CS-GY 6043 Lecture 10 - More Approximation Algorithms

Junior Francisco Garcia (jfg388@nyu.edu)

December 4, 2023

1 Introduction

In this lecture, we covered the following additional approximation algorithms and their corresponding analysis:

1. Greedy Set Cover
2. K-center
3. Maximum matching and min-size maximal matching via maximal matching.
4. Lightest Vertex Cover via LP Rounding

We also ended the lecture with a hand-waved introduction to randomized algorithms.

2 Greedy Set Cover

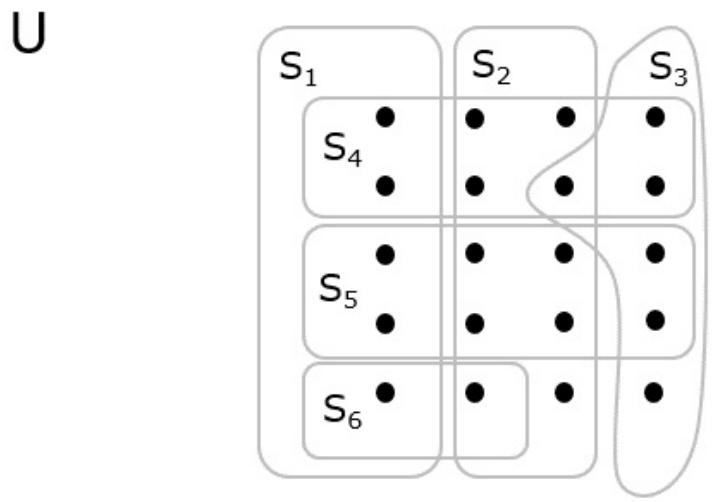

Figure 1: In this diagram, the dots represent the elements present in the universal set U that are divided into different sets, $S = \{S1, S2, S3, S4, S5, S6\}$. The minimum number of sets covering all the elements is $3 : \{S1, S2, S3\}$. Obtained directly from [5].

2.1 Problem Introduction

We must first begin with the definition of the **Set Cover Problem**, a problem we covered in a previous lecture.

- Let $S_1, S_2, \ldots, S_m$ be sets.
- Define $X = \bigcup_{i=1}^{m} S_i$.
- A collection $C = \{S_1, S_2, \ldots, S_m\}$ is a cover if $\bigcup_{S_i \in C} S_i = X$. Our goal is to find a cover of minimum size.

When finding a set cover in a collection of sets, a very natural and intuitive strategy when picking sets is just to do so in a greedy fashion.

```
Greedy Set Cover algorithm

Repeat until all elements of X are covered:
    Pick the set Si with the largest number of elements that are still
    uncovered
```

2.2 Greedy Set Cover Analysis

Let's consider the variables:

- $n = |X|$: Total number of elements in the set X.
- $k = \text{OPT}$: Number of sets in the minimum set cover.
- n_i: Number of elements left uncovered after picking set i.
- $n_0 = n$: Initially, all n elements are uncovered.

When we are at step $i+1$, we have n_i elements left, which are covered by the OPT. Therefore, there is at least one set that covers $\frac{n_i}{k}$ elements (due to the pigeonhole principle, at least one set in OPT has to cover $\frac{n_i}{k}$).

The update rule for n_i is given by:

$$n_{i+1} \leq n_i - \frac{n_i}{k} = n_i \left(1 - \frac{1}{k}\right)$$

By repeatedly applying this update rule, we get:

$$n_i \leq n_0 \left(1 - \frac{1}{k}\right)^i$$

Using the inequality $1 - x \leq e^{-x}$ for all x (with equality if and only if $x = 0$), we can write:

$$1 - \frac{1}{k} < e^{-\frac{1}{k}}$$

Thus, we have:

$$n_i \leq n_0 \left(1 - \frac{1}{k}\right)^i < n_0 e^{-\frac{i}{k}} = n e^{-\frac{i}{k}}$$

When $i = k \ln n$, n_i becomes strictly less than $n e^{-\ln n} = 1$, which means no elements remain to be covered. Therefore, the greedy algorithm will use at most $k \ln n$ sets, making it a factor-$\ln n$ approximation algorithm.

3 K-Center

3.1 K-Center Problem Introduction

In this problem, we are given a set of points in a plane $p = \{p_1, p_2, \ldots, p_n\}$ and an integer k, $1 \leq k \leq n$. Our goal is to find a collection of centers $c_1, c_2, \ldots, c_k$ such that together all circles cover all points and the radius of the largest circle is as small as possible. This can be described as follows:

$$\text{cost}(C) := \max_i \max_j d(x_i, c_j)$$

Our goal here is to minimize cost C.

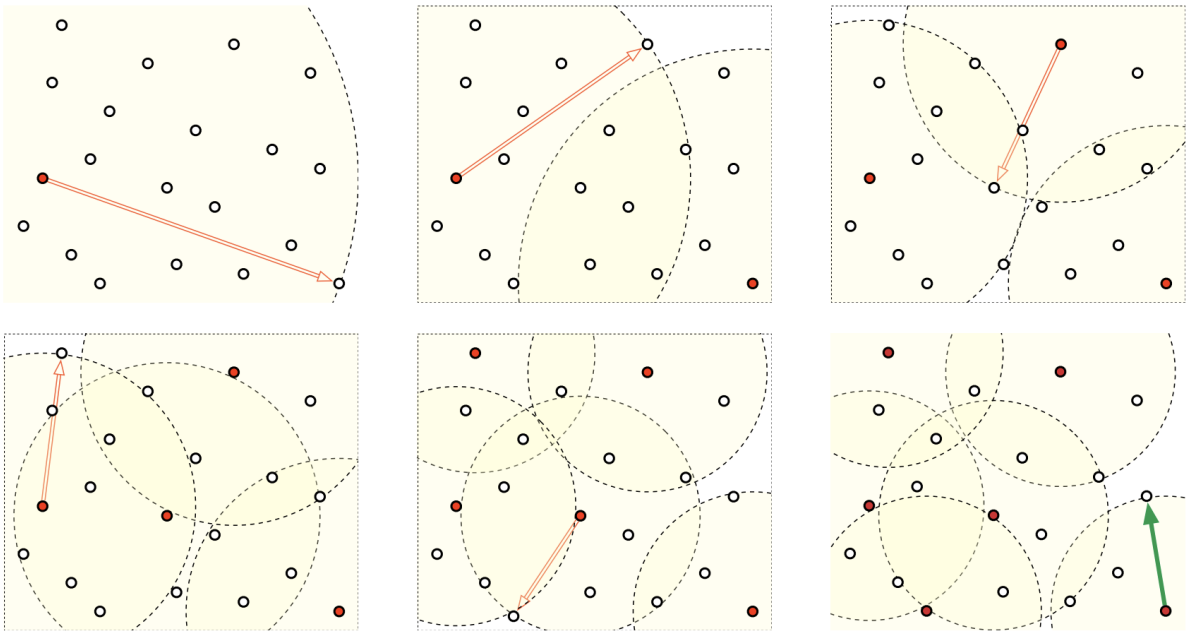

Figure 2: The first six iterations of Gonzalez's k-center clustering algorithm, obtained directly from Jeff Erikson's notes on approximation algorithms [2].

This is an NP-hard problem, but we can arrive at a 2-factor approximation using a greedy strategy as well. This strategy consists of choosing the centers one at a time, starting with an arbitrary point as the first center. In each subsequent iteration, we choose the point that is farthest from any previously selected center point to be the next center. Below is a detailed explanation of the pseudo-code adopted directly from Jeff's Notes [2]:

```
GonzalezKCenter(P, k):
for i ← 1 to n
    d_i ← ∞
c1 ← p1
for j ← 1 to k
    r_j ← 0
    for i ← 1 to n
        d_i ← mind_i, |p_i - c_j|
        if r_j < d_i
            r_j ← d_i
            c_j+1 ← p_i
return c1, c2, ..., ck
```

3.2 K-Center Analysis

Let OPT denote the optimal k-center clustering radius for plane p.

Consider c_i and r_i, which denote the i-th center point and the i-th clustering radius computed in step i of the GonzalezKCenter algorithm.

Each center c_j has a distance of at least r_{j-1} from any previously chosen center point c_i where $i < j$. This is because, when selecting c_j, the algorithm looks at all points that are not yet centers and selects the one that is farthest away from the nearest of the already chosen centers $c_1, c_2, \ldots, c_{j-1}$. The radius r_{j-1} is the maximum distance any point has to travel to reach its nearest center after $j-1$ centers have been selected. If a potential new center were any closer than r_{j-1} to the existing centers, it would not be the farthest point from the existing centers.

The more centers we add as the algorithm progresses, the subsequent radii decrease. Thus, for $i < j$, we have $r_i \geq r_j$. This also means that the radius r_k, corresponding to the last chosen center c_k, is less than or equal to all distances among any two previously selected centers: $|c_i c_j| \geq r_k$ for all i, j where $i \neq j$.

At least one cluster in the optimal clustering contains at least two of the points $c_1, c_2, \ldots, c_{k+1}$. This follows from the pigeonhole principle. Thus, by the triangle inequality, we must have $|c_i c_j| \leq 2 \cdot \text{OPT}$ for some indices i and j. This is because points c_i and c_j are in the same disk in the optimal solution.

- c^*: This is a center in the optimal solution that is closest to both c_i and c_j.
- r^*: This is the radius of the cluster in the optimal solution that contains c_i and c_j. Since they are in the same disk, r^* is the maximum distance any point in this cluster has to travel to its center c^*, and it cannot exceed the optimal clustering radius, OPT.

Using the triangle inequality:

$$r_k \leq |c_i c_j| \leq |c_i c^*| + |c_j c^*| < 2r^*$$

Hence, $r_k \leq 2r^*$ and since $r^* \leq \text{OPT}$, it follows that $r_k \leq 2 \cdot \text{OPT}$ [2].

4 Maximum Matching and Min-Size Maximal Matching via Maximal Matching

Consider an undirected graph $G = (V, E)$.

- A matching in a graph is a set of edges such that no two edges share a common vertex.
- A maximal matching cannot be extended by adding edges. Finding one is an easy polynomial-time solution.
- A maximum matching has the maximum number of edges possible in a matching. This also has a polynomial-time solution, though it may be more complex than finding a maximal matching.
- Finding a minimal cardinality maximal matching is an NP-hard problem.

If we pick an arbitrary maximal matching, we have a 2-factor approximation to the optimal solution to a minimal cardinality maximal matching.

Proof:

Pick two arbitrary maximal matchings M_1 and M_2 in G.

Claim: $\frac{1}{2}|M_1| \leq |M_2| < 2|M_1|$

Ignore all the places where they have the same edge.

We can visualize this in the following diagram, which depicts an instance of the worst-case non-overlapping case of two maximal matchings.

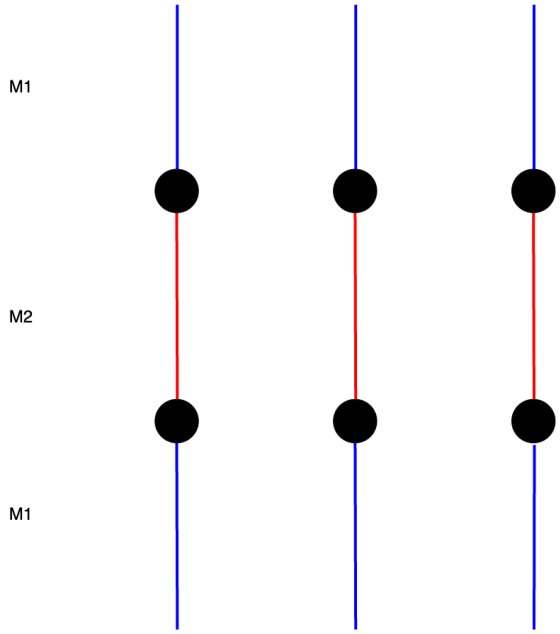

Figure 3: Two Maximal matchings in G.

In this diagram, you can clearly see that the size of M_1 is at most 2 times larger than M_2. It is important to note that this is not a complete formal proof.

5 Lightest Vertex Cover via LP Rounding

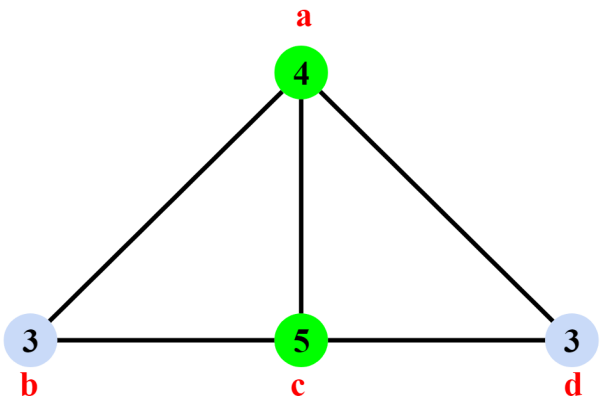

Figure 4: In this graph, the vertex cover $\{A, C\}$ with a weight of 9 is the lightest vertex cover. Adapted from [4].

Consider a generalization of the minimum vertex cover problem in which every vertex v of the input graph has a non-negative weight $w(v)$, and the goal is to compute a vertex cover C with minimum total weight $w(C) = \sum_{v \in C} w(v)$.

This problem can be cast as an instance of integer linear programming (ILP), where each feasible solution to the ILP corresponds to a vertex cover. The ILP formulation is as follows:

$$\text{minimize} \sum_v w(v) \cdot x(v)$$

subject to

$$x(u) + x(v) \geq 1 \text{ for every edge } uv,$$

$$x(v) \in \{0, 1\} \text{ for every vertex } v.$$

Let OPT denote the weight of the lightest vertex cover. We then consider the linear program obtained from this ILP by removing the integrality constraint; this is known as the linear relaxation of the ILP. Notice the relaxation occurring in the boldened text:

$$\text{minimize} \sum_v w(v) \cdot x(v)$$

subject to

$$x(u) + x(v) \geq 1 \text{ for every edge } uv,$$

$$\mathbf{0 \leq x(v) \leq 1 \text{ for every vertex } } v.$$

Let x^* denote the optimal solution to this linear relaxation, and let $\text{OPT}_{\text{LP}} = \sum_v w(v) \cdot x^*(v)$. The optimal fractional solution x^* is a lower bound on the optimal integral solution: $\text{OPT} \geq \text{OPT}_{\text{LP}}$.

To derive a good approximation to the lightest vertex cover, we round the optimal fractional solution. For each vertex v, let

$$x_0(v) = \begin{cases} 1 & \text{if } x^*(v) \geq 1/2, \\ 0 & \text{otherwise.} \end{cases}$$

This rounding process has the following implications:

- For every edge uv, the constraint $x^*(u) + x^*(v) \geq 1$ implies that $\max\{x^*(u), x^*(v)\} \geq 1/2$, and therefore either $x_0(u) = 1$ or $x_0(v) = 1$. Thus x_0 is the indicator function of a vertex cover, ensuring that every edge is covered by at least one vertex in the cover.
- $x_0(v) \leq 2x^*(v)$ for every vertex v. This implies that [1]

$$\sum_v w(v) \cdot x_0(v) \leq 2 \sum_v w(v) \cdot x^*(v) = 2 \cdot \text{OPT}_{\text{LP}} \leq 2 \cdot \text{OPT}.$$

6 Introduction to Randomized Algorithms

Randomized algorithms use a degree of randomness as part of their logic [1]. They often include randomness as part of their input to achieve good average case behavior, as illustrated in Figure 5. The following are some key aspects:

- **Correctness:** A randomized algorithm does not necessarily guarantee absolute correctness for every execution. There are two main types:
  - *Monte Carlo algorithms*, which have a bounded error probability.
  - *Las Vegas algorithms*, which always produce the correct result or report failure.

[1] Please refer to [2] to obtain the original source explanation of this analysis.

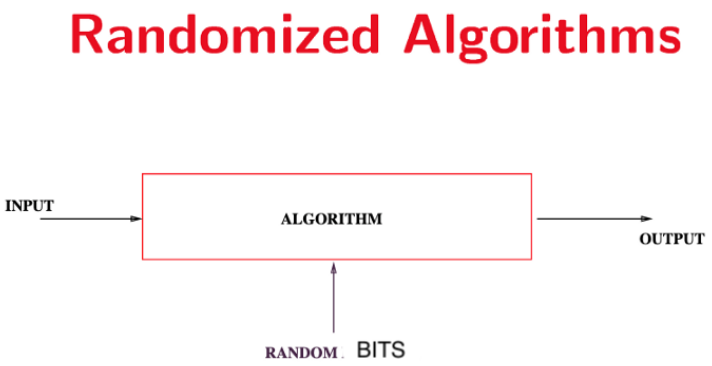

Figure 5: A diagram explaining randomized algorithms, obtained from [3]

- **Termination:** Whether a randomized algorithm always stops depends on its type and the desired goal of the algorithm designer. Some might always terminate, but the time taken to terminate might vary, while others might have a small probability of running indefinitely to ensure better performance.
- **Performance:** The performance of randomized algorithms can be analyzed in terms of expected runtime, or "fast with high probability" for example. While not always the fastest in the worst case, they often have good average-case performance or simplicity that outweighs worst-case scenarios.

References

[1] Wikipedia contributors. 2023. Randomized algorithm. `https://en.wikipedia.org/wiki/Randomized_algorithm`. Accessed: 2023-11-28.

[2] Jeff Erikson. 2018. Notes on Approximation Algorithms. `https://jeffe.cs.illinois.edu/teaching/algorithms/notes/J-approx.pdf` Accessed: 2023.

[3] Chad Malla. 2019. Randomized Algorithms: how to tackle the Contention Resolution Problem. `https://www.freecodecamp.org/news/randomized-algorithms-part-1-d89986bb685b/`. *freeCodeCamp* (Apr 2019).

[4] SleekPanther. 2023. Approximation Algorithm for the NP-Complete problem of finding a vertex cover of minimum weight in a graph with weighted vertices. `https://github.com/SleekPanther/minimum-weighted-vertex-cover-approximation-algorithm`. Accessed: 2023-11-28.

[5] TutorialsPoint. Accessed 2023. Design and Analysis of Algorithms: Set Cover Problem. `https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_set_cover_problem.htm`