Royal Holloway, University of London
Department of Computer Science

Full Unit Project

Simulated Annealing Timetabling Simulation

by
Josh Godley

Contents

1 Introduction **4**

1.1 Background 4

1.1.1 Travelling Salesman Problem 5

1.2 Simulated Annealing 7

1.2.1 History 7

1.2.2 Process 8

1.3 Motivation 9

2 System Specification **11**

2.1 Background 11

2.2 Requirements 11

2.3 Implementation 12

3 Design **13**

3.1 Aims 13

3.2 High Level Design 14

3.2.1 Randomness 14

3.2.2 Functionality 15

4 Software Engineering **16**

4.1 Overview 16

4.2 In the beginning: Travelling Salesman 16
4.2.1 Simulated Annealer 17
4.2.2 Subsystems 19
4.3 Timetabling System 22
4.3.1 Important Considerations 22
4.3.2 Changes 27
4.3.3 Subsystems 28

5 Worked Examples 32
5.1 Travelling Salesman 32
5.2 Timetabling Simulation 34

6 Testing 37

7 Assessment 39
7.1 Problems and Enhancements 39
7.1.1 Copying Arrays 39
7.1.2 Boltzmann Formula 40
7.2 Realisations 40
7.3 Critical Analysis 41

8 User Guide 43
8.1 Compiling the system 43
8.2 Using the system 43
8.2.1 Travelling Salesman 44
8.2.2 Timetabling Simulation 46

Chapter 1

Introduction

When I first started implementing my project, I created a simulated annealing system to solve the Travelling Salesman problem, which I will now go into further.

1.1 Background

The timetabling problem is a problem that we have all come into contact with in some form or another, whether it be at work or in some form of education we all know what a timetable is, and how they are put together. The one main issue that comes to light without fail when planning a timetable is a fairly obvious one, how does one organise the timetable into the hourly slots in the best possible way for everyone who it applies to? That is to say, what is the best way of organising the slots such that none of the slots have any clashes of any type, and more importantly insuring that everyone can only be in one place at once, for example if a student were assigned to be in two places at once this would be impossible, let alone if a lecturer were to be assigned to more than one separate class at once. Over the years, there have been many plausible methods to finding an optimal solution, although in this project I am going to focus on but one of these methods: Simulated Annealing.

1.1.1 Travelling Salesman Problem

The travelling salesman problem is a problem in combinatorial optimisation that is studied in theoretical computer science, that involves being given a list of cities and their pairwise distances, that is the distances in between each and every city, with the task of finding the shortest possible route that visits each city exactly once.

The problem was first formulated back in the 1930's as a mathematical problem and is one of the most studied problems in combinatorial optimization, as well as being used as a benchmark for many optimisation methods. Although the problem is optimisationally complicated due to the sheer number of possible outcomes, there are some heuristics that are known that allow for problems involving up to tens of thousands of cities to be solved.

There are numerous potential applications for this problem, some examples of which being:

- Road/City planning
- Microchip manufacturing
- Logistics
- DNA sequencing

Although these applications may not seem to relate to the Travelling Salesman Problem as it was originally designed, in each case the concept of a 'city' means a very different thing, for example in microchip manufacturing the cities may represent soldering points or fragments of DNA in the case of DNA sequencing. Even though these application relate to the Travelling Salesman Problem, they all involve additional constraints on the system such as limited resources or time limitations which make solving the problem considerably harder.

Finding a solution

Finding an exact solution to the Travelling Salesman Problem is only realistically possible on relatively small size problems, due to the complicated nature of the problem. There are three traditional methods for devising a solution to the problem, which are:

- Design an algorithm to find an exact solution, which is realistically implausible as mentioned above
- Finding 'suboptimal' solutions
- Using 'subproblems' of the initial problem to allow for better or exact heuristics

The most commonly taken path of the three above is to find a 'suboptimal' solution to the problem, that is to devise an algorithm that will deliver either seemingly or probably good solutions, although these solutions cannot be proven to be optimal.

Complexity

The Travelling Salesman Problem, and most variants of it have been proven to be NP-hard, in other words they are complete for the complexity class FP^{NP}, with the only exception to this being the decision problem variant, which is NP-complete. This remains the case even if one were to remove the restriction of only being able to visit each city once as the optimal tour in the planar case is the one which visits each city only once.

Exact Solutions

The simplest and most direct solution to the Travelling Salesman Problem is to calculate every possible route and see which one is the 'cheapest' to traverse, using a 'brute-force' search method of going through each possible route systematically, although the running time for this approach falls within a polynomial factor $O(n!)$, which rapidly tends towards infinity, rendering this approach impractical even with a small number of cities to traverse. Even with current methods and the amount of time the problem has been under scrutiny, it is difficult to improve on a solving time that falls outside of these bounds.

There are many heuristics and algorithms that have been devised to provide 'suboptimal' solutions to this problem, for example the Nearest-Neighbour algorithm which travels to the closest unvisited node for every move through the route, although this generally only produces results with similar ending distances compared to other methods and heuristics.

In my case however, I have chosen to use simulated annealing to find a solution to the problem as opposed to the other possible solutions mentioned

above.

Probably Good Solutions

The main difference between solving the travelling salesman problem using *simulated annealing* and solving it using other well-known algorithms is that most algorithms used to solve this problem only search for optimal or suboptimal results, that is to say results that can be proven to be optimal, whereas *simulated annealing* on the other hand produces results and solutions that are deemed probably or seemingly good. These solutions differ from optimal solutions in that they appear to be optimal in the reduction of the total distance of the route throughout the map of 'cities', however unlike optimal solutions they cannot be proven to be optimal as the solutions produced each time will differ to the others that are produced.

1.2 Simulated Annealing

1.2.1 History

The annealing process was originally devised by metallurgists as a technique involving the controlled heating and cooling of a material to increase the size of the crystals that make it up, thus strengthening the bonds between these crystals at the same time as reducing the material's defects. Simulated Annealing is a generic probabilistic metaheuristic for the mathematical problem of global optimization, that is to say that it allows for a good approximation of the global minimum of a function with a large search space. This means that it allows estimation of the smallest value a function will have on average when the function has a very large range of possibilities for its outcome. The original annealing process worked as follows: the material being worked on is heated to such a point that it is glowing hot, then it is allowed to cool for a certain amount of time before being reheated to the same state. This is repeated until acceptable results are received, and the metal has been deemed suitably strengthened and clean of impurities.

1.2.2 Process

The simulated annealing process works under the same concept, but is applied in a very different way; the process is basically a set of repeated loops that continue to run until an acceptable system state has been attained. It does this by starting off with the set of lecture slots filled randomly, and with every iteration of the annealer two slots are picked and swapped at random, then the system state as a whole is re-evaluated and given a 'score' depending on the current system state, that is to say the ordering of the various lectures/classes in the timetable.

This score is calculated by giving each individual lecture/class a score, so every system should have the same value if there are no clashes. However if there are any clashes/shared slots, then a multiplier is applied to the total score depending on how many of said clashes there are. This will greatly increase the overall score of that particular state, thus making it obvious that this state is not an improvement of the previous in any way.

The scoring values are implemented to allow the system to determine relatively easily whether an improvement has been made after the last iteration. It does this by determining whether or not the new state's score is lower than that of the state that was previously deemed the 'optimal' state. If the score is higher however, there is a formula that determines whether or not to accept this new state, which looks as follows:

$$P(state1, state2, T) \tag{1.1}$$

If state2 $<$ state1, then the new state becomes the 'currently accepted' state, as a better score results in instant acceptance of the state.
If however state 1 $<$ state2, then the third value is used in calculating whether or not the new state should be accepted. This value is the 'temperature' of the system, which links back to when annealing was only a metallurgical process, and is critical to the system functioning correctly. The actual formula I implemented for accepting values outside of absolute improvements is as follows:

$$randomDouble > e^{\frac{-\delta}{T}} \tag{1.2}$$

The way this formula works is as follows: δ is the difference between the two states, and T is the current temperature value of the system. Every time a state is ajudged for acceptance, a random double between the values of 0 and 1 is generated, and the exponential of $\frac{-\delta}{T}$ is calculated and compared to this random number. If the random number is greater than the value of the acceptance formula, the current state is accepted as the new best state, else it is rejected. This means that the longer the system runs, the smaller and tighter the margin for accepting states which aren't absolute improvements, due to the temperature decreasing gradually.

The scoring system is effectively the core system behind the simulated annealing process, as that is where 'good' and 'bad' moves are determined, and the temperature only decrements after a certain number of 'good' or 'bad' moves have taken place (in my case, 500 good moves and 1000 bad). This allows for a gradual reduction in the number of 'bad' moves that are accepted which don't improve the system state in any way as process is running; that is the acceptance values for 'bad' moves are slowly refined to produce a far higher quality result than if a 'downhill' approach were to be taken i.e. only accepting values which are an improvement on the currently accepted one.

1.3 Motivation

When I first began researching my project, the thought of creating a simulated annealing system seemed somewhat daunting to me, however after reading up on the various aspects of the annealing process prior to it being converted to a computational optimisation approximate solution when it was still merely a process used my metallurgists, I found I could relate to the process and better understand it as a whole. This was due to the physical application of the annealing process, as when I had to think of it merely in terms of numbers and processes it became quite confusing. Upon reading further into how the annealing process was applied to metalwork, I came to understand how it could be applied to a mathematical or optimisational problem due to the physical

application which allowed me to visualise the process.

I first became interested in the simulated annealing process after having looked at various other heuristics that are used to solve the travelling salesman problem, such as the nearest neighbour (or greedy) algorithm, and I discovered that the annealing process is an almost unique approach to combinatorial optimisation problems due to the random nature of the swaps that occur, and instead of searching for an abolutely optimal solution it produces approximately or probably good solutions. Enticed by this prospect, I was eager to begin an implementation of a system that used the simulated annealing process to achieve its end result, and this is where the travelling salesman problem was brought into the equation; I first began working on said problem so that I could get a grasp of the ins and outs of the simulated annealing process.

Chapter 2

System Specification

2.1 Background

The timetabling problem is a problem that we have all come into contact with in some form or another, whether it be at work or in some form of education we all know what a timetable is, and how they are put together. The one main issue that comes to light without fail when planning a timetable is a fairly obvious one, how does one organise the timetable into the hourly slots in the best possible way for everyone who it applies to? That is to say, what is the best way of organising the slots such that none of the slots have any clashes of any type, and more importantly insuring that everyone can only be in one place at once, for example if a student were assigned to be in two places at once this would be impossible, let alone if a lecturer were to be assigned to more than one separate class at once. Over the years, there have been many plausible solutions to finding an optimal timetable, although in this project I will be focusing on but one of these solutions: Simulated Annealing.

2.2 Requirements

The problem of organising a timetable requires a solution which takes as input different courses, and the people who are in charge of the courses to ensure that no implausible situation can occur such as a teacher having to be in two classes at once. After taking this input, the system then needs to organise these

lectures into an order which fits the hours of the timetable, with no clashes or other such errors. The system must run smoothly and be easy to use and most importantly execute quickly and efficiently. It should have a Graphical User Interface which should be clear, clean and simple, to reduce confusion amongst users, and abolutely should handle errors intelligently and cleanly.

2.3 Implementation

I will be implementing the system using the Java programming language, due to it's cross-platform nature as this will allow users of many different kinds of systems to all use my program should it be necessary. Also, I intend for the system to be:

- Implemented in a modular fashion, that is to say that each individual part of the program should be easily removable/changeable without causing any adverse effects to the rest of the system.
- Efficient and concise i.e. only use as much memory/processor time as is necessary before terminating.
- Easy to use - using Java's built-in GUI classes, I will provide a clean, self-explanatory user interface.

Chapter 3

Design

3.1 Aims

My main aims when designing both the travelling salesman and timetabling system simulations were for the implementation to be able to:

- Produce probably or seemingly optimal solutions to the travelling salesman problem
- Organize a timetable full of classes into a probably optimal order.
- Remove clashes from said timetable.
- Produce said results within reasonable time constraints depending on the problem.

To keep the implementation as simple as possible with the same functionality, I included the following additional requirements:

- The minimum number of nodes that can be input to the travelling salesman problem must be 4 or greater, as below this, there are no changes to be applied to the route.
- The number of iterations of the simulated annealing system will be limited to prevent the system running for longer than is necessary, as after a certain point no worthwhile changes will happen.

3.2 High Level Design

This section will discuss the considerations I had to take whilst designing an implementation for the two simulations, as well as the a basic synopsis of the functionality of each simulation.

3.2.1 Randomness

One further consideration I had whilst designing the implementations for the two separate simulations was the degree of randomness to include in the simulations. I eventually decided upon quite a high level of randomness due to ease of use and running, which I will now go into further detail about.

Travelling Salesman

The degree of randomness in the travelling salesman simulation will be such that every time the simulation is run, each node in the map will be given randomly generated coordinates to allow for greater differentiation between different routes. As well as this, when these nodes are put into some kind of order at the start of the simulation, this ordering will be random such that even if the same map of nodes is produced in two separate run-throughs of the simulation, their ordering will be different, thus producing a different problem with differing solutions.

Timetabling System

Whilst designing the timetabling system simulation, I thought long and hard about how to best implement the simulation - I was unsure whether to use a high-degree of randomness so that the simulation could be run quickly and easily, or whether to have all of the lectures initially input and organised by the user to provide the starting system state so that the simulation would have a high level of customisability. In the end, I decided to implement the simulation using a certain degree of randomness, that is to say the lectures are randomly added to the empty timetable at the start in a random order, with each slot being assigned anywhere between 0 and 4 lectures randomly. This is in order to allow the simulation to be run quickly and easily by the user, yet still producing the same level and quality of results as if the user had personally tweaked and configured the simulation to their personal requirements.

3.2.2 Functionality

This section will give a basic synopsis of the functionality of each simulation.

Travelling Salesman

The travelling salesman implementation works by taking a map of nodes as input with a start point specified, and then proceeds to randomly swap two nodes on the route before calculating the total distance of these two routes. If there has been an absolute improvement on the total distance of the route, the new route is accepted as the current optimal route before continuing to the next iteration, whereas if there hasn't been an absolute improvement in the route then the Boltzmann formula is used to determine whether or not the new route should be accepted.

Timetabling System

The timetabling system implementation works by taking a blank timetable and randomly filling it with lectures as mentioned above, then using the simulated annealing process mentioned in the randomness section prior to this one it randomly chooses two lectures assigned to different hour slots and swaps them before calculating the 'score' of the current timetable. This score is calculated by giving each lecture a point score, which is summed for every lecture every time the score is calculated, before multiplying said score by a multiplier. The multiplier is what brings the variation to the scores calculated from the timetable configuration as every time a clash is encountered, that is to say there is more than one lecture assigned to a particular slot in the timetable the multiplier is incremented by one, so the more clashes there are, the higher the score will be for that particular configuration.

Chapter 4

Software Engineering

4.1 Overview

I began my implementation of a simulated annealing timetabling system by implementing a solution for the travelling salesman problem first, as this is traditionally used for teaching simulated annealing methods due to the appropriateness of the application. This took most of my time throughout the course of the project, as I required a thorough understanding of how the simulated annealing process actually works before trying a real-world implementation of my own. After finishing this implementation of a solution to the travelling salesman problem, I had to take the annealing code from that and modify it to function correctly in the timetabling system. Once this was done, I created a GUI and tweaked the system to an acceptable standard and the system was ready for release.

4.2 In the beginning: Travelling Salesman

I began my implementation of a solution to the travelling salesman problem (TSP) by researching more into the problem to allow me to fully understand what kind of solution was necessary. As the TSP is such a complex problem when it comes to solving it, I researched the possible solutions to the problem and made a comparison between them and the simulated annealing method. I came to the conclusion that there are very few methods for producing an optimal solution to the problem, and most of the devised heuristics and methods to

this day do not provide optimal solutions to the problem, they are mainly used as approximation algorithms which produce approximately or probably good solutions in that they are close to but not quite the optimal solution, however there are many more solutions if you use this method than only searching for an optimal answer.

I decided upon using the following design structure in my initial implementation:

- An Annealer class which will handle all the decision making and swapping of the system, as well as producing the final results
- A Point class which holds all the relevant information of the various points, e.g. their coordinates
- A Runner class which will initialize the system and actually start the annealing process
- A Panel class which will be some derivation/extension of the JPanel class for displaying graphical results

The first important decision I had to make upon beginning my implementation was which data structures to use to store all of the necessary data that will be passed throughout the system during its runtime. Initially, I chose to use ArrayLists from the Java API to store all the data relating to the annealer and the points, that is to say having an ArrayList of points that would hold every points coordinates, as well as having ArrayLists that hold the current optimal route and the proposed new optimal route. Upon implementing said ArrayLists however, I found them to be inefficient and less suitable for their purpose than simple arrays of point objects, so I changed all of the ArrayLists in my code to arrays to simplify and speed it up.

4.2.1 Simulated Annealer

Once I had implemented the base systems, such as the arrays of data and created the classes necessary for running my system, I then had to go about implementing the actual annealer itself. This proved to be far more difficult than it would have initially seemed, as the system required a large amount of tweaking and ensuring that every variable and method was doing precisely what it should

have been. I began by using only while loops with variables set to terminate the system upon acquiring a suitable route through the map of nodes, then came to find this an inefficient implementation of an annealing system. Subsequently, I renovated the annealer class and set it up as a series of for loops and if statements all within a while statement which watches for the temperature value to tend to zero before exiting.

The code I had for this looks as follows:

Listing 4.1: Annealing Code Snippet

```
1  while(temperature >= 10e-10)
2          {
3                  while(good < 500 && bad < 1000){
4                          swap(course);
5                          dis1 = totalDistance(course);
6                          dis2 = totalDistance(accepted);
7                          delta = dis2 - dis1;
8
9                          if(delta > 0 || rngA.nextDouble() >
                               Math.exp(-delta/temperature)) {
10                                 copyArray(course, accepted);
11                                 good++;
12                         }
13                         else
14                                 bad++;
15                         }
16                 }
17 }
```

In this snippet, you can see that there is the overall while loop which only breaks when temperature drops below 10e-10, or basically when it gets close to zero, which allows the annealing process to run until only the finest error margin is accepted. Just after this, there is another while loop which only breaks once either 500 ‘good’ moves or 1000 ‘bad’ moves have occured, which basically resets the good and bad count, and decrements the temperature slightly after every complete iteration to narrow the error acceptance margin.

Inside this loop, the swapping of the nodes occurs as is shown by the swap(course), which randomly picks two elements from the route and swaps them round to alter route. After this, two variables are declared, dis1 and dis2, which are the total distances of the proposed optimal route, and the current best route respectively. Delta is the difference in total distance, or ’score’ between

the two routes, which is one of the inputs to the actual Boltzmann distribution formula used for determining whether a route should be accepted or not.

After that comes the really crucial part of the annealing code, the acceptance bounds; as you can see, the first statement is simply an if statement which checks to see if there has been an absolute improvement on the current optimal route, i.e. if delta is less than 0, and if it is then the route is accepted as the current optimal or best route, and the 'good' counter is incremeneted by one.

Next is the actual Boltzmann distribution formula, which works as follows: a random double between the values of 0 and 1 is generated, then this value is compared to the results of the Boltzmann formula i.e. $e^{(-\delta/T)}$, and if the randomly generated number is larger than the result of this formula, the route is within acceptance bounds, and will be accepted as the new optimal route before the 'good' counter is once again incremented. If this is not the case, then the else statement is in place to reject the currently proposed optimal route, and keeping the already accepted best route for the next iteration, whilst incrementing the 'bad' counter to show that a rejection has occured.

4.2.2 Subsystems

Apart from the actual simulated annealer itself, the system has various other subsystems that are critical to the overall functioning of the program. The main susbsystems are:

- swap() method for swapping entries in the arrays of points
- totalDistance() method for calculating the total distance of the routes through the array of points
- getDistance() method for calculating the distance between two points
- copyArray() method for duplicating arrays

swap()

The swap method in my system is a fairly simple one, albeit crucial to the correct functioning of the program, as it is one of the integral parts of the annealing system. The code for it looks as follows:

Listing 4.2: Swap code

```
1  public void swap(Point[] arr) {
2                  int firstSwap = rngA.nextInt((arr.length));
3                  int secondSwap = rngA.nextInt(arr.length);
4                  Point swapPoint = arr[firstSwap];
5                  arr[firstSwap] = arr[secondSwap];
6                  arr[secondSwap] = swapPoint;
7          }
```

The swap method work by generating two random integers which are within the total size of the array that is passed into the method as a parameter. Next, the element in the array with its index in the array equal to the randomly generated number is copied to a new Point object, and then swapped with the element with its index in the array equal to the first number. Finally, the copy taken of the other element in the array or the 'swap point' is then swapped with the first randomly chosen element, thus finalising the swap of the two points.

totalDistance()

Listing 4.3: totalDistance code

```
1  public double totalDistance(Point[] arr) {
2
3                  double distance = 0;
4                  double addTo;
5
6                  for(int i = 0; i < arr.length-1; i++) {
7                          Point get1 = arr[i];
8                          Point get2 = arr[i+1];
9                          addTo = getDistance(get1, get2);
10                         distance += addTo;
11                 }
12                 return distance;
13         }
```

The totalDistance method is a relatively simple method in that it just recursively calls getDistance on all the elements in the array, adding each distance to the total of the previous distances until the total distance of the entire route has been calculated, which is then returned as a double. As you can see, this is done by using a for loop which terminated when it gets to the penultimate iteration to allow for the distance to the last element in the array to be calculated correctly, with the variable addTo being the one which is changed after every

iteration and added to the current total, distance.

getDistance()

Listing 4.4: getDistance code

```
1 public double getDistance(Point p1, Point p2) {
2                 double xdiff = p2.x - p1.x;
3                 double ydiff = p2.y - p1.y;
4                 double distance =
                      Math.sqrt((xdiff*xdiff)+(ydiff*ydiff));
5                 return distance;
6         }
```

The getDistance method is fundamental to the system, as it is used to calculate the distance between each of the points on the route. It does this by using the Math.sqrt() method from the Java API to calculate the distances between the points using pythagoras. More specifically, it uses the Math.sqrt() method to calculate the distances between the points by taking the difference in the two points' x and y coordinates, then squaring these differences and getting the square root of their sum to give the distance between the two.

copyArray()

Listing 4.5: Array copying code

```
1 public void copyArray(Point[] from, Point[] to) {
2                 for(int i = 0; i < from.length; i++){
3                         to[i] = from[i];
4                 }
5         }
```

The copyArray method is probably one of the simplest in my system, although it has a crucial role to play; the only way to duplicate the current best array to allow for random swapping of two points in said array is by using this method, as merely setting array1 = array2 doesn't duplicate or copy any of the elements in the arrays, it merely sets the two array objects to point to the same section of memory where the array is being held. This led to countless problems in the system, so I devised this method to copy all of the elements from one array to a second, specified in the parameters of the method.

The result() method that is run just before displaying the graphical representations returns the final 'optimal' solution that is the output of the annealer

method, along with various interesting and useful information such as the total number of nodes in the route, the scores of the starting and 'optimal' routes as well as the number of overall good and bad swaps that occured whilst the annealer was running.

One of the last things that I implemented into my system was a method to display a graphical representation of the starting and final route through the system. I did this by implementing an extension of the JPanel class in the Java API, and overloading the paint() method so that every time the annealing algorithm finishes, the starting route is diplayed along with another window with the resulting 'optimal' route, although this was a fairly routine operation as it involved simply reading through the array of points for each case, and plotting each point as a square in the JPanel, then finally drawing the lines that represent travel between two cities to mark the route taken.

Some examples of the outputs of my travelling salesman solution are shown in figures 4.1 and 4.2, along with some example graphs of how the total distance decreases in correlation to the temperature dropping in figures 4.3 and 4.4.

4.3 Timetabling System

4.3.1 Important Considerations

The first thing I had to decide upon when moving on from my implementation of a solution to the travelling salesman problem is how I was going to store all of the various data and information related to the timetabling system, much the same as with the travelling salesman except there were further considerations to take when planning. In many ways, the data structure of my timetabling system is similar to that of the travelling salesman implementation, however there were some additional considerations I had to take whilst planning my implementation. These were such things as:

- More than one element in the array can be assigned to the same slot on the timetable
- Checking for duplicates instead of scoring a route
- Lecturers with clashes is utterly implausible
- Visual representation - how to best display a timetable

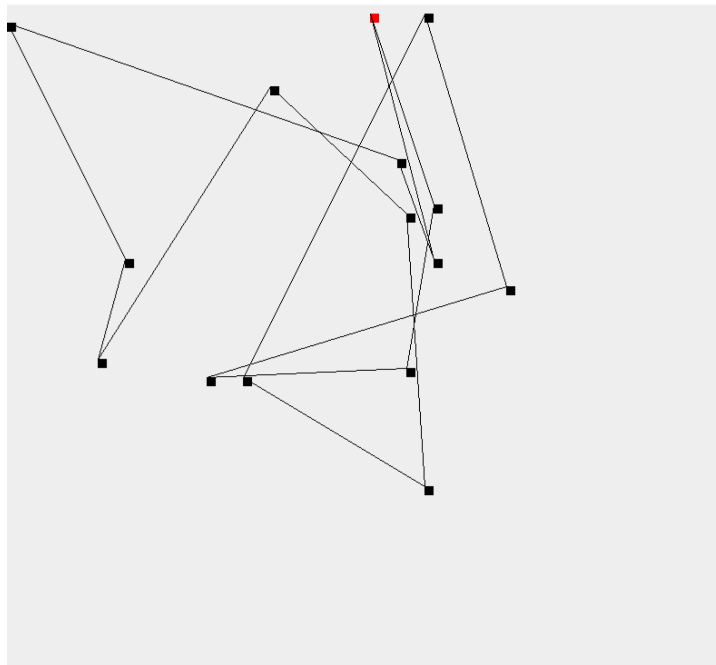

Figure 4.1: Example output of a starting route

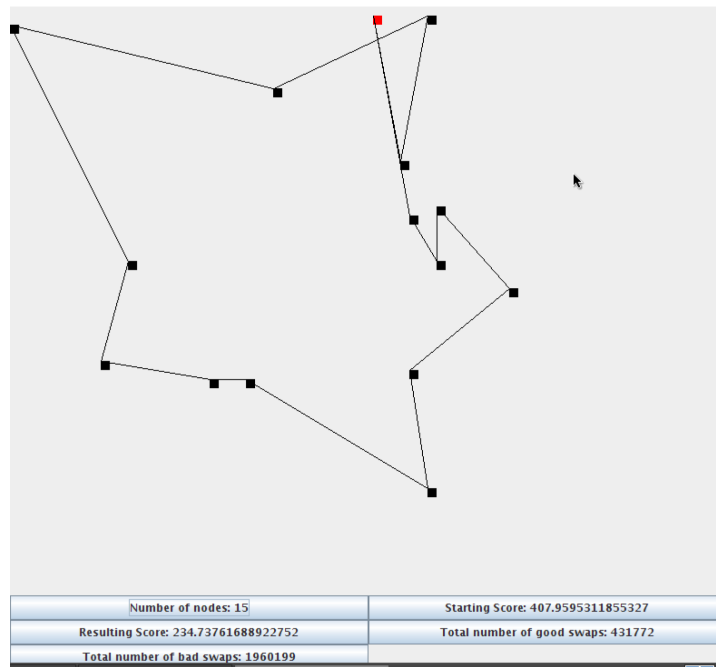

Figure 4.2: Example output of a resulting route

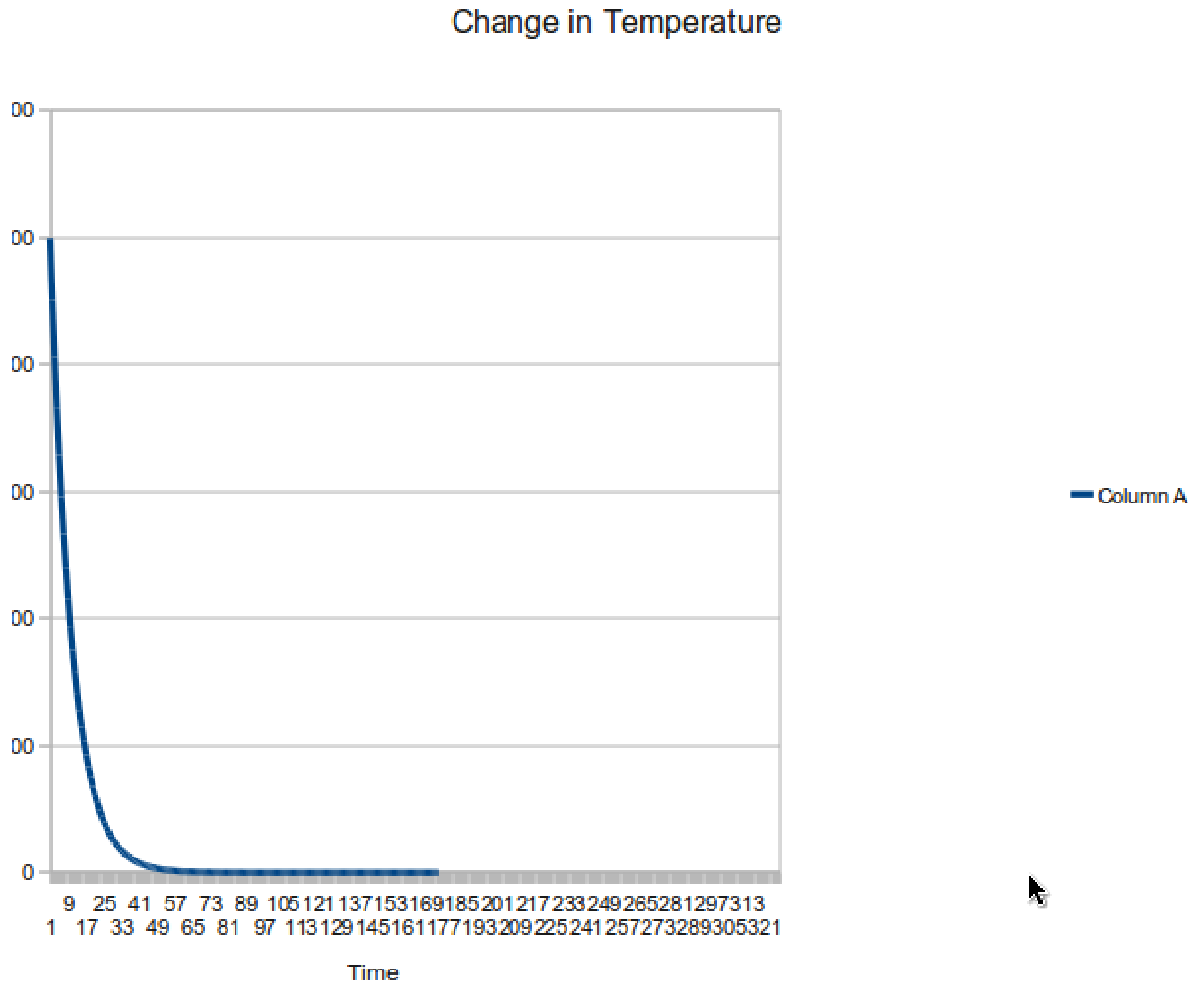

Figure 4.3: Example graph of temperature decreasing with time

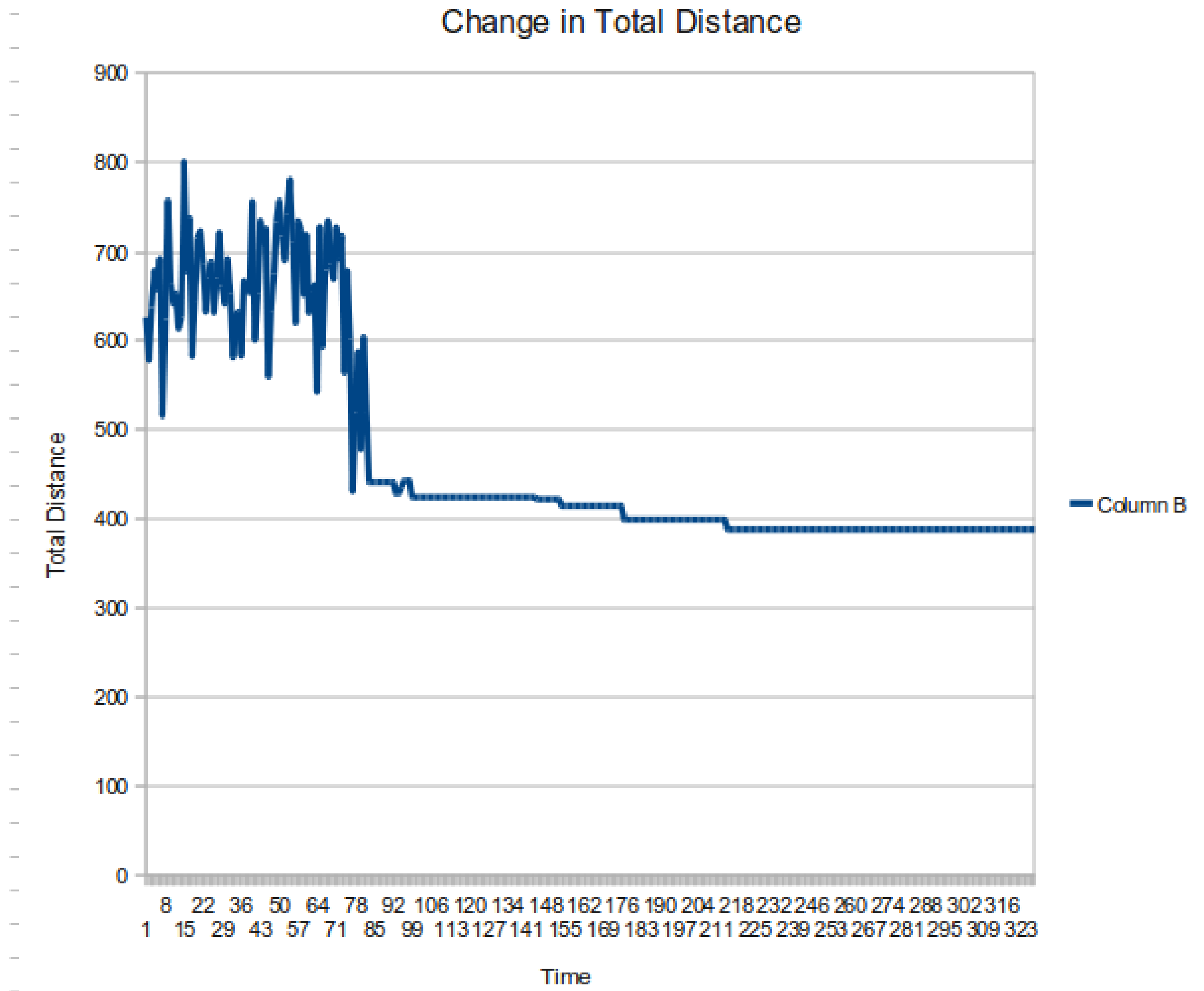

Figure 4.4: Example graph of total distance decreasing with time

- Input and output of data

Overall, the timetabling system isn't wildly different to the travelling salesman solution, as simulated annealing is applicable to them both however a certain amount of tweaking and modification had to be performed on the travelling salesman annealing system to make it compatible with the timetabling system.

4.3.2 Changes

I started by creating the arrays of slot objects, which are similar to the point objects from before, however instead of having coordinates, each slot has an array of integers which stores the indices of every course assigned to it, thus allowing for relatively easy checking to see whether a slot is free or not, and if it isn't then how many lectures are assigned to it.

My next challenge was to modify the annealing method from the travelling salesman solution so that it functioned correctly within the timetabling system. I began by changing the way it evaluated the total distance of the routes through the map of nodes so that instead it goes through the array of course indices for each slot in the timetable, checking for each individual one whether or not it has any lectures assigned to it, and if so, how many. The way the scoring system functions in the timetabling system is quite different to how it worked in the travelling salesman solution, which I will go into further detail about later in this paper.

The next problem to overcome was that of how to best display a visual representation of the timetable when it's finished, but in the end I opted for having a JPanel with an ordered set of JButtons to allow for relatively easy organisation of the labels into some kind of timetable-esque design, with times and days for each column and row with the JButtons in their order below. The reason I opted for JButtons as opposed to JLabels was that they are easier to read due to the individual borders they have, as well as ease of organisation - JButtons are easier on the eye when organised in a large grid than a large block of text in the form of JLabels.

4.3.3 Subsystems

There were quite a few differences between my implementations of the travelling salesman and timetabling simulations, although the most pertinent were as follows:

- Every slot in the timetable can have multiple lectures assigned to it.
- Each slot object has its own individual array of integers which are the indices of the lectures assigned to it.
- Swapping doesn't swap an entire slot object, it only swaps one of the elements from each array of course indices.

Swapping

Listing 4.6: Timetable swapping code

```
1  public void swap(Slot[] list){
2          //Picks two random courses in the timetable and swaps
               their slots.
3          int swap1 = rng.nextInt(slots);
4          int swap2 = rng.nextInt(slots);
5
6          Slot tempSlot1 = new Slot(bestList[swap1]);
7          Slot tempSlot2 = new Slot(bestList[swap2]);
8          int courseIndex1 = 0;
9          int courseIndex2 = 0;
10         int course1 = -1;
11         int course2 = -1;
12
13         if(tempSlot1.courseIndex.size() > 0){
14                 courseIndex1 =
                       rng.nextInt(tempSlot1.courseIndex.size());
15                 course1 =
                       tempSlot1.courseIndex.get(courseIndex1);
16                 tempSlot1.removeCourse(course1);
17         }
18
19         if(tempSlot2.courseIndex.size() > 0){
20                 courseIndex2 =
                       rng.nextInt(tempSlot2.courseIndex.size());
21                 course2 =
                       tempSlot2.courseIndex.get(courseIndex2);
22                 tempSlot2.removeCourse(course2);
23         }
```

```
24
25         if (course1 != -1){
26                 tempSlot2.addCourse(course1);
27         }
28         if (course2 != -1){
29                 tempSlot1.addCourse(course2);
30         }
31
32         copyArray(bestList, swapList);
33
34         swapList[swap1] = tempSlot1;
35         swapList[swap2] = tempSlot2;
36 }
```

As you can see, most of the problems that occured in migrating the simulated annealing system from the travelling salesman problem to the timetabling problem due to the differences in class structure between the two simulations. In the travelling salesman simulation, the swapping of nodes in the route was a relatively simple affair that just took two point object and swapped their positions in the array and thus their positions in the ordering of the route, however in the timetabling simulation certain changes were necessary to ensure that the simulated annealing system functioned correctly on it.

More specifically the changes I made were how the details for each slot stored its respective details for swapping lectures - instead of having an array of points in order and swapping two of these points, the array of slot objects in ordered such that every 8 elements in the array refers to an 8 hour working day with each slot representing an hour. Each of these point objects contains an ArrayList of integers which stores the indicies of the courses that are assigned to it, and when a swap occurs one of these indices is removed from the slot object in question and added to another randomly chosen slot object before the same procedure is run on the other slot object.

Scoring

Listing 4.7: Timetable scoring code

```
1 public double evaluate(Course[] courselist, Slot[] list) {
2         //Calculates the total score for each proposed 'optimal'
              timetable.
3         double totalPoints = 0;
4         int multiplier = 1;
5         int addTo = 10;
6         for(int i=0; i<list.length; i++){
```

```
7           //Steps through each slot in the timetable with
                a course (or multiple courses) and sums their
                 point scores.
8           if(list[i].courseIndex.size() > 0){
9           for(int j = 0; j < list[i].courseIndex.size();
                j++){
10                  totalPoints +=
                        courselist[list[i].courseIndex.get(j)].points;
11                  }
12          }
13      }
14      for(int i=0; i<list.length; i++){
15          //Checks for duplicates/clashes in the
                timetable, and increments the multiplier for
                each clash.
16          if(list[i].courseIndex.size()>1){
17          for(int j = 0; j<list[i].courseIndex.size();
                j++)
18                  multiplier+=
                        courselist[list[i].courseIndex.get(j)].points;
19          }
20      }
21      addTo *= multiplier;
22      totalPoints += addTo;
23      return totalPoints;
24  }
```

One of the other considerations I had to take into account was that of calculating the scores of each respective system state - in the travelling salesman implementation, the scores were calculated simply by calculating the total distance throughout the route by following Pythagoras' Theorem. In this case however the scoring system could not be implemented as simply due to the differences in the problems, although I devised the following system for calculating a system state's score:

- Firstly, for each lecture that has been assigned to a slot, a reference is made to the list of courses and the point score for each course is summed.
- Next, a loop iterates through every element in the array of slot objects, checking to see if they have more than one lecture assigned to them.
- If this is the case, then a 'multiplier' variable is incremented by one for each individual clash that occurs.
- Once the loop has iterated through every element in the array, the sum of

Starting Timetable

	Monday	Tuesday	Wednesday	Thursday	Friday
9	English	CLASH! x3	CLASH! x2	CLASH! x5	CLASH! x5
10	CLASH! x5	History	CLASH! x2	CLASH! x3	CLASH! x3
11	CLASH! x3	CLASH! x3	CLASH! x3	Spanish	
12					
13					
14					
15					
16					

Figure 4.5: Example output of a starting timetable - unordered

	Monday	Tuesday	Wednesday	Thursday	Friday
9	English	Spanish	English	Spanish	Maths
10	History	Spanish	Science	Spanish	Geography
11	Science	Science	Maths	English	Science
12	History	English	History	Maths	English
13	English	Science	Geography	History	History
14	Spanish	English	Spanish	Maths	English
15	English	English	History	Maths	Science
16	Spanish	Science	Science	History	Spanish

Figure 4.6: Example output of a resulting timetable - ordered with no clashes

the point scores for each course is multiplied by this ‘multiplier’ variable giving the total score for that system state.

Some examples of the output of my simulation are shown in figures 4.5 and 4.6:

Chapter 5

Worked Examples

In this section I will show you an example of each of the two simulations - with step by step explanations of the exact process for each system.

5.1 Travelling Salesman

In this example, I will be running the simulated annealing travelling salesman solution with a map of 4 nodes to keep the example simple enough to be able to explain it. I began by running the system and entering in the number of nodes I wish to create before the output shown in figure 5.1 was displayed.

As you can see this is clearly not an optimal route as it is not the logically shortest route from start to finish, however this is the starting route so it shouldn't be optimal. This is where the simulated annealing process begins, as it starts by picking a random point in the route and swapping it with another randomly picked point in the route. This process is repeated until the temperature has decreased to such a degree that it is close to 0, with random swaps occuring with every iteration of the annealing process and the proposed optimal route being evaluated against the first route until an improved route is found, then that route replaces the original as the optimal route and the process continues. Eventually, when the temperature has dropped by such an amount that the program exits the route that is deemed as optimal upon termination is returned via the return() method as the resulting route of the process. This route could be the optimal solution to the problem, but the way the simulated annealing process works is that it outputs probably good results i.e. they cannot be proven to be optimal, and the output of the example we are running here is

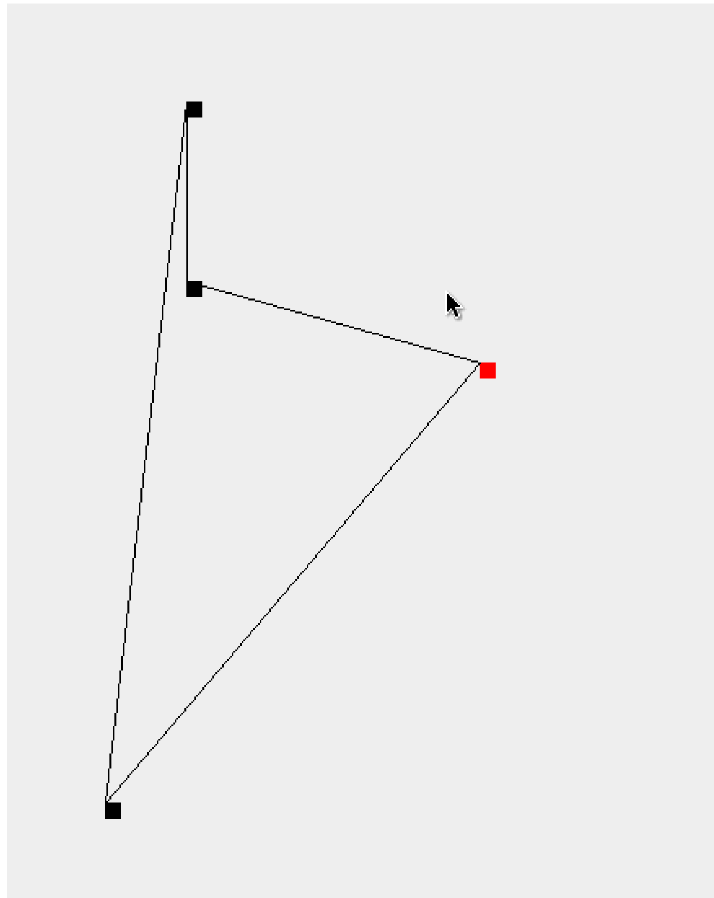

Figure 5.1: Output of a starting route

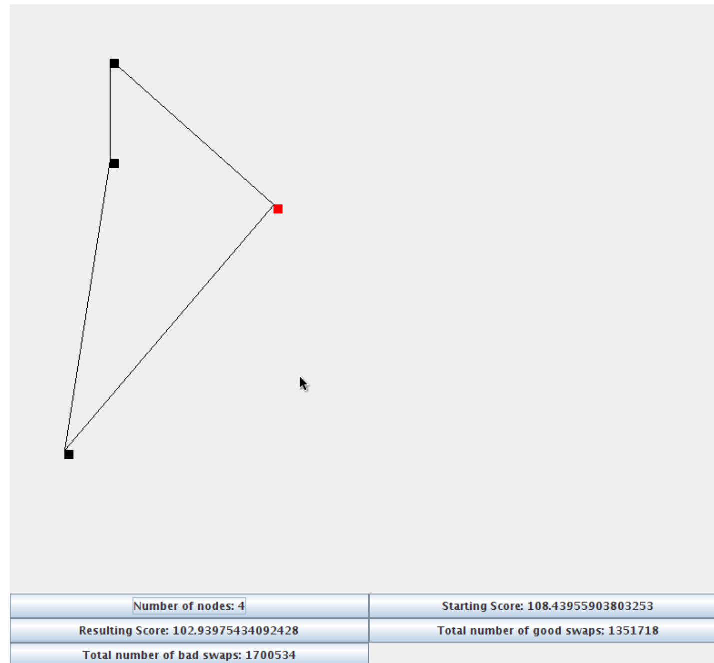

Figure 5.2: Output of a resulting route

shown in figure 5.2.

5.2 Timetabling Simulation

In this example, I will be running the timetabling simulation with a randomly generated timetable full of courses randomly assigned to random slots in the timetable. I began by running the actual timetabling system and the starting output is shown in figure 5.3.

As you can see from the initial output of the simulation all of the courses have been assigned to slots in the timetable, however only about half of the slots actually have courses assigned to them. Of those that do have courses assigned to them, most of them have clashes due to the random fashion in which the courses are assigned to slots when the simulation is initially run. This is where the simulated annealing process commences, with the total score of the starting system state being calculated and used for comparison with the produced

	Monday	Tuesday	Wednesday	Thursday	Friday
9	CLASH! x3	CLASH! x2	CLASH! x5	CLASH! x3	CLASH! x3
10	CLASH! x3	CLASH! x5	CLASH! x4	CLASH! x3	CLASH! x3
11	Maths	CLASH! x3	English	Spanish	
12					
13					
14					
15					
16					

Figure 5.3: Output of an unordered timetable - with clashes

system states up until the initial system state is replaced by a 'better' one. For every iteration of the annealing process, as in the travelling salesman solution, two random slots in the array of slots are picked, and within these two slot objects a single random course is picked for each (that had been assigned to that slot), and the two indices are swapped between the two slots, thus performing a random swap of two courses in the timetable. This process is repeated, with the evaluation of each respective system state determining whether or not it should be accepted as the new 'optimal' system state, until as in the travelling salesman solution the temperature tends to 0 and the 'optimal' system state at that time is returned as the resulting, ordered timetable.

The timetable output from this example is show in figure 5.4, and as you can see all clashes have been removed, and each of the courses has been assigned its own slot.

	Monday	Tuesday	Wednesday	Thursday	Friday
9	History	English	Spanish	Spanish	PE
10	History	PE	PE	Maths	History
11	Maths	English	Spanish	PE	English
12	History	PE	Maths	Spanish	Maths
13	History	Maths	History	History	Spanish
14	English	IT	PE	IT	IT
15	Maths	Maths	History	English	Spanish
16	Spanish	Maths	English	English	PE

Figure 5.4: Output of an ordered timetable - with all clashes removed

Chapter 6

Testing

Due to the limited interaction the user has with my system, there weren't many tests to perform from the users point of view as there is very little that can cause the system to produce an error or even crash. Coupled with the fact that I developed this entire system myself and tested the system as I was implementing it to ensure that it worked, there really were very few tests I could perform from a verification or validation point of view on my two respective solutions.

Test Number	1
Test Details	Do the 2 buttons which start the respective simulations work?
Expected Outcome	Clicking on either of the two buttons will run the respective simulation.
Actual Outcome	Clicking on either of the two buttons runs their respective simulation.
Conclusion	The buttons are functioning as intended.

Test Number	2
Test Details	Does the button which runs the travelling salesman solution work?
Expected Outcome	Clicking on the button will run the travelling salesman solution with the number of nodes specified.
Actual Outcome	Clicking on the button runs the solution as expected.
Conclusion	The button is functioning as intended.

Test Number	3
Test Details	Does the JTextField which takes in the number of nodes to create only accept integers?
Expected Outcome	Entering anything other than a positive integer should change the label's text to read "Please only enter an integer > 3!", without the system crashing.
Actual Outcome	The label's text changed as expected.
Conclusion	Input validation works as expected.

Test Number	4
Test Details	Does closing any of the windows stop the system running and close the rest of the system?
Expected Outcome	Closing any of the windows produced by the system should exit the system, and close any remaining open windows.
Actual Outcome	All windows close, and the system exits correctly.
Conclusion	The windows all function correctly when they are closed.

Test Number	5
Test Details	Does the simulated annealing algorithm only accept states where delta < 0 or the Boltzmann formula accepts them?
Expected Outcome	Only system states with an absolute improvement in score or whose scores are accepted by the Boltzmann formula should be accepted.
Actual Outcome	Only the intended system states are accepted with 'good' moves.
Conclusion	The annealing system is functioning as intended.

Chapter 7

Assessment

Due to the nature of the project I chose there was very little in the way of books/literature for me to reference when actually implmenting my system, or even when I was researching the theory before starting the implementation. Therefore I have no bibliography or other acknowledgements other than to my advisor throughout this project, Dr.Adrian Johnstone as he taught me what I needed to know to complete my project implementation.

7.1 Problems and Enhancements

7.1.1 Copying Arrays

Throughout the duration of my project, I encountered several problems that I had to overcome in order to achieve a fully functioning simulated annealing system. One of the most pertinent of these problems occured when I was trying to create a duplicate of the array of points in my travelling salesman solution, the problem I encountered was that every time a random swap occured on two of the nodes in the array, the optimal array would be set to the new randomly swapped array no matter whether the score was better or worse than the optimal solution. I discovered after some hardship that the problem I was encountering was occuring when I was setting the array that was to have two of its elements swapped to be equal to the currentky accepted optimal array. I concluded that this problem as instead of creating a copy of the array, it was merely pointing both array variables to the same array of point objects, meaning that when a swap took place, it affected both array rather than just the one that was initially

intended to have two of its elements swapped.

I overcame this problem eventually by implementing the copyArray method, which is explained in Section 4.2.2 of this report, which handled the problem stepping through each element of the array to be copied, and copies each point object to the corresponding element in the other array.

7.1.2 Boltzmann Formula

Actually implementing the simulated annealer, or more specifically the Boltzmann formula used in deciding whether or not to accept new system states as the optimal state proved a rather difficult task due to the intricate nature of the system. What I mean by this is the inter-dependance of the various variables and loops in the annealing system, such as setting the initial temperature and the rate at which it decrements as elements such as these are crucial to the proper functioning of the system. Tweaking the annealer to a state that I deemed acceptable was quite a time consuming task, as upon changing some of these values the system seemed to break, so it was vital that I found the correct configuration of all the system variables to allow the system to run as expected.

The values relating to the Boltzmann formula in specific were the most difficult to configure correctly, as the formula has to be exactly right otherwise only absolute improvements on the system state would be accepted, which was a problem I ran into during the implementation of my system. In the end I managed to overcome this by tweaking each individual part of the simulated annealer, then running the system to see what changes occured (if any), and eventually it worked to my satisfaction.

7.2 Realisations

Throughout all of this project, I have encountered certain situations that have forced me to make realisations not just about programming but also the theory behind simulated annealing. One of the most important realisations I had was that I am capable of overcoming almost any problem I encountered without asking for help, I merely needed to test myself and have the confidence in my ability to get things wrong and learn from them. When I ran into such problems as the array copying problem mentioned above it took much contemplation and

debugging to find the source of the problem, but I managed to solve most of my problems myself before having to ask for help.

Whilst implementing the solutions for both the travelling salesman and the timetabling problems, I was required to have quite a thorough knowledge of the simulated annealing process in order to implement it effectively. From the start of my project, I struggled to grasp the theory behind the simulated annealing process when I was studying it purely from a theoretical point of view, however when I came to begin the implementation of my two solutions, I found that I would need to get my head around the theory in order for the project to work at all. Once I had begun implementation though I found the simulated annealing process a lot easier to comprehend as I was able to visualise the process once I had a practical application for it. In the end I managed to get my head around the theory mostly through a lot of trial and error with emphasis on the errors due to the intricate nature of the annealing process - every variable is crucial to the system functioning correctly and had to have exactly the right value.

Other than these considerations, I only wish I could have know of more practical applications for the simulated annealing process early on in the project's lifetime, as I believe that if I had understood the theory earlier than I eventually managed to it would have made the whole implementation process a lot easier and more efficient. That said, I am glad that I encountered the problems that I did encounter as they forced me to learn lessons that I may not have learned otherwise to proceed in my implementation.

7.3 Critical Analysis

Throughout the entirety of my project, I had to learn a new method for solving combinatorial optimisation problems such as the Travelling Salesman problem and the Timetabling problem and during this learning process I came to make certain realisations about the simulated annealing process and its underlying theory. One of the main realisations I came to was that even though the two different implementations that I chose for this method are vastly different the theory behind them when referring to the simulated annealing system is the same, it's just that the two implementations require two quite different approaches to finding a solution. To be more specific, the two separate problems require two different approaches from a programming point of view, but the simulated annealing process doesn't change - in one, swapping two nodes and

re-evaluating every iteration is what is required, whereas a timetable has to be set out in an entirely different fashion. This is one of the main reasons, as I mentioned earlier, that I wish I had found more practical applications for the simulated annealing process other than the actual metallurgical annealing process it is based on, and the travelling salesman and timetabling problems.

Chapter 8

User Guide

This section is a guide for users on how to compile and then run my system on their own computers.

8.1 Compiling the system

The system comes as a collection of .java files which need to be compiled to .class files in order for them to be run by the Java Virtual Machine. To do this, you need only run the command "javac *.java" from a command line once you have moved to the directory containing the source files. Once the code has been compiled, running the system is as simple as running the command "java Runner" once again from the command line, when you are in the same directory the source files were stored and compiled to.

8.2 Using the system

Upon first running the system, you will be presented with a window with two buttons in, that looks as follow:

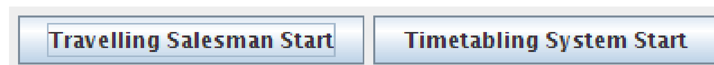

Figure 8.1: The first window that is displayed upon running the system.

As you can see, there are only two options from here, either click on the button which runs the travelling salesman system, or the button which runs the timetabling simulation.

8.2.1 Travelling Salesman

Once you click on the button which starts the travelling salesman system, you will be presented with a window that looks as follows:

Figure 8.2: The settings window for the Travelling Salesman system.

Once you have entered your preferred number of nodes to use in creating the map of points, and clicked on the "Start Simulation" button, the system will start to run. Once the system has completed, you will be presented with two windows that looks as follows:

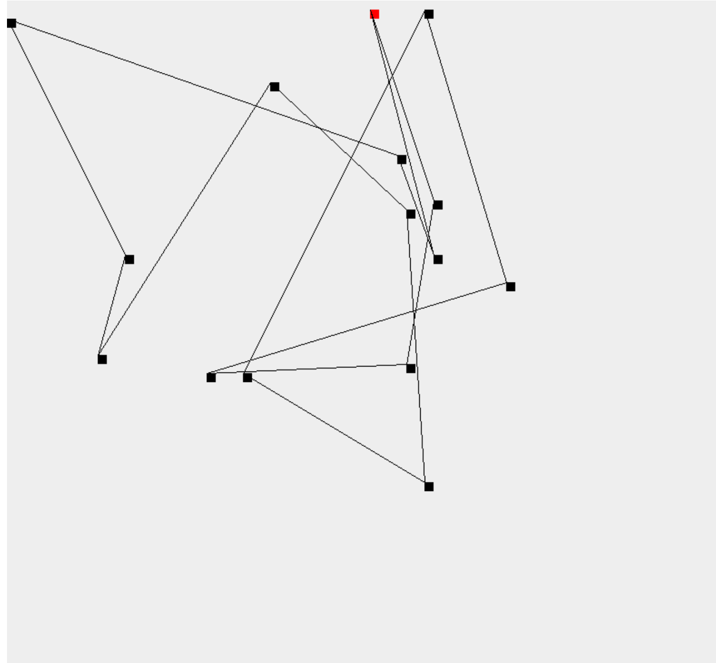

Figure 8.3: The window that shows the starting route of the TSP

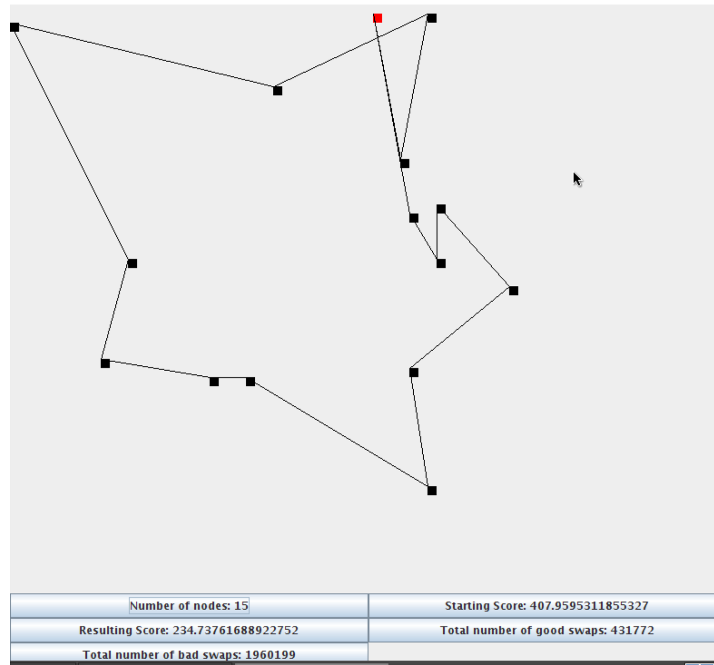

Figure 8.4: The window that shows the resulting route and numerical results.

These are the two windows that show you the starting route the system began the annealing process on, and the resulting route that the system gave as an output along with the specific numerical results of the simulation.

8.2.2 Timetabling Simulation

Once you click on the button which starts the timetabling solution, there are no settings to be changed in the system each time it's run as most of the content is randomly generated, so the system runs as soon as you click on the button. Once the process has completed, you will be presented with two windows:

Starting Timetable

	Monday	Tuesday	Wednesday	Thursday	Friday
9	English	CLASH! x3	CLASH! x2	CLASH! x5	CLASH! x5
10	CLASH! x5	History	CLASH! x2	CLASH! x3	CLASH! x3
11	CLASH! x3	CLASH! x3	CLASH! x3	Spanish	
12					
13					
14					
15					
16					

Figure 8.5: The window that shows the starting timetable which the annealing process is started on.

	Monday	Tuesday	Wednesday	Thursday	Friday
9	English	Spanish	English	Spanish	Maths
10	History	Spanish	Science	Spanish	Geography
11	Science	Science	Maths	English	Science
12	History	English	History	Maths	English
13	English	Science	Geography	History	History
14	Spanish	English	Spanish	Maths	English
15	English	English	History	Maths	Science
16	Spanish	Science	Science	History	Spanish

Figure 8.6: The window that shows the resulting, ordered timetable output from the annealing process.