Efficient Methods and Techniques for the Open-Shop Scheduling Problem

By

Steve Bou Ghosn

Submitted in partial fulfillment of the requirements

For the Degree of Master of Science

Thesis Advisor: Dr. Haidar Harmanani

Computer Science and Mathematics Division

LEBANESE AMERICAN UNIVERSITY

February 2007

LEBANESE AMERICAN UNIVERSITY

School of *Arts* and Sciences

Thesis Approval

Student Name	STEVE BOU GHOSN	I.D.#: *199805410*
Thesis Title:	*EFFICIENT METHODS AND TECHNIQUES FOR THE OPEN-SHOP SCHEDULING PROBLEM*	
Program:	**Computer Science**	
Division /Dept:	**Computer Science and Mathematics**	
School:	School of Arts and Sciences, Byblos	
Approved by:		
Thesis Advisor:	*DR. HAIDAR M. HARMANANI*	[redacted]
Member	*DR. DANIELLE AZAR*	
Member	*DR. CHADI NOUR*	
Member		
Date:	NOVEMBER 29, 2006	

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Haidar Harmanani for his guidance throughout my M.S. studies. I would also like to thank Dr. Danielle Azar and Dr. Chadi Nour for being on my Thesis committee. Finally, I would like to thank my friends and family for their continuous support.

Table of Contents

Table of Figures

Table of Tables

Abstract

In this paper we investigate the use of two different heuristic techniques to the open-shop scheduling problem and we make a comparison between them. The open-shop scheduling problem is NP hard and due to that it's very important to find heuristic approaches that can generate better approximate solutions. This work first focuses on solving the open-shop scheduling problem using genetic algorithms. We present an interesting implementation of genetic operators that combines the use of deterministic moves and pure random moves. We then perform tuning and testing of our approach and present detailed results for each problem instance of the Taillard benchmarks. We also compare our results with those obtained in other recent research works on the subject. In the second part of our work we focus on an approach based on simulated annealing. We perform tuning and testing for our annealing approach and present detailed result comparisons for all the Taillard Benchmarks. Finally we compare both the results obtained by ga and annealing and conclude that even though all results are good, in general our annealing implementation seems to perform better than our GA implementation, especially for larger problem sizes. We also justify the reasons why we think our ga approach didn't perform as good as the annealing.

Chapter 1 Introduction

1.1 The Job Shop Scheduling Problem

The general job shop scheduling problem is an optimization problem with tremendous theoretical and practical importance, and has been the focus of intensive research during the last decade. A job shop consists of a number of jobs to be processed, each of these jobs is composed of tasks or operations, and each of these tasks is to be processed on a particular processor. There are a limited number of *processors* or *machines*, each of which performs a different task. A schedule for a given processor specifies the exact time interval, *start time* and *finish time*, during which each job in the shop is to be processed in that particular processor. A schedule for the shop is a set of processor schedules, one for each processor and each schedule satisfies resources constraints. Among the individual processor schedules, the one with the latest completion time determines the overall finish time of the shop schedule, also known in the literature as *makespan*. The optimal schedule (or optimal *makespan*) is the one which has the least finish time among all possible schedules.

1.2 The Open Shop Scheduling Problem

In the general job shop problem the tasks that compose a job are usually dependent on each other, and should be processed in a particular order. The open shop scheduling problem (OSSP) is a variation of the general job scheduling problem and it is the main focus of this work. The open shop scheduling problem differs from the general

job shop problem mainly in that the tasks that compose a job can be processed in any particular order. A real life situation that illustrates the open shop job scheduling problem could be that of a large automotive garage with specialized areas. A car may require the following work: *replace exhaust pipes and muffler*, *align wheels*, and *tune up*. These three tasks may be carried out in any order. However, since the exhaust system, alignment, and tune-up sections are in different buildings, it is not possible to perform two tasks simultaneously.

Stated formally, an open shop consists of $n \geq 1$ jobs and $m \geq 1$ processors, each job i ($1 \leq i \leq n$) is composed of m tasks. The processing time for task j of job i is t_{ji}, task j of job i is to be processed on processor j, where $1 \leq j \leq m$. A schedule for a processor j is a sequence of *tuples* <i, start_time(i), finish_time(i)>, that implies job i is processed continuously on processor j from start_time(i) to finish_time(i). There may be more than one *tuple* per job and it is required that each job i spends exactly t_{ji} total time on processor j.

A non preemptive schedule is one in which the individual processor schedule has at most one *tuple* <i, s(i),f(i)> for each job i to be scheduled. A schedule in which no restriction is placed on the number of *tuples* per job per processor is preemptive. All non preemptive schedules are also preemptive, while the reverse is not true.

The open shop scheduling problem is also interesting from the theoretical standpoint, despite it being seemingly simpler than the general job shop problem due to its lack of ordering constraints. This is because, as previous research has shown, removing the ordering constraints allows solving efficiently the job shop problem only if we use preemptive scheduling, but not the non preemptive one. In fact, determining the optimal non preemptive schedule for the open shop is an NP hard problem [1].

To illustrate the OSSP, we present in Table 1 a small but popular benchmark instance of a problem consisting of 4 jobs and 4 processors:

Jobs / Processors	Job 1	Job 2	Job 3	Job 4
Processor 1	5	24	29	43
Processor 2	70	80	56	64
Processor 3	45	58	29	45
Processor 4	83	45	61	74

Table 1: A 4 x 4 benchmark problem for the OSSP

In Table 2 and Figure 1 we present a possible schedule for the sample instance in Table 1 that isn't optimal, followed by an optimal schedule shown in Table 3 and Figure 2.

Machine	Job	Operation	Operation Length	Start Time	End Time
1	3	1,3	29	0	29
1	4	1,4	43	80	123
1	2	1,2	24	150	174
1	1	1,1	5	233	238
2	2	2,2	80	0	80
2	1	2,1	70	80	150
2	4	2,4	69	150	219
2	3	2,3	56	219	275
3	1	3,1	45	0	45
3	2	3,2	58	80	138
3	3	3,3	29	150	179
3	4	3,4	45	219	264
4	4	4,4	74	0	74
4	3	4,3	61	80	141
4	2	4,2	83	150	233
4	1	4,1	45	233	278

Table 2: Schedule for benchmark problem in Table 1 with the makespan = 278

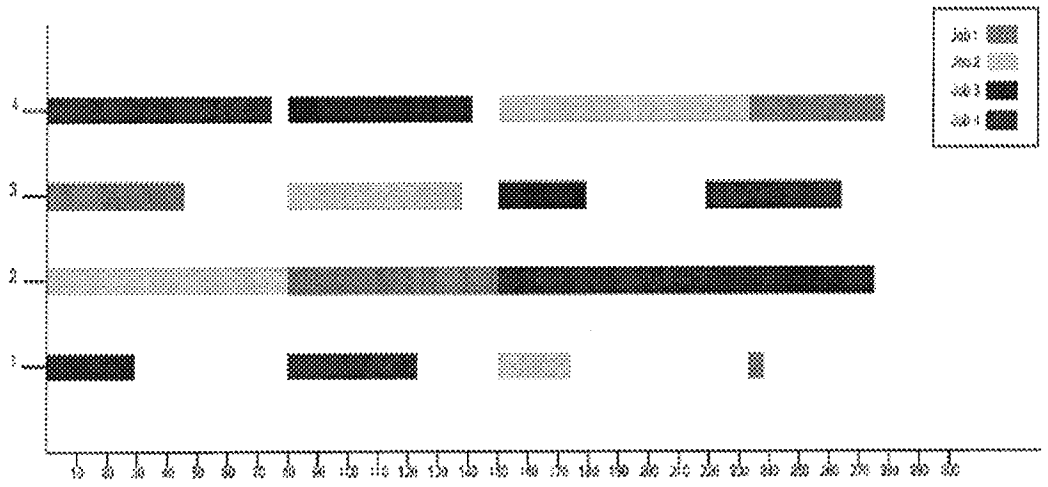

Figure 1: A chart representing the results in Table 2

Machine	Job	Operation	Operation Length	Start Time	End Time
1	3	1,3	29	0	29
1	4	1,4	43	80	123
1	1	1,1	5	150	155
1	2	1,2	24	219	243
2	2	2,2	80	0	80
2	1	2,1	70	80	150
2	4	2,4	69	150	219
2	3	2,3	56	219	275
3	1	3,1	45	0	45
3	2	3,2	58	80	138
3	3	3,3	29	150	179
3	4	3,4	45	219	264
4	4	4,4	74	0	79
4	3	4,3	61	80	141
4	2	4,2	45	141	186
4	1	4,1	83	192	275

Table 3: Optimal schedule for problem in Table 1 with makespan = 275

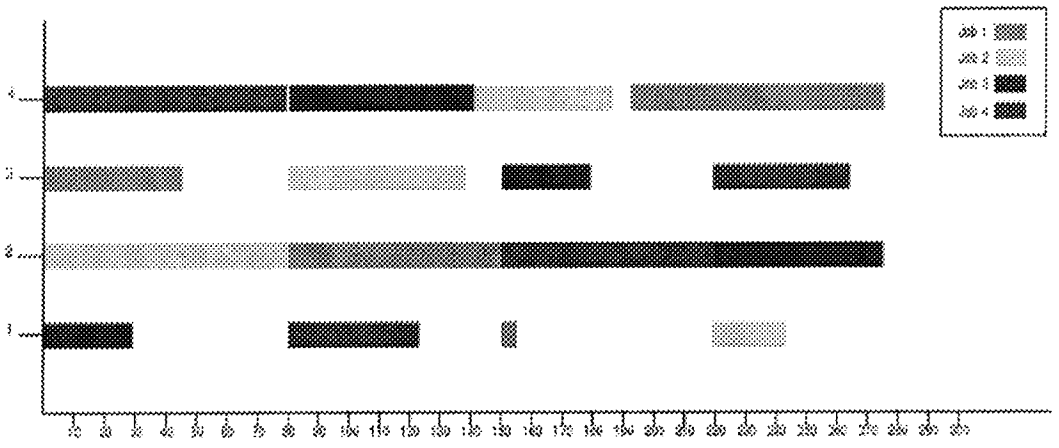

Figure 2: A chart representing the results in Table 3

We can establish two partial lower bounds for the optimal schedule *makespan*. The first one is the maximum completion time among the jobs to process. Every job must

necessarily take a processing time which is at least the sum of the times required to perform each of its component tasks. Therefore, the optimal *makespan* can't be smaller than the maximum completion time among the jobs. The second lower bound is the maximum completion time for the jobs allocated to a given processor. Every processor must necessarily take a processing time which is at least the sum of the times required to perform each of the jobs allocated to it. Therefore, if t_{ji} is the time that is allocated to job i in processor j, then:

$$L1 = \max_{i=1}^{n} \left(\sum_{j=1}^{m} Tij \right)$$

$$L2 = \max_{j=1}^{m} \left(\sum_{i=1}^{n} Tij \right)$$

The maximum of these two partial lower bounds give us a final lower bound for the optimal makespan:

$$L = \max\left(\max_{i=1}^{n} \left(\sum_{j=1}^{m} Tij \right), \max_{j=1}^{m} \left(\sum_{i=1}^{n} Tij \right)\right)$$

Equation 1: Lower Bound of Optimal makespan

Equation 1 shows a lower bound for the optimal makespan that can be easily determined in all cases. The optimal makespan can never be less than the lower bound but is not necessarily equal to the lower bound. In fact, as previously

mentioned finding the optimal makespan for the non preemptive open shop scheduling is an NP complete problem. Because of that, most of the research has focused on finding efficient heuristic methods that generate approximate solutions, such as genetic algorithms, simulated annealing and tabu search. But there has been also some research done on exact methods like branch and bound.

1.3 Branch and Bound

The general idea of Branch and Bound is to divide the problem's search space into smaller subregions and then discard some of those regions that are determined to be redundant or useless in order to find the optimal solution. A branch and bound procedure basically relies on two strategies or methods, a branching and a bounding strategy. The branching strategy is in charge of covering the feasible area of the search space by several smaller feasible sub-regions (for example by portioning it). This procedure is then applied recursively to each of the sub-regions, forming a tree structure denominated the search tree or branch and bound tree. The bounding strategy determines updated lower and upper bounds per iteration and determines which areas of the search tree are now out of bounds and therefore can be safely discarded. How good a branch and bound algorithm is depends directly on the quality of the branching strategy and the bounding strategy, if these are well tailored for the problem at hand, then the search performance can increase dramatically as the algorithm would just be focusing on relevant search areas.

1.4 Genetic Algorithms

Genetic algorithms (GA) are approximate techniques to find solutions to complex optimization problems. These solutions, despite not being exact, are usually very close to the optimal. Genetic algorithms in a way mimic the evolutionary process of nature by trying to select the best out of a group of solutions(natural selection) and by recombining promising solutions into a new one to generate a new "offspring" that could be closer to optimal. Genetic algorithms represent the possible solutions of a problem in the format of *chromosomes*. A set of chromosomes (candidate solutions) represents a population.

An initial population is determined randomly, then the fitness (the potential of a solution to be the optimal) of all chromosomes/solutions is computed. Based on the fitness of each chromosome, the algorithm then proceeds to either combining different chromosomes to generate new ones or to randomly modify particular chromosomes individually. The former is called crossover or *sexual* reproduction (since it involves a pair of chromosomes) and the latter is called mutation or *asexual* reproduction. Crossover and mutation are the main operators used by genetic algorithms in order to generate a new evolved population that becomes the input for the next iteration of the algorithm. The idea is that on each iteration the population of solutions should further evolve into fitter solutions that are closer and closer to the optimal, the iterations are usually called generations.

In order to use genetic algorithms to solve a problem, we require first a valid way of encoding a solution so that it can be can be manipulated by the algorithm. For example, a solution could be represented by a group of numbers, characters, boolean values, etc. We also require a way to evaluate how good a given solution is, therefore we must supply a fitness function that analyzes a chromosome and determines how fit it is with respect to the problem we are attempting to solve. Additionally, we require appropriate genetic operators (crossover and mutation) which should be tailored specifically to the problem, because the performance of the GA and the quality of the solutions we obtain will greatly depend on them. Last but not least, we need a termination criterion that could be simply a limit on the number of generations or iterations.

1.5 Simulated Annealing

Simulated Annealing is one of the most popular heuristic techniques used for solving optimization problems. Some of the optimization problems that have been tackled using annealing are: the *traveling salesman*, *graph partitioning*, *linear arrangement*, *job scheduling* and many others. Annealing basically models the process by which a metal is heated at very high temperature to the extent that its atoms gain sufficient energy to break their chemical bonds and become free to move, and its respective cooling process afterwards, by which the temperature is lowered gradually in order for the atoms to solidify back into a structure. In order for the heated material to properly crystallize, a proper initial heating temperature and a proper cooling rate

must be used. When applying this to optimization problems, we map material to solutions that are being altered. The initial temperature is the state at which we have our initial solution and during the cooling process we try to alter the solution gradually, following a schedule, such that by the time the temperature is close to 0, we would obtain a solution which is the optimal or very close to the optimal. By altering the solution we basically mean exploring its close neighborhood and trying to find a better neighbor solution which would become the current solution. If we choose the wrong initial temperature or a cooling rate that is too slow or too fast, we might end up with a solution of poor quality.

The main property of annealing that makes it very effective is the fact that it doesn't only accept improving solutions but it also has some tolerance to accept solutions which have a worse value than the current. This is very important because it prevents the stagnation of the solution in local optima and allows a more thorough exploration of the search space. The heuristic determines whether to accept a solution or not basing on the current temperature, if the temperature is high then the probability of accepting a given solution even if it has a lower score is high, as the temperature decreases also the probability of choosing non improving solutions decreases; This is all matching the annealing process in which as a given metal starts to be cooled in a controlled manner, the atoms of the metal will be able to move with more freedom at higher temperatures than at lower temperatures. The movement of atoms is analogous to replacing current solutions with neighbors during the optimization process.

The probability used in annealing in order to determine if we should accept a move at a given temperature is:

$$\Pr ob = e^{-(\Delta E / K_B T)}$$

Where T denotes the current temperature, K_B is the Boltzmann constant and ΔE is the difference between the value of the solution we are trying to choose and the value of the current solution. The complete simulated annealing process is shown in Figure 3:

```
Algorithm Simulated Annealing( S0,T0,α)
(S0 is the initial solution)
(T0 is the initial temperature)
(α is the cooling rate)

Begin
  T = T0;
  CurS = S0;
  BestS =curS;
  CurCost = Cost(CurS);
  BestCost = Cost(BestS);
  Repeat
        T = αT;
  Until (Time = 0);
  Return (BestS);
End
```

Figure 3: Simulated annealing algorithm

1.6 Problem Description and Thesis Outline

In this thesis we will develop and evaluate heuristic solutions to solve the OSSP using GA and simulated annealing. We will first present a study of the most important research done concerning the open shop problem. All related work and our comments are discussed in Chapter 2. Then we will dedicate Chapter 3 to explain our approach to solve the open shop problem using genetic algorithms. We will also have a section dedicated exclusively to discuss testing and results in this chapter. In Chapter 4 we will expose a different approach based on Simulated Annealing. A section will be dedicated to presenting the results obtained in details. Finally in Chapter 5 we compare both approaches and present our final conclusions.

Chapter 2 Related Work

One of the earliest and most significant efforts related to the open shop scheduling problem is the work by Gonzalez et al [1]. In their work, they propose a linear time algorithm to find the schedule with the optimal (minimum) finish time for an open shop with two processors, and a polynomial time algorithm to find the optimal schedule for an open shop with three processors or more on the condition that preemption was allowed. In the case of finding a non preemptive optimal schedule for an open shop with more than three processors no exact algorithm was found that could solve it in polynomial time. Furthermore, the authors proved that the problem is NP hard by reducing it to the partition problem. The partition problem is an optimization problem that belongs to a set of very complex problems collectively named NP complete, these problems share the peculiarity and differ from other problems in computer science and operations research in the fact that researchers don't seem to find any algorithm that solves the problem efficiently for large input. To be considered efficient an algorithm should solve the problem in an order of polynomial time[1].

---

[1] A problem is considered NP hard if an NP complete problem can be reduced to it, meaning that if a technique was found to solve one of the problems, then the same technique could be applied to solve the other.

2.1 Exact Methods

Exact Methods or algorithms are those that obtain a solution for a problem that can be proven and verified to be correct. Many exact methods are considered exhaustive because they analyze a large percentage of the total search space of a problem. Some advanced optimization algorithms like Branch & Bound can obtain exact solutions while greatly reducing the extent of the search space region that they have to process. Exact methods (even the most powerful ones) aren't efficient when solving NP complete problems due to the humongous search space of these problems. The algorithms can find the optimal solution but take a lot of time when the input is moderately large; the performance they offer doesn't allow solving the open shop efficiently for large input. Despite that, researchers have proposed a few exact algorithms that can be useful in solving the open shop for some particular cases. The most popular of these methods is Branch and Bound.

2.1.1 Branch and Bound

In general Branch & Bound doesn't seem to be suitable to solve the job shop scheduling problem, for example Carlier & Ipinson [2] came up with a branch and bound algorithm that produced quite good results but took too much time even for medium scale benchmarks like the 10 x 10 (10 jobs and 10 machines). This was because the method relied heavily on schedule generation and this proved to be infeasible when dealing with the large search space of the job shop problem. Since

the open shop has even a bigger search space than the general job shop problem, it can be safely guessed that the branch and bound might not be the best approach to tackle this problem. That said, there have been a few efforts to apply branch and bound to the open shop and the results obtained are quite acceptable. One of the most impressive works is that of Brucker et al [3], who based their initial efforts on the resolution of a one machine problem with positive and negative time lags. Their later work aimed at fixing in each node the disjunctions on the critical path of a heuristic solution. This method worked for most problem instances, but some problems of size 7 were still not solvable. Gueret et al [4] improved on Brucker's algorithm by using intelligent backtracking, that is when discarding a node not just going back to the direct parent, but rather proceeding further through the ancestors until finding a more relevant section that would avoid losing any optimal solutions.

2.2 Approximate and Heuristic Methods

A heuristic is a technique that solves a problem by obtaining results or solutions that can't be directly proven to be correct. Heuristics are useful due to their ability to produce good solutions that even though not necessarily optimal are quite close to the optimal, in a reasonable amount of time. Because of the large search space in NP complete problems, exact methods can never find an optimal solution in acceptable time if the input is too big; therefore recent research has focused more on heuristics in order to solve these problems. Heuristics sacrifice accuracy in exchange of computational performance and conceptual simplicity. Some examples of very

powerful and popular heuristics are genetic algorithms, simulated annealing, ant colony optimization and tabu search. Concerning the open shop scheduling, most of the recent research is based on genetic algorithms, but there are some very innovative approaches that combine the use of GA with another heuristic like Tabu Search in order to obtain improved results. We will first discuss the pure GA approaches in the literature and then we will follow with a discussion of the hybrid GA approaches.

2.2.1 Uniform Genetic Algorithm approaches

The first researcher to use genetic algorithms to attempt to solve job shop scheduling problems using genetic algorithms was Davis [5]. His implementation wasn't very refined; it used a memory intensive chromosome representation and very simple genetic operators. Still his contribution was important, as he proved the feasibility of using GA and set the basis for future efforts. Among other subsequent meaningful efforts of applying GA to job shop scheduling problems are those of Nakano [6] who represented the chromosome using a binary encoding and used special algorithms for repairing genomes. His work was able to obtain results that were competitive with Branch & Bound results in less time, but wasn't able to improve on the best results obtained. Fang, Ross and Corne [7] were able to improve on Nakano [6] by using the ordinal representation that Grefenstette et al [8] applied to the TSP. This representation had the advantage of producing only valid schedules when altered by the mutation and crossover operators, allowing more accurate solutions in

considerably les time. They also suggested how to adjust their approach to work with the open shop scheduling problem, and later in Fang, Ross and Corne [9] they actually adapted their previous approach to the open shop scheduling problem and implemented it obtaining good results.

Fang, Ross and Corne [9] basically use chromosomes of size 2p where p is equal to the number of jobs multiplied by the number of operations per job (the number of operations per job is equal to the number of machines), and allow each gene in the chromosome to have a value which is in the range [1, j], where j is the total number of jobs. The fitness function interprets a chromosome by analyzing each pair of consecutive genes in the following way: The first gene in the pair is interpreted as an operation index x, while the next gene is a job index y. Thus the chromosome is next interpreted as to basically schedule the x^{th} unhandled operation of the y^{th} unhandled job as early as possible. In order to schedule jobs and operations they use a schedule builder that keeps track of which jobs have already been handled and which operations have been handled for each job. The schedule builder accomplishes this task by using a circular list[2]. The fitness function uses the schedule builder to always schedule jobs and operations in the earliest possible slot and keeps track of and returns the highest finishing time which is the *makespan* of the schedule and the fitness of the chromosome.

---

[2] If the values of the genes are bigger than the size of the list, the modulus operand is applied with the size of the list to find the actual index in the list that the value represents.

2.2.2 Hybrid Genetic Algorithm Approaches

Even though the uniform GA approaches marked a clear improvement over the performance of the exact methods, the results obtained still left room for improvement. Many recent research efforts attempted the use of a hybrid GA instead of the classic pure approach in order to improve on the quality of the solutions. This involves combining the use of GA with another heuristic. Usually these approaches use the heuristic to perform local search, while allowing the GA to perform the search for global optima through the subset of local optima.

Fang, Ross and Corne [9] proposed two different hybrid GA implementations. They represented the solution using the same style of chromosomes they used in their pure GA approach but with the difference that only job indexes were encoded and not the operations. Therefore they used chromosomes of size p, half the size of their pure GA chromosome representation. In their first hybrid implementation, they decide on a fixed heuristic beforehand and use that heuristic to choose the operation at each step (As opposed to having the operation encoded in the chromosome like the pure approach). So basically the values of the genes are job indices, assuming the index value is 'a' that tells us to choose an operation using the fixed heuristic and then schedule that operation for the a^{th} uncompleted job at the earliest time possible. The scheduler works in the same way as in their pure GA approach that we discussed in section 2.2.1.

In their second implementation instead of using a fixed heuristic to choose the operation to schedule for a job, they used many different heuristics depending on each particular situation. In order to know which heuristic to use in each particular case, they encoded it in the chromosome in a separate gene that was next to the gene containing the job index. Just like the value for a gene corresponding to a job is an index to the list of available jobs, also the value that corresponds to a heuristic encoding is an index into the list of available heuristics. They called this approach the "evolving heuristic choice" as opposed to the previous approach that was denominated the "fixed heuristic choice".

Khoury and Miryala [10] presented a Hybrid GA implementation similar to what is proposed in Fang et. al [9]. Basically they encoded the solutions in chromosomes with values representing job indexes to handle. Just like Fang's approach the job index is an index to an unhandled job and we apply the modulus operator in order to obtain an actual unfinished job index. Also handling a job consisted mainly in scheduling one of the currently unhandled operations at the earliest available slot. To choose which of the unhandled operations of a given job to schedule first, they use a heuristic that gives priority to operations with larger processing times. The results obtained from this hybrid GA proved to be better than the two other approaches that were discussed by the authors in the same work. One of these approaches was using a pure GA that had the operations also encoded in the chromosome and one that used a variation on genetic algorithms called a selfish gene algorithm. All in all, their hybrid GA proved to be the most successful approach (they ran each GA 100 times for each

sample of Taillard's standard benchmarks [11]) and it even managed to obtain a solution better than the best known at that time for one of the 20 X 20 problem instances.

Comment On Khoury and Miryala's hybrid approach:

Khoury and Miryala's second approach suggests just encoding a hint on what is the next job we should try to schedule, this is as opposed to clearly encoding not just the job but also the precise task in that job, like he does in his Permutation GA. This of course in a way makes the possible range of values in a chromosome smaller as we are just encoding part of the problem, which is a schedule that only tells us the order in which to choose the jobs to schedule but not the tasks. So basically each chromosome doesn't really represent one exact schedule for the open shop problem but actually a group of different schedules depending on the order we choose for tackling the individual tasks that compose each job. This means that for each solution the GA obtains, we need to do a second exploration in order to find the actual optimal schedule, this second exploration is usually done with a different heuristic (annealing, tabu search, etc) and that's why this kind of encoding is usually used with hybrid approaches. Khoury and Miryala [10] use a simple algorithm which just decides on the order of tasks by handling the tasks with the larger completion times first, basically equating each solution of his GA to one actual schedule and skipping a more thorough exploration altogether. This shortens the search space but at the same time removes possible optimal solutions. That's why its no surprise that for some large

problem instances they seem to get the optimal answer while for some medium size instances they seems to obtain results that are far off the optimal. This is because in some problems we could be lucky to have an optimal or near optimal solution that has the tasks in each job ordered according to maximum completion time, but that is not necessarily always the case and it's not appropriate to assume it unless one can provide valid proof that the optimal will be in this format in most or all cases. From our testing experience, optimal solutions don't necessarily follow that pattern.

In our work we have based our encoding on the Permutation GA instead of trying to improve on the second approach presented in [10] (The "Hybrid" GA) despite the fact that the latter is reported as having much better results for the large benchmarks. The reason for not using that approach is because encoding the chromosome in that way would violate an essential condition for a good encoding which is that the encoding should potentially be able to represent all possible solutions in the solution space.

Chapter 3 Solution Implementation using Genetic Algorithms

3.1 Chromosome Encoding

When dealing with possible ways of encoding the solutions for the OSSP we must take into consideration that a good encoding must satisfy the following conditions:

a) The encoding should have the potential to represent all possible solutions in the solution space (including all optimal solutions).

b) The encoding should be appropriate with respect to the operators, meaning that the operators should be able to manipulate the chromosomes in order to explore successfully the solution space and be able to gradually approach the optimal.

A solution for the OSSP consists of a schedule or a given order in which the operations that conform each job are to be executed. Since we are trying a pure GA approach (as opposed to the hybrid GA approach), we have chosen to encode every chromosome as a set of integers each corresponding to an operation in the schedule. The possible gene values are in the range from 0 to $[(\boldsymbol{m} \times \boldsymbol{n}) - 1]$ where $\boldsymbol{m}$ is the number of processors and $\boldsymbol{n}$ the total number of jobs. The integer value of each gene can be decoded to obtain the operation number and the number of the job that owns that operation, the decoding is done in the following way: if the value of a given gene is $\mathbf{V}$ where $(0 \leq \mathbf{V} < \boldsymbol{m} \times \boldsymbol{n})$, then the processor that owns the operation is the quotient of the division of the value by the number of processors $(\mathbf{V} \text{ div } \boldsymbol{m})$ and the

operation number is the remainder of the same division (**V** mod ***m***). For example in the case of the 4 x 4 problem instance, a gene value of 13 would be interpreted as the second operation of the third job (the remainder is 1, but we consider 0 the first operation). As described above, a complete chromosome describes a schedule and taking as an example the 4 x 4 problem again, a chromosome would consist of 16 genes (since there are in total 4 x 4 operations) and the values of the chromosomes would be permutations of the values from 0 to 15. A possible chromosome sample would be the one in Figure 4:

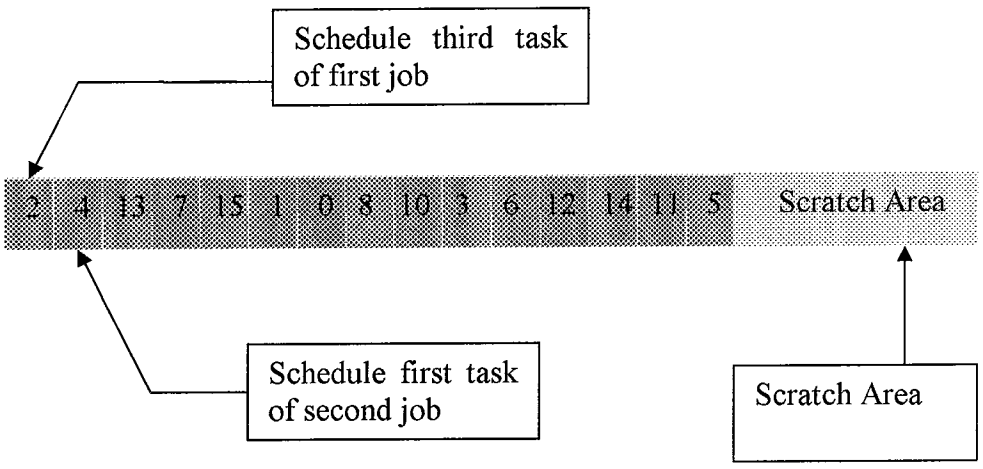

Figure 4: Chromosome encoding and interpretation

This would be interpreted as: first schedule the third task of the first job, then schedule the first task of the second job, then schedule the second task of the fourth job and so on. The actual scheduling procedure will be discussed later when we deal with the implementation of the objective function. So far, this way of encoding is the

same one used by Khoury and Miryala [10] in their first solution implementation (the permutation GA). We also would like to provide an explanation on why we are not using a hybrid approach at all. The reason why we have opted for a pure approach is that as this being our first research attempt on this area, we wanted to first explore what is the maximum possible performance and accuracy that can be obtained from the pure approaches in order to be properly aware of their limitations. We might work on improving our results by trying hybrid approaches in future research. One big advantage of using a pure approach is time, meaning that if a solution is found using genetic algorithms and a full encoding then we don't need to spend further time processing that solution; whereas in a hybrid approach all those solutions are just potential and we would need to spend extra processing time running a second heuristic in each of those solutions to obtain a real complete solution.

Coming back to our approach, besides the data area of the chromosome that we described previously, we will add an extra area that we will call the chromosome *scratch* area. We will describe in more details how the *scratch* area works in later sections, but for the time being we just want to explain that the size of the *scratch* area will depend on the amount of information on mutation decisions that we want to communicate. The *scratch* area basically serves as a storage for information that the objective function writes in order to help the operators make a more intelligent move, also occasionally the operators could also store information in the *scratch* area that could be of use to the objective function. This area is called *scratch* because it doesn't store actual scheduling information but rather some hints and other info

useful to help the operators make more informed choices, the information in the *scratch* area can sometimes also save computation time by providing some data that the receiving function would have to otherwise compute on its own. The *scratch* area consists mainly of as many pairs of genes as the amount of mutation moves we want to perform in each call to the mutation operator, and one gene that stores a position in the data area of the chromosome at which the schedule represented by the chromosome becomes invalid according to a certain criteria, for example exceeding an established time limit. Knowing at which point the schedule becomes invalid can be useful when applying crossover in order to make a more intelligent merge of chromosomes, as we will see later when discussing the implementation of the operators used.

3.2 Chromosome Initialization

The procedure to initialize a chromosome consists of first initializing the data area in order to obtain random solutions. We must make sure that the value of each gene is in the range [0 : (***n*** x ***m***) – 1] (where ***n*** = number of jobs and ***m*** = number of processors) and that all the ***m*** x ***n*** genes in the data area have all different values. In order to achieve this we use the following algorithm:

We generate a random number in the range [0 : (***n*** x ***m***) – 1] and we check if the value is already present in the data area of the chromosome. If the value isn't present then we set that value in the current gene of the chromosome, otherwise if the value is

already present we keep repeating this process until we obtain a new value that isn't present. We do this for each gene in the data area of the chromosome.

Concerning the *scratch* area, we initialize it by setting each gene in this area to 0. At this stage it is not important to have any meaningful value in the scratch area, as these values will be set by the operators and the objective function at a later phase. The value of 0 is used for initialization and indicates there is no information set.

The pseudo code in Table 4 illustrates the initialization procedure:

```
Void ChromosomeInitialization( Chromosome C )
{
     For All genes in C do
          Set gene = numberMachines * numberJobs;
     For all genes in the data area of C do
     {
          Range = [ 1, numberMachines * numberJobs ];
          R = GenerateRandomInteger( Range );
          While ( ValueIsInChromosome( R, C ) ) do
               R = GenerateRandomInteger( Range );
          Set gene to R;
     }

     For all genes in the scratch area of  C do
          Set gene to 0;

     Return; //all done
}
```

Table 4: Chromosome Initialization Procedure

3.3 Fitness Function

The objective is to minimize the makespan, in order to do this we make use of a fitness function to determine the value of a chromosome. Since each chromosome represents a schedule, the fitness function analyzes the chromosomes in order to

determine the makespan (finish time) of the schedule. So the fitness function basically receives a chromosome as input and returns (1 / makespan) as output.

In order to determine the makespan, we apply the following procedure to all genes in the data area of the chromosome: First we decode the gene value to obtain the job number and the machine number. Next from the job and machine number we obtain the operation length (time this operation requires for completion). Finally we schedule the operation.

After we have scheduled all operations we find the machine with the latest finish time and we use its finish time as the *makespan* for the chromosome schedule.

When scheduling operations we always try to schedule operations at the earliest time possible, we do that by checking the idle time sections between scheduled operations. We try to fit a new operation in the earliest idle time area that is big enough to fit the operation length, if there is no available time slot that is large enough then we schedule the operation after the last operation that was scheduled in that machine. Another constraint is that if the job that owns that operation is currently scheduled as active in another machine at a given time that intersects with the time interval in which the job is active in the current machine, then we can't schedule the operation even if we find a large enough gap. This is because of the basic constraint of the OSSP that specifies that a job can't be processed in two machines at the same time. Figure 5 shows some sample pseudo code that illustrates some of the workings of the objective function and the scheduler:

```
Real Objective( Chromosome C )
{
     var machineNumber = 0, jobNumber = 0, operationLength = 0;
     score = 0, tmp = 0;
     invalid = false, done = false;
     Info is an array of values

     For all machines do
     {
          Set total idle time to 0;
          InitializeScheduleTable(machine);
     }

     For all genes in the data area of chromosome C do
     {
          machineNumber = GetMachineNumber(gene);
          jobNumber = GetJobNumber(gene);
          operationLength = GetOperationLength( jobNumber,
          machineNumber );
          tmp = ScheduleOperation( jobNumber, machineNumber,
          operationLength );
          If ( !invalid and tmp > score )
               score = tmp;
          if ( !done and invalid )
          {
               Set last chromosome gene to the index of the current
               gene;
               done = true;
          }

     }

     if ( score < bestSoFar )
     {
          bestSoFar = score;
          aux  =  bestSoFar - lowerBound;
          if ( aux < idleTimeLimit + 1 )
          {
               aux = idleTimeLimit - aux;
               For all machines do
                    Update idle time allowed ( -(aux + 1) );
               idleTimeLimit  =  bestSoFar - lowerBound - 1;
          };
     };

     Range = ( 1, maxNutMoves )
     numberMoves = RandomInteger( Range );
     Info = GetMoveInfoFromSchedule( C, numberMoves );

     For as many moves as determined by numberMoves
          Set pair of genes in the scratch area = to pair of values
in Info;

     return score;

};
```

Figure 5: Objective function pseudo code

3.4 Intelligent Move Logic

In this section we will describe the analysis that the objective function does in order to determine a possible intelligent swap that could be applied to two genes in a chromosome in order to improve the schedule it represents. The objective function determines as many possible swaps as determined by the *maxNumberOfMutations* parameter. The results are pairs of values indicating positions in the data area of the chromosome to be swapped and are stored in the scratch area of the chromosome (except in the last gene). Those intelligent swaps are hints that are intended for the operators that process that chromosome (mutation and crossover). The idea is that instead of having for example a purely random mutator like the one used in [10], the mutator will be able to rather make an intelligent choice by following the hints communicated by the objective function through the scratch area, so basically chromosomes contain meta data that helps the different operators make a more informed choice (without having to go through the hassle and performance consuming action of rebuilding the schedule and analyzing it) while still keeping the operators lightweight and performance efficient.

It remains to define the criteria used to determine what kind of swap in the chromosome would be proper, in the sense that it would have stronger possibilities of leading us to an optimal solution after generations of evolution. It must be noted that making only intelligent moves could constrain the solutions too much and get us stuck in a local optima; therefore it's important to combine the intelligent approach with a random approach. The amount of randomness and intelligence must be

balanced through tuning, we will discuss this in more details later when we deal with testing and tuning.

The criteria for judging intelligent moves is based on the property that the optimal finish time can never be less than the sums of the operation lengths of all operations that have to run in a particular machine and also it can't be less than the total completion time of the operations that conform any single job. Basically, the maximum of these two sums ie the maximum job completion time and the maximum machine running time becomes a lower bound for the makespan of the schedule. This means that if we are striving to find a utopian solution that would basically be the lower bound (it could exist or not) we could base our analysis in that assumption and make GA operator decisions based on that; For example we would know how much idle time we are allowed for each machine by checking the difference of the total running time in that machine with respect to the lower bound. If a given schedule contains any machine that is exceeding it's allowed idle scheduling time then we can analyze the idle time gaps in the scheduling table for that particular machine and perform a move that allows us to remove or minimize one of those particular idle time gaps. One technique that could be done to perform this (which we used) is that after finding an idle time gap that we would like to remove (to hopefully improve our schedule), we find another operation that in a way constrained the scheduling of the current operation and move that one to another position. To do this we work on the fact that if there is idle time before this operation then it means that for sure this job is

being processed in another machine up till the starting time of its operation in the machine we are currently analyzing. This is because as we discussed before there are two reasons that constrain our scheduling of operations: 1) whether the idle time gap can fit the operation and 2) whether the job that owns the operation is not being executed in another machine at that same time. Since in this case obviously there was a big enough idle time gap to schedule the operation, the only reason why we scheduled it later is because this job was being executed on another machine, therefore we scheduled it exactly at the finish time of the execution. So in order to hopefully minimize the gap we actually swap the constraining operation to any other place in the schedule (another position in the chromosome), but this will be the job of the mutator, the Objective function merely writes the move hints and later the mutator will perform the actual moves (if it deems proper). The moves are written in the scratch area as pairs consisting of an operation to adjust and a corresponding constraining causal operation.

Obviously, it would be wrong to assume the optimal is equal to the lower bound as this could be not the case and we would end up in a futile quest for a never to be found "holy grail" solution. Therefore it would seem that the assumptions necessary in order to decide on intelligent moves are too risky, but from empirical research experience we can see that the optimal for most if not all problem instances tends to be quite close to the optimal, therefore one can establish a safety margin and add it to the lower bound to make a new bound according to which to judge and decide on the intelligent moves. In practice we always keep track of the allowed idle time for a

given machine and everytime a better solution is found we immediately update the allowed idle times on each machine to make the algorithm become more "picky" and strive for solutions that improve on the current found results.

Having said that, a question that remains is what criteria to use in order to choose the gap to adjust, in a given machine schedule that could have two or more gaps which basically contribute on breaking the idle time allowed limit. There are many possible approaches for this; we basically used two on our implementation and testing: The first approach simply traverses the gaps in order, then chooses the first problematic gap. The second approach traverses the gaps randomly and again chooses the first problematic gap found. We will discuss the different results from these two approaches in the section for tuning and testing, at the end of the chapter. In fact, the different techniques in selecting the gap to adjust are basically regulated by a tuning parameter that we have experimented with during our testing.

Another important piece of information we can gather is the gene position at which this schedule proves to be an "invalid" solution. The term "invalid" here doesn't refer to a chromosome that represents an unfeasible schedule like for example having two equal genes or having a gene which isn't in the range [0, (m x n) – 1] where m = number of processors and n = number of jobs; in general we are basically maintaining chromosome integrity through the processing done by all the operators involved, in the next section we take a look at how the different operators work and we will confirm this. So basically the term "invalid solution" refers in this context to a solution that has exceeded the current sought level of quality, ie the quality of finish

time. If the current best finish time is for instance **T** any schedule which has a finish time $\geq$ **T** is considered invalid as it's not improving on our current goals. While we are evaluating a schedule using the objective function, after scheduling each operation we can know the current finish time of the machine at which we are scheduling the operation, when we find an operation that causes a machine to exceed its allowed finish time, then we keep track of the position of the operation that "breaks" the schedule and store that position at the place reserved for it precisely at the last gene of the scratch area. This particular piece of information becomes useful when performing crossover as we will explain later when dealing with the crossover operator.

Figure 6 shows some sample pseudo code illustrating the intelligent move choosing and analyzing:

```
Void GetMoveInfoFromSchedule( integer info1, integer info2 )
{
      Var scheduleInfo si = NULL, scheduleInfo _si = NULL;
      count = 0, ret = 0, nextAvailableTime = 0;
      startTimeRange = 0, endTimeRange = 0;
      idleTime = 0, maxIdleTime = 0, index = 0, index2 = 0;
      Totals is a global array which contains the total completion time
      for each machine(the summation of the operation lengths of all
      its tasks)

      //determine which machine schedule to fix
      case 0 and case 3:
            for all machines
            {
                  //set si to entry for last job of the current
                  //machine's schedule table
                  si = scheduleTable( machine, numberJobs );
                  if ( si->finishTime - Totals[machine]  >
                  idleTimeAllowed[machine])
                  {
                        Set index = current machine number;
                        timeToImprove = si->finishTime;
                        break out of loop;
                  }
            };

      case 1 and case 4:

            tmp = 0;
            Loop undefinetely
            {
                  Range = [1, numberMachines]
                  tmp = RandomInteger( Range );
                  //set si to entry for last job of the random
                  //machine's schedule table
                  si = scheduleTable( tmp, numberJobs );

                  if ( si->finishTime - Totals[tmp]  >
                  idleTimeAllowed[tmp] )
                  {
                        index  =  tmp;
                        timeToImprove = si->finishTime;
                        break out of loop;
                  }
            };
```

Figure 6: Intelligent Move Logic

```
case 0 and case 1:
      for all jobs do
      {
            si = scheduleTable( index, job );
            if ( job = = 1  ) //it's the first job in the list
            {
                  idleTime = si->startTime - _si->finishTime;
            }
            else
            {
                  idleTime = si->startTime;
            };
            if ( idleTime > maxIdleTime )
            {
                  maxIdleTime = idleTime;
                  index2 = job;
            }
            _si = si;
      };

case 3 and case 4:
      int tmp = 0;
      loop undefinetely
      {
            Range = [0, numberJobs - 1]
            tmp = RandomInteger( Range );
            si = scheduleTable(index, tmp);
            if ( tmp > 1 ) //if not the first job on the list
            {
                  _si = scheduleTable(index, tmp - 1);
                  idleTime = si->startTime - _si->finishTime;
            }
            else
            {
                  idleTime = si->startTime;
            };
            if ( idleTime > 0 )
            {
                  index2 = tmp;
                  break out of loop;
            }
      };


 _si = scheduleTable(index, index2);
infol = index * numberJobs + _si->jobNumber;
machineIndex = index;
timeToImprove =  scheduleTable(index, numberJobs).finishTime;
PosInfo1 = GetIndexOfValueInChromosome( C, infol );
```

Figure 6: Intelligent Move Logic (cont.)

```
        //Now let's get info2
        for all machines
        {
              if ( i != index )
              {
                    for all jobs
                    {
                          si = scheduleTable(i,j);
                          if ( si->jobNumber = = _si->jobNumber and
                                si->finishTime = = _si->startTime )
                          {
                                index = i;
                                break out of loop;
                          }

                    };
              };
        };

        info2 = index * numberJobs + _si->jobNumber;
        PosInfo2 = GetIndexOfValueInChromosome( C, info2 );


};
```

Figure 6: Intelligent Move Logic (cont.)

3.5 Genetic Operators

3.5.1 Mutation Operator Implementation

The mutation operator performs mainly swaps between the genes of a given chromosome. These swaps can be both, intelligent or just random; we will discuss both approaches in this section. In order to determine whether to use the random or the intelligent approach we set a parameter indicating a probability, and then based on that probability we determine if we should use the random or the intelligent approach every time the mutator is called. It's very important to tune this parameter, and in fact extensive time has been taken during testing to find the optimal adjustment of the tuning variable, in order to reach the proper balance between randomness and

intelligence. The problem of following a purely intelligent approach is that it might constrain too much the solutions we get and might make us get stuck on local optima. The problem of following a purely random approach is that for large search spaces (such as is the case for the medium to large problem instances in popular benchmarks like Taillard) the genetic algorithm could take a huge amount of time in order to even fairly approach the global optimal. Therefore as mentioned above it's very important to find a balance between both approaches to obtain the best results.

The first approach is purely random and basically consists of choosing a random number in the range [0, $(m \times n) - 1$] and then search through the data area of the chromosome until we find a gene that has a value equal to that. Then we select another random number and repeat the same procedure. Finally we swap the two genes in order to alter the chromosome.

The second approach which is more intelligent uses the information stored in the scratch area by the objective function. As explained in the previous section, the objective section has intelligently chosen different pairs consisting of an operation that is scheduled in such a way that it affects the performance of the schedule and another operation that seems to cause the first one to be scheduled in that way. For each of these pairs the mutator chooses randomly one of the operations hinted and then moves it to another position in the chromosome (the new position is chosen randomly just like in approach 1). The number of pairs and therefore mutation moves

is controlled by parameter maxMutationMoves, in most of our testing we have usually set this parameter to 1 in order not to alter chromosomes too much in one mutation operation. Nonetheless we also tried using a more aggressive mutator but it didn't prove to deliver better results as we will see in the section dealing with testing and tuning.

Figure 7 shows some sample pseudocode that illustrates the workings of the mutator:

```
//Mutation operator for the Permutation GA
Mutator( Chromosome C, Real mutationProbability )
{
      IntegerArray auxArray;

      //store value of last gene in aux
      aux = C.getGene( C.length( ) - 1 );
      for all gene values in the scratch area
      {
            auxArray.Add( value );
      };

      if (  FlipCoin( mutationProbability )  )
      {
            Range = [1,2]
            option = RandomInteger( Range );

            check ( option )
            {
            case 1:
                  //intelligent (deterministic) mutator
                  for all pairs of values in the scratch area
                  {
                        Pair = auxArray.GetNextPair();
                        i = RandomInteger( [0, 1] );
                        if ( i = = 1 )
                        index1 = GetIndexinChromosome( C, Pair.1 );
                        else index1 = GetIndexinChromosome( C,
                        Pair.2 );

                        index2 = RandomInteger( 1, C.dataAreaSize );
                        tmp = C[index1];
                        Delete( C, index1 );
                        Insert(  C, tmp, index2 );
                  };
                  Break out;


            case 2:
                  //random mutator
                  for all jobs do
                  {
                        index1 = RandomInteger( 0, C.dataAreaSize
                        );
                        index2 = RandomInteger( 0, C.dataAreaSize
                        );
                        tmp = C[index1];
                        C[index1] = C[index2];
                        C[index2] = tmp;
                  };
                  Break out;
            }
      }

}
```

Figure 7: Mutation operator pseudo code

3.5.1 Crossover Operator Implementation

The Crossover operator consists of 2 different possible modules, one performs something similar to a uniform crossover but not precisely that as it can't be really applied as is to this case (it could be applied but we would need the objective function to be aware of invalid chromosomes which is not the approach we have chosen as described in previous chapters). The second approach uses the information stored in the last gene of the scratch area by the Objective function.

As usual we use a probability parameter that can be tuned in order to determine how often to use each different approach, it's also possible to adjust the parameter so that a particular approach is used exclusively.

The first approach as previously mentioned is similar to uniform crossover; We can't simply apply uniform crossover, which consists of taking the first gene from chromosome one, the second gene from chromosome two, then the third again from one and the fourth from two, and so on until all genes in the new chromosome 3 resulting from the merge of 1 and 2 is complete. The reason this approach is not feasible is because we would be breaking the rule that states we cannot have duplicate genes, as this method does not enforce the chromosome integrity for that matter.

So the method we use is a variation of this algorithm which consists of the following steps (Please note that these operations are only applied to the data area of the chromosome):

1) We extract the value of the current gene in chromosome one (the current gene is initially the first one), if the value isn't yet present in the new chromosome, then we insert the value in the next available empty gene of the new chromosome.
2) If the value is already present in the new chromosome then we repeat step 1 by selecting the next gene until a value that isn't present in the new chromosome is found.
3) We then do the same procedure for chromosome 2.
4) We keep repeating this procedure until the new chromosome has all the values of its genes set.

This approach guarantees that every pair of genes in the resulting chromosome consists of a gene that originates from chromosome 1 and one from chromosome 2 , Hence it's similarity to the uniform crossover approach while having the great advantage of keeping chromosome integrity.

The second more intelligent approach attempts to do crosspoint crossover but not at any random point but rather at a special position determined using the information in the last gene of the scratch area. This information basically tells us the position of the chromosome at which the schedule becomes inappropriate as it has lost its potential

as a quality solution. To illustrate why this information could become useful, take for example the case where we just choose the crossover point randomly and that point is less than or equal to the position where the schedule becomes inappropriate, in that case if we use that section as the start of the new chromosome then we would be generating a low quality schedule(with a makespan which is equal or bigger than the current), on the other hand if we use a crosspoint which occurs before the critical position and we cross it over with the section from the second chromosome, we could be generating a potential good candidate solution(with a better makespan than the current one). In this implementation we can also randomly decide from which of the two chromosomes to take the cross point and which one to use as the start of the new chromosome. After crossover we set the scratch area of the new chromosome to 0s in order to initialize it so that the objective function will fill it with proper information after evaluating it.

Figure 8 shows a sample pseudo code that illustrates the Crossover function:

```
//Crossover operator for the permutation GA
PGACrossover( Chromosome C1, Chromosome C2,
              Chromosome C3 )
{
      //C1 and C2 are the 2 chromosomes that procreate C3 (new
      //chromosome). All chromosomes have the same size

      Integer crossPoint = 0, next = 0;
      Integer aux = C1.getGene( C1.size );

      C3.Initialize( );
      if ( !avoidInvalidSchedules )
      {

            count = 0;
            while ( count < C3.size )
            {
                  if (  RandomInteger([0,1]) = = 0 )
                  {
                        for all genes in C1
                        {
                              value = C1.getGene( gene );
                              if ( !ValueInChromosome( value, C3 ))
                                    break out of loop;
                        }
                  }
                  else
                  {
                        for all genes in C2
                        {
                              value = C2.getGene( gene );
                              if ( !ValueInChromosome( value, C3
                              ) )
                              break out of loop;
                        }
                  }
                  C3->setGeneValue( count, value );
                  count = count + 1;
            }
      }
      else
      {
            if ( aux = = 0 )
                  crossPoint = C1.size / 2;
            else crossPoint = RandomInteger( [0, aux - 2] );

            for all genes in C3 before the crossPoint
                  C3->setGeneValue( gene, C1.getGeneValue( gene )
                  );

            For all genes in C3 after the crossPoint
            {
                  value = C2->getNextGene( );
                  While ( ValueInChromosome( value, C3 ) )
                        value = C2->getNextGene( );
                  C3->setGeneValue( gene, value );
            };

            For all genes in the scratch area of C3
                  C3->setGeneValue( gene, 0 );
      };
{
```

Figure 8: Crossover operator pseudocode

3.6 Tuning and Testing

In this section we will explain the tuning process and the results we obtained after testing using the optimal parameters. The first parameters we experimented with were the population size and the number of generations to run the GA. After considerable testing and experimental experience (by starting with a high population and high number of generations and then gradually lowering them down) and taking into consideration the results when combining with the other tuning parameters (which will be discussed after), It was determined that the optimal population size and number of generations to run the GA are 400 and 1000 respectively.

As explained in the preceding sections, we have used an approach the combines the use of randomness and intelligence when using the operators that genetically alter chromosomes or generate new offspring. One important parameter that was discussed when dealing with the mutation operator was the *maxNumberMutations* parameter which basically specifies the degree of alteration a chromosome will suffer on each call to the mutator. In the case of the intelligent approach it specifies how many intelligent swaps we store in the chromosome *scratch* area and perform during mutation phase. During the tuning process we realized that the optimal value for this parameter is one, as changing chromosomes too much in one mutation iteration seems to make the evolution process slower, apparently because we might be constraining the solutions.

One very important parameter to tune is the probability of mutation and crossover. Normally when using genetic algorithms, the crossover probability should be high whereas the mutation probability should be relatively low. During our testing we realized that a high crossover probability doesn't seem to work for the OSSP, the higher the crossover probability, the longer it took to get results and they were farther from the optimal. On the other hand increasing the mutation probability seemed to improve both the quality of the results and the number of iterations required to reach them. We think the reason for this results is that it's hard to come out with a feasible crossover operator that truly reflects the requirements of the OSSP, the current implementation doesn't really guarantee that an offspring chromosome would contain the best of its two originating progenitors, because there is no guarantee that the resulting schedule would be better, as its just a random merge of the previous schedules resulting in a schedule that has little to do with any of the originating ones. In the case of mutation the change to the schedule is lighter so it's easier to control the convergence of the chromosome to an optimal solution. After tuning we determined that an optimal crossover rate is between 0.2 and 0.3, and the optimal mutation rate is from 0.6 to 0.8. The mutation is what really helps the solution evolve intelligently whereas the crossover is just another mechanism to introduce some chaos to the solutions in case the intelligent mutations are constraining the solutions too much, so that we prevent getting stuck in local optima. That's the reason why we are using a low crossover rate.

Within the mutation itself, it's also important to regulate the amount of deterministic moves, because using a lot of intelligence can restrict the chromosome evolution too much and not let the solutions exceed a given limit (local optima), the crossover is supposed to help with that, but since the mutation probability is much higher and the crossover basically breaks the schedule completelely; we need another mechanism in order to introduce some randomness in the mutation by making just a small random variation to the schedule rather than making dramatic changes to the whole schedule as in the crossover. We have discussed this approach before in the mutation operator section and also the need for a probability parameter that determines the amount of randomness and intelligence used in the mutator. From our testing we have determined that the optimal value for this parameter is an intelligence probability of 0.6 and therefore a randomness probability of 0.4.

It's also important to note that we are using a roulette wheel selector as this kind of selector proved to obtain the best results among other selection criteria during the tuning phase.

We used the Taillard set of benchmarks as the basis to evaluate our results. These benchmarks are a set of job shop problem instances which were proposed by Taillard with the intention of providing a common base of comparison for the results from all different methods used to solve these problems. These problem instances are easy to generate and their size corresponds to that used in industrial problems. For the open shop, Taillard proposes problems were the number of processors and jobs is the same

(m x m), allows for 6 different sizes (4, 5, 7, 10, 15, 20) and provides 10 problem instances for each size.

Table 5 shows the results obtained after running each of the Taillard Benchmark problem instances using the optimal parameters. The GA was run 30 times for each particular problem instance and we are presenting the best found solution and also the average of all solutions found through the 30 runs.

Problem Instance	Optimal (Current Best)	Best Solution	Difference(%)	Average (30 runs)
4 x 4 – 1	193	193	0	195
4 x 4 – 2	236	236	0	241
4 x 4 – 3	271	272	0.4	273
4 x 4 – 4	250	250	0	255
4 x 4 – 5	295	295	0	298
4 x 4 – 6	189	189	0	193
4 x 4 – 7	201	201	0	207
4 x 4 – 8	217	217	0	221
4 x 4 – 9	261	263	0.8	269
4 x 4 – 10	217	217	0	221
5 x 5 – 1	300	303	1.0	309
5 x 5 – 2	262	265	1.1	271
5 x 5 – 3	323	335	3.7	343
5 x 5 – 4	310	321	3.5	331
5 x 5 – 5	326	338	3.7	344
5 x 5 – 6	312	318	1.9	327
5 x 5 – 7	303	309	2.0	312
5 x 5 – 8	300	305	1.7	307
5 x 5 – 9	353	361	2.3	369
5 x 5 – 10	326	336	3.1	343
7 x 7 – 1	435	454	4.4	464
7 x 7 – 2	443	461	4.1	485
7 x 7 – 3	468	470	0.4	496
7 x 7 – 4	463	472	1.9	485
7 x 7 – 5	416	419	0.7	435

Table 5: Taillard Benchmark Results

Problem Instance	Optimal (Current Best)	Best Solution	Difference(%)	Average (30 runs)
7 x 7 – 6	451	465	3.1	475
7 x 7 – 7	422	430	2.4	455
7 x 7 – 8	424	434	1.9	457
7 x 7 – 9	458	470	2.6	493
7 x 7 – 10	398	408	2.5	434
10 x 10 – 1	637	675	6.0	710
10 x 10 – 2	588	601	2.2	619
10 x 10 – 3	598	610	2.0	632
10 x 10 – 4	577	640	11.0	665
10 x 10 – 5	640	659	2.9	668
10 x 10 – 6	538	600	11.5	620
10 x 10 – 7	616	632	2.6	639
10 x 10 – 8	595	610	2.5	625
10 x 10 – 9	595	615	3.4	637
10 x 10 – 10	596	621	4.2	643
15 x 15 – 1	937	1127	20.3	1159
15 x 15 – 2	918	1135	23.6	1197
15 x 15 – 3	871	1025	17.7	1131
15 x 15 – 4	934	1088	16.5	1167
15 x 15 – 5	946	1107	17.0	1192
15 x 15 – 6	933	1073	15.0	1165
15 x 15 – 7	891	1058	18.7	1143
15 x 15 – 8	893	1063	19.0	1136
15 x 15 – 9	899	1013	12.7	1119
15 x 15 – 10	902	1064	17.9	1157
20 x 20 – 1	1155	1314	13.7	1407
20 x 20 – 2	1241	1360	11.8	1428
20 x 20 – 3	1257	1382	9.9	1415
20 x 20 – 4	1248	1378	10.4	1410
20 x 20 – 5	1256	1420	13.1	1452
20 x 20 – 6	1204	1393	15.7	1425
20 x 20 – 7	1294	1431	10.6	1493
20 x 20 – 8	1169	1326	18.2	1412
20 x 20 – 9	1289	1377	12.4	1443
20 x 20 – 10	1241	1383	11.4	1417

Table 6: Taillard Benchmark Results (continued)

Figure 9 and Figure 10 show plots representing the results we obtained for a small 4 x 4 problem instance and a large 15 x 15 problem instance

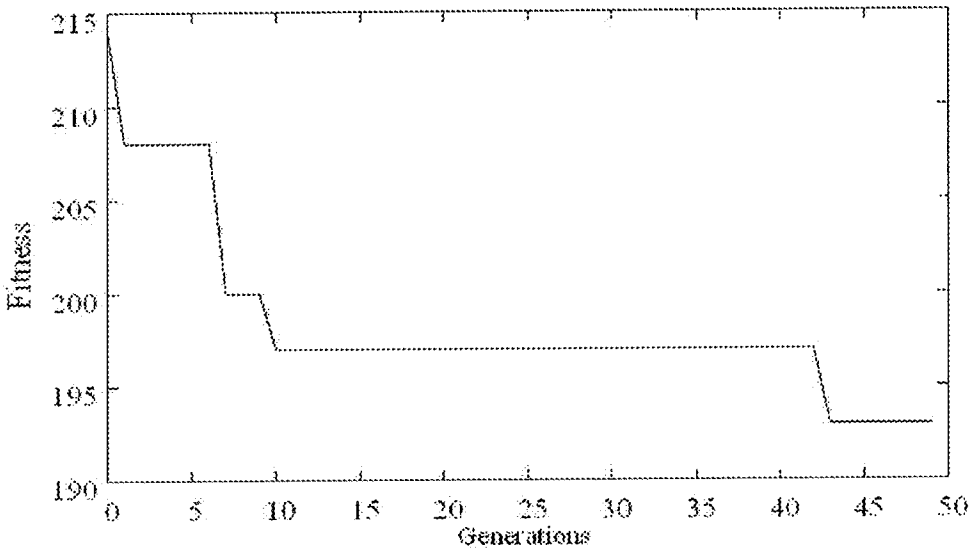

Figure 9: Plot for Taillard 4 x 4 – 0

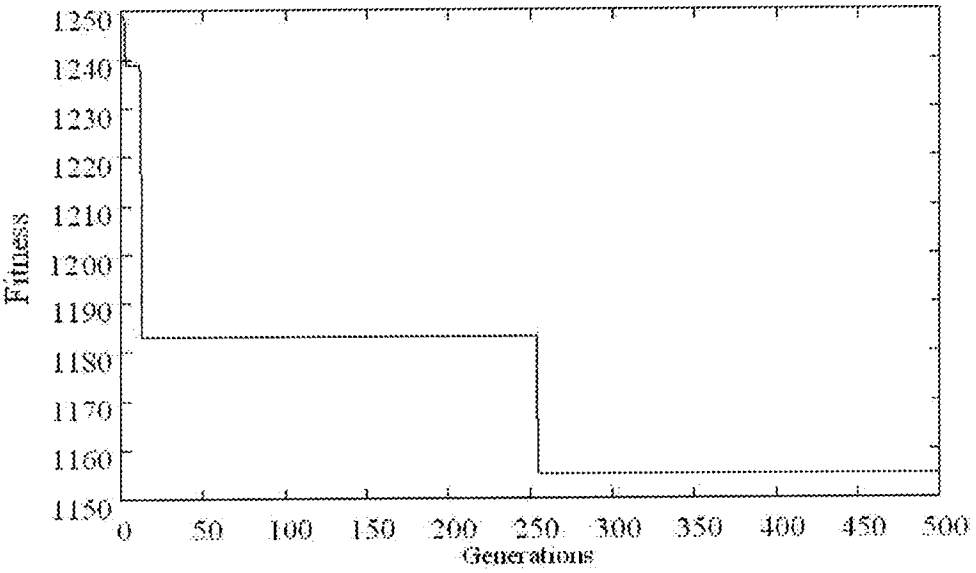

Figure 10: Plot for Taillard 15 x 15 – 0

In order to compare the results from this approach with some of the results obtained in the current research, we provide Table 7 which shows the results obtained using this method against those obtained by Khoury and Miryala [10]. The table shows 15 problem instances selected from Taillard Benchmarks, the first 5 samples correspond to small size problems (4 x 4), the next 5 samples correspond to medium size problems (7 x 7) and the final 5 samples are large size problems (15 x 15). For each problem we provide the lower bound, the current best and both the best solution and the average of all solutions for our method (intelligent permutation GA) and for both of the approaches presented in [10], the Permutation GA and the Hybrid GA. Please note that in [10] the results are obtained after running 100 times each problem, whereas ours were obtained after 30 runs for each problem.

Prob. Instance	Optimal (Current Best)	Intelligent Permutation GA Best / Average	Khoury's Permutation GA Best /Average	Khoury's Hybrid GA Best/Average
4 x 4 – 1	193	193/195	193/194	213/213
4 x 4 – 2	236	236/241	236/240	240/244
4 x 4 – 3	271	272/273	271/271	293/293
4 x 4 – 4	250	250/255	250/252	253/255
4 x 4 – 5	295	295/298	295/299	303/304
7 x 7 – 1	435	454/464	438/462	447/455
7 x 7 – 2	443	461/485	455/477	454/460
7 x 7 – 7	422	430/455	443/464	450/456
7 x 7 – 9	458	470/493	465/483	467/475
7 x 7 – 10	398	408/434	405/426	406/411
15 x 15 – 1	937	1127/1159	957/998	937/948
15 x 15 – 3	871	1025/1131	904/946	871/886
15 x 15 – 4	934	1088/1167	969/992	934/944
15 x 15 – 8	893	1063/1136	928/962	893/905
20 x 20 – 1	1155	1314/1407	1230/1269	1165/1190

Table 7: Comparison between the results obtained by the different methods

From table 3.2 we can see that the results from the Intelligent Permutation GA seem to be inferior to both the permutation GA and the Hybrid GA [10]. We have already discussed in a preceding section why the Hybrid GA is not in our opinion a reliable method to evaluate the OSSP. In the case of the permutation GA, this is supposed to be equivalent to the intelligent GA but with the probability for a random move set to 1 and hence that of an intelligent move set to 0. From our empirical experience increasing the intelligence probability, increases the quality of the results, but even then we don't get the results obtained in [10]. We are not sure what is the cause of the difference in results, perhaps the fact that in their work the GA was run 100 times per

problem instance as opposed to the 30 times we used. In any case we can see that none of the 2 methods give results that are close enough to the optimal for large problem instances. The reason why our GA doesn't seem to be a good method to use for the OSSP could be due to the use of an improper crossover operator. Our GA implementation seems to mainly rely on the mutation operator and this in a way undermines the purpose of the GA method itself. That said it is easy to come up with a simple crossover operator but it's very difficult to figure out a proper Crossover operator that would really make the GA work for the OSSP. Our conclusion in this chapter is that it is difficult to come up with a proper GA implementation that gives high quality results for the OSSP; Hence we will try to explore a different Heuristic that doesn't involve merging between existing solutions but rather relies on efficient and optimized alteration of a given solution in order to gradually reach an optimal. Possible heuristics that satisfy these conditions are Simulated Annealing and Tabu Search. We chose to use Simulated Annealing in our research and we will discuss our implementation, tuning and results in the next chapter.

Chapter 4 Solution Implementation using Simulated Annealing

4.1 Solution Encoding

We will encode our solution in the same format used for the chromosome encoding we presented in the previous chapter for the genetic algorithms implementation. The encoding as presented previously consists of a set of integers each corresponding to an operation in the schedule. The possible values for the elements that compose a solution are in the range from 0 to $[(m \times n) - 1]$ where m is the number of processors and n the total number of jobs. The integer value of each element can be decoded to obtain the operation number and the number of the job that owns that operation, the decoding is done in the same way described in the genetic algorithm encoding section. A possible solution for the 4 x 4 OSSP Problem is shown in Figure 11 below:

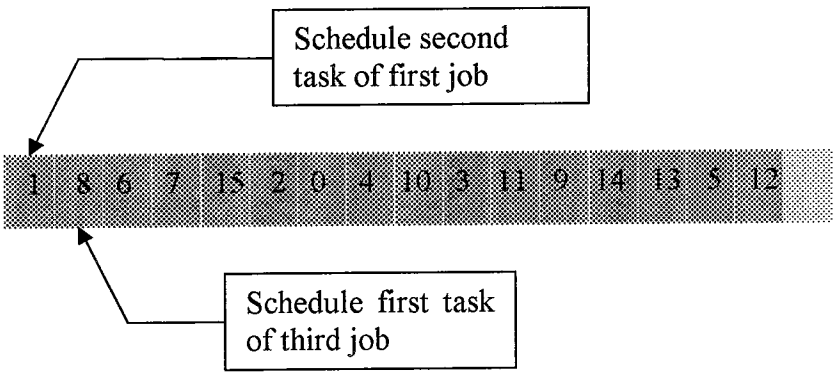

Figure 11: Solution encoding and interpretation.

This would be interpreted as: first schedule the second task of the first job, then schedule the first task of the third job, then schedule the third task of the second job and so on. This way of encoding is again the same used for the genetic algorithm encoding except that we don't include a scratch area anymore. The reason why a *scratch* area is no longer needed is because we experimentally determined that the use of deterministic moves becomes counterproductive when using annealing, this will be further discussed later when we deal with testing and tuning. So this encoding is basically the same one used by Khoury and Miryala [10] in their first solution implementation (the permutation GA).

We judge this encoding to be the most appropriate as it satisfies both essential conditions: All possible solutions in the solution space can be represented using the encoding and it is possible to come up with valid operators that can alter the chromosomes in a way that makes possible to explore successfully the solution space and be able to gradually approach the optimal.

4.2 Solution Initialization

The procedure to initialize a solution consists of obtaining random values for the elements that compose a given solution. We must make sure that the value of each element is in the range [0, $(n \times m) - 1$] (where n = number of jobs and m = number of processors) and that all the $m \times n$ elements in the solution have all different values. In order to achieve this we use the following algorithm:

We generate a random number in the range [0, (n x m) – 1] and we check if the value is already present in the solution. If the value isn't present then we set that value in the current element of the solution, otherwise if the value is already present we keep repeating this process until we obtain a new value that isn't present. We do this for each element in the solution.

4.3 Cost Function

Since each solution represents a schedule, the objective function analyzes the solutions in order to determine the makespan (finish time) of the schedule. So the objective function basically receives an encoded solution as input and returns the makespan as output.

In order to determine the makespan, we apply the following procedure to all elements in the solution: First we decode the element value to obtain the job number and the machine number. Next from the job and machine number we obtain the operation length (time this operation requires for completion). Finally we schedule the operation.

After we have scheduled all operations we find the machine with the latest finish time and we return its finish time as the makespan for the chromosome schedule.

When scheduling operations we always try to schedule operations at the earliest time possible, we do that by checking the idle time sections between scheduled operations. We try to fit a new operation in the earliest idle time area that is big enough to fit the operation length, if there is no available time slot that is large enough then we

schedule the operation after the last operation that was scheduled in that machine. Another constraint is that if the job that owns that operation is currently scheduled as active in another machine at a given time that intersects with the time interval in which the job is active in the current machine, then we can't schedule the operation even if we find a large enough gap. This is because of the basic constraint of the OSSP that specifies that a job can't be processed in two machines at the same time.

4.4 Neighbor Function

The neighbor function is used to find another solution in the same neighborhood of this one. This function is similar to the mutator we used for the genetic algorithms implementation with the exception that we only apply random moves as opposed to a proportion of random and intelligent moves as used for the GA implementation. The reason why we discard completely the deterministic moves is that empirically they have proved to decrease the quality of the solutions that are found, we will further discuss this in the tuning and testing section. It's also important to note that the support for deterministic moves is more crucial in GA because we use a mutator that would accept all random solutions without discrimination, but in annealing depending on the current temperature we could be accepting solutions that are non improving instead of only accepting improving solutions. As the temperature goes down the solutions we accept get more constrained until we only accept solutions that move us closer to the optimal. This support in annealing controls the randomness of the approach and provides some kind of intelligence that allows us to move gradually

towards the global optimal while avoiding getting stuck in local optima. The neighbor function performs mainly swaps between the elements that compose a given solution. Since a solution is valid to the extent that it represents a valid schedule, swapping a couple of elements will slightly alter the order of the execution of operations in the schedule resulting in a new valid schedule which is similar to its originating schedule. These swaps as described before are determined randomly. The number of swaps we perform in each call to the neighbor function in order to generate a neighbor is controlled by a tuning parameter, the number of these swaps should be small though, in order not to alter solutions too much in a single annealing iteration. Besides pure random swaps we also occasionally use other different strategies that when used with restraint can lead to better performance and quality of solutions. We will now describe these techniques, note that the use of these techniques is regulated using a tuning parameter and the probability for their use is set to a very low value as they can sometimes break completely the pattern of a given current solution. These approaches are mainly used as extra support to avoid stagnation in local optima.

4.4.1 Shift and Rotate Operator

One technique that can be used to alter a solution in a more dramatic way than using swaps while still keeping some of the patterns of the original solution and guaranteeing that the resulting solution would be valid is by shifting and rotating the solution.

I n order to do this we first determine a random number in the range [1, NS], where NS is the number of elements that compose a given solution. We then shift the solution to the right or to the left (this is determined by a toss of coin random function) and as we shift we rotate the shifted elements to the other side. For example in the case of the 3 x 3 problem instance, if the following solution:

2,4,7,0,9,1,5,8,6,3

Is shifted and rotated to the left 4 slots, we would obtain the following new valid solution:

9,1,5,8,6,3,2,4,7,0

And if we shift and rotate it 3 slots to the right then we would obtain the following valid solution:

8,6,3,2,4,7,0,9,1,5

4.4.2 Non Uniform Swap Operator

We have previously discussed the use of a tuning parameter that would determine how many random element swaps to perform in order to find a neighbor to the current solution. That is, in every call to the Neighbor function we perform as many swaps as determined by that parameter. This approach is what we would call a uniform swap operator. Another approach we have used in our testing is that of a non uniform swap operator, meaning an operator that doesn't always perform all the swaps determined by the tuning parameter but rather checks a particular condition in order to decide if it

should perform a particular swap or not. The actual condition we have experimented with in our research is that of setting a given probability and trying to randomly generate a number in a given range for all swap attempts, if the generated number is valid when confronted with the probability then we perform the swap, otherwise we don't. For example:

Assuming the number of swap moves is 5 and the probability of a swap is 0.7; and also assuming that the random numbers generated for each swap (in the range [0,1]) are: 0.2, 0.65, 0.75, 0.81, 0.37,

Then this would imply that only 2 out of the 5 swap moves wouldn't be performed, in particular the third and four swap moves are out of the probability range and therefore aren't performed. This technique can introduce a lot of variation since in every call to the neighbor function we could be performing a different number of swaps, which is good for exploration and avoiding local optima. At the same time this approach must be used with restraint by reducing it's frequency of use (giving it a low probability) because it can change a solution in a drastic way as opposed to other methods like the shift operator which are more prone to preserve the structure and essence of a solution. Also as we mentioned before the use of more than one swap move at each call of the neighbor function can also break solutions even if done in a uniform way, so we must be very careful when deciding on the frequency of use of each method.

4.5 Metropolis Annealing

The basic annealing heuristic algorithm consists on setting an initial temperature and gradually decreasing the temperature until we reach absolute zero. During this gradual cooling procedure we examine the neighborhood of the current solution state at each temperature level and we shift to a different solution state that could be an improvement or not. The lower the temperature the less willing the algorithm is to accept non improving moves. Theoretically, once the temperature reaches 0 only improving moves are accepted. We refer as temperature level each of the elements of the chain of temperatures that are obtained by multiplying the current temperature by the cooling factor, the starting temperature for the chain is determined of course by the initial temperature annealing parameter. In the basic annealing approach we only move to one neighborhood state at each temperature level.

In our research and testing we have used a variation of the annealing heuristic algorithm that has proved to deliver better results. We use the Metropolis annealing which differs from the basic annealing mainly in the fact that at each temperature level we do a more thorough exploration of the neighborhood. Instead of just trying to find one neighbor, we try to explore the neighborhood to find M neighbors, where M is a new tuning parameter which determines the depth of exploration and keeps varying during the annealing process. We can interpret M as the time until the next parameter update, basically a parameter update consist of lowering the temperature by multiplying it by the cooling factor and increasing the current Time parameter by M.

The Time parameter keeps track of the elapsed time and it's used to determine when we should end the annealing process, in order to determine when to stop we also set another parameter called Maxtime which is basically the total time that we plan to run the annealing process. Note that we don't just use the temperature reaching 0 as the stopping criteria as we use to do for basic annealing, this is because M keeps growing at each temperature level, so every level takes more time than the previous one and this can result in the process taking a huge amount of time before the temperature reaches 0, that's why setting a time limit is a more effective stopping criteria. In order to determine how much to increase M at each temperature level, we use an extra parameter called β which is a tuning constant.

Figure 12 shows pseudocode that illustrates the Metropolis annealing process:

```
AnnealingMetropolis( S0, T0, α, β, M, Maxtime )

      {S0 is the intial solution}
      {T0 is the starting temperature}
      {α is the cooling rate}
      {β is a constant}
      {Maxtime is the total time to run the annealing process}
      {M is the time until the next parameter update}
      {CurSol and BestSol are the current and best solutions
      respectively}

Begin
      T = T0;
      CurSol = S0;
      BestSol = CurSol;
      CurCost = Objective( CurSol );
      BestCost = Objective( BestSol );
      Time = 0;
      Repeat
            Metropolis( CurSol, CurCost, BestSol, BestCost, T, M );
            Time = Time + M;
            T = αT;
            M = βM;
      While ( Time >= Maxtime )
      Return ( BestSol )
End

Metropolis( CurSol, CurCost, BestSol, BestCost, T, M )

Begin
      Repeat
            NewSol = Neighbor( CurSol );
            NewCost = Cost( NewSol );
            aCost = (NewCost - CurCost );
            If (  aCost < 0 ) Then
                  CurSol = NewSol;
                  If ( NewCost  < BestCost ) Then
                        BestSol = NewSol;
                  Endif
            Else
                  If ( RAND < e^(-aCost/T) ) Then
                        CurSol = NewSol;
                  Endif
            Endif
            M = M - 1;
      While ( M <> 0 )
End
```

Figure 12: Annealing Metropolis algorithm pseudo code

4.6 Additional Optimizations

The use of the Metropolis approach can become excessively time consuming specially as the value of M grows. Since our objective is not just to obtain high quality solutions that are close to the optimal but also to obtain them in a reasonable

amount of time, the need to find supportive improvements or optimizations arises. There are basically two ways of improving the performance of the metropolis algorithm, the first way is by improving the performance of the OSSP scheduling algorithms we are using so that they are optimized to maximum and run in the least possible time. The second way is to "tweak" the metropolis process, meaning to alter the standard procedure in a way that we sacrifice some of its solution quality but gain a good performance improvement instead. In our work we have implemented both approaches and we will proceed to discuss them:

We used the second approach by adding a new tuning parameter that we called the beta interval and basically this parameter regulates how often we increment M by the beta factor. Instead of incrementing M at every temperature level, we would be skipping the M increment in as many temperature levels as determined by the current value of the beta interval parameter. This approach has proven in our testing to obtain similar quality results but with a dramatic time improvement.

Concerning the first approach, an example of this in our work can be seen in the scheduling algorithms we use; Particularly in a performance improvement we did to the algorithm that determines if a current job we plan to schedule in a given machine isn't active at that time in any of the other machines. We originally had a list for each machine that consisted of all jobs already scheduled in that machine and their times of schedule. Therefore every time we were trying to schedule a job in a given machine, we had to loop through all other machines and at each iteration traverse the list of one

of the machines and determine if the given job was being processed at that machine. This proved to be extremely time consuming when used in our Metropolis approach, so in order to improve the performance we added lists for each job which basically contain info of the remaining available time in each job. Therefore whenever we want to know if a given job is already active at a time, we just check one list, that of the job to be scheduled instead of checking (m-1) lists (m is the total number of machines).

4.7 Tuning and Testing

In this section we will explain the tuning process and the results we obtained after testing using the optimal parameters. One of the fundamental parameters for the annealing process is the starting temperature, in order to determine the optimal starting temperature we used the following method which is described in [12]. This approach is based on the fact that initial temperature should ideally allow all possible solutions whether improving or non improving to be accepted. This is because in an ideal annealing process as the temperature decreases, the algorithm becomes more reluctant to accept non improving solutions, and ideally when the temperature reaches 0 only improving moves are accepted.

Since the initial temperature T_0 must allow all possible moves to be accepted, that means that the acceptance proportion must be 1 or close to 1.

$P(T_0)$ = Number of moves accepted / Total Number of Moves Attempted

We determine the initial temperature T_0 by initially setting it to a small value and computing $P(T_0)$, if the value isn't close to 1 then we keep gradually incrementing the

temperature by multiplying it by a constant K (K > 1) and repeating this procedure until we reach a proportion which is 1 or very close to 1. The temperature we use to obtain that proportion becomes our initial temperature. This procedure models the process of heating the material until all its atoms are completely free. After performing this process we determined that the optimal initial temperature(T_0) should have a value of 400.

Another very important parameter we need to tune for our cooling schedule is the rate at which we decrease the temperature. The cooling rate is determined by the α parameter which should be smaller than 1 in order to decrease the temperature. Also since we want to decrease the temperature as slowly as possible it follows that we should select a value of α that is very close to 1. In our empirical testing we have determined that the optimal value range is $0.99 \leq \alpha \leq 0.999$, and for most samples setting α to 0.999 seems to give us the best results.

Since the approach we have followed for our annealing implementation is a metropolis approach, then the heuristic doesn't run until the temperature reaches 0 but rather until we reach the previously defined maxtime tuning parameter. This maxtime parameter depends on the problem sample, usually bigger samples require longer running time in order to achieve good results. The parameters that we should tune for the metropolis approach are the M and β parameters. In our testing we have found out that the optimal values for these parameters are 5 and 1.05 respectively.

We also discussed in section 4.6 an additional improvement for time performance and in order to achieve that we introduced a new betaInterval parameter. In our testing we

started with a high value for this parameter and kept decreasing it until we found a compromise of a time performance improvement and yet not a very high interval value. The optimal value for the betaInterval parameter proved to be 20 iterations.

Another important empirical observation was that increasing the probability of intelligence and deterministic approaches in the heuristic used to result in lower results quality. The better results were obtained by using an almost purely random approach.

The following are the results obtained after running each of the Taillard Benchmark problem instances using the optimal parameters. The Simulated Annealing process was run 30 times for each particular problem instance and we are presenting the best found solution and also the average of all solutions found through the 30 runs.

Problem Instance	Optimal (Current Best)	Best Solution	Difference (%)	Average (30 runs)
4 x 4 – 1	193	193	0	194
4 x 4 – 2	236	236	0	240
4 x 4 – 3	271	271	0	273
4 x 4 – 4	250	250	0	252
4 x 4 – 5	295	295	0	300
4 x 4 – 6	189	189	0	191
4 x 4 – 7	201	201	0	205
4 x 4 – 8	217	217	0	218
4 x 4 – 9	261	261	0	267
4 x 4 – 10	217	217	0	221
5 x 5 – 1	300	300	0	303
5 x 5 – 2	262	262	0	265
5 x 5 – 3	323	323	0	330
5 x 5 – 4	310	310	0	323
5 x 5 – 5	326	326	0	337
5 x 5 – 6	312	312	0	325
5 x 5 – 7	303	303	0	314
5 x 5 – 8	300	300	0	310
5 x 5 – 9	353	353	0	357
5 x 5 – 10	326	326	0	336
7 x 7 – 1	435	435	0	446
7 x 7 – 2	443	447	0.9	462
7 x 7 – 3	468	482	3.0	499
7 x 7 – 4	463	473	2.2	484
7 x 7 – 5	416	419	0.7	421
7 x 7 – 6	451	465	3.1	475
7 x 7 – 7	422	422	0	441
7 x 7 – 8	424	427	0.7	437
7 x 7 – 9	458	458	0	474
7 x 7 – 10	398	408	2.5	410

Table 8: Taillard Benchmark Results

Problem Instance	Optimal (Current Best)	Best Solution	Difference (%)	Average (30 runs)
10 x 10 – 1	637	645	1.3	662
10 x 10 – 2	588	589	0.2	614
10 x 10 – 3	598	611	2.2	647
10 x 10 – 4	577	577	0	599
10 x 10 – 5	640	645	0.8	653
10 x 10 – 6	538	549	2.04	574
10 x 10 – 7	616	632	2.6	643
10 x 10 – 8	595	610	2.5	621
10 x 10 – 9	595	615	3.4	621
10 x 10 – 10	596	602	1.0	623
15 x 15 – 1	937	942	0.5	952
15 x 15 – 2	918	968	5.4	985
15 x 15 – 3	871	878	0.8	904
15 x 15 – 4	934	963	3.1	971
15 x 15 – 5	946	999	5.6	1008
15 x 15 – 6	933	957	2.6	963
15 x 15 – 7	891	912	2.4	953
15 x 15 – 8	893	929	4.1	933
15 x 15 – 9	899	905	0.7	918
15 x 15 – 10	902	909	0.8	924
20 x 20 – 1	1155	1200	3.9	1211
20 x 20 – 2	1241	1296	4.4	1324
20 x 20 – 3	1257	1282	1.99	1293
20 x 20 – 4	1248	1274	2.1	1290
20 x 20 – 5	1256	1289	2.6	1305
20 x 20 – 6	1204	1243	3.2	1289
20 x 20 – 7	1294	1337	3.3	1362
20 x 20 – 8	1169	1215	3.9	1231
20 x 20 – 9	1289	1307	1.4	1384
202 x 20 – 10	1241	1293	4.19	1326

Table 8: Taillard Benchmark Results (cont.)

Figure 13 and Figure 14 show plots representing the results we obtained for a small 4 x 4 problem instance and a large 15 x 15 problem instance

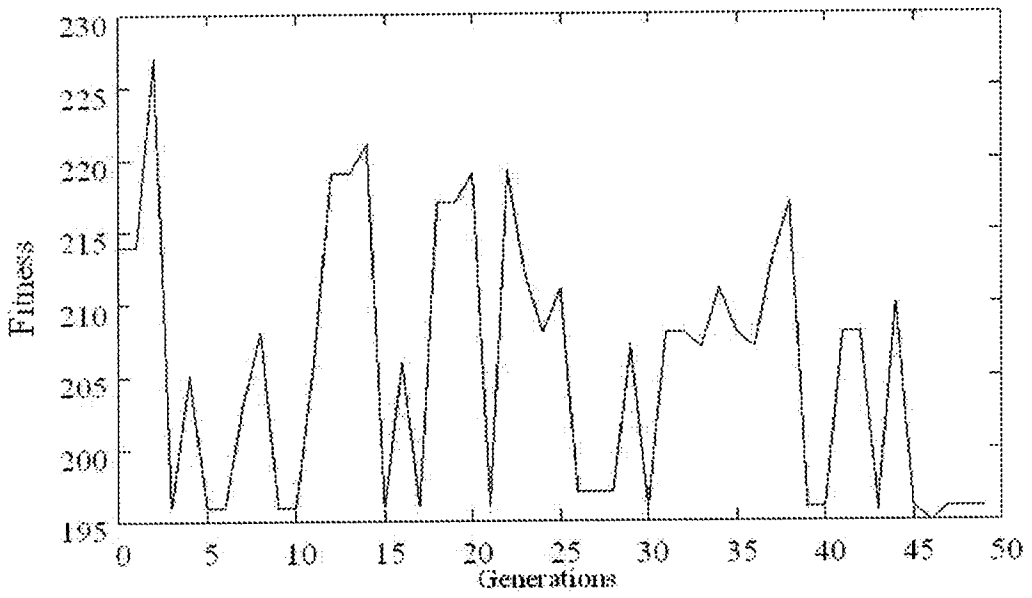

Figure 13: Plot for Taillard 4 x 4 – 0

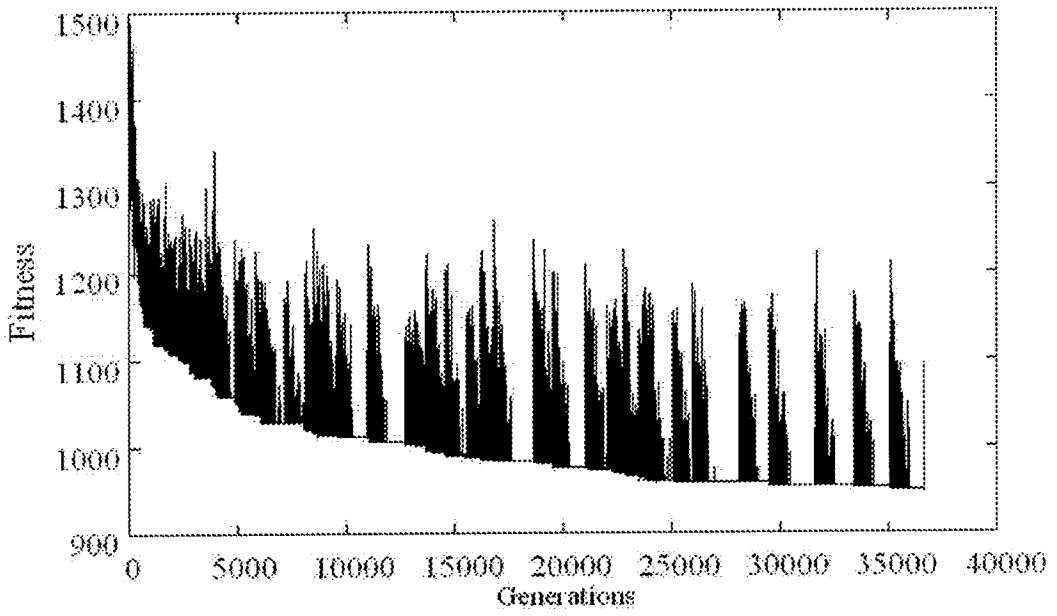

Figure 14: Plot for Taillard 15 x 15 – 0

In order to compare the results from this approach with some of the results obtained in the current research, we provide the following table which shows the results obtained using this method against those obtained by Khoury and Miryala [10] and against those obtained in a recent work by Liaw [13]. Liaw makes use of a hybrid genetic algorithm that relies on tabu search. The table shows all problem instances from Taillard Benchmarks, the first 20 samples correspond to small size problems (4 x 4) and (5 x 5), the next 20 samples correspond to medium size problems (7 x 7) and (10 x 10), and the final 20 samples are large size problems (15 x 15) and (20 x 20). For each problem we provide the lower bound, the current best and both the best solution and the average of all solutions for our method (Metropolis Simulated Annealing), for both approaches in [10] (the Permutation GA and the Hybrid GA) and the best solution obtained by Liaw [13]. Please note that the results in [10] are obtained after running 100 times each problem, whereas ours were obtained after 30 runs for each problem. We don't include a comparison with our own GA results from the previous chapter because we will compare and discuss those results in the following chapter, which deals with the conclusions from our work.

Problem Instance	Optimal (Current Best)	Metropolis Simulated Annealing Best/ Average	Khoury's Permutation GA Best /Average	Khoury's Hybrid GA Best/Average	Liaw's Best
4 x 4 – 1	193	193/194	193/194	213/213	193
4 x 4 – 2	236	236/240	236/240	240/244	236
4 x 4 – 3	271	271/273	271/271	293/293	271
4 x 4 – 4	250	250/252	250/252	253/255	250
4 x 4 – 5	295	295/300	295/299	303/304	295
4 x 4 – 6	189	189/191	189/192	209/219	189

Table 9: Results Comparison of the different methods

Problem Instance	Optimal (Current Best)	Metropolis Simulated Annealing Best/ Average	Khoury's Permutation GA Best /Average	Khoury's Hybrid GA Best/Average	Liaw's Best
4 x 4 – 7	201	201/205	201/202	203/203	201
4 x 4 – 8	217	217/218	217/219	224/224	217
4 x 4 – 9	261	261/267	261/264	281/281	261
4 x 4 – 10	217	217/221	217/219	230/230	217
5 x 5 – 1	300	300/303	301/312	323/324	300
5 x 5 – 2	262	262/265	262/271	269/279	262
5 x 5 – 3	323	323/330	331/345	353/355	323
5 x 5 – 4	310	310/323	N/A	N/A	310
5 x 5 – 5	326	326/337	N/A	N/A	326
5 x 5 – 6	312	312/325	312/328	327/339	312
5 x 5 – 7	303	303/314	N/A	N/A	303
5 x 5 – 8	300	300/310	N/A	N/A	300
5 x 5 – 9	353	353/357	353/367	373/376	353
5 x 5 – 10	326	326/336	326/340	341/343	326
7 x 7 – 1	435	435/446	438/462	447/455	435
7 x 7 – 2	443	447/462	455/477	454/460	443
7 x 7 – 3	468	482/499	N/A	N/A	468
7 x 7 – 4	463	473/484	N/A	N/A	463
7 x 7 – 5	416	419/421	N/A	N/A	416
7 x 7 – 6	451	465/475	N/A	N/A	451
7 x 7 – 7	422	422/441	443/464	450/456	422
7 x 7 – 8	424	427/437	N/A	N/A	424
7 x 7 – 9	458	458/474	465/483	467/475	458
7 x 7 – 10	398	408/410	405/426	406/411	398
10 x 10 – 1	637	645/662	667/705	655/672	637
10 x 10 – 2	588	589/614	N/A	N/A	588
10 x 10 – 3	598	611/647	N/A	N/A	598
10 x 10 – 4	577	577/599	586/618	581/589	577
10 x 10 – 5	640	645/653	N/A	N/A	640
10 x 10 – 6	538	549/574	555/583	541/549	538
10 x 10 – 7	616	632/643	N/A	N/A	616
10 x 10 – 8	595	610/621	N/A	N/A	595
10 x 10 – 9	595	615/621	627/646	598/618	595

Table 9: Results Comparison of the Different Methods (cont.)

Problem Instance	Optimal (Current Best)	Metropolis Simulated Annealing Best/ Average	Khoury's Permutation GA Best /Average	Khoury's Hybrid GA Best/Average	Liaw's Best
10 x 10 – 10	596	602/623	623/645	605/618	596
15 x 15 – 1	937	942/952	967/998	937/948	937
15 x 15 – 2	918	968/985	N/A	N/A	918
15 x 15 – 3	871	878/904	904/946	871/886	871
15 x 15 – 4	934	963/971	969/992	934/944	934
15 x 15 – 5	946	999/1008	N/A	N/A	946
15 x 15 – 6	933	957/963	N/A	N/A	933
15 x 15 – 7	891	912/953	N/A	N/A	891
15 x 15 – 8	893	929/933	928/962	893/905	893
15 x 15 – 9	899	905/918	N/A	N/A	899
15 x 15 – 10	902	909/924	N/A	N/A	902
20 x 20 – 1	1155	1200/1211	1230/1269	1165/1190	1155
20 x 20 – 2	1241	1296/1324	N/A	N/A	1241
20 x 20 – 3	1257	1282/1293	1292/1346	1257/1267	1257
20 x 20 – 4	1248	1274/1290	N/A	N/A	1248
20 x 20 – 5	1256	1289/1305	1315/1353	1256/1267	1256
20 x 20 – 6	1204	1243/1289	1266/1305	1207/1224	1204
20 x 20 – 7	1294	1337/1362	N/A	N/A	1294
20 x 20 – 8	1169	1215/1231	N/A	N/A	1169
20 x 20 – 9	1289	1307/1384	1339/1380	1289/1293	1289

Table 9: Results Comparison of the Different Methods (cont.)

From the table we can see that the results from the Metropolis Simulated Annealing process are a big improvement over the GA results we obtained in the previous chapter. Also we can verify that in general the results are slightly superior to the permutation GA in [10]. On the other hand the hybrid approach in [10] seems to deliver inferior results for the small problems but superior results for the large benchmarks, We already explained our take on this on our comments on this

approach in chapter 2, we just present the results here for completeness but we don't consider its valid to compare any of these methods with the hybrid approach in [10]. On the other hand Liaw [13] obtains the optimal in all cases and claims to take a maximum of approx. 2 minutes time for the largest benchmarks (20 x 20). His technique consists on generating random solutions and running tabu search on each solution separately in order to improve it to a local optimal, then all the local optima solutions become the population fed to a GA that determines the global optimal. In contrast our Metropolis Simulated Annealing heuristic runs around 2-3 minutes for a 4 x 4 problem, 3-4 minutes for a 5 x 5 problem, 8-10 minutes for a 7 x 7, 20-25 for a 10 x 10 and for the large problems it requires 40 – 45 mins for a 15 x 15 problem and around 60 mins for a 20 x 20 problem. These time frames are a huge improvement when compared to the almost 40 minutes that use to take to find an acceptable solution for the 7 x 7 problem using the permutation GA approach. But these times seem to pale in comparison to the times Liaw states are required to obtain the optimals; but in a further analysis we realize that the 2 minutes Liaw states to run the GA are not taking into consideration the time to move all members of the population to a local optimal. If we are very optimistic and assume a population of 100 solutions and we just give one minute of time in order for simulated annealing to upgrade the solution to a local optima, we are already taking at least 100 minutes for any problem instance. For these reasons we find the results we obtain from the Metropolis Simulated Annealing method are a good compromise between solution quality and time performance.

Chapter 5 Final Conclusions

This thesis proposed two different heuristic solutions to the OSSP based on GA and annealing. The proposed techniques yielded acceptably good solutions in almost all attempted benchmarks. Table 10 shows a comparison between the final results obtained using the GA method and the SA method. The results consist of the best solution found and the average solution after running the process 30 times using both methods:

Problem Instance	Optimal (Current Best)	Best Solution SA	Average Solution SA(30 runs)	Best Solution GA	Average Solution GA (30 runs)
4 x 4 – 1	193	193	194	193	195
4 x 4 – 2	236	236	240	236	241
4 x 4 – 3	271	271	273	272	273
4 x 4 – 4	250	250	252	250	255
4 x 4 – 5	295	295	300	295	298
4 x 4 – 6	189	189	191	189	193
4 x 4 – 7	201	201	205	201	207
4 x 4 – 8	217	217	218	217	221
4 x 4 – 9	261	261	267	263	269
4 x 4 – 10	217	217	221	217	221
5 x 5 – 1	300	300	303	303	309
5 x 5 – 2	262	262	265	265	271
5 x 5 – 3	323	323	330	335	343
5 x 5 – 4	310	310	323	321	331
5 x 5 – 5	326	326	337	338	344
5 x 5 – 6	312	312	325	318	327
5 x 5 – 7	303	303	314	309	312
5 x 5 – 8	300	300	310	305	307

Table 10: Results Comparison between GA and SA

Problem Instance	Optimal (Current Best)	Best Solution SA	Average Solution SA(30 runs)	Best Solution GA	Average Solution GA (30 runs)
5 x 5 – 9	353	353	357	361	369
5 x 5 – 10	326	326	336	336	343
7 x 7 – 1	435	435	446	454	464
7 x 7 – 2	443	447	462	461	485
7 x 7 – 3	468	482	499	470	496
7 x 7 – 4	463	473	484	472	485
7 x 7 – 5	416	419	421	419	435
7 x 7 – 6	451	465	475	465	475
7 x 7 – 7	422	422	441	430	455
7 x 7 – 8	424	427	437	434	457
7 x 7 – 9	458	458	474	470	493
7 x 7 – 10	398	408	410	408	434
10 x 10 - 1	637	645	662	675	710
10 x 10 – 2	588	589	614	601	619
10 x 10 – 3	598	611	647	610	632
10 x 10 – 4	577	577	599	640	665
10 x 10 – 5	640	645	653	659	668
10 x 10 – 6	538	549	574	600	620
10 x 10 – 7	616	632	643	632	639
10 x 10 – 8	595	610	621	610	625
10 x 10 – 9	595	615	621	615	637
10 x 10 - 10	596	602	623	621	643
15 x 15 – 1	937	942	952	1127	1159
15 x 15 – 2	918	968	985	1135	1197
15 x 15 – 3	871	878	904	1025	1131
15 x 15 – 4	934	963	971	1088	1167
15 x 15 – 5	946	999	1008	1107	1192
15 x 15 – 6	933	957	963	1073	1165
15 x 15 – 7	891	912	953	1058	1143
15 x 15 – 8	893	929	933	1063	1136
15 x 15 – 9	899	905	918	1013	1119
15 x 15 - 10	902	909	924	1064	1157

Table 10: Results Comparison between GA and SA (cont.)

Problem Instance	Optimal (Current Best)	Best Solution SA	Average Solution SA(30 runs)	Best Solution GA	Average Solution GA (30 runs)
20 x 20 – 1	1155	1200	1211	1314	1407
20 x 20 – 2	1241	1296	1324	1360	1428
20 x 20 – 3	1257	1282	1293	1382	1415
20 x 20 – 4	1248	1274	1290	1378	1410
20 x 20 – 5	1256	1289	1305	1420	1452
20 x 20 – 6	1204	1243	1289	1393	1425
20 x 20 – 7	1294	1337	1362	1431	1493
20 x 20 – 8	1169	1215	1231	1326	1412
20 x 20 - 9	1289	1307	1384	1377	1443

Table 10: Results Comparison between GA and SA (cont.)

From the results in this table we observe that the annealing algorithm outperformed the GA, especially for medium to large size problems. We have also observed that the larger the problem instance is, the greater the difference by which the result quality in SA surpasses that of GA. While both methods were ran for approximately the same time, annealing always obtained results that were reasonably closer to the optimal than those obtained by GA.

We attribute this to the crossover operators we have used. For GA to work properly it is essential to have a crossover operator that is appropriate to the problem one is trying to solve, meaning that when crossover is applied we should be obtaining offspring solutions that are a combination of the qualities of their progenitor solutions. In the case of the OSSP and our chromosome representation and crossover

implementation, despite the fact that we try to combine both chromosome schedules and keep the best of each, most of the time the resulting schedule is totally different than the originating ones. In an effort to improve these results we embedded more intelligence into the different operators, however this ended up restricting the solution.

In our testing we realized during the tuning of the crossover and mutation parameters, that whenever we increased the crossover rate the results quality tended to decrease. This decrease in solution quality worsened depending on how large the problem size was. On the other hand the changes to the mutation parameter were directly proportional with the quality of the results, the best results were obtained when the mutation probability was 1.

To illustrate this let's take for example the large benchmark 150: when running GA for 1 run (population 100 and 500 generations) with the crossover probability set to 0.6 and the mutation set to 1, the following results are obtained (see Figure 15). In this case the optimal found is 1186.

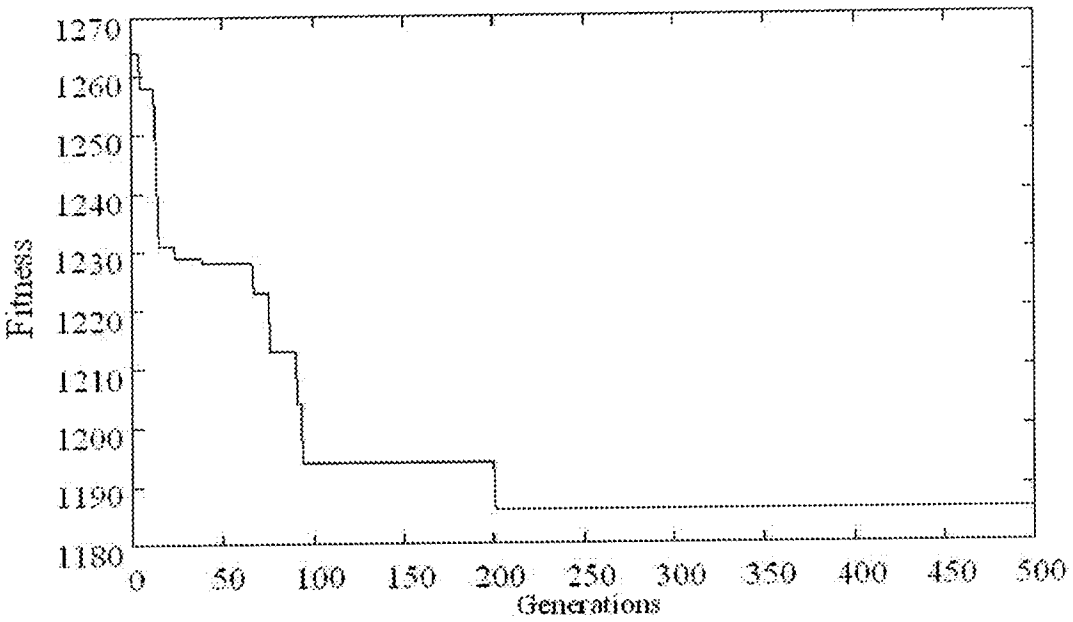

Figure 15: Case Study (150, 500, 100, 1, 0.6) Generation Optimal Fitness Plot

As shown above, the plot Figure 15 displays the changes in the fitness of the best chromosome for each generation, of running Taillard 150 for 500 generation of a population of 100 chromosomes, with a mutation probability of 1, and a crossover probability of 0.6

If we reduce the crossover to 0.3 while keeping the mutation constant, we obtain the following results shown in Figure 5.2. In this case the optimal found is 1134.

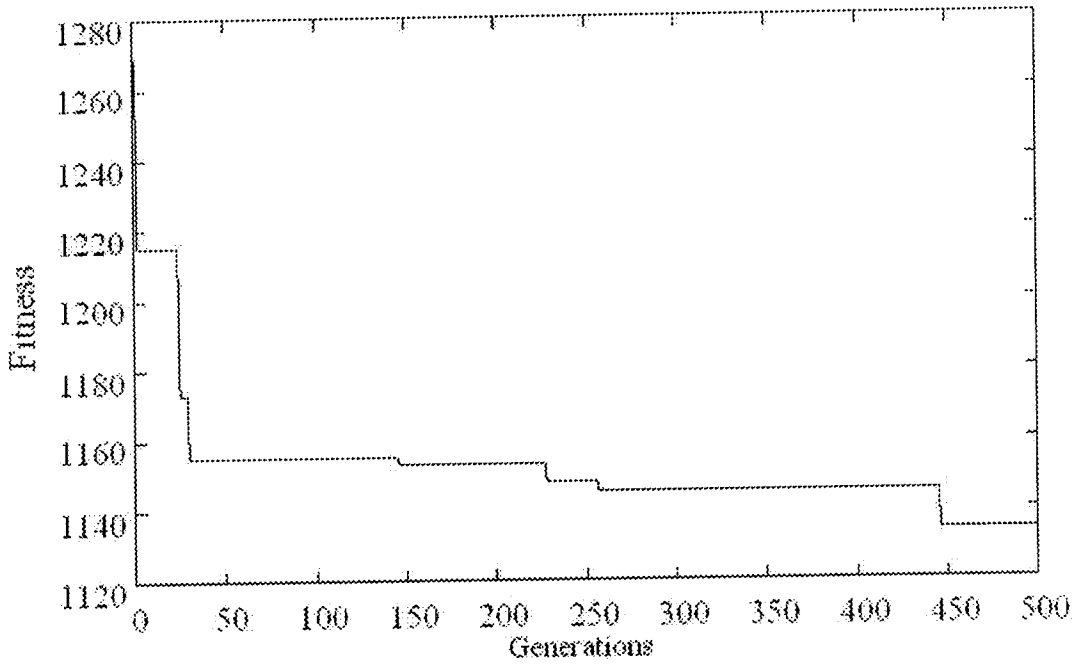

Figure 16: Case Study (150, 500, 100, 1, 0.3) Generation Optimal Fitness Plot

As shown above, the plot Figure 16 displays the changes in the fitness of the best chromosome for each generation, of running Taillard 150 for 500 generation of a population of 100 chromosomes, with a mutation probability of 1, and a crossover probability of 0.3

We use these particular results to illustrate a general trend that we discovered during the tuning and testing phase. From these results we observe that the mutation has a more important effect in the positive results than the crossover and the latter if used in high frequency can even become detrimental to the solution. So basically our research showed us that in order to obtain the best results using our operators, we had to set the mutation to a very high probability and the crossover to a low probability, this

showed that the mutation with its gradual and local changes was the main player for obtaining the results whereas the crossover acted just as a way to escape from local optima in some cases, by breaking the solution completely. This notion kind of contradicted the idea of using GA, as usually for a problem to be properly expressed the crossover rate should be high and the mutation low. This observation that the mutation operator was more effective is what led us to experiment with annealing which is a technique mainly based on a mutating the solution.

We though that for the way we implemented our operators and representation, a more powerful mutating technique such as the one used in annealing which basically combines a mutator with a powerful hill climbing strategy, would give us better results, and this proved to be the case. It's important to clarify that we aren't claiming that annealing is in general a better technique to solve the OSSP, we are mainly claiming that is hard to come with a proper crossover operator implementation for this problem. And that using the Metropolis SA gave better results than using GA with our crossover operator implementation.

We also experimented adding extra operators to our ga, like a shift rotate operator and an non uniform crossover. The non uniform crossover we implemented consisted in applying a 2 point crossover but instead of swapping the whole region between the 2 points, we compared individually the bits and we swapped them according to a fixed probability. Besides this we also experimented changing the criteria of machine selection when trying to decrease idle time gaps, we tried choosing the machine with the highest finish time as opposed to the first found and random choice criterias we

previously discussed. All of these new features didn't seem to bring any serious improvement with respect to the previous results obtained. Therefore, this further confirms our theory that a pure ga approach might not be as suitable as other approaches for the open shop scheduling problem.

Bibliography

1. Gonzalez, T. & Sahni, S. (1976). Open shop scheduling to minimize finish time. *Journal of the Association for Computing Machinery*, *23*(4), 665 – 679.
2. Carlier, J. & Pinson, E. (1989). An algorithm for solving the job shop problem. *Management Science*, *35*(2),164-176.
3. Brucker, P., Huring, J., & Wostmann, B. (1997). A branch and bound algorithm for the open-shop problem. *Discrete Applied Mathematics*, 76, 43-59
4. Gueret, C. & Prins, C. (1998). Classical and new heuristics for the open-shop problem: A computational evaluation. *European Journal of Operational Research*, *107*, 306-314
5. Davis, L. (1985). Job shop scheduling with genetic algorithms. In J.J. Grefenstette (Ed.), *Proceedings of the First International Conference on Genetic Algorithms and their Applications* (pp.136-140). San Mateo : Morgan Kaufmann.
6. Nakano, R. (1991). Conventional genetic algorithms for job-shop problems. In R. K. Belew & L. B. Booker (Eds.), *Proceedings of the Fourth International Conference in Genetic Algorithm* (pp.474-479). San Mateo : Morgan Kaufmann.

7. Fang, H.L., Ross, P. ,& Corne, D. (1993). A promising genetic algorithm approach to job shop scheduling, rescheduling and open-shop scheduling problems. In S. Forrest (Ed.), *Proceedings of the Fifth International Conference in Genetic Algorithms* (pp.375-382). San Mateo : Morgan Kaufmann.

8. Grefenstette, J.J., Gopal, R., Rosmaita, B. ,& Van Gucht, D. (1985). Genetic algorithms for the travelling salesman problem. In S.. Forrest (Ed.), *Proceedings of the First International Conference in Genetic Algorithms and their Application* (pp.60-168). San Mateo : Morgan Kaufmann.

9. Fang, H.L., Ross, P. ,& Corne, D. (1994). A promising hybrid ga/heuristic approach for open-shop scheduling problems. *Proceedings of the 11 th European Conference on Artificial Intelligence* (pp.590-594). [n.p.] : Wiley.

10. Khuri, S. & Miryala, S.R. Genetic algorithms for solving open shop scheduling problems. California, USA : San Jose State University.

11. Taillard, E. Benchmarks for basic scheduling problems. *European Journal of Operations Research, 64*, 278-285

12. Sadiq, M.S. & Habib, Y. [n.d]. *Iterative computer algorithms with applications in ingeneering: Solving combinatorial optimization problems.* [n.p.] : IEEE Computer Society.

13. Liaw, C.F. A hybrid genetic algorithm for the open shop scheduling problem. *European Journal of Operational Research, 124*, 28-42