This document is a short note summarizing the various approximation algorithms for hard optimization problems seen in the class. We saw algorithms for hard problems that did not give the optimal solution, but always gave a solution that was provably close to the optimum. Formally, the approximation ratio achieved by an algorithm for a minimization problem is defined as

\[ \alpha = \max_{I} \frac{\text{apx}(I)}{\text{opt}(I)} \]

where we take the ratio over all instances \( I \) of the problem, \( \text{apx}(I) \) is the output of the algorithm and \( \text{opt}(I) \) is the optimal value of that instance.

It is important to note that although most of these problems are equivalent in terms of their complexity/hardness in calculating the optimum, they are vastly different when it comes to how well we can approximate them. The approximation ratios can range from being arbitrarily small (PTAS for Knapsack) to being polynomially large (\( O(\sqrt{n}) \) for 3-coloring).

1. Minimum Vertex Cover

Given a graph \( G = (V, E) \), a subset of the vertices \( S \subseteq V \) is called a vertex cover if every edge has at least one endpoint in \( S \). The Min-Vertex-Cover problem asks for us to find a vertex cover of minimum size. This problem is hard to solve exactly. However, there is a simple approximation algorithm for it:

1. Construct a maximal matching \( M \) of \( G \).
2. Put both the endpoints of each edge in \( M \) in the vertex cover.

The maximal matching \( M \) can be constructed greedily in \( O(|E|) \) time by repeatedly taking an edge and discarding all the incident edges to the two endpoints of that edge. It is easy to see that the algorithm always outputs a vertex cover. If the output is not a cover, then there is an edge both of whose endpoints are not in the cover. But then this edge could have been added to \( M \), thus contradicting the maximality of \( M \).

Let \( |M| \) be the number of edges in the matching \( M \). Note that any vertex cover has to have size at least \( |M| \), since otherwise there will exist at least one edge in \( M \) that has neither endpoint in the cover, which will contradict the definition of the cover. So if \( \text{opt} \) is the size of the minimum vertex cover, then

\[ \text{opt} \geq |M|. \]

If \( \text{apx} \) is the size of the vertex cover output by the algorithm, then we have

\[ \text{apx} = 2|M| \leq 2\text{opt} \]

1. A similar definition can be used for maximization problems.
2. They are all \( \mathcal{NP} \)-complete. What this means will be seen later in the course.
thus giving us a 2-approximation algorithm (using Definition (1)).

2. Bin Packing

Here we consider \( n \) items with corresponding sizes \( s_1, \ldots, s_n \), where each \( s_i \in (0, 1] \). We have to pack these items into a number of unit sized bins. The objective is to use the least amount of bins possible. The problem is \( \mathcal{NP} \)-hard, but can be approximated by the following procedure called First-fit:

For each item, try to pack it in one of the open bins if there is space, otherwise open a new bin and put the item in it.

First-fit will indeed produce a valid packing. A lower bound for the least number of bins (\( \text{opt} \)) is \( \lceil \sum_i s_i \rceil \). At the end of the procedure, every bin except for the last will be no less than half full. If not, then the contents of the second ‘less than half full’ bin could easily have gone into the first ‘less than half full’ bin. Thus, if \( \text{apx} \) is the number of bins used by the First-fit procedure, then

\[
\frac{\text{apx} - 1}{2} \leq \sum_i s_i \leq \text{opt},
\]

giving us a 2-approximation algorithm (using Definition (1)).

3. Set Cover

Here we are given a base set \([n] = \{1, \ldots, n\}\) and a bunch of subsets \( S_1, \ldots, S_m \). The goal is to pick the minimum number of subsets out of these such that their union would be a cover, that is, their union would be equal to the base set.

The approximation algorithm proceeds by greedily picking subsets one-by-one. At each step, we pick the subset that covers the maximum number of uncovered elements. We stop when all the elements are covered.

To analyze this, let \( C^* \) be the collection of subsets in the optimal cover. As usual, \( \text{opt} = |C^*| \) is the minimum number. Let \( E_i \) be the set of uncovered elements after picking \( i \) subsets greedily. The following observation is key:

At step \( i \), there is at least one subset in \( C^* \) that covers at least \( |E_i|/\text{opt} \) elements. If not, then the subsets in \( C^* \) cannot possibly cover all elements.

This means that at least these many elements are covered at each step. Thus,

\[
|E_{i+1}| \leq |E_i| - \frac{|E_i|}{\text{opt}} \leq |E_0|(1 - \frac{1}{\text{opt}})^i = n(1 - \frac{1}{\text{opt}})^i.
\]

Let’s say that the number of steps taken by the algorithm is \( \text{apx} + 1 \). In the last but one step, we still have at least one uncovered element. Thus,

\[
|E_{\text{apx}+1}| \geq 1
\]

\[
n(1 - \frac{1}{\text{opt}})^{\text{apx}} \geq 1
\]

\[
n \cdot e^{-\frac{\text{apx}}{\text{opt}}} \geq 1
\]

(using \( 1 + x \leq e^x \)).

\[
\text{apx} \leq \text{opt} \cdot \ln n
\]

giving an \( O(\ln n) \)-approximation.
4. Knapsack

We have a knapsack of capacity $W$ and $n$ items with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$. The goal is to pick items to fill the knapsack (never going beyond the total weight $W$) so as to maximize the total value of the picked items. There are various exact dynamic programming solutions to this problem, but they do not run in polynomial time in the size of the input.

Here is one that runs in $O(n^2 V)$ time, where $V$ is the maximum of all the values of the items. Let $A(i, v)$ represent the minimum total weight of any subset selected only from the items $\{1, \ldots, i\}$ such that the total value of the items in the subset is exactly $v$. If there is no such subset, then $A(i, v) = \infty$. The entries $A(1, v)$, where $v$ ranges from $\{0, \ldots, nV\}$ can be calculated easily. Subsequent entries obey the recurrence

$$A(i + 1, v) = \begin{cases} 
\min \{A(i, v), A(i, v - v_i + 1) + w_{i+1}\} & \text{if } v_{i+1} \leq v, \\
A(i, v) & \text{otherwise}.
\end{cases}$$

The answer to the problem would be the maximum $v$ such that $A(n, v) \leq W$.

We will now see how to use the above procedure to construct a Polynomial Time Approximation Scheme (PTAS). Given a parameter $\varepsilon$, the algorithm will take time $O(n^3 / \varepsilon)$ and have an approximation factor of $(1 - \varepsilon)$. Let $K = \frac{\varepsilon V}{n}$. We will run the above procedure with new values of the items defined as

$$v'_i = \left\lfloor \frac{v_i}{K} \right\rfloor$$

Note that now there are only $\lfloor n/\varepsilon \rfloor$ different values, hence the efficient running time. Let $C^*$ be the optimal set of items and let $C'$ be the set of items output by the algorithm. The new item values $v'_i$ can be smaller than $v_i$’s, but by not more than $K$. Hence,

$$\text{value}(C^*) - K \cdot \text{value}'(C^*) \leq nK.$$

Now, the approximate solution should be at least as good as $\text{value}'(C^*)$. Thus,

$$\text{value}(C') \geq K \cdot \text{value}'(C') \geq \text{value}(C^*) - nK = \text{value}(C^*) - \varepsilon V \geq (1 - \varepsilon) \text{value}(C^*).$$

5. Metric TSP

Given a complete graph with costs $c_e$ associated with every edge $e$, the objective here is to find a tour of minimum total cost. A tour is a path in the graph that visits all cities and returns to the starting point. We will assume the metric condition on the costs in the graph, that is for any three vertices $u, v, w$, the following triangle inequality is always satisfied:

$$c_{\{u,v\}} \leq c_{\{u,w\}} + c_{\{w,v\}}$$

The algorithm consists of building an MST (Minimum Spanning Tree) on the graph. If we double the edges in the MST, then we can walk along the tree, starting at any vertex, covering all the vertices and using up the two copies of each edge. But that is not a tour. To convert this walk into a tour, we simply skip the vertices that we have already encountered in the walk. At the end we return to the starting vertex in the walk. This converts the walk into a tour, and skipping cities does not increase the total cost due to the triangle inequality. The cost is

$$\text{apx} \leq 2 \cdot \text{cost(MST)} \leq 2 \cdot \text{opt}.$$
where the last inequality comes from the fact that deleting any edge in a minimum cost tour results in a spanning tree.

6. Randomized approximations

In this section we sketch two examples of how we can use randomization in designing approximation algorithms.

The first problem we consider is MAX-3SAT, where the input is a collection of $m$ clauses $C_1, \ldots, C_m$ in $n$ variables $\{x_1, \ldots, x_n\}$. Each clause is an OR (‘∨’) of three variables (or their negations). A clause is satisfied by an assignment of ‘True’ or ‘False’ values to the variables if it evaluates to True. The objective is to maximize the number of clauses simultaneously satisfied by any given assignment.

The approximation algorithm for this simply flips a fair coin for each variable to decide its value. Under this random assignment, the probability that any particular clause is not satisfied is $1/8$. Let $I_C$ be the indicator variable for clause $C$ being satisfied. Then $\mathbb{E} I_C = 1 - 1/8 = 7/8$. By linearity of expectation, the expected number of clauses satisfied would thus be

$$\sum_{i=1}^{m} \mathbb{E} I_{C_i} = \frac{7}{8}m.$$ 

This gives a $7/8$ approximation, since the maximum value can be $m$.

Another interesting problem is MAX-3Lin, where instead of clauses, our input consists of a collection of $m$ boolean equations in three variables. A boolean equation in three variables looks like

$$x_1 + x_{100} - x_7 \equiv 0 \pmod{2}.$$ 

Note that if we fix two variable values in some equation, then the third variable completely determines whether the equation is satisfied or not. Then if we use the same randomized coin-flipping strategy as MAX-3SAT, the probability that any particular equation is satisfied is $1/2$. By the same arguments as before, we end up satisfying $m/2$ equations in expectation.

7. Approximate Coloring

Given a graph $G = (V, E)$, a coloring of $G$ assigns colors to the vertices in $V$ such that there is no monochromatic edge in $E$, that is, no edge has both the endpoints of the same color.

Now we are given a graph that is guaranteed to be colorable with 3 colors. We are asked to color the graph with as few colors as possible. The algorithm is based on the following two observations:

- A 2-colorable graph can be efficiently colored with two colors. We start by coloring an arbitrary vertex red. This forces its neighbors to get the color blue and the neighbors of those neighbors red and so on. Thus by propagation we always end up with a valid coloring.
- A graph with maximum degree $\Delta$ can be colored with at most $\Delta + 1$ colors. At each step, simply pick an uncolored vertex, look at the colors of its neighbors (we can have at most $\Delta$ different colors), and assign a color that is different from all of them.

$3$The degree of a vertex is the number of neighbors of that vertex.
The algorithm proceeds as follows:

(1) Pick a vertex with degree $\geq \sqrt{n}$. The subgraph formed by the neighbors of this vertex will be 2-colorable. So we can color the neighborhood with two colors and give another color to the picked vertex. Now we forget all the vertices that we colored and never reuse the three colors. Repeat the process till there is no vertex with degree $\geq \sqrt{n}$.

(2) The remaining graph has maximum degree $\sqrt{n} - 1$. So we can color it with $\sqrt{n}$ colors.

The number of vertices with degree $\geq \sqrt{n}$ is at most $n/\sqrt{n} = \sqrt{n}$. So the first step uses at most $3\sqrt{n}$ colors. The second step uses $\sqrt{n}$ colors. Thus in total we have used at most $4\sqrt{n}$ colors to color the graph.