Approximation Algorithms

Approximation Algorithms

Although it is hopeless to design a polynomial-time algorithm for an NP-hard optimization problem, it is possible to design a polynomial-time algorithm that finds the *sub-optimal* solutions. For example, a trivial algorithm for computing a vertex cover is to output all vertices in the graph. How do we compare these sub-optimal algorithms? Time complexity should be a concern but not the only or primary concern now. How close their solutions are to the optimal solution is more or equally important.

A polynomial-time approximation algorithm with **approximation ratio** ρ will guarantee to find a solution with value $\geq \frac{OPT}{\rho}$ for maximization problem, or with value $\leq \rho \cdot OPT$ for minimization problem.

By this definition, an approximation ratio is always ≥ 1. But frequently, you may also see people say that the ratio is 0.5 for a maximization problem. That is also acceptable. What they really mean is that the ratio is $\frac{1}{0.5} = 2$ according to our definition.

Let's check a couple of simple examples:

Closest String

Closest String: Given n length-L binary strings $s_1, \ldots, s_n$, compute a new string t, such that $d = \max_i d_H(t, s_i)$ is minimized. Here d_H is the Hamming distance.

This is a minimization problem. It has been proved that it is NP-hard. The following approximation algorithm has ratio 2:

Algorithm: return s_1.

Proof: Because of Triangular Inequality, $d(s_1, s_i) \leq d(t_{opt}, s_1) + d(t_{opt}, s_i) \leq 2d$. QED.

There have been many approximation algorithms developed for NP-hard problems. And common methods in developing approximation algorithms have been proposed too. Our short introduction here won't cover very much of this field; but at least it can let you be aware of the existence of this field.

Max-Cut

Max-Cut: Given $G = \langle V, E \rangle$, a cut is a partition $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$. The size of the cut is the number of edges whose two vertices are in V_1 and V_2, respectively. Find the maximum-sized cut.

Recall that in the approximation algorithm chapter, we introduced a randomized algorithm to find a cut of expected size $\frac{|E|}{2}$. Since the optimal cut size cannot exceed $|E|$, that algorithm is a

randomized approximation algorithm with ratio 2. Furthermore, that algorithm can be derandomized, so the derandomized algorithm is a deterministic approximation algorithm with ratio 2.

The currently best known approximation ratio for approximation algorithm for Max-Cut is $\alpha = \frac{\pi}{2} \cdot \max_{0 \le \theta \le \pi} \frac{1-\cos\theta}{\theta} \approx 1.139$.

Inapproximability

There is a limit for the approximation ratio. If $P \neq NP$, then the approximation ratio cannot be 1. Sometimes one can prove stronger results.

Theorem: if $P \neq NP$, then max-cut does not have a polynomial-time approximation algorithm with ratio better than $\frac{17}{16} = 1.0625$.

In rare case, the hardness result can match the algorithm result. For example, if the "unique games conjecture" is true, then $\alpha = \frac{\pi}{2} \cdot \max_{0 \le \theta \le \pi} \frac{1-\cos\theta}{\theta} \approx 1.139$ is the best possible approximation ratio for maximum cut.

Vertex Cover

Vertex Cover: Given a graph, find the minimum sized set of vertices that every edge has an endpoint in it (the edge is therefore "*covered*").

An application of vertex cover is to monitor all the links on a network, where you want to use minimum number of monitors (vertices) to monitor all links (the edges). Another example one can imagine is to construct stations for highway patrols,
where you want to use the minimum number of stations to cover all highways.

Max Independent Set: Let $G = \langle V, E \rangle$ be a graph. An independent set V' is a subset of V such that no edge connects two vertices in V'. Find the maximum sized independent set.

It is known that max independent set is NP-hard. We take this fact as a given, and prove the following theorem.

Theorem: Vertex Cover is NP-complete.

Proof: It is in NP because once a subset of vertices is given, it takes polynomial time to count the size and check if it covers all edges. To prove it is NP-hard, we reduce Independent set to it.

Given an instance of independent set, $I = (G, k)$. Let n be the number of vertices of G. Construct an instance of vertex cover as $I' = (G, n-k)$. It is polynomial-time. So the remaining is to prove that I and I' have the same answers. We only need to show that G has a vertex cover of size $n-k$ if and only if it has an independent set of size k.

Claim: Let $G = \langle V, E \rangle$ be a graph. V' is a vertex cover if and only if $V \setminus V'$ is an independent set.

If V' is a vertex cover, then every edge e has a vertex in V'. Therefore, there is no edge e that has both vertices in $V \setminus V'$. The other direction is also true.

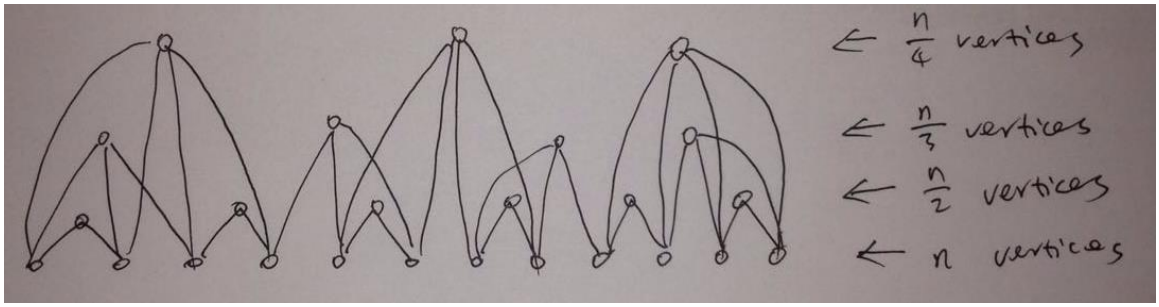
QED.

A very natural idea to design an approximation algorithm is by using a greedy strategy.

Algorithm VC-Greedy

1. $C \leftarrow \emptyset$
2. While E is not empty
    a. Pick an uncovered vertex u with maximum degree.
    b. $C \leftarrow C \cup \{u\}$
    c. Remove all edges incident to u from E.

Unfortunately, VC-Greedy does not perform very well. The following figure shows a design of a bad example. The design starts with n vertices in the first level. Then n levels of new vertices are added to the graph. Each vertex in level i connects to i vertices in level 1. This way, the greedy algorithm would use roughly $\Theta(n \log n)$ vertices in the cover. But the optimal cover has a size of n.



Algorithm VC-Matching.

1. $C \leftarrow \emptyset$
2. While E is not empty
    a. Arbitrarily pick $e = (u, v) \in E$.
    b. $C \leftarrow C \cup \{u, v\}$
    c. Remove all edges incident to u, v from E.

Theorem: Algorithm VC has approximation ratio 2.

Proof: Clearly VC is a vertex cover. Next let's prove the ratio. Suppose we pick k edges during the execution of algorithm VC. These k edges form a matching (they don't share vertices). So,

the optimal vertex cover should contain at least one vertex from each of these k edges. Thus $OPT \geq k$. On the other hand, algorithm VC's output has size at most $2k$.

QED.

Remark: An approximation algorithm is just a heuristic algorithm that happens to have a provable worst-case performance guarantee.

Remark: Since we are using worst-case analysis, it is always possible that on some instances, an algorithm with a larger ratio may actually perform better than an algorithm with a lower ratio. Additionally, it is also possible that the ratio we can prove for an algorithm is not tight. The algorithm may have a smaller ratio, but we can only prove the larger ratio because of our inability to find a mathematical proof.

Remark: Once algorithm VC outputs a set U of size m, we can use other heuristics to improve the cover. (Exercise: Think about many heuristic ways to reduce U.) There is no guarantee that you can improve. And even if you successfully found another set U' of size $m' < m$, there is no guarantee that m' is the optimal. But what conclusion can you draw between the relation of m' and the optimal? (Exercise: think about it.)

Exercise: Prove that if there is a polynomial-time algorithm that can guarantee to find a vertex cover of size $\leq OPT + c$ for a constant c, then P=NP.

Travelling-salesman Problem

Travelling-salesman problem (TSP) with arbitrary weight: Given a weighted complete graph $G = \langle V, E \rangle$, find a Hamilton cycle that has the minimum total edge weight.

Theorem: If $P \neq NP$, then for any function $f(n)$ that can be computed in polynomial time, TSP cannot be approximated with ratio $f(n)$.

Proof: Let $G = \langle V, E \rangle$ be an instance of the Hamiltonian cycle problem, which is known to be NP-hard. Construct a complete graph G' with the same vertex set. The edge weight is defined as

$$w(e) = \begin{cases} 1, \text{if } e \in E; \\ n \times f(n), \text{if } e \notin E. \end{cases}$$

If G has a Hamiltonian cycle, then the TSP on G' has cost n. If G has no Hamiltonian cycle, then the TSP will have to use at least one edge with weight $n \times f(n)$. Thus, having a polynomial-time algorithm to approximate TSP with ratio $f(n)$ will solve the Hamilton cycle problem in polynomial time with this reduction.

QED.

So we focus on TSP of which the edge weight satisfies some properties. For example, Triangular Inequality is a natural one – because if we allow a vertex is visited twice, then the path distance on the graph actually satisfies the Triangular Inequality.

Travelling-salesman problem (TSP): Given a weighted complete graph $G = \langle V, E \rangle$. The edge weight satisfies Triangular Inequality. Find a Hamilton cycle that has the minimum total edge weight.

Algorithm TSP:

1. Construct a minimum spanning tree T.
2. Let $v_{i_1} \to v_{i_2} \to \cdots \to v_{i_n}$ be the vertices sorted according to their first visit during a depth-first traversal on T.
3. Output $v_{i_1} \to v_{i_2} \to \cdots \to v_{i_n} \to v_{i_1}$ as the solution.

Theorem: Algorithm TSP is a ratio 2 approximation algorithm.

Proof: A Hamiltonian cycle has cost higher than a Hamiltonian path; and a Hamiltonian path is a spanning tree. Therefore, the cost of the optimal Hamiltonian cycle is at least the weight of a MST.

On the other hand, consider a solution constructed by Algorithm TSP: $v_{i_1} \to v_{i_2} \to \cdots \to v_{i_n} \to v_{i_1}$. If we walk from $v_{i_j} \to v_{i_{j+1}}$ only through the path on the MST, then each edge on the MST is used at most twice. Thus, the total cost would be $2 \times MST$. Furthermore, because of the triangular inequality, walking on MST has an equal or larger cost than using the edge $v_{i_j} \to v_{i_{j+1}}$ on the graph directly. Thus, the algorithm's solution has a cost at most $2 \times MST$.

QED.

In special distance metric, it is possible to achieve much better ratio. We give the following theorem without proof.

Theorem (S. Arora): If under geometric distance in space $\Re^d$, then TSP has a randomized polynomial-time approximation scheme (PTAS) that achieves approximation ratio $1 + \epsilon$ in time

$$O\left(n(\log n)^{O\left(\sqrt{d}\cdot\frac{1}{\epsilon}\right)^{d-1}}\right).$$

Pay attention to the tradeoff between time complexity and the approximation ratio. But overall, this is good time complexity in theory: nearly linear time.

This is called a polynomial-time approximation scheme (PTAS). Basically, a PTAS for a problem is an algorithm that takes an additional input $\epsilon > 0$, and runs in time $O\left(n^{f(\epsilon)}\right)$ for some function f.

Set Cover

Set Cover: Given a collection of sets $S_1, \dots, S_m$, where $S_i \subseteq [1, n]$. A cover is a sub-collection of sets $\{S_{i_1}, \dots, S_{i_k}\}$ such that $S_{i_1} \cup \dots \cup S_{i_k} = [1, n]$. Minimize k, the number of sets in the cover.

Application: You host a party with n people. You can order m different flavors of pizza. Each flavor can feed a subset of the people. How to find the minimum number of flavors to cover all people?

Vertex cover is a special case of set cover, where all the edges are the elements to cover. Edges that are incident with a vertex correspond to a set. Since vertex cover is NP-hard, so is set cover.

Algorithm Greedy-Cover

1. $C \leftarrow \emptyset$
2. While there are uncovered elements
    a. Find set S that covers the maximum number of uncovered elements
    b. $C \leftarrow C \cup \{S\}$

Note that this algorithm is the same as VC-greedy when the set cover instance is reduced from a vertex cover problem.

Theorem. The Greedy-Cover algorithm is a polynomial-time approximation algorithm for set cover with ratio $1 + \log d$, where $d \leq n$ is the maximum size of a set in the input.

Proof: Without loss of generality, suppose $S_1, \dots, S_p$ are the sets added to the collection by the algorithm in order. Suppose the optimal solution is $OPT = \{S_{i_1}, \dots, S_{i_k}\}$. We'd like to prove $p \leq (1 + \log d) \cdot k$.

Each set S_i chosen in 2a has cost 1. Distribute this cost to the newly covered elements by S_i. That is, if $S_i \setminus \left(\bigcup_{j=1}^{i-1} S_j\right)$ contains r elements, then each of these r elements is charged $\frac{1}{r}$. Denote the charge received by element j by $c(j)$. Then $\sum_{j=1..n} c(j) = p$.

Consider any set S in the optimal solution with size d'. By reordering the elements, assume that $j_{d'}, j_{d'-1}, \dots, j_1$ are the elements of S in their order of being first covered by the Greedy-Cover algorithm. When j_t is to be covered for the first time, at least t elements in S are not covered yet. Thus, the set selected by the algorithm to cover j_t would cover at least t elements. Therefore, $c(j_t) \leq \frac{1}{t}$. Thus,

$$\sum_{j \in S} c(j) = \sum_{t=1}^{d'} c(j_t) \leq \sum_{t=1}^{d'} \frac{1}{t} \leq 1 + \log d' \leq 1 + \log d.$$

Hence,

$$p = \sum_{j=1..n} c(j) \le \sum_{S \in OPT} \sum_{j \in S} c(j) \le \sum_{S \in OPT} (1 + \log d) = (1 + \log d) \cdot k.$$

QED.

Knapsack, FPTAS

Knapsack Problem: Given objects $1, 2, \ldots, n$ that each has an integer size s_i and integer profit p_i, a capacity B, find a subset of items $S \subseteq [1, n]$, such that $\sum_{i \in S} s_i \le B$, and the total profit $p_S = \sum_{i \in S} p_i$ is maximized.

Application: Imagine a thief is trying to steal valuable items from a store. He has only a knapsack that can hold a certain weight. So he wants to select a combination of items that have the maximum total value, but the total weight is below the capacity of the knapsack.

Application: Imagine a small business owner receives more jobs than his capacity, and needs to decide which jobs to take within his capacity in order to maximize profit.

Knapsack has an interesting dynamic programming algorithm of which the time complexity is *pseudo-polynomial*.

Let $D[i, p]$ be the minimum weight needed to gain profit p with a subset of items from $[1, i]$. Clearly $D[i, 0] = 0$. For convenience, let $D[0, p] = \infty$ for any $p > 0$ to indicate that it is impossible.

In order to gain profit p from items 1 to i, item i can be either picked or not. If i is picked, then $D[i, p] = D[i - 1, p - p_i] + s_i$. If i is not picked, then $D[i, p] = D[i - 1, p]$. Thus,

$$D[i, p] = \min\{D[i - 1, p], D[i - 1, p - p_i] + s_i\}.$$

It is straightforward to have a dynamic programming algorithm to compute $D[i, p]$ for all i and p. Then find the maximum p that satisfies $D[n, p] \le B$. Backtrace from there.

Time complexity: computing $D[i, p]$ for each i and p takes $O(1)$ time. There are n possible i, and $P = \sum_i p_i$ possible p. Thus, the time complexity is $O(nP)$, where P is the total profit of all items.

Exercise: Think of a careful implementation of the same algorithm to use $O(n \cdot P_{opt})$ time, where P_{opt} is the profit of the optimal solution.

Although Knapsack-DP looks like a polynomial-time algorithm, it is not – because the value of P can be an exponential of the number of bits used to input all p_i. We say that Knapsack-DP has a pseudo-polynomial time complexity. When every p_i is a polynomial of n, then Knapsack-DP is polynomial-time. If without such a constraint, it has been proven that Knapsack is NP-hard.

But in practice, do we really need so many bits to encode a profit? If there is a one-million dollar item, does the thief really care about another item that is worth 100 dollars? This idea leads to the following approximation algorithm.

Let p_{max} be the maximum item value. We assume the item's size does not exceed B. Otherwise we can safely discard it from the computation.

For each i, let $p_i' = \left\lfloor p_i \cdot \frac{k}{p_{max}} \right\rfloor$ for a large number k that is to be determined later. This way, p_i' takes values from 0 to k, and the value of each item is approximately, proportionally represented by p_i'.

Treat p_i' as the new profit for each item. The previous dynamic programming algorithm can be used to find the optimal subset of items S_{alg} under the new profit values. Output S_{alg} as an approximation solution of the old problem. The time complexity is $O(n^2 \cdot k)$.

We want to prove the approximation ratio. The following useful fact is obvious from the definition

$$p_i' \le p_i \cdot \frac{k}{p_{max}} < p_i' + 1.$$

Suppose the profit of S_{alg} is ALG' under the new profits, and ALG under the old profits. Similarly, let S_{opt} be the optimal solution for the old problem. S_{opt} has profit OPT under the old profits, and profit OPT' under the new profits. Because ALG' is optimal under the new profits, we have $OPT' \le ALG'$.

Together with the fact that $p_i' \le p_i \cdot \frac{k}{p_{max}} < p_i' + 1$, we have the following:

$$OPT \cdot \frac{k}{p_{max}} \le OPT' + n \le ALG' + n \le ALG \cdot \frac{k}{p_{max}} + n.$$

Thus, $\frac{OPT}{ALG} \le 1 + \frac{n \cdot p_{max}}{k \cdot ALG} \le 1 + \frac{n}{k}$.

For any $\epsilon > 0$, let $k = n/\epsilon$, we get an approximation ratio $1 + \epsilon$ with time complexity $O(n^2 \cdot k) = O\left(n^3 \cdot \frac{1}{\epsilon}\right)$.

This is called a fully polynomial-time approximation scheme (FPTAS). It is stronger than PTAS. To achieve approximation ratio $1 + \epsilon$, a PTAS runs in $O\left(n^{f(\epsilon)}\right)$ time for some function f. But a FPTAS runs in time $O\left(n^c \cdot f(\epsilon)\right)$ for some constant c and function f.

Linear Programming for Weighted Vertex Cover

Let's examine the vertex cover again. Now each vertex has a weight associated to it. We want to find a vertex cover that minimizes the total weight of the vertices used. The unweighted vertex cover is therefore a special case, where each vertex has weight of 1.

Now the maximal-matching algorithm does not work anymore. We solve this weighted version with another standard approximation-algorithm design technique: linear-programming relaxation and rounding.

Let $V = \{v_1, \dots, v_n\}$ be the vertices and $E = \{e_1, \dots, e_m\}$ be the edges. Let w_i be the weight of v_i. Let $x_i = 0,1$ be variables indicating whether v_i is used in the vertex cover. Then the cost of the solution is

$$\sum_{i=1}^{n} w_i \cdot x_i .$$

To ensure x_i provides a cover, we need to ensure that for each edge $e = (v_i, v_j)$, at least one of x_i and x_j is 1. That is $x_i + x_j \geq 1$. Overall, we get an integer-linear-programming (ILP) problem:

$$\begin{cases} \min \sum_{i=1}^{n} w_i \cdot x_i ; \\ x_i + x_j \geq 1, \text{ for each edge } e = (x_i, x_j); \\ x_i = 0,1. \end{cases}$$

So, the weighted vertex cover problem can be solved by using a software package that can solve integer linear programming. But in general, integer linear programming is also NP-hard. (Vertex Cover is NP-hard, and we just provided a reduction from Vertex Cover to Integer Linear Programming). So, such a package will likely take exponential time in the worst case.

But Linear Programming (LP) is polynomial-time solvable. So we relax the ILP problem to an LP problem:

$$\begin{cases} \min \sum_{i=1}^{n} w_i \cdot x_i ; \\ x_i + x_j \geq 1, \text{ for each edge } e = (x_i, x_j); \\ 0 \leq x_i \leq 1. \end{cases}$$

This is called **LP relaxation**. Now we can find a solution $\tilde{x}_i$ in polynomial time. Under this solution, the optimized goal is minimized. Denote it with W_{LP}. Similarly, denote the optimal goal for ILP by W_{ILP} . We know that $W_{LP} \leq W_{ILP}$ because LP is a relaxation.

Next we need to build an integer solution of x_i from $\tilde{x}_i$. It is as simple as

$$x_i = \begin{cases} 0, \text{ if } \tilde{x}_i < \frac{1}{2} ; \\ 1, \text{ if } \tilde{x}_i \geq \frac{1}{2} . \end{cases}$$

Theorem: The above algorithm (LP relaxation + rounding) has approximation ratio 2.

Proof: Now we get a solution with total weight W_{ALG}. First, this is a vertex cover because each edge makes sure that $\tilde{x}_{i_j} + \tilde{x}_{i'_j} \geq 1$, so at least one of the two vertices is rounded to 1. Secondly,

the rounding will enlarge a variable to at most twice of the original value. So W_{ALG} is at most twice of W_{LP}. That is, $W_{ALG} \leq 2 \cdot W_{LP} \leq 2 \cdot W_{ILP}$.

QED.

Farthest String, Closest String, Linear-programming Relaxation

Farthest string: Given n length-L strings $s_1, \ldots, s_n$, find another string s, such that $\min_{1 \leq i \leq n} d(s, s_i)$ is maximized. Here d is Hamming distance, $n > 1$.

Closest string: Given n length-L strings $s_1, \ldots, s_n$, find another string s, such that $\max_{1 \leq i \leq n} d(s, s_i)$ is minimized. Here d is Hamming distance, $n > 1$.

Both problems were proved to be NP-hard.

In designing error-correction codes, the farthest string problem can be used to find a new code that is the farthest from all other existing codes.

In bioinformatics, these two problems were special cases of the more generalized Distinguishing String problem:

Distinguishing string: Given two sets of length-L strings S and S', two numbers d and d', find a distinguishing string t, such that $d(t, s) \leq d$ for every $s \in S$ and $d(t, s') \geq d'$ for every $s' \in S'$. Here d is Hamming distance.

The distinguishing string can be used to detect the presence of the DNA of a group of bacteria/viruses, while avoiding the false positives from the human DNA.

In this section, we study Farthest String and Closest String on the binary alphabet. But these algorithms can be easily extended to non-binary cases.

Chernoff's bound: Let $X_1, \ldots, X_n$ be n independent 0-1 random variables. $\Pr(X_i = 1) = p_i$. $X = \sum_{i=1}^{n} X_i$ be the sum. $\mu = E[X] = \sum_{i=1}^{n} p_i$. For any $\delta > 0$,

$$\Pr(X \geq (1+\delta)\mu) \leq \exp\left(-\frac{\delta^2 \mu}{3}\right),$$

$$\Pr(X \leq (1-\delta)\mu) \leq \exp\left(-\frac{\delta^2 \mu}{2}\right).$$

Lemma: If n is a polynomial of L, then the maximum distance d_{opt} for Farthest String is at least $\frac{L}{2} - O\left(\sqrt{L \log n}\right)$.

Proof: Consider a random binary string t. For any input string s_i, $d(t, s_i)$ is the summation of L independent 0-1 variables, each has probability $\frac{1}{2}$ to be 1. By Chernoff's bound,

$$\Pr\left(d(t,s_i) \le (1-\delta)\cdot\frac{L}{2}\right) \le \exp\left(-\frac{\delta^2 L}{4}\right).$$

Let $\delta = \frac{4\cdot\sqrt{\log n}}{\sqrt{L}}$, the above formula becomes

$$\Pr\left(d(t,s_i) \le \frac{L}{2} - 2\cdot\sqrt{L\log n}\right) \le \exp(-4\log n) = n^{-4}.$$

Adding all probabilities up for all i, we get

$$\Pr\left(\exists i \text{ such that } d(t,s_i) \le \frac{L}{2} - 2\cdot\sqrt{L\log n}\right) \le n^{-3}.$$

Thus,

$$\Pr\left(d(t,s_i) \ge \frac{L}{2} - 2\cdot\sqrt{L\log n} \text{ for every } i\right) \ge 1 - n^{-3}.$$

Since the probability is positive, there is at least one t such that $d(t,s_i) \ge \frac{L}{2} - 2\cdot\sqrt{L\log n}$ for every i.

QED.

Recall that this is called the probability method to prove the existence of something.

For a binary string s, we use $s[j]$ to denote the 0-1 letter at position j. We also use $s[j]$ as the 0-1 integer value. Let $t = x_1x_2\ldots x_L$ be the to-be-computed binary string. Then

$$d(t,s_i) = \sum_{j=1}^{L}\left(s_i[j] + (1-2\cdot s_i[j])\cdot x_j\right).$$

Notice that the right hand side is a linear combination of the 0-1 variables $x_1, x_2, \ldots, x_L$. The Farthest String problem is converted to the following ILP problem:

$$\begin{cases} \max d; \\ d(t,s_i) \ge d, \quad i = 1,\ldots,n; \\ x_j = 0 \text{ or } 1, \qquad j = 1,\ldots,L. \end{cases}$$

Note that $d(t,s_i)$ represents the linear combination $\sum_{j=1}^{L}\left(s_i[j] + (1-2\cdot s_i[j])\cdot x_j\right)$. The optimal solution of this ILP gives an optimal farthest string. Suppose d_{opt} be the optimal value of d for this ILP.

Relax the ILP to LP by replacing the $x_j = 0$ or 1 with condition $0 \le x_j \le 1$. LP can be solved in polynomial time to give us the optimal solution $0 \le \tilde{x}_j \le 1$. The corresponding value of d is $\tilde{d}_{opt}$. Since it is a relaxation, we know that $\tilde{d}_{opt} \ge d_{opt}$.

Trouble is that $\tilde{x}_j$ is not 0-1. Let's use the following **randomized rounding** procedure to obtain a 0-1 solution x_j: For each j, let $x_j = 1$ with probability $\tilde{x}_j$, and 0 with probability $1 - \tilde{x}_j$. Let $t = x_1 \dots x_L$ be the sequence obtained by this randomized rounding procedure.

Fix a string s_i.

$$\mu_i = E[d(t, s_i)] = \sum_{j=1}^{L} \left(s_i[j] + (1 - 2 \cdot s_i[j]) \cdot E[x_j]\right) = \sum_{j=1}^{L} \left(s_i[j] + (1 - 2 \cdot s_i[j]) \cdot \tilde{x}_j\right) \geq \tilde{d}_{opt} \geq d_{opt} .$$

By Chernoff's bound:

$$\Pr\left(d(t, s_i) \leq (1 - \delta) d_{opt}\right) \leq \Pr(d(t, s_i) \leq (1 - \delta)\mu_i) \leq \exp\left(-\frac{\delta^2 \mu_i}{2}\right) \leq \exp\left(-\frac{\delta^2 d_{opt}}{2}\right).$$

By the Lemma we can safely assume $d_{opt} \geq \frac{L}{3}$. Let $\delta = \frac{\sqrt{9 \log n}}{\sqrt{L}}$,

$$\Pr\left(d(t, s_i) \leq (1 - \delta) d_{opt}\right) \leq \exp\left(-\frac{3 \log n}{2}\right) = n^{-\frac{3}{2}} .$$

Adding up all input strings s_i, we have

$$\Pr\left(\exists i \text{ s.t. } d(t, s_i) \leq (1 - \delta) d_{opt}\right) \leq n^{-\frac{1}{2}} .$$

Therefore,

$$\Pr\left(d(t, s_i) \geq (1 - \delta) d_{opt} \text{ for every } i\right) \geq 1 - n^{-\frac{1}{2}} .$$

In another word, the t obtained with random rounding has a $1 - o(1)$ probability to be an approximation solution with ratio better than $1 - o(1)$. We obtained a randomized FPTAS.

Remark: LP-relaxation (plus randomized rounding) is a powerful method to design approximation algorithms.