Cornell University, Spring 2019	**CS 4820: Algorithms**
Randomized approximation algorithms for MAX-CUT	**May 1, 2019**

1 Randomized Approximation Algorithms

Randomized techniques give rise to some of the simplest and most elegant approximation algorithms. These notes present one example, namely the max-cut problem.

2 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. Unlike the minimum cut problem, which can be solved in polynomial time, the max-cut problem is NP-hard. We will not present the proof in these notes.

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.

One thing that's a bit unsatisfying about the randomized approximation algorithm is that it isn't guaranteed to produce a cut whose weight is at least half the weight of the maximum cut. Instead, the analysis only shows that the *expected* weight of the cut produced by the algorithm is at least half the weight of the maximum cut. Any particular time that one runs the algorithm, it might get unlucky and produce a cut that's much worse than a 2-approximation.

A typical way to address this deficiency is to run the algorithm several times, yielding cuts $(A_1, B_1), \ldots, (A_k, B_k)$, and output whichever one of these cuts has the largest weight. In a later lecture on Karger's randomized min-cut algorithm, we'll see an example of how to analyze this technique for boosting the success probability of a randomized algorithm by repeated sampling. The analysis technique also furnishes a method for bounding the number of independent trials necessary to succeed with probability $1 - \delta$ for any specified parameter $\delta > 0$.

An even better way to reduce the uncertainty surrounding randomized algorithms is to *derandomize* them, i.e. to modify the randomized algorithm into a deterministic one that achieves the same performance guarantee (in this case, a 2-approximation) while increasing the running time by at most a polynomial factor.

2.1 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 $[k]$ denote the set $\{1, \dots, k\}$ and assign to each vertex v a distinct subset $S(v) \subseteq [k]$. Our choice of $k = \lceil \log_2(n) \rceil$ ensures that $[k]$ has enough subsets to assign a distinct one to each vertex; the mapping from vertices v to subsets $S(v)$ can be arbitrary as long as it is one-to-one. Now let R be a uniformly random subset of $[k]$, and partition the vertex set V into the subsets

$$A_R = \{v \mid S(v) \cap R \text{ has an even number of elements}\} \tag{1}$$
$$B_R = \{v \mid S(v) \cap R \text{ has an odd number of elements}\}. \tag{2}$$

Lemma 1. *Consider the random cut (A_R, B_R) defined in (1)-(2). For any edge $e = (u, v)$, the probability that $e \in E(A_R, B_R)$ equals $\frac{1}{2}$.*

Proof. Edge $e = (u, v)$ belongs to $E(A_R, B_R)$ if and only if $|S(u) \cap R|$ and $|S(v) \cap R|$ have opposite parity. Let

$$\begin{aligned} X &= S(u) \cap S(v) \\ Y &= S(u) \setminus S(v) \\ Z &= S(v) \setminus S(u) \end{aligned}$$

and observe that X, Y, Z are disjoint, $S(u) = X \cup Y$, and $S(v) = X \cup Z$. Then we have

$$\begin{aligned} &|S(u) \cap R| \text{ and } |S(v) \cap R| \text{ have opposite parity} \\ &\qquad\Leftrightarrow\quad |S(u) \cap R| + |S(v) \cap R| \text{ is odd} \\ &\qquad\Leftrightarrow\quad |X \cap R| + |Y \cap R| + |X \cap R| + |Z \cap R| \text{ is odd} \\ &\qquad\Leftrightarrow\quad |(Y \cup Z) \cap R| \text{ is odd} \end{aligned}$$

Let $W = Y \cup Z$. We are left with the task of showing that $|W \cap R|$ is odd with probability $\frac{1}{2}$. Since R is a uniformly random subset of $[k]$ and $W \subseteq [k]$, the set $W \cap R$ is a uniformly random subset of W.

The sets $S(u), S(v)$ are distinct, so there is at least one element $i \in [k]$ that belongs to one of the sets but not the other. The set W, by definition, consists of all the elements in $Y \cup Z$, i.e. all of the elements that belongs to one of $S(u), S(v)$ but not the other. Hence W is non-empty; let i denote an element of W. The subsets of W can be grouped into pairs $(T, T \cup \{i\})$ where T runs through all of the subsets that do not contain i. Each pair has one odd-cardinality subset and one even-cardinality subset, so the odd-cardinality subsets and the even-cardinality subsets of W are equinumerous. Since $W \cap R$ is a uniformly random subset of W, it follows that the probability that $|W \cap R|$ is odd equals $\frac{1}{2}$ as claimed. $\square$

At this point, we have derived an alternative randomized 2-approximation algorithm that samples $R \subseteq [k]$ uniformly at random and outputs the cut (A_R, B_R). The big advantage of this randomized algorithm, compared to the one analyzed earlier, is that it has so few potential outputs that we can simply enumerate them all in polynomial time. In fact, recalling that $k = \lceil \log_2(n) \rceil$ we find that the set $[k]$ has $2^k = O(n)$ distinct subsets R. For each such set, in $O(kn) = O(n \log n)$ time we can compute the partition (A_R, B_R) and in $O(m)$ time we can calculate the weight of the edge set $E(A_R, B_R)$. Repeating these operations for each of the $O(n)$ sets $R \subseteq [k]$, and taking the largest-weight cut among the collection $\{(A_R, B_R) \mid R \subseteq [k]\}$, we obtain a deterministic 2-approximation algorithm for max-cut with running time $O(n^2 \log n + mn)$.

2.2 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 1 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 2 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.

2.3 Epilogue: Improving the approximation factor for max-cut

Although the randomized 2-approximation algorithm and its deterministic greedy counterpart, Algorithm 2, are both extremely simple, for many years no one knew if any polynomial-time algorithm could achieve an approximation factor better than 2 in the worst case. Then in 1994, Michel Goemans and David Williamson discovered an algorithm with approximation factor roughly 1.13823, based on *semidefinite programming* (SDP). (For some reason, the approximation factor of the Goemans-Williamson algorithm is most often expressed as 0.878, which is the reciprocal of 1.13823.) Semidefinite programming is a generalization of linear programming, and subsequent to the Goemans-Williamson discovery it has found many other applications in algorithm design. We will not explain SDP in CS 4820, but it is covered in CS 6820 if you're interested to learn more about it.