a disjoint union of simple paths and cycles. If edge (u, v) is followed by edge (z, w) in a simple path or cycle then we must have $v = z$. Since two edges with the same endpoint cannot appear in a matching, they must belong alternately to M and M^*. Since edges alternate, every cycle has the same number of edges

in each matching and every path has at most one more edge in one matching than in the other. Thus, if $|M| \leq |M^*|$ there must be at least $|M^*| - |M|$ vertex-disjoint augmenting paths with respect to M.

c. Every vertex matched by M must be incident with some edge in M'. Since P is augmenting with respect to M', the left endpoint of the first edge of P isn't incident to a vertex touched by an edge in M'. In particular, P starts at a vertex in L which is unmatched by M since every vertex of M is incident with an edge in M'. Since P is vertex disjoint from $P_1, P_2, \ldots, P_k$, any edge of P which is in M' must in fact be in M and any edge of P which is not in M' cannot be in M. Since P has edges alternately in M' and $E - M'$, P must in fact have edges alternately in M and $E - M$. Finally, the last edge of P must be incident to a vertex in R which is unmatched by M'. Any vertex unmatched by M' is also unmatched by M, so P is an augmenting path for M. P must have length at least l since l is the length of the shortest augmenting path with respect to M. If P had length exactly l then this would contradict the fact that $P_1 \cup \ldots \cup P_k$ is a maximal set of vertex disjoint paths of length l because we could add P to the set. Thus P has more than l edges.

d. Any edge in $M \oplus M'$ is in exactly one of M or M'. Thus, the only possible contributing edges from M' are from $P_1 \cup \ldots \cup P_k$. An edge from M can contribute if and only if it is not in exactly one of M and $P_1 \cup \ldots \cup P_k$, which means it must be in both. Thus, the edges from M are redundant so $M \oplus M' = (P_1 \cup \ldots \cup P_k)$ which implies $A = (P_1 \cup \ldots \cup P_k) \oplus P$.

   Now we'll show that P is edge disjoint from each P_i. Suppose that an edge e of P is also an edge of P_i for some i. Since P is an augmenting path with respect to M', either $e \in M'$ or $e \in E - M'$. Suppose $e \in M'$. Since P is also augmenting with respect to M, we must have $e \in M$. However, if e is in M and M' then e cannot be in any of the P_i's by the definition of M'. Now suppose $e \in E - M'$. Then $e \in E - M$ since P is augmenting with respect to M. Since e is an edge of P_i, $e \in E - M'$ implies that $e \in M$, a contradiction.

   Since P has edges alternately in M' and $E - M'$ and is edge disjoint from $P_1 \cup \ldots \cup P_k$, P is also an augmenting path for M, which implies $|P| \geq l$. Since every edge in A is disjoint we conclude that $|A| \geq (k+1)l$.

e. Suppose M^* is a matching with strictly more than $|M| + |V|/(l+1)$ edges. By part b there are strictly more than $|V|/(l+1)$ vertex-disjoint augmenting paths with respect to M. Each one of these contains at least l edges, so it is incident on $l+1$ vertices. Since the paths are vertex disjoint, there are strictly more than $|V|(l+1)/(l+1)$ distinct vertices incident with these paths, a contradiction. Thus, the size of the maximum matching is at most $|M| + |V|/(l+1)$.

f. Consider what happens after iteration number $\sqrt{|V|}$. Let M^* be a maximal matching in G. Then $|M^*| \geq |M|$ so by part b, $M \oplus M^*$ contains at least $|M^*| - |M|$ vertex disjoint augmenting paths with respect to M. By part

c, each of these is also a an augmenting path for M. Since each has length $\sqrt{|V|}$, there can be at most $\sqrt{|V|}$ such paths, so $|M^*| - |M| \leq \sqrt{|V|}$. Thus, only $\sqrt{|V|}$ additional iterations of the repeat loop can occur, so there are at most $2\sqrt{|V|}$ iterations in total.

g. For each unmatched vertex in L we can perform a modified BFS to find the length of the shortest path to an unmatched vertex in R. Modify the BFS to ensure that we only traverse an edge if it causes the path to alternate between an edge in M and an edge in $E - M$. The first time an unmatched vertex in R is reached we know the length k of a shortest augmenting path. We can use this to stop our search early if at any point we have traversed more than that number of edges. To find disjoint paths, start at the vertices of R which were found at distance k in the BFS. Run a DFS backwards from these, which maintains the property that the next vertex we pick has distance one fewer, and the edges alternate between being in M and $E - M$. As we build up a path, mark the vertices as used so that we never traverse them again. This takes $O(E)$, so by part f the total runtime is $O(\sqrt{V}E)$.