UNLIMITED

RSRE
MEMORANDUM No. 4468

ROYAL SIGNALS & RADAR ESTABLISHMENT

PRACTICAL COMPUTATIONAL COMPLEXITY

Author: D E Symonds

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

RSRE MEMORANDUM No. 4468

UNLIMITED

CONDITIONS OF RELEASE

0105886 303455

********** DRIC U

********** ***** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

Defence Research Agency Electronics Division

Memorandum 4468

Title: Practical Computational Complexity

Author: D Symonds

Date: April 1991

Abstract

Computational Complexity Analysis is a very powerful tool for use in designing algorithms. Unfortunately, the analysis can sometimes be misleading for the range of input with which the algorithm will actually be used.

The aim of this paper is to first give an overview of what computational complexity is, and how to analyse it, and to then explain what parts of complexity analysis can cause confusion, and how to avoid these mistakes. The paper also seeks to investigate ways of solving problems faster by sacrificing certain operating requirements.

INTENTIONALLY BLANK

1 Algorithms and Complexity

1.1 Algorithms

The concept of an algorithm as a method to solve a problem was first proposed formally by the Persian, Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî in approximately 850 AD, in his book 'Kitab al Jabr W'al-muqabala' (Rules of Restoration and Reduction). The word first appeared in a dictionary with it's current meaning in 1957, and has received it's widest use in the field of computer science, where algorithms are implemented as *programs*. An *algorithm* is defined as a method to solve a given *problem* step by step, (although with the advent of parallel computers and operating systems, it is likely that algorithms which carry out many steps at once will begin to be more widely used). A problem can be formally defined in terms of

- A *domain* - all possible data to which the problem could apply.
- An *instance* - a sublist from the problems domain.
- A *question* applicable to any instance in the domain.

We could therefore define an alphabetical sort problem relating to lists of two, two-character strings from the set "a,b,c" in the following manner:

```
Set of symbols           : {a b c}
Set of strings (domain)  : {aa ab ac ba bb bc ca cb cc}
Set of lists (instances) : {(aa aa) (aa ab) ... (cc cb) (cc cc)}
Question : Given any two element list of two character alphabetical strings
           from the set {aa ab ... cb cc}, what is the alphabetically sorted
           version of that list?
```

1.2 Estimated Running Time

To more effectively use algorithms to provide us with answers to problems, it would be useful to have a way of comparing algorithms to see which particular one best serves our purpose. If we assume that accuracy in our answers is the first requirement, the second requirement should be one of speed, that is: of two algorithms which will always solve a problem to the same degree of accuracy, the better one will be that which solves the problem in the shortest time. In order to provide a source of general information about an algorithm, running time is usually estimated as some function $f(n)$, where n will be some feature of the instance size. In the case of a sort problem, n would be the the number of elements in the list. Where there is more than one factor which could affect the run-time of an algorithm, n is accepted to be the factor which will generally vary most widely, over the instances. For sort data, another factor which may affect run-time is the *size*, in digits, of the largest possible element. This will tend to affect the run-time to a much lesser extent, though, than will the number of elements, since it will generally be more nearly constant, and thus n is taken to be the size of the list.

An example of an algorithm to solve the alpha-numeric sort problem might be the Bubblesort algorithm. The name is derived from the method: on every pass through the list the next largest item is dropped to a position where all items below it are larger, and the small items 'bubble' upwards towards the top of the list.

In its worst case, the algorithm loops from $i=1$ to $i=(n-1)$, making $n-i$ comparisons each time. Its estimated running time will thus be

$$\sum_{i=1}^{n-1}(n-i) \;=\; \sum_{i=1}^{n-1} n \;-\; \sum_{i=1}^{n-1} i \;=\; \frac{n}{2}(n-1)$$

This means that the running time is $\propto \frac{1}{2}(n^2 - n)$. To ensure that differences in calculation time in different machines are ignored in analysis, it is usual to count steps rather than machine cycles, and to use *order notation* or *O-notation* to express the running time. Order notation is a notation for functions, proposed by (Bachmann, 1892) which has been applied to computational complexity. Thus we can say that the running time of an operation is $O(f(n))$, meaning that there will be some constant c, such that the running time for an input of size n is bounded by $cf(n)$. c will change from machine to machine, and thus O-notation is used.

We can thus say that the run-time for Bubblesort will be $O(n^2 - n)$, but it is more usual to express it merely as $O(n^2)$, since there will always be a value of n at which the run-time will be so large that the difference to this run-time made by any extra, lower order terms of n will be smaller than errors in calculation due to limits in machine precision and will thus become insignificant. More formally, given a constant c_1, such that running time is $\leq c_1(n^2 - n)$, there is another constant c_2 (namely $2 \times c_1$) such that running time is $\leq c_2 n^2$, for all $n > 0$.

Bubblesort Algorithm

```
Input: An n-sized list of alpha-numeric items.
Output: An alpha-numerically sorted list of the input items.

for (i=(n-1) to 1 step -1)
 {
   for (j=1 to i)
    {
      if ( element(j) > element(j+1) )
       {
         swap element(j) and element(j+1)
       }
    }
   if ( no swaps occurred )
    {
      output list then finish
    }
 }
```

1.3 Worst-case Analysis

In an effort to improve run-time some algorithms make assumptions about the instance data (e.g. that the list is randomly ordered, in a sort problem). The run-time will depend, to a large extent on how these assumptions hold, because if the assumptions are wrong, the algorithm will often slow down tremendously. [1] This means that many clever algorithms can take a number of steps which varies by orders of magnitude,depending on the particular instance. This will introduce a statistical dependency in their running time. To attempt a description of computational complexity which is meaningful for any instance, it is usual to take a *Worst Case* analysis. Thus $f(n)$ will refer to the *largest* number of steps the algorithm could take for any n. To say that the Bubblesort Algorithm takes $O(n^2)$ run-time, then, implies that the worst possible run-time would occur in the case where n^2 swaps were made, which would correspond to a list originally sorted in reverse order. This method of analysis provides a guarantee that the algorithm will run within certain bounds.

[1] It is assumed that the algorithm will *always* seek to find the correct answer, and could not make an error if it's assumptions were wrong. If a short-cut in a sort algorithm has a (however slim) chance of resulting in an incorrectly sorted list, this algorithm does not solve the problem.

1.4 Polynomial Time

Algorithms can be placed on a scale by referring to their run-times as a function of their instance size. It would also be useful to have some method of drawing a line where we can say an algorithm has good[2] or bad run-time. It is usual to refer to an algorithm as good, with respect to run-time, only if the algorithms complexity is bounded by a polynomial, or slower-increasing function, of the instance size. For large n, any polynomial function will, eventually, always grow at a smaller rate than will an exponential function, such as 2^n or $n!$. Exponential[3] functions are therefore regarded as bad. Run-times such as $\log n$ or $n \log n$ are also regarded as good, since $\log n < n$ and $n \log n < n^2$. An algorithm whose run-time is bounded by a polynomial function over it's instance size is known as a *Polynomial-Time Algorithm.*

2 $\mathcal{P}$ and $\mathcal{NP}$

2.1 Decision Problems and Polynomial Time Reduction and Transformation

The considerations above lead to a classification of problems into the groups of easy or hard, depending on whether the fastest (worst case) algorithm which can solve them is good (Polynomial time or better) or bad (Exponential time).

To make a fair comparison between the run-times of algorithms pertaining to different problems, it is necessary to standardize the question posed by the problem in some way. One way this can be done is to restrict any theories relative to problems to *decision problems*, i.e. problems whose questions can be answered only with a *yes* or a *no* answer. This would seem to suggest that theories can only be applied to a very small set of problems, but this is not the case. Any problem can be reduced to a decision problem with only a polynomial change in time, by using a main algorithm to call a sub-algorithm which will solve the decision problem. We say that a problem A can be *polynomially reduced* to a problem B if there is an algorithm for A that calls B as a sub-algorithm, and the algorithm for A runs in polynomial time when the time for each call of B is counted as a single step. Thus, if there is a polynomial time algorithm for B, and A can be polynomially reduced to B, then there is a polynomial time algorithm for A. This implies the rule that, for two time functions, $f(n)$, and $g(n)$, time $O(f(n)) \times O(g(n)) =$ time $O(f(n)g(n))$. The bubblesort algorithm described above is essentially an algorithm using a call to a sub-algorithm to solve it's problem. The decision problem in this case is: *Is item j greater than item $(j+1)$?* The main bubblesort algorithm calls this smaller algorithm a polynomial number of times, counting each call as one step, and is thus said to run in polynomial time with respect to n.
There are two useful types of polynomial reduction:

- *Reductions which prove a problem is easy.* These reduce a problem to some known polynomial time problem.
- *Reductions which prove a problem is hard.* If a known *hard* problem will reduce to a problem of unknown complexity, that problem is also hard. Also, reducing a problem of unknown complexity to a provably hard problem will prove it to be hard.

A *polynomial time transformation* is a reduction between decision problems, where the algorithm calls the reduced form only once, and uses the answer as it's own.

2.2 $\mathcal{P}$ and $\mathcal{NP}$ Problems

The class $\mathcal{P}$ contains all the decision problems for which a polynomial time algorithm exists.

The class $\mathcal{NP}$ consists of those decision problems which, while thought not to be polynomial, have characteristics which make them seem easier than provably hard problems. Three different, but equivalent formulations have emerged over the last two decades to define these characteristics.

[2]The terms good, bad, easy and hard seem very subjective terms, but are in common usage in computational complexity literature, and are thus used here, for consistency.

[3]by Stirling's Formula, $x! = \sqrt{2\pi}x^{x+\frac{1}{2}} \exp(-x + \frac{\theta}{12x})$ where $x > 0, 0 < \theta < 1$, and is thus exponential

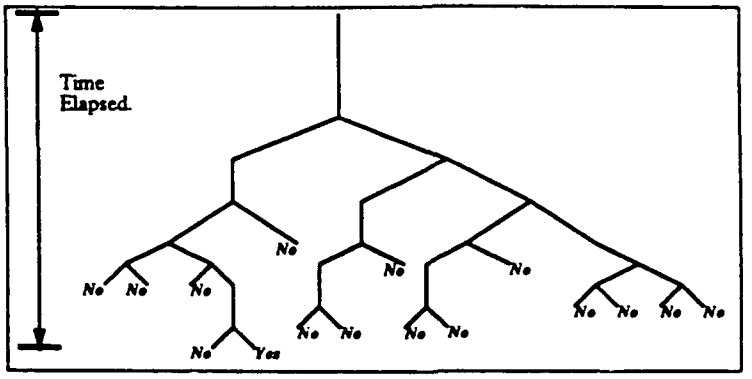


Figure 1: A possible path for a non-deterministic algorithm.

The first definition involves the *succinct certificate property*. A succinct certificate is a small mathematical object that establishes beyond doubt that the instance has a 'yes' answer. There must also be an algorithm which, knowing what the answer is, can *check* it in polynomial time. More formally, a decision problem A has the succinct certificate property if there is a certificate checking algorithm $C \in \mathcal{P}$, whose instances are of the form (I, S) where I is an instance of A and S is an object whose size is bounded by a polynomial function of the size of I, and such that I is a yes instance of A and there exists S such that (I, S) is a yes instance of C (Lawler *et al*, 1985).

The second criterion for a problem to be in the set of $\mathcal{NP}$ problems is that it is solvable in polynomial time by a *non-deterministic algorithm*. This is an algorithm which can, at any stage, branch into more than one copy of itself, each doing the same job, but using a different route. In the course of this branching, decisions are made at random, i.e. there is no indication of whether this is the best route to choose. This means that the algorithm branches out exponentially as a function of time, generating huge numbers of random routes, at least one of which should be the correct route. If any one of the branches thus created reports a 'yes' answer, then the overall answer of the algorithm will be 'yes' (see figure 1). If the number of steps taken by the path which reported an affirmative answer is bounded by a polynomial function of the instance size, then the algorithm has solved the problem non-deterministically, in polynomial time ($\mathcal{NP}$ for $\mathcal{N}$on-deterministic $\mathcal{P}$olynomial time). These algorithms, while useful theoretical tools, are unrealistic in practice.

The third criterion to show if a problem is in $\mathcal{NP}$, is if the problem is polynomial time transformable to an $\mathcal{NP}$ *complete* problem (see next sub-section).

2.3 $\mathcal{NP}$ Complete

A problem A is $\mathcal{NP}$ *complete* if

- $A \in \mathcal{NP}$
- All other problems in $\mathcal{NP}$ can be polynomial time transformed to A.

This means that an algorithm which can solve an $\mathcal{NP}$ complete problem can solve *any problem in $\mathcal{NP}$ with only a polynomially bounded difference in time*. Thus if an algorithm is found which can solve any one $\mathcal{NP}$ complete problem in polynomial time, then $\mathcal{NP} \equiv \mathcal{P}$, because all of the $\mathcal{NP}$ problems are transformable to an $\mathcal{NP}$ complete problem. To prove that a new problem A is $\mathcal{NP}$ complete, it is not necessary to go through the process of trying to transform every problem in $\mathcal{NP}$ to A, but merely to transform any known $\mathcal{NP}$ complete problem to A.

2.4 $\mathcal{NP}$ hard

Another class of problems, $\mathcal{NP}$ *hard*, are those problems (decision or otherwise) which, while they are *not known* to be in $\mathcal{NP}$, are similar to $\mathcal{NP}$ complete problems in that all $\mathcal{NP}$ problems are polynomial time *reducible* (the sub-algorithm is called a polynomial number of times to provide information to help solve a problem), but not necessarily transformable (the sub-algorithm is called once, and it's answer used as the final answer to the problem), to them. Thus if any of these problems could be solved in polynomial time, then $\mathcal{NP}\equiv\mathcal{P}$.

3 Practical Considerations

3.1 Worst Case Analysis

To analyse the computational complexity of an algorithm, using the methods described above, it is necessary to make several assumptions about what type of calculation is needed. One such assumption made in the theory of computational complexity is that the *worst case* analysis should be the path pursued in analysing complexity. This worst case analysis means that analysis should be based on the largest possible number of steps that an algorithm can take, over the instance size and that the best algorithm will be the one with the smallest worst case time function $f(n)$, assuming that all proposed algorithms solve the problem to the same degree of accuracy. This method is chosen to provide some guarantee of the complexity of the algorithms being analysed. Unfortunately it is quite possible for this method to overlook the case where an algorithm could have a few extremely rare and pathological cases with a large number of steps, which could overshadow a usually fast and efficient performance.

A *best case* analysis would be more inaccurate in it's assumptions since best cases are often even more pathological than worst cases. In it's best case, for example, the Bubblesort algorithm (section 1.2) would make $n-1$ comparisons, since the list would already be sorted.

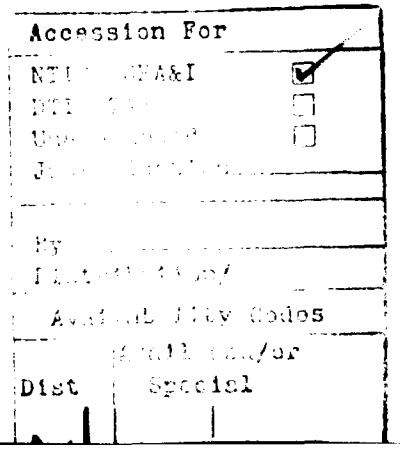
Another method of analysing complexity could be to make an *average case* measure, to analyse how the algorithm will perform in normal, everyday usage. Most clever algorithms will introduce some statistical dependencies, which frequently make it difficult, and sometimes impossible to provide a workable average case analysis. But this shouldn't necessarily mean that the average case be ignored completely, and solely the worst case analysis relied upon, when looking at new algorithms.

One of the fastest sorting algorithms known, the *Quicksort* algorithm for sorting lists, falls foul of a worst case analysis. Quicksort uses a recursive divide and conquer technique to split lists into smaller and smaller sections, which it then concatenates in the correct order until it ends with a sorted list.

Quicksort Algorithm

```
Input: An n-sized list of alpha-numeric items, list( n ).
Output: An alpha-numerically sorted list of the input items, sorted( n ).

qsort( list( n ) )
 {
   x = list[ (integer) n/2 ]
   for ( i=0 to i=n i<>x )
    {
      u[i] = list[i] where list[i] < x
      v[i] = list[i] where list[i] >= x
    }
   u2[] = qsort( u( |u| ) ) , v2[] = qsort( v( |v| ) )
   sorted = u2 ++ x ++ v2
 }
```



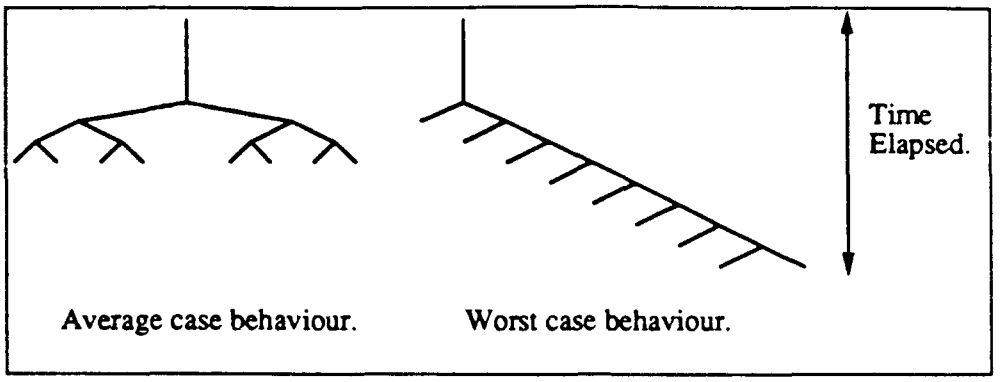


Figure 2: Worst and Average Case Behaviour of Quicksort

At each recursion, the algorithm begins by selecting the item from the centre of the unsorted sub-list. In it's worst case, where this item is either the smallest or largest in the list, n recursions, each involving n comparisons, would need to be made. This indicates that quicksort is an $O(n^2)$ algorithm (see Figure 2), and since there are other algorithms, including *Heapsort* (Wirth, 1986), which always run in $O(n \log n)$ time, quicksort should be categorized as a slow algorithm, and subsequently ignored.

Making an average case analysis of quicksort involves finding the expected position, in the sorted list, of any randomly chosen item in the unsorted list. Assuming that the initial data are chosen at random, and have a uniform distribution, we can expect the average position of any datum to be the median value of the range, which would put it in the centre of the sorted list. The average case is, therefore, that each new sub-list would comprise approximately one half of its parent, and this would mean that the number of divisions would be $\log n$. Since n comparisons are made for each division, our average case analysis gives us a complexity of $O(n \log n)$, the same order of complexity, on average, as is heapsort (see figure 2).

The worst case analysis then, tells us that quicksort is as slow as is bubblesort, and the average case analysis indicates that it is level with heapsort. Experimental evidence shows that quicksort is almost twice as fast as heapsort on typical sorting problems (Wirth, 1986). It is therefore advisable when examining new algorithms, when possible, to attempt an average case analysis to complement the worst case analysis. If it is impossible to determine beforehand what this average case time function will be, then tests can be carried out to determine the average case function experimentally.

3.2 Constants and Slow Growing Functions

Another assumption made by computational complexity theory is that a fast growing time function will always, at some point, overshadow a slow growing time function, so that constants and lesser functions of n can be ignored (See section 1.2). Thus, although an algorithm may run in $c(n^3 + 1000000n^2)$ time, where c is some constant, we say that it runs in $O(n^3)$ time. This is a correct assumption, because eventually (at $n > 1000000$, in this case) the n^3 term will always overshadow the n^2 term, but this value of n may be larger than any which we are likely to use in practise. This means that in the case of an extremely large constant for a lower order term of n than that usually deemed computationally dominant, it is possible that for all values of n which will ever be used, the lower order term will overshadow a higher. When analysing a new algorithm, therefore, we should bear in mind the range of n, and acknowledge that, within the useful range of n the fastest growing term of n may not be computationally dominant. Also, an algorithm with a low order of complexity may be slower than a high complexity algorithm, in the useful range of n.

An example of such an occurance is the process of matrix inversion. It can be shown that the usual processes for matrix inversion, Gauss-Jordan Elimination, LU-Decomposition, and Singular Value Decomposition, are all $O(n^3)$ (Press *et al*, 1988). There is a method of matrix inversion, however,

which is $O(n^{\log_2 7})$ (Strassen, 1969), and which is not generally used on matrices of size under about 10000 x 10000, so it would seem that the computationally 'better' method is being overlooked. When estimates are made on the size of matrix for which Strassen's algorithm would actually be faster, however, bearing in mind the extra number of additions which Strassen's algorithm makes (Press *et al*, 1988), it is found that for $n \leq$ about 10000 LU-Decomposition is faster than Strassen's Algorithm. This means that Strassen's method is better for large matrices, where it is in fact often used, but in the range of $n \leq$ about 10000, the $O(n^{\log_2 7})$ algorithm is slower than many $O(n^3)$ algorithms.

3.3 Other Variable Factors in Algorithms.

One of the premises of computational complexity is that, where there are several variables on which the complexity might depend, the variable which will tend to either be the most frequently changed, in usage, or that will increase more rapidly in the complexity function, should be the variable on which the run-time function is based, in complexity analysis. This means that, once analysis is completed for this variable, the others are ignored. The problem with this approach is that when the time comes to change one of the other variables, there is no guarantee that the time function is not going to grow unacceptably quickly. It may be useful, then, to analyse complexities for more than one variable of the algorithm, where we see that these variables are likely to be frequently changed in common usage.

4 Machine Factors and Reductions in Complexity

4.1 Reducing Complexity by Accuracy Versus Time

Once the complexity has been analysed for an algorithm, it is possible to estimate what size a problems instance can be before the problem becomes too big to be solved in an amount of time commensurate with the urgency of the problem. It is also quite feasible that the size of problem we wish to solve could be far out of the range of the time which we have available. We then need some other algorithm which solves the problem faster. For $\mathcal{NP}$ or $\mathcal{NP}$-complete problems, however, it is thought to be impossible to find a polynomial time algorithm, and any exponential time function is going to grow out of range very quickly, as n increases, so that new, still exponential, algorithms cannot improve greatly on this growth rate.

One way to counter this problem may be to make certain sacrifices in the accuracy we require in our answer, perhaps moving to polynomial time in doing so. A classic $\mathcal{NP}$-complete problem is the Travelling Salesman Problem (Lawler *et al*, 1985). The Travelling Salesman Problem (TSP) has an instance comprising of n cities and an $n \times n$ distance matrix pertaining to these cities, and a question which requires the shortest tour of the n cities, visiting each city only once.

The simplest algorithm for solving the TSP is the exhaustive search algorithm, try every permutation of the tour, and remember the shortest permutation.

This algorithm will always look at $(n-1)!$ permutations, and will take n steps in calculating the distance of the tour, which makes it $O(n!)$. This means that if three cities would take one second, four cities would take four seconds, and so on, until fifteen cities, which would take one hundred and fifteen years and sixty seven days. Obviously this algorithm is of no practical use for solving the TSP for more than just a few cities.

Another algorithm which will solve the TSP is the *branch and bound* algorithm.[4]

[4] Thanks to M. Bedworth who coded and tested the A* Algorithm for me.

A* Algorithm

```
Input: A set of n cities.
Output: An optimal cycle of these cities, visitng each only once.

/* to search a tree to depth I and branching factor J */

best_cost = infinity ;
search (1, 0) ;

search (i, current_cost)
{

 if (i=I)
 {
   if (current_cost<best_cost)
     best_cost=current_cost ;
 }
 else
 {
   for (j=1 to J)
     {
       if (current_cost+link_cost(i,j)+cost_remaining(i,j)<best_cost)
         search(i+1, current_cost+link_cost(i,j) ) ;
     }
 }

}
```

The branch and bound algorithm is quite difficult to analyse, in the average case, since it is statistically dependent. In it's worst case, that it has to check every path, it will perform an exhaustive search, and is thus $O(n!)$. In practise, I have found the complexity to be $O(x^n)$, where $x \approx 3$, meaning that it is the same order as is an exhaustive search (exponential time), with merely a smaller constant (See Footnote 3). In fact, in order to *solve* the TSP, all we can hope for is to keep lowering this constant, since the TSP's decision problem problem is $\mathcal{NP}$-complete (Lawler *et al*, 1985), and is not believed possible to be solved in less than exponential time.

Another way to approach the problem might be to risk losing some of the accuracy required by the problem, in exchange for a faster, possibly polynomial, algorithm, thereby changing the question to require a *good* tour of the cities, visiting each only once. This measure of the 'goodness' of the tour depends on the user, and on the extent to which they require to save time. Probably the *quickest* algorithm on any problem under several thousand cities, to construct a reasonable tour, relative to a random one, is the *greedy* algorithm. The algorithm is called greedy because it looks only for a local optimum, and does not attempt to optimise the global distance. The greedy algorithm can be stated simply: make the next city in the tour the nearest one which has not yet been visited. The greedy algorithm is $O(n^2)$, since it loops from $i = n - 1$ to $i = 1$ cities, making $n - i$ comparisons each time. The constant is very low. In my tests, the number of calculations made was $\approx \frac{1}{2}n^2$.

The error made by the greedy algorithm is obviously quite large, since the algorithm can wander about naively between the cities, until, for the last few cities, it will have to jump wildly about over vast distances, trying to catch all the cities it has missed on it's path so far. The greedy algorithm will perform much better than a completely random route, but still leaves much to be desired.

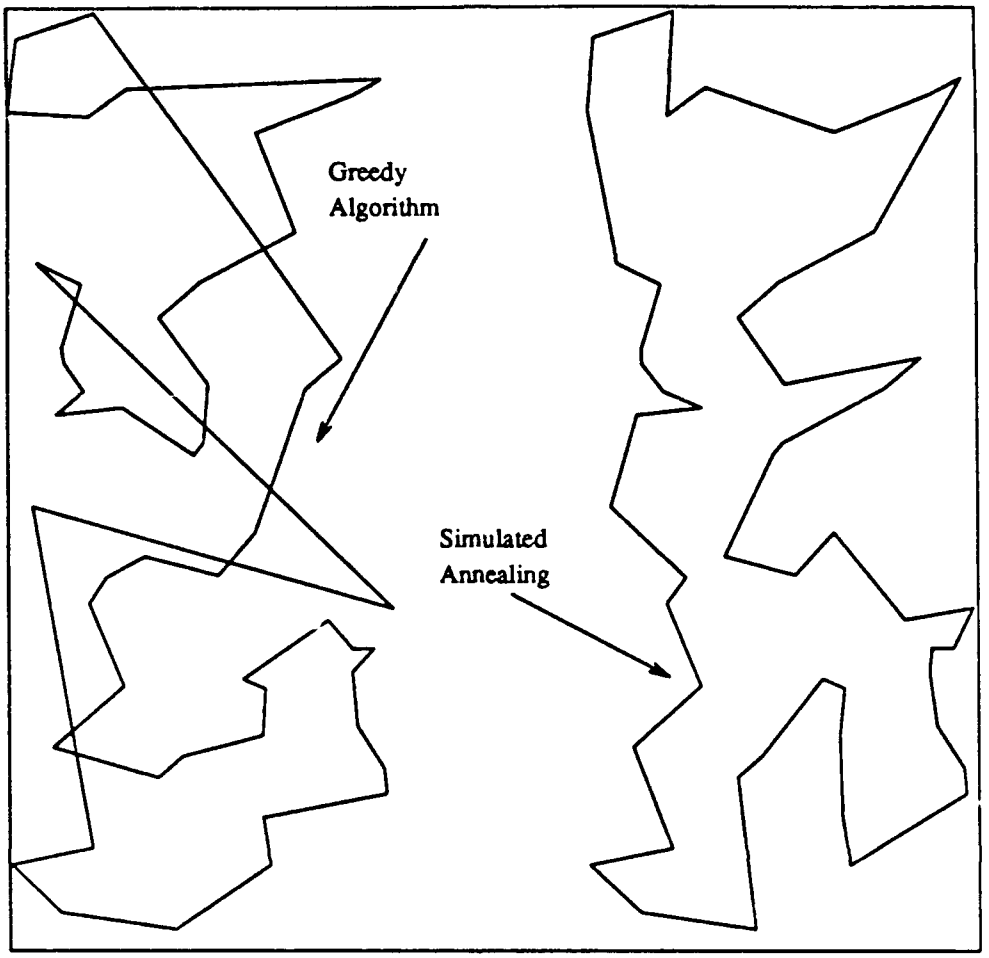


Figure 3: Performance of Greedy Algorithm and Simulated Annealing on a fifty city problem

A much better algorithm for *accuracy* is Simulated Annealing (Kirkpatrick *et al*, 1983) (see Figure 3). The simulated annealing algorithm models a collection of atoms at equilibrium at a given temperature. At every stage in the simulation, a small, random displacement is given to one of the atoms, and ΔE, the resulting energy change, is computed. If this change is less than or equal to zero, that is, if energy is lost from the system then the displacement is unconditionally accepted. If that energy change is greater than zero, that is, if energy will be gained by the system from the outside, then the probability that the change will be accepted is $P(\Delta E) = \exp(-\Delta E/k_B T)$, where k_B is the Boltzmann constant and T is the temperature. It is found that when *annealing*, or cooling slowly, rather than rapidly, the atoms in the system align themselves very regularly, whereas if the system is quenched, or cooled very rapidly, the structure misaligns badly, and the resulting solid is brittle. This model can be applied to the TSP quite easily. An *annealing schedule* is defined, such that for each temperature, a certain number of moves will be tried, and then the system will be cooled by a constant factor for the next iteration. If, at any stage, no successful moves occur, the process is stopped, and the system declared locally optimal. Each move, in the TSP, is one of two types (Press *et al*, 1988):

- A section of the path is removed, then replaced with the same cities running in the opposite order.
- A section of the path is removed, then replaced in between two cities on another section of the path.

The change, ΔE is then taken to be the change in distance imposed by this swap.

The complexity of simulated annealing is impossible to analyse, since the algorithm is somewhat statistically dependent, and the annealing schedule and stopping criteria are user controlled. It is possible to obtain a linear graph of time, or indeed an $O(n)$, $O(n^2)$, or any function of n, against n, by constraining the algorithm to one iteration only, and binding the number of moves per iteration to the time function required. $100n$, for example, is the function used in the experiments graphed in Figure 4. These graphs are of the time function of a simulated annealing algorithm constrained to one cooling step, and $100n$ moves per step, plotted against an A* exhaustive search algorithm, and a greedy algorithm, and the errors of the annealing and greedy algorithm compared to the perfect answer of the A* algorithm[5]. It should be stressed that the graph of a full, unconstrained simulated annealing algorithm against time involves heavy fluctuation, and is not linear, since the algorithm is statistically dependent.

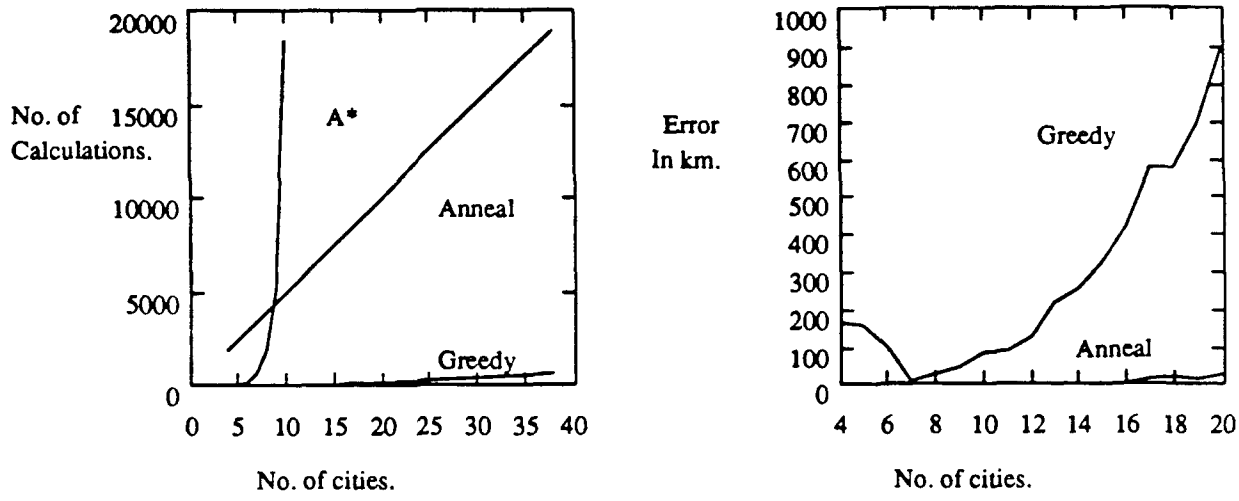


Figure 4: Performance of Simulated Annealing against A* and Greedy Algorithms.

To demonstrate the performance of the simulated annealing algorithm on a useful application, it was fed with the positions of thirty eight English county towns, to construct a fly-over tour around England (see Figure 5). There were no constraints on the number of iterations, and the algorithm was bound to an $O(n)$ number of moves, each temperature iteration.

It is probable that both the accuracy of the answer produced and the time taken by the simulated annealing algorithm can be improved upon in future algorithms. For now, though, simulated annealing shows that all of the $\mathcal{NP}$ problems can be answered in polynomial time, if we are willing to sacrifice a little accuracy, since the TSP is $\mathcal{NP}$-complete, and thus all $\mathcal{NP}$ problems are polynomially reducible to it.

[5]The error graph is plotted only to twenty cities, since it was found that the time taken by the particular A* algorithm used would have been unacceptably large for over twenty cities, and this paper wouldn't have been completed in this century, were it to run on thirty five cities.

little accuracy, since the TSP is $\mathcal{NP}$-complete, and thus all $\mathcal{NP}$ problems are polynomially reducible to it.

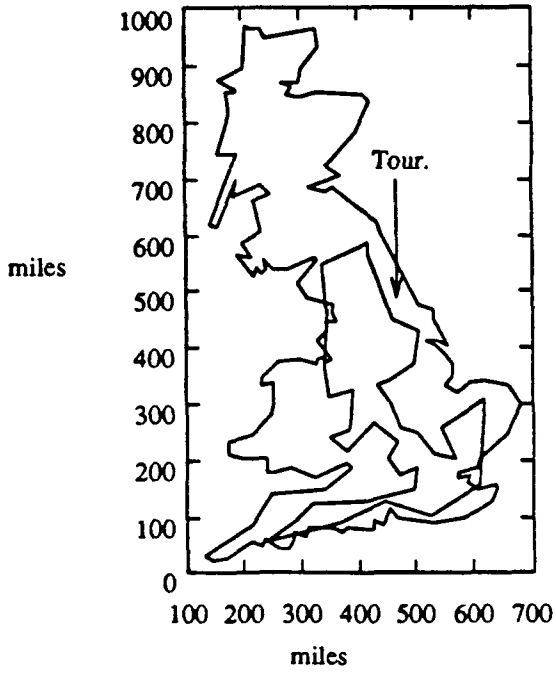


Figure 5: Tour of the County towns of England using Simulated Annealing

4.2 Memory Versus Time

It is quite likely that there will arise a case where we are not willing to sacrifice any of the accuracy required by a problem for a faster time. In this case, we need to lessen the time function of any algorithms in different ways. One way in which it might be possible to do this is to allow the algorithm to use larger amounts of memory, to cut down on the time it takes to solve the problem, i.e. to sacrifice memory for time.

Conclusions

Computational Complexity Analysis, although a powerful tool in algorithm design, can sometimes give us an impression of an algorithm which may not be very applicable in the range of input which we will use. It is useful, therefore to sometimes pursue a wider set of methods in calculating complexity and in analysing it's effect than are usually advised.

There are ways of countering 'hard' problems, through sacrificing a little accuracy, or memory, which can reduce the run-time of the algorithm significantly - even 'solving' $\mathcal{NP}$ problems in polynomial time.

In general, Computational Complexity Analysis can be a useful and powerful tool if a little extra thought is applied to it's application, but *can* be misleading if used carelessly.

References

Bachmann, Paul Gustav Heinrich, 1892,
Analytische Zahlentheorie.

Kirkpatrick, S, Gelatt, Jr, C.D., Vecchi, M.P, 1983,
"Optimization by Simulated Annealing," Science, Vol. 220,
Pages 671-680.

Lawler, E.L, Lenstra, J.K, Rinnooy Kan, A.H.G, Shmoys, D.B, 1985,
The Travelling Salesman Problem, A Guided Tour of Combinatorial Optimization,
Pages 52, 1-15, 37-85.

Press, W.H, Flannery, B.P, Teukolsky, S.A, Vetterling, W.T, 1988,
Numerical Recipes in C, The Art of Scientific Computing,
Pages 28-81, 81-84, 343-352.

Strassen, Volker, 1969,
"Gaussian Elimination is not Optimal," Numerische Mathematik, Vol. 13,
Page 354.

Wirth, Niklaus, 1986,
Algorithms and Data Structures,

REPORT DOCUMENTATION PAGE

DRIC Reference Number (If known) .....

Overall security classification of sheet ....UNCLASSIFIED.....
(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
MEMO 4468	APRIL	1991

Originators Name and Location
RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title
PRACTICAL COMPUTATIONAL COMPLEXITY

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period

Project Number	Other References

Authors	Pagination and Ref
SYMONDS, D	13

Abstract

Computational Complexity Analysis is a very powerful tool for use in designing algorithms. Unfortunately, the analysis can sometimes be misleading for the range of input with which the algorithm will actually be used.

The aim of this paper is to first give an overview of what computational complexity is, and how to analyse it, and to then explain what parts of complexity analysis can cause confusion, and how to avoid these mistakes. The paper also seeks to investigate ways of solving problems faster by sacrificing certain operating requirements.

Abstract Classification (U,R,C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)
UNLIMITED

INTENTIONALLY BLANK