

Introduction to Algorithms

Part 3: \mathcal{P} , \mathcal{NP} Hard Problems

- 1) *Polynomial Time: \mathcal{P} and \mathcal{NP}*
- 2) *\mathcal{NP} -Completeness*
- 3) *Dealing with Hard Problems*
- 4) *Lower Bounds*
- 5) *Books*

Chapter 1: Polynomial Time: \mathcal{P} and \mathcal{NP}

The field of *complexity theory* deals with how fast can one solve a certain type of problem. Or more generally how much resources does it take: time, memory-space, number of processors etc. We assume that the problem is solvable.

1.1 Computations, Decisions and Languages

The most common resource is time: number of steps. This is generally computed in terms of n , the length of the input string. We will use an informal model of a computer and an algorithm. All the definitions can be made precise by using a model of a computer such as a Turing machine.

While we are interested in the difficulty of a computation, we will focus our hardness results on the difficulty of yes–no questions. These results immediately generalize to questions about general computations. It is also possible to state definitions in terms of *languages*, where a language is defined as a set of strings: the language associated with a question is the set of all strings representing questions for which the answer is Yes.

1.2 The Class \mathcal{P}

The collection of all problems that can be solved in polynomial time is called \mathcal{P} . That is, a decision question is in \mathcal{P} if there exists an exponent k and an algorithm for the question that runs in time $O(n^k)$ where n is the length of the input.

\mathcal{P} roughly captures the class of practically solvable problems. Or at least that is the conventional wisdom. Something that runs in time 2^n requires double the time if one adds one character to the input. Something that runs in polynomial time does not suffer from this problem.

\mathcal{P} is robust in the sense that any two reasonable (deterministic) models of computation give rise to the same definition of \mathcal{P} .

◇ *True Boolean Formulas*

A *boolean formula* consists of variables and negated variables (known collectively as *literals*), and the operations “and” and “or”. We use \vee for “or”, \wedge for “and”, and \bar{x} for “not x ”. For example

$$x \wedge (x \vee y \vee \bar{z}) \wedge (\bar{x} \vee \bar{y})$$

An *assignment* is a setting of each of the variables to either TRUE or FALSE. For example if x and y are TRUE and z is FALSE, then the above boolean formula is FALSE.

|| TRUEBF
 || Input: ϕ a boolean formula, assignment ψ
 || Question: Does ψ make ϕ TRUE?

Given a boolean function and an assignment of the variables, we can determine whether the formula is TRUE or not in linear time. (A formula can be evaluated like a normal arithmetic expression using a single stack.) So it follows that the TRUEBF problem is in P.

◇ *Paths*

Recall that a *path* is a sequence of edges leading from one node to another.

|| PATH
 || Input: graph G , nodes a and b
 || Question: Is there a path from a to b in G ?

This problem is in P. To see if there is a path from node a to node b , one might determine all the nodes reachable from a by doing for instance a breadth-first search or Dijkstra's algorithm. Note that the actual running time depends on the representation of the graph.

1.3 Nondeterminism and NP

What happens if we allow our machines to receive advice? We define a nondeterministic algorithm as one which receives two input: the regular input and a *hint*. The input is considered to be a Yes-instance iff there is some hint that causes the algorithm to say yes. So, the input is a No-instance only if every hint causes the algorithm to say no. A hint that causes the algorithm to say yes is called a *certificate*.

The collection of all problems that can be solved in polynomial time using nondeterminism is called NP. That is, a decision question is in NP if there exists an exponent k and an nondeterministic algorithm for the question that for all hints runs in time $O(n^k)$ where n is the length of the input.

◇ *Satisfiability*

In dealing with boolean formulas, a *clause* is the “or” of a collection of literals. A formula is said to be in *conjunctive form* if it is the “and” of several clauses.

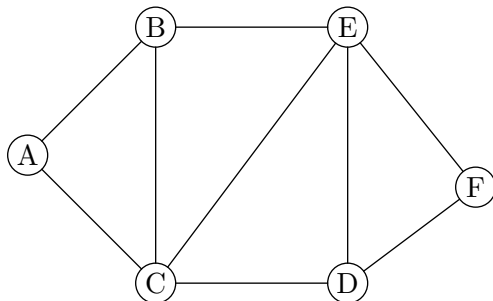
|| SAT
 || Input: ϕ a boolean formula in conjunctive form
 || Question: Is there an assignment that makes ϕ TRUE?

The SAT problem is in NP. The certificate is the assignment of the variables. However, the number of assignments is exponential, so there does not exist an obvious polynomial-time algorithm.

◇ *Hamiltonian path*

A *hamiltonian path* from node a to node b is a path that visits every node in the graph exactly once.

EXAMPLE. In the graph drawn, there is a path from every node to every other node. There is a hamiltonian path from A to F but not one from C to E .



|| HAMPATH
|| Input: graph G , nodes a and b
|| Question: Is there a hamiltonian path from a to b in G ?

It is far from obvious how to decide HAMPATH in polynomial time. (Note that trying all possible paths does not work since there are exponentially many.) But there is a fast nondeterministic program for it: the certificate is the actual path.

The time taken by the nondeterministic algorithm is at most quadratic in the number of nodes of the graph. And so HAMPATH is in NP. On the other hand, its complement does not appear to be so. No simple general certificate of the nonexistence of a hamiltonian path springs to mind.

◇ *What is Nondeterminism?*

Thus NP can be viewed as those problems that are *checkable* in polynomial time: the certificate provides a proof of a Yes-answer. So we have this Prover/Verifier situation. This problem can be decided by a Verifier in polynomial time if he is given a hint from an omniscient Prover. Note that the correct answer is made: if there is a hamiltonian path then the correct hint will be verified and the Verifier will say yes; and if there is no hamiltonian path then there is no way the Verifier can be conned into saying yes.

Nondeterminism is more usually defined as allowing a program more than one possible next move and the program accepts the input iff there is some possible sequence of computations that lead to an accept state. This notion is equivalent to the one defined as above: the certificate is a record of the correct next moves.

1.4 P versus NP

It would seem that P and NP might be different sets. In fact, probably the most important unsolved problems in Mathematics and Computer Science today is:

CONJECTURE. $P \neq NP$

If the conjecture is true, then many problems for which we would like efficient algorithms do not have them. Which would be sad. If the conjecture is false, then much of cryptography is under threat. Which would be sad.

Exercises

1. Define and compare the concepts of: polynomial, logarithmic, linear, quadratic, exponential, superexponential, superlinear.
2. Consider a list of implications involving literals: an *implication* $a \rightarrow b$ means that if a is TRUE then b is TRUE. Give a polynomial-time algorithm to determine if a set of implications are satisfiable.
3. One is given a boolean function `GoodTest` which takes as input a string (and runs in precisely 1 second irrespective of the length of the string). You must devise a polynomial-time algorithm for the following problem:

|| `GoodDivision`
|| Input: A string of length n
|| Question: Can the string be divided into pieces such that each piece is
|| good (returns true from `GoodTest`)?

Chapter 2: NP-Completeness

While we cannot determine whether $P = NP$ or not, we can, however, identify problems that are the hardest in NP. These are called the NP-complete problems. They have the property that if there is a polynomial-time algorithm for any one of them then there is a polynomial-time algorithm for every problem in NP.

2.1 Reductions

For decision problems A and B , A is said to be **polynomial-time reducible** to B (written $A \leq_p B$) if there is a polynomial-time computable function f such that

$$q \text{ is a Yes-instance of } A \iff f(q) \text{ is a Yes-instance of } B$$

That is, f translates questions about A into questions about B while preserving the answer to the question.

The key result:

- LEMMA. a) If $A \leq_p B$ and B in P, then A in P.
b) If $A \leq_p B$ and A not in P, then B not in P.

PROOF. Suppose the reduction from A to B is given by the function f which is computable in $O(n^k)$ time. And suppose we can decide questions about B in $O(n^\ell)$ time. Then we build a polynomial-time decider for A as follows. It takes the input q , computes $f(q)$ and then sees whether $f(q)$ is a Yes-instance of B or not. Does the program run in polynomial-time? Yes. If q has length n then the length of $f(q)$ is at most $O(n^k)$ (since a program can only write one symbol each step). Then the test about B takes $O(n^{k\ell})$ time. And that's polynomial. \diamond

2.2 NP-completeness

We need a definition:

- A decision problem S is defined to be NP-complete if*
- It is in NP; and*
 - For all A in NP it holds that $A \leq_P S$.*

Note that this means that:

- If S in NP-complete and S in P, then $P=NP$.
- If S is NP-complete and T in NP and $S \leq_p T$, then T is NP-complete.

2.3 Examples

There are tomes of NP-complete problems. The standard method to proving NP-completeness is to take a problem that is known to be NP-complete and reduce it to your problem. What started the whole process going was Cook's original result:

THEOREM. SAT is NP-complete.

We omit the proof. Some more examples:

- The 3SAT problem is NP-complete:

|| 3SAT
|| Input: ϕ a boolean formula in conjunctive form with three literals per
|| clause
|| Question: Is there a satisfying assignment?

- HAMPATH is NP-complete.
- Domination. A set of nodes in a graph is a *dominating set* if every other node is adjacent to at least one of these nodes. For example, in the graph on page 4, $\{A, D\}$ is a dominating set.

The DOMINATION problem is NP-complete:

|| DOMINATION
|| Input: Graph G and integer k
|| Question: Does there exist a dominating set of G of at most k nodes?

2.4 Primes, Composites and Factorization

An old problem is to find a fast algorithm which determines whether a number is prime or not, and if composite, determines a factorization. Now, it is important to realize that a number m is input into a program in binary. Thus the length of the input is $\log_2 m$. That is, we want algorithms that run in time polynomial in the number of bits, not in time proportional to the number itself.

So define PRIME as determining whether a binary input is prime and COMPOSITE as determining whether it is composite. It is clear that COMPOSITE is in NP: simply guess a split into two factors and then verify it by multiplying the two factors. It is not immediately clear what a certificate for primeness looks like, but elementary number theory provides an answer, and so it has been known for many years that PRIME is in NP.

The question of compositeness was conjectured by some to be NP-complete. However, in 2002 it was shown that PRIME is in P, and hence so is COMPOSITE. Nevertheless, there is still no polynomial-time algorithm for determining the factorization of a composite number. Which is a good thing in some respects, since RSA public-key cryptography assumes that factorization is hard.

2.5 Proving NP-completeness by Reduction

◇ *3SAT*

We show *3SAT* is NP-complete. Since *3SAT* is a “restriction” of *SAT*, it must be in NP since *SAT* is.

We now show how to reduce *SAT* to *3SAT*. Our task is to describe a polynomial-time algorithm which takes as input a boolean formula ϕ in conjunctive form, and produces a boolean formula ψ in conjunctive form with 3 literals per clause such that ϕ is satisfiable iff ψ is.

The idea is to take each clause C of ϕ and replace it by a family D of clauses. For example, suppose there was a clause $C = a \vee b \vee c \vee d \vee e$. What does this mean? This means that in a satisfying assignment of ϕ at least one of the five literals must be TRUE. It turns out that one can simulate this by

$$D = (a \vee b \vee x) \wedge (\bar{x} \vee c \vee y) \wedge (\bar{y} \vee d \vee e)$$

where x and y are totally new variables. We need to show two things: (1) If all the literals in C are FALSE, then D is FALSE; and (2) If one of the literals in C is TRUE, then there exist settings of x and y such that D is TRUE. (Why?)

To prove (1), observe that if all of a through e are FALSE and D is TRUE, then by first clause of D it holds that x is TRUE. Then by the second clause it holds that y is TRUE. But then the third clause is FALSE. (2) is proved similarly. It is left to the reader (DO IT!) to explain how to deal with clauses C of other sizes.

So this construction yields a boolean formula ψ in the right form. If ϕ is satisfiable, then there is an assignment where each clause is TRUE. This can be extended to assignment where ϕ is TRUE. On the other hand, if an assignment evaluates ϕ to FALSE, then one of the clauses must be false and one of the corresponding family of clauses in ψ must be false. The last thing to check is that the process of conversion can in fact be encoded as a polynomial-time algorithm. Which it can.

◇ *Clique*

A *clique* in a graph is a set of nodes each pair of which is joined by an edge.

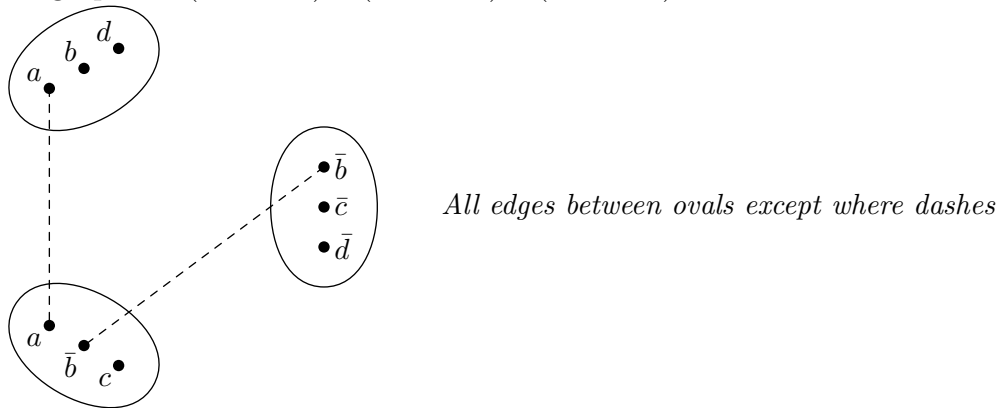
CLIQUE
Input: Graph G and integer k
Question: Does there exist a clique with k nodes?

We show that *CLIQUE* is NP-complete. First we must check that the problem is in NP. The program guesses k nodes and checks they form a clique.

We now show how to reduce **3SAT** to **CLIQUE**. That is, we describe a procedure which takes a boolean formula ϕ and produces a graph G_ϕ and integer m such that ϕ is satisfiable iff G_ϕ has a clique with m nodes.

Consider the input ϕ : suppose it has c clauses. For each clause and each literal in that clause, create a node (so we have $3c$ nodes). Then join two nodes if they are from different clauses unless they correspond to opposite literals (that is, do not join a to \bar{a} etc.). The result is a graph G_ϕ .

This is the graph for $(a \vee \bar{b} \vee c) \wedge (a \vee b \vee d) \wedge (\bar{b} \vee \bar{c} \vee \bar{d})$:



Claim: $\langle \phi \rangle \mapsto \langle G_\phi, c \rangle$ is the desired reduction.

First of all, the graph can be constructed in polynomial-time. Then observe that a clique of size c would have to contain exactly one node from each clause (since it cannot contain two nodes from the same clause). Then note that the clique cannot contain both a and \bar{a} . Thus, if we set all the literals in the clique to be true, we obtain a satisfying assignment. On the other hand, if we take a satisfying assignment, then a clique with c nodes can be obtained by taking one true literal from each clause.

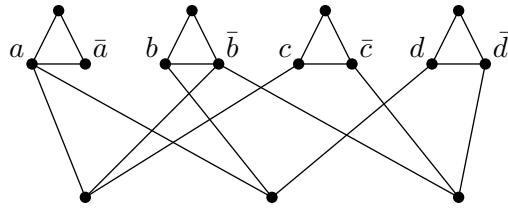
◇ **Domination**

We show **DOMINATION** is NP-complete. First we must check that the problem is in NP. But that is not hard. The program guesses k nodes and then checks whether they form a dominating set or not.

We now show how to reduce **3SAT** to **DOMINATION**. That is, we describe a procedure which takes a boolean formula ϕ and produces a graph G_ϕ and an integer m such that ϕ is satisfiable iff there is a dominating set of G_ϕ of m nodes.

Suppose input ϕ in conjunctive form has c clauses and a total of m variables. For each clause, create a node. For each variable v , create a triangle with one node labeled v and one labeled \bar{v} . Then for each clause, join the clause-node to (the nodes corresponding to) the three literals that are in that clause. The result is a graph G_ϕ .

This is the graph for $(a \vee \bar{b} \vee c) \wedge (a \vee b \vee d) \wedge (\bar{b} \vee \bar{c} \vee \bar{d})$:



Claim: $\langle \phi \rangle \mapsto \langle G_\phi, m \rangle$ is the desired reduction.

If ϕ has a satisfying assignment, then let D be the set of the m nodes corresponding to the TRUE literals in the assignment. Then each triangle is dominated (there's one node of D in each triangle). And each clause-node is dominated, since one of the literals in that clause must be TRUE.

Conversely, suppose G_ϕ has a dominating set of size m . Then D must consist of one node from each triangle, since all the unlabeled nodes of the triangles must be dominated. Now to be dominating every clause must be connected to one literal in D . So if we set all the literals (corresponding to nodes) in D to TRUE, we have a satisfying assignment.

◇ *Independence*

An independent set in a graph is a set of nodes no pair of which is joined by an edge.

INDEPENDENCE
Input: Graph G and integer k
Question: Does there exist an independent set with k nodes?

This problem is NP-complete. We simply reduce from CLIQUE by taking the complement of the graph: change every edge into a non-edge and vice versa.

◇ *Independence with Maximum Degree 3*

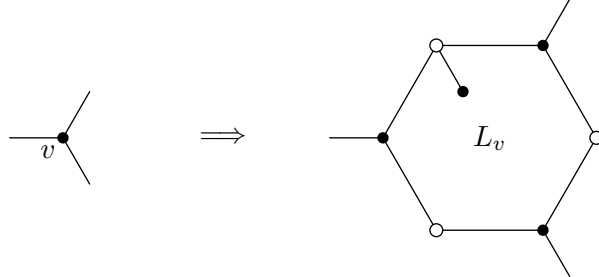
The degree of a node is the number of edges incident with it. We now show that the INDEPENDENCE problem remains hard even if the graph is guaranteed to have no node with degree more than 3.

IndepMaxDeg3
Input: Graph G with maximum degree 3 and integer k
Question: Does there exist a dominating set of G of at most k nodes?

We now show how to reduce INDEPENDENCE to IndepMaxDeg3. The input is a graph G and integer k . We produce a new graph H and integer l .

The construction is to replace each node v in G with a larger subgraph, call it L_v . In particular, if v has degree d , then L_v has $2d + 1$ nodes: it consists of a cycle of length $2d$

and one leaf adjacent to one node in the cycle. The nodes of the cycle are colored black and white alternately such that the leaf is adjacent to a white node; the leaf is colored black. For each original edge in G , say joining vw , there is an edge joining a black node of L_v and a black node of L_w .



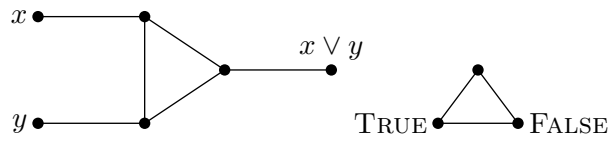
If D denotes the sum of the degrees in the original graph, the claim is that the reduction is:

$$\langle G, k \rangle \mapsto \langle H, D + k \rangle$$

We need to argue that G has an independent set with k nodes iff H has an independent set with $D + k$ nodes. For, given any independent set in the graph outside L_v , one can always add the d white nodes of L_v . The only way to take $d + 1$ nodes in L_v is to take the black nodes, but then one cannot take a black node in any L_w if v and w were adjacent in G .

Exercises

1. Define **SPATH** as the problem of given a graph G , nodes a and b and integer k , is there a path from a to b of length at most k . Define **LPATH** similarly except that there is a path of length at least k (and no repeated node). Show that **SPATH** is in P. Show that **LPATH** is NP-complete. (You may assume the NP-completeness of **HAMPATH**.)
2. Show that the independence number of a graph with maximum degree 2 can be computed in polynomial time.
3. Show that if $P=NP$ then there is a polynomial-time algorithm which on input a graph finds a hamiltonian path if one exists. (Note: this is not immediate.)
4. Show that if $P=NP$ then there is a polynomial-time algorithm which on input ϕ finds a satisfying assignment if one exists.
5. Show that the **DOMINATION** problem is still hard in bipartite graphs. (Hint: mimic the reduction from **3SAT** but change the way a and \bar{a} are connected.)
6. © Show that deciding if a graph has a 3-coloring of the nodes is NP-complete. *Hint:* You might want to reduce from **3SAT** using the gadgets pictured below.



Chapter 3: Dealing with Hard Problems

When the problem is known to be NP-complete (or worse), what you gonna do? The best one can hope for is an algorithm that is guaranteed to be close. Alternatively, one can ask for an algorithm that is close most of the time, or maybe is correct but only its average running time is fast.

3.1 Approximation Algorithms

These are algorithms which run fast and are guaranteed to be close to the correct answer. A *c-approximation algorithm* is guaranteed to be within a factor c of the exact solution.

For example, we saw earlier that there is a polynomial-time algorithm for maximum matching in a graph. But there is a simple 2-approximation algorithm: start with any edge and keep on adding edges to the matching until you cannot proceed. This is called a *maximal* matching.

LEMMA. Every maximal matching M has size at least half the maximum matching number

PROOF. Let M^* be a maximum matching. Since M is maximal, none of the edges of M^* can be added to M . Thus for each edge of M^* , one of its ends must already be used in M . Thus the total number of nodes in M must be at least the number of edges in M^* : but every edge has two nodes, whence the result. \diamond

The result is accurate. For example, if one has a path with three edges, then the maximum matching number is 2, but one can get a maximal matching with just the central edge.

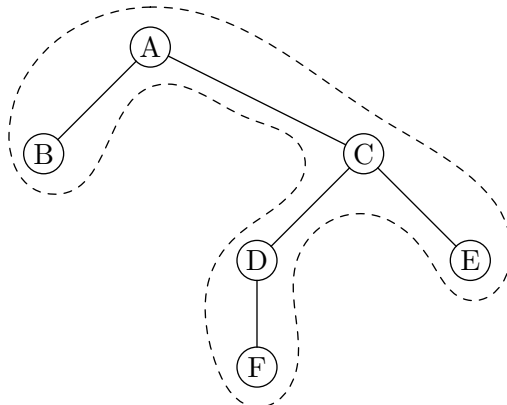
\diamond *Traveling Salesman Problem*

Recall that a hamiltonian cycle is a cycle that visits all nodes exactly once and ends up where it started. In the traveling salesman problem TSP, one has the complete graph and each edge has a weight, and the task is to find the hamiltonian cycle with minimum weight. Since the HAMPATH problem is NP-complete, it is easy to show/believe that TSP is NP-complete too.

Now suppose the graph obeys the *triangle inequality*: for all nodes a, b, c , the weight of the edge (a, b) is at most the sum of the weights of the edges (a, c) and (b, c) . Such would be true if the salesman were on a flat plane, and the weights represented actual distances.

There is a simple 2-approximation algorithm for this version. Start with a minimum spanning tree T . Since a tour is a connected spanning subgraph, every tour has weight at least that of T . Now, if one uses every edge of T twice, then one can visit every node at least once and end up where one started.

To turn this into a tour we simply skip repeated nodes. For example, in the picture the traversal of T goes $ABACDFDCECA$ and we short-cut this to $ABCDFEA$. We rely on the triangle inequality for the shortcut to be no worse than the original path.



◇ *Coloring 3-colorable graphs*

In an earlier exercise you were asked to show that deciding whether a graph is 3-colorable is NP-complete. But consider the following problem: suppose you know that the graph is 3-colorable (or at least your algorithm need only work for 3-colorable graphs). How easy is it to find a 3-coloring?

Well it seems to be very difficult. For a long time the best polynomial-time algorithm known used $O(\sqrt{n})$ colors to color a 3-colorable graph. This algorithm rested on two facts:

1. One can easily 2-color a bipartite (meaning 2-colorable) graph.
2. If a graph has maximum degree Δ , one can easily produce a coloring with at most $\Delta + 1$ colors.

The only fact about 3-colorable graphs that we need is that in a 3-coloring, the neighbors of a node are colored with two colors; that is, the neighborhood of a node is bipartite.

Proving this and putting this all together is left as an interesting exercise.

◇ *NP-completeness*

It can be shown that finding a c -approximation of an independent set, for any constant c , is NP-complete. Indeed, Arora et al. showed that there are problems where for sufficiently small ε , coming within n^ε of the correct value is NP-complete.

3.2 Randomized Algorithms

Sometimes randomness can be the key to obtaining a fast algorithm. There are two basic types of randomized algorithms:

1. A *Monte Carlo* algorithm use randomness and the answer is guaranteed to be correct most of the time.
2. A *Las Vegas* algorithm use randomness, the answer is guaranteed to be correct, but the running time is only an average.

We give here only a simple example. Suppose you have two copies of the same long binary file at two different locations: call them F_1 and F_2 . Then a hacker has a go at F_1 . You would like to check whether the file is corrupted. Obviously this can be achieved by sending the one file to the other location. But we would like to minimize the number of bits that are communicated. This can be done with random check-bits.

Assuming that you have the same random number generator at both ends, you can do the following. Compute a random subset S of the bits by tossing a coin for each bit. Then compute the XOR of the bits of F_i corresponding to S : call the result c_i . Obviously, if F_1 is uncorrupted then $c_1 = c_2$.

LEMMA. If $F_1 \neq F_2$ then $c_1 \neq c_2$ 50% of the time.

PROOF. Consider the last bit in which the files differ: call it b . Let u_1 and u_2 be the calculations so far. If $u_1 = u_2$, then $c_1 \neq c_2$ iff S includes b : which is a 50-50 event. But if $u_1 \neq u_2$, then $c_1 \neq c_2$ iff S does not include b , again a 50-50 event. \diamond

Thus, if $F_1 \neq F_2$ there is a 50-50 chance that this random check-bit is different. Thus we have a Monte Carlo algorithm. If you perform this computation 100 times, and each check bit matches, then there is only a 1 in 2^{100} chance that the files are different. This is about the same odds as winning the lottery, four weeks in a row!

Exercises

1. Give a fast algorithm to 2-color a bipartite (meaning 2-colorable) graph.
2. Give a fast algorithm to color the nodes of a graph with at most $\Delta + 1$ colors, where Δ denotes the maximum degree.
3. Hence show how to color a 3-colorable graph with $O(\sqrt{n})$ colors.
4. © The k -center problem is to place k facilities (e.g. fire-stations), such that the farthest that anyone has to travel to the nearest fire-station is as small as possible. Use a greedy approach to produce a 2-approximation.

5. Explain how to use an unbiased coin to choose among 3 people. How many tosses does your algorithm take?
6. Suppose you have two n -bit files at different locations and they are known to differ in exactly one bit. Give an algorithm to determine which bit they differ in that uses $O(\log n)$ communication.
7. Use dynamic programming to obtain an algorithm for TSP that runs in time $O(2^n)$.

Chapter 4: Lower Bounds

In Algorithms, we are able to prove lower bounds only for very simple problems. We look here at some problems related to sorting.

4.1 Information Theory

Suppose you have to guess a number between 1 and 100 by asking yes–no queries. You might start by asking “Is it 1?” Then “Is it 2?” and so on. But one can do better by asking first “Is it larger than 50?” and if the answer is yes then asking “Is it larger than 75?” and so on. In other words, we can perform a binary search. In the worst case the binary search will take 7 queries. (Try it!) The question is: Is 7 the best we can hope for?

The answer to that question is yes. And the argument is as follows. The *problem* is:

Determine a number between 1 and 100.

There are 100 possible *solutions*—call the set S . Each time one asks a yes–no question. This splits the set up into sets S_Y and S_N . If the person answers yes, then we are left with the set S_Y . If she answers no, then we are left with the set S_N . One of these sets must have at least 50 elements. So in the worst case we are left with at least 50 elements. In other words:

In the worst case the number of solutions comes down by a factor of at most 2 for each query.

We are done when there is only one possible solution remaining. So if we define I as the number of solutions to the problem then

The number of queries needed, in the worst case, is at least $\log_2 I$

In the example above, the lower bound is $\log_2 100$ which, rounded up, is 7. Note that:

- The lower bound does not assume a particular algorithm, but rather deals with the best one could hope for in any algorithm. It depends only on the number of solutions.
- The lower bound is also valid for the average case behavior of any algorithm. The idea is that one query will on average decrease the number of solutions by a factor of at most 2.

The information-theory lower bound is the only general procedure out there for providing estimates of the time required by the best possible algorithm. In applying information theory, we must determine or estimate I ; this is not always so simple.

4.2 Examples

SORTING.

Here the problem is: given a list of numbers, output them in sorted order. The solution is the sorted list. For a list of n numbers there are $I = n!$ solutions. So a lower bound on sorting is given by

$$\log_2(n!)$$

Since there is an approximation

$$n! \approx \left(\frac{n}{e}\right)^n$$

$\log_2(n!)$ is about $n \log_2 n - O(n)$.

This means that any comparison-based sorting algorithm must take at least this many comparisons (in the worst case). And hence the best we can hope for time-wise for a sorting algorithm is $O(n \log n)$. Merge sort comes close to this value.

MAXIMUM.

Here the problem is: given a list of numbers, output the maximum. For a list of n elements there are n answers. So a lower bound on maximum-finding is given by

$$\log_2 n$$

However, there is no algorithm for finding the maximum that runs this fast. (This problem is looked at again in the next section.)

SET-MAXIMA.

Suppose one has as input the values of A , B and C and one must output the maximum in each of the three sets $\{A, B\}$, $\{A, C\}$ and $\{B, C\}$. Naïvely there are 2 possible answers for the maximum in each set, so one might think that $I = 2^3$. However, this is not correct. It cannot happen that the maximum in $\{A, B\}$ is A , the maximum in $\{B, C\}$ is B and the maximum in $\{C, A\}$ is C . (Why not?) In fact $I = 6$ (or $3!$) as there is one possible output answer for each ordering of the set $\{A, B, C\}$.

4.3 Adversarial Arguments

We want to know about the worst-case behavior of a comparison-based algorithm. Consider, for example, computing the maximum of n elements using comparisons: this takes at least $n - 1$ comparisons. The idea is that if one wants a winner out of n players, each player except for the winner must lose at least one game, hence there must be $n - 1$ games.

This type of argument can be generalized to what is called an *adversarial argument*: we assume that we are asking a question of an adversary who is making up the data as he goes along. Of course, he is not allowed to invalidate the answers to previous questions.

Consider the problem of computing the maximum and minimum of n elements simultaneously. For an adversarial argument, we must describe a strategy for the adversary. He will create two buckets: L and H for low and high. Initially, both buckets are empty. Each value in L will be less than each value in H .

For each question (which is of the form “Is $A[i]$ bigger than $A[j]$ ”), the adversary does the following.

1. If neither element has been assigned to a bucket, he arbitrarily puts one in L , one in H .
2. If only one element has been assigned to a bucket, then he puts the other element in the other bucket.
3. Otherwise the adversary does nothing.

Then to answer the question, the adversary chooses any answer that is compatible with previous information and respects the property that all of H is bigger than all of L .

Obviously, the user must ask a question about each element (okay, we should assume $n \geq 2$). So every element will eventually be assigned to a bucket. Furthermore, the user has to determine the maximum in H and the minimum in L . Each question provides information about only one thing: (a) the assignment to buckets, (b) the maximum in H , or (c) the minimum in L .

Assigning to buckets takes at least $n/2$ comparisons. Determining the maximum in H and the minimum in L (which are disjoint) together takes $(|H| - 1) + (|L| - 1) = n - 2$ comparisons. Thus, determining the maximum and minimum in the set takes at least $3n/2 - 2$ comparisons.

Exercises

1. Sorting takes $O(n \log n)$ comparisons. How long does it take to **check** whether a list is sorted or not?
2. © Consider a set-maxima problem where one is given a list of n numbers and n subsets from this list, and one must find the maximum in each subset. Find an information-theoretic lower bound on the number of comparisons needed.
3. © Use an adversarial argument to give a $3n/2 - O(1)$ lower bound on the number of comparisons needed to find the median of a list of numbers.

Chapter 5: Books

I have made much use of:

- *Data Structures and Algorithms*,
A. Aho, J. Hopcroft and J. Ullman
- *Fundamentals of Algorithmics*,
G. Brassard and P. Bratley
- *Introduction to Algorithms*,
T. Cormen, C. Leiserson and R. Rivest
- *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*,
F.T. Leighton
- *Data Structures and Algorithm Analysis*,
M.A. Weiss
- *Introduction to Computer Theory*,
D.I.A. Cohen
- *Introduction to Discrete Mathematics*,
S.C. Althoen and R.J. Bumcrot
- *Mathematical Theory of Computation*,
Z. Manna
- *Introduction to the Theory of Computation*,
M. Sipser