Cornell University, Fall 2012 — **CS 6820: Algorithms**
Lecture notes on approximation algorithms — **October 10–15, 2010**

1 Introduction: Approximation Algorithms

For many important optimization problems, there is no known polynomial-time algorithm to compute the exact optimum. In fact, when we discuss the topic of NP-completeness later in the semester, we'll see that a great many such problems are all equivalently hard to solve, in the sense that the existence of a polynomial-time algorithm for solving any one of them would imply polynomial-time algorithms for all the rest.

The study of approximation algorithms arose as a way to circumvent the apparent hardness of these problems by relaxing the algorithm designer's goal: instead of trying to compute an exactly optimal solution, we aim to compute a solution whose value is as close as possible to that of the optimal solution. However, in some situations it is desirable to run an approximation algorithm even when there exists a polynomial-time algorithm for computing an exactly optimal solution. For example, the approximation algorithm may have the benefit of faster running time, a lower space requirement, or it may lend itself more easily to a parallel or distributed implementation. These considerations become especially important when computing on "big data," where the input size is so astronomical that a running time which is a high-degree polynomial of the input size (or even quadratic, for that matter) cannot really be considered an efficient algorithm, at least on present-day hardware.

To make the definition of approximation algorithms precise, we say that an algorithm ALG for a maximization problem is an α-approximation algorithm (or that its approximation factor is α) if the following inequality holds for every input instance x:

$$\mathrm{ALG}(x) \le \mathrm{OPT}(x) \le \alpha \cdot \mathrm{ALG}(x).$$

Here $\mathrm{OPT}(x)$ denotes the value of the problem's objective function when evaluated on the optimal solution to input instance x, and $\mathrm{ALG}(x)$ denotes the algorithm's output when it runs on input instance x. Note that the definition only requires the algorithm to output a number (approximating the value of the optimal solution) and not the approximate solution itself. In most cases, it is possible to design the algorithm so that it also outputs a solution attaining the value $\mathrm{ALG}(x)$, but in these notes we adopt a definition of approximation algorithm that does not require the algorithm to do so.

Similarly, for a minimization problem, an α-approximation algorithm must satisfy

$$\mathrm{OPT}(x) \le \mathrm{ALG}(x) \le \alpha \cdot \mathrm{OPT}(x).$$

Note that in both cases the approximation factor α is a number greater than or equal to 1.

2 Approximation Algorithms Based on Linear Programming

Linear programming is an extremely versatile technique for designing approximation algorithms, because it is one of the most general and expressive problems that we know how to solve in polynomial time. In this section we'll discuss three applications of linear programming to the design and analysis of approximation algorithms.

2.1 LP Rounding Algorithm for Weighted Vertex Cover

In an undirected graph $G = (V, E)$, if $S \subseteq V$ is a set of vertices and e is an edge, we say that S *covers* e if at least one endpoint of e belongs to S. We say that S is a *vertex cover* if it covers every edge. In the weighted vertex cover problem, one is given an undirected graph $G = (V, E)$ and a weight $w_v \geq 0$ for each vertex v, and one must find a vertex cover of minimum combined weight.

We can express the weighted vertex cover problem as an integer program, by using *decision variables* x_v for all $v \in V$ that encode whether $v \in S$. For any set $S \subseteq V$ we can define a vector $\mathbf{x}$, with components indexed by vertices of G, by specifying that

$$x_v = \begin{cases} 1 & \text{if } v \in S \\ 0 & \text{otherwise.} \end{cases}$$

S is a vertex cover if and only if the constraint $x_u + x_v \geq 1$ is satisfied for every edge $e = (u, v)$. Conversely, if $\mathbf{x} \in \{0, 1\}^V$ satisfies $x_u + x_v \geq 1$ for every edge $e = (u, v)$ then the set $S = \{v \mid x_v = 1\}$ is a vertex cover. Thus, the weighted vertex cover problem can be expressed as the following integer program.

$$\begin{array}{lll} \min & \sum_{v \in V} w_v x_v & \\ \text{s.t.} & x_u + x_v \geq 1 & \forall e = (u, v) \in E \\ & x_v \in \{0, 1\} & \forall v \in V \end{array} \tag{1}$$

To design an approximation algorithm for weighted vertex cover, we will transform this integer program into a linear program by relaxing the constraint $x_v \in \{0, 1\}$ to allow the variables x_v to take fractional values.

$$\begin{array}{lll} \min & \sum_{v \in V} w_v x_v & \\ \text{s.t.} & x_u + x_v \geq 1 & \forall e = (u, v) \in E \\ & x_v \geq 0 & \forall v \in V \end{array} \tag{2}$$

It may seem more natural to replace the constraint $x_v \in \{0, 1\}$ with $x_v \in [0, 1]$ rather than $x_v \geq 0$, but the point is that an optimal solution of the linear program will never assign any of the variables x_v a value strictly greater than 1, because the value of any such variable could always be reduced to 1 without violating any constraints, and this would only improve the objective function $\sum_v w_v x_v$. Thus, writing the constraint as $x_v \geq 0$ rather than $x_v \in [0, 1]$ is without loss of generality.

It is instructive to present an example of a fractional solution of (2) that achieves a strictly lower weight than any integer solution. One such example is when G is a 3-cycle with vertices u, v, w, each having weight 1. Then the vector $\mathbf{x} = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}\right)$ satisfies all of the constraints of (2) and the objective function evaluates to $\frac{3}{2}$ at $\mathbf{x}$. In contrast, the minimum weight of a vertex cover of the 3-cycle is 2.

We can solve the linear program (2) in polynomial time, but as we have just seen, the solution may be fractional. In that case, we need to figure out how we are going to post-process the fractional solution to obtain an actual vertex cover. In this case, the natural idea of rounding to the nearest integer works. Let $\mathbf{x}$ be an optimal solution of the linear program (2) and define

$$\tilde{x}_v = \begin{cases} 1 & \text{if } x_v \geq 1/2 \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

Now let $S = \{v \mid \tilde{x}_v = 1\}$. Note that S is a vertex cover because for every edge $e = (u, v)$ the constraint $x_u + x_v \geq 1$ implies that at least one of x_u, x_v is greater than or equal to $1/2$.

Finally, the analyze the approximation ratio of this algorithm, we observe that the rounding rule (3) has the property that for all v,

$$\tilde{x}_v \leq 2x_v.$$

Letting S denote the vertex cover chosen by our LP rounding algorithm, and letting OPT denote the optimum vertex cover, we have

$$\sum_{v\in S} w_v = \sum_{v\in V} w_v\tilde{x}_v \leq 2\sum_{v\in V} w_v x_v \leq 2\sum_{v\in \mathrm{OPT}} w_v,$$

where the final inequality holds because the fractional optimum of the linear program (2) must be less than or equal to the optimum of the integer program (1) because its feasible region is at least as big.

2.2 Primal-Dual Algorithm for Weighted Vertex Cover

The algorithm presented in the preceding section runs in polynomial time, and we have seen that it outputs a vertex cover whose weight is at most twice the weight of the optimum vertex cover, a fact that we express by saying that its *approximation factor* is 2.

However, the algorithm needs to solve a linear program and although this can be done in polynomial time, there are much faster ways to compute a vertex cover with approximation factor 2 without solving the linear program. One such algorithm, that we present in this section, is a *primal-dual approximation algorithm*, meaning that it makes choices guided by the linear program (2) and its dual but does not actually solve them to optimality.

Let us write the linear programming relaxation of weighted vertex cover once again, along with its dual.

$$\begin{array}{lll} \min & \sum_{v\in V} w_v x_v & \\ \text{s.t.} & x_u + x_v \geq 1 & \forall e = (u,v) \in E \\ & x_v \geq 0 & \forall v \in V \end{array} \tag{4}$$

$$\begin{array}{lll} \max & \sum_{e\in E} y_e & \\ \text{s.t.} & \sum_{e\in\delta(v)} y_e \leq w_v & \forall v \in V \\ & y_e \geq 0 & \forall e \in E \end{array} \tag{5}$$

Here, the notation $\delta(v)$ denotes the set of all edges having v as an endpoint. One may interpret the dual LP variable y_e as *prices* associated to the edges, and one may interpret w_v as the *wealth* of vertex v. The dual constraint $\sum_{e\in\delta(v)} y_e \leq w_v$ asserts that v has enough wealth to pay for all of the edges incident to it. If edge prices satisfy all the constraints of (5) then *every* vertex has enough wealth to pay for its incident edges, and consequently every vertex set S has enough combined wealth to pay for all of the edges covered by S. In particular, if S is a vertex cover then the combined wealth of the vertices in S must be at least $\sum_{e\in E} y_e$, which is a manifestation of *weak duality*: the optimum value of the dual LP is a lower bound on the optimum value of the primal LP.

The dual LP insists that we maximize the combined price of all edges, subject to the constraint that each vertex has enough wealth to pay for all the edges it covers. Rather than exactly maximizing the combined price of all edges, we will set edge prices using a natural (but suboptimal) greedy heuristic: go through the edges in arbitrary order, increasing the price of each one as much as possible without violating the dual constraints. This results in the following algorithm.

Algorithm 1 Primal-dual algorithm for vertex cover

```
1: Initialize $S = \emptyset$, $y_e = 0 \; \forall e \in E$, $s_v = 0 \; \forall v \in V$.
2: for all $e \in E$ do
3:     $\delta = \min\{w_u - s_u, w_v - s_v\}$
4:     $y_e = y_e + \delta$
5:     $s_u = s_u + \delta$
6:     $s_v = s_v + \delta$
7:     if $s_u = w_u$ then
8:         $S = S \cup \{u\}$
9:     end if
10:    if $s_v = w_v$ then
11:        $S = S \cup \{v\}$
12:    end if
13: end for
14: return $S$
```

The variables s_v keep track of the sum $\sum_{e \in \delta(v)} y_e$ (i.e., the left-hand side of the dual constraint corresponding to vertex v) as it grows during the execution of the algorithm. The rule for updating S by inserting each vertex v such that $s_v = w_v$ is inspired by the principle of *complementary slackness* from the theory of linear programming duality: if x^* is an optimal solution of a primal linear program and y^* is an optimal solution of the dual, then for every i such that $x_i^* \neq 0$ the i^{th} dual constraint must be satisfied with equality by y^*; similarly, for every j such that $y_j^* \neq 0$, the j^{th} primal constraint is satisfied with equality by x^*. Thus, it is natural that our decisions of which vertices to include in our vertex cover (primal solution) should be guided by keeping track of which dual constraints are tight ($s_v = w_v$).

It is clear that each iteration of the main loop runs in constant time, so the algorithm runs in linear time. At the end of the loop processing edge $e = (u, v)$, at least one of the vertices u, v must belong to S. Therefore, S is a vertex cover. To conclude the analysis we need to prove that the approximation factor is 2. To do so, we note the following loop invariants — statements that hold at the beginning and end of each execution of the **for** loop, though not necessarily in the middle. Each of them is easily proven by induction on the number of iterations of the **for** loop.

1. $\mathbf{y}$ is a feasible vector for the dual linear program.

2. $s_v = \sum_{e \in \delta(v)} y_e$.

3. $S = \{v \mid s_v = w_v\}$.

4. $\sum_{v \in V} s_v = 2 \sum_{e \in E} y_e$.

Now the proof of the approximation factor is easy. Recalling that $\sum_{e \in E} y_e \leq \sum_{v \in \text{OPT}} w_v$ by weak duality, we find that

$$\sum_{v \in S} w_v = \sum_{v \in S} s_v \leq \sum_{v \in V} s_v = 2 \sum_{e \in E} y_e \leq 2 \sum_{v \in \text{OPT}} w_v.$$

2.3 Greedy Algorithm for Weighted Set Cover

Vertex cover is a special case of the *set cover* problem, in which there is a set U of n elements, and there are m subsets $S_1, \ldots, S_m \subseteq U$, with positive weights $w_1, \ldots, w_m$. The goal is to choose a subcollection of the m subsets (indexed by an index set $\mathscr{I} \subseteq \{1, \ldots, m\}$), such that $\bigcup_{i \in \mathscr{I}} S_i = U$, and to minimize the combined weight $\sum_{i \in \mathscr{I}} w_i$. We will analyze the following natural *greedy algorithm* that chooses sets according to a "minimum weight per new element covered" criterion. (The variable T in the pseudocode below keeps track of the set of elements that are not yet covered by $\bigcup_{i \in \mathscr{I}} S_i$.)

Algorithm 2 Greedy algorithm for set cover

```
1: Initialize $\mathscr{I} = \emptyset, T = U$.
2: while $T \neq \emptyset$ do
3:     $i = \arg\min_k \left\{ \frac{w_k}{|T \cap S_k|} \;\middle|\; 1 \le k \le m,\ T \cap S_k \neq \emptyset \right\}$.
4:     $\mathscr{I} = \mathscr{I} \cup \{i\}$.
5:     $T = T \setminus S_i$.
6: end while
7: return $\mathscr{I}$
```

It is clear that the algorithm runs in polynomial time and outputs a valid set cover. To analyze the approximation ratio, we will use the linear programming relaxation of set cover and its dual.

$$\begin{array}{lll} \min & \sum_{i=1}^{m} w_i x_i & \\ \text{s.t.} & \sum_{i : j \in S_i} x_i \ge 1 & \forall j \in U \\ & x_i \ge 0 & \forall i = 1, \ldots, m \end{array} \tag{6}$$

$$\begin{array}{lll} \max & \sum_{j \in U} y_j & \\ \text{s.t.} & \sum_{j \in S_i} y_j \le w_i & \forall i = 1, \ldots, m \\ & y_j \ge 0 & \forall j \in U \end{array} \tag{7}$$

It will be helpful to rewrite the greedy set cover algorithm by adding some extra lines that do not influence the choice of which sets to place in $\mathscr{I}$, but merely compute extra data relevant to the analysis. Specifically, in the course of choosing sets to include in $\mathscr{I}$, we also compute a vector $\mathbf{z}$ indexed by elements of U. This is not a feasible solution of the dual LP, but at the end of the algorithm we scale it down to obtain another vector $\mathbf{y}$ that is feasible for the dual LP. The scale factor α will constitute an upper bound on the algorithm's approximation ratio. This is called the method of *dual fitting*.

Algorithm 3 Greedy algorithm for set cover

```
1: Initialize $\mathscr{I} = \emptyset$, $T = U$, $z_j = 0 \; \forall j \in U$.
2: while $T \neq \emptyset$ do
3:     $i = \arg\min_k \left\{ \frac{w_k}{|T \cap S_k|} \;\middle|\; 1 \le k \le m,\ T \cap S_k \neq \emptyset \right\}$.
4:     $\mathscr{I} = \mathscr{I} \cup \{i\}$.
5:     for all $j \in T \cap S_i$ do
6:         $z_j = \frac{w_i}{|T \cap S_i|}$.
7:     end for
8:     $T = T \setminus S_i$.
9: end while
10: $\alpha = 1 + \ln\left(\max_{1 \le i \le m} |S_i|\right)$.
11: $\mathbf{y} = \frac{1}{\alpha}\mathbf{z}$.
12: return $\mathscr{I}$
```

The following three loop invariants are easily shown to hold at the beginning and end of each **while** loop iteration, by induction on the number of iterations.

1. $\sum_{j \in U} z_j = \sum_{i \in \mathscr{I}} w_i$.
2. For all $j \in U$, if the algorithm ever assigns a nonzero value to z_j then that value never changes afterward.

Below, in Lemma 1, we will show that the vector $\mathbf{y}$ is feasible for the dual LP (7). From this, it follows that the approximation ratio is bounded above by $\alpha = 1 + \ln\left(\max_{1 \le i \le m} |S_i|\right)$. To see this, observe that

$$\sum_{i \in \mathscr{I}} w_i = \sum_{j \in U} z_j = \alpha \sum_{j \in U} y_j \le \alpha \sum_{i \in \mathrm{OPT}} w_i,$$

where the last line follows from weak duality.

Lemma 1. *The vector* $\mathbf{y}$ *computed in Algorithm* (3) *is feasible for the dual linear program* (7).

Proof. Clearly $y_j \ge 0$ for all j, so the only thing we really need to show is that $\sum_{j \in S_i} y_j \le w_i$ for every set S_i. Let $p = |S_i|$ and denote the elements of S_i by $s_0, s_1, \ldots, s_{p-1}$, where the numbering corresponds to the order in which nonzero values were assigned to the variables z_j by Algorithm 3. Thus, a nonzero value was assigned to z_{s_0} before z_{s_1}, and so on. We know that

$$z_{s_0} \le \frac{w_i}{p} \tag{8}$$

because at the time the value z_{s_0} was assigned, all of the elements of S_i still belonged to T. In that iteration of the while loop, the cost-effectiveness of S_i was judged to be w_i/p, the algorithm chose a set with the same or better cost-effectiveness, and all of the values z_j assigned during that iteration of the while loop were set equal to the cost-effectiveness of that set. Similarly, we know that for all $q < p$,

$$z_{s_q} \le \frac{w_i}{p - q} \tag{9}$$

because at the time the value z_{s_q} was assigned, all of the elements $s_q, s_{q+1}, \ldots, s_{p-1}$ still belonged to T. In that iteration of the while loop, the cost-effectiveness of S_i was judged to be $w_i/(p-q)$ or smaller, the algorithm chose a set with the same or better cost-effectiveness, and all of the values z_j assigned during that iteration of the while loop were set equal to the cost-effectiveness of that set.

Summing the bounds (9) for $q = 0, \ldots, p-1$, we find that

$$\sum_{j \in S_i} z_j \leq w_i \cdot \left(\frac{1}{p} + \frac{1}{p-1} + \cdots + \frac{1}{2} + 1 \right) < w_i \cdot \left(1 + \int_1^p \frac{dt}{t} \right) = w_i \cdot (1 + \ln p) \, .$$

The lemma follows upon dividing both sides by α. $\square$

3 Randomized Approximation Algorithms

Randomized techniques give rise to some of the simplest and most elegant approximation algorithms. This section gives several examples.

3.1 A Randomized 2-Approximation for Max-Cut

In the max-cut problem, one is given an undirected graph $G = (V, E)$ and a positive weight w_e for each edge, and one must output a partition of V into two subsets A, B so as to maximize the combined weight of the edges having one endpoint in A and the other in B.

We will analyze the following extremely simple randomized algorithm: assign each vertex at random to A to B with equal probability, such that the random decisions for the different vertices are mutually independent. Let $E(A, B)$ denote the (random) set of edges with one endpoint in A and the other endpoint in B. The expected weight of our cut is

$$\mathbb{E} \left(\sum_{e \in E(A,B)} w_e \right) = \sum_{e \in E} w_e \cdot \Pr(e \in E(A, B)) = \frac{1}{2} \sum_{e \in E} w_e.$$

Since the combined weight of all edges in the graph is an obvious upper bound on the weight of any cut, this shows that the expected weight of the cut produced by our algorithm is at least half the weight of the maximum cut.

3.1.1 Derandomization using pairwise independent hashing

In analyzing the expected weight of the cut defined by our randomized algorithm, we never really used the full power of our assumption that the random decisions for the different vertices are mutually independent. The only property we needed was that for each pair of vertices u, v, the probability that u and v make different decisions is exactly $\frac{1}{2}$. It turns out that one can achieve this property using only $k = \lceil \log_2(n) \rceil$ independent random coin tosses, rather than n independent random coin tosses.

Let $\mathbb{F}_2$ denote the field $\{0, 1\}$ under the operations of addition and multiplication modulo 2. Assign to each vertex v a distinct vector $\mathbf{x}(v)$ in the vector space $\mathbb{F}_2^k$; our choice of $k = \lceil \log_2(n) \rceil$ ensures that the vector space contains enough elements to assign a distinct one to each vertex. Now let r be a uniformly random vector in $\mathbb{F}_2^k$, and partition the vertex set V into the subsets

$$A_{\mathbf{r}} = \{ v \mid \mathbf{r} \cdot \mathbf{x}(v) = 0 \}$$
$$B_{\mathbf{r}} = \{ v \mid \mathbf{r} \cdot \mathbf{x}(v) = 1 \}.$$

For any edge $e = (u, v)$, the probability that $e \in E(A_{\mathbf{r}}, B_{\mathbf{r}})$ is equal to the probability that $\mathbf{r} \cdot (\mathbf{x}(v) - \mathbf{x}(u))$ is nonzero. For any fixed nonzero vector $\mathbf{w} \in \mathbb{F}_2^k$, we have $\Pr(\mathbf{r} \cdot \mathbf{w} \neq 0) = \frac{1}{2}$ because the set of $\mathbf{r}$ satisfying $\mathbf{r} \cdot \mathbf{w} = 0$ is a linear subspace of $\mathbb{F}_2^k$ of dimension $k - 1$, hence exactly 2^{k-1} of the 2^k possible vectors r have zero dot product with $\mathbf{w}$ and the other 2^{k-1} of them have nonzero dot product with $\mathbf{w}$. Thus, if we sample $\mathbf{r} \in \mathbb{F}_2^k$ uniformly at random, the expected weight of the cut defined by $(A_{\mathbf{r}}, B_{\mathbf{r}})$ is at least half the weight of the maximum cut.

The vector space $\mathbb{F}_2^k$ has only $2^k = O(n)$ vectors in it, which suggests a deterministic alternative to our randomized algorithm. Instead of choosing $\mathbf{r}$ at random, we compute the weight of the cut $(A_{\mathbf{r}}, B_{\mathbf{r}})$ for every $r \in \mathbb{F}_2^k$ and take the one with maximum weight. This is at least as good as choosing r at random, so we get a deterministic 2-approximation algorithm at the cost of increasing the running time by a factor of $O(n)$.

3.1.2 Derandomization using conditional expectations

A different approach for converting randomization approximation algorithms into deterministic ones is the *method of conditional expectations.* In this technique, rather than making all of our random decisions simultaneously, we make them sequentially. Then, instead of making the decisions by choosing randomly between two alternatives, we evaluate both alternatives according to the conditional expectation of the objective function if we fix the decision (and all preceding ones) but make the remaining ones at random. Then we choose the alternative that optimizes this conditional expectation.

To apply this technique to the randomized max-cut algorithm, we imagine maintaining a partition of the vertex set into three sets A, B, C while the algorithm is running. Sets A, B are the two pieces of the partition we are constructing. Set C contains all the vertices that have not yet been assigned. Initially $C = V$ and $A = B = \emptyset$. When the algorithm terminates C will be empty. At an intermediate stage when we have constructed a partial partition (A, B) but C contains some unassigned vertices, we can imagine assigning each element of C randomly to A or B with equal probability, independently of the other elements of C. If we were to do this, the expected weight of the random cut produced by this procedure would be

$$w(A, B, C) = \sum_{e \in E(A,B)} w_e + \frac{1}{2} \sum_{e \in E(A,C)} w_e + \frac{1}{2} \sum_{e \in E(B,C)} w_e + \frac{1}{2} \sum_{e \in E(C,C)} w_e.$$

This suggests the following deterministic algorithm that considers vertices one by one, assigning them to either A or B using the function $w(A, B, C)$ to guide its decisions.

Algorithm 4 Derandomized max-cut algorithm using method of conditional expectations

```
1: Initialize $A = B = \emptyset$, $C = V$.
2: for all $v \in V$ do
3:     Compute $w(A + v, B, C - v)$ and $w(A, B + v, C - v)$.
4:     if $w(A + v, B, C - v) > w(A, B + v, C - v)$ then
5:         $A = A + v$
6:     else
7:         $B = B + v$
8:     end if
9:     $C = C - v$
10: end for
11: return $A, B$
```

The analysis of the algorithm is based on the simple observation that for every partition of V into three sets A, B, C and every $v \in C$, we have

$$\frac{1}{2}w(A+v, B, C-v) + \frac{1}{2}w(A, B+v, C-v) = w(A, B, C).$$

Consequently

$$\max\{w(A+v, B, C-v),\ w(A, B+v, C-v)\} \geq w(A, B, C)$$

so the value of $w(A, B, C)$ never decreases during the execution of the algorithm. Initially the value of $w(A, B, C)$ is equal to $\frac{1}{2}\sum_{e \in E} w_e$, whereas when the algorithm terminates the value of $w(A, B, C)$ is equal to $\sum_{e \in E(A,B)} w_e$. We have thus proven that the algorithm computes a partition (A, B) such that the weight of the cut is at least half the combined weight of all edges in the graph.

Before concluding our discussion of this algorithm, it's worth noting that the algorithm can be simplified by observing that

$$w(A+v, B, C-v) - w(A, B+v, C-v) = \frac{1}{2}\sum_{e \in E(B,v)} w_e - \frac{1}{2}\sum_{e \in E(A,v)} w_e.$$

The algorithm runs faster if we skip the step of actually computing $w(A+v, B, C-v)$ and jump straight to computing their difference. This also means that there's no need to explicitly keep track of the vertex set C.

Algorithm 5 Derandomized max-cut algorithm using method of conditional expectations

```
1: Initialize $A = B = \emptyset$.
2: for all $v \in V$ do
3:     if $\sum_{e \in E(B,v)} w_e - \sum_{e \in E(A,v)} w_e > 0$ then
4:         $A = A + v$
5:     else
6:         $B = B + v$
7:     end if
8: end for
9: return $A, B$
```

This version of the algorithm runs in linear time: the amount of time spent on the loop iteration that processes vertex v is proportional to the length of the adjacency list of that vertex. It's also easy to prove that the algorithm has approximation factor 2 without resorting to any discussion of random variables and their conditional expectations. One simply observes that the property

$$\sum_{e \in E(A,B)} w_e \geq \sum_{e \in E(A,A)} w_e + \sum_{e \in E(B,B)} w_e$$

is a loop invariant of the algorithm. The fact that this property holds at termination implies that $\sum_{e \in E(A,B)} w_e \geq \frac{1}{2}\sum_{e \in E} w_e$ and hence the algorithm's approximation factor is 2.

3.1.3 Semidefinite programming and the Goemans-Williamson algorithm

So far, in our discussion of max-cut, we have made no mention of linear programming. It's worth considering for a moment the question of whether the natural linear programming relaxation of the max-cut problem can achieve an approximation factor better than 2. It's actually not so easy to write down the natural linear programming relaxation of max-cut. We can define decision variables $\{x_v \mid v \in V\}$, taking values in $[0,1]$, with the intended semantics that $x_v = 0$ if $v \in A$ and $x_v = 1$ if $v \in B$. The trouble is that the natural way to write the objective function is $\sum_{e \in E} |x_u - x_v|$, which is not a linear function because the absolute value function is non-linear. A workaround is to define a variable y_e for each edge e, with the intended semantics that $y_e = 1$ if e crosses the cut (A,B) and otherwise $y_e = 0$. This suggests the following linear programming relaxation of max-cut.

$$\begin{array}{lll} \max & \sum_{e \in E} y_e & \\ \text{s.t.} & y_e \le x_u + x_v & \forall e = (u,v) \in E \\ & y_e \le (1 - x_u) + (1 - x_v) & \forall e = (u,v) \in E \\ & 0 \le x_v \le 1 & \forall v \in V \end{array} \tag{10}$$

As noted earlier, for every partition of V into two sets A, B we can set $x_v = 0$ if $v \in A$, $x_v = 1$ if $v \in B$, and $y_e = 1$ if and only if e crosses the cut (A,B); this yields a valid integer solution of the linear program (10) such that the LP objective value equals the number of edges crossing the cut.

Unfortunately, for every graph the linear program (10) also has a fractional solution whose objective value equals m, the number of edges in the graph. Namely, setting $x_v = \frac{1}{2}$ for all v and $y_e = 1$ for all e satisfies the constraints of the linear program and achieves the objective value m. Since there exists graphs whose maximum cut contains barely more than half the edges (e.g., a complete graph on n vertices), solving this LP relaxation of max-cut only yields a 2-approximation, the same approximation factor achieved by the much simpler algorithms presented above.

For many years, it was not known whether *any* polynomial-time approximation algorithm for max-cut could achieve an approximation factor better than 2. Then in 1994, Michel Goemans and David Williamson discovered an algorithm with approximation factor roughly 1.14, based on *semidefinite programming* (SDP). Since then, SDP has found an increasing number of applications algorithm design, not only in approximation algorithms (where SDP has many other applications besides max-cut), but also in machine learning and high-dimensional statistics, coding theory, and other areas.

So what is semidefinite programming? An SDP is an optimization problem that seeks the maximum of a linear function over the set of symmetric positive semidefinite $n \times n$ matrices, subject to linear inequality constraints. See Lemma 2 below for the definition and some basic properties of symmetric positive semidefinite matrices. For now, we limit ourselves to the following remarks motivating why semidefinite programming is a powerful tool.

1. Semidefinite programming is solvable (to any desired precision) in polynomial time. The solution may contain irrational numbers, even if the input is made up exclusively of rational numbers, which is why the parenthetical remark about "to any desired precision" was necessary.

2. One of the main themes of algorithm design is, "When you have a discrete optimization problem that you don't know how to solve, formulate a related continuous optimization problem that you *do* know how to solve, and then try to figure out how to transform a solution of the continuous problem into a solution (either exact or approximate) or the original discrete problem. An obvious consequence of this theme is: *any time someone discovers a continuous optimization problem that can be solved by an efficient algorithm, that's a potential opportunity to design better algorithms for discrete optimization problems too.* This remark alone could justify the importance of SDP's in algorithm design.

3. Any linear program can be reformulated as a semidefinite program, by optimizing over the set of *diagonal* positive semidefinite matrices. Thus, SDP is at least as powerful as LP.

4. Often, one thinks of LP as relaxing a discrete optimization problem by allowing $\{0,1\}$-valued quantities to take continuous (scalar) values. In the same spirit, SDP can be thought of as further relaxing the problem by allowing scalar quantities to take (potentially high-dimensional) vector values.

To define semidefinite programming, we start with a lemma about real symmetric matrices. Any matrix satisfying the equivalent conditions listed in the lemma is called a *symmetric positive semidefinite (PSD)* matrix. The notation $A \succeq 0$ denotes the fact that A is PSD.

Lemma 2. *For a real symmetric matrix A, the following properties are equivalent.*

1. *Every eigenvalue of A is non-negative.*

2. *A can be expressed as a weighted sum of the form*

$$A = \sum_{i=1}^{m} c_i y_i y_i^{\mathsf{T}} \tag{11}$$

where the coefficients c_i are non-negative.

3. *$A = XX^{\mathsf{T}}$ for some matrix X.*

4. *There exists a vector space containing vectors $\mathbf{x}_1, \dots, \mathbf{x}_n$ such that A is the matrix of dot products of these vectors, i.e. for $1 \le i, j \le n$, the dot product $\mathbf{x}_i \cdot \mathbf{x}_j$ occurs in the i^{th} row and j^{th} column of A.*

5. *For every vector z, A satisfies the inequality $z^{\mathsf{T}} A z \ge 0$.*

Proof. To prove that the first property implies the second, we use the fact that every real symmetric matrix is orthogonally diagonalizable, i.e. that A can be written in the form $A = QDQ^{\mathsf{T}}$ where Q is an orthogonal matrix and D is a diagonal matrix whose entries are the eigenvalues of A. Defining c_i to be the i^{th} diagonal entry of D, and y_i to be the i^{th} column of Q, the equation $A = QDQ^{\mathsf{T}}$ expands as $A = \sum_{i=1}^{n} c_i y_i y_i^{\mathsf{T}}$, as desired.

It is elementary to prove that (2) $\rightarrow$ (3) $\rightarrow$ (5) $\rightarrow$ (1), as follows. If A is represented by a weighted sum as in (11), then $A = XX^{\mathsf{T}}$ where X is a matrix with k columns whose i^{th} column is $\sqrt{c_i} \cdot y_i$. If $A = XX^{\mathsf{T}}$, then for every z we have $z^{\mathsf{T}} A z = (z^{\mathsf{T}} X)(X^{\mathsf{T}} z) = \|X^{\mathsf{T}} z\|^2 \ge 0$. If the inequality $z^{\mathsf{T}} A z \ge 0$ is satisfied for every vector z, then in particular it is satisfied whenever z is an eigenvector of A with eigenvalue λ. This implies that $0 \le z^{\mathsf{T}} A z = z^{\mathsf{T}}(\lambda z) = \lambda \|z\|^2$ and hence that $\lambda \ge 0$. Having proven that (1) $\rightarrow$ (2) in the preceding paragraph, we may now conclude that all properties except possibly the fourth are equivalent. Finally, observe that the fourth property is simply a restatement of the third, adopting the notation $\mathbf{x}_i$ to denote the vector representing the i^{th} row of X. Thus, (3) and (4) are trivially equivalent. $\square$

To begin relating the max-cut problem to a semidefinite program, it begins to reformulate max-cut as a problem about labeling vertices of a graph with $\{\pm 1\}$ rather than $\{0, 1\}$. Encoding cuts in that way, we can write max-cut as the following *quadratic* optimization problem.

$$\begin{array}{lll} \max & \frac{1}{2}\sum_{e=(u,v)\in E}(1 - x_u x_v) & \\ \text{s.t.} & x_v^2 = 1 & \forall v \in V \end{array} \tag{12}$$

The next step in transforming this into a semidefinite program is to treat the variables x_v as vectors rather than scalars. We'll change the notation to $\mathbf{x}_v$ to reflect this shift.

$$\begin{array}{lll} \max & \frac{1}{2}\sum_{e=(u,v)\in E}(1 - \mathbf{x}_u \cdot \mathbf{x}_v) & \\ \text{s.t.} & \|\mathbf{x}_v\|^2 = 1 & \forall v \in V \end{array} \tag{13}$$

For every $\{\pm 1\}$ solution of (12) there is a corresponding solution of (13) in which the vectors $\{\mathbf{x}_v \mid v \in V\}$ are all equal to $\pm\mathbf{w}$ for some fixed unit vector $\mathbf{w}$. On the other hand, there are also solutions of (13) that do not correspond to any $\{\pm 1\}$ solution of (12), so (13) is a relaxation of (12) and an optimal solution of (13) might be able to achieve a strictly higher objective value than the size of the maximum cut in the graph G.

Using Lemma 2 we know that (13) is equivalent to solving the following semidefinite program to optimize over PSD matrices $A = (a_{uv})$ whose entries are indexed by ordered pairs of vertices of G.

$$\begin{array}{lll} \max & \frac{1}{2}\sum_{e=(u,v)\in E} 1 - a_{uv} & \\ \text{s.t.} & a_{vv} = 1 & \forall v \in V \\ & A \succeq 0 & \end{array} \tag{14}$$

Furthermore, given the matrix A that solves (14), we can obtain, in polynomial time, a matrix X such that $A = XX^{\mathsf{T}}$. (This is tantamount to computing the eigenvalues and eigenvectors of A. It would be more accurate to say that we can compute X to arbitrary precision in polynomial time.) The rows of this X constitute the vectors $\mathbf{x}_v$ representing a solution of (13).

The question now arises: given the vectors $\{\mathbf{x}_v\}$, how do we output a cut in G such that the number of edges in the cut is approximately as large as $\frac{1}{2}\sum(1 - \mathbf{x}_u \cdot \mathbf{x}_v)$? The idea that works in this algorithms, and many other applications of semidefinite programming, is to choose a random hyperplane through the origin, partitioning $\mathbb{R}^n$ into two halfspaces. Then we partition the vertices of the graph according to which halfspace their corresponding vector belongs to. If (u, v) is an edge of G with corresponding vectors $\mathbf{x}_u, \mathbf{x}_v$, and if θ denotes the angle between those two vectors, then the probability that the random halfspace separates $\mathbf{x}_u$ from $\mathbf{x}_v$ is θ/π, whereas the contribution of edge (u, v) to the objective function of the SDP is $\frac{1}{2}(1 - \mathbf{x}_u \cdot \mathbf{x}_v) = \frac{1}{2}(1 - \cos\theta)$. It is an elementary exercise in calculus to prove that

$$\forall \theta \in [0, \pi] \; \frac{\theta}{\pi} \geq (0.878) \cdot \left[\frac{1}{2}(1 - \cos\theta)\right].$$

Therefore, the approximation ratio of the algorithm is at most $1/(0.878) \approx 1.14$.

3.2 A Randomized 2-Approximation for Vertex Cover

For the unweighted vertex cover problem (the special case of weighted vertex cover in which $w_v = 1$ for all v) the following incredibly simple algorithm is a randomized 2-approximation.

Algorithm 6 Randomized approximation algorithm for unweighted vertex cover

```
1: Initialize $S = \emptyset$.
2: for all $e = (u, v) \in E$ do
3:     if neither $u$ nor $v$ belongs to $S$ then
4:         Randomly choose $u$ or $v$ with equal probability.
5:         Add the chosen vertex into $S$.
6:     end if
7: end for
8: return $S$
```

Clearly, the algorithm runs in linear time and always outputs a vertex cover. To analyze its approximation ratio, as usual, we define an appropriate loop invariant. Let OPT denote any vertex cover of minimum cardinality. Let S_i denote the contents of the set S after completing the i^{th} iteration of the loop. We claim that for all i,

$$\mathbb{E}[|S_i \cap \text{OPT}|] \geq \mathbb{E}[|S_i \setminus \text{OPT}|]. \tag{15}$$

The proof is by induction on i. In a loop iteration in which $e = (u, v)$ is already covered by S_{i-1}, we have $S_i = S_{i-1}$ so (15) clearly holds. In a loop iteration in which $e = (u, v)$ is not yet covered, we know that at least one of u, v belongs to OPT. Thus, the left side of (15) has probability at least 1/2 of increasing by 1, while the right side of (15) has probability at most 1/2 of increasing by 1. This completes the proof of the induction step.

Consequently, letting S denote the random vertex cover generated by the algorithm, we have $\mathbb{E}[|S \cap \text{OPT}|] \geq \mathbb{E}[|S \setminus \text{OPT}|]$ from which it easily follows that $\mathbb{E}[|S|] \leq 2 \cdot |\text{OPT}|$.

The same algorithm design and analysis technique can be applied to weighted vertex cover. In that case, we choose a random endpoint of an uncovered edge (u, v) with probability inversely proportional to the weight of that endpoint.

Algorithm 7 Randomized approximation algorithm for weighted vertex cover

```
1: Initialize $S = \emptyset$.
2: for all $e = (u, v) \in E$ do
3:     if neither $u$ nor $v$ belongs to $S$ then
4:         Randomly choose $u$ with probability $\frac{w_v}{w_u+w_v}$ and $v$ with probability $\frac{w_u}{w_u+w_v}$.
5:         Add the chosen vertex into $S$.
6:     end if
7: end for
8: return $S$
```

The loop invariant is

$$\mathbb{E}\left[\sum_{v \in S_i \cap \text{OPT}} w_v\right] \geq \mathbb{E}\left[\sum_{v \in S_i \setminus \text{OPT}} w_v\right].$$

In a loop iteration when (u, v) is uncovered, the expected increase in the left side is at least $\frac{w_u w_v}{w_u+w_v}$ whereas the expected increase in the right side is at most $\frac{w_u w_v}{w_u+w_v}$.

4 Linear Programming with Randomized Rounding

Linear programming and randomization turn out to be a very powerful when used in combination. We will illustrate this by presenting an algorithm of Raghavan and Thompson for a problem of routing paths in a network to minimize congestion. The analysis of the algorithm depends on the *Chernoff bound*, a fact from probability theory that is one of the most useful tools for analyzing randomized algorithms.

4.1 The Chernoff bound

The Chernoff bound is a very useful theorem concerning the sum of a large number of independent random variables. Roughly speaking, it asserts that for any fixed $\beta > 1$, the probability of the sum exceeding its expected value by a factor greater than β tends to zero exponentially fast as the expected sum tends to infinity.

Theorem 3. *Let $X_1, \ldots, X_n$ be independent random variables taking values in $[0,1]$, let X denote their sum, and let $\mu = \mathbb{E}[X]$. For every $\beta > 1$,*

$$\Pr(X \geq \beta\mu) < e^{-\mu[\beta\ln(\beta)-(\beta-1)]}. \tag{16}$$

For every $\beta < 1$,

$$\Pr(X \leq \beta\mu) < e^{-\mu[\beta\ln(\beta)-(\beta-1)]}. \tag{17}$$

Proof. The key idea in the proof is to make use of the moment-generating function of X, defined to be the following function of a real-valued parameter t:

$$M_X(t) = \mathbb{E}\left[e^{tX}\right].$$

From the independence of $X_1, \ldots, X_n$, we derive:

$$M_X(t) = \mathbb{E}\left[e^{tX_1}e^{tX_2}\cdots e^{tX_n}\right] = \prod_{i=1}^{n}\mathbb{E}\left[e^{tX_i}\right]. \tag{18}$$

To bound each term of the product, we reason as follows. Let Y_i be a $\{0,1\}$-valued random variable whose distribution, conditional on the value of X_i, satisfies $\Pr(Y_i = 1 \mid X_i) = X_i$. Then for each $x \in [0,1]$ we have

$$\mathbb{E}\left[e^{tY_i}\middle| X_i = x\right] = xe^t + (1-x)e^0 \geq e^{tx} = \mathbb{E}\left[e^{tX_i}\middle| X_i = x\right],$$

where the inequality in the middle of the line uses the fact that e^{tx} is a convex function. Since this inequality holds for every value of x, we can integrate over x to remove the conditioning, obtaining

$$\mathbb{E}\left[e^{tY_i}\right] \geq \mathbb{E}\left[e^{tX_i}\right].$$

Letting μ_i denote $\mathbb{E}[X_i] = \Pr(Y_i = 1)$ we find that

$$\left[e^{tX_i}\right] \leq \left[e^{tY_i}\right] = \mu_i e^t + (1-\mu_i) = 1 + \mu_i(e^t - 1) \leq \exp\left(\mu_i(e^t-1)\right),$$

where $\exp(x)$ denotes e^x, and the last inequality holds because $1 + x \leq \exp(x)$ for all x. Now substituting this upper bound back into (18) we find that

$$\mathbb{E}\left[e^{tX}\right] \leq \prod_{i=1}^{n}\exp\left(\mu_i(e^t-1)\right) = \exp\left(\mu(e^t-1)\right).$$

The proof now splits into two parts depending on whether $\beta > 1$ or $\beta < 1$. In both cases we will be choosing $t = \ln\beta$ for a reason to be disclosed later. If $\beta > 1$ then $t = \ln\beta > 0$, hence $e^{tX} \geq e^{t\beta\mu}$ whenever $X \geq \beta\mu$. Since $e^{tX} > 0$ regardless, we have $\mathbb{E}\left[e^{tX}\right] \geq e^{t\beta\mu}\Pr(X \geq \beta\mu)$ and

$$\Pr(X \geq \beta\mu) \leq \exp\left(\mu(e^t - 1 - \beta t)\right). \tag{19}$$

If $\beta < 1$ then $t = \ln\beta < 0$, hence $e^{tX} \geq e^{t\beta\mu}$ whenever $X \leq \beta\mu$. Since $e^{tX} > 0$ regardless, we have $\mathbb{E}\left[e^{tX}\right] \geq e^{t\beta\mu}\Pr(X \leq \beta\mu)$ and

$$\Pr(X \leq \beta\mu) \leq \exp\left(\mu(e^t - 1 - \beta t)\right). \tag{20}$$

In both cases, our choice of t is designed to minimize the right-hand side of (19) or (20); elementary calculus reveals that the global minimum is attained when $t = \ln\beta$. Substituting this value of t into (19) and (20) completes the proof of the theorem. $\square$

Corollary 4. *Suppose $X_1, \ldots, X_k$ are independent random variables taking values in $[0,1]$, such that $\mathbb{E}[X_1 + \cdots + X_k] \leq 1$. Then for any $N > 2$ and any $b \geq \frac{3\log N}{\log\log N}$, where $\log$ denotes the base-2 logarithm, we have*

$$\Pr(X_1 + \cdots + X_k \geq b) < \frac{1}{N}. \tag{21}$$

Proof. Let $\mu = \mathbb{E}[X_1 + \cdots + X_k]$ and $\beta = b/\mu$. Applying Theorem 3 we find that

$$\begin{aligned}\Pr(X_1 + \cdots + X_k \geq b) &\leq \exp\left(-\mu\beta\ln(\beta) + \mu\beta - \mu\right) \\ &= \exp\left(-b(\ln(\beta) - 1) - \mu\right) \leq e^{-b(\ln(\beta/e))}.\end{aligned} \tag{22}$$

Now, $\beta = b/\mu \geq b$, so

$$\frac{\beta}{e} \geq \frac{b}{e} \geq \frac{3\log N}{e\log\log N}$$

and

$$\begin{aligned}b\ln\left(\frac{\beta}{e}\right) &\geq \left(\frac{3\ln N}{\ln(\log N)}\right)\cdot\ln\left(\frac{3\log N}{e\log\log N}\right) \\ &= 3\ln(N)\cdot\left(1 - \frac{\ln(\log\log N) - \ln(3) + 1}{\ln(\log N)}\right) > \ln(N),\end{aligned} \tag{23}$$

where the last inequality holds because one can verify that $\ln(\log x) - \ln(3) + 1 < \frac{2}{3}\ln(x)$ for all $x > 1$ using basic calculus. Now, exponentiating both sides of (23) and combining with (22) we obtain the bound $\Pr(X_1 + \cdots + X_k \geq b) < 1/N$, as claimed. $\square$

4.2 An approximation algorithm for congestion minimization

We will design an approximation algorithm for the following optimization problem. The input consists of a directed graph $G = (V, E)$ with positive integer edge capacities c_e, and a set of source-sink pairs (s_i, t_i), $i = 1, \ldots, k$, where each (s_i, t_i) is a pair of vertices such that G contains at least one path from s_i to t_i. The algorithm must output a list of paths $P_1, \ldots, P_k$ such that P_i is a path from s_i to t_i. The load on edge e, denoted by ℓ_e, is defined to be the number of paths P_i that traverse edge e. The congestion of edge e is the ratio ℓ_e/c_e, and the algorithm's objective is to minimize congestion, i.e. minimize the value of $\max_{e\in E}(\ell_e/c_e)$. This problem turns out to be NP-hard, although we will not prove that fact here.

The first step in designing our approximation algorithm is to come up with a linear programming relaxation. To do so, we define a decision variable $x_{i,e}$ for each $i = 1, \dots, k$ and each $e \in E$, denoting whether or not e belongs to P_i, and we allow this variable to take fractional values. The resulting linear program can be written as follows, using $\delta^+(v)$ to denote the set of edges leaving v and $\delta^-(v)$ to denote the set of edges entering v.

$$\begin{array}{lll}
\min & r & \\
\text{s.t.} & \sum_{e \in \delta^+(v)} x_{i,e} - \sum_{e \in \delta^-(v)} x_{i,e} = \begin{cases} 1 & \text{if } v = s_i \\ -1 & \text{if } v = t_i \\ 0 & \text{if } v \neq s_i, t_i \end{cases} & \forall i = 1, \dots, k, \ v \in V \\
& \sum_{i=1}^{k} x_{i,e} \leq c_e \cdot r & \forall e \in E \\
& x_{i,e} \geq 0 & \forall i = 1, \dots, k, \ e \in E
\end{array} \tag{24}$$

When $(x_{i,e})$ is a $\{0, 1\}$-valued vector obtained from a collection of paths $P_1, \dots, P_k$ by setting $x_{i,e} = 1$ for all $e \in P_i$, the first constraint ensures that P_i is a path from s_i to t_i while the second one ensures that the congestion of each edge is bounded above by r.

Our approximation algorithm solves the linear program (24), does some postprocessing of the solution to obtain a probability distribution over paths for each terminal pair (s_i, t_i), and then outputs an independent random sample from each of these distributions. To describe the postprocessing step, it helps to observe that the first LP constraint says that for every $i \in \{1, \dots, k\}$, the values $x_{i,e}$ define a network flow of value 1 from s_i to t_i. Define a flow to be *acyclic* if there is no directed cycle C with a positive amount of flow on each edge of C. The first step of the postprocessing is to make the flow $(x_{i,e})$ acyclic, for each i. If there is an index $i \in \{1, \dots, k\}$ and a directed cycle C such that $x_{i,e} > 0$ for every edge $e \in C$, then we can let $\delta = \min\{x_{i,e} \mid e \in C\}$ and we can modify $x_{i,e}$ to $x_{i,e} - \delta$ for every $e \in C$. This modified solution still satisfies all of the LP constraints, and has strictly fewer variables $x_{i,e}$ taking nonzero values. After finitely many such modifications, we must arrive at a solution in which each of the flow $(x_{i,e})$, $1 \leq i \leq k$ is acyclic. Since this modified solution is also an optimal solution of the linear program, we may assume without loss of generality that in our original solution $(x_{i,e})$ the flow was acyclic for each i.

Next, for each $i \in \{1, \dots, k\}$ we take the acyclic flow $(x_{i,e})$ and represent it as a probability distribution over paths from s_i to t_i, i.e. a set of ordered pairs (P, π_P) such that P is a path from s_i to t_i, π_P is a positive number interpreted as the probability of sampling P, and the sum of the probabilities π_P over all paths P is equal to 1. The distribution can be constructed using the following algorithm.

Algorithm 8 Postprocessing algorithm to construct path distribution

```
1: **Given:** Source $s_i$, sink $t_i$, acyclic flow $x_{i,e}$ of value 1 from $s_i$ to $t_i$.
2: Initialize $\mathcal{D}_i = \emptyset$.
3: while there is a path $P$ from $s_i$ to $t_i$ such that $x_{i,e} > 0$ for all $e \in P$ do
4:     $\pi_P = \min\{x_{i,e} \mid e \in P\}$
5:     $\mathcal{D}_i = \mathcal{D}_i \cup \{(P, \pi_P)\}$.
6:     for all $e \in P$ do
7:         $x_{i,e} = x_{i,e} - \pi_P$
8:     end for
9: end while
10: return $\mathcal{D}_i$
```

Each iteration of the **while** loop strictly reduces the number of edges with $x_{i,e} > 0$, hence the algorithm must terminate after selecting at most m paths. When it terminates, the flow $(x_{i,e})$ has value zero (as otherwise there would be a path from s_i to t_i with positive flow on each edge) and it is acyclic because $(x_{i,e})$ was initially acyclic and we never put a nonzero amount of flow on an edge whose flow was initially zero. The only acyclic flow of value zero is the zero flow, so when the algorithm terminates we must have $x_{i,e} = 0$ for all e.

Each time we selected a path P, we decreased the value of the flow by exactly π_P. The value was initially 1 and finally 0, so the sum of π_P over all paths P is exactly 1 as required. For any given edge e, the value $x_{i,e}$ decreased by exactly π_P each time we selected a path P containing e, hence the combined probability of all paths containing e is exactly $x_{i,e}$.

Performing the postprocessing algorithm 8 for each i, we obtain probability distributions $\mathcal{D}_1, \ldots, \mathcal{D}_k$ over paths from s_i to t_i, with the property that the probability of a random sample from $\mathcal{D}_i$ traversing edge e is equal to $x_{i,e}$. Now we draw one independent random sample from each of these k distributions and output the resulting k-tuple of paths, $P_1, \ldots, P_k$. We claim that with probability at least 1/2, the parameter $\max_{e \in E} \{\ell_e / c_e\}$ is at most αr, where $\alpha = \frac{3 \log(2m)}{\log \log(2m)}$. This follows by a direct application of Corollary 4 of the Chernoff bound. For any given edge e, we can define independent random variables $X_1, \ldots, X_k$ by specifying that

$$X_i = \begin{cases} (c_e \cdot r)^{-1} & \text{if } e \in P_i \\ 0 & \text{otherwise.} \end{cases}$$

These are independent and the expectation of their sum is $\sum_{i=1}^{k} x_{i,e}/(c_e \cdot r)$, which is at most 1 because of the second LP constraint above. Applying Corollary 4 with $N = 2m$, we find that the probability of $X_1 + \cdots + X_k$ exceeding α is at most $1/(2m)$. Since $X_1 + \cdots X_k = \ell_e (c_e \cdot r)^{-1}$, this means that the probability of ℓ_e / c_e exceeding αr is at most $1/(2m)$. Summing the probabilities of these failure events for each of the m edges of the graph, we find that with probability at least 1/2, none of the failure events occur and $\max_{e \in E} \{\ell_e / c_e\}$ is bounded above by αr. Now, r is a lower bound on the parameter $\max_{e \in E} \{\ell_e / c_e\}$ for *any* k-tuple of paths with the specified source-sink pairs, since any such k-tuple defines a valid LP solution and r is the optimum value of the LP. Consequently, our randomized algorithm achieves approximation factor α with probability at least 1/2.