26 Maximum Flow

26.1 Flow network

26.1-1

> Show that splitting an edge in a flow network yields an equivalent network. More formally, suppose that flow network G contains edge (u, v), and we create a new flow network G' by creating a new vertex x and replacing (u, v) by new edges (u, x) and (x, v) with $c(u, x) = c(x, v) = c(u, v)$. Show that a maximum flow in G' has the same value as a maximum flow in G.

Suppose the maximum flow of a graph $G = (V, E)$ with source s and destination t is $|f| = \sum f(s, v)$, where $v \in V$ are vertices in the maximum flow between s and t.

We know every vertex $v \in V$ must obey the Flow conservation rule. Therefore, if we can add or delete some vertices between s and t without changing $|f|$ or violating the Flow conversation rule, then the new graph $G' = (V', E')$ will have the same maximum flow as the original graph G, and that's why we can replace edge (u, v) by new edges (u, x) and (x, v) with $c(u, x) = c(x, v) = c(u, v)$.

After doing so, vertex v_1 and v_2 still obey the Flow conservation rule since the values flow in to or flow out of v_1 and v_2 do not change at all. Meanwhile, the value $|f| = \sum f(s, v)$ remains the same.

In fact, we can split any edges in this way, even if two vertex u and v doesn't have any connection between them, we can still add a vertex y and make $c(u, y) = c(y, v) = 0$.

To conclude, we can transform any graph with or without antiparallel edges into an equivalent graph without antiparallel edges and have the same maximum flow value.

26.1-2

> Extend the flow properties and definitions to the multiple-source, multiple-sink problem. Show that any flow in a multiple-source, multiple-sink flow network corresponds to a flow of identical value in the single-source, single-sink network obtained by adding a supersource and a supersink, and vice versa.

Capacity constraint: for all $u, v \in V$, we require $0 \le f(u, v) \le c(u, v)$.

Flow conservation: for all $u \in V - S - T$, we require $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$.

26.1-3

> Suppose that a flow network $G = (V, E)$ violates the assumption that the network contains a path $s \rightsquigarrow v \rightsquigarrow t$ for all vertices $v \in V$. Let u be a vertex for which there is no path $s \rightsquigarrow u \rightsquigarrow t$. Show that there must exist a maximum flow f in G such that $f(u, v) = f(v, u) = 0$ for all vertices $v \in V$.

(Removed)

26.1-4

> Let f be a flow in a network, and let α be a real number. The ***scalar flow product***, denoted αf, is a function from $V \times V$ to $\mathbb{R}$ defined by
>
> $$(\alpha f)(u, v) = \alpha \cdot f(u, v).$$
>
> Prove that the flows in a network form a ***convex set***. That is, show that if f_1 and f_2 are flows, then so is $\alpha f_1 + (1 - \alpha) f_2$ for all α in the range $0 \le \alpha \le 1$.

(Removed)

26.1-5

> State the maximum-flow problem as a linear-programming problem.

$$\begin{aligned} \max \quad & \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) \\ \text{s. t.} \quad & 0 \le f(u, v) \le c(u, v) \\ & \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) = 0 \end{aligned}$$

26.1-6

> Professor Adam has two children who, unfortunately, dislike each other. The problem is so severe that not only do they refuse to walk to school together, but in fact each one refuses to walk on any block that the other child has stepped on that day. The children have no problem with their paths crossing at a corner. Fortunately both the professor's house and the school are on corners, but beyond that he is not sure if it is going to be possible to send both of his children to the same school. The professor has a map of his town. Show how to formulate the problem of determining whether both his children can go to the same school as a maximum-flow problem.

(Removed)

26.1-7

> Suppose that, in addition to edge capacities, a flow network has ***vertex capacities***. That is each vertex v has a limit $l(v)$ on how much flow can pass though v. Show how to transform a flow network $G = (V, E)$ with vertex capacities into an equivalent flow network $G' = (V', E')$ without vertex capacities, such that a maximum flow in G' has the same value as a maximum flow in G. How many vertices and edges does G' have?

(Removed)

26.2 The Ford-Fulkerson method

26.2-1

> Prove that the summations in equation (26.6) equal the summations in equation (26.7).

(Removed)

26.2-2

> In Figure 26.1(b), what is the flow across the cut $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$? What is the capacity of this cut?

$$\begin{aligned} f(S,T) &= f(s,v_1) + f(v_2,v_1) + f(v_4,v_3) + f(v_4,t) - f(v_3,v_2) = 11 + 1 + 7 + 4 - 4 = 19, \\ c(S,T) &= c(s,v_1) + c(v_2,v_1) + c(v_4,v_3) + c(v_4,t) = 16 + 4 + 7 + 4 = 31. \end{aligned}$$

26.2-3

> Show the execution of the Edmonds-Karp algorithm on the flow network of Figure 26.1(a).

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 $\{s, v_2, v_4, v_3, t\}$ 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.

26.2-4

> In the example of Figure 26.6, what is the minimum cut corresponding to the maximum flow shown? Of the augmenting paths appearing in the example, which one cancels flow?

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).

26.2-5

> Recall that the construction in Section 26.1 that converts a flow network with multiple sources and sinks into a single-source, single-sink network adds edges with infinite capacity. Prove that any flow in the resulting network has a finite value if the edges of the original network with multiple sources and sinks have finite capacity.

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.

26.2-6

> Suppose that each source s_i in a flow network with multiple sources and sinks produces exactly p_i units of flow, so that $\sum_{v \in V} f(s_i, v) = p_i$. Suppose also that each sink t_j consumes exactly q_j units, so that $\sum_{v \in V} f(v, t_j) = q_j$, where $\sum_i p_i = \sum_j q_j$. Show how to convert the problem of finding a flow f that obeys these additional constraints into the problem of finding a maximum flow in a single-source, single-sink flow network.

$c(s, s_i) = p_i$, $c(t_j, t) = q_j$.

26.2-7

> Prove Lemma 26.2.

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.

26.2-8

> Suppose that we redefine the residual network to disallow edges into s. Argue that the procedure FORD-FULKERSON still correctly computes a maximum flow.

(Removed)

26.2-9

> Suppose that both f and f' are flows in a network G and we compute flow $f \uparrow f'$. Does the augmented flow satisfy the flow conservation property? Does it satisfy the capacity constraint?

(Removed)

26.2-10

> Show how to find a maximum flow in a network $G = (V, E)$ by a sequence of at most $|E|$ augmenting paths. (Hint: Determine the paths after finding the maximum flow.)

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.

26.2-11

> The ***edge connectivity*** of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show how to determine the edge connectivity of an undirected graph $G = (V, E)$ by running a maximum-flow algorithm on at most $|V|$ flow networks, each having $O(V)$ vertices and $O(E)$ edges.

Create an directed version of the graph. Then create a flow network out of it, resolving all antiparallel edges. All edges' capacities are set to 1. Pick any vertex that wasn't created for antiparallel workaround as the sink and run maximum-flow algorithm with all vertexes that aren't for antipararrel workaround (except the sink) as sources. Find the minimum value out of all $|V| - 1$ maximum flow values.

26.2-12

> Suppose that you are given a flow network G, and G has edges entering the source s. Let f be a flow in G in which one of the edges (v, s) entering the source has $f(v, s) = 1$. Prove that there must exist another flow f' with $f'(v, s) = 0$ such that $|f| = |f'|$. Give an $O(E)$-time algorithm to compute f', given f, and assuming that all edge capacities are integers.

(Removed)

26.2-13

> Suppose that you wish to find, among all minimum cuts in a flow network G with integral capacities, one that contains the smallest number of edges. Show how to modify the capacities of G to create a new flow network G' in which any minimum cut in G' is a minimum cut with the smallest number of edges in G.

(Removed)

26.3 Maximum bipartite matching

26.3-1

> Run the Ford-Fulkerson algorithm on the flow network in Figure 26.8 (c) and show the residual network after each flow augmentation. Number the vertices in L top to bottom from 1 to 5 and in R top to bottom from 6 to 9. For each iteration, pick the augmenting path that is lexicographically smallest.

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 $(7, t)$.

26.3-2

> Prove Theorem 26.10.

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 nth iteration. On the $(n + 1)$th 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 $(n + 1)$th 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.

26.3-3

> Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let G' be its corresponding flow network. Give a good upper bound on the length of any augmenting path found in G' during the execution of FORD-FULKERSON.

(Removed)

26.3-4 ⋆

> A ***perfect matching*** is a matching in which every vertex is matched. Let $G = (V, E)$ be an undirected bipartite graph with vertex partition $V = L \cup R$, where $|L| = |R|$. For any $X \subseteq V$, define the ***neighborhood*** of X as
>
> $$N(X) = \{y \in V : (x, y) \in E \text{ for some } x \in X\},$$
>
> that is, the set of vertices adjacent to some member of X. Prove ***Hall's theorem***: there exists a perfect matching in G if and only if $|A| \leq |N(A)|$ for every subset $A \subseteq L$.

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', and 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.

26.3-5 ⋆

> We say that a bipartite graph $G = (V, E)$, where $V = L \cup R$, is d***-regular*** if every vertex $v \in V$ has degree exactly d. Every d-regular bipartite graph has $|L| = |R|$. Prove that every d-regular bipartite graph has a matching of cardinality $|L|$ by arguing that a minimum cut of the corresponding flow network has capacity $|L|$.

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 $|L|$.

26.4 Push-relabel algorithms

26.4-1

> Prove that, after the procedure $\text{INITIALIZE-PREFLOW}(G, S)$ terminates, we have $s.e \leq -|f^*|$, where f^* is a maximum flow for G.

(Removed)

26.4-2

> Show how to implement the generic push-relabel algorithm using $O(V)$ time per relabel operation, $O(1)$ time per push, and $O(1)$ time to select an applicable operation, for a total time of $O(V^2E)$.

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 $\text{PUSH}(u, v)$ 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 $u.overflow = \text{NIL}$. 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 $\text{RELABEL}(u)$ 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.

26.4-3

> Prove that the generic push-relabel algorithm spends a total of only $O(VE)$ time in performing all the $O(V^2)$ relabel operations.

(Removed)

26.4-4

> Suppose that we have found a maximum flow in a flow network $G = (V, E)$ using a push-relabel algorithm. Give a fast algorithm to find a minimum cut in G.

(Removed)

26.4-5

> Give an efficient push-relabel algorithm to find a maximum matching in a bipartite graph. Analyze your algorithm.

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 $O(V(E+V))$.

26.4-6

> Suppose that all edge capacities in a flow network $G=(V,E)$ are in the set $\{1,2,\ldots,k\}$. Analyze the running time of the generic push-relabel algorithm in terms of $|V|$, $|E|$, and k. (Hint: How many times can each edge support a nonsaturating push before it becomes saturated?)

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 $2|V|^2+2|V||E|+2k|V||E|=O(kVE)$.

26.4-7

> Show that we could change line 6 of INITIALIZE-PREFLOW to
>
> ```
> 6 s.h = |G.V| - 2
> ```
>
> without affecting the correctness or asymptotic performance of the generic pushrelabel algorithm.

(Removed)

26.4-8

> Let $\delta_f(u,v)$ be the distance (number of edges) from u to v in the residual network G_f. Show that the GENERIC-PUSH-RELABEL procedure maintains the properties that $u.h<|V|$ implies $u.h\le\delta_f(u,t)$ and that $u.h\ge|V|$ implies $u.h-|V|\le\delta_f(u,s)$.

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 $s.h-|V|=0\le\delta_f(s,s)=0$ and $u.h=0\le\delta_f(u,t)$ 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 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.

26.4-9 ⋆

> As in the previous exercise, let $\delta_f(u, v)$ be the distance from u to v in the residual network G_f. Show how to modify the generic push-relabel algorithm to maintain the property that $u.h < |V|$ implies $u.h = \delta_f(u, t)$ and that $u.h \geq |V|$ implies $u.h - |V| = \delta_f(u, s)$. The total time that your implementation dedicates to maintaining this property should be $O(VE)$.

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.

26.4-10

> Show that the number of nonsaturating pushes executed by the GENERIC-PUSH-RELABEL procedure on a flow network $G = (V, E)$ is at most $4|V|^2|E|$ for $|V| \geq 4$.

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 $4|V|^2|E|$.

26.5 The relabel-to-front algorithm

26.5-1

> Illustrate the execution of RELABEL-TO-FRONT in the manner of Figure 26.10 for the flow network in Figure 26.1(a). Assume that the initial ordering of vertices in L is $\langle v_1, v_2, v_3, v_4 \rangle$ and that the neighbor lists are
>
> $$\begin{aligned} v_1.N &= \langle s, v_2, v_3 \rangle, \\ v_2.N &= \langle s, v_1, v_3, v_4 \rangle, \\ v_3.N &= \langle v_1, v_2, v_4, t \rangle, \\ v_4.N &= \langle v_2, v_3, t \rangle. \end{aligned}$$

When we initialize the preflow, we have 29 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 9 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_2 first because of the ordering of its neighbor list. This means that 9 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 9 so it increases its height to 3 to send 9 units to v_4. It's height increased so it goes to the of the list. Then, we consider v_4, which is overflowing. It pushes 7 units to v_3. Since it is still overflowing by 2, it increases its height to 4 and pushes the rest back to v_2 and goes to the front of the list. Up next is v_2, which increases its height by 2 to sends its overflow to v_4. 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. Last but not least, v_3 pushes its overflow of 7 units to t, and gives us a maximum flow of 23.

26.5-2 ⋆

> We would like to implement a push-relabel algorithm in which we maintain a firstin, first-out queue of overflowing vertices. The algorithm repeatedly discharges the vertex at the head of the queue, and any vertices that were not overflowing before the discharge but are overflowing afterward are placed at the end of the queue. After the vertex at the head of the queue is discharged, it is removed. When the queue is empty, the algorithm terminates. Show how to implement this algorithm to compute a maximum flow in $O(V^3)$ time.

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

```
PUSH-RELABEL-QUEUE(G, s)
    INITIALIZE-PREFLOW(G, s)
    let q be a new empty queue
    for v ∈ G.Adj[s]
        PUSH(q, v)
    while q.head != NIL
        DISCHARGE(q.head)
        POP(q)
```

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 $\text{DISCHARGE}(u)$, add the line "if $v.e > 0$, $\text{PUSH}(q, v)$." 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 $\text{PUSH}(u, v)$, 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.

26.5-3

> Show that the generic algorithm still works if RELABEL updates $u.h$ by simply computing $u.h = u.h + 1$. How would this change affect the analysis of RELABEL-TO-FRONT?

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 1 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 1, it will still work when we have all of the operations causing it to increase by exactly 1. 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.

26.5-4 ⋆

> Show that if we always discharge a highest overflowing vertex, we can make the push-relabel method run in $O(V^3)$ time.

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 $\text{PUSH}(u, v)$. After this, no ad- missible 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 $O(V^3)$.

26.5-5

> Suppose that at some point in the execution of a push-relabel algorithm, there exists an integer $0 < k \leq |V| - 1$ for which no vertex has $v.h = k$. Show that all vertices with $v.h > k$ are on the source side of a minimum cut. If such a k exists, the ***gap heuristic*** updates every vertex $v \in V - \{s\}$ for which $v.h > k$, to set $v.h = \max(v.h, |V| + 1)$. Show that the resulting attribute h is a height function. (The gap heuristic is crucial in making implementations of the push-relabel method perform well in practice.)

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 without 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 Escape problem

> An $n \times n$ ***grid*** is an undirected graph consisting of n rows and n columns of vertices, as shown in Figure 26.11. We denote the vertex in the ith row and the jth column by (i, j). All vertices in a grid have exactly four neighbors, except for the boundary vertices, which are the points (i, j) for which $i = 1$, $i = n$, $j = 1$, or $j = n$.
>
> Given $m \leq n^2$ starting points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ in the grid, the ***escape problem*** is to determine whether or not there are m vertex-disjoint paths from the starting points to any m different points on the boundary. For example, the grid in Figure 26.11(a) has an escape, but the grid in Figure 26.11(b) does not.
>
> **a.** Consider a flow network in which vertices, as well as edges, have capacities. That is, the total positive flow entering any given vertex is subject to a capacity constraint. Show that determining the maximum flow in a network with edge and vertex capacities can be reduced to an ordinary maximum-flow problem on a flow network of comparable size.
>
> **b.** Describe an efficient algorithm to solve the escape problem, and analyze its running time.

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 Minimum path cover

> A ***path cover*** of a directed graph $G = (V, E)$ is a set P of vertex-disjoint paths such that every vertex in V is included in exactly one path in P. Paths may start and end anywhere, and they may be of any length, including 0. A ***minimum path cover*** of G is a path cover containing the fewest possible paths.
>
> **a.** Give an efficient algorithm to find a minimum path cover of a directed acyclic graph $G = (V, E)$. (Hint: Assuming that $V = \{1, 2, \ldots, n\}$, construct the graph $G' = (V', E')$, where

$$V' = \{x_0, x_1, \ldots, x_n\} \cup \{y_0, y_1, \ldots, y_n\},$$
$$E' = \{(x_0, x_i) : i \in V\} \cup \{(y_i, y_0) : i \in V\} \cup \{(x_i, y_j) : (i, j) \in E\},$$

and run a maximum-flow algorithm.)

b. Does your algorithm work for directed graphs that contain cycles? Explain.

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 Algorithmic consulting

Professor Gore wants to open up an algorithmic consulting company. He has identified n important subareas of algorithms (roughly corresponding to different portions of this textbook), which he represents by the set $A = \{A_1, A_2, \ldots, A_n\}$. In each subarea A_k, he can hire an expert in that area for c_k dollars. The consulting company has lined up a set $J = \{J_1, J_2, \ldots, J_m\}$ of potential jobs. In order to perform job J_i, the company needs to have hired experts in a subset $R_i \subseteq A$ of subareas. Each expert can work on multiple jobs simultaneously. If the company chooses to accept job J_i, it must have hired experts in all subareas in R_i, and it will take in revenue of p_i dollars.

Professor Gore's job is to determine which subareas to hire experts in and which jobs to accept in order to maximize the net revenue, which is the total income from jobs accepted minus the total cost of employing the experts.

Consider the following flow network G. It contains a source vertex s, vertices $A_1, A_2, \ldots, A_n$, vertices $J_1, J_2, \ldots, J_m$, and a sink vertex t. For $k = 1, 2, \ldots, n$, the flow network contains an edge (s, A_k) with capacity $c(s, A_k) = c_k$, and for $i = 1, 2, \ldots, m$, the flow network contains an edge (J_i, t) with capacity

> $c(J_i, t) = p_i$. For $k = 1, 2, \ldots, n$ and $i = 1, 2, \ldots, m$, if $A_k \in R_i$, then G contains an edge (A_k, J_i) with capacity $c(A_k, J_i) = \infty$.
>
> **a.** Show that if $J_i \in T$ for a finite-capacity cut (S, T) of G, then $A_k \in T$ for each $A_k \in R_i$.
>
> **b.** Show how to determine the maximum net revenue from the capacity of a minimum cut of G and the given p_i values.
>
> **c.** Give an efficient algorithm to determine which jobs to accept and which experts to hire. Analyze the running time of your algorithm in terms of m, n, and $r = \sum_{i=1}^{m} |R_i|$.

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.

Problem 26-4 Updating maximum flow

> Let $G = (V, E)$ be a flow network with source s, sink t, and integer capacities. Suppose that we are given a maximum flow in G.
>
> **a.** Suppose that we increase the capacity of a single edge $(u, v) \in E$ by 1. Give an $O(V + E)$-time algorithm to update the maximum flow.
>
> **b.** Suppose that we decrease the capacity of a single edge $(u, v) \in E$ by 1. Give an $O(V + E)$-time algorithm to update the maximum flow.

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 Maximum flow by scaling

> Let $G = (V, E)$ be a flow network with source s, sink t, and an integer capacity $c(u, v)$ on each edge $(u, v) \in E$. Let $C = \max_{(u,v)\in E} c(u, v)$.
>
> **a.** Argue that a minimum cut of G has capacity at most $C|E|$.
>
> **b.** For a given number K, show how to find an augmenting path of capacity at least K in $O(E)$ time, if such a path exists.
>
> We can use the following modification of $\text{FORD-FULKERSON-METHOD}$ to compute a maximum flow in G:

```
MAX-FLOW-BY-SCALING(G, s, t)
    C = max_{(u, v) ∈ E} c(u, v)
    initialize flow f to 0
    K = 2^{floor(lg C)}
    while K ≥ 1
        while there exists an augmenting path p of capacity at least K
augment flow f along p
```

```
            K = K / 2
        return f
```

c. Argue that MAX-FLOW-BY-SCALING returns a maximum flow.

d. Show that the capacity of a minimum cut of the residual network G_f is at most $2K|E|$ each time line 4 is executed.

e. Argue that the inner **while** loop of lines 5–6 executes $O(E)$ times for each value of K.

f. Conclude that MAX-FLOW-BY-SCALING can be implemented so that it runs in $O(E^2 \lg C)$ time.

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 The Hopcroft-Karp bipartite matching algorithm

In this problem, we describe a faster algorithm, due to Hopcroft and Karp, for p finding a maximum matching in a bipartite graph. The algorithm runs in $O(\sqrt{V}E)$ time. Given an undirected, bipartite graph $G = (V, E)$, where $V = L \cup R$ and all edges have exactly one endpoint in L, let M be a matching in G. We say that a simple path P in G is an ***augmenting path*** with respect to M if it starts at an unmatched vertex in L, ends at an unmatched vertex in R, and its edges belong alternately to M and $E - M$. (This definition of an augmenting path is related to, but different from, an augmenting path in a flow network.) In this problem, we treat a path as a sequence of edges, rather than as a sequence of vertices. A shortest augmenting path with respect to a matching M is an augmenting path with a minimum number of edges.

Given two sets A and B, the ***symmetric difference*** $A \oplus B$ is defined as $(A - B) \cup (B - A)$, that is, the elements that are in exactly one of the two sets.

a. Show that if M is a matching and P is an augmenting path with respect to M, then the symmetric difference $M \oplus P$ is a matching and $|M \oplus P| = |M| + 1$. Show that if $P_1, P_2, \ldots, P_k$ are vertex-disjoint augmenting paths with respect to M, then the symmetric difference $M \oplus (P_1 \cup P_2 \cup \cdots \cup P_k)$ is a matching with cardinality $|M| + k$.

The general structure of our algorithm is the following:

```
HOPCROPFT-KARP(G)
    M = ∅
    repeat
        let P = {P[1], P[2], ..., P[k]} be a maximal set of vertex-
disjoint shortest augmenting paths with respect to M
        M = M ⊕ (P[1] ∪ P[2] ∪ ... ∪ P[k])
    until P == ∅
    return M
```

The remainder of this problem asks you to analyze the number of iterations in the algorithm (that is, the number of iterations in the **repeat** loop) and to describe an implementation of line 3.

> **b.** Given two matchings M and M^* in G, show that every vertex in the graph $G' = (V, M \oplus M^*)$ has degree at most 2. Conclude that G' is a disjoint union of simple paths or cycles. Argue that edges in each such simple path or cycle belong alternately to M or M^*. Prove that if $|M| \leq |M^*|$, then $M \oplus M^*$ contains at least $|M^*| - |M|$ vertex-disjoint augmenting paths with respect to M.
>
> Let l be the length of a shortest augmenting path with respect to a matching M, and let $P_1, P_2, \ldots, P_k$ be a maximal set of vertex-disjoint augmenting paths of length l with respect to M. Let $M' = M \oplus (P_1 \cup \cdots \cup P_k)$, and suppose that P is a shortest augmenting path with respect to M'.
>
> **c.** Show that if P is vertex-disjoint from $P_1, P_2, \ldots, P_k$, then P has more than l edges.
>
> **d.** Now suppose that P is not vertex-disjoint from $P_1, P_2, \ldots, P_k$. Let A be the set of edges $(M \oplus M') \oplus P$. Show that $A = (P_1 \cup P_2 \cup \cdots \cup P_k) \oplus P$ and that $|A| \geq (k+1)l$. Conclude that P has more than l edges.
>
> **e.** Prove that if a shortest augmenting path with respect to M has l edges, the size of the maximum matching is at most $|M| + |V|/(l+1)$.
>
> **f.** Show that the number of **repeat** loop iterations in the algorithm is at most $2\sqrt{|V|}$. (Hint: By how much can M grow after iteration number $\sqrt{|V|}$?)
>
> **g.** Give an algorithm that runs in $O(E)$ time to find a maximal set of vertexdisjoint shortest augmenting paths $P_1, P_2, \ldots, P_k$ for a given matching M. Conclude that the total running time of HOPCROFT-KARP is $O(\sqrt{V}E)$.

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

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

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 $M \oplus (P_1 \cup \cdots \cup P_k)$.

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

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 $P_1 \cup \cdots \cup P_k$, 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 $O(\sqrt{V}E)$.