Randomized algorithms: Two examples and Yao's Minimax Principle

Maximum Satisfiability

Consider the problem MAXIMUM SATISFIABILITY (MAX-SAT). Bring your knowledge up-to-date on the SATISFIABILITY problem.

MAXIMUM SATISFIABILITY:
Given a set of clauses, set each of the variables to TRUE or FALSE such as to maximize the number of clauses that become TRUE, i.e. of which at least one of the literals is true.

There is a trivial reduction from SATISFIABILITY to MAXIMUM SATISFIABILITY, proving NP-hardness of MAX-SAT. In this course I assume everyone to have basic knowledge on complexity theory (P, NP, and NP-Completeness (NP-hardness)). Those who need can make themselves up-to-date by reading e.g. the first few chapters of Garey and Johnson's *Computers and Intractability: a guide to the theory of NP-Completeness* or any other basic theoretical computer science book on algorithms.

Consider the following extremely simple randomised algorithm (RA) for MAX-SAT: Set each of the variables independently to TRUE with probability 1/2.

Theorem 0.1. *RA gives a solution with an expected number of* TRUE *clauses greater than or equal to* 1/2 *times the optimal number of* TRUE *clauses.*

Proof. Suppose we have a Boolean expression (instance of MAX-SAT) with m clauses. For $i = 1, \ldots, m$ let $Z_i = 1$ denote the event that clause C_i is TRUE, and $Z_i = 0$ otherwise. If C_i contains k literals then the probability that none of them is TRUE is 2^{-k}, by independence of the random TRUE-FALSE assignment. Thus, $Pr\{Z_i = 1\} = 1 - 2^{-k} \geq 1/2$. Hence, $E[Z_i] \geq 1/2$. This holds for all i. The expected number of TRUE clauses is $E[\sum_{i=1}^{m} Z_i]$. Using the useful linearity property of expectations gives

$$E[\sum_{i=1}^{m} Z_i] = \sum_{i=1}^{m} E[Z_i] \geq \frac{1}{2}m.$$

Since m is clearly an upper bound on the optimal number of TRUE clauses, this proves the theorem. □ □

The example shows the power of randomisation. It is a simple algorithm that performs reasonably well. It also reveals the interpretation of a randomised algorithm as a probability distribution over the class of deterministic algorithms: in this case, denoting by n the number of variables, it gives probability 2^{-n} to each of the 2^n deterministic algorithms that blindly set variables to TRUE or

FALSE and probability 0 to any other algorithm. Notice that for these blind algorithms it is easy to construct a Boolean expression on n variables of which none of the clauses will be TRUE. Randomisation avoids the bad performance of the deterministic algorithms that receive a positive probability of being chosen.

The MAX-SAT example was concerned with approximation. What if we would be interested in the exact answer: what is the probability that RA produces an optimal solution. With respect to this question RA performs very badly: if there are c optimal solutions then RA will produce any of them with probability $c2^{-n}$. We will find out that we cannot expect an efficient randomised algorithm to perform essentially better for MAX-SAT.

Minimum Cut in a graph

A cut in a graph is a set of edges whose removal splits the graph into two or more components. We are interested in the minimum cardinality cut. We know the problem is well solved through solving a maximum flow problem on it. But let us study a much simpler randomised algorithm.

RA: Select from the graph $G = (V, E)$ uniformly at random an edge and contract it. Contracting an edge means identifying its two end vertices. If the two vertices incident to an edge have a common neighbour then contraction of it creates parallel edges. We keep the parallel (multiple) edges. We repeat the procedure iteratively to the newly created graph. At each step the number of vertices of the graph is reduced by 1. We continue until only two vertices are left.

Notice the a cut in the graph after one contraction is also a cut in the graph before the contraction. Hence, the multiple edges between the two vertices remaining at the end of the algorithm correspond to a cut in the original graph.

We study the probability that this simple randomised algorithm finds a minimum cut. Let C be an optimal cut and let it have size k. Notice that C will not be found if any of its edges is contracted (selected) by the algorithm. So we should study the probability that none of the k edges of C are contracted by RA.

First consider the first iteration of RA. First observation: each edge of E is selected with probability $1/|E|$. Second observation: None of the vertices has degree $< k$, otherwise such a vertex would induce a cut of size smaller than k and therefore C could not be optimal. Therefore, $|E| \geq nk/2$. Hence,

$$Pr\{C \text{ does not survive 1st step}\} \leq \frac{k}{nk/2} = \frac{2}{n}.$$

Therefore,

$$Pr\{C \text{ survives 1st step}\} \geq 1 - \frac{2}{n}.$$

Now consider the second iteration:

$$
\begin{aligned}
&Pr\{C \text{ survives 2nd step}\} = \\
&\quad Pr\{C \text{ survives 2nd step} \mid C \text{ survived 1st step}\} Pr\{C \text{ survives 1st step}\} + \\
&\qquad Pr\{C \text{ survives 2nd step} \mid C \text{ has not survived 1st step}\} Pr\{C \text{ does not survive 1st step}\} = \quad (1) \\
&\quad Pr\{C \text{ survives 2nd step} \mid C \text{ survived 1st step}\} Pr\{C \text{ survives 1st step}\}
\end{aligned}
$$

Again we will bound the probability that C does not survive the 2nd step given that it has survived the 1st step. We have now $n-1$ vertices and, by the same argument as before, each of these has degree at least k, whence the number of edges is at least $(n-1)k/2$. Thus,

$$Pr\{C \text{ does not survive 2nd step} \mid C \text{ survived 1st step} \leq \frac{k}{(n-1)k/2} = \frac{2}{n-1}$$

and therefore,

$$Pr\{C \text{ survives 2nd step} \mid C \text{ survived 1st step}\} \geq 1 - \frac{2}{n-1}.$$

Inserting into (1) yields

$$Pr\{C \text{ survives 2nd step}\} \geq (1 - \frac{2}{n-1})(1 - \frac{2}{n}).$$

Similarly, by induction we can prove that

$$Pr\{C \text{ survives } h\text{−th step}\} \geq \Pi_{i=1}^{h}(1 - \frac{2}{n-(i-1)})$$

and in particular, after $n-2$ steps a graph on 2 vertices survives:

$$Pr\{C \text{ survives}\} = \Pi_{i=1}^{n-2}(1 - \frac{2}{n-i+1}) = \Pi_{i=1}^{n-2}\frac{n-i-1}{n-i+1} = \frac{2}{n(n-1)} > \frac{2}{n^2}.$$

Thus, the algorithm gives an error in declaring that the cut found is a min cut with probability at most $1 - \frac{2}{n^2}$. Repeating the algorithm $n^2/2$ times brings the probability of not finding a min cut down to

$$(1 - \frac{2}{n^2})^{n^2/2} = \frac{1}{e}.$$

This can even be decreased further to any arbitrarily small probability ϵ by a number of repetitions of the algorithm which is polynomial both in n and in $1/\epsilon$.

The algorithm is an example of a Monte Carlo algorithm. These are algorithms with a so-called one-sided probability of failure. If the algorithm were to answer the question if in a graph a cut of size at most k exists, then if the algorithm outputs a yes-answer, then it is certainly correct, since it will have found a cut of size at most k. But if it outputs a no-answer then it could be that it failed to find a min cut. Thus a no-answer may not be correct.

Related to such Monte Carlo algorithms is the definition of a randomised complexity class, the class RP. For this class we introduce the elementary computer operation of a fair coin-flip. Equivalently we may assume that it takes one operation to draw an element from a set of size polynomially bounded in the size of the input.

RP is the class of decision problems Π for which a randomised algorithm A exists that runs in polynomial time (deterministically), such that for any instance π of Π the following holds:
a) If π has a yes-answer $\Rightarrow Pr\{A(\pi) \text{ gives yes} - \text{answer}\} \geq 1/2$;
b) If π has a no-answer $\Rightarrow Pr\{A(\pi) \text{ gives no} - \text{answer}\} = 1$.

In this definition $A(\pi)$ is the outcome of applying A to π.

Clearly P$\subset$RP$\subset$NP. The last one follows from the fact that for a yes-answer instance of a problem in NP, a single certificate exists on which an algorithm verifies in polynomial time the yes-answer. Since for problem in RP there exist for each yes-answer a probability at least 1/2 of obtaining a yes-answer by a polynomial time algorithm, then at least one running (realisation) of the randomised algorithm must lead to a yes-answer.

In the definition the probability 1/2 is chosen arbitrarily. As we mentioned with the min cut algorithm repeating the algorithm several times can bring the probability of failure down to any arbitrarily small constant as long as the probability of failure of a single run is bounded by $1/poly(|\pi|)$, the reciprocal of a number that is bounded by a polynomial function of the input size.

Many open questions remain with respect to this class in the spirit of the NP=P? question. One of them is RP$\subset$NP$\cap$co-NP? For other related questions and some more related complexity classes based on randomised algorithms I refer to the last part of Chapter 1 of the book.

Our conclusion for the Min Cut problem based on the analysis of the simple randomised algorithm is that the problem is in RP. This is no news, since we already knew it belonged to P.

Yao's Minimax Principle

We will explore a game theory approach to the performance analysis of randomised algorithms which will lead to a particularly useful tool for establishing lower bounds on the performance of randomised algorithms.

To start, think of the world-wide known problem Paper-Scissors-Stone. Call the two players R (for the row-player) and C (for the column-player). We define the pay-off matrix M of the game by letting M_{ij} denote the amount that C pays to R if R plays strategy i and C plays strategy j. Thus in Paper-Scissors-Stone the pay-off matrix becomes:

	Paper	Scissors	Stone
Paper	0	-1	1
Scissors	1	0	-1
Stone	-1	1	0

Clearly R likes to maximise what C needs to pay to her: he seeks

$$V_R = \max_i \min_j M_{ij},$$

whereas C seeks to minimize what she needs to pay to R:

$$V_C = \min_j \max_i M_{ij}.$$

In Paper-Scissors-Stone $V_R = -1$ and $V_C = 1$. This game is said to have no solution. The main approach to get around this problem is introducing randomisation. In game theory deterministic strategies are called *pure* strategies and randomised strategies are called *mixed* strategies. A mixed strategy for R is a vector p of probabilities p_i with which R plays strategy i, $i = 1, \ldots, m$, corresponding to the rows of M. Similarly, $q = (q_1, \ldots, q_n)$ defines a randomised strategy for C, corresponding to the columns of M. The pay-off becomes now a random variable with expected value

$$p^T Mq.$$

Again R tries to maximise its expected pay-off and C tries to minimise its expected payments:

$$W_R = \max_p \min_q p^T Mq;$$

$$W_C = \min_q \max_p p^T Mq.$$

Von Neumann showed that for any two-person zero-sum game $W_R = W_C$. (In a zero-sum game the money remains among the players.) Nowadays we can show this easily by strong LP-duality.

Let us consider $\max_p \min_q p^T Mq$ more closely. Once player R chooses her strategy p, then player C will choose q such as to minimise $p^T Mq$, which is written out $\sum_{j=1}^n (p^T M_j) q_j$, where we use M_j to denote the j-th column of M. But given p, $p^T M_j$ is a fixed number, say a_j. Clearly, minimising $\sum_j a_j q_j$ subject to $q_j \geq 0$ and $\sum_j q_j = 1$ is solved by selecting $\min_j a_j$ and setting the corresponding coordinate of q to 1 and all the others to 0. Thus, given p the best strategy for C is a deterministic strategy which we represent by the unit-vector e_j having 1 at coordinate j and 0's elsewhere:

$$\max_p \min_q p^T M q = \max_p \min_j p^T M e_j.$$

Similarly we can argue that

$$\min_q \max_p p^T M q = \min_q \max_i e_i^T M q.$$

Van Neumann now implies Loomis' Theorem:

$$\max_p \min_j p^T M e_j = \min_q \max_i e_i^T M q.$$

As a result $\min_j p^T M e_j \leq \max_i e_i^T M q$ for every pair of strategies p, q.

Yao discovered the usefulness of these game theoretical results for studying the performance of randomised algorithms. Think of solving a problem as a two-person zero-sum game in which C is the player who plays (chooses) an algorithm and R is the player who plays an instance of the problem. The pay-off matrix can be any performance measure we are interested in, such as the ratio between the solution value obtained by the algorithm played by C and the optimal value on an instance played by R. Or it could be the running time of an algorithm played by C on an instance played by R.

Define $\mathcal{A}$ as the class of all possible (deterministic) algorithms that C could choose from (for instance all polynomial time algorithms) and $\mathcal{I}$ as the class of all possible instances of the problem that R could choose from. Let $M(I, A)$ be the desired performance measure of algorithm A when applied to instance I.

A randomised strategy I_p for R is nothing more than selecting a random problem instance. Therefore $E[M(I_p, A)]$ is the expected performance measure of algorithm A on random instance I_p. Analysis of this expectation is known in the literature under the name *probabilistic analysis of algorithms.*

On the other hand a random strategy A_q for C is nothing more than a randomised algorithm. Therefore $E[M(I, A_q)]$ is the expected performance measure of randomised algorithm A_q on instance I. And $\max_{I \in \mathcal{I}} E[M(I, A_q)]$ is the worst-case expected performance measure of randomised algorithm A_q.

Remembering the corollary of Loomis' Theorem we have

$$\min_{A \in \mathcal{A}} E[M(I_p, A)] \leq \max_{I \in \mathcal{I}} E[M(I, A_q)].$$

In words: the expected performance of the best deterministic algorithm on any random instance I_p is a lower bound on the worst-case expected performance of any randomised algorithm. This is what is called Yao's Minimax Principle.

Thus to get a lower bound on the worst-case performance of any randomised algorithm it suffices to specify a random instance of the problem and analyse what any deterministic algorithm can achieve on this instance. Of course, better bounds are obtained by choosing more clever random instances. In fact, we know that there must be a random instance on which the best deterministic algorithm will give exactly the best attainable worst-case performance of any randomised algorithm, but it may not always be easy to detect that random instance.

Material

During the first lecture I treated from [MR]
Section 1.1 and parts of 1.5.
Section 2.2.1 and 2.2.2