How to Cope with NP-Completeness

Kurt Mehlhorn

June 12, 2013

NP-complete problems arise frequently in practice. In today's lecture, we discuss various approaches for coping with NP-complete problems.

When NP-completeness was discovered, algorithm designers could take it as an excuse. It is true, that I have not found an efficient algorithm. However, nobody will ever find an efficient algorithm (unless P = NP).

Nowadays, NP-completeness is considered a challenge. One searchers for exact algorithms with running time c^n with small c, one searches for approximation algorithms with a good approximation ratio, one tries to prove that the problem is hard to approximate unless P = NP, one searches for special cases that are easy to solve or easy to approximate, and one introduces parameters besides input size to describe the problem and to analyze the running time in.

Contents

1 Exact Algorithms

1.1 Enumeration

The simplest exact algorithm is complete enumeration. For example, 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.

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 [NOT06] 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.

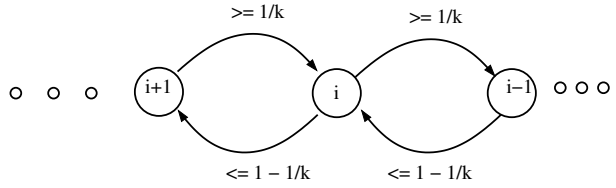
Local Search There is a simple randomized algorithm due to Schöning (1999) that runs in time $(4/3)^n$ where n is the number of clauses and decides 3-SAT with high probability [Sch99]. The algorithm works as follows (adapted from [Sch01]). We assume that all clauses have at most k literals.

do $T = 2\left(\frac{2(k-1)}{k}\right)^n \ln\frac{1}{\varepsilon}$ times (this choice guarantees an error probability of ε
  Choose an assignment $x \in \{0,1\}^n$ uniformly at random;
  do $C_k n$ times (C_k is a (small) constant; see below)
    if x is satisfying, stop and accept (this answer is always correct)
    Let C be a clause that is not satisfied
    choose a literal in C at random and flip the value of the literal in x.
declare the formula unsatisfiable (this is incorrect with probability at most ε)

The algorithm is surprisingly simple to analyze. The analysis determines the value of T as a function of the desired error probability.

Suppose that the input formula F is satisfiable and let x^* be some satisfying assignment. We split the set of all assignments into blocks. Block B_i contains all assignments with Hamming distance exactly i to x^*.

Let x be our current assignment and let C be a clause 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^*. If we flip the value of one of these ℓ literals, the Hamming distance of x and x^* 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 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 $x \in B_0$; it may accept earlier when x equals some other satisfying assignment.



So 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 $i+1$. We want to know 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 string 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.

The initial assignment x has some Hamming distance to x^*. Clearly, this distance is binomially distributed, i.e.,

$$\text{prob}(\text{Hamming distance} = i) = \binom{n}{i} 2^{-n}.$$

For this (infinite) Markov chain it is known (see, for example, [GS92]) assuming $k \geq 3$:

- The probability[1] of reaching 0 from i. exponentially in i. More precisely,

$$\text{prob}(\text{absorbing state 0 is reached} \mid \text{process started in state } i) = (\frac{1}{k-1})^i.$$

- Walks starting in state i and reaching state 0 are relatively short[2]. More precisely,

$$\text{E}[\text{number of steps until state 0 is reached} \mid \begin{array}{l}\text{process started in state } i \text{ and the absorbing state 0 is reached}\end{array}] = C_k i,$$

  where $C_k = \frac{k^2-2k+2}{k(k-2)}$. WARNING: this is NOT the constant that is given in Schoening's article. Schoening states $C_k = \frac{k}{k-2} = 1 + \frac{2}{k-2}$ without proof. My calculation yields $C_k = 1 + \frac{2}{k(k-2)}$. WARNING

Combining these facts with the initial probability for state i, we obtain

[1] Let p be the probability of reaching state 0 from state 1. Such a walk has the following form: the last transition is from 1 to 0 (probability $1/k$). Before that it performs $t \geq 0$ loops, i..e, it goes from 1 to 2 and then returns with probability p back to 1 (probability $((k-1)/k)p)^t$. Thus

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

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

[2] Let L be the expected length of a walk from 1 to 0. Such a walk has the following form: the last transition is from 1 to 0 (length 1). Before that it performs exactly $t \geq 0$ loops, i..e, it goes from 1 to 2 and then returns back to 1 (probability $((k-1)/k)p)^t(1/k)$, where p is as in the previous item) and length $t(L+1)$. Thus

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

This solves to $L = \frac{k^2-2k+2}{k(k-2)}$. The expected length of a walk starting and i and reaching 0 is i times this number.

- for the probability of reaching state 0:

$$\begin{aligned}\text{prob(the absorbing state 0 is reached)} &= \sum_{0\le j\le n}\binom{n}{i}2^{-n}(1/(k-1))^i \\ &= 2^{-n}\left(1+\frac{1}{k-1}\right)^n = \left(\frac{k}{2(k-1)}\right)^n\end{aligned}$$

- and for the expected number of steps of a walk reaching state 0:

$$\text{E[\# of steps until 0 is reached} \mid \text{0 is reached]} = \sum_{i\ge 0} C_k i\binom{n}{i}2^{-n} = C_k\frac{n}{2^n}\sum_{i\ge 1}\binom{n}{i-1} = \frac{C_k n}{2}.$$

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

$$\text{prob(0 is reached in at most } C_k n \text{ steps} \mid \text{0 is reached)} \ge 1/2.$$

Combining this with the probability that 0 is reached, we obtain

$$\text{prob(after at most } C_k n \text{ steps the state 0 is reached)} \ge (1/2)(k/(2(k-1)))^n.$$

Let $p = (1/2)(k/(2(k-1)))^n$. The probability that x^* is reached from a random x is at least p. If we repeat the experiment T times, the probability that we never reach x^* is bounded by $(1-p)^T = \exp(T\ln(1-p)) \le \exp(-Tp)$ (recall that $\ln(1-p) \le -p$ for $0 \le p < 1$). In order to have error probability at most ε, it therefore suffices to choose T such that $\exp(-Tp) \le \varepsilon$, i.e.,

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

Theorem 1 (Schoening) *In time $O(2\left(\frac{2(k-1)}{k}\right)^n \ln\frac{1}{\varepsilon}\text{poly}(n,m))$ one can decide satisfiability of formulas n variables, m clauses, and at most k literals per clause with (one-sided) error ε.*

A derandomization with essentially the same time bound was obtained recently by Moser and Schreder [].

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

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

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 [ABCC06] for a detailed computational study of the TSP.

2 Approximation Algorithms

Approximation 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.

Problems vary widely in their approximability.

2.1 The Knapsack Problem: A PTAS via Scaling and Dynamic Programming

The Knapsack Problem is easily stated. We are given n items, , each with a value and a weight. Let v_i and w_i be the value and the weight of the i-th item. We are also given a weight bound W. The goal is to determine a subset $S \subseteq [n]$ of the items, maximizing the value $\sum_{i \in S} v_i$ and obeying the weight constraint $\sum_{i \in S} w_i \leq W$. We may assume $w_i \leq W$ for all i.

The Greedy Heuristic: We order the items by value per unit weight, i.e., by v/w. We start with the empty set S of items and then iterate over the items in decreasing order of value-per-unit-weight-ratio. If an item fits, i.e., the current weight plus the weight of the item does not exceed the weight bound, we add it, otherwise we discard it. Let V_g be the value computed.

The greedy heuristic is simple, but no good. Consider the following example. We have two items: the first one has value 1 and weight 1 and the second one has value 99 and weight 100. The weight bound is 100. The greedy heuristic considers the first item first (since 1/1 > 99/100) 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.

Return the maximum of V_g and $\max_i v_i$.

Lemma 1 *The modified greedy algorithm achieves a value of $V_{opt}/2$ and runs in time $O(n \log n)$.*

Proof: Order the items by decreasing ratio of value to weight, i.e., $v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n$. 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 $V_g \geq v_1 + \ldots + v_k$.

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

$$V_{opt} \leq v_1 + \ldots + v_k + v_{k+1} \leq V_g + \max_i v_i \leq 2\max(V_g, \max_i v_i).$$

The time bound is obvious. We sort the item according to decreasing ratio of value to weight and then iterate over the items. ∎

Dynamic Programming: We assume that the values are integers. We show how to compute an optimal solution in time $O(n^2 v_{max})$.

Clearly, the maximum value of the knapsack is bounded by nv_{max}. We fill a table $B[0, nv_{max}]$ such that, at the end, $B[s]$ is the minimum weight of a knapsack of value s for $0 \leq s \leq nv_{max}$. We fill the table in phases 1 to n and maintain the invariant that after the i-th 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 items, we can only obtain value 0, and we do so at a weight of zero. Consider now the i-th 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.

$B[0] = 0$ and $B[s] = \infty$ for $s \geq 1$.
for i from 1 to n do
  for $s = nv_{max}$ step -1 downto v_i do
  $B[s] = \min(B[s], B[s - v_i] + w_i)$
let s be maximal such that $B[s] \leq W$; return s.

Lemma 2 *The dynamic programming algorithm solves the knapsack problem in time $O(n^2 v_{max})$.*

Proof: Obvious. ∎

A small trick improves the running time to $O(nV_{opt})$. We maintain a value K which is the maximum value seen so far. We initialize K to zero and replace the inner loop by

  $K = K + v_i$
  for $s = K$ step -1 downto v_i do
  $B[s] = \min(B[s], B[s - v_i] + w_i)$

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.,

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

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

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

Thus

$$V_{approx} \ge (1 - \frac{nS}{V_{opt}}) V_{opt}.$$

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

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. We obtain.

Theorem 2 *Let $\varepsilon > 0$. In time $O(n^2/\varepsilon)$ one can compute a solution to the knapsack problem with $V_{approx} \ge (1-\varepsilon)V_{opt}$.*

2.2 Set Cover: A Greedy Algorithm

We are given subsets S_1 to S_n of some ground set U. Each set come with a cost $c(S_i) \ge 0$. The goal is to the cheapest way to cover U, i.e.,

$$\min \sum_{i \in I} c(S_i) \text{ subject to } \cup_{i \in I} S_i = U.$$

The greedy strategy applies naturally to the set cover problem: 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.

$C = \emptyset$
while $C \neq U$ do
  Let i minimize $c(S_i)/|S_i \setminus C|$.
  $C = C \cup S_i$
  set the price of every $e \in S_i \setminus C$ to $price(e) = c(S_i)/|S_i \setminus C|$.
output the sets picked.

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

Lemma 3 *For all k,* $price(e_k) \leq \text{OPT}/(n-k+1)$.

Proof: Consider the iteration in which e_k is covered. Clearly, the sets in the optimal solution cover the remaining elements for a cost of at most OPT, i.e., at a cost of $\text{OPT}/(|U \setminus C| \leq \text{OPT}/(n-k+1)$ per element. Thus there must be a set which covers elements at this cost or less. The algorithm picks the most cost-effective set and hence $price(e_k) \leq \text{OPT}/(n-k+1)$. ∎

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

Proof: By the preceding Lemma, the cost of the solution produced by the greedy algorithm is bounded by

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

∎

No better approximation ratio can be proved for this algorithm as the following example shows.

The ground set has 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.

Hardness of Approximation: It is unlikely that there is a better algorithm for set-cover than the greedy one.

Theorem 4 *If there is an approximation algorithm for set cover with approximation ratio* $o(\log n)$ *then* $\text{NP} \subseteq \text{ZTIME}(n^{O(\log\log n)})$.

$\text{ZTIME}(T(n))$ is the set of problems which have a Las Vegas algorithm with running time $T(n)$.

2.3 Maximum Satisfiability: A Randomized Algorithm

We consider formulae in 3-CNF. Max-3-SAT is the problem of satisfying a maximum number of clauses in a given formula Φ.

Lemma 4 *Let Φ be a formula in which every clause has exactly distinct three literals. There is a randomized algorithm that satisfies an expected number of 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 falsifies C. 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 the book by Vazirani or by Motwani and Raghavan.

A student pointed out to me that the algorithm above is easily derandomized. Let $\{C_1,\ldots,C_m\}$ be a set of clauses (not necessarily of length 3). Let I_i be an indicator variable for the i-th clause. It is one if C_i is satisfied and zero otherwise. Then

$$\mathrm{E}[\#\text{ of satisfied clauses}] = \mathrm{E}[I_1+\ldots+I_m] = \mathrm{E}[I_1]+\ldots+\mathrm{E}[I_m] = \sum_{1\le i\le m} \mathrm{prob}(C_i \text{ is true}).$$

A clause with k distinct literals is true with probability $1-2^{-k}$.

We can now use the method of conditional expections. Note that

$$\mathrm{E}[\#\text{ of satisfied clauses}] = (\mathrm{E}[\#\text{ of satisfied clauses} \mid x_1=0]+\mathrm{E}[\#\text{ of satisfied clauses} \mid x_1=1])/2.$$

And hence one of the two expections on the right is at least the expection on the left. Since we can compute the expections, we can select the better choice between $x_1=0$ and $x_1=1$. In this way, we find deterministically an assignment that has the same guarantee as the randomized algorithm.

What can one achieve deterministically? Consider the following two assignments. Set all variables to TRUE or set all variables to FALSE. Since any clause is satisfied by all but one of the assignments to its literals, every clause is satisfied by one of the assignments. Hence one assignment satisfies at least half of the clauses.

Karloff-Zwick have shown how to satisfy at 7/8 of the clauses in any satisfiable formula. This is not the same as a 7/8-approximation algorithm.

2.4 Vertex Cover

Consider the vertex cover problem. We are given a graph $G=(V,E)$. The goal is to find the smallest subset of the vertices that covers all the edges.

Lemma 5 *A factor 2-approximation to the optimum vertex cover can be computed in polynomial time.*

Proof: Compute a maximal matching M and output the endpoints of the edges in the matching. A maximal matching can be computed greedily. We iterate over the edges in arbitrary order. If an edge is uncovered we add it to the matching.

Clearly, the size of the cover is $2|M|$. An optimal cover must contain at least one endpoint of every edge in M. ∎

2.5 Hardness of Approximation

This section follows the book by Vazirani [Vaz03].

How does one show that certain problems do not have good approximation algorithms unless P = NP. The key are gap-introducing and gap-preserving reductions.

Assume now that we have a polynomial time reduction from SAT to MAX-3-SAT with the following property. It maps an instance Φ of SAT to a instance Ψ of MAX-3-SAT such that

- if Φ is satisfiable, Ψ is satisfiable.
- if Φ is not satisfiable, every assignment to the variables in Ψ satisfies a fraction less than $(1-\alpha)$ of the clauses of Ψ.

where $\alpha > 0$ is a constant.

Theorem 5 *In the situation above, there is no approximation algorithm for Max-3-SAT that achieves an approximation ratio of* $1-\alpha$*, unless* P = NP.

Proof: Assume otherwise. Let Φ be any formula and let Ψ be constructed by the reduction. Construct a maximizing assignment x for Ψ using the approximation algorithm and let m be the number of clauses in Ψ. If x satisfies at least $(1-\alpha)m$ clauses of Ψ, declare Φ satisfiable. Otherwise, declare Φ unsatisfiable. The correctness is almost immediate.

If Φ is satisfiable, there is a satisfying assignment for Ψ and hence the assignment produced by the approximation algorithm must satisfy at least $(1-\alpha)m$ clauses of Ψ. Thus the algorithm declares Φ satisfiable.

On the other hand, if Φ is not satisfiable, any assignment for the variables in Ψ satisfies less than $(1-\alpha)m$ clauses of Ψ. In particular, the assignment returned by the approximation algorithm cannot do better. Thus Ψ is declared unsatisfiable. ∎

.

2.6 The PCP-Theorem (Probabilistically Checkable Proofs)

We are all familiar with the characterization of NP via witnesses (proofs) and polynomial time verifiers. A language L belongs to NP if and only if there is a deterministic polynomial time Turing machine V (called the verifier) with the following property. On input x, V is provided with an additional input y, called the proof, such that:

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

In probabistically checkable proofs, we make the verifier a probabilistic machine (this makes the verifier stronger), but allow it to read only part of the proof in any particular run.

A language L belongs to PCP$(\log n, 1)$ if and only if there is polynomial time Turing machine V (called the verifier) and constants c and q with the following properties. On input x of length n, V is provided with x, a proof y, and a random string[3] r of length $c \log n$.

It performs a computation querying at most q bits of y. The q bits probed depend on x and r, but not[4] on y.

If $x \in L$, then there is a proof y that makes V accept with probability 1.

If $x \notin L$, then for every proof y, V accepts with probability less than $1/2$.

The probabilities are computed with respect to the random choices r.

Theorem 6 (Arora, Lund, Motwani, Sudan, and Szegedy) NP = PCP$(\log n, 1)$.

The direction PCP$(\log n, 1) \subseteq$ 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.

Consider 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 gives rise to an optimization problem which does not have a 1/2-approximation algorithm unless P = NP.

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

Theorem 7 *Maximize Accept Probability has no factor 1/2 approximation algorithm unless* P = NP.

Proof: Assume otherwise. Let Φ be any formula and let y be the proof returned by approximation alg. We run V with y and all random strings and compute the acceptance probability p. If p is less than $1/2$ we declare Φ non-satisfiable, If $p \geq 1/2$, we declare Φ satisfiable.

Why is this correct? If Φ is satisfiable, there is proof with acceptance probability 1. Hence the approximation alg 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 the approximation alg, we must have $p < 1/2$. ∎

[3] It is convenient to make V a deterministic Turing machine and to provide the randomness explicitly.

[4] This requirement is not essential. Assume we would allow the bits read to depend on y. The first bit inspected is independent of y, the second bit may depend on the value of the first bit, and so on. So there are only $1 + 2 + \ldots 2^{q-1}$ potential bits that can be read.

2.7 Max-3-SAT is hard to approximate

We give a gap-introducing reduction from SAT to MAX-3-SAT.

Let V be a PCP$(\log n, 1)$-verifier for SAT. On an input Φ of length n it uses a random string r of length $c \log n$ and inspects (at most) q positions of the proof.

Thus there are at most qn^c bits of the proof that are inspected for some random string. Let $B = B_1 \ldots B_{qn^c}$ be a bit-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 desired properties. If Φ is satisfiable, Ψ is satisfiable. 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 $B_{i(r,q)}$.

Let us pause and see what we have achieved.

(A) 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.

(B) If Φ is not satisfiable, every proof makes V accept with probability less than 1/2. Hence every B falsifies 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 falsifies f_r falsifies at least one clause 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

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

An assignment satisfying C can be extended to an assignment satisfying C'. Conversely, an assignment falsifying C falsifies at least one clause of C'.

What have we achieved? Ψ consists of no more than $n^c 2^q (q-2)$ clauses. Each clause has 3 literals.

(A) 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.

(B) If Φ is not satisfiable, every proof makes V accept with probability less than 1/2. Hence every B falsifies more than half of the functions f_r. Hence B falsifies more than $n^c/2$ clauses in Ψ.

Let $\alpha = (n^c/2)/(n^c 2^q (q-2)) = 1/((q-2)2^{q+1})$.

Theorem 8 *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. Hastad showed that the theorem holds for any $\alpha = 1/8 - \varepsilon$ for any $\varepsilon > 0$.

2.8 From Max-3-SAT to Vertex Cover

Step 1: One shows that Max-3-SAT stays hard to approximate if one introduces the additional constraint that each variable appears at most 30 times.

Step 2: From Max-3-SAT(30) to vertex cover. One uses the standard reduction.

2.9 Approximation Algs for Problems in P

3 General Search Heuristics

4 Parameterized Complexity

4.1 Parameterized Complexity for Problems in P

References

[ABCC06] D. Applegate, B. Bixby, V. Chvatal, and W. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[GS92] G. R. Grimmett and D. R. Stirzacker. *Probability and Random Processes*. Oxford University Press, 1992.

[NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[Sch99] U. Schöning. A probabilistic algorithm for k-sat and constraint satisfaction problems. FOCS, 1999.

[Sch01] Uwe Schöning. New algorithms for k-sat based on the local search principle. In *Mathematical Foundations of Computer Science 2001*, volume 2136 of *Lecture Notes in Computer Science*, pages 87–95. Springer Berlin Heidelberg, 2001.

[Vaz03] V. Vazirani. *Approximation Algorithms*. Springer, 2003.