2

How to Cope with NP-Completeness

Great Ideas in Theoretical Computer Science
Saarland University, Summer 2014

NP-hard optimization problems appear everywhere in our life. Under the assumption that $\mathsf{P} \neq \mathsf{NP}$ there is no efficient algorithm for any of these problems. One relaxation of these problems is to find an approximate solution with desired approximation guarantee. Theoreticians proved that efficient algorithms exist for certain NP-hard problems, and simple greedy strategy usually provides good approximation. On the downside, for a few problems even obtaining a reasonable approximation is as hard as finding the optimal solution. This line of research studies NP-hard problems from two aspects:

- *Approximation Algorithms* studies design of polynomial-time algorithms with certain approximation guarantees.
- *Inapproximability* studies the hardness of approximating NP-hard problems with certain approximation ratios.

In addition to these, *Parameterized Complexity* seeks for exact algorithms with running time $O(c^n)$ for smaller c and, besides the input size, the influence of other parameters in designing algorithms for hard problems. *Average-Case Complexity* associates input of an NP-hard problem with a distribution, and studies the complexity of hard problems in the average-case. *Study of Randomness* provides simple randomized algorithms for hard problems, and these randomized algorithms usually give a desirable output with high probability.

In today's lecture we give an introductory discussion on approximation algorithms, and inapproximability.

2.1 Exact Algorithms

We start with exact algorithms for NP-hard problems, and these algorithms always give optimal solutions of the problems, with super-polynomial time.

Enumeration & Local Search. We first discuss exact algorithms for NP-hard problems. The simplest exact algorithm is complete enumeration. For example, for the SAT problem, in order to find out whether a boolean formula over variables x_1 to x_n is satisfiable, we could simply iterate over all 2^n possible assignments.

Can we do better than enumerating 2^n possible assignments? Many short-cuts are possible. For example, if the formula contains a unit-clause, i.e., a clause consisting of a single literal, this literal must be set to true and hence only half of the assignments have to be tried. Note that unit-clauses arise automatically when we systematically try assignments, e.g., if $x_1 \vee x_2 \vee x_3$ is a clause and we have set x_1 and x_2 to false, then x_3 must be set to true. There are many rules of this kind that can be used to speed-up the search for a satisfying assignment. We refer the reader to [11] for an in-depth discussion. There has been significant progress on speeding-up SAT-solvers in recent years and good open source implementations are available.

Local Search. Local search is a strategy for solving computationally hard optimization problems. Basically a local search algorithm moves from one candidate solution to another in the searching space by applying local changes, until a solution with certain criteria is found. We illustrate this method by a simple randomized algorithm for the 3-SAT problem by Schöning [12] that runs in time $(4/3)^n$, where n is the number of variables. The algorithm adapted from [13] is described in Algorithm 2.1. For simplicity we assume that all clauses have at most k literals.

Algorithm 2.1 A randomized algorithm for SAT

```
1: $T \leftarrow 0$;
2: while $T \leq 2 \cdot \left(\frac{2\cdot(k-1)}{k}\right)^n \cdot \ln\frac{1}{\varepsilon}$ do
3:     $T \leftarrow T + 1$
4:     Choose an assignment $x \in \{0,1\}^n$ uniformly at random
5:     for count $\leftarrow 1$ to $C_k \cdot n$ do                          ▷ $C_k$ is a constant
6:         if $\varphi(x) = 1$ then return $\underline{\varphi \text{ is satisfiable}}$      ▷ This answer is always correct
7:         Let $C$ be a clause of $\varphi$ that is not satisfied
8:         Choose a literal in $C$ at random and flip the value of the literal in $x$
9: return $\underline{\varphi \text{ is unsatisfiable}}$                  ▷ This is incorrect with probability at most $\varepsilon$
```

Now we analyze Algorithm 2.1. Suppose that the input formula φ has n variables and φ is satisfiable. Let $x^\star$ be one such satisfiable assignment. We split the set of 2^n possible assignments into blocks, where block B_i consist of all assignments x whose Hamming distance to $x^\star$ is i, i.e.

$$B_i \triangleq \left\{ x \in \{0,1\}^n \;\middle|\; \sum_{1\leq j\leq n} x_j \oplus x_j^\star = i \right\},$$

where $\oplus$ represents the XOR operation of boolean variables.

Let x be our current assignment, and let C be a clause of φ that is not satisfied by x. Then all literals of C are set to false by x and $\ell \geq 1$ literals of C are set to true by $x^\star$. If we flip the value of one of these ℓ literals, the Hamming distance of x and $x^\star$ decreases by one. If we flip the value of one of the other at most $k - \ell$ literals of C, the Hamming distance will increase

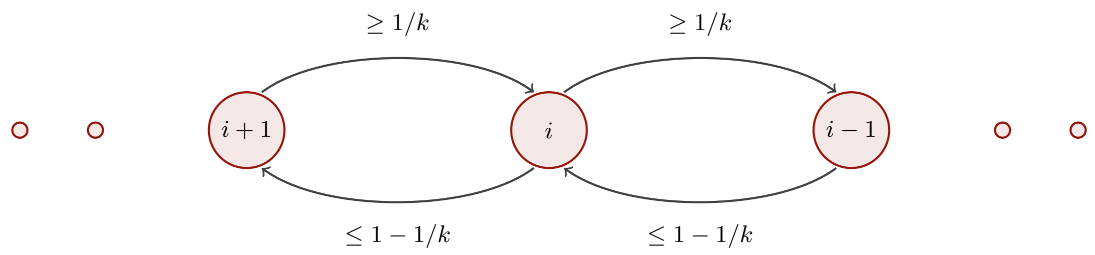

Figure 2.1: Explanation of Algorithm 2.1.

by one. Thus, if x belongs to B_i, the next x belongs to B_{i-1} with probality at least $1/k$ and belongs to B_{i+1} with probability at most $1-1/k$. The algorithm accepts at the latest when $x \in B_0$, and it may accept earlier when x equals some other satisfying assignment.

Hence, in order to analyze the algorithm, it suffices to analyze the random walk with states 0, 1, 2, . . . and transition probability $1/k$ from state i to state $i-1$ and probability $1-1/k$ from state i to state $i+1$, as illustrated in Figure 2.1. So we only need to analyze the probability of reaching state 0 when starting in state i and the expected length of a walk leading us from state i to state 0.

What should we expect? The walk has a strong drift away from zero and hence the probability of ever reaching state 0 should decrease exponentially in the index of the starting state. For the same reason, most walks will never reach state 0. However, walks that reach state 0 should be relatively short. This is because long walks are more likely to diverge.

Now we start our formal analysis. Remember that the initial assignment x is chosen uniformly at random from $\{0,1\}^n$. We look at the Hamming distance between x and $x^\star$. Clearly, the distribution is binomially distributed, i.e.,

$$\mathbf{Pr}\left[\text{Hamming distance} = i\right] = \binom{n}{i} \cdot 2^{-n}. \tag{2.1}$$

We first state two claims before formally proving the performance of the algorithm.

Claim 2.1. *The probability of reaching state 0 from state i is exponential in i. Formally, it holds that*

$$\mathbf{Pr}\left[\textit{absorbing state 0 is reached} \mid \textit{process started in state } i\right] = \left(\frac{1}{k-1}\right)^i.$$

Proof. Let p be the probability of reaching state 0 from state 1. Such a walk has the following form: the last transition is from state 1 to state 0, with probability $1/k$. Before that it performs a walk of length $2t \geq 0$ starting from and ending at state 1 without visiting state 0, with probability $((k-1)/k)p)^t$. Therefore it holds that

$$p = \frac{1}{k}\sum_{t\geq 0}\left(\frac{k-1}{k}\cdot p\right)^t = \frac{1}{k}\cdot\frac{1}{1-\frac{k-1}{k}\cdot p} = \frac{1}{k-(k-1)\cdot p}.$$

This equation has solutions $p=1$ and $p=\frac{1}{k-1}$. We exclude the former on semantic reasons.

Thus the probability of reaching state $i-1$ from state i is $1/(k-1)$ and the probability of reaching it from state i is the ith power of this probability. □

Claim 2.2.

$$\mathbf{E}\left[\#\textit{ steps until state 0 is reached} \;\middle|\; \begin{array}{l}\textit{process started in state } i \textit{ and the absorb-}\\ \textit{ing state 0 is reached}\end{array}\right] = C_k i,$$

where $C_k = 1 + \frac{2}{k(k-2)}$.

Proof. Let L be the expected length of a walk from state 1 to state 0. Such a walk has the following form: the last transition is from state 1 to state 0 (length 1). Before that it performs a walk of length $2t \geq 0$ starting from and ending at state 1 without visiting state 0, with probability

$$\left(\frac{k-1}{k}\cdot p\right)^t \cdot \frac{1}{k},$$

where p is the same as in Claim 2.1 and the length of such walk is $t\cdot(L+1)$. Thus

$$\begin{aligned} L &= 1 + \sum_{t\geq 0}\left(\frac{k-1}{k}\cdot p\right)^t \cdot \frac{1}{k}\cdot t\cdot(1+L) = 1 + (L+1)\cdot\sum_{t\geq 1} t\cdot k^{-t-1} \\ &= 1 + (L+1)\sum_{t\geq 1}\frac{\mathrm{d}}{\mathrm{d}k}(-k^{-t}) = 1 - (L+1)\frac{\mathrm{d}}{\mathrm{d}k}\frac{1/k}{1-1/k} \\ &= 1 - (L+1)\frac{\mathrm{d}}{\mathrm{d}k}(k-1)^{-1} = 1 + (L+1)(k-1)^{-2}. \end{aligned}$$

The solution of this equation is $L = \frac{k^2-2k+2}{k(k-2)}$, and therefore the expected length of a walk starting from i and reaching 0 is i times this number. □

Theorem 2.3 (Schöning)**.** *There is an algorithm that decides the satisfiability of a boolean formula with n variables, m clauses, and at most k literals per clause. The algorithm has one sided error ε and runs in time* $O\left(\left(\frac{2(k-1)}{k}\right)^n \cdot \mathrm{poly}(n,m)\cdot \ln\frac{1}{\varepsilon}\right)$.

Proof. Combing (2.1), Claim 2.1 and Claim 2.2, we have that

$$\begin{aligned} &\mathbf{Pr}\left[\text{the absorbing state 0 is reached}\right] \\ &= \sum_{0\leq i\leq n}\binom{n}{i}2^{-n}\left(\frac{1}{k-1}\right)^i = 2^{-n}\left(1+\frac{1}{k-1}\right)^n = \left(\frac{k}{2(k-1)}\right)^n, \end{aligned} \tag{2.2}$$

and the expected number of steps for a random walk reaching state 0 is

$$\begin{aligned} &\mathbf{E}\left[\#\text{ of steps until 0 is reached} \mid \text{0 is reached}\right] \\ &= \sum_{i\geq 0} C_k\cdot i\cdot\binom{n}{i}\cdot 2^{-n} = C_k\cdot\frac{n}{2^n}\cdot\sum_{i\geq 1}\binom{n-1}{i-1} = \frac{C_k\cdot n}{2}. \end{aligned}$$

By Markov's inequality the probability that a non-negative random variable is less than twice its expectation is at least 1/2. Thus

$$\mathbf{Pr}\left[\,0 \text{ is reached in at most } C_k n \text{ steps} \mid 0 \text{ is reached}\,\right] \geq 1/2. \tag{2.3}$$

Combining (2.2) and (2.3), we obtain

$$\mathbf{Pr}\left[\,\text{after at most } C_k n \text{ steps the state 0 is reached}\,\right] \geq \frac{1}{2}\left(\frac{k}{2(k-1)}\right)^n .$$

Now we set

$$p = \frac{1}{2}\left(\frac{k}{2(k-1)}\right)^n ,$$

and the probability that a satisfiable assignment $x^\star$ is reached from a random x is at least p. If we repeat the experiment T times, the probability that we never reach $x^\star$ is bounded by $(1-p)^T = \exp(T\ln(1-p)) \leq \exp(-Tp)$, since $\ln(1-p) \leq -p$ for $0 \leq p < 1$. In order to have error probability at most ε, it suffices to choose T such that $\exp(-Tp) \leq \varepsilon$, i.e.,

$$T \geq \frac{1}{p}\ln\frac{1}{\varepsilon} = 2\left(\frac{2(k-1)}{k}\right)^n \ln\frac{1}{\varepsilon}. \qquad \square$$

Branch-and-Bound-and-Cut. For optimization problems, the branch-and-bound and branch-and-bound-and-cut paradigm are very useful. We illustrate the latter for the Travelling Salesman Problem.

Problem 2.4 (Traveling Salesman Problem, TSP)**.** *Given a complete graph with nonnegative edge costs, find a minimum cost cycle visiting every vertex exactly once.*

The definition of subtour elimination LP for the traveling salesman problem in a graph $G = (V, E)$ is given below. Let c_e be the cost of edge e. We have a variable x_e for each edge.

$$\begin{aligned} \min \sum_e c_e x_e & & \\ \text{subject to } \sum_{e\in\delta(v)} x_e &= 2 & \text{for each vertex } v \\ \sum_{e\in\delta(S)} x_e &\geq 2 & \text{for each set } S \subseteq V, \emptyset \neq S \neq V \\ x_e &\geq 0 & \end{aligned}$$

We solve the LP above. If we are lucky, the solution is integral and we found the optimal tour. In general, we will not be lucky. We select a fractional variable, say x_e, and branch on it, i.e., we generate the subproblems $x_e = 0$ and $x_e = 1$. We solve the LP for both subproblems. This gives us lower bounds for the cost of optimal tours for both cases. We continue to work on the problem for which the lower bound is weaker (= lower). We may also want to introduce additional cuts. There are additional constraints known for the TSP. There are also generic methods for introducing additional cuts. We refer the reader to [1] for a detailed computational study of the TSP.

2.2 Approximation Algorithms

Approximation algorithms construct provably good solutions and run in polynomial-time. Algorithms that run in polynomial-time but do not come with a guarantee on the quality of the solution found are called *heuristics*. Approximation algorithms may start out as heuristics. Improved analysis turns the heuristic into an approximation algorithm.

2.2.1 Vertex Cover

Our first approximation algorithm is for the vertex cover problem. We will show that a simple greedy strategy gives a 2-appproximate solution.

Problem 2.5 (Vertex Cover)**.** *Given a graph $G = (V, E)$, find a subset $S \subseteq V$ of minimum size such that every edge in G has at least one endpoint incident at S.*

Lemma 2.6. *A factor 2-approximation to the optimum vertex cover can be computed in polynomial-time.*

Proof. For an input graph G, we compute a maximal matching M of G, and output the endpoints of the edges in M. A maximal matching can be computed greedily. For instance, we can iterate over the edges in arbitrary order, and add one edge e in the matching if the endpoints of e are uncovered.

Clearly, the size of the cover is $2|M|$. Since an optimal cover must contain at least one endpoint of every edge in M, we obtain a 2-approximate solution. □

2.2.2 Set Cover

Problem 2.7 (Set Cover)**.** *Let U be a set of n items, and $S_1, \ldots, S_k$ be subsets of U. Every set S_i has cost $\text{cost}(S_i) \geq 0$. Find a miminum cost collection of sets that covers all items in U.*

We show a simple greedy strategy that yields a $O(\log n)$-factor approximation for this NP-hard problem. One intuition behind the algorithm is that, we always pick the most effective set, i.e., the set that covers uncovered elements at the smallest cost per newly covered element. More formally, let C be the set of covered elements before an iteration. Initially, C is the empty set. Define the cost-effectiveness of S_i as $c(S_i)/|S_i \setminus C|$, i.e., as the cost per uncovered elements covered by S_i. Given this, each iteration simply consists of finding the best S_i and updating set C. See Algorithm 2.2 for formal description.

Let OPT be the cost of the optimum solution. For the analysis, number the elements in the ground set U in the order in which they are covered by the algorithm above. Let e_1 to e_n be the elements in this numbering.

Lemma 2.8. *For all k,* $\text{price}(e_k) \leq \mathsf{OPT}/(n - k + 1)$.

Proof. Let $C = \{e_1, \ldots, e_{i-1}\}$ be the set of covered elements after t iterations by the greedy algorithm. Assume that the uncovered $n - i + 1$ elements in $U \backslash C$ are covered by ℓ sets

Algorithm 2.2 Greedy Set Cover Algorithm [5, 9, 10]

```
1: $C \leftarrow \emptyset$;
2: while $C \neq U$ do
3:     Let $i$ be the index minimizing $c(S_i)/|S_i \setminus C|$.
4:     $C \leftarrow C \cup S_i$;
5:     set the price of every $e \in S_i \setminus C$ to $\mathrm{price}(e) = c(S_i)/|S_i \setminus C|$
6: Return the picked set.
```

$\left\{S^\star_{j_1}, \ldots, S^\star_{j_\ell}\right\}$ in the optimal solution. Therefore

$$\begin{aligned}
\frac{\mathsf{OPT}}{n-i+1} &\geq \frac{1}{n-i+1}\sum_{k=1}^{\ell} c\left(S^\star_{j_k}\right) = \frac{1}{n-i+1}\sum_{k=1}^{\ell}\left|S^\star_{j_k}\backslash C\right| \frac{c\left(S^\star_{j_k}\right)}{\left|S^\star_{j_k}\backslash C\right|} \\
&\geq \frac{\sum_{k=1}^{\ell}\left|S^\star_{j_k}\backslash C\right|}{n-i+1} \min_{k\in[l]} \frac{c\left(S^\star_{j_k}\right)}{\left|S^\star_{j_k}\backslash C\right|} \geq \min_{k\in[l]} \frac{c\left(S^\star_{j_k}\right)}{\left|S^\star_{j_k}\backslash C\right|} \\
&\geq \min_j \frac{c\left(S_j\right)}{\left|S_j\backslash C\right|} = \mathrm{price}\left(e_i\right).
\end{aligned}$$

□

Theorem 2.9. *The greedy algorithm for the set cover problem produces a solution of cost at most $H_n\mathsf{OPT}$, where $H_n = 1 + 1/2 + 1/3 + \ldots + 1/n$ is the n-th Harmonic number.*

Proof. By Lemma 2.8 the cost of the solution produced by the greedy algorithm is bounded by

$$\mathsf{OPT}\sum_{1\leq k\leq n}\frac{1}{n-k+1} = \mathsf{OPT}\cdot H_n.$$

□

Since $H_n = O(\log n)$, Algorithm 2.2 has approximation ratio of $O(\log n)$.

The following example shows that no better approximation ratio can be proved for this algorithm: Let the ground set be of size n. We have the following sets: a singleton set covering element i at cost $1/i$ and a set covering all elements at cost $1+\varepsilon$. The optimal cover has a cost of $1+\varepsilon$, however the greedy algorithm chooses the n singletons in the sequence $\{n, n-1, \ldots, 1\}$.

The following result shows that, under certain complexity assumptions, essentially this simply greedy strategy achieves the best approximation ratio that we can hope for.

Theorem 2.10 ([7])**.** *If there is an approximation algorithm for set cover with approximation ratio $o(\log n)$, then $\mathsf{NP} \subseteq \mathsf{ZTIME}\left(n^{O(\log\log n)}\right)$, where $\mathsf{ZTIME}(T(n))$ is the set of problems which have a Las Vegas algorithm with runtime $T(n)$.*

2.2.3 Knapsack

Problem 2.11 (Knapsack Problem). *Given a set $S = \{a_1, \ldots, a_n\}$ of objects, with specified weight $\mathrm{weight}(a_i) = w_i$ and value $\mathrm{value}(a_i) = v_i$. For a capacity W, find a subset of objects whose total weight is bounded by W and total value is maximized.*

For simplicity we may assume $w_i \leq W$ for all i.

The Greedy Heuristic. Let us first look at a simple greedy heuristic. We order the items a_i by value per unit weight, i.e., by $\mathrm{value}(a_i)/\mathrm{weight}(a_i)$. We start with the empty set S of items and then iterate over the items in decreasing order of the ratio $\mathrm{value}(a_i)/\mathrm{weight}(a_i)$, i.e., if the current weight plus the weight of the current item a_i does not exceed the weight bound, then we add a_i, otherwise we discard a_i. Let V_{greedy} be the value computed.

The greedy heuristic is simple, but no good. Consider the following example. Let $S = \{a_1, a_2\}$. The first item a_1 has value 1, and weight 1, and the second item a_2 has value 99 and weight 100. Let $W = 100$. Since $1/1 > 99/100$, the greedy heuristic considers the first item first and adds it to the knapsack. Having added the first item, the second item will not fit. Thus the heuristic produces a value of 1. However, the optimum is 99.

A small change turns the greedy heuristic into an approximation algorithm that guarantees half of the optimum value.

$$\text{Return the maximum of } V_{\mathrm{greedy}} \text{ and } v_{\max}, \quad \text{where } v_{\max} \triangleq \max_i\{v_i\}.$$

Lemma 2.12. *This modified greedy algorithm runs in time $O(n \log n)$, and achieves a value of $V_{\mathrm{opt}}/2$, where V_{opt} is the optimal value of the problem.*

Proof. Since we only need to sort items according to the ratio of value to weight and iterate over items, the runtime $O(n \log n)$ is obvious.

Now we look at the approximation ratio of the algorithm. Order the items by decreasing ratio of value to weight, i.e.,

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}.$$

Let k be minimal such that $w_1 + \ldots + w_{k+1} > W$. Then $w_1 + \ldots + w_k \leq W$. The greedy algorithm will pack the first k items and maybe some more. Thus $V_{\mathrm{greedy}} \geq v_1 + \ldots + v_k$.

The value of the optimal solution is certainly bounded by $v_1 + \ldots + v_k + v_{k+1}$. Thus

$$V_{\mathrm{opt}} \leq v_1 + \ldots + v_k + v_{k+1} \leq V_{\mathrm{greedy}} + v_{\max} \leq 2 \max\left\{V_{\mathrm{greedy}}, v_{\max}\right\}. \qquad \square$$

Dynamic Programming. We assume that the values are integers, and let $v_{\max}$ be the maximum value of item in set S. We show how to compute an optimal solution in time $O(n^2 \cdot v_{\max})$.

Clearly, the maximum value of the knapsack is bounded by $n \cdot v_{\max}$. We fill a table $B[0, n \cdot v_{\max}]$ such that, at the end, $B[s]$ is the minimum weight of a knapsack of value s for $0 \leq s \leq n \cdot v_{\max}$. We fill the table in phases 1 to n and maintain the invariant that after the ith phase:

$$B[s] = \min\left\{ \sum_{1\leq j\leq i} w_j x_j : \sum_{1\leq j\leq i} v_j x_j = s \text{ and } x_j\text{'s} \in \{0,1\} \right\}.$$

The minimum of the empty set is ∞. The entries of the table are readily determined. With no item, we can only obtain value 0, and we do so at a weight of zero. Consider now the ith item. We can obtain a value of s in one of two ways: either by obtaining value s with the first $i-1$ items or by obtaining value $s - v_i$ with the first $i-1$ items. We choose the better of the two alternatives. The formal description in shown in Algorithm 2.3 below.

Algorithm 2.3 An approximation algorithm for the Knapsack Problem

```
1: $B[0] \leftarrow 0$;
2: for $i \leftarrow 1$ to $n \cdot v_{\max}$ do $B[i] = \infty$
3: for $i \leftarrow 1$ to $n$ do
4:     for $s \leftarrow n \cdot v_{\max}$ downto $v_i$ do
5:         $B[s] \leftarrow \min(B[s], B[s - v_i] + w_i)$
6: Return the maximum $s$ such that $B[s] \le W$
```

It is straightforward to have the following result.

Lemma 2.13. *Algorithm 2.3 solves the knapsack problem in time $O(n^2 \cdot v_{\max})$.*

A small trick improves the running time to $O(n \cdot V_{\text{opt}})$. We maintain a value K which is the maximum value seen so far. We initialize K to zero and replace the inner loop (Line 3–5) by

```
$K \leftarrow K + v_i$
for $s \leftarrow K$ downto $v_i$ do
    $B[s] \leftarrow \min(B[s], B[s - v_i] + w_i)$
```

Observe that this running time is NOT polynomial as $v_{\max}$ may be exponential in the size of the instance.

Scaling. We now improve the running time by scaling at the cost of giving up optimality. We will see that we can stay arbitrarily close to optimality.

Let S be an integer. We will fix it later. Consider the modified problem, where the scale the values by S, i.e., we set $\hat{v}_i = \lfloor v_i / S \rfloor$. We can compute the optimal solution to the scaled problem in time $O(n^2 v_{\max}/S)$. We will next show that an optimal solution to the scaled problem is an excellent solution to the original problem.

Let $x = (x_1, \ldots, x_n)$ be an optimal solution of the original instance and let $y = (y_1, \ldots, y_n)$ be an optimal solution of the scaled instance, i.e.,

$$V_{\text{opt}} = \sum_i v_i x_i = \max \left\{ \sum_{1 \le i \le n} v_i z_i : \quad \sum_{1 \le i \le n} w_i z_i \le W \text{ and } z_i\text{'s} \in \{0,1\} \right\}, \text{and}$$

$$\sum_i \hat{v}_i y_i = \max \left\{ \sum_{1 \le i \le n} \hat{v}_i z_i : \quad \sum_{1 \le i \le n} w_i z_i \le W \text{ and } z_i\text{'s} \in \{0,1\} \right\}.$$

Let $V_{\text{approx}} = \sum_i v_i y_i$ be the value of the knapsack when filled according to the optimal solution of the scaled problem. Then

$$\begin{aligned}
V_{\text{approx}} = \sum_i v_i y_i = S \sum_i \frac{v_i}{S} y_i \\
&\geq S \sum_i \left\lfloor \frac{v_i}{S} \right\rfloor y_i \\
&\geq S \sum_i \left\lfloor \frac{v_i}{S} \right\rfloor x_i && \text{since } y \text{ is an optimal solution of scaled instance} \\
&\geq S \sum_i \left(\frac{v_i}{S} - 1 \right) x_i \\
&\geq \sum_i v_i x_i - S \sum_i x_i \\
&\geq \sum_i v_i x_i - n \cdot S \\
&= V_{\text{opt}} - n \cdot S. && \text{since } x \text{ is an optimal solution of original instance}
\end{aligned}$$

Thus

$$V_{\text{approx}} \geq \left(1 - \frac{n \cdot S}{V_{\text{opt}}} \right) V_{\text{opt}}.$$

It remains to choose S. Let $\varepsilon > 0$ be arbitrary. Set $S = \max(1, \lfloor \varepsilon V_{\text{opt}}/n \rfloor)$. Then $V_{\text{approx}} \geq (1-\varepsilon)V_{\text{opt}}$, since $V_{\text{approx}} = V_{\text{opt}}$ if $S = 1$. The running time becomes $O(\min(n \cdot V_{\text{opt}}, n^2/\varepsilon))$.

This is nice, but the definition of S involves a quantity that we do not know. The modified greedy algorithm comes to rescue. It determines V_{opt} up to a factor of two. We may use the approximation instead of the true value in the definition of S. The following theorem summarized our result.

Theorem 2.14. *For any $\varepsilon > 0$, there is an algorithm that runs in time $O(n^2/\varepsilon)$ and computes a solution to the knapsack problem with $V_{\text{approx}} \geq (1-\varepsilon) \cdot V_{\text{opt}}$.*

2.2.4 Maximum Satisfiability

We continue with Maximum Satisfiability problem. Comparing with the SAT problem which asks whether a boolean formula is satisfiable, Max-k-SAT problem asks an assignment that maximizes the total number of satisfied clauses.

Problem 2.15 (Max-k-SAT)**.** *Let Φ be a CNF formula where each clause has at most k literals. Find an assignment that maximizes the total number of satisfied clauses.*

Lemma 2.16. *Let Φ be a formula in which every clause has exactly three distinct literals. There is a randomized algorithm that satisfies in expectation 7/8 of the clauses.*

Proof. Consider a random assignment x. For any clause C, the probability that x satisfies C is $7/8$ because exactly one out of the eight possible assignments to the variables in C

are not satisfied. Thus the expected number of satisfied clauses is 7/8 times the number of clauses. □

It is crucial for the argument above that every clause has exactly three literals and that this literals are distinct. Clauses of length 1 are only satisfied with probability $1/2$ and clauses of length 2 only with probability $3/4$. So the randomized algorithm does not so well when there are short clauses. However, there is an algorithm that does well on short clauses, see [14] for detailed discussion.

2.3 The PCP-Theorem (Probabilistically Checkable Proofs)

Since the discovery of NP-completeness in 1972, researchers have tried to find efficient algorithms for approximating optimal solutions of NP-hard problems. They also realized that even providing approximate solutions of an NP-hard problem may be difficult. The PCP-theorem discovered in 1992 shows that for certain problems computing an approximate solution is as hard as computing the exact solutions. Besides this, the PCP-theorem gives another equivalent formulation of set NP.

New Characterizations of NP. Loosely speaking, a probabistically checkable proof system for a language L is a randomized algorithm having direct access to individual bits of a binary string. This string presents a proof, and typically will be accessed only partially by the verifier. The verifier is supposed to decide whether a given input x belongs to the language L based on the partial information it reads and coin tosses. The requirement is that the verifier always accepts a string $x \in L$, and rejects a string $x \notin L$ with probability at least $1/2$. The formal definition is as follows:

Definition 2.17 (Probabilistic Checkable Proofs-PCP)**.** *A probabilistically checkable proof system of a language L is a probabilistic polynomial-time algorithm M, called verifier, such that*

- *Completeness: For every $x \in L$, there exists an oracle π_x such that*

$$\mathbf{Pr}\left[\, M^{\pi_x}(x) = 1 \,\right] = 1.$$

- *Soundness: For every $x \notin L$ and every oracle π it holds that*

$$\mathbf{Pr}\left[\, M^{\pi}(x) = 1 \,\right] \leq 1/2,$$

  where the probability is taken over M's internal coin tosses.

Definition 2.18 (Complexity of PCP(r, q))**.** *Let $r, q : \mathbb{N} \mapsto \mathbb{N}$ be two functions. The complexity class* PCP(r, q) *consists of languages having a probabilistically checkable proof system M such that*

- *On input $x \in \{0,1\}^\star$, the verifier M uses at most $r(|x|)$ coin tosses.*
- *On input $x \in \{0,1\}^\star$, the verifier M makes at most $q(|x|)$ queries.*

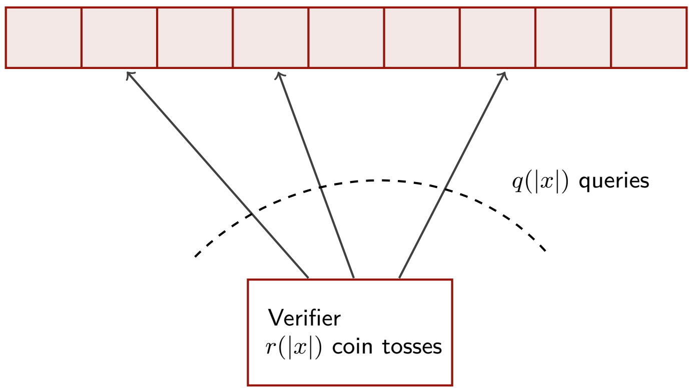

Figure 2.2: Explanation of a PCP system. With the help of an oracle, the verifier only reads $q(|x|)$ bits of the proof.

Figure 2.2 explains the role of $q(|x|)$ and $r(|x|)$ in a verifier.

One important consequence of the PCP-theorem is a new characterization of NP. From last lecture, we know that a language L belongs to NP if and only if there is a deterministic polynomial-time algorithm V (called the verifier), such that for any $x \in \{0,1\}^\star$ it holds that

- If $x \in L$, there is a certificate y such that V accepts.
- If $x \notin L$, there is no y such that V accepts.

Based on PCP-systems we can show the following theorem:

Theorem 2.19 ([2, 4])**.** NP $=$ PCP$(O(\log n), O(1))$.

The direction PCP$(O(\log n), O(1)) \subseteq$ NP is easy (Guess y and then run V for all random strings of length $c \log n$. Accept if V accepts for all random strings).

The other direction is a deep theorem. The crux is bringing the error probability below $1/2$. With a larger error bound, the statement is easy. Let us look at the following algorithm for SAT. The verifier interprets y as an assignment and the random string as the selector of a clause C. It checks whether the assignment satisfies the clause. Clearly, if the input formula is satisfiable, the verifier will always accept (if provided with a satisfying assignment). If the input formula is not satisfiable, there is always at least one clause that is not satisfied. Therefore the verifier will accept with probability at most $1 - 1/m$, where m is the number of clauses.

The PCP-theorem is a powerful tool to show hardness of approximation results. To illustrate this, we look at a toy problem.

Problem 2.20 (Maximize Accept Probability)**.** *Let V be a* PCP$(O(\log n), O(1))$*-verifier for SAT. On input Φ, find a proof y that maximizes the acceptance probability of V.*

We use the PCP-theorem to show that this problem does not have a 1/2-approximation algorithm unless P $=$ NP.

Theorem 2.21. *Maximize Accept Probability has no factor $1/2$ approximation algorithm unless* P $=$ NP.

Proof. The proof is by contradiction, and we assume that there is such algorithm $\mathcal{A}$. Let Φ be any formula and let y be the proof returned by algorithm $\mathcal{A}$. We run V with y and all random strings and compute the acceptance probability p. If p is less than $1/2$, then we declare Φ non-satisfiable. Otherwise we declare Φ satisfiable.

Why is this correct? If Φ is satisfiable, there is proof with acceptance probability 1. Hence algorithm $\mathcal{A}$ must return a proof with acceptance probability at least $1/2$. If Φ is not satisfiable, there is no proof with acceptance probability at least $1/2$. In particular, for the proof returned by algorithm $\mathcal{A}$, we must have $p < 1/2$. □

Hardness of Approximating Max-3-SAT. We show how to use the PCP-theorem to prove hardness of approximating Max-3-SAT problem.

Theorem 2.22. *There is no approximation algorithm for Max-3-SAT with an approximation guarantee of $1-\varepsilon$ for a constant ε, assuming* P $\neq$ NP.

Proof. We give a gap-introducing reduction from SAT to MAX-3-SAT. Let V be a PCP$(O(\log n), O(1))$-verifier for SAT. On an input Φ of length n, verifier V uses a random string r of length $c \log n$ and inspects at most q positions of the proof, where q is a constant.

Hence there are at most qn^c bits of the proof that are inspected for all random strings. Let $B = B_1 \ldots B_{qn^c}$ be a binary string that encodes these places of the proof. The other places of the proof are irrelevant. We will construct a formula Ψ over variables B_1 to B_{qn^c} with the following properties: (1) If Φ is satisfiable, Ψ is satisfiable. (2) If Φ is not satisfiable, then any assignment does not satisfy a constant fraction of the clauses in Ψ.

For any random string r, the verifier inspect a particular set of q bits, say the bits indexed by $i(r,1)$ to $i(r,q)$. Let $f_r(B_{i(r,1)}, \ldots, B_{i(r,q)})$ be a boolean function of q variables that is one if and only if the verifier accepts with random string r and the q bits of the proof as given by $B_{i(r,1)}$ to $B_{i(r,q)}$.

Notice that this construction have the following two properties: (1) If Φ is satisfiable, there is a proof y that makes V accept with probability one. Set B according to y. Then all functions f_r, $r \in \{0,1\}^{c \log n}$, evaluate to true. (2) If Φ is not satisfiable, every proof makes V accept with probability less than $1/2$. Hence every B does not satisfy more than half of the functions f_r. Each f_r is a function of q variables and hence has a conjunctive normal form Ψ_r with at most 2^q clauses. Each clause is a disjunction of q literals. An assignment that sets f_r to 0 does not satisfy at least one the clauses in Ψ_r.

Let Ψ' be the conjunction of the Ψ_r, i.e., $\Psi' = \bigwedge_r \Psi_r$. We finally turn Ψ' into a formula Ψ with exactly three literals per clause using the standard trick. Consider a clause $C = x_1 \vee x_2 \vee \ldots \vee x_q$. Introduce new variables y_1 to y_{q-2} and consider

$$C' = (x_1 \vee x_2 \vee y_1) \wedge (\neg y_1 \vee x_3 \vee y_2) \wedge \ldots \wedge (\neg y_{q-2} \vee x_{q-1} \vee x_q).$$

An assignment satisfying C can be extended to an assignment satisfying C'. Conversely, an assignment that does not satisfy C has at least one unsatisfied clause in C'.

What have we achieved? Ψ consists of no more than $n^c 2^q(q-2)$ clauses. Each clause has 3 literals. Therefore we have that: (1) If Φ is satisfiable, there is a proof y that makes V accept with probability one. Set B according to y. Then all functions f_r, $r \in \{0,1\}^{c \log n}$, evaluate to true and hence Ψ is satisfied. (2) If Φ is not satisfiable, every proof makes V accept with probability less than 1/2. Hence, every B does not satisfy more than half of the functions f_r, and thus B has more than $n^c/2$ unsatisfied clauses in Ψ. □

By setting $\alpha = (n^c/2)/(n^c 2^q(q-2)) = 1/((q-2)2^{q+1})$, we have the following result:

Theorem 2.23. *The above is a gap introducing reduction from SAT to MAX-3-SAT with* $\alpha = 1/((q-2)2^{q+1})$.

The theorem can be sharpened considerably. Johan Håstad showed that the theorem holds for any $\alpha = 1/8 - \varepsilon$ for any $\varepsilon > 0$.

2.4 Further Reading

The book *Approximation Algorithms* by Vijay Vazirani covers basic techniques for designing approximation algorithms [14]. The first connection between PCP and hardness of approximation was shown in [8], which shows the connections to max-Clique. The connection to max-3-SAT was given in [3]. A simplified proof of the PCP theorem is shown by Irit Dinur [6].

References

[1] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. *The traveling salesman problem: a computational study*. Princeton University Press, 2006.

[2] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and hardness of approximation problems. In *33rd Annual IEEE Symposium on Foundations of Computer Science (FOCS'92)*, pages 14–23, 1992.

[3] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, 1998.

[4] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs; a new characterization of NP. In *33rd Annual IEEE Symposium on Foundations of Computer Science (FOCS'92)*, pages 2–13, 1992.

[5] V. Chvátal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[6] Irit Dinur. The pcp theorem by gap amplification. *J. ACM*, 54(3):12, 2007.

[7] Uriel Feige. A threshold of ln *n* for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

[8] Uriel Feige, Shafi Goldwasser, László Lovász, Shmuel Safra, and Mario Szegedy. Interactive proofs and the hardness of approximating cliques. *J. ACM*, 43(2):268–292, 1996.

[9] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, 1974.

[10] Lászlo Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

[11] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to DPLL(t). *J. ACM*, 53(6):937–977, 2006.

[12] Uwe Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *40th Annual IEEE Symposium on Foundations of Computer Science (FOCS'99)*, pages 410–414, 1999.

[13] Uwe Schöning. New algorithms for k-sat based on the local search principle. In *26th International Symposium on Mathematical Foundations of Computer Science (MFCS'01)*, pages 87–95, 2001.

[14] Vijay V. Vazirani. *Approximation algorithms*. Springer, 2001.