IDENTIFYING ALGORITHMIC VULNERABILITIES THROUGH SIMULATED ANNEALING

by

S. Andrew Johnson

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Mathematical and Computer Sciences).

Golden, Colorado
Date 04/11/2005

Signed: S. Andrew Johnson
S. Andrew Johnson

Approved: D P Mehta
Dr. Dinesh Mehta
Thesis Advisor

Golden, Colorado
Date 04/11/2005

Graeme Fairweather
Dr. Graeme Fairweather
Professor and Head,
Department of Mathematical and Computer Sciences

ABSTRACT

Real–time software systems with tight performance requirements are abundant. These systems frequently use many different algorithms and if these algorithms were to experience worst case behavior, the system may not be able to meet its performance requirements. Unfortunately, the inputs which would cause worst case behavior are often unknown, making it very difficult to defend against them. In this thesis, I present a method for finding the worst case inputs to different algorithms using simulated annealing, a combinatorial optimization method. I show that this method is successful in finding worst case inputs to several sorting algorithms, using several measures of an algorithm's behavior.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This thesis has been a major undertaking, and it was made much more tolerable through the help of the people aroud me.

I would like to thank my advisor, Dinesh Mehta, for his help and patience with me; without his advice, there would have never been a thesis. Also, Lars Nyland and Ramki Thurimella for their willingness to be on my committee.

The support and encouragement of my family has been a great help, and has helped me through the sometimes long days.

Most importantly, my fiancé, Angela, for being there for me, supporting me, and loving me, even when I seemed to ignore planning for our life together to finish this work.

Chapter 1

INTRODUCTION

In this thesis, I show that it is possible to create a general system capable of finding inputs that elicit worst case behavior in various algorithms. While this is a relatively new field in algorithmic studies, Crosby & Wallach (2003) has shown worst case inputs to be a valid concern given the nature of today's real time systems. I have used simulated annealing, a randomized optimization technique, as the basis of this system.

When an algorithm has differing average case and worst case asymptotic complexities, there still remains the task of discovering what proportion of inputs falls into each category and which specific inputs, or types of inputs, can trigger worst case behavior. Any system which uses the algorithm could be adversely affected if worst case behavior occurred. Without the knowledge of what inputs elicit worst case behaviors, it is often difficult to predict when such behavior will occur and to guard against it.

In many algorithms, the worst case behavior is much worse, perhaps with a higher polynomial degree in N, than the average case behavior. When these algorithms are incorporated into systems with tight performance requirements, the system may be disrupted by a worst case input. Whether these inputs come through an attack, or through the normal course of the system's operation, the system may fail.

While the fact that this behavior occurs is commonly known, as algorithms become more complicated it becomes less common to know which inputs generates

the worst case behavior. Differences in the implementation of algorithms can also lead to slightly different worst case behaviors.

A systematic method of discovering the worst case inputs would be helpful in the design and choice of algorithms:

- Knowledge of worst case inputs would help an algorithm designer modify a given algorithm to improve performance in the worst case.
- Software developers would benefit by being able to choose an algorithm with a worst case input that was not likely to occur in their system.
- Real time systems may be able to reject worst case inputs to keep the overall system running well
- Security from algorithmic complexity attacks becomes possible

Typical asymptotic complexity analysis looks at the source code, or psuedocode, for an algorithm, but does not attempt to find a worst case generating input. When searching for a worst case input, analysis of the source code essentially involves running the algorithm by hand to see its behavior. For this reason, analysis of compiled and running code may be just as effective.

While an exhaustive search over all the inputs of a given size would certainly find the worst case inputs, it is not practical in a real system. For instance, in the situation where the order of the inputs is important, such as with sorting, N elements yield $N!$ different inputs, which would be intractable for N as small as 20. In the case of even a simple hash function, such as mod 100, as few as ten elements could create an intractable number of inputs.

Where an exhaustive search is not possible, some method must be used to decrease the search space. Random sampling would be possible if the worst case inputs

were a significant fraction of possible inputs. However, in many of the algorithms, there are very few worst case inputs compared to average case inputs. The number of inputs which would have to be tried before finding a worst case input could be impractical, and there would be no way of knowing how close to the worst case any of the inputs were.

In Chapter 2, I will talk about the previous work related to my research, both in eliciting worst case behavior, and in the use of simulated annealing. Chapter 3 covers various methods of analysis, as well as the metric by which I evaluated these methods. Chapter 4 covers the specific results for various sorting routines. Finally, in Chapter 5, I will discuss my conclusions and the future work related to my research.

Chapter 2

PREVIOUS WORK

2.1 Worst Case Behavior

There have been two papers that dealt with forcing worst case behavior in an algorithm. The first of these is Crosby & Wallach (2003) which worked with hash tables. This paper also showed the problems that can occur in a real system when worst case inputs are allowed. The second, McIlroy (1999), shows a method for discovering worst case inputs to quicksort.

2.1.1 Hashing

In Crosby & Wallach (2003), Crosby and Wallach find a worst case input to a hash table. A hash table stores data in a series of direct addressed buckets. A hashing function is used to compute a bucket index from a data key. Each bucket is capable of storing at least one element, and in some cases more. When an element is inserted into the hash table, its key is used to calculate a bucket index, and the element is placed in that bucket. The number of possible keys is generally much larger than the number of buckets, and the hashing function is designed to create a uniform distribution of keys over buckets.

When two keys map to the same bucket index, it is considered a collision. There are two main courses of action for when a collision occurs. Chaining puts the new element into the bucket with the other element, typically by making the bucket a

linked list of elements. Open-addressing uses some secondary function to iterate through all the buckets until an empty bucket is found, putting the new element in the empty bucket, and filling each bucket with a maximum of one element.

For each of these methods, the expected distribution of input elements is such that, if there are b buckets and N elements, $O(N/b)$ elements map to each bucket. In this case, the work required to find an element with a given key is $O(1 + N/b)$. However, in the worst case, all elements could map to the same bucket, which would require $O(N)$ work to find an arbitrary element. Hashing, as well as common simple implementations of hash tables, are discussed in Cormen *et al.* (2003)

In Crosby & Wallach (2003), the authors look at hashing functions which include an internal state in order to hash inputs that are arbitrarily long. Depending on the internal state, the hashing function will act slightly differently. The hash function initially will have an internal state of 0, then the first 32 bits of input will be hashed based on the internal state. The internal state is changed to the hash value and the next 32 bits of input are hashed. This allows long strings of input which differ slightly in the middle to return very different hash values, as opposed to a system which only hashed some set portion of the input, typically the start or end.

In this type of hashing function, it is possible to find several 32bit inputs which, if hashed with the initial state of the function, will leave the state unchanged. After finding several of these inputs, which the authors called generators, they can be concatenated in any order and any number of times to create long inputs with the same hash value. These generators can be found by an exhaustive search of valid 32bit inputs, and the search can be stopped whenever a sufficient number of generators has been found. Some hashing functions use an XOR operation, in which case the generators can be directly computed rather than needing an exhaustive search.

The authors used this system to discover many colliding inputs to hash table implementations in Perl 5.6.1, Perl 5.8.0, Squid Internet object cache and Bro, a network intrusion detection system. In each of these systems, the hash table checks for duplicates before insertion, so in the worst case, each insertion takes $O(N)$ time, or $O(N^2)$ time for the insertion of N elements.

The authors showed that in each of these systems, hash collisions could significantly affect the time the system took. In Perl, this resulted in scripts which took a couple of seconds on random data taking hours against the same amount of colliding data. In Bro, a system designed to analyze network traffic to find attack, this resulted in as much as 70% of the traffic being dropped, which would allow attacks on the network to go unnoticed. This showed that worst case behavior in these systems could make them essentially unusable.

The authors were able to compute these hash collisions by analyzing the hashing function and using it directly to find collisions. In the case of Bro, which uses an XOR operation to calculate the hash values, they were able to calculate collisions based on the algebraic structure of the hashing function. This method of finding collisions would not be possible if the hash table was examined as a whole without the source code.

2.1.2 Quicksort

In McIlroy (1999), the author shows that it is possible to force the worst case behavior in several implementations of quicksort. Quicksort will sort a list by first choosing a pivot element and partitioning relative to it, so that all elements less than or equal to the pivot are on one side in the list, and all elements greater than the

pivot are on the other side. Each side of the pivot is treated as a new array, and quicksort is called recursively on it.

Quicksort has worst case behavior when all elements fall on the same side of the pivot at each step. This results in a new subproblem which is almost the same as the original problem, and makes quicksort run in $O(N^2)$ time. In general, the elements are expected to be approximately equally distributed on each side of the pivot element, which makes the average run time $O(N \log N)$.

The Unix implementation of quicksort allows a user to pass in the comparison function, so that user-created data types can be compared correctly. The author showed that a comparison function can be chosen that can make virtually all elements appear greater than the pivot. This is possible because once the pivot is chosen, it is involved in all future comparisons. The author's comparison function essentially marked elements whenever two unmarked elements were compared, assigning increasing values to the marked elements. When a marked and an unmarked element were compared, the function would report that the unmarked element was larger. After choosing the pivot, the pivot would be marked, and the compare function would report that all other elements were greater than the pivot.

The method that the authors present assumes several things about the quicksort implementation that it is working against, from McIlroy (1999):

- The implementation is single threaded.
- Choosing the pivot takes $O(1)$ comparisons: all other comparisons are for partitioning.
- The comparisons of the partitioning phase are contiguous and involve the pivot.
- The only data operations performed are comparison and copying.

- Comparisons involve only input data values or copies thereof.

In practice, this includes most normal implementations of quicksort, both with simple and with median of 3 style pivot selection.

The code is reproduced from McIlroy (1999) in Figures 2.1 and 2.2.

```
#include <stdlib.h>

int     val;                    /* item values */
int     ncmp;                   /* number of comparisons */
int     nsolid;                 /* number of solid items */
int     candidate;              /* pivot candidate */
int     gas;                    /* gas value */

#define freeze(x) val[x] = nsolid++

int cmp(const void *px, const void *py)  /* per C standard */
{
     const int x = (const int)px;
     const int y = (const int)py;
     ncmp++;
     if(val[x]==gas && val[y]==gas)
          if(x == candidate)
               freeze(x);
          else
               freeze(y);
     if(val[x] == gas)
          candidate = x;
     else if(val[y] == gas)
          candidate = y;
     return val[x] = val[y];              /* only the sign matters */
}
```

Figure 2.1. Killer Adversary to Quicksort code

Since the values in the list were set on the fly to create the worst case input, this system fails whenever the input must be set before the algorithm runs, such as when

```
int antiqsort(int n, int * a)
{
     int i;
     int ptr = malloc(n*sizeof(*ptr));
     val = a;
     gas = n - 1;
     nsolid = ncmp = candidate = 0;
     for(i=0; i<n; i++) {
          ptr[i] = i;
          val[i] = gas;
     }
     qsort(ptr, n, sizeof(*ptr), cmp);
     free(ptr);
     return ncmp;
}
```

Figure 2.2. Killer Adversary to Quicksort code continued

a copy of the data is made. The author gets around this by supplying quicksort with a list of pointers, so that the underlying data can be changed. This method makes use of the observation that having the partition step put as many elements on a single side as possible would yield worst case behavior.

After the quicksort function was run, the sorted array of pointers could be ignored, and the original array would contain a worst case, or near worst case, input. This new input could be used directly to elicit worst case behavior.

If the quicksort function uses a random choice of pivot elements, this method will still force worst case behavior in the initial run, but the resulting array will not be a worst case input. No static array of numbers will always give worst case behavior if the choice of pivot element is random.

2.2 Simulated Annealing

Simulated annealing is an optimization technique commonly used in combinatorial problems. It is discussed in Sait & Youssef (1999), and is based on the physical annealing process that is used in the manufacturing of crystalline and metallic substances to create a defect-free product. The defect free state is typically a minimum energy state, and simulated annealing is able to use a process analogous to its physical counterpart to find the minimum in a general system.

In physical annealing, a substance is melted and heated to a very high temperature. It is then cooled very slowly so that it remains very close to thermodynamic equilibrium throughout the process. The molecules are very mobile, ideally completely mobile, at the initial high temperature and become monotonically less mobile as the temperature decreases. Molecules naturally tend toward lower energy states, and will "settle" into place as the system is cooled and their mobility decreases.

If the system is cooled too quickly, the molecules lose mobility before reaching the lowest energy state and high energy areas, defects, are formed in the resulting substance. There is virtually no risk of cooling the system too slowly, if production time did not matter, because the system will merely stay at the lowest energy arrangement.

In simulated annealing, an analogous method is used; the problem is set up as a system in which the lowest energy arrangement is the aimed for answer. A random process is used, similar to the random movements of molecules in a heated liquid, which allows the system to change its state. The simulated annealing method simulates a temperature which determines how much the system is allowed to change. Eventually, the system should settle into a low energy arrangement.

Simulated annealing is able to avoid being stuck in a local minimum due to its

random nature. It not only takes steps toward the lowest energy state, but it will take steps into higher energy states with some probability, which decreases with the temperature of the system.

In order to use simulated annealing to solve a problem, several things must be defined:

1. State: A way of representing all possible solutions to the problem.
2. Move: A method of randomly moving from one state to another. This should be set up to make every possible state reachable through a series of moves from an initial state.
3. Cost function: A function able to evaluate a given state and return an "energy level" to the system.
4. Schedule: A way of determining how many moves are made at a given temperature, and how the temperature should change.

Once these are defined, the process of using simulated annealing is quite simple. Following the schedule, the temperature is set, and at every iteration, a random move is made to a new state and the cost is calculated. If the cost has decreased, the move is accepted. If the cost has increased, the move is accepted with probability:

$$p = \exp(-\delta C/T), \tag{2.1}$$

where δC is the change in cost from the previous state to this state, computed using the cost function. If a move is accepted, simulated annealing simply moves on to the next iteration. If a move is not accepted, the old state is restored and then the next

iteration is performed. The pseudocode for simulated annealing is given in Figure 2.3.

```
SimulatedAnnealing(initialState, initialTemp, finalTemp, iterations)
  E = Energy(initialState)
  State = initialState
  temp = initialTemp
  while (temp >= finalTemp)
  do
    for I = 1 to iterations
    do
      newState = Move(State)
      newE = Energy(newState)
      if ( random() < exp(-(newE-E)/temp)
        state = newState
        E = newE
    done
    temp = reduceTemp(temp)
  done
```

Figure 2.3. Simulated annealing psuedocode

The mechanism of the cost function is not important to the workings of simulated annealing, so essentially any function can be used. The cost function must be set to the needs of the problem, and may include the ability to "throw out" solutions which are not feasible. This cannot always be done, such as in the case when feasible solutions are disjoint so that simulated annealing may need to travel through several infeasible states to get to a feasible, and perhaps optimal, state.

The moves in the system should be small enough that the optimal solution can be reached. If the moves are too large, then the answer will be only approximately optimal, and will often end up fluctuating around the true optimal answer. Moves

that are too small will only make the algorithm take more iterations, which may be a consideration, but does not affect the quality of a resulting answer.

The annealing schedule is perhaps the most important aspect of simulated annealing. It follows a simple to state, but difficult to accomplish, principle; at the initial temperature, virtually any move should be accepted, and as the algorithm progresses, the temperature should be lowered until virtually no moves with increased cost are accepted. The goal of the schedule is to keep the algorithm progressing. If the final temperature is too low, the algorithm may spend significant time running even after a solution has been essentially determined. If the initial temperature is not high enough, the algorithm may never get out of any local minimum that it starts in.

There are several schemes that have been traditionally used to determine the schedule. The initial temperature and final temperatures must be determined experimentally, since they depend on the scaling of the cost function. The method used to drop the temperature depends in some extent on the properties of the solution space. An exponential scheme can often be used to quickly "drop into" the global minimum, but there are also more reactive methods for lowering the temperature based on the simulated annealing results.

Chapter 3

RUNTIME ANALYSIS

For simulated annealing to work effectively, the cost function must give an accurate measure of the quality of a state. *In searching for the worst case input to an algorithm, the cost of a state is the amount of work that the algorithm required for a certain input. The cost should then be negated to allow the simulated annealing system to find a minimum.*

There are two basic ways of measuring the work an algorithm does. The first way is through asymptotic complexity measurements. This is a theoretical measure of the algorithm's work, and is not specific to a given input into the algorithm. The second method, and the one which I have used, is a practical analysis of the algorithm's runtime.

All experiments were run on a 300 MHZ Intel Pentium II running Fedora Core 2, Linux 2.6.7-1.494.2.2 with 192MB of RAM.

3.1 Overview

In my research, I used various methods of runtime analysis on the different algorithms as the simulated annealing cost function, negated to have decreasing cost with increasing runtime. In simulated annealing, the cost of a state will be calculated by running an algorithm to be tested under one of these analysis methods. I attempted several methods as a means of cross verification of results, and with the understanding that all methods may not be easily applied to every algorithm.

In choosing a method of analysis for the algorithms, there are five criteria which need to be examined. In order of importance, they are:

1. Accuracy: how well the analysis reflects the algorithm's performance.
2. Precision: the granularity of the analysis.
3. Invasiveness: how much access to the workings of the algorithm, or its implementation, is needed to perform the analysis.
4. Overhead: the amount of time that must be devoted to running the analysis operations.

Each of these areas impacts the ability of an analysis method to be used to do the type of algorithmic analysis needed in this case.

3.1.1 Accuracy

The accuracy of the method is primarily important. Any method of determining the behavior of an algorithm should report only information about the actual run of the algorithm. If the algorithm is being timed, the timing method should begin timing immediately before the algorithm starts to execute, and end timing immediately afterwards. No activity outside that of the algorithm should be included in its runtime.

Although the ideal would be a perfectly accurate method, one which was completely effective at segregating the behavior of the algorithm from other activity on the system, this is not always possible. Often, overhead from the analysis method itself will be included. Additionally, the operation of the system may force some system operations not associated with the algorithm into the analysis. When this cannot

be avoided, it is important to keep the interference constant. Simulated annealing examines the change in cost that resulted from each move the algorithm made, not the magnitude of the cost itself, so any constant overhead of the analysis method will not be a problem.

If the interference is essentially random, there is the chance that some moves will be seen to have artificially higher or lower cost than they should. When the magnitude of these random disturbances is low relative to the total cost of the algorithm, they can also be ignored. Moves will be approximately correctly evaluated, and the method will still converge on a solution, although more slowly than otherwise. The interference may force the number of iterations needed to increase, in order to allow time to converge. If the magnitude of the disturbances is large, simulated annealing essentially needs to find the lowest cost solution combined with the lowest cost interference, which may take many extra iterations.

The final type of interference is that which is in some way dependent upon the running of the algorithm. In this case, the reported cost of the algorithm may include non-random costs which aritificially promote some inputs over others. These costs are typically associated with a specific system, and are not indicative of the behavior of the algorithm in general. An example is when the behavior of an algorithm is affected by limited access to cache or by scheduling operations of the system.

3.1.2 Precision

The precision of the analysis method is also very important. The analysis method should ideally return costs which are unique for each input. This allows a total ranking of the inputs to the algorithm, and lets simulated annealing find the absolute worst case input. Even without the problem of finding a unique worst case answer, the

analysis method must report its results in a high enough resolution to be useful. If the resolution is too low, many states which should be considered distinct will instead be given the same cost. If consecutive states cannot be distinguished by cost, simulated annealing will essentially wander randomly between them. This means that precision must be high not only in terms of the overall runtime of the algorithm, but also in terms of the changes in the algorithm's runtime due to a small change in the output.

While simulated annealing should still eventually find the input with the highest reported cost, a low precision analysis method also brings into question the quality of the answer. If the worst tenth of the inputs cannot be distinguished, there will be many reported worst answers. Since the found inputs are meant to be used as a pattern for inputs which will illicit worst case behavior, having many different worst inputs may also make the worst case pattern very difficult to distinguish. It is much more difficult to see that 25 different inputs are each 75% sorted than a single input which is 95% sorted.

3.1.3 Invasiveness

Invasiveness is important to the analysis process because it gives a measure of what is required to analyze an algorithm. Each analysis method should treat the algorithm to be tested as a black box as much as possible. The less invasive the method is, the more easily applied to differing algorithms it will be.

I have defined four levels of invasiveness, each determined by how much access to the algorithm and the system it is running on the method needs to work.

Level 1: The algorithm is treated as a black box running on a system which reports no information.

Level 2: The algorithm is a black box, but the system needs to report some information.

Level 3: The algorithm is in a precompiled binary, but the system can report and control the execution of code.

Level 4: The algorithm's source code must be modified to report information.

The first three levels roughly correspond to running the precompiled algorithm on different types of machines: for level one, the algorithm could run on a remote machine, level two on a local machine, and level three on an emulator.

3.1.4 Overhead

Overhead is a big consideration with the various analysis methods. Overhead is the amount of work the system must do in order to use the method. If there is very little work for the method compared to what it is analyzing, the time taken for the analysis will be very close to the time for the algorithm to run. With some methods there may be a substantial amount of work so that a large amount of time may be taken on the analysis method, above any work that the algorithm does.

Some overhead is unavoidable, as any analysis method will have to do some work and it may be necessary to have a large amount of overhead for an analysis method which gives significant information about the algorithm. There are many methods which only require some constant amount of work before and after the algorithm runs, for these methods, the overhead will be the same regardless of the algorithm being analyzed. There are also several methods that require work while the algorithm is running which means that the overhead will increase as the running time of the algorithm increases.

3.2 Methods of Analysis

I examined several methods, and settled on four which were able to meet my needs.

3.2.1 Get Time of Day

The get time of day method is basically a wall clock measure of the execution time of the algorithm, based on a system call named `gettimeofday`. It requests the computer's current time before and after the algorithm ran and by subtracting the start time from the end time, the amount of time that passed while the algorithm was running can be found.

Example code is in Figure 3.1. In the example code, `start->tv_sec` gives the number of seconds, and `start->tv_usec` gives the number of microseconds.

```
#include <sys/time.h>

...

timeval *start = new timeval();
gettimeofday(start, null);

RUN_ALGORITHM

timeval *end = new timeval();
gettimeofday(end, null);

cost = (start->tv_sec * 1000000 + start->tv_usec) -
       (end->tv * 1000000_sec + end->tv_usec)
```

Figure 3.1. Get Time of Day example code

- This method has a fairly low accuracy since it merely looks at the amount of time that passes, without regard to how much of that time the algorithm was actually working. Any other work that the system does during that time will also be included, making the algorithm appear slower.

- The Get Time of Day method returns time in seconds and microseconds. This is not enough to distinguish uniquely between all states, leaving many of the states with the same cost, but it is enough to easily distinguish a few near worst case inputs from the average case inputs.

- Since this method only needs to know the time before and after the algorithm runs, it falls into invasiveness level 1, the least invasive level.

- The overhead for this method is about six microseconds per call, which is an acceptable amount, and significantly small compared to the execution time of the algorithms.

3.2.2 Global Cost

The global cost method is an accounting method for determining algorithmic performance. Some operation, or set of operations, in the algorithm are considered to be costly, or indicative of overall performance. Initially, before the algorithm runs, a global cost variable is set to zero. When these operations are performed, the global cost variable is increased. After the algorithm has finished, the global cost is returned as the simulated annealing cost. In the case of a single costly operation, this global cost may be incremented each time the operation is performed, but when there are several costly operations, it may be important to set the cost of each independently.

- This method is perfectly accurate, since it has definite costs associated with certain actions and nothing else is measured. The only difficulty is in deciding what should be measured.

- This method is also completely precise, giving whatever level of granularity is wanted.

- The global cost method falls into invasiveness level 4, needing access to the source code in order to analyze an algorithm. It may be possible to get around needing the source code to the actual algorithm if the the algorithm can be supplied a user defined function. A good example is in sorting; many sort functions allow the user to supply a custom comparison function. If the comparison is the only operation which is considered costly, a custom comparison function can be supplied which can increment the global cost variable.

  For algorithms that do not let the user supply any component, or when a supplied component is not considered costly, this method will only work if the algorithm's source code can be modified. Even with access to the algorithm's source code, this method may not be easy to implement. It may be a simple task to add code to increment a variable, but it is often difficult to determine which operation will be counted. If several operations are chosen, there must be some way of balancing their contributions to the cost, which may require a second level of analysis to find their relative impact on the algorithm, perhaps by measuring the relative run-times of the operations.

- Since this method inserts code which runs while the algorithm is running, its overhead is in some way proportional to the runtime of the algorithm. This

allows it to be used on algorithms with very short run times without interference, but on very long running algorithms, this overhead my be restrictive.

3.2.3 Instruction count

Counting the number of instructions which the algorithm executes is the most similar to a normal algorithmic complexity analysis. This method simply looks at the amount of code the algorithm executes for a given input. I have chosen to use a count of machine instructions rather than instructions in the source code, since a complex single instruction in the source code may hide a lot of work for the machine. Machine instructions are much closer in comparative execution time.

For my research, I have implemented this system using gdb, a c++ debugger. The gdb system is able to control and interact with the program that it is debugging, providing typical debug functionality. This may be considered to be very invasive, but the methods that I am using can be done on precompiled code.

I make use of several abilities of gdb:

- Setting breakpoints in code.
- Running a program from pre-compiled code (without specific compilation flags set).
- Read and set variables in programs compiled with debug information.
- Create user-defined commands.
- Run commands from a script without interaction.
- Loop commands based off of program variable values.

To run this system, a flag variable is set in the code, and a breakpoint is made, just before the call to the algorithm to be analyzed. Gdb is set to execute one instruction at a time, counting as it goes, until the flag variable changes value after the algorithm returns. The number of instructions executed is then stored in the cost variable in the simulated annealing code, and the program is advanced without interruption to the next breakpoint.

The gdb script I used is given in Figure 3.2 and the changes in the simulated annealing code are in Figure 3.3.

```
# let output scroll past one page
set height 0

# the counti command
define counti

# temporary count variable
set var $tcount = 0

# stopper variable set in the code to stop this command
while stopper != 1

# single step in machine instructions
stepi
set var $tcount = $tcount + 1
end

# set the global cost variable to the count of instructions
set var globalCost = $tcount
continue
end
```

Figure 3.2. Gdb script

```
stopper = 0;

BREAKPOINT
RUN_ALGORITHM

stopper = 1;
```

Figure 3.3. Instruction count example code

This script defines a command called counti, following the naming convention used in gdb to suffix the letter "i" to commands dealing with machine instructions, which counts the number of machine instructions between a breakpoint and a flag variable being set in the code. This command steps one machine instruction at a time through a program and increments a temporary variable. When it reaches the point in the code where the flag variable, in this case called "stopper" has been set, it stores the value from the temporary variable into the program variable called "globalCost." It then continues execution as normal until the next breakpoint.

The "stepi" command works even on programs that were not compiled with debug information, so it can be used on any executable code. The simulated annealing system is compiled with debug information, so the variables it uses are exposed to the debugger.

- In general, this system is very accurate, measuring the exact number of machine instructions the algorithm used and nothing else. There may be some cases where this is not a good measure of the normal work done in running the algorithm. For instance, if branch mis-prediction is a large component of the running time of the algorithm, this may be hidden by a count of machine instructions, if only those instructions that reach the commit phase are counted.

Similarly, if some instructions are more costly than others, and the algorithm could run slower with an input that performs proportionally more slow instructions but less instructions overall than with some other input with a high number of instructions, this method may not be applicable.

- This method is very precise, keeping a perfect count of the instructions run.
- This method falls into invasiveness category three, as running the program in a debugger in a step by step fashion is analogous to running it in an emulator.
- Using the debugger to count machine instructions requires running the entire system in the debugger, and adds a lot of overhead to the system. Compared to other systems, this method produces significant slowdown in the algorithm, since every machine instruction done by the algorithm is followed by significant work in the debugger. In practice, I have found this method to be at least 80,000 times slower than the same code running outside the debugger. This makes it very hard to use.

3.2.4 Read Clock Cycles

This is a variation of a wall clock method. Instead of looking at the actual time that the system reports, the number of clock cycles which have passed can be found. This method does not distinguish between clock cycles used by the algorithm and those used by the other processes on the computer.

This system is implemented using the `rdtsc` instruction available on modern desktop processors. The code is given in 3.4. In the exampe code, the calls to `cpuid` are necessary to keep the call to `rdtsc` from jumping forwards through the pipeline.

Multiple calls to cpuid are needed because the time to execute it decreases when it has been called recently. The rdtsc call returns a 64 bit counter.

```
unsigned int low, high;
unsigned long long start;
asm volatile("cpuid\n\t"
        "cpuid\n\t"
        "cpuid\n\t"
        "rdtsc"
        :"=a"(low),"=d"(high)
        :
        :"%ebx","%ecx"
        ); // put time in high/low
start = ((long long)high << 32) | low;

RUN_ALGORITHM

unsigned long long end;
asm volatile("cpuid\n\t"
        "rdtsc"
        :"=a"(low),"=d"(high)
        :
        :"%ebx","%ecx"
        ); // put time in high/low
end = ((long long)high << 32) | low;

cost = end - start;
```

Figure 3.4. Clock Cycles example code

- This method is no more accurate than the Get time of day method, naively including all clock cycles that have passed on the system in its cost, regardless of whether or not they were used by the algorithm.
- Clock cyles provide a much finer resolution than microseconds, so this method has a significant increase in precision from the get time of day method.

- This method is in invasiveness level 2, needing to read the clock cycles from the machine the algorithm is running on.
- There is very little overhead for this method as the system hardware is able to return the number of clock cycles without passing through any operating system code. It may, however, be difficult to implement this method on all systems as access to the assembly level instructions is needed.

3.2.5 Times

The times method is a variation of the wall clock method which records only the time used by the process. Calls are made to the `times` system utility before and after the algorithm runs and the returned values are subtracted, giving the time used by the algorithm.

Example code is given in Figure 3.5.

```
#include <sys/times.h>

...

tms *start = new tms();
times(start);

RUN_ALGORITHM

tms *end = new tms();
times(end);
cost = end->tms_utime - start->tms_utime;
```

Figure 3.5. Times method example code

- This method is very accurate, attempting to measure only the actual time the algorithm used. This is limited by how well the operating system keeps track of the process and obviously includes any overhead associated with the methods the operating system uses.

- The time is reported in `CLOCK_CYCLES` which could be defined on a system by system basis. In practice, they are usually set to 1/60th of a second. This is not enough to distinguish between different run times in the algorithms, except in the most extreme cases, giving this method a very low precision.

- The times method is in invasiveness level 2, needing the system to record the amount of time spent processing each task. Most modern operating systems do this type of accounting.

- The overhead is a little higher than for the get time of day method.

3.2.6 Get Resource Usage

The Get Resource Usage analysis method will give several various measures of an algorithms performance. It is based on a system call of `getrusage`, which returns a large amount of information about running process. This information includes the amount of time the process has spent on the cpu, the number of context switches which have been made, and the number of minor and major page faults, among others.

This method works in the same way as previous methods by comparing the resource usage before and after the algorithm runs and using that as a measure of performance. This method does, however, measure more than the amount of time that has passed.

Example code is given in Figure 3.6. In the example code, `end->DATA` and `start->DATA` point to the resource to be considered.

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

...

rusage *start = new rusage();
getrusage(RUSAGE_SELF,start);

RUN_ALGORITHM

rusage *end = new rusage();
getrusage(RUSAGE_SELF,end);

cost = end->DATA - start->DATA;
```

Figure 3.6. Get Resource Usage method example code

- This method is fairly accurate, dependent on the quality of bookkeeping done by the operating system.
- The precision of this method is moderate. Some resources have very precise measurements or counts, while others have low precision measurements.
- This method is under invasiveness level 2, needing the system to report a large amount of information.
- The overhead of this method is much higher than the other constant overhead methods, due to the amount of data it requests.

3.3 Experimental Comparison of Methods

I compared these methods in two ways. Initially, each method was implemented to determine its accuracy, precision, invasiveness and overhead. This allowed me to narrow it down to 4 methods which might be useable. These four methods were then compared to each other to determine their relative quality.

3.3.1 Initial study

Each method was used on several runs of an algorithm, noting the change in cost from one input to an adjacent input as well as the overhead associated with the method and the effect that system load had on the measure.

The results are summed up in Table 3.1.

Table 3.1. Quality of analysis methods

Method	Accuracy	Precision	Invasiveness	Overhead
Get Time Of Day	LOW	LOW	LOW-1	Constant-MID
Global Cost	HIGH	HIGH	HIGH-4	Percent-LOW
Instruction Count	HIGH	HIGH	MID-3	Percent-V.HIGH
Clock Cycles	LOW	HIGH	MID-2	Constant-LOW
Times	MID	V.LOW	MID-2	Constant-MID
Get Resource Usage	MID	MID	MID-2	Constant-HIGH

3.3.2 Comparative study

Based on the individual analysis of methods, I chose three which I believed were most suitable, the clock cycle measure, the get time of day method, and the global cost method. I used each of these methods to analyze a quicksort algorithm with the

same set of inputs, to see how well they correlated. The cost was found for a sorted list and one hundred random permutations of a list of 50 numbers. Each cost was calculated one hundred times and the minumum cost used. Each permutation was then ranked by its cost according to each method.

I also included the instruction count for its high accuracy. It made a good benchmark for the other methods, even though it had too much overhead to be practical in the simulated annealing system.

The number of inversions in the rankings of the inputs was used as a measure of the correlation between methods. This is the number of swaps that a bubble sort algorithm would take to convert one list of rankings into the other. This is a fairly common way of comparing two lists to see how similar they are.

A simple example can help illustrate this concept. If we create a list of inputs labeled by their rankings according to method I, but sort them by their rankings according to method II, we could get a list like: 1 4 2 3 5. A bubble sort algorithm would perform two swaps in order to sort this list by the labels, first swapping 4 and 2 and then swapping 4 and 3, so we would say that method II's ranking had two inversions with respect to method I. Typically the number of inversions is symmetrical; if we labeled the inputs by the ranking in method II and sorted by the ranking in method I we would get the list A C D B E, which also has two inversions.

With each method, there were some inputs which had the same cost, Table 3.2 shows the number of inversions if these same cost inputs are left in their initial random arrangement in each list.

If instead, inputs with the same cost are sorted relative to their order in the instruction count method, essentially giving a method the benefit of the doubt, a lower number of inversions results. Table 3.3 gives methods with low precision the most

Table 3.2. Comparison of Analysis Methods

Method	Global Cost	Get Time of Day	Instruction Count	Clock Cycles
GlobalCost	0	1194	540	1148
Get Time of Day	1194	0	886	692
Instruction Count	540	886	0	832
Clock Cycles	1148	692	832	0

help, while barely changing the number of inversions for the most precise method, the clock cycle measure, which only had two pairs of inputs with the same cost. This table is not symmetrical because some of the inputs have been rearranged.

For example, if we again label by method I rankings and sort by method II rankings, we may arrive at a list like 1 4 2 3 5. If elements 4, 2, and 3 have the same cost with method II, their ordering is at this point arbitrary. They could be rearranged while still being sorted with respect to the method II rankings to eliminate all inversions. However, if we reverse the process and label them by method II rankings and sort them by method I rankings, there is only one way to sort the list, A C D B E. Nothing can be rearranged and to exactly match the order in method II would take two swaps, even though the order of C, D, and B was completely arbitrary in method II.

This method of rearranging the inputs with the same cost was used in Tables 3.3 and 3.4. In both cases, the column lists the method which was able to rearrange same cost inputs and the row lists the method it was compared against.

The third table, Table 3.4, shows the results if we penalize low precision inputs by reverse sorting the elements with equal costs. Each method also gets some inversions compared to itself whenever it gives two inputs the same cost. This occurs because

same cost inputs are rearranged to create inversions, and shows how this comparison is affected by same cost inputs.

Table 3.3. Comparison of Analysis Methods, Helping Low Precision Methods

Method	Global Cost	Get Time of Day	Instruction Count	Clock Cycles
GlobalCost	0	672	538	1146
Get Time of Day	1169	0	885	691
Instruction Count	521	382	0	831
Clock Cycles	1128	132	831	0

Table 3.4. Comparison of Analysis Methods, Penalizing Low Precision Methods

Method	Global Cost	Get Time of Day	Instruction Count	Clock Cycles
GlobalCost	52	1703	541	1148
Get Time of Day	1221	1031	888	693
Instruction Count	573	1413	3	833
Clock Cycles	1180	1163	834	2

The expected number of inversions in a random list of this length is approximately 2500, so these methods are fairly consistent with each other.

Much of the difference that is shown between the methods is from a difference in what they measure. For instance, the global cost method measures the number of comparisons that quicksort does. Since a very simple version of quicksort was used, using the first array element as the pivot, sorted inputs become worst case in the number of comparisons. However, a sorted input will not require any swaps, which may save a large amount of time. While the worst case input is not likely to change,

inputs with a medium cost may be affected when an input that takes several more comparisons takes many less swaps.

Overall, the instruction count method is probably the most accurate, directly reflecting what the algorithm does. The other methods make measurements that are slightly more removed from the running of the algorithm, or which may be influenced by other system activity. This is reflected in the tables, where the instruction count method matches the other methods most closely. Due to the overhead associated with the instruction count method, the other methods are preferrable whenever they can be used to find good results.

Chapter 4

SORTING ALGORITHMS

I used simulated annealing with each of the four algorithm analysis methods chosen in the last chapter to find the worst case inputs to various sorting algorithms. I looked at the major sorting algorithms, quicksort, mergesort, and heapsort.

To use simulated annealing to find worst case inputs, I used the four timing methods from the previous chapter to find the cost of running the sorting algorithm. This cost was negated to allow simulated annealing to find a worst case input as a minimum.

For the simulated annealing state, I chose a permutation of a list. Since all of these sorting algorithms look at the relative values of elements, permuatations of a list of non-repeated values effectively represent all inputs. If the behavior of the algorithm is much different with repeated values, this will not allow the simulated annealing system to find those worst case inputs.

To move from one state to another, two random elements are swapped. All states are valid since there is no permutation of the list that the sorting algorithms would not be able to sort.

All experiments were run on a 300 MHZ Intel Pentium II running Fedora Core 2, Linux 2.6.7-1.494.2.2 with 192MB of RAM.

4.1 Quicksort

Quicksort is a commonly used sorting method. It is a divide and conquer algorithm consisting of several steps.

1. Select a pivot element
2. Partition the input array into an array of elements greater than the pivot and an array of elements less than the pivot.
3. Run quicksort on each array from step 2.

I used four different implementations of quicksort, finding the worst case inputs using each method. These implementations had increasingly complex methods of choosing a pivot, and in the final version, more complex behavior for small lists.

4.1.1 Simple Quicksort

The first version of quicksort I tested was from Pollatsek (2004). It is a simple implementation used basically for teaching. It uses a simple pivot selection, choosing the first element in the array as the pivot. It works recursively, and does all the work of the algorithm in a single function.

An ideal implementation of quicksort would do one comparison of each element with the pivot during the partition phase. This method works from each side, walking pointers through the list from the left while elements are less than the pivot and from the right while elements are greater than or equal to the pivot. When the pointers stop, the elements they point to are swapped, unless the pointers have passed each other. This allows there to be an extra comparison whenever the pointers pass each other, adding to the total cost.

With this algorithm, the worst case input is whenever all of the elements appear on one side of the pivot. This occurs when either all the elements are higher than the pivot or all the elements are lower than the pivot. This givestwo choices of pivot element at each iteration, with $N-1$ iterations in the worst case, for a total of 2^{N-1} worst case inputs out of $N!$ possible inputs. In the worst case there are N iterations and each iteration, in the worst case, does k comparisons due to the overlap, where k is the number of elements in the array. The total number of comparisons in the worst case can be found from Equation 4.1.

$$\sum_{i=N}^{2} i = \frac{N(N+1)}{2} - 1 \tag{4.1}$$

I used inputs which were permutations of an array containing the numbers one through fifty. From Equation 4.1 the worst case number of comparisons when $N = 50$ is 1274. The global cost method was able to find many inputs with 1274 comparisons.

All tables show the cooling rate (α) and the size factor. The size factor is multiplied by the size (N) to give the number of iterations at each temperture. The average costs given in the table are the average of the random initial permutations, and the SA costs are the average of the answers simulated annealing found. When applicable, a worst case column indicates the actual worst cost that could be attained in the algorithm.

In the case with the most tries, there were 516,000 tries out of approximately $3.5 * 10^{64}$ permutations in total. There are approximately $1.1 * 10^{15}$ worst case inputs, giving the odds of randomly getting a single worst case input in a sample of 517500 being approximately one in 10^{49}.

I performed the same experiment with inputs of 100 and 200 elements, using $15 * 100 = 1500$ and $15 * 200 = 3000$ random swaps per iteration respectively. With

100 input elements, simulated annealing still found the worst case inputs in some cases, but the average found input was slightly below the worst case. When there were 200 input elements, simulated annealing did not find a worst case input on any of the trials. These results are cused by the super–linear growth of the search space with growth of the input size. In the case of 200 input elements, only about 1 in 10^{345} permutations is a worst case input, and the odds of finding a worst case input during a trial are about 1 in 10^{339}. The full results for the global cost method are given in Table 4.1.

With the other analysis methods, it is much harder to say exactly how many worst case inputs there should be. As the precision of a method is increased, it becomes less likely that inputs will have the same cost, so it is less likely that there will be many inputs that have the same worst cost. Obviously, there is no simple formula for the number of microseconds or machine cycles needed for the worst case input. With the other methods, however, similar changes in the cost of inputs were found between the average inputs and the found worst case inputs as were found in the number of comparisons. Results for the other methods are given in Tables 4.2 and 4.3.

An example of a worst case input is given in Figure 4.1.

4.1.2 Modified Quicksort

For the second algorithm, I tested a modified version of quicksort. This was an implementation from Gray (2004) which I modified so that the global cost measure favored a forward sorted list over a reverse sorted list. This let there be only a single worst case input for the global cost measure, to see how well the simulated annealing system would do. The highest cost in the global cost method is exactly $N(N-1)/2$.

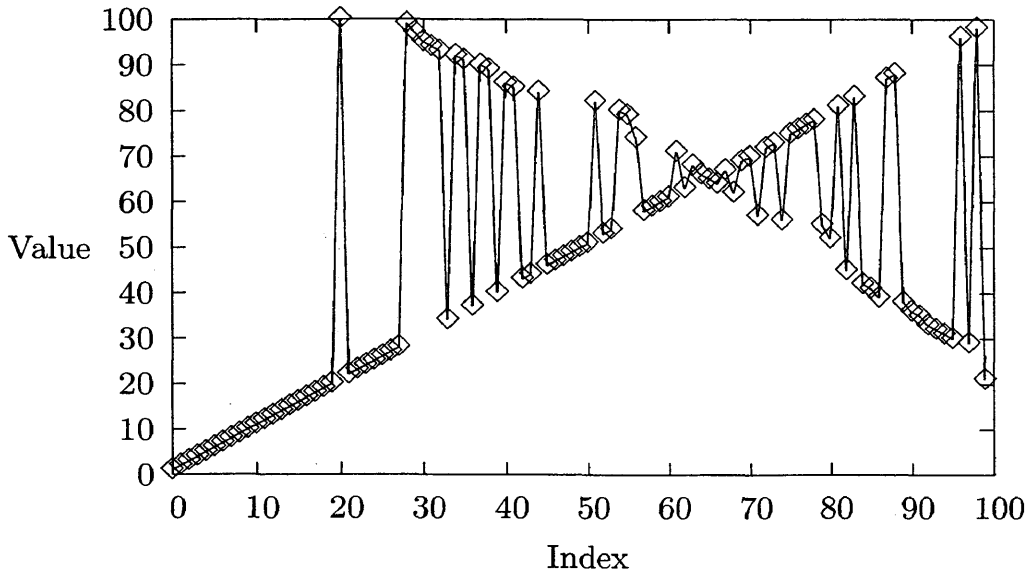

Figure 4.1. Worst case input to simple quicksort

Table 4.1. Simple quicksort results, GlobalCost method

Input Size	α	Iterations	Time (s)	Avg. Cost	Worst Case	SA Cost
50	0.80	10	1	331	1274	850
		15	2	336	1274	1000
	0.95	10	6	314	1274	1201
		15	8	314	1274	1236
	0.99	15	44	346	1274	1274
100	0.80	10	5	832	5049	2123
		15	8	779	5049	2377
	0.95	10	25	779	5049	3207
		15	41	744	5049	3848
	0.99	15	251	740	5049	4950
200	0.80	10	23	1794	20099	4777
		15	38	1850	20099	5636
	0.95	10	121	1976	20099	7278
		15	195	1970	20099	8395
	0.99	15	1261	1762	20099	15215

Table 4.2. Simple quicksort results, Clock Cycles method

Input Size	α	Iterations	Time (s)	Avg. Cost	SA Cost
50	0.80	10	2	9569	12164
		15	4	9682	12401
	0.95	10	15	9526	13719
		15	22	9556	13470
	0.99	15	127	9258	16436
100	0.80	10	14	22840	28383
		15	21	21672	28048
	0.95	10	65	21837	29793
		15	100	22004	30241
	0.99	15	424	21944	33080
200	0.80	10	63	50591	66869
		15	96	50524	67368
	0.95	10	274	50629	64803
		15	377	49514	65786
	0.99	15	1717	49617	66080

Table 4.3. Simple quicksort results, Get Time of Day method

Input Size	α	Iterations	Time (s)	Avg. Cost	SA Cost
50	0.80	10	3	39	48
		15	4	39	50
	0.95	10	15	39	52
		15	24	39	55
	0.99	15	126	39	61
100	0.80	10	14	84	107
		15	22	86	113
	0.95	10	67	86	115
		15	102	81	127
	0.99	15	472	83	140
200	0.80	10	60	180	962
		15	90	181	956
	0.95	10	266	179	958
		15	395	182	969
	0.99	15	1780	178	980

Since the modification only affected the global cost method, the results for the other methods were not important. The results are given in Table 4.4.

Simulated annealing had an easier time finding the worst case inputs in this case than in the simple quicksort because of the single global minimum. In the simpe quicksort implementation, there were 2^N global minima while in the the modified version there is a single minimum. This made a much smoother search space where it was easier to find swaps that increased the global cost.

An example of a worst case input is given in Figure 4.2.

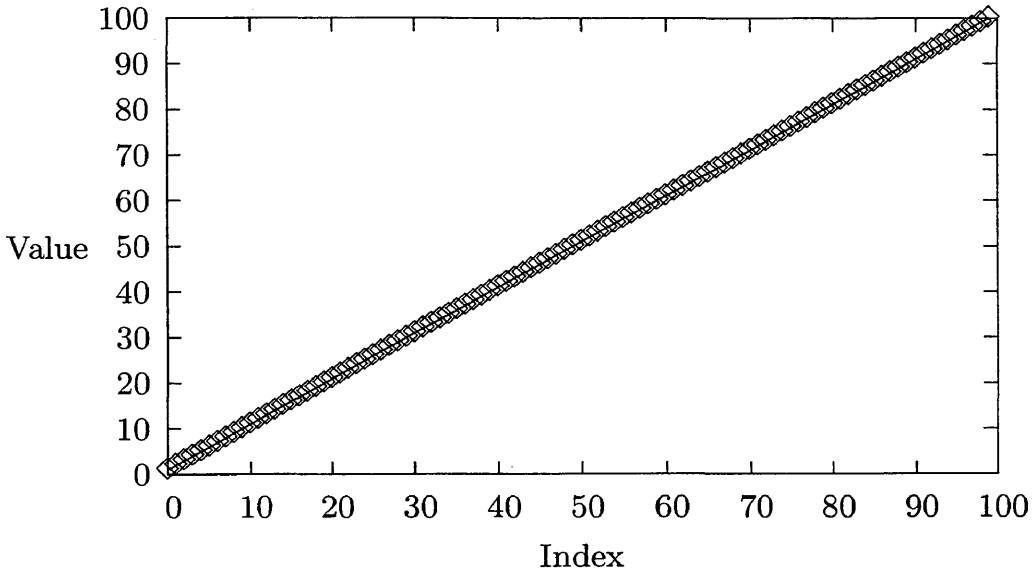

Figure 4.2. Worst case input to modified quicksort

4.1.3 Median of Three Quicksort

The median of three quicksort is the first quicksort algorithm modified to choose the median of the first, middle, and last array elements for the pivot. The global

Table 4.4. Modified quicksort results, GlobalCost method

Input Size	α	Iterations	Time (s)	Avg. Cost	Worst Case	SA Cost
50	0.80	10	0.6	237	1225	1137
		15	1	266	1225	1204
	0.95	10	5	224	1225	1220
		15	8	230	1225	1225
	0.99	15	43	219	1225	1225
100	0.80	10	5	602	4950	3677
		15	8	628	4950	4388
	0.95	10	27	560	4950	4914
		15	42	567	4950	4921
	0.99	15	233	581	4950	4949
200	0.80	10	25	1383	19900	7821
		15	38	1394	19900	9817
	0.95	10	130	1366	19900	16004
		15	203	1478	19900	18793
	0.99	15	720	1535	19900	19847

cost reports the number of comparisons that were done in the partition phase of the algorithm.

The results for this algorithm are found in Tables 4.5, 4.6, and 4.7.

4.1.4 STL sort

The STL implementation of sort is much more advanced than the others. It makes use of various methods for speeding up the quicksort function, and for selecting a pivot element wisely. This is a version of quicksort which may be commonly used by developers. It is discussed in Josuttis (1999).

An example of a worst case input to STL sort is given in Figure 4.3.

The results for this algorithm are given in Tables 4.8, 4.9, and 4.10.

Table 4.5. Median of Three quicksort results, GlobalCost method

Input Size	α	Iterations	Time (s)	Avg. Cost	Worst Case	SA Cost
50	0.80	10	0.4	151	627	340
		15	1	154	627	418
	0.95	10	4	154	627	450
		15	7	152	627	471
	0.99	15	36	155	627	504
100	0.80	10	4	389	2502	1016
		15	6	375	2502	939
	0.95	10	20	379	2502	1065
		15	32	414	2502	1208
	0.99	15	167	445	2502	1372
200	0.80	10	21	928	10002	2208
		15	31	911	10002	2185
	0.95	10	96	1015	10002	2591
		15	144	950	10002	2891
	0.99	15	755	927	10002	3241

Table 4.6. Median of three quicksort results, Clock Cycles method

Input Size	α	Iterations	Time (s)	Avg. Cost	SA Cost
50	0.80	10	2	6866	8487
		15	3	6882	7909
	0.95	10	10	6876	7939
		15	16	6956	8039
	0.99	15	84	6849	8145
100	0.80	10	10	14554	15959
		15	14	14492	15908
	0.95	10	44	14467	16033
		15	69	14513	16047
	0.99	15	350	14473	18113
200	0.80	10	43	30555	32345
		15	64	30577	32369
	0.95	10	191	30710	35938
		15	287	30596	36982
	0.99	15	1474	30400	39845

Table 4.7. Median of three quicksort results, Get Time of Day method

Input Size	α	Iterations	Time (s)	Avg. Cost	SA Cost
50	0.80	10	2	29	33
		15	3	29	37
	0.95	10	13	29	33
		15	20	29	33
	0.99	15	107	30	33
100	0.80	10	10	57	72
		15	17	57	66
	0.95	10	51	57	68
		15	78	56	67
	0.99	15	400	56	72
200	0.80	10	45	114	196
		15	70	114	200
	0.95	10	205	114	434
		15	309	114	876
	0.99	15	1580	113	930

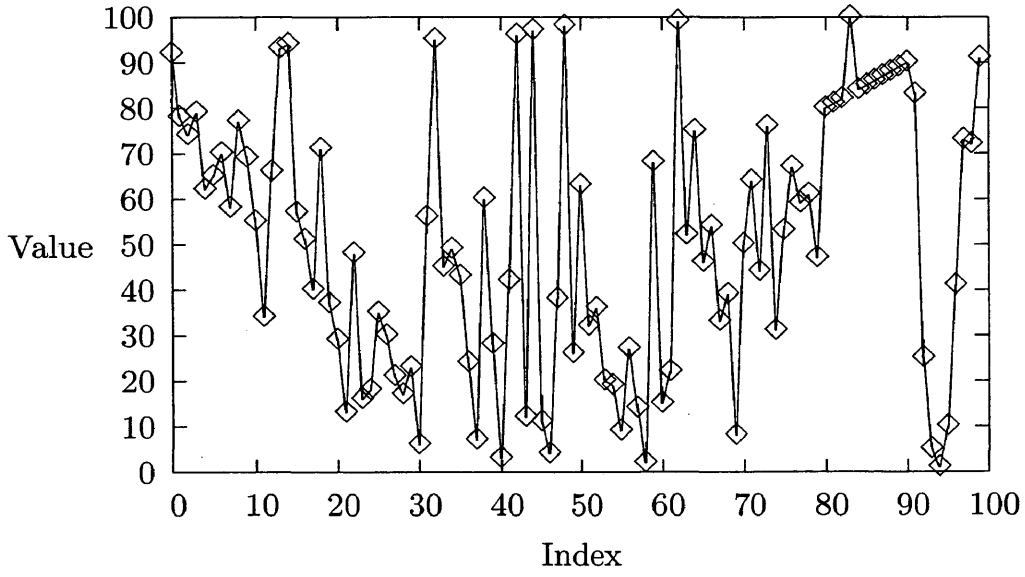

Figure 4.3. Worst case input to STL sort

Table 4.8. STL sort results, GlobalCost method

Input Size	α	Iterations	Time	Avg. Cost	SA Cost
50	0.80	10	1	311	652
		15	2	320	667
	0.95	10	7	334	747
		15	11	342	755
	0.99	15	65	330	817
100	0.80	10	7	764	1509
		15	10	725	1519
	0.95	10	36	747	1678
		15	56	789	1705
	0.99	15	320	746	1900
200	0.80	10	34	1773	3364
		15	54	1750	3544
	0.95	10	182	1742	3653
		15	265	1776	3875
	0.99	15	1485	1742	4212

Table 4.9. STL sort results, Clock Cycles method

Input Size	α	Iterations	Time (s)	Avg. Cost	SA Cost
50	0.80	10	2	5374	5872
		15	3	5373	5950
	0.95	10	10	5384	5982
		15	16	5373	5977
	0.99	15	82	5447	6004
100	0.80	10	10	12382	14333
		15	15	12317	12885
	0.95	10	47	12291	14436
		15	71	12310	14015
	0.99	15	362	12286	15586
200	0.80	10	47	28228	29165
		15	71	28132	30750
	0.95	10	209	28184	31289
		15	316	28166	34919
	0.99	15	1605	28148	36674

Table 4.10. STL sort results, Get Time of Day method

Input Size	α	Iterations	Time (s)	Avg. Cost	SA Cost
50	0.80	10	2	25	28
		15	4	26	28
	0.95	10	14	24	29
		15	21	25	28
	0.99	15	110	26	28
100	0.80	10	11	51	53
		15	18	51	54
	0.95	10	54	51	54
		15	82	50	54
	0.99	15	426	50	110
200	0.80	10	50	110	122
		15	77	110	880
	0.95	10	229	110	870
		15	346	110	472
	0.99	15	1774	110	875

4.2 Mergesort

Mergesort is a sorting algorithm that runs in worst case $O(N \log N)$ time. It works by splitting the input array into individual elements and then recursively merging elements and lists together into larger sorted lists

4.2.1 qsort

Despite its name, the linux qsort function is an implementation of mergesort. It only uses a quicksort implementation if it is unable to allocate enough memory for a second array the size of the input array. This is the case when mergesort would run in $O(N \log(N)^2)$ time.

The results for this algorithm are given in Tables 4.11, 4.12, and 4.13.

Table 4.11. Linux qsort, GlobalCost method

Input Size	α	Iterations	Time (s)	Avg. Cost	SA Cost
50	0.80	10	1	221	237
		15	2	218	237
	0.95	10	7	225	237
		15	11	219	237
	0.99	15	60	220	237
100	0.80	10	7	540	573
		15	10	539	573
	0.95	10	32	543	573
		15	50	540	573
	0.99	15	261	540	573
200	0.80	10	32	1273	1345
		15	50	1277	1345
	0.95	10	147	1280	1345
		15	220	1286	1345
	0.99	15	1131	1287	1345

Table 4.12. Linux qsort results, Clock Cycles method

Input Size	α	Iterations	Time (s)	Avg. Cost	SA Cost
50	0.80	10	3	9500	10169
		15	4	9549	10175
	0.95	10	15	9533	10182
		15	23	9579	10208
	0.99	15	118	9533	10336
100	0.80	10	15	20582	21550
		15	21	20628	22807
	0.95	10	66	20679	21564
		15	100	20623	22735
	0.99	15	512	20600	26907
200	0.80	10	66	44397	5240
		15	98	44552	53533
	0.95	10	288	44428	51555
		15	433	44406	56270
	0.99	15	2232	44406	53559

Table 4.13. Linux qsort results, Get Time of Day method

Input Size	α	Iterations	Time (s)	Avg. Cost	SA Cost
50	0.80	10	3	39	41
		15	5	39	41
	0.95	10	17	39	41
		15	27	39	41
	0.99	15	144	39	41
100	0.80	10	15	78	94
		15	24	78	82
	0.95	10	74	78	83
		15	109	78	122
	0.99	15	565	78	145
200	0.80	10	68	161	930
		15	103	161	933
	0.95	10	307	162	947
		15	457	163	929
	0.99	15	2336	161	1003

4.2.2 STL stablesort

The STL stablesort function is an implementation of the mergesort routine. It divids the initial array differently than the qsort function, so it needs a different number of comparisons.

An example of a worst case input to this version of mergesort is given in Figure 4.4.

The results for this algorithm are given in Tables 4.14, 4.15, and 4.16.

4.3 Heapsort

Heapsort is a sorting algorithm which builds a min heap and then continuously removes the minimum element to build the sorted array. It runs in worst

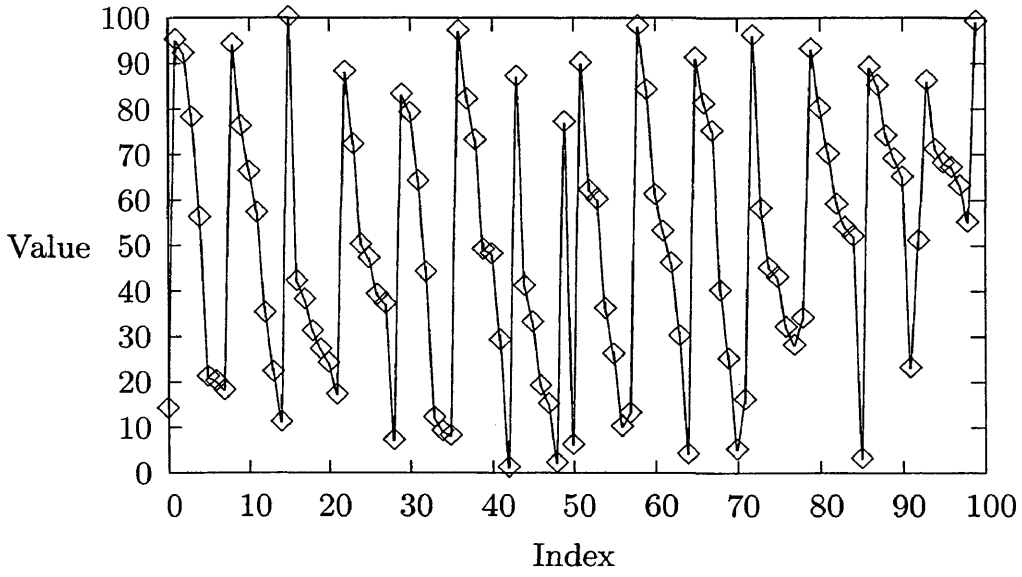

Figure 4.4. Worst case input to STL stablesort

Table 4.14. STL stable sort, Global Cost method

Input Size	α	Iterations	Time (s)	Avg. Cost	SA Cost
50	0.80	10	1	246	322
		15	2	245	323
	0.95	10	6	239	323
		15	9	252	323
	0.99	15	52	251	323
100	0.80	10	6	605	761
		15	9	600	762
	0.95	10	30	591	763
		15	44	598	763
	0.99	15	234	600	763
200	0.80	10	28	1400	1723
		15	43	1386	1724
	0.95	10	127	1386	1727
		15	192	1409	1727
	0.99	15	984	1416	1727

Table 4.15. STL stable sort results, Clock Cycles method

Input Size	α	Iterations	Time (s)	Avg. Cost	SA Cost
50	0.80	10	2	8876	9973
		15	4	8884	10185
	0.95	10	13	8986	10217
		15	22	9126	10261
	0.99	15	111	8943	11291
100	0.80	10	14	20417	24422
		15	20	20401	25398
	0.95	10	65	20368	23073
		15	97	20452	23368
	0.99	15	496	20423	28048
200	0.80	10	60	42101	48294
		15	91	40814	49599
	0.95	10	264	40664	49722
		15	399	40720	49142
	0.99	15	2036	40633	52535

Table 4.16. STL stable sort results, Get Time of Day method

Input Size	α	Iterations	Time (s)	Avg. Cost	SA Cost
50	0.80	10	3	39	41
		15	5	39	42
	0.95	10	17	39	51
		15	26	39	42
	0.99	15	141	39	46
100	0.80	10	15	78	81
		15	23	78	91
	0.95	10	70	77	81
		15	107	78	113
	0.99	15	554	78	124
200	0.80	10	63	152	921
		15	98	150	971
	0.95	10	280	150	921
		15	423	151	937
	0.99	15	2159	151	1012

case $O(N \log N)$ time. The implementation that I tested was the STL partial_sort() function. It is described in Josuttis (1999).

An example of a worst case input to heapsort is given in Figure 4.5.

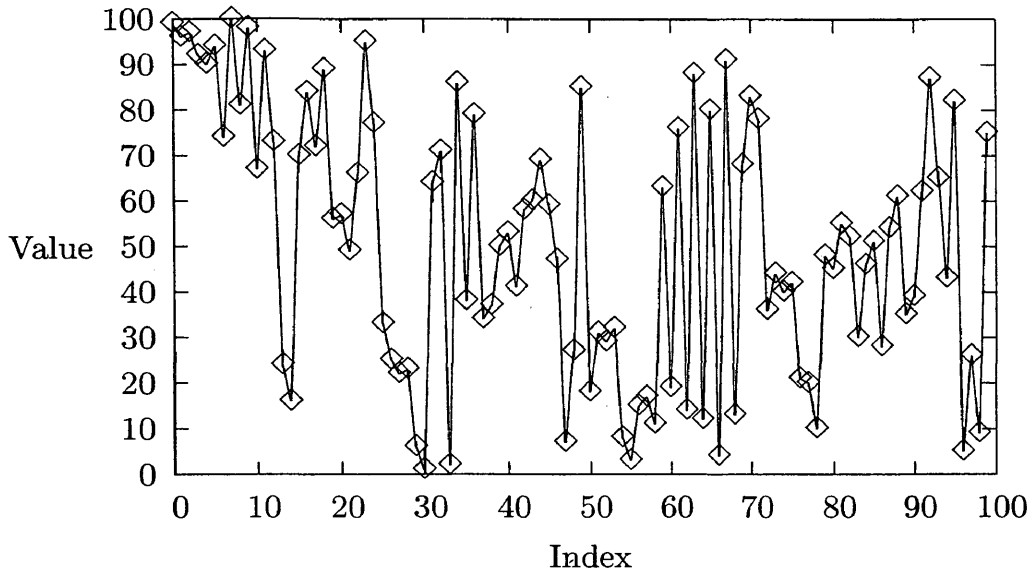

Figure 4.5. Worst case input to heapsort

The results for this algorithm are given in Tables 4.17, 4.18, and 4.19.

4.4 Analysis

In each algorithm, simulated annealing was able to inputs with significantly greater cost than the average input. In general, the global cost measurement showed the greatest change, and had the best results, the Get Time of Day method was able to find definite worst case inputs in some cases, and the clock cycle method did poorly. The poor performance of the clock cycle method was probably due in large part to the scaling of the cost. Analysis of the method showed that a few initial moves were

Table 4.17. Heapsort results, GlobalCost method

Input Size	α	Iterations	Time (s)	Avg. Cost	SA Cost
50	0.80	10	1	297	339
		15	2	298	340
	0.95	10	8	295	344
		15	12	295	343
	0.99	15	61	294	347
100	0.80	10	7	695	780
		15	11	694	783
	0.95	10	35	699	794
		15	53	693	789
	0.99	15	281	696	804
200	0.80	10	36	1597	1741
		15	54	1600	1756
	0.95	10	163	1591	1771
		15	241	1595	1781
	0.99	15	1235	1596	1806

Table 4.18. Heapsort results, Clock Cycles method

Input Size	α	Iterations	Time (s)	Avg. Cost	SA Cost
50	0.80	10	4	18295	18868
		15	7	188275	18874
	0.95	10	21	18384	19056
		15	33	18306	19006
	0.99	15	174	18258	19071
100	0.80	10	23	41866	45929
		15	34	41858	45809
	0.95	10	103	41829	46563
		15	154	41865	47664
	0.99	15	789	42232	48688
200	0.80	10	102	92534	102449
		15	154	92596	106836
	0.95	10	450	92530	105028
		15	450	92530	105028
	0.99	15	3438	92587	107660

Table 4.19. Heapsort results, Get Time of Day method

Input Size	α	Iterations	Time (s)	Avg. Cost	SA Cost
50	0.80	10	5	70	71
		15	8	69	72
	0.95	10	25	69	72
		15	38	69	72
	0.99	15	199	69	72
100	0.80	10	24	149	577
		15	37	150	841
	0.95	10	110	149	877
		15	163	149	912
	0.99	15	844	149	917
200	0.80	10	105	324	1114
		15	159	323	1135
	0.95	10	468	323	1165
		15	706	326	1116
	0.99	15	3586	324	1204

able to find inputs of higher cost, but there was very little progress for the rest of the run. This may also have been caused by interference during the algorithm's run. If an earlier run was likely to receive some rather large interference, it would be difficult for the algorithm to find an input with a higher cost.

In general, simulated annealing was able to find inputs which produced considerable slowdown in the algorithms. Although the input sizes were fairly small, if the worst case inputs which were found were extrapolated to larger input sizes, the algorithms could be even more drastically slowed.

Chapter 5

SUMMARY AND FUTURE WORK

In my research, I have shown that it is possible to create a generic system that finds the worst case input to an algorithm. Simulated annealing has been shown to be one method which can be effectively used. I have also discussed various methods of algorithmic analysis which can be used to determine the cost of a given input into an algorithm.

In the future, the following extensions could be made:

- This system could be extended by introducing more algorithms of varying natures to it. Hash tables, selection algorithms, and binary trees could all be tested.

- This system could be adapted to systems where the order of the input elements was not the important factor in creating a worst case input. For instance, the values of the elements, and not their order, is important for hash tables.

- Additional work could also be done to improve upon the method of counting machine instructions. While it has been shown that the gdb debugger is capable of doing this, a faster method would be far preferable. The use of an emulator may significantly speed up this method.

- Other combinatorial techniques could be used, such as genetic algorithms.

REFERENCES

Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., & Stein, Clifford. 2003. *Introduction to Algorithms, second edition.* McGraw–Hill.

Crosby, Scott A., & Wallach, Dan S. 2003. Denial of service via algorithmic complexity attacks. *Pages 29–44 of: Proceedings*, vol. 12. USENIX Security Symposium, Washington D.C.

Gray, Neil. 2004. *A Beginners C++.* draft.

Josuttis, Nicoai M. 1999. *The C++ Standard Library, A Tutorial and Reference.* Addison–Wesley.

McIlroy, M. D. 1999. A Killer Adversary for Quicksort. *Software-Practice and Experience*, **29(0)**, 1–4.

Pollatsek, David. 2004. *Quicksort.* Available online from Carleton College at: www.mathcs.carleton.edu/courses/course_resources/cs227_w96/pollatsd/qsort.html.

Sait, Sadiq M., & Youssef, Habib. 1999. *Iterative Computer Algorithms with Applications in Engineering, Solving Combinatorial Problems.* IEEE Computer Society.