18.415/6.854 Advanced Algorithms November 1994

Approximation Algorithms

Lecturer: Michel X. Goemans

1 Introduction

Many of the optimization problems we would like to solve are NP-hard. There are several ways of coping with this apparent hardness. For most problems, there are straightforward exhaustive search algorithms, and one could try to speed up such an algorithm. Techniques which can be used include divide-and-conquer (or the refined branch-and-bound which allows to eliminate part of the search tree by computing, at every node, bounds on the optimum value), dynamic programming (which sometimes leads to pseudo-polynomial algorithms), cutting plane algorithms (in which one tries to refine a linear programming relaxation to better match the convex hull of integer solutions), randomization, etc. Instead of trying to obtain an optimum solution, we could also settle for a suboptimal solution. The latter approach refers to *heuristic* or "rule of thumb" methods. The most widely used such methods involve some sort of local search of the problem space, yielding a locally optimal solution. In fact, heuristic methods can also be applied to polynomially solvable problems for which existing algorithms are not "efficient" enough. A $\Theta(n^{10})$ algorithm (or even a linear time algorithm with a constant of 10^{100}), although efficient from a complexity point of view, will probably never get implemented because of its inherent inefficiency.

The drawback with heuristic algorithms is that it is difficult to compare them. Which is better, which is worse? For this purpose, several kinds of analyses have been introduced.

1. **Empirical analysis**. Here the heuristic is tested on a bunch of (hopefully meaningful) instances, but there is no guarantee that the behavior of the heuristic on these instances will be "typical" (what does it mean to be typical?).

2. **Average-case analysis**, dealing with the average-case behavior of a heuristic over some distribution of instances. The difficulty with this approach is that it can be difficult to find a distribution that matches the real-life data an algorithm will face. Probabilistic analyses tend to be quite hard.

3. **Worst-case analysis**. Here, one tries to evaluate the performance of the heuristic on the worst possible instance. Although this may be overly pessimistic, it gives a stronger guarantee about an algorithm's behavior. This is the type of analysis we will be considering in these notes.

To this end, we introduce the following definition:

Definition 1 *The performance guarantee of a heuristic algorithm for a minimization (maximization) problem is α if the algorithm is guaranteed to deliver a solution whose value is at most (at least) α times the optimal value.*

Definition 2 *An α-approximation algorithm is a polynomial time algorithm with a performance guarantee of α.*

Before presenting techniques to design and analyze approximation algorithms as well as specific approximation algorithms, we should first consider which performance guarantees are unlikely to be achievable.

2 Negative Results

For some hard optimization problems, it is possible to show a limit on the performance guarantee achievable in polynomial-time (assuming $P \neq NP$). A standard method for proving results of this form is to show that the existence of an α-approximation algorithm would allow you to solve some NP-complete decision problem in polynomial time. Even though NP-complete problems have equivalent complexity when exact solutions are desired, the reductions don't necessarily preserve approximability. The class of NP-complete problems can be subdivided according to how well a problem can be approximated.

As a first example, for the traveling salesman problem (given nonnegative lengths on the edges of a complete graph, find a tour — a closed walk visiting every vertex exactly once — of minimum total length), there is no α-approximation for any α unless $P = NP$. Indeed such an algorithm could be used to decide whether a graph (V, E) has an Hamiltonian cycle (simply give every edge e a length of 1 and every non-edge a very high or infinite length).

As another example, consider the bin packing problem. You have an integer T and weights $x_1, \dots, x_n \in [0, T]$, and you want to partition them into as few sets ("bins") as possible such that the sum of the weights in each set is at most T. It is NP-complete to decide whether k bins are sufficient.

In fact, there is no α-approximation algorithm for this problem, for any $\alpha < 3/2$. To see this, consider the partition problem: given weights $x_1, \dots, x_n \in [0, S]$ whose total sum is $2S$, is there a *partition* of the weights into two sets such that the sum in each set is S? This is the same as asking: are two bins sufficient when each bin has capacity S? If we had an α-approximation algorithm ($\alpha < 3/2$), we could solve the partition problem[1]. In general, if the problem of deciding whether a value is at most k is NP-complete then there is no α-approximation algorithm with $\alpha < \frac{k+1}{k}$ for the problem of minimizing the value unless $P = NP$.

[1]But wait, you exclaim — isn't there a polynomial-time approximation scheme for the bin packing problem? In fact, very good approximation algorithms can be obtained for this problem if you allow additive as well as multiplicative constants in your performance guarantee. It is our more restrictive model that makes this negative result possible. See Section 6 for more details.

Until 1992, pedestrian arguments such as this provided essentially the only known examples of non-approximability results. Then came a string of papers culminating in the result of Arora, Lund, Motwani, Sudan, and Szegedy[1] (based on the work of Arora and Safra [2]). They introduced a new characterization of NP in terms of probabilistically checkable proofs (PCP). In the new characterization, for any language L in NP, given the input x and a "proof" y of polynomial length in x, the verifier will toss $O(\log n)$ coins (where n is the size of x) to determine $k = O(1)$ positions or bits in the string y to probe; based on the values of these k bits, the verifier will answer "yes" or "no". The new characterization shows the existence of such a verifier V and a proof y such that (i) if $x \in L$ then there exists a proof y such that V outputs "yes" independently of the random bits, (ii) if $x \notin L$ then for every proof y, V outputs "no" with probability at least 0.5.

From this characterization, they deduce the following negative result for MAX 3SAT: given a set of clauses with at most 3 literals per clause, find an assignment maximizing the number of satisfied clauses. They showed the following:

Theorem 1 *For some $\epsilon > 0$, there is no $1 - \epsilon$-approximation algorithm*[2] *for MAX 3SAT unless $P = NP$.*

The proof goes as follows. Take any NP-complete language L. Consider the verifier V given by the characterization of Arora et al. The number of possible output of the $O(\log n)$ toin cosses is $S = 2^{O(\log n)}$ which is polynomial in n. Consider any outcome of these coin tosses. This gives k bits, say $i_1, \ldots, i_k$ to examine in the proof y. Based on these k bits, V will decide whether to answer yes or no. The condition that it answers yes can be expressed as a boolean formula on these k bits (with the Boolean variables being the bits of y). This formula can be expressed as the disjunction ("or") of conjunctions ("and") of k literals, one for each satisfying assignment. Equivalently, it can be written as the conjunction of disjunction of k literals (one for each rejecting assignment). Since k is $O(1)$, this latter k-SAT formula with at most 2^k clauses can be expressed as a 3-SAT formula with a constant number of clauses and variables (depending exponentially on k). (More precisely, using the classical reduction from SAT to 3-SAT, we would get a 3-SAT formula with at most $k2^k$ clauses and variables.) Call this constant number of clauses $M \leq k2^k = O(1)$. If $x \in L$, we know that there exists a proof y such that all SM clauses obtained by concatenating the clauses for each random outcome is satisfiable. However, if $x \notin L$, for any y, the clauses corresponding to at least half the possible random outcomes cannot be all satisfied. This means that if $x \notin L$, at least $S/2$ clauses cannot be satisfied. Thus either all SM clauses can be satisfied or at most $SM - \frac{S}{2}$ clauses can be satisfied. If we had an approximation algorithm with performance guarantee better than $1 - \epsilon$ where $\epsilon = \frac{1}{2M}$ we could decide whether $x \in L$ or not, in polynomial

[2]In our definition of approximation algorithm, the performance guarantee is less than 1 for *maximization* problems.

time (since our construction can be carried out in polynomial time). This proves the theorem.

The above theorem implies a host of negative results, by considering the complexity class MAX-SNP. defined by Papadimitriou and Yannakakis [21].

Corollary 2 *For any MAX-SNP-complete problem, there is an absolute constant* $\epsilon > 0$ *such that there is no* $(1-\epsilon)$*-approximation algorithm unless* $P = NP$.

The class of MAX-SNP problems is defined in the next section and the corollary is derived there. We first give some examples of problems that are complete for MAX-SNP.

1. MAX 2-SAT: Given a set of clauses with one or two literals each, find an assignment that maximizes the number of satisfied clauses.

2. MAX k-SAT: Same as MAX 2-SAT, but each clause has up to k literals.

3. MAX CUT: Find a subset of the vertices of a graph that maximizes the number of edges crossing the associated cut.

4. The Travelling Salesman Problem with the triangle inequality is MAX-SNP-hard. (There is a technical snag here: MAX-SNP contains only maximization problems, whereas TSP is a minimization problem.)

2.1 MAX-SNP Complete Problems

Let's consider an alternative definition of NP due to Fagin [9]. NP, instead of being defined computationally, is defined as a set of predicates or functions on structures G:

$$\exists S \forall x \exists y \; \psi(x, y, G, S)$$

where ψ is a quantifier free expression. Here S corresponds to the witness or the proof.

Consider for example the problem SAT. We are given a set of clauses, where each clause is the disjunction of literals. (A literal is a variable or its negation.) We want to know if there is a way to set the variables true or false, such that every clause is true. Thus here G is the set of clauses, S is the set of literals to be set to true, x represents the clauses, y represents the literals, and

$$\psi(x, y, G, S) = (P(G, y, x) \wedge y \in S) \vee (N(G, y, x) \wedge y \notin S)$$

Where $P(G, y, x)$ is true iff y appears positively in clause x, and $N(G, y, x)$ is true iff y appears negated in clause x.

Strict NP is the set of problems in NP that can be defined without the the third quantifier:

$$\exists S \forall x \; \psi(x, G, S)$$

where ψ is quantifier free.

An example is 3-SAT, the version of SAT in which every clause has at most 3 literals. Here $x = (x_1, x_2, x_3)$ (all possible combinations of three variables) and G is the set of possible clauses; for example $(x_1 \vee x_2 \vee x_3)$, $(\overline{x_1} \vee x_2 \vee \overline{x_3})$, and so forth. Then ψ is a huge conjunction of statements of the form: If (x_1, x_2, x_3) appears as $(\overline{x_1} \vee x_2 \vee \overline{x_3})$, then $x_1 \notin S \vee x_2 \in S \vee x_3 \notin S$.

Instead of asking that for each x we get $\psi(x, G, S)$, we can ask that the number of x's for which $\psi(x, G, S)$ is true be maximized:

$$\max_S |\{x : \psi(x, G, S)\}|$$

In this way, we can derive an optimization problem from an SNP predicate. These maximization problems comprise the class MAX-SNP (MAXimization, Strict NP) defined by Papadimitriou and Yannakakis [21]. Thus, MAX 3SAT is in MAX-SNP.

Papadimitriou and Yannakakis then introduce an *L-reduction* (L for linear), which preserverses approximability. In particular, if P L-reduces to P', and there exists an α-approximation algorithm for P', then there exists a $\gamma\alpha$-approximation algorithm for P, where γ is some constant depending on the reduction.

Given L-reductions, we can define MAX-SNP complete problems to be those $P \in$ MAX-SNP for which $Q \leq_L P$ for all $Q \in$ MAX-SNP. Some examples of MAX-SNP complete problems are MAX 3SAT, MAX 2SAT (and in fact MAX kSAT for any fixed $k > 1$), and MAX-CUT. The fact that MAX 3SAT is MAX-SNP-complete and Theorem 1 implies the corollary mentioned previously.

For MAX 3SAT, ε in the statement of Theorem 1 can be chosen can be set to 1/74 (Bellare and Sudan [5]).

Minimization problems may not be able to be expressed so that they are in MAX-SNP, but they can still be MAX-SNP hard. Examples of such problems are:

- TSP with edge weights 1 and 2 (i.e., $d(i, j) \in \{1, 2\}$ for all i, j). In this case, there exists a 7/6-approximation algorithm due to Papadimitriou and Yannakakis.

- Steiner tree with edge weights 1 and 2.

- Minimum Vertex Cover. (Given a graph $G = (V, E)$, a vertex cover is a set $S \subseteq V$ such that $(u, v) \in E \Rightarrow u \in S$ or $v \in S$.)

3 The Design of Approximation Algorithms

We now look at key ideas in the design and analysis of approximation algorithms. We will concentrate on minimization problems, but the ideas apply equally well to maximization problems. Since we are interested in the minimization case, we know that an α-approximation algorithm H has cost $C_H \leq \alpha C_{OPT}$ where C_{OPT} is the cost of the optimal solution, and $\alpha \geq 1$.

Relating C_H to C_{OPT} directly can be difficult. One reason is that for NP-hard problems, the optimum solution is not well characterized. So instead we can relate the two in two steps:

1. $LB \leq C_{OPT}$
2. $C_H \leq \alpha LB$

Here LB is a lower bound on the optimal solution.

3.1 Relating to Optimum Directly

This is not always necessary, however. One algorithm whose solution is easy to relate directly to the optimal solution is Christofides' [6] algorithm for the TSP with the triangle inequality ($d(i,j) + d(j,k) \leq d(i,k)$ for all i, j, k). This is a $\frac{3}{2}$-approximation algorithm, and is the best known for this problem. The algorithm is as follows:

1. Compute the minimum spanning tree T of the graph $G = (V, E)$.
2. Let O be the odd degree vertices in T. One can prove that $|O|$ is even.
3. Compute a minimum cost perfect matching M on the graph induced by O.
4. Add the edges in M to E. Now the degree of every vertex of G is even. Therefore G has an Eulerian tour. Trace the tour, and take shortcuts when the same vertex is reached twice. This cannot increase the cost since the triangle inequality holds.

We claim that $Z_C \leq \frac{3}{2} Z_{TSP}$, where Z_C is the cost of the tour produced by Christofides' algorithm, and Z_{TSP} is the cost of the optimal solution. The proof is easy:

$$\frac{Z_C}{Z_{TSP}} \leq \frac{Z_T + Z_M}{Z_{TSP}} = \frac{Z_T}{Z_{TSP}} + \frac{Z_M}{Z_{TSP}} \leq 1 + \frac{1}{2} = \frac{3}{2}.$$

Here Z_T is the cost of the minimum spanning tree and Z_M is the cost of the matching. Clearly $Z_T \leq Z_{TSP}$, since if we delete an edge of the optimal tour a spanning tree results, and the cost of the minimum spanning tree is at most the cost of that tree. Therefore $\frac{Z_T}{Z_{TSP}} \leq 1$.

To show $\frac{Z_M}{Z_{TSP}} \leq \frac{1}{2}$, consider the optimal tour visiting only the vertices in O. Clearly by the triangle inequality this is of length no more than Z_{TSP}. There are an even number of vertices in this tour, and so also an even number of edges, and the tour defines two disjoint matchings on the graph induced by O. At least one of these has cost $\leq \frac{1}{2}Z_{TSP}$, and the cost of Z_M is no more than this.

3.2 Using Lower Bounds

Let

$$C_{OPT} = \min_{x \in S} f(x).$$

A lower bound on C_{OPT} can be obtained by a so-called *relaxation.* Consider a related optimization problem $LB = \min_{x \in R} g(x)$. Then LB is a lower bound on C_{OPT} (and the optimization problem is called a relaxation of the original problem) if the following conditions hold:

(1) $$S \subseteq R$$

(2) $$\mathrm{g}(x) \leq f(x) \text{ for all } x \in S.$$

Indeed these conditions imply

$$LB = \min_{x \in R} g(x) \leq \min_{x \in S} f(x) = C_{OPT}.$$

Most classical relaxations are obtained by using linear programming. However, there are limitations as to how good an approximation LP can produce. We next show how to use a linear programming relaxation to get a 2-approximation algorithm for Vertex Cover, and show that this particular LP relaxation cannot give a better approximation algorithm.

3.3 An LP Relaxation for Minimum Weight Vertex Cover (VC)

A vertex cover U in a graph $G = (V, E)$ is a subset of vertices such that every edge is incident to at least one vertex in U. The vertex cover problem is defined as follows: Given a graph $G = (V, E)$ and weight $w(v) \geq 0$ for each vertex v, find a vertex cover $U \subseteq V$ minimizing $w(U) = \sum_{v \in U} w(v)$. (Note that the problem in which nonpositive weight vertices are allowed can be handled by including all such vertices in the cover, deleting them and the incident edges, and finding a minimum weight cover of the remaining graph. Although this reduction preserves optimality, it does not maintain approximability; consider, for example, the case in which the optimum vertex cover has 0 cost (or even negative cost).)

This can be expressed as an integer program as follows. Let $x(v) = 1$ if $v \in U$ and $x(v) = 0$ otherwise. Then

$$C_{OPT} = \min_{x \in S} \sum_{v \in V} w(v)x(v)$$

where

$$S = \left\{ x \in R^{|V|} : \begin{array}{ll} x(v) + x(w) \geq 1 & \forall (v,w) \in E \\ x(v) \in \{0,1\} & \forall v \in V \end{array} \right\}.$$

We now relax S, turning the problem into a linear program:

$$LB = \min_{x \in R} \sum_{v \in V} w(v)x(v)$$

$$R = \left\{ x \in R^{|V|} : \begin{array}{ll} x(v) + x(w) \geq 1 & \forall (v,w) \in E \\ x(v) \geq 0 & \forall v \in V \end{array} \right\}.$$

In order to show that R is a relaxation, we must show that it satisfies conditions 1 and 2. Condition 1 clearly holds, as $0, 1 \geq 0$. Furthermore, condition 2 also holds, since the objective function is unchanged. Thus, we can conclude that $LB \leq C_{OPT}$, and we can prove that an algorithm H is an α-approximation algorithm for VC by showing $C_H \leq \alpha LB$.

The limitation of this relaxation is that there are instances where $LB \sim \frac{1}{2}C_{OPT}$. This implies that it cannot be used to show any $\alpha < 2$, since if we could then H would give a better answer than the optimum. One such instance is K_n: the complete graph on n vertices, with all vertices weight 1. All the nodes but one must be in the cover (otherwise there will be an edge between two that are not, with neither in the cover set). Thus, $C_{OPT} = n - 1$. The relaxation, on the other hand, can have $x(v) = \frac{1}{2}, \forall v \in V$. Thus, $LB \leq \frac{n}{2}$, which means $LB \sim \frac{1}{2}C_{OPT}$.

3.4 How to use Relaxations

There are two main techniques to derive an approximately optimal solution from a solution to the relaxed problem.

1. **Rounding**
   Find an optimal solution x^* to the relaxation. Round $x^* \in R$ to an element $x' \in S$. Then prove $f(x') \leq \alpha g(x^*)$ which implies

   $$f(x') \leq \alpha LB \leq \alpha C_{OPT}$$

   Often randomization is helpful, as we shall see in later sections. In this case $x^* \in R$ is randomly rounded to some element $x' \in S$ so that $E[f(x')] \leq \alpha g(x^*)$. These algorithms can sometimes be derandomized, in which case one finds an x'' such that $f(x'') \leq E[f(x')]$.

2. **Primal-Dual**
   Consider some weak dual of the relaxation:
   $$\max\{h(y) : y \in D\} \leq \min\{g(x) : x \in R\}$$
   Construct $x \in S$ from $y \in D$ such that
   $$f(x) \leq \alpha h(y) \leq \alpha h(y_{\max}) \leq \alpha g(x_{\min}) \leq \alpha C_{OPT}.$$
   Notice that y can be any element of D, not necessarily an optimal solution to the dual.

We now illustrate these techniques on the minimum weight vertex cover problem.

3.4.1 Rounding applied to VC

This is due to Hochbaum [16]. Let x^* be the optimal solution of the LP relaxation. Let
$$U = \{v \in V : x^*(v) \geq \frac{1}{2}\}$$
We claim U is a 2-approximation of the minimum weight VC. Clearly U is a vertex cover, because for $(u,v) \in E$ we have $x^*(u) + x^*(v) \geq 1$, which implies $x^*(u) \geq 1/2$ or $x^*(v) \geq 1/2$. Also
$$\sum_{v \in U} w(v) \leq \sum_{v \in V} w(v) 2x^*(v) = 2LB$$
since $2x^*(v) \geq 1$ for all $v \in U$.

3.4.2 Primal-Dual applied to VC

This is due to Bar-Yehuda and Even [4]. First formulate the dual problem. Let $y \in R^{|E|}$; the elements of y are $y(e)$ for $e = (u,v) \in E$. The dual is:
$$\max \sum_{e \in E} y(e)$$
$$\sum_{u : e=(v,u) \in E} y(e) \leq w(v) \quad \forall v \in V \tag{3}$$
$$y(e) \geq 0 \quad \forall e \in E. \tag{4}$$

Initialize C (the vertex cover) to the empty set, $y = 0$ and $F = E$. The algorithm proceeds by repeating the following two steps while $F \neq \emptyset$:

1. Choose some $e = (u,v) \in F$. Increase $y(e)$ as much as possible, until inequality (3) becomes tight for u or v. Assume WLOG it is tight for u.

2. Add u to C and remove all edges incident to u from F.

Clearly C is a vertex cover. Furthermore
$$\sum_{v \in C} w(v) = \sum_{v \in C} \sum_{u : e=(v,u) \in E} y(e) = \sum_{e=(v,u) \in E} |C \cap \{v,u\}| y(e) \leq \sum_{e \in E} 2y(e) \leq 2LB.$$

4 The Min-Cost Perfect Matching Problem

In this section, we illustrate the power of the primal-dual technique to derive approximation algorithms. We consider the following problem.

Definition 3 *The* Minimum-Cost Perfect Matching Problem (MCPMP) *is as follows: Given a complete graph $G = (V, E)$ with $|V|$ even and a nonnegative cost function $c_e \geq 0$ on the edges $e \in E$, find a perfect matching M such that the cost $c(M)$ is minimized, where $c(M) = \sum_{e \in M} c_e$.*

The first polynomial time algorithm for this problem was given by Edmonds [8] and has a running time of $O(n^4)$ where $n = |V|$. To date, the fastest strongly polynomial time algorithm is due to Gabow [10] and has a running time of $O(n(m + n \lg n))$ where $m = |E|$. For dense graphs, $m = \Theta(n^2)$, this algorithm gives a running time of $O(n^3)$. The best weakly polynomial algorithm is due to Gabow and Tarjan [12] and runs in time $O(m\sqrt{n\alpha(m,n)\log n}\log nC)$ where C is a bound on the costs c_e. For dense graphs with $C = O(n)$, this bound gives an $O^*(n^{2.5})$ running time.

As you might suspect from these bounds, the algorithms involved are fairly complicated. Also, these algorithms are too slow for many of the instances of the problem that arise in practice. In this section, we discuss an approximation algorithm by Goemans and Williamson [13] that runs in time $O(n^2 \lg n)$. (This bound has recently been improved by Gabow, Goemans and Williamson [11] to $O(n(n + \sqrt{m \lg \lg n}))$.) Although MCPMP itself is in PTIME, this algorithm is sufficiently general to give approximations for many NP-hard problems as well.

The algorithm of Goemans and Williamson is a 2-approximation algorithm — it outputs a perfect matching with cost not more than a factor of 2 larger than the cost of a minimum-cost perfect matching. This algorithm requires that the costs c_e make up a metric, that is, c_e must respect the triangle inequality: $c_{ij} + c_{jk} \geq c_{ik}$ for all triples i, j, k of vertices.

4.1 A linear programming formulation

The basic idea used in the 2-approximation algorithm of Goemans and Williamson is linear programming and duality. The min-cost perfect matching problem can be formulated as a linear program. The algorithm does not directly solve the linear program, but during its operation, it can compute a feasible solution to the dual program. This dual feasible solution actually certifies the factor of 2 approximation. Before writing down the linear program, we start with an observation.

Consider a matching M and a set $S \subset V$ of vertices with $|S|$ odd. If M is a perfect matching, then since $|S|$ is odd, there must be some edge in the matching that has one endpoint inside S and the other outside. In other symbols, let $\delta(S)$ be the set of edges in E with exactly one endpoint in S; if M is a perfect matching and $|S|$ is odd, then $M \cap \delta(S) \neq \emptyset$.