Using Mathematical Induction to Design Computer Algorithms

by

Udi Manber

Computer Sciences Technical Report #660

August 1986

USING MATHEMATICAL INDUCTION TO DESIGN COMPUTER ALGORITHMS

Udi Manber*

Department of Computer Science
University of Wisconsin
Madison, WI 53706

May 1987

Key words. combinatorial algorithms, computational complexity, design of algorithms, mathematical induction, proof techniques.

ABSTRACT

An analogy between proving mathematical theorems and designing computer algorithms is explored in this paper. This analogy provides an elegant methodology for designing algorithms, explaining their behavior, and understanding their key ideas. The paper identifies several mathematical proof techniques, mostly based on induction, and presents analogous algorithm design techniques. Each of these techniques is illustrated by several examples of algorithms.

* This research was supported in part by an NSF grant MCS-8303134, and an NSF Presidential Young Investigator Award.

1. Introduction

This paper presents a methodology, based on mathematical induction, for approaching the design and the teaching of combinatorial algorithms. While this methodology does not cover all possible ways of designing algorithms it does cover many known techniques. It also provides an elegant intuitive framework for explaining the design of algorithms in more depth. The heart of the methodology lies in an analogy between the intellectual process of proving mathematical theorems and that of designing combinatorial algorithms. We claim that although these two processes serve different purposes and achieve different types of results they are more similar than it seems. This claim is established in this paper by presenting a series of examples of known algorithms, each explained in terms of the analogous proof techniques. We include examples of recursion, divide and conquer, dynamic programming, greedy algorithms, and plane sweep, among others. Our aim in this paper is not only to highlight the methodology, but also to introduce the reader to several interesting algorithms by showing how these algorithms were (or could have been) developed. In this respect, this paper can serve as an introduction by example to the design of combinatorial algorithms. We believe that students can get more motivation, greater depth, and better understanding of algorithms by this methodology.

We usually think of mathematical induction in the way it is defined in Peano axioms. If a property P is true for 1 and its truth for n implies its truth for $n+1$ then it is true for all natural numbers. Over the years many variations of this proof technique have been developed. There is a very direct connection between mathematical induction and algorithms (Knuth calls induction an *algorithmic proof technique* [Kn73a]). Indeed, mathematical induction has been used in the theory of algorithms for a long time, usually to prove correctness of algorithms. This is done by associating *assertions* with certain steps of the algorithm and proving that they hold initially and that they are invariant under certain operations. This method was originally suggested by Goldstine and von Neumann, and developed later by Floyd and others. Dijkstra [Di76] and Gries [Gr81] present a methodology similar to ours to develop programs hand in hand with the proof of their correctness. We borrow some of their techniques, but our emphasis is different. We concentrate on the high level algorithmic ideas without going down to the actual program level. PRL [BC85] and later NuPRL [Co86] use mathematical proofs as the basic part of a program development system.

In this paper we concentrate on the use of mathematical induction as a tool for explaining and designing high level algorithms. Our goals are mainly pedagogical, but of course whenever something can be explained better it is usually understood better. We identify several proof techniques, mostly based on induction, and show analogous algorithm design techniques. Among the proof techniques we discuss are strengthening the induction hypothesis, choosing the induction sequence wisely, double induction, reverse induction, and maximal counterexample. The novelty of our approach is twofold. First we collect seemingly different techniques of algorithm design under one umbrella, and secondly we utilize known mathematical proof techniques for algorithm design. The latter is especially important since it opens the door to the use of powerful techniques that have been developed for many years in another discipline.

The presentation given here is intuitive and non-formal. The paper is self contained, except for the assumption that the reader is familiar with mathematical induction and basic data structures. For each proof technique we explain the analogy briefly and present one or more examples of algorithms. The emphasis of the presentation is on how to use this methodology in the teaching and the design of algorithms. Our goal is not to present an algorithm in a way that makes it easier for a programmer to translate into a program, but rather in a way that makes it easier to *understand*. The algorithms are explained in this paper through a creative process rather than as finished products. Our goals in teaching algorithms are not only to show students how to solve particular problems, but also to help them solve new problems in the future. Teaching the thinking involved in designing an algorithm is as important as teaching the details of the solution. We believe that the methodology presented in this paper enhances the understanding of this thinking process.

One important aspect of computer algorithms that we mostly omit in this paper is data structures. Data structures are much like construction techniques in the eyes of an architect. No building can be designed without a comprehensive knowledge of these techniques. In every step of the design one has to consider the possibilities and the associated costs. But designing a building takes more than that. In the same way, a complicated algorithm requires more than just knowing the right data structures, although this is necessary. It requires a global view of the problem — the picture of the building. We concentrate on understanding and enjoying this view. Our goal is to extract the key algorithmic ideas and to clarify them. Separating implementation and data structure details from algorithmic techniques is very helpful in explaining algorithms.

It is important to note that although induction suggests implementation by recursion, this is not necessarily the case. In many cases, iterative implementation is just as easy, even if the algorithm was designed with induction in mind, and it is generally more efficient. For each algorithm we include references in the literature where a detailed description is given. We begin with three examples arranged in an increasing level of difficulty. We then present several mathematical proof techniques and their analogous algorithm design techniques. The analogy is illustrated in each case by one or more examples of algorithms.

2. A simple example

Consider the following problem. You are arranging a conference of scientists from different disciplines and you have a list of people you want to invite. You assume that everyone on the list will agree to come under the condition that there will be ample opportunity to exchange ideas. For each scientist you write down the names of all other scientists on the list with whom interaction is likely. You would like to invite as many people on the list as possible, but you want to guarantee that each one will have at least k others to interact with. You do not really have to arrange for the interactions, and, in particular, you do not have to make sure that there is enough time. You just want to lure everyone to the conference. How do you decide who to invite?

We start with some standard terminology. A *graph* $G=(V,E)$ consists of a set of *vertices* V and a set of *edges* E, which are pairs of vertices. The graph may be *directed*, in which case the edges are directed, or *undirected*. The vertices may correspond to some entities or elements and the edges indicate some relationships among them. A *degree* of a vertex v, denoted by $d(v)$ is the number of edges adjacent to v. In a directed graph we distinguish between the *indegree* which is the number of edges coming into a vertex, and the *outdegree* which is the number of edges going out of a vertex. An *induced subgraph* $H=(U,F)$, $U \subseteq V$, $F \subseteq E$, is a subgraph of G consisting of the vertex set U and all the edges in G adjacent only to vertices in U.

For our problem, the vertices of the graph correspond to the scientists and two vertices are connected if there is a potential for the two corresponding scientists to exchange ideas. The problem above corresponds to the following graph theoretic problem. Given an undirected graph G and an integer k, find the maximal induced subgraph H of G such that all vertices of H have degree $\geq k$ (in H), or conclude that no such induced subgraph exists. As in all the examples in this paper, the reader is encouraged to try to solve this problem before reading further and then compare his/her solution to the one given.

A direct approach to solve the problem above is by removing vertices whose degree is $<k$. One immediately notices though that as vertices are removed with their adjacent edges the degrees of other vertices may be reduced. When a degree of a vertex becomes less than k it should be removed. This however raises questions about the order of removal. Should we remove all the vertices of degree $<k$ first and then deal with vertices whose degrees were reduced, or should we remove first one vertex of degree $<k$ and continue with affected vertices? (These two approaches correspond to breadth-first vs. depth-first.) Will both approaches lead to the same result? Will it be maximal? All these questions are pretty easy to answer; the approach described below makes them even easier.

Instead of thinking about the algorithm as a sequence of steps a computer has to make in order to calculate the result, think of it as a *proof* that the algorithm exists. We do not suggest attempting a formal proof (at least not at the first stage). The idea is only to imitate the steps one takes in proving a theorem in order to gain insight into the problem. We need to find the maximal induced subgraph that satisfies the conditions above. Here is a ''proof'' by induction.

Induction hypothesis: *We know how to solve the problem above provided that the number of vertices is $<n$.*

As is always the case with inductive proofs, we do not need to prove the statement above directly. We only need to prove that it is true for $n=1$, and its truth for $n-1$ vertices implies its truth for n vertices. Consider a graph with n vertices. If all the vertices have degrees $\geq k$ then the whole graph satisfies the conditions and we are done. Otherwise, there exists a vertex v with degree $<k$. It is obvious that the degree of v remains $<k$ in any induced subgraph, hence v cannot satisfy the conditions of the problem. Consequently, v and its adjacent edges can be removed without affecting the conditions of the theorem. After v is removed the graph has $n-1$ vertices and by the induction hypothesis we know how to solve the problem.

We are now basically done. The algorithm and the answers to the questions raised above are now clear. Any vertex of degree $<k$ can be removed. The order of removals is immaterial. The remaining graph after all these removals must be the maximal one because these removals are *mandatory*. It is also clear that the algorithm is correct because we designed it by proving its correctness! □

While the basic idea for the algorithm is clear we have not discussed implementation questions. What data structures should one use? What is the fastest way to find a vertex of degree $<k$ and how to remove it? As was mentioned in the introduction, we generally omit discussion on implementation in this paper. (In this case, efficient implementation is straightforward.)

One can argue at this point that the approach presented above is no more than designing algorithms through their proof of correctness. Putting the proof of correctness at the center of algorithm design has been suggested long ago and there is nothing new here. This is true, but there is more to it. We claim that the reason the methodology in this case may seem like no more than a proof of correctness is mainly because *the application of induction in this case is straightforward.* The induction is on the number of vertices, any vertex satisfying a simple condition can be chosen, and the removal is very easy. In the next examples the induction is less straightforward and, as a result, the methodology does not resemble just a proof but it is actually a different way of looking at algorithms.

3. Second example — the celebrity problem

This is a popular exercise in algorithm design. It is a nice example of a problem that has a solution which does not require scanning the whole data (or even a significant part of it). A *celebrity* is defined as someone who is known by everyone but does not know anyone. The problem is to identify the celebrity among n people, if such exists, by only asking questions of the form ''excuse me, do you know this person over there?'' (The assumption is that all the answers are correct, and that even the celebrity will answer.) The goal is to minimize the number of questions. Since there are $n(n-1)/2$ pairs of persons, there is potentially a need to ask $n(n-1)$ questions, in the worst case, if the questions are asked arbitrarily. It is not clear that one can do better in the worst case.

More technically, if we build a directed graph with the vertices corresponding to the persons and an edge from A to B if A knows B, then a celebrity corresponds to a *sink* (no pun intended). That is, a vertex with indegree $n-1$ and outdegree 0. It is clear that no more than one sink can exist (if there are two vertices with outdegree 0 then the maximal indegree can be at most $n-2$). The $n \times n$ adjacency matrix whose ij entry is 1 if the i'th person knows the j'th person is given. The problem is to identify a sink by looking at as few entries from the matrix as possible.

Again, instead of thinking about the algorithm as a sequence of questions we think of it as a proof that the celebrity exists. We attempt a proof by induction. That is, we assume that we know how to find a celebrity among $n-1$ persons and try to develop a method for finding a celebrity among n persons.

Induction hypothesis: *We know how to solve the celebrity problem among $<n$ persons.*

Consider the n'th person. Since, by definition, there is at most one celebrity, there are three possibilities: 1) the celebrity is among the first $n-1$ persons, 2) the n'th person is the celebrity, and 3) there is no celebrity. The first case is the easiest to handle. By the induction hypothesis we find the celebrity among the $n-1$ persons. Then, we only need to check that the n'th person knows the celebrity, and that the celebrity doesn't know the n'th person. The other two cases are more difficult since, to determine whether the n'th person is the celebrity, we may need to ask $2(n-1)$ questions. The total number of questions in the worst case may then be $Q(n)$ such that $Q(n) = Q(n-1) + 2(n-1)$. That leads to exactly $n(n-1)$ questions in the worst case, which we tried to avoid. We need another approach.

Our goal is to reduce the size of the problem and then use induction. Any reduction in size will do. We have the freedom to choose the best or the most convenient way to reduce the problem. If the straightforward way (choosing an arbitrary n'th person in this case) is not very efficient we look for other ways. It may be hard to identify a celebrity, but it is probably easier to identify someone as a non–celebrity. After all, there are definitely more non–celebrities than celebrities. Eliminating someone from consideration may be enough to reduce the problem from n to $n-1$. Moreover, we do not need to eliminate someone specific; anyone will do. Suppose we ask Alice whether she knows Bob. If she does then she cannot be a celebrity; if she doesn't then Bob cannot be a celebrity. We can eliminate one of them with one question.

Now consider again the 3 cases above. We do not just take an arbitrary person as the n'th person. We use the idea above to eliminate either Alice or Bob, and then solve the problem for the other $n-1$. We are guaranteed that case 2 will not occur since the person eliminated cannot be the celebrity. Furthermore, if case 3 occurs, namely there is no celebrity among the $n-1$ persons, then, since the n'th person was eliminated, there is no celebrity among the n persons. Only case 1 remains, but, as was mentioned above, this case is easy. If there is a celebrity among the $n-1$ persons it takes two more questions to verify that this is a celebrity for the whole set. Otherwise there is no celebrity.

In the worst case at most 3 questions are asked per person. (One question to remove one person from consideration initially, and two more questions to verify that the celebrity found by induction is indeed a celebrity.) The solution above shows that it is possible to identify a celebrity by looking at at most $3n$ entries in the adjacency matrix, even though a-priori the solution may be sensitive to each of the $n(n-1)$ entries.

The key idea to this elegant solution is to reduce the problem from n to $n-1$ in a non–trivial way. The moral of this example is that sometimes it pays off to spend some effort (in the case above — one question) to achieve the reduction more effectively. Do not just start by considering an *arbitrary* reduced problem of size $n-1$ and attempt to extend it. Select a *particular* reduced problem of size $n-1$. We will see more examples where substantial time is spent just for constructing the right order of induction, and that time is well spent. The main problem encountered by an algorithm designer is how to find the best way to achieve the reduction. The same problem is encountered by a theorem prover, and many different proof techniques have been developed. In this paper we explore such proof techniques and show how they can help the algorithm designer organize his/her search for an algorithm.

4. Third example — closest pair

Given a set of n points in the plane, we want to find the distance between the two closest points in the set. A straightforward solution is to check the distances between all pairs and take the minimal one. This requires $n(n-1)/2$ distance computations and $n(n-1)/2-1$ comparisons. The straightforward solution using induction would proceed by removing a point, solving the problem for $n-1$ points, and considering the extra point.

Induction hypothesis: *We know how to solve the closest pair problem for $<n$ points.*

However, if the only information gathered through the solution of the $n-1$ case is the minimum distance, then the distances of the n'th point to all the $n-1$ points must be checked. As a result, the total number of distance computations is $T(n)$ which satisfies the recurrence relation

$$T(n) = T(n-1) + n-1, \quad T(2) = 1.$$

It is easy to see that $T(n) = n(n-1)/2$, and in fact this is the same solution as the first one presented above. We want to find a faster solution. The algorithm described below was developed by Shamos and Hoey [SH75]; descriptions of it can also be found in [PS85] or [Sed83].

Consider again the inductive solution. In mathematical proofs the only goal is to conclude the proof as elegantly and as clearly as possible. In particular, in inductive proofs it is not important at all how long it would take to mechanically follow the proof for n from the proof of the base case. It is only important to show that this can be done. Obviously, in computer algorithms this factor is very important. When we consider the analogy and look for a proof we have to be conscious about efficiency. One way to improve the running time of an algorithm is through balancing, leading to a well known technique called *divide and conquer*. The idea of divide and conquer is to divide the problem into several subproblems of smaller sizes (preferably of equal sizes), solve them recursively, and then combine the solutions to a solution of the original problem. Putting it in our terminology, we assume that a solution to smaller problems is known and we only need to show how to use (combine) it to form a solution to the large problem. Divide and conquer (as almost any recursive technique) epitomizes the use of induction in designing algorithms, but, as will become apparent, it is not the only one.

Instead of considering one point at a time in the example above, we divide the set into two equal parts. The induction hypothesis can stay the same, but instead of reducing the problem of n points to the problem of $n-1$ points we reduce it to two problems with $n/2$ points. We assume that n is a power of 2 so that it is always possible to divide it into 2 equal parts. This assumption is justified in section 8. As was mentioned in the previous section, it is worthwhile to make some effort in dividing the problem so that it is easier to use the induction to combine the solutions. It seems reasonable here to divide the set by dividing the plane into two disjoint parts each containing half the set. The easiest way of doing this is by sorting all the points according for example to their x-

coordinates and dividing the plane by the vertical line that bisects the set (see Figure 1). (If several points lie on the vertical line we divide them arbitrarily.) The reason we choose this division is to minimize the work of combining the subsolutions. If the two parts interleave in some fashion then it will be harder to check for closest pairs.

Assume that the closest distance in the first subset is d_1 and in the second it is d_2. Assume further, without loss of generality, that $d_1 \le d_2$. We need to find the closest distance in the whole set, namely we have to see whether there is a point in one subset with a distance $<d_1$ to a point in the other subset. First we notice that it is sufficient to consider only the points that lie in a strip of width $2d_1$ centered around the vertical separator of the two subsets (see Figure 1). No other points can possibly be of distance less than d_1 from points in the other subset. Using this observation one can usually eliminate many points from consideration, but, in the worst case, all the points can still reside in the strip and we cannot ''afford'' to use the straightforward algorithm on them. Another less obvious observation is that, given a point p in that strip, there are very few points on the other side that can possibly be closer to it than d_1. The reason for that is the fact that all the points in one side of the strip are at least d_1 apart. In particular, given a point p with y-coordinate y_p only the points on the other side with y-coordinate y_q such that $|y_p - y_q| < d_1$ need be considered. There could be at most 6 such points on one side of the strip (see Figure 2 for the worst case). As a result, if we sort all points according to their y-coordinates and scan the points in order we need to check each point only against a constant number of other points (instead of $n-1$).

Let's study the performance of this algorithm. First we evaluate the time it takes to combine the subsolutions (i.e., the time to do the proof). We need to sort (according to

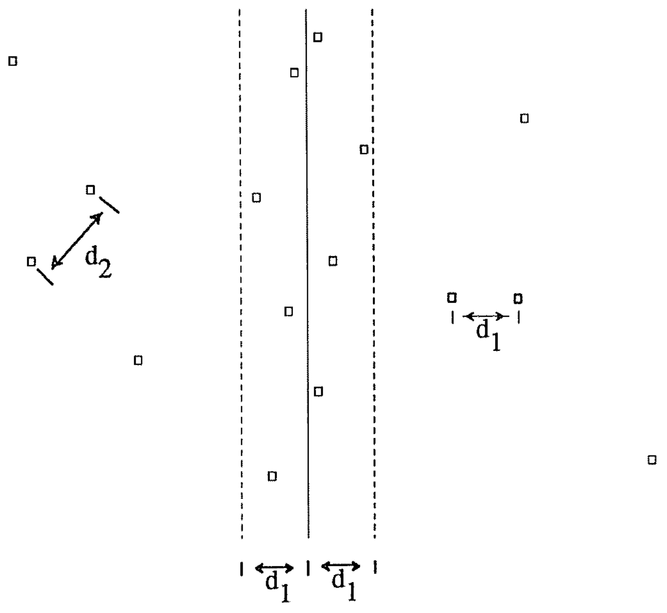

Figure 1: The closest pair problem

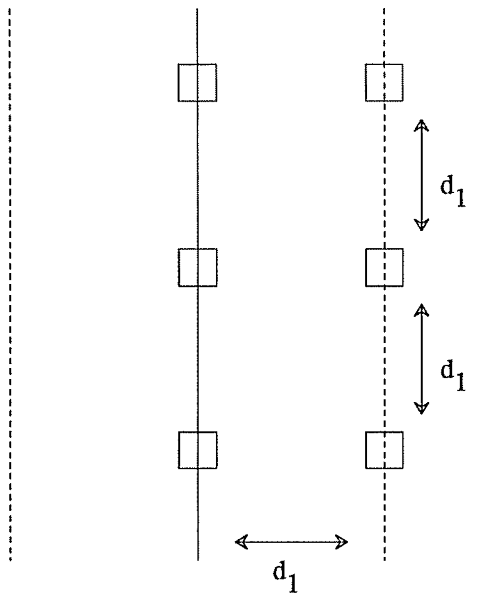

Figure 2: The worst case of 6 points d_1 apart

y-coordinates) and then linearly scan the points performing at most a constant number of distance computations for each point (it turns out that comparing against 4 other points is sufficient [Sed83]). It takes $O(n \log n)$[†] to sort and $O(n)$ to scan. To solve the problem of n points we solve two subproblems of $n/2$ points each and spend $O(n \log n)$ in combining the solutions (plus $O(n \log n)$ once for sorting the x-coordinates at the beginning). This leads to the following recurrence relation.

$$T(n) = 2T(n/2) + O(n \log n), \quad T(2) = 1.$$

The solution of this relation is $T(n) = O(n \log^2 n)$ (see for example [PB85]). This is much better than $O(n^2)$, but we still want to do better than that.

The key idea here is to *strengthen the induction hypothesis*. The reason we have to spend $O(n \log n)$ time in the combining step is the sorting. Viewing it in terms of the analogy – we can ‘‘prove’’ the closest pair problem provided we can ‘‘prove’’ the sorting problem. Although we know how to prove (solve) the sorting problem directly, it takes too long. Can we somehow solve the sorting problem at the same time we are solving the closest pair problem? That is, include the sorting in the induction hypothesis for the closest pair. The hypothesis becomes

Induction hypothesis: *Given a set of $<n$ points in the plane, we know how to find the closest distance and how to output the set sorted according to y-coordinates.*

[†] We say that a function $f(n)$ is $O(g(n))$ if there are constants a and b such that $f(n) \leq a\, g(n) + b$ for all n.

To prove the hypothesis we need to show how to extend the solution for sets of $<n$ elements to a set of size n. We have already shown how to find the minimal distance if we know how to sort. Hence, the only thing that needs to be done is to show how to sort the set of size n if we know the sorted order of the smaller sets. This is exactly the sorting technique known as *merge-sort.* It is not hard to merge two sorted sets of size $n/2$ into one sorted set of size n in $n-1$ comparisons; we leave that to the reader. The main advantage of this strengthening of the hypothesis is that we do not have to sort every time we combine the solutions – we only have to merge. We have to be very careful when we change the induction hypothesis. The reduced problem must always be exactly the same as the larger problem. When the hypothesis is changed it is easy to overlook this principle. In the example above, the new problem requires that all points be sorted, whereas in the old problem only the points in the strips were considered. Therefore we need to merge everything first to get the sorted order and only then eliminate the points outside the strips and run the rest of the algorithm.

Since merging and eliminating points take $O(n)$ the recurrence relation becomes

$$T(n) = 2T(n/2) + O(n),\ T(2) = 1,$$

which implies $T(n) = O(n \log n)$. This is a substantial improvement over the straightforward solution. □

5. Strengthening the induction hypothesis

Strengthening the induction hypothesis is probably the most important technique of proving mathematical theorems with induction. When attempting an inductive proof one often encounters the following scenario. Denote the theorem by P. The induction hypothesis can be denoted by $P(<n)$ and the proof must conclude that $P(<n) \Rightarrow P(n)$. Many times one can add another assumption, call it Q, under which the proof becomes easier. That is, it is easier to prove $[P \text{ and } Q](<n) \Rightarrow P(n)$. The assumption seems correct but it is not clear how to prove it. The trick is to include Q in the induction hypothesis. One now has to prove that $[P \text{ and } Q](<n) \Rightarrow [P \text{ and } Q](n)$. P and Q is a stronger theorem than just P, but many times stronger theorems are easier to prove (Polya [Po57] calls this principle **the inventor's paradox**). This process can be repeated and, with the right added assumptions, the proof becomes tractable. The closest point problem is a good example of how this principle is used to improve algorithms. Another good example, which we leave as an exercise, is presented by Bates and Constable [BC85] (they call the technique *generalization*): Given sequence $x_1, x_2, ..., x_n$ of real numbers (not necessarily positive) find the subsequence $x_i, x_{i+1}, ..., x_j$ such that the sum of the numbers in it is maximal among all subsequences. (The problem was suggested by Bently.) In this section we give one more example. We also use this method in the following sections.

The next example illustrates the most common error made while using this technique, which is to ignore the fact that an additional assumption was added and forget to adjust the proof. In other words, proving that $[P \text{ and } Q](<n) \Rightarrow P(n)$, without even noticing that Q was assumed. In our analogy this translates to ``solving'' a smaller problem that is not exactly the same as the original problem.

5.1. Minimal cost spanning trees

Consider a network of computers connected through bidirectional links. There is a positive cost associated with sending a message on each of the links (corresponding for example to the current load on that link). The cost is assumed to be independent of the direction of the message. We want to broadcast a message to all the computers starting from an arbitrary computer. The cost of the broadcast is the sum of the costs of the links used to forward the message. The network can be represented by an undirected graph with costs on the edges. For simplicity we assume that the costs are unique. The problem is to find a fixed connected subgraph (corresponding to the links used in the broadcast), containing all the nodes, such that the sum of the costs of the edges is minimal. It is not hard to see that the subgraph must be a tree. If any cycle had been present then we could have broken it by deleting one of its edges; the graph would still be connected but the cost would be smaller since all costs are positive. This subgraph is called the *minimal cost spanning tree* (MCST). Our goal is to find a fast algorithm for computing the MCST. For further reading on this problem see for example [AHU74].

The straightforward induction hypothesis is the following.

Induction hypothesis: *We know how to find the MCST given a graph with $<m$ edges.*

Given the MCST problem with m edges, how do we reduce it to to a problem with $<m$ edges? It is not hard to see that the minimal cost edge must be included in the MCST. (If the minimal cost edge is not included then adding it to the MCST would create a cycle; removing any other edge from the cycle creates a tree again, but with smaller cost.) We can thus remove the minimal cost edge from consideration and apply induction to the rest of the graph, which now contains less edges. This procedure is not valid as stated (i.e., the ''proof'' is wrong). The reason of course is that the problem with the smaller graph is not exactly the same as the original problem. The selection of one edge limits the selection of other edges. We cannot emphasize it too strongly — the induction hypothesis has to be precisely defined and followed.

The solution is to adjust the induction hypothesis, in this case strengthen it, to make it easier to prove. If, after removing an edge, the problem changes, we try to change the induction hypothesis to reflect this modified problem. The new induction hypothesis for the minimal cost spanning tree could look as follows.

Induction hypothesis: *We know how to find the MCST given a graph with $<m$ edges and an additional set S of edges that must belong to the MCST.*

We are now ready for the proof. We are given a graph with m edges and a set of edges S that belongs to the MCST. We need to find a way to ''handle'' one edge. Let's try again the edge with the minimal cost, call it e. If adding e to S creates a cycle then e cannot belong to the MCST since S is known to in the MCST and the MCST cannot contain a cycle. If, on the other end, e does not create a cycle then, by essentially the same proof of the minimal cost edge above, e must belong to the MCST. Both cases take care of e and reduce the problem to one with $<m$ edges. The ''proof'' is completed by induction. □

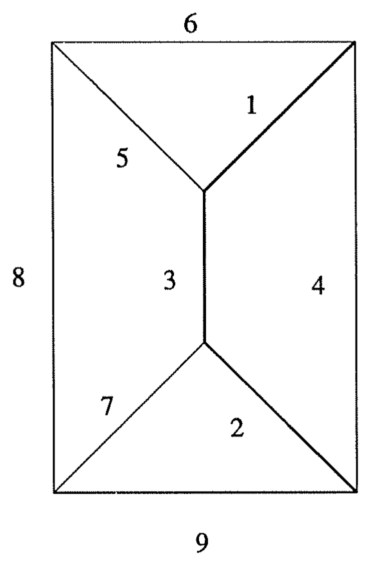

Figure 3: An example of the MCST algorithm

Implementing the algorithm efficiently requires good data structures. As we said before, we omit this part of the design in this paper. An example is given in Figure 3. The edges marked in bold are the ones that belong to S. The edge labeled 4 is the next to be considered, but it obviously cannot be selected. The edge labeled 5 should be selected since otherwise it can replace another edge with higher cost.

This technique is commonly called the ''greedy algorithm,'' since a new selection is made iteratively by taking the best possible current choice without regarding the global picture. In this case the greedy algorithm leads to the best solution, but, very often, it is only a heuristic. Using appropriate data structures, the running time of this algorithm is $O(m \log m)$, where m is the number of edges in the graph. The fastest known asymptotic running time for MCST (using some sophisticated data structures) is $O(m\ \beta(m,n))$, where $\beta(m,n) = \min\ \{i \mid \log^{(i)} n \le m/n\}$ [FT84].

6. Choosing the induction sequence wisely

Many times it is not straightforward to discover how to achieve the reduction in problem size. We have already seen reductions by removing a vertex with small degree, an edge with small cost, and by dividing a set of points in the plane by a vertical line. These examples include going from n to $n-1$ (the most common case), and from n to $n/2$. It is not necessary to choose an arbitrary element or an arbitrary set of elements. Sometimes it is worthwhile to make a very specific choice, for example the maximal element according to some constraints. Anything that will reduce the size of the problem is appropriate. The choice should be made very carefully. Many times this is the only hard part of the problem and once the right choice is made the rest is easy. This is extremely important in mathematics. One never jumps into an induction proof without thinking first how to choose the induction sequence. As expected, it is also very important in algorithm design as is shown in the following examples.

6.1. Cutting an Eulerian planar graph without dropping pieces

This problem involves a set of parts, laid out on a metal plate, which need to be cut in a way that optimizes certain objectives [MI84]. In this example we simplify the problem and concentrate on a very special limited objective. First we consider the layout as a planar graph and the problem becomes one of traversing all the edges of a graph in a certain order. Second, for simplicity we assume that the graph is Eulerian, which means that given any vertex v there exists a path that starts and ends at v and includes every edge exactly once. (The condition for the existence of such a path – discovered by Euler in 1736 – is that the degree of each vertex is even.) Third, we consider, as our only objective, cutting the graph without ''lifting'' the cutter in such a way that each piece is ''dropped'' (i.e., all its outer edges are cut) only after all its interior edges are cut. This objective is important since dropped pieces that require further cuts need to be handled separately. Our problem is to design an algorithm that generates the right order of cutting (see Figure 4). The reader is encouraged to try to solve this problem (a paper and a scissor are perfect design tools) before reading further; it is not straightforward.

Choosing the right induction sequence is crucial for this problem. We have to be extra careful here because the conditions of the problem are not hereditary. That is, a valid solution to a subproblem may not be part of a valid solution to the whole problem. The addition of an edge may change the whole order of cutting since we have to make sure that this edge is cut before all the pieces that contain it are completely cut. It turns

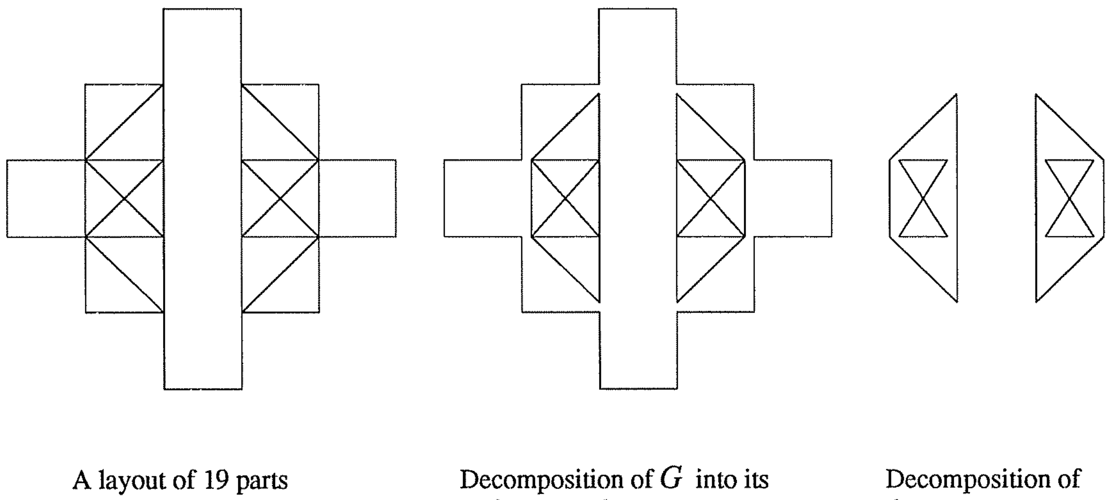
A layout of 19 parts
Decomposition of G into its perimeter and two components
Decomposition of the components

Figure 4: Example of the cutting problem

out that the best way to proceed with the induction in this case is to remove not one edge or one vertex but the whole perimeter of the graph.

Induction hypothesis: *We know how to solve the cutting problem for graphs with $<m$ edges.*

Consider a graph $G=(V,E)$ with n vertices and m edges that satisfies the conditions of the problem. Let P be the closed path that follows the perimeter of the graph as it is laid in the plane (see Figure 4). Denote the graph that results from removing P by G^1. We cannot apply the induction hypothesis yet since G^1 may not be connected. Let the connected components of G^1 be $G_1^1, G_2^1, \ldots, G_k^1$. Each component is Eulerian since by removing P, which is a cycle, the degree of each vertex is either unchanged or reduced by an even number. By the induction hypothesis we know how to cut each G_i^1 in a valid order.

The reason we chose the perimeter is because its edges can be cut last. Cutting edges inside the perimeter does not affect the pieces that border the outside. To cut G we start with any vertex of P and cut along the edges of P until the first vertex v_j belonging to some component G_j^1 is encountered. Each such component must have at least one vertex in common with P since G was connected before removing P. We then continue by cutting G_j^1 completely (we know how to do it by induction). Since we have only traversed the perimeter of G, and v_j was the first common vertex of G_j^1 and P, no piece of G that does not belong to G_j^1 can be cut at this stage. Hence, all the conditions are satisfied relative to G. By the Eulerian property after we cut G_j^1 we end at the same vertex v_j. We then continue with the same process, i.e., cutting P until the next component is met and proceeding by induction, until the whole graph is cut. The final cutting order is illustrated in Figure 5.

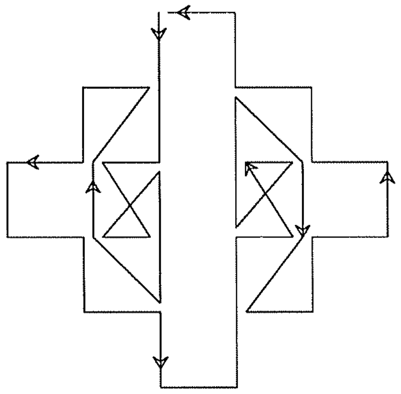

Figure 5: The final cutting order

The proof is almost complete except for one deficiency. We have applied the induction to traverse the smaller components not in an arbitrary way. We always started the traversal from a vertex on the boundary of the components. Can we always do that? The induction hypothesis only states that traversal is possible. We need to strengthen it and include the statement that the traversal can start from any vertex on the boundary. Since we started traversing G from an arbitrary vertex on P this stronger hypothesis is valid for the proof. □

6.2. All shortest paths problem

A *path* in a graph is a sequence of edges. Given a graph (directed or undirected) with n vertices such that there is a cost associated with each edge, we want to find the minimal cost paths between all pairs of vertices. Since we usually think of the costs as corresponding to distances this problem is called the *all pairs shortest paths problem* (see for example [AHU74]). For simplicity we discuss only how to find the costs of the shortest paths rather than the paths themselves. We assume that the graph is directed; the same arguments hold for undirected graphs.

As usual, let's start with straightforward induction. We can either induct on the edges or on the vertices. What is involved in terms of shortest paths in adding a new edge, say (u,w), to a graph? First, the edge may form a shorter path between u and w. Furthermore, there may be other shorter paths that use (u,w). In the worst case, we need to check, for every pair of vertices v_1 and v_2 whether the cost of the shortest path from v_1 to u plus the cost of (u,w) plus the cost of the shortest path from w to v_2 is shorter than the known path. Overall, for every new edge we may have to make $O(n^2)$ checks. Since the number of edges may be up to $O(n^2)$ this leads to an $O(n^4)$ algorithm.

What is involved in terms of shortest paths in adding a new vertex v to a graph? We first need to find the costs of the shortest paths from v to all other vertices and from all other vertices to v. The shortest way of getting from v to say u is the minimum, over all vertices w adjacent to v, of the costs of the edge (v,w) plus the shortest path from w to u (which is already known by induction). The shortest way of getting from u to v can be found similarly. We are not done yet. We still have to check for any pair of vertices whether there exists a shorter path between them using the new vertex v. For any vertices u and w we check the cost of getting from u to v plus the cost of getting from v to w and compare to the previously known shortest path. Overall it takes at most $3n^2$ comparisons for each added vertex for a total of $O(n^3)$ algorithm. The induction by vertices is thus better than the induction by edges, but there exists a better yet induction method.

The trick is to leave the number of edges and vertices fixed, and restrict the types of paths. The induction addresses the removals of these restrictions on the paths until, at the end, all possible paths are considered. We label the vertices from 1 to n. A path from u to w is called a *k–path* if, except for u and w, the highest labeled vertex on the path is labeled k.

Induction hypothesis: *We know the costs of the shortest paths between all pairs of vertices provided only k-paths, k<m, are considered.*

The base of the induction is $m=1$, in which case only direct edges can be considered and the solution is obvious. Assume the induction hypothesis for m and consider $m+1$. We now have to consider the shortest m-paths between all pairs of vertices, and check whether they improve on the k-paths for $k<m$. Denote by v_m the vertex labeled m. Any shortest m-path must include v_m exactly once. The shortest m-path between u and w is the shortest k-path (for some $k<m$) between u and v_m appended by the shortest k-path between v_m and w. By induction we already know the cost of all shortest k-paths for $k<m$, hence we only need to sum two costs to find the shortest m-path. We now compare it to the the cost of the previously known shortest k-path between u and w ($k<m$). There is at most only one sum and one comparison per pair, and the induction sequence is of length n. The total number of additions (and comparisons) is thus at most n^3. This algorithm is not only faster (by a constant factor) than the straightforward induction on vertices, but it is also extremely simple to program. The algorithm is given below.

```
Initially cost($x,y$) := cost of the edge ($x,y$) if it exists, or $\infty$ otherwise ;
**for** $m$ := 1 to $n$ do  { the induction sequence }
    **for** all pairs of vertices $x$ and $y$ **do**
        **if** cost($x,v_m$) + cost($v_m,y$) < cost($x,y$) **then**
            cost($x,y$) :=  cost($x,v_m$) + cost($v_m,y$) ;
```

□

This algorithm is due to Floyd [Fl62] and it is very similar to Warshall's transitive closure algorithm (see for example [AHU74]). Both algorithms are examples of a technique known as *dynamic programming*. In dynamic programming one usually solves all subproblems of smaller size, stores the solutions, and uses them in an organized and efficient manner to compute larger subproblems. Thus, dynamic programming is another example of our inductive approach.

In many cases it is worthwhile to spend some time and effort to discover the best induction sequence. A good simple example of this is binary search. Consider binary search in terms of divide and conquer. The set is divided into two subsets and the search continues recursively. To minimize the cost we would like to search only one subset and eliminate the other one. If we divide the set arbitrarily we cannot eliminate anything. Instead, we first sort the set, then we compare the search key to the middle set element, and eliminate accordingly. Interpolation search [Kn73b] carries this approach even further. Instead of comparing always to the middle element, a computation (interpolation) is performed to find the most likely place where the search key occurs, assuming that the elements are randomly distributed. In both cases spending more time to perform a better recursion pays off in the end.

Another very elegant example of this approach is an algorithm by Kirkpatrick and Seidel [KS86] for finding the convex hull of a set of points in the plane. Describing the algorithm is beyond the scope of this paper; we only highlight one aspect of it. The algorithm uses a variation of divide and conquer. As usual, the problem is first divided into two almost equal subproblems. Then, a special algorithm is used to figure out how to combine the solutions before they are even found! This algorithm can sometimes eliminate a substantial part of the subproblems making the recursive computation faster. Kirkpatrick and Seidel name this special divide and conquer technique ``marriage before conquest''.

7. Double induction

Double induction is another common variation of induction. In many cases the statement of the theorem depends on several parameters and, as a result, the induction has to consider all of them. Instead of considering all the parameters at the same time it is sometimes easier to consider them separately. The induction hypothesis advances in two or more different steps depending on which parameter is growing. If, for example, the theorem involves graphs, then there may be one hypothesis for constant number of vertices and growing number of edges and another one for growing number of vertices. Generally, it is possible to define a new parameter, which depends on all the other parameters, whose growth defines the right order of induction. Finding such a parameter usually involves trying each original parameter separately until the right combination is found. In our analogy, the right combination is the one that makes the algorithm the most efficient. It is thus important to have the flexibility of different possible orders of induction from which to choose the best combination. The following example illustrates this principle.

7.1. Intersections of line segments

Given a set of n horizontal and m vertical line segments in the plane, we want to find all intersections among them. This problem is important for example in the design of VLSI circuits [Ul84]. A circuit may contain hundred of thousands of ''wires'' and the designer has to make sure that there are no unexpected intersections. We assume, for simplicity, that line segments of the same type (i.e., horizontal or vertical) do not overlap. The induction can be defined for example in terms of the number of horizontal lines.

Induction hypothesis: *We know how to report all intersections among a set of* $<n$ *horizontal and arbitrarily many vertical line segments in the plane.*

Given n horizontal line segments, take for example the top horizontal line and compare it against all vertical lines. This is obviously correct but it is too expensive. The number of comparisons is $O(nm)$. We want it to be proportional to the actual number of intersections (since we have to spend at least that much time just reporting them) plus possibly some amount for constructing the data structures.

We start by trying to strengthen the hypothesis. We concentrate on information about the line segments that we can gain without much additional cost. For example, if the n'th horizontal line segment h_n is the top one then we may save comparisons by eliminating those vertical line segments whose top endpoint is lower than the y-coordinate of h_n. These vertical line segments need not be compared anymore. Here is the first attempt at strengthening the hypothesis.

Induction hypothesis: *We know how to report all intersections among a set of* $<n$ *horizontal and arbitrarily many vertical line segments and eliminate all vertical line segments whose top endpoint is below the top horizontal line segment.*

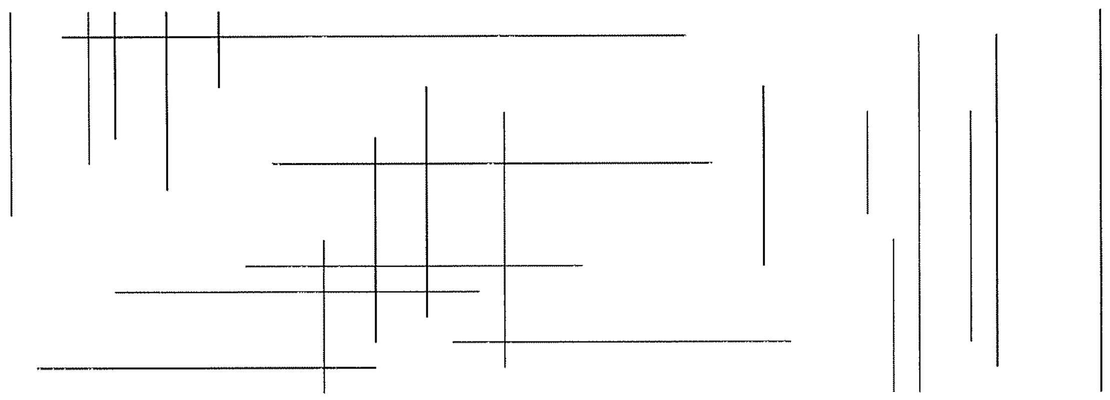

Figure 6: Horizontal and vertical line intersection

It is not hard to ''prove'' this modified induction. We again take the top horizontal line segment h_n, compare it against all the remaining vertical line segments and eliminate those whose top endpoint is too low. While this modification may save many comparisons the number of comparisons is still $O(mn)$ in the worst case. There are two reasons for that. First, there may be many vertical line segments whose x-coordinate is out of the range of the horizontal line segments (see the right side of Figure 6). These segments may not intersect but they cannot be simply eliminated. We deal with this problem later. Second, although we eliminated the vertical line segments that are too low we have not eliminated those that are too high. Consider the base case. The first horizontal line segment will have to be compared against all the vertical ones. It would be much better if we could compare it only to those vertical line segments whose bottom endpoint is below it (see the left side of Figure 6).

We have to consider the case of adding a new horizontal line and the case of adding a new vertical line separately. Being able to do it separately gives us extra flexibility. The principle of the double induction, in this case, is to find the optimal order in which the induction should be carried out. The problem we had with the efficiency of the previous algorithm was that ''high'' vertical line segments were considered too early. When a new horizontal line segment h_n is considered we would like to compare it only against those vertical line segments whose bottom endpoint is below h_n and whose top endpoint is above h_n. In other words, given a vertical line segment we should start considering it only after its bottom endpoint is below the current horizontal line segment, and stop considering it when its top endpoint is below the current horizontal line segment. This suggests ordering the different line segments according to their y-coordinates. Each horizontal line segment will appear once, and each vertical line segment will appear twice — once for its bottom endpoint and once for its top endpoint. Intersections will be checked when a new horizontal line segment is encountered. The checking will be only against a list of vertical line segments that are still *candidates* for intersection. This candidate list will be maintained by the algorithm.

Induction hypothesis: *Given a set of horizontal and vertical line segments with a total of $<k$ distinct y-coordinates, we know how to report all intersections and maintain a list of vertical line segments that are still candidates for further intersections.*

There are 3 cases for handling the k'th y-coordinate (which is the highest one):

1. This is a top endpoint of a vertical line segment — remove it from the list of candidates.
2. This is a bottom endpoint of a vertical line segment — add it to the list of candidates.
3. This is a horizontal line segment — check intersections with all the vertical line segments in the list of candidates.

The algorithm is now complete, and for most cases the number of comparisons is reduced substantially. However, in the worst case this algorithm still requires $O(mn)$ comparisons because of the problem mentioned above. We have eliminated the need to compare against vertical line segments that are too high or too low, but there may still be too many vertical line segments to the side of the horizontal line segments.

Given a horizontal line segment we want to minimize the number of comparisons to vertical line segments in the candidate list. But since the candidate list contains only vertical line segments that are in the right range for the y-coordinates, we need only check the x-coordinates. As a result, this is now a one dimensional problem. We need to check which of the vertical line segments in the candidate list has an x-coordinate that is in the x range of the horizontal line segment being considered. This problem is called a *one-dimensional range query*. One way to solve it is to sort the vertical line segments according to their x-coordinates, find the closest one to the left endpoint of the horizontal line segment, and scan the sorted list until an x value greater than the right endpoint is reached. The running time is proportional to the sorting time plus the search time plus the number of intersections. We do not really need to sort every time a new segment is encountered. We only need a data structure that allows us to insert a new element, delete an element, and perform a search and a linear scan as described above.

This is a basic problem in data structures. Fortunately, there are several data structures, for example balanced trees, that can support insert, delete, and search in $O(\log n)$ per operation (n being the number of elements in the candidate list) and linear scan in time proportional to the number of elements found. The final induction hypothesis is the following.

Induction hypothesis: *Given a set of horizontal and vertical line segments with a total of $<k$ distinct y-coordinates we know how to report all intersections and maintain a sorted list of vertical line segments (according to their x-coordinates) that are still candidates for further intersections.*

Overall, our algorithm costs $O(n \log n)$ for data structure operations plus an additional cost proportional to the number of actual intersections. This is the best possible in the worst case. □

The technique presented above is known as *plane sweep* (one can think of the induction sequence as following a horizontal line that sweeps the plane). This algorithm is due to Bently and Ottmann [BO79] (see also [PS85]). A second approach to this problem is divide and conquer [GW84]. It follows the strengthening of the induction approach.

Another example for the use of double induction in algorithm design is *multidimensional divide and conquer* [Ben80]. This technique applies to problems involving n points in k-dimensional space (i.e., n vectors). The induction is applied both to the number of points and to the dimension. One assumes the solution for $<n$ points in k-dimensional space *and* for n points in $k-1$-dimensional space. The latter assumption is usually needed in order to combine two $n/2$ solutions in k dimensions.

8. Reversed induction

This is a little known technique that is not often used in mathematics but is often used in computer science. The idea is to first use an infinite set S (e.g., $S = \{2^k\}$, k=1,2,..) as the base case for the induction. That is, prove that the theorem $P(n)$ holds for all values of n which belong to S. Then go ``backwards'' proving that the validity of $P(n)$ for n implies its validity for $n-1$. A very good example of the use of this technique is the elegant proof (due to Cauchy) of the arithmetic mean vs. geometric mean inequality (see for example [BB61]). Usually in mathematics it is not easier to go from n to $n-1$ than it is from $n-1$ to n, and it is much harder to prove an infinite base case rather than a simple one. When designing algorithms on the other hand it is almost always easy to go from n to $n-1$, that is, to solve the problem for smaller inputs. For example, one can introduce ``dummy'' inputs that do not affect the outcome. As a result, it is sufficient in many cases to design the algorithm not for inputs of all sizes, but only for sizes taken from an infinite set. The most common use of this principle is designing algorithms only for inputs of size n which is a power of 2. It makes the design much cleaner and eliminates many ``dirty'' details. Obviously these details will have to be resolved eventually. But it is much easier first to solve the main problems and worry about details later. We used the assumption that n is a power of 2 in the closest pair problem and we will use it again later on.

9. Maximal counterexample

A distinctive and powerful technique for proving mathematical theorems is by assuming the contrary and finding a contradiction. Usually this is done in a completely non-constructive manner, which is not very helpful in our analogy. Sometimes though the contradiction is achieved by a procedure similar to induction. The idea is as follows: First, it is shown that the theorem holds for small cases. Second, a contradiction to the theorem is assumed. Third, since the theorem holds for some instances and it is assumed that it does not hold for all instances, the maximal instance (under some criteria) for which the theorem holds is considered. The final and main step is to present a contradiction, usually to the maximality assumption. We present one example in which this technique is very helpful in designing algorithms.

9.1. Perfect matching in very dense graphs

A *matching* in an undirected graph $G=(V,E)$ is a set of edges that have no vertex in common. (The edges serve to match their two vertices, and no vertex can be matched to more than one other vertex.) A matching with n edges in a graph with $2n$ vertices is called a *perfect matching* (it is obviously optimal). Finding perfect matchings, and in general maximum size matchings, is very important for various applications [GM84]. In this example we consider a very restricted case. We assume that there are $2n$ vertices in the graph and all of them have degrees of at least n. It turns out that under these conditions a perfect matching always exists. We first present the proof of this fact, and then show how to modify the proof to get an algorithm for finding a perfect matching.

The proof is by maximal counterexample [Lov79]. Consider a graph $G=(V,E)$ such that $|V|=2n$ and the degree of each vertex is at least n, and assume that a perfect matching does not exist. Consider the matching M in G with the maximum number of edges. $|M| < n$ by the assumption, and obviously $|M| \geq 1$ since any edge is by itself a matching. Since M is not perfect there are at least 2 non-adjacent vertices v_1 and v_2 which are not included in M (i.e., they are not incident to an edge in M). These two vertices have at least $2n$ distinct edges coming out of them. All of these edges lead to vertices that are covered by M since otherwise such an edge could be added to M. Since the number of edges in M is $<n$ and there are $2n$ edges from v_1 and v_2 adjacent to them, at least one edge from M, say (u_1,u_2), is adjacent to 3 edges from v_1 and v_2. Assume, without loss of generality, that those 3 edges are (u_1,v_1), (u_1,v_2), and (u_2,v_1). It is easy to see that by removing the edge (u_1,u_2) from M and adding the two edges (u_1,v_2), and (u_2,v_1) we get a larger matching, contradicting the maximality assumption (see Figure 7).

It may seem at first that this proof cannot yield an algorithm since it starts by taking the matching with maximum cardinality. Had we known how to find such a matching we could have solved the problem. However, the steps of the contradiction work for any matching; they present a contradiction only for the maximum matching. We can start by taking any matching, for example find one through a greedy algorithm that simply adds edges until no more edges can be added, and extend this matching using the ''proof''. By the proof the matching can be extended to a perfect matching.

The main reason that the contradiction proof is helpful here is because it does not really require a matching with maximum cardinality; it is sufficient to consider a *maximal* matching, namely a matching that cannot be extended by simply adding more edges. The proof shows how to extend such maximal matchings. In general, one should look for a simple, possibly greedy, algorithm to find a subsolution with a certain property, and then attempt to extend the solution.

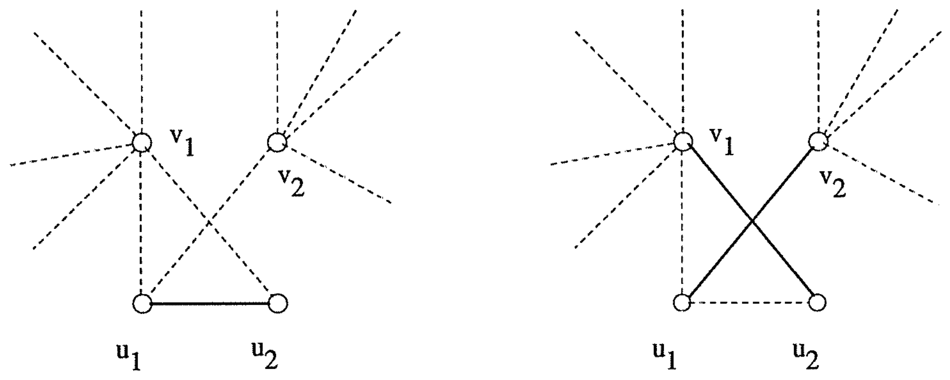

Figure 7: Extending a matching

10. Other proof techniques

We concentrated in this paper on proof techniques that are based on induction. There are many other techniques and more room for analogies, some obvious, and some not so obvious. For example, many mathematical theorems are proved by series of lemmas, and this corresponds directly to the idea of modular design and structured programming. The idea of proof by contradiction has analogies in the design of algorithms. It would be very interesting to study analogies to proofs ''by similar arguments.'' There are undoubtedly more such examples and we hope to continue this work. In this section we briefly explore two more proof techniques, unrelated to induction, which have analogous counterparts in the design of algorithms.

10.1. Probabilistic algorithms

Using probability to prove combinatorial properties is a very powerful technique. Erdös and Spencer present many such examples [ES74]. In a nutshell, the idea is to prove that among a set of objects the probability that an object has certain properties is greater than zero, which is an indirect proof that these properties occur. This method translates to algorithms in the following way. Suppose that we are searching for an object with certain properties, and we know of a probabilistic proof that it exists. We try to follow the probabilistic proof, generate random events when appropriate, and find the object with some positive probability. We can repeat this process many times until we succeed. If the probabilities work in our favor this can be a very effective algorithm. Probabilistic algorithms have been receiving a lot of attention since the discovery of probabilistic primality testing algorithms by Rabin [Ra76] and Solovay and Strassen [SS77]. In this short section we highlight the method by a simple example.

Let S be a large fixed set. The input to the problem is a collection of subsets of S, $S_1, S_2, \ldots, S_k$, each containing exactly r elements. The number of subsets k, satisfies $k \le 2^{r-2}$. We want to color each element of S with one of two colors, red or blue, such that each subset S_i contains at least one red and one blue element. Such coloring is called a *valid* coloring.

It is not even clear that the above coloring is always possible. We present a very simple probabilistic algorithm which is adapted from a probabilistic proof of existence of such coloring. The algorithm is almost as simple as possible:

> Take every element of S and assign it color red or blue at random (with probability 1/2) independently of the other elements.

This algorithm obviously does not always lead to a valid coloring. Let's calculate the probability of success. The probability that all elements of S_i are colored red is 2^{-r}. The probability that at least one of the k subsets is colored only red is no more than $\sum_1^k 2^{-r} = k2^{-r} \le 1/4$ (since we have a bound on k). Hence the probability that a random coloring is valid is at least 1/2 (since there is also no more than a 1/4 probability of a subset entirely colored blue). This is a proof that a valid coloring always exists (otherwise the probability must be exactly zero). It is also a very good algorithm. The expected number of times we need to run the algorithm to get a valid coloring is 2. Each assignment of colors can be easily tested for validity.

This is a very simple application of probabilistic methods. They are rarely so simple. (The general problem of finding a valid coloring when the subsets are arbitrary is NP-Complete [Lo73].) The type of algorithm presented above is known as a *Las Vegas* algorithm. This is a probabilistic algorithm that never gives a wrong result, but its running time may be arbitrarily long (although the expected running time is short). In contrast, a *Monte Carlo* algorithm is one that may give a wrong result with very small probability (although its running time can be bounded). That is, there is no way to efficiently validate the results of the algorithm. The algorithms for primality testing are Monte Carlo algorithms.

10.2. Checking all assumptions

A common technique that is very important in proving almost any theorem is to search the proof thoroughly for assumptions or steps that are not essential. Removing such assumptions results in a better theorem. It is also sometimes an indication that the proof may be wrong. Quoting Polya and Szego [PS27]: ''One should scrutinize each proof to see if one has in fact made use of all the assumptions; one should try to get the same consequence from fewer assumptions... and one should not be satisfied until counterexamples show that one has arrived at the boundaries of the possibilities.'' The same is true for algorithms. It sounds simple, but many times it is not.

Finding the first and second largest elements in a set

Given an unsorted set of n elements (n a power of 2 for simplicity), the problem is to find the first and second largest elements. (This problem was first suggested, in the context of arranging tennis tournaments, by Lewis Carroll; see [Kn73b].) For this example we are looking for an algorithm that minimizes only the number of comparisons of elements from the set (corresponding to the number of games in the tournament). We try the usual divide and conquer by dividing the set S of size n into two subsets P and Q of size $n/2$. By a straightforward induction we know the first and second largest elements of P and Q, denote them by p_1, p_2, and q_1, q_2 respectively. It is easy to see that two more comparisons are necessary and sufficient to find the first and second largest elements of S. That is, compare the two maximals p_1 and q_1, and then the ''loser'' against the second largest of the ''winner'' (see Figure 8 – the dashed lines correspond to the comparisons). This approach leads to the recurrence relation $T(2n)=2T(n)+2$, $T(2)=1$, whose solution is $T(n)=3n/2-2$. If the two comparisons are necessary, how can we improve on the total number of comparisons? Looking carefully at the comparisons in Figure 8 one sees that q_2 will not be used further in the algorithm and hence the computation leading to its discovery was unnecessary. However, until we compare p_1 to q_1, we do not know whether p_2 or q_2 can be ignored. If we knew which subset was going to ''lose'' then we could use the regular maximum finding algorithm for this subset saving many comparisons. The answer: delay the computation of the second largest to the end. Keep only a list of candidates for second largest. Here is the updated induction hypothesis:

Induction hypothesis: *Given a set of size $<n$ we know how to find the maximum element and a ''small'' set of candidates for the second maximum element.*

We have not defined a value for ''small'' in the hypothesis. We will discover the appropriate value when we develop the algorithm.

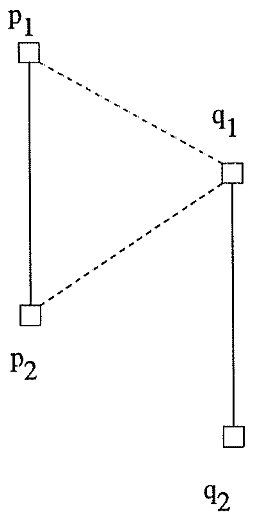

Figure 8: Finding the largest and second largest elements

The algorithm proceeds as follows. Given a set S of size n we divide it into two subsets P and Q of size $n/2$. By the induction hypothesis we know the largest elements of the two sets, p_1 and q_1, plus a set of candidates for the second largest, C_P and C_Q. We compare p_1 and q_1 and take the largest, say p_1, to be the maximum of S. We then discard C_Q since all elements of C_Q are less than q_1 which is at most the second largest, and add only q_1 to C_P. At the end we get the maximum and a set of candidates from which we choose the second maximum directly. The number of comparisons for finding the maximum satisfies the recurrence relation $T(n) \le 2T(n/2)+1$, $T(2)=1$, which implies that $T(n)=n-1$. It is easy to see that the size of the candidate set is $\log_2 n$, hence it takes $\log_2 n-1$ more comparisons to find the second largest. The total number of comparisons is $n-1+\log_2 n\ -1$, which, incidently, is the best possible (see [Kn73b]).

11. Conclusions

We have presented a methodology for explaining and approaching the design of combinatorial algorithms. The benefits of having such a general methodology are twofold. First, it gives a more unified ''line of attack'' to an algorithm designer. Given a problem to solve, one can go through the techniques described and illustrated in this paper and attempt a solution. Since these techniques have something in common (namely mathematical induction) the process of trying them all can be better understood and easier to carry out. Second, it gives a more unified way to explain existing algorithms. It gives a different point of view. It allows the student to be more involved in the creative process. The proof of correctness of the algorithm becomes a more integral part of the description.

It remains to be seen whether this methodology can be adopted in the classroom and whether it can help to design better solutions to algorithmic problems. Judging from my limited experience, I am hopeful.

A natural question arising from this paper is to find techniques developed originally for computer science that can be useful for proving mathematical theorems. One interesting example appears in [MT85], where a relationship between some measure of non-convexity of n-dimensional shapes and the complexity of *non-deterministic* algorithms is shown. The motivation was to establish lower bounds on the complexity of non-deterministic algorithms, but one can also derive upper bounds on the non-convexity of some shapes by exhibiting fast non-deterministic algorithms.

Acknowledgements

Thanks to Rachel Manber for numerous inspiring conversations about induction and algorithms, and to Darrah Chavey for many helpful comments.

REFERENCES

[AHU74]
A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms,* Addison Wesley, 1974.

[BC85]
J. L. Bates and R. L. Constable, ''Proofs as Programs,'' *ACM Transactions pn Prog. Lang. and Syst.,* Vol. 7, pp. 113-136, (January 1985).

[BB61]
E. Beckenbach and R. Bellman, *An introduction to inequalities,* New Mathematical Library, Random House, 1961.

[Ben80]
J. L. Bently, ''Multidimensional divide-and-conquer,'' *Communication of the ACM,* Vol. 23, pp. 214-229, (April 1980).

[BO79]
J. L. Bently and T. Ottmann, ''Algorithms for reporting and counting geometric intersections,'' *IEEE Trans. on Computers,* pp. 643-647, (Sept. 1979).

[Co86]
R. L. Constable et al, *Implementing Mathematics with the Nuprl Proof Development System,* Prentice Hall, 1986.

[Di76]
E. W. Dijkstra, *A discipline of programming.* Prentice Hall, Englewood Cliffs, 1976.

[ES74]
P. Erdös and J Spencer, *Probabilistic methods in combinatorics.* Academic Press, 1974.

[Fl62]
R. W. Floyd, ''Algorithm 97: shortest paths,'' *Communications of the ACM,* **5**:6, pp. 345, (1962).

[FT84]
M. L. Fredman and R. E. Tarjan, ''Fibonacci heaps and their uses in improved network optimization algorithms,'' *25th Annual Symposium on Foundations of Computer Science,* pp. 338-346, (1984).

[GM84]
M. Gondran and M. Minoux, *Graphs and Algorithms,* John Wiley & Sons, 1984.

[Gr81]
D. Gries, *The science of programming,* Springer–Verlag, 1981.

[GW84]
R. H. Guting and D. Wood, ''Finding rectangles intersections by divide-and-conquer,'' *IEEE Trans. on Computers,* pp. 771-775, (July 1984).

[KS86]
D.G. Kirkpatrick and R. Seidel, ''The Ultimate Planar Convex Hull Algorithm?''

SIAM Journal on Computing, pp. 287-299, (February 1986).

[Kn73a]
D. E. Knuth, *The Art of Computer Programming,* Vol 1: Fundamental Algorithms, Second edition, Addison Wesley, 1973.

[Kn73b]
D. E. Knuth, *The Art of Computer Programming,* Vol 3: Sorting and Searching, Addison Wesley, 1973.

[Lov73]
L. Lovasz, ''Coverings and colorings of hypergraphs,'' *Proc. 4th Southeastern Conf. on Combinatorics, Graph Theory, and Computing,* Utilitas Mathematica Pub., Winnipeg, (1973), pp. 3-12.

[Lov79]
L. Lovasz, *Combinatorial problems and exercises,* North Holland, 1979.

[MI84]
U. Manber and S. Israni, ''Pierce Point Minimization and Optimal Torch Path Determination in Flame Cutting,'' *Journal of Manufacturing Systems*, Volume 3, No. 1, pp. 81-89, (1984).

[MT85]
U. Manber and M. Tompa, ''The Complexity of Problems on Probabilistic, Nondeterministic, and Alternating Decision Trees,'' *Journal of the ACM,* **32**, pp. 720-732, (July 1985).

[Po57]
G. Polya, *How to solve it,* Second Edition, Princeton University Press, 1957.

[PB85]
P.W. Purdom and C.A. Brown, *The Analysis of Algorithms,* Holt, Rinehart and Winston, 1985.

[PS27]
G. Polya and G. Szego, *Aufgaben und Lehrsatze aus der Analysis,* Vol I, Berlin, Springer, 1927, page 7.

[PS85]
F. Preparata and M. I. Shamos, *Computational Geometry,* Springer–Verlag, 1985.

[Ra76]
M. O. Rabin, ''Probabilistic Algorithms,'' in *Algorithms and Complexity, Recent Results and New Directions,* J.F. Traub Ed., Academic Press, New York, 1976, pp. 21-39.

[Sed83]
R. Sedgewick, *Algorithms,* Addison Wesley, 1983.

[SH75]
M. I. Shamos and D. Hoey, ''Closest-point problems,'' in *16th Annual Symposium on Foundations of Computer Science,* 1975.

[SS77]
R. Solovay and V. Strassen, ''A Fast Monte-Carlo Test for Primality,'' *SIAM*

Journal on Computing, **6**, 1977, pp. 84-85.

[Ul84]
J. D. Ullman, *Computational aspects of VLSI,* Computer Science Press, 1984.