Enhancing the Performance of Search Heuristics

Variable Fitness Functions and other Methods to Enhance Heuristics for Dynamic Workforce Scheduling

Stephen Mark Remde

submitted for the degree
of Doctor of Philosophy

Department of Computing
University of Bradford

2009

Abstract

Scheduling large real world problems is a complex process and finding high quality solutions is not a trivial task. In cooperation with Trimble MRM Ltd., who provide scheduling solutions for many large companies, a problem is identified and modelled. It is a general model which encapsulates several important scheduling, routing and resource allocation problems in literature. Many of the state-of-the-art heuristics for solve scheduling problems and indeed other problems require specialised heuristics tailored for the problem they are to solve. While these provide good solutions a lot of expert time is needed to study the problem, and implement solutions.

This research investigates methods to enhance existing search based methods. We study hyperheuristic techniques as a general search based heuristic. Hyperheuristics raise the generality of the solution method by using a set of tools (low level heuristics) to work on the solution. These tools are problem specific and usually make small changes to the problem. It is the task of the hyperheuristic to determine which tool to use and when. Low level heuristics using exact/heuristic hybrid method are used in this thesis along with a new Tabu based hyperheuristic which decreases the amount of CPU time required to produce good quality solutions. We also develop and investigate the *Variable Fitness Function* approach, which provides a new way of enhancing most search-based heuristics in terms of solution quality. If a fitness function is pushing hard in a certain direction, a heuristic may ultimately fail because it cannot escape local minima. The Variable Fitness Function allows the fitness function to change over the search and use objective measures not used in the fitness calculation. The Variable Fitness Function and its ability to generalise are extensively tested in this thesis.

The two aims of the thesis are achieved and the methods are analysed in depth. General conclusions and areas of future work are also identified.

Acknowledgements

I would like to thank everyone at Trimble MRM Ltd. for their time and insightful input on this project especially Evgeny Selensky, Bob Laithwaite, Paul Sexton and Jon Spragg.

Special thanks to my supervisors Peter Cowling and Keshav Dahal for their invaluable input and support throughout the duration of this project.

I also would like to thank Nic Colledge and everyone in the MOSAIC Research group for their support, distractions and friendship.

Finally I thank my family and especially my partner, Georgi, for their support and encouragement throughout the project.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Heuristic optimisation is used when an optimal solution is not required or exact methods are computationally intractable. Heuristics find good solutions that are not guaranteed to be optimal, however they are much quicker and usually find “good enough” solutions in a reasonable amount of time. Here the cost of solving the problem optimally in terms of CPU time is traded for the cost of a less than optimal solution.

Heuristics are also used when an optimal solution is not important because the model itself is not accurate or is based on assumptions. In this case, an optimal solution to the model of the problem may not represent an optimal solution in the real world and so solving the modelled problem exactly may not mean the real world problem is solved exactly making heuristic solutions even more attractive. This is usually always the case when modelling real world problems.

Over the years, as the problems we study get more complex and less easy to understand, heuristic solutions have become more and more attractive. The main focus of this thesis is a real world workforce scheduling problem which shares a lot of similarities with different scheduling problems in the literature and brings many features of the problems together into a single problem. Due to its complex and “messy” nature,

heuristic optimisation is ideal. A model for the problem was devised with the help of Trimble MRM and reflects the most important aspects of scheduling problems they tackle daily. It involves assigning multiple resources to geographically dispersed tasks. Tasks may require multiple resources to be completed and the resources must have the correct skill to do the task. Further making this problem messy is the fact that the time for a task is not known until resources are assigned, as the duration of the task is a function of its requirements and the competency of the resources assigned to it.

Many factors cannot be modelled accurately in this real world problem such as travel, the time it will take to complete a task, and the availability of resources and real world disturbances such as traffic for example. As such, exact methods are not always required. Furthermore, measuring the costs and the benefits of scheduling is an imprecise task as it involves human factors like customer satisfaction. These can only be estimated adding further inaccuracies to the calculation of a solution's quality. In this thesis we build new heuristic solutions to this problem using local search and hyperheuristics. We develop a hyperheuristic framework and a methodology for generating many low level heuristics combining exact and heuristic methods.

Complex heuristic solutions usually require a lot of expert time to implement and modify. If the problem of a heuristic solution to a complex problem changes slightly, then the heuristic may be inefficient and require changes. We aim to develop methods which can modify existing heuristics and potentially save a lot of this expert time whilst increasing performance in terms of CPU time and solution quality.

1.1. Aims

The overall aim of this research is to investigate new techniques for enhancing existing search based optimisation techniques. We have identified two key ways in which a search based optimisation technique can be enhanced:

i) decreasing the CPU time required to run

ii) improve final solution quality

The first aim is achieved with a new Tabu search method called Binary Exponential Back-Off which is addressed in Chapter 4. Decreasing the amount of CPU time required to produce a good solution to a problem means that problems can be solved quicker or larger problems can be solved without the need for better hardware. We aim to reduce the CPU time required without a large loss on solution quality.

The second aim is achieved with the Variable Fitness Function described in Chapter 5 and tested in Chapter 6 and 7. This aim requires the enhancement of existing solution methods to provide a better solution, in a similar amount of CPU time.

1.2. Contributions

- Recent works related to heuristic solutions of combinatorial problems especially relating to scheduling have been reviewed (Chapter 2).
- A real world scheduling problem has been identified and modelled including the dynamic side where disruption to the schedules are modelled (Chapter 3).
- Local search heuristics have been developed to greater understand the dynamic workforce scheduling problem in this thesis (Chapter 4).
- Methods of splitting problems into smaller problems to be solved exactly have been identified and heuristic methods for choosing which to subproblem solve

have been used resulting in Exact/Heuristic Hybrids (Section 4.1-4.4, Published in (Remde et al., 2007)) .

- New method for dynamically adjusting individual low level heuristics Tabu tenures in a hyperheuristic framework using Binary Exponential Back Off ideas are shown to be very effective (Section 4.5, Published in (Remde et al., 2009), Submitted to (Remde et al., Submitted)).
- The Variable Fitness Function methodology is detailed and motivated as a new method for guiding search based heuristics (Chapter 5, Published in (Remde et al., 2008) (Dahal et al., 2008) (Cowling et al., Submitted)).
- The Variable Fitness Function is used to solve various problems with different characteristics (Chapter 6 and 7, Published in (Remde et al., 2008) (Dahal et al., 2008) (Cowling et al., Submitted)).
- Methods are developed to try and anticipate bottlenecks when using constructive scheduling in order to avoid them (Section 6.3, Submitted to (Cowling et al., Submitted)).

Authored Academic Papers

The following papers were derived directly from the work in this thesis. The experimentation, analysis and write up were done by me, with brainstorming and corrections were done with the co-authors. Where applicable I presented the work at the conference.

S. M. Remde, P. I. Cowling, K. P. Dahal, N. J. Colledge, "**Exact/Heuristic Hybrids Using rVNS and Hyperheuristics for Workforce Scheduling**" in Proceedings of EvoCOP 2007, Springer LNCS 4446, 2007, pp. 188-197. (One of three papers of 81 nominated for the best in the conference)

S. M. Remde, P. I. Cowling, K. P. Dahal, N. J. Colledge, "**Evolution of Fitness Function to Improve Heuristic Performance**" in proceedings of Learning and Intelligent Optimization (LION) II, Springer LNCS 5313, 2008, pp 206-219.

K. P. Dahal, S. M. Remde, P. I. Cowling, N. J. Colledge, "**Improving Metaheuristic Performance by Evolving a Variable Fitness Function**" in Proceedings of EvoCOP 2008, Springer LNCS 4972, 2008, pp. 170-181.

S. M. Remde, P. I. Cowling, K. P. Dahal, N. J. Colledge. "**Binary Exponential Back Off for Tabu Tenure in Hyperheuristics**" in Proceedings of EvoCOP 2009, Springer LNCS 5482, 2009.

P. I. Cowling, S. M. Remde, K. P. Dahal, N. J. Colledge, "**Evolution of Fitness Functions to Improve Optimisation**" Submitted to Journal of Heuristics.

S. M. Remde, P. I. Cowling, K. P. Dahal, N. J. Colledge. **Binary Exponential Back Off Experimental Investigation and Comparison** Submitted to the Special issue of the Journal of the Operational Research Society on Heuristic Optimisation.

Co-Authored Academic Papers

In the following papers I was involved in creative input, brainstorming, experimentation, software development and/or proof reading. This work is not included in this thesis.

P. I. Cowling, N. J. Colledge, K. P. Dahal, and S. M. Remde, "**The Trade Off between Diversity and Quality for Multi-objective Workforce Scheduling**" in Proceedings of EvoCOP 2006, Springer LNCS 3906, 2006, pp. 13-24. (One of three best paper nominations)

P. I. Cowling, S. M. Remde, K. P. Dahal, N. J. Colledge, "**Evolution of Fitness Functions to Enhance Heuristics**" Extended abstract in proceedings of Graph and Optimization Meeting 2008 (GOM 2008), 2008.

N. J. Colledge, P. I. Cowling, K. P. Dahal, S. M. Remde, E. Selensky, "**A Comparison of Multiobjective and Weighted-Sum Genetic Algorithms for Dynamic Scheduling** planned for submission to Evolutionary Computation, 2010.

N. J. Colledge, K. P. Dahal, P. I. Cowling, S. M. Remde, E. Selensky, "**The Value of Slack Time in Mobile Dynamic Workforce Scheduling**" planned for submission to the European Journal of Operational Research, 2009.

1.2. Thesis Outline

Chapter 2 contains literature relating to scheduling problems in the literature and solution methods.

Chapter 3 details the mathematical model we will use to model the dynamic workforce scheduling problem studied in this thesis and similarities and differences with problems in the literature are identified.

Chapter 4 describes the preliminary work we did with this model, using local search based methods to solve the dynamic workforce scheduling problem. Exact/Hybrid low level heuristics are generated and binary exponential back off Tabu search is explored as a way of reducing neighbourhood sizes and search time.

Chapter 5 identifies the need for the Variable Fitness Function and defines how to use and evolve them. The way it works is compared to common metaheuristics.

In Chapter 6 we present three case studies of the Variable Fitness Function where it has been used to enhance local search based optimisation heuristics for the TSP, a board game (Virus) and the Workforce Scheduling problem. Good evidence of how the Variable Fitness Function enhances the search is shown in the TSP case study. The multi objective nature of the virus game is exploited by the Variable Fitness Function when players are evolved with to change strategy over the course of the game. The workforce scheduling case study showed that some of the evolved Variable Fitness Function made the search exhibit characteristics of the well known right-left shift heuristic, showing that it could be used to evolve new unknown heuristics.

Chapter 7 uses the Variable Fitness Function to enhance metaheuristics for the static and dynamic aspect of the scheduling problem and identifies a case where Variable Fitness Functions may not be ideal.

Finally, in Chapter 8 we discuss the conclusions of this thesis and outline further work

Chapter 2

Literature Review

This section of the thesis reviews related problem literature and methods used to solve these problems. First, literature related to the Resource Constrained Project Scheduling Problem is presented as this problem is one of the closest to ours in the literature.

A brief discussion of solution fitness evaluation is discussed and then selected solution methods split into search based approaches, population based approaches and hyperheuristic approaches.

2.1. Resource Constrained Project Scheduling Problem

The problem we consider is a combination of many of the scheduling problems in literature and reflects the real work situation identified by our industrial sponsor. This section describes some of the relevant scheduling problems in literature.

For the Resource Constrained Project Scheduling Problem (RCPSP) we refer to Pinedo and Chao (1999). The RCPSP is a generalisation of many common scheduling problems including job-shop, open-shop and flow-shop scheduling problems (Hartmann, 1999). Activities have to be scheduled under resource and precedence constraints. Precedence constraints require that a task may not start until all its preceding tasks have finished. Resource constraints require that the resources an activity needs are available when the activity is scheduled. Scheduling an RCPSP involves assigning start times to each of the activities. This problem is Non-deterministic Polynomial-time hard (NP-Hard) (Garey and Johnson, 1979). Literature introducing solution methods for the RCPSP can be found later in this chapter.

The Multiple-mode RCPSP (MRCPSP) (Bouleimen and Lecocq, 2003) extends the RCPSP. In MRCPSP, there is the option of having non renewable resources and resources that are only available during certain periods. In addition, the duration of a task is dependent on the resource that is used during the task, much like the problem we study. As this is a generalisation of an NP-Hard problem, it is itself NP-Hard. Less work has been done for the MRCPSP than the RCPSP and usually involves problems with a small number of modes. This makes the problem slightly harder to solve than the RCPSP. If the dynamic workforce scheduling problem we study was simplified to an MRCPSP then there would be hundreds to thousands of modes for each task which

would grow exponentially with the number of resources – this will be seen later. Mori and Tseng (1997) use a genetic algorithm to solve the MRCPSP. They use a direct representation to encode the solution, storing the order in which the activity should be scheduled and the mode it is to use in the chromosome. They compare the method with a stochastic construction method (STOCON) of Drexl and Gruenewald (1993) for 600 problems of different sizes and complexities. The results suggest that for very small problems, the random stochastic construction is better however with 30 or more tasks, the GA is far superior.

Both the RCPSP and MRCPSP are well studied in literature and an extensive invited review can be found in Brucker et al. (1999) with a more up to date experimental review in Hartmann and Kolisch (2000) and Kolisch and Hartmann (2006).

2.1.1. Solution Representations

Heuristics that gradually improve a schedule into a fitter one generally require two things: a schedule representation scheme and a schedule evaluation procedure. When representing the schedule indirectly, a schedule generation scheme is needed to turn the representation into a schedule. Heuristics that use these representations do not operate directly on a schedule but instead modifies the schedule representation (SR) of the schedule for convenience (Kolisch and Hartmann, 1999).

2.1.2. Schedule Generation Scheme

A Schedule Generation Scheme is an algorithm for creating a schedule given a schedule representation. With the problems discussed the solutions are not usually directly represented. This is usually because the direct representations are harder to manipulate.

There are two main types of schedule generation scheme (and these are adapted for specific Schedule Representations).

A parallel schedule generation scheme operates by starting at time t=0. All activities that have not been inserted into the schedule and whose precedence and time constraints are valid are considered to be scheduled. If more than one activity meets these criteria some method of tie breaking is used (usually from information in the SR). When no more schedules can be inserted into the schedule at the current start time, t is incremented and the process is repeated until all activities have been inserted. A parallel schedule generation scheme is used to solve the Multi Mode RCPSP in Ozdamar (1999).

A serial schedule generation scheme works by considering one task at a time. The order is that of the SR and each activity is scheduled in turn as early as possible considering time and precedence constraints.

Kolisch (1996) found that both are able to provide optimal feasible solutions when there are no constraints on resources, however a serial schedule generation scheme produced active schedules (a schedule where no task can start any earlier without changing another tasks start time) and a parallel schedule generation scheme produced non-delay schedules (in Job shop scheduling this is defined as a schedule where no machine is idle when it could be doing something - see (Sprecher, Kolisch and Drexl, 1995) for definitions).

2.1.3. Schedule Representation

Kolisch and Hartmann (1999) discuss various schedule representation schemes for the RCPSP of which the two most important are Activity List (AL) and Random Key (RK). In a working paper (Debels et al., 2004), a new SR to overcome some of the weaknesses

of RK and calls this the Standardised Random Key (SRK), these will be discussed later. An AL representation of a schedule is a vector that specifies the order of the activity, $\lambda = [a_1, a_2, ..., a_n]$. This can be used in a serial schedule generation scheme naturally or can be used to break ties in a parallel schedule generation scheme. A study (Hartmann and Kolisch, 2000) found that, heuristics that make use of AL generally perform better than those that use RK. This is based on computational evidence and no discussion was given as to why this might be, however we discuss some reasons later.

In RK form, the schedule is represented as a vector with *n* elements and a solution corresponds to a point in Euclidian (*n+1*)-space, such that the *i*-th element of the vector represents the priority of the *i-th* activity. $\rho = [r_1, r_2, ..., r_n]$. An RK can easily be transformed into an AL by sorting the tasks by their priority values. The priority values are usually real numbers which means that this representation can be easily manipulated mathematically. For example, say we have two good solutions, ρ_1 and ρ_2, drawing a line between these points and bisecting it to get a new point $\rho_3 = \frac{\rho_1 + \rho_2}{2}$ may be a good place to look for a new solution.

Priority rule representation is based on a list of priority rules $\pi = [\pi_1, \pi_2, ..., \pi_n]$ where π_i is a priority value used to determine the i-th activity to be scheduled. Priority Rule representation has been first used by (Dorndorf and Pesch, 1995) and adapted for the RCPSP by (Hartmann, 1997) and by (Kolisch, 1996). Some priority rules are shown here in Table 2.1. When this representation is used to build a schedule, the schedule generation scheme determines the *i*-th activity to be scheduled using the *i*-priority rule. This is similar to Hyper-heuristic work reviewed later in this chapter.

Table 2.1. Sample Priority Rules

Rule	Description
LFT	Latest finish time
LST	Latest start time
MTS	Most total successors
MSLK	Minimum Slack
WRUP	Weighted resource utilization and precedence
GRPW	Greatest rank positional weight

see (Hartmann, 1997) (Ulusoy and Ozdamar, 1989) (Kolisch, 1996)

Debels et al.(2004) discusses the reasons why heuristics using AL form might perform better than those using RK form and proposes a new representation, the Standardised Random Key (SRK). The benefit of the SRK is it reduces the search space as there are less schedule representations thus speeding up the search process. The main disadvantage (present in both AL and RK) overcome using this method, is the fact that multiple SRs map to the same schedule. There are 4 reasons this happens:

Scaling of the Euclidean Space (RK only)

In RK form, scaling points can result in the same schedule. For example $\rho_1 = [4,6,8]$, $\rho_2 = [40,60,80]$, $\rho_3 = [3,4]$ would all produce the same schedule, as would many others.

For this reason there is an infinite to one mapping of RK SRs to schedules. This is overcome by replacing the priority values with the activities rank. In our example, $\rho_1 = \rho_2 = \rho_3 = [2,3]$

Precedence constraints (RK only)

In RK form, priority values are not constrained by precedence. This is not a huge problem as the schedule generation scheme should account for this; however it is still increasing the SR to a single schedules ratio. To eliminate this problem, the SRK values of each activity is set to the ranked order of the activities obtained from the schedule generation scheme.

Timing anomalies (RK and AL)

In RK and AL form, activities earlier in the AL (or with a higher priority in the RK) can be scheduled after than those later in the AL (or those with a lower priority in the RK) due to precedence, time or resource constraints. This could result in more than one AL or RK mapping to a single schedule. SRK uses a "topological" ordering

Activities with the same starting times (RK and AL)

There exists two ALs or RKs that will produce the same schedule due to two activities being scheduled at the same time. For a very simple example, say we had two resources and two tasks (that can be done by either resource), with no precedence or time constraints. The two AL representations of this are:

$$\lambda_1 = 1|2 \text{ and } \lambda_2 = 2|1$$

These represent the same schedule. In SRK format, schedules with the same starting time are given the same rank. Thus, these would become $\lambda' = 1|1$

2.2. Solution Evaluation

When modelling a real-world decision problem as a problem of combinatorial optimisation, it is usually assumed that that there is a single underlying objective (fitness) measure to allow automatic comparison between candidate solutions. In real problems, this objective measure is almost always a function of several underlying sub-objectives relating to revenue, cost, staff (Thiagarajan and Rajendran, 2005) and customer satisfaction, sustainability (Viana and de Sousa, 2000) etc., and solution heuristics are often highly tailored to deal with complex problem-specific decision rules. In commercial computerised decision support systems a weight is usually assigned to each sub-objective to reflect its relative importance, and the objective

consists of a weighted sum of sub-objectives (Thiagarajan and Rajendran, 2005). Allowing the user to make these choices of relative importance up front often works well in practice, since it allows (and empowers) users to make difficult a priori decisions of importance as a decision support system is implemented, and potentially to engage in "what if?" analysis of different sub-objective weights, when time allows (MacCarthy and Wilson, 2001). The Valuated State Space method (Vanesian et al., 2007) further empowers users by reducing the problem of choosing weights to one of ranking solutions. Multiobjective approaches (Deb, 2004) offer an interesting way forward that does not require the user to consider weights directly, although the issue of effectively comparing many different Pareto optimal solutions have generally prevented their application in problem areas where understanding a single solution is already a significant challenge for the user (Josephson, 1998).

2.3. Search Based Solution Methods

2.3.1. Local Search Methods

Local search (Aarts and Lenstra, 2003) methods operate on a schedule or schedule representation known as the candidate solution. It requires the definition of a neighbourhood of a solution that defines solutions in the search space which are neighbours. The search is made by moving from one candidate solution to another candidate solution from its direct neighbourhood. A neighbourhood defines solutions which can be obtained by making (usually a small) change to the current solution. Simple moves for combinatorial optimisation problems include 2-opt (where 2 components of the solution are "swapped" or changed), k-opt (where k components of

the solution are changed), insert (where a new component is inserted into the solution) and remove (where a component is removed from the solution). The moves are based on local information, and continue until a termination condition is met (usually a certain number of iterations or a certain amount of CPU time spent). Figure 2.1 shows a simple improvement only local search for a minimisation problem. At each iteration a new solution is picked from the current solutions neighbourhood if it improves the fitness.

```
N(s)  defines the neighbourhood of solution s
f(s)  defines the fitness of solution s
LocalSearch(s)
        do
                choose s' ∈ N(s)
                if f(s') < f(s) then s := s'
        loop until terminate
```

Figure 2.1. A Simple Greedy local search

2.3.2. Fast Local Search

Fast Local Search (FLS) along with Guided Local Search (GLS) (Tsang and Voudouris, 1997) has been used to solve British Telecom's Workforce Scheduling problem which is a lot like the one studied in this thesis. The schedule is represented by an activity list and the neighbourhood is a defined as swapping two tasks.

Fast Local Search (Burke, Cowling and Keuthen, 2001) enhances local search by associating an activation bit with each position in the permutation. Initially all these bits are set to 1 (or "on"). The activation bit is set to 0 (or "off") if every possible swap has been made involving that position without an improvement. When one is found the activation bits of the two positions swapped are set back to 1. Only positions with activation bits of 1 are considered in the neighbourhood, which has the advantage of speeding up the search as repeated bad moves are not tried.

2.3.3. Guided Local Search

Simple local search algorithms like this suffer from settling in local optima – a state where no neighbouring solution has a better fitness. The Guided Local Search (Voudouris and Tsang, 1999) (GLS) is a method for overcoming this problem. Guided Local Search attempts to modify the fitness function to change the direction the search heads when a local optima has been found. Features are identified and penalties for solutions exhibiting these features are increased when the solution is headed toward local optima. It redefines the objective function thusly $f'(s) = f(s) + \lambda \sum_{i=1,F} p_i I_i(s)$ where λ is the weighting for the GLS, F is the number of features, p_i is the penalty value for the i-th feature and $I_i(s) = 1$ when *s* exhibits feature *i*, otherwise 0.

When the search settles on a local optima the utility of penalising each feature is calculated and the feature or features with the largest utility will be penalised by increasing their penalty values. This has the effect of changing the fitness function and forcing the search to move in another direction away from the local optima.

2.2.4. Simulated Annealing

Simulated Annealing (SA) is a local search method that was inspired by the physical annealing process studied in statistical annealing (Aarts and Korst, 1989). SA was first applied to the RCPSP by Boctor (1996). The process uses an iterative neighbourhood generation procedure that follows search directions which improve the fitness function value. The difference being, that while exploring the solution space the SA method offers the possibility to accept worse solutions. This helps the search have a chance to escape local optima.

```
SimulatedAnnealing(s)
    do
        choose s' ∈ N(s) at random
        Δ = f(s') − f(s)
        if Δ < 0 then
            s := s'
        else
            P = e^(−Δ/T)
            choose y_random uniformly at random on [0,1]
            if P > y_random then
                s := s'
            end if
        end if
    loop until terminate
```

Figure 2.2. Simulated Annealing Pseudo Code

Figure 2.2 shows that the probability of accepting a worse solution decreases as the temperature (*T*) decreases and that this probability is also proportional to how bad the move is. The temperature is decreased at each iteration which helps diversity at the beginning and helps intensify the search toward the end.

Bouleimen and Lecocq (2003) apply this method to the RCPSP and the MRCPSP. For the multimode version, the neighbourhood also includes mutation of an activity's mode. In the survey the heuristic was shown to be competitive and performed well ranking about midway of the tested heuristics (Kolisch and Hartmann, 2006).

2.2.5. Tabu-Search

Tabu search (Glover, Taillard and De Werra, 1993) is a method to help guide a local search process. Modern Tabu search is based on the seminal work by Glover (1986) and Hansen (1986). Tabu search works by restricting and guiding the search from the

history of the search, exploiting good moves and avoiding bad ones. In order to do this, Tabu search stores information about the search.

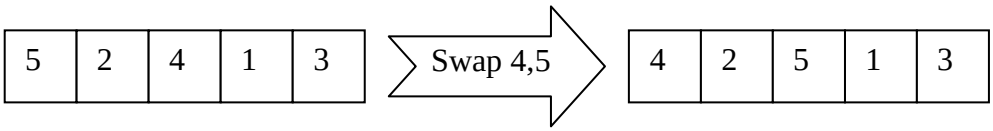
The following section will discuss the features of Tabu search with an example of how they could be used in a combinatorial problem and how they have been used in literature. Next uses of Modifications of the Tabu search framework will be discussed and finally its similarity with other meta-heuristics. Pseudo code for a simple Tabu search is given in Figure 2.3.

1. Initialisation
    a. $x_{current}$ - random starting solution is generated
    b. *Initialise H* (the memory of the seach)
2. Choice and termination
    a. Generate the subset $CandN(x_{current}) := \{x'$ belongs to $N(x_{current}) :$ the move from $x_{current}$ to x' is not Tabu or it satisfies an aspiration criteria$\}$
    b. Choose x_{next} from *CandN* such $f(x_{next})$ is minimised (for a minimisation problem)
    c. Make the move from $x_{current}$ to x_{next} Tabu
    d. Terminate is maximum number of iterations has been reached.
3. Update
    a. Set $x_{current} = x_{next}$
    b. If $c(x_{current}) > c(x_{best})$ set $x_{best} = x_{current}$
    c. Repeat step 2.

Figure 2.3. Tabu Search Pseudo Code

Features

This section discusses the features of Tabu search and illustrates the memories of Tabu search and how they can be used. A simple combinatorial optimisation problem is used to illustrate the features. Neighbourhoods in Tabu search are generated by performing moves on the current solution. This is easily defined as the swapping of two elements in an Activity List:

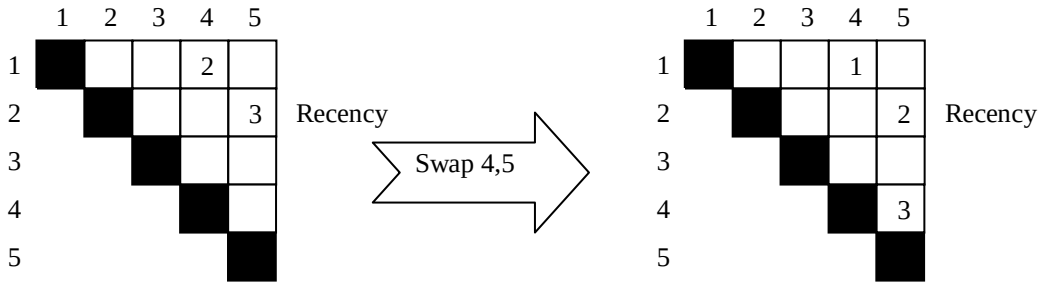


Here we can see that the move "Swap 4,5" will the result in the order (5, 2, 4, 1, 3) changing to (4, 2, 5, 1, 3). Tabu search uses a memory (or history) of the search. The key features of Tabu search will be discussed next.

Recency

Recency is a short term memory and records how long moves are Tabu for. This can be used to avoid undoing a recent move or stopping the repetition of bad moves.

For our combinatorial optimisation example, a simple table can be constructed to store recency. The following diagram shows how the table will be updated after a swap (with a Tabu tenure of 3):



The above diagram shows that "Swap 2, 5" was used most recent (and has a Tabu tenure of 3) and "Swap 1, 4" was used before that (and has a Tabu tenure of 2). After the move "Swap 4, 5" has been made, we can see the effects it has had on the Tabu memories. The move now has a Tabu tenure of 3 (and the other Tabu tenures have been decreased by 1).

Choosing a Tabu tenure is a problem in itself. These are usually done statically or dynamically, or attribute dependant dynamically.

- Static rules usually define the Tabu tenure as fixed (e.g. t=7) or as a function of the problem size (e.g. $t=n^{½}$ – where n is an measure of the problem size).

Abdulah et. al. (2007) uses simple Tabu search for capacitated examination timetabling. The neighbourhoods are very large and so improvement graphs (Thompson and Orlin, 1989) are used to first cull bad solutions before the Tabu search is used to evaluate the neighbourhood. Recency information is used and Tabu tenures of t=2, t=4 and t=6 are experimented with and benchmarked against commonly used timetabling problems in (Carter and Laporte, 1996). The method provides competitive results and in two of the six instances, the best results (in these cases a Tabu tenure of t=2 and t=4 both got the same result). From my analysis of the results it is clear to see that the objective value (which is to be minimised) is proportional to the Tabu tenure. Work complementary to this would be to try Tabu tenure of 1 and 0, as these results suggest the local search may be better than Tabu search. Laugna et. al. (1999) did experiments to find optimal Tabu tenure. A Tabu tenure of $2n^{½}$ is found to be optimal for the linear ordering problem they studied.

- Simple dynamic rules pick may pick t randomly each time between two bounds t_{min} and t_{max}. (e.g. t_{min}=1 and t_{max}=5 or t_{min}=0.9 $n^{½}$ and t_{max}=1.1 $n^{½}$). Tailard et. al. (1991) uses this method to solve the quadratic assignment problem and recalculates the Tabu list size every $2t_{max}$ iterations.
- An attribute dependant dynamic Tabu tenure selection, the Tabu tenure is determined as in the simple dynamic method above, except that the bounds are determined by an attribute of the move (e.g. Quality or influence). This is very true for difficult problems such as scheduling (Dell'Amico and Trubian, 1993) where Tabu tenures are given relating to the quality of move performed; longer Tabu tenures are given to the reverse move of high quality moves.

Research (Glover and Taillard, 1993) (where Tabu search is applied to a quadratic assignment problem), (Taillard, 1991) and (Dell'Amico and Trubian, 1993) show that dynamic rules tend to work better than static rules where they use Tabu search to solve.

Frequency

Frequency records how frequently a move has been made. This can be used to try those moves that have not been used much (to try new search directions). Frequency is a long

term memory and is less often used. Laguna et. al. (1999) and Glover et. al. (1993) use the frequency memory as follows: The move evaluation function is adjusted to by a weight frequency of the move. They note that the weight is dependent on move type and on the neighbourhood, however in the quadratic assignment problem, it is approximately proportional to $m^{½}\theta$ where m is the size of the neighbourhood and θ is the standard deviation of the frequencies.

A outcome of using such an adjustment was that the optimal Tabu list size did not grow with the problem size and that dynamic Tabu list sizes became less important.

Quality

The Quality of a move indicates how good the move is or how well it performed. This is a useful indicator to how well it will perform in future. The Quality of a move in (Dell'Amico and Trubian, 1993) is used to determine the Tabu tenure applied in a job shop scheduling problem. They use 3 rules to determine how the Tabu tenure is adjusted depending on how the move affects the search. If after the move:

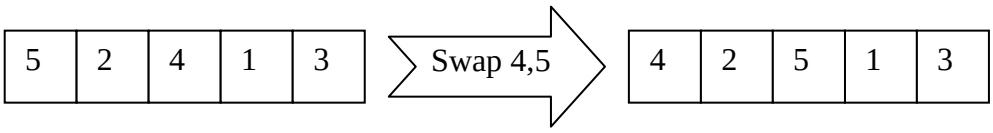
- The objective value is better than the best, the Tabu length is set to 1.
- The objective value is better than the previous iteration (we are in an improving phase) and the Tabu length is greater than a threshold *min* then decrease the Tabu tenure.
- The objective value is worse than the previous iteration (we are in an improving phase) and the Tabu length is greater than a threshold *max* then increase the Tabu tenure.

min and *max* are chosen randomly every 800 iterations. A similar method where min and max are fixed to solve the 2-dimensional orthogonal packing problem in (Harwig, Barnes and Moore, n.d.). This involves packing 2-dimensional orthogonal objects into bins as optimally as possible (without wasted space). A solution is first built using a greedy method and then an adaptive Tabu search is used to improve it. The adaptive

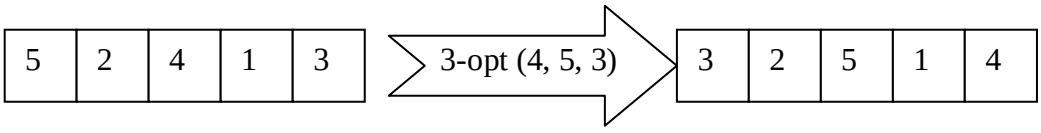
Tabu search works with a dynamic Tabu tenure, adjusting the tenure making it short if there is an improvement, and longer if there is not. Ejection chains are also used to make infeasible moves feasible. Say, for example putting an item in a bin was impossible (because there is no room), an item in the bin maybe moved to another bin to make this move possible (in which doing so might need to move another item, thus creating an ejection chain). In their words, "an ejection chain is an imbedded neighbourhood construction that compounds the neighbourhoods of simple moves to create more complex and powerful moves". The results are compared to literature and improve previous results by an average of 25%.

Influence

Influence records how a move influenced the search (usually a measure of how much change in the solution the move will make). In our combinatorial optimisation, influence is easily defined as the number of positions the elements have moved. In the "Swap 4, 5" example, the influence is 2.



If our moves were more complex, for example we could consider all n-opt moves where $1<n<5$, then we would sum the absolute difference in the positions of all the elements. For example:



would have an influence of 2+2+4=8. Influence is used in (Laguna, Marti and Campos, 1999) for intensification and diversification purposes. Moves with high influence can be seen as moves which will diversify the solution (as it will move to a region of the search

space far from the current solution) and results with low influence will intensify the search.

Aspiration

Another feature of Tabu search is Aspiration criteria. Aspiration criteria are introduced to allow exceptions where Tabu moves could be considered. The most commonly used aspiration criteria is to remove the Tabu restriction if a Tabu move yields a better solution than the best found so far (used by for example (Dell'Amico and Trubian, 1993), (Glover and Taillard, 1993)). This can be useful when a recent move has made a Tabu move desirable again. Aspiration criteria are not always used.

Intensification and Diversification

The features described so far have shown how TS intensifies the search, using aspiration criteria and Tabu restrictions to guide the search to good solutions and avoid undoing good moves respectively. Good search strategies need diversification methods too. This can be done by using the frequency and influence memories of Tabu search to try moves not yet tried or those which will change the solution greatly. Much work has been done on Tabu search but rarely are all the features used. This may be as the some features maybe problem dependant and thus not applicable or no advantage would be gained by using them.

A linear ordering problem is solved in Laguna et. al. (1999) using a Tabu-search with intensification and diversification strategies. Diversification and intensification are done in two ways. Firstly a Tabu search is used and switches between intensification and diversification phases. During an intensification phase, moves are selected with a probability proportional to its influence. During a diversification phase, the Tabu search uses the frequency memory to make moves that are likely to explore areas of the search space previously unexplored. Secondly, long term intensification is achieved using path

relinking (Glover, 1998) (at the end of an intensification stage, path relinking is used in the direction of other elite solutions found so far in the attempt to find better solutions) and long term diversification is achieved by trying to construct solutions that are "far away" from the elite set of solutions found so far. The long term methods are used when the Tabu search converges. The method is used as stated, but also variations where long term intensification/long term diversification/long term intensification and diversification are not used. Experiments are done to good values of Tabu tenure and length of the intensification and diversification phases. These are found to be $2n^{½}$, n and 0.5n respectively (where n is the problem size). Using all the diversification and intensification methods was shown to be the best method when it was compared to state of the art methods used in Chanas and Kobylanski (1996) and Becker (1967). It produce the fittest results and although it was not the fastest (and not the slowest), when all the methods are given the same amount of CPU time (0.5 seconds, 1 second, 20 seconds…), it still produced the fittest results.

Hybrid Tabu search Methods

In Burke, De Causmaecker and Berghe (1999) they attempt to tackle a nurse rostering problem in Belgian hospitals with a hybrid Tabu search. This problem is highly constrained and usually solved manually. It involves assigning duties to people with different qualifications, regulations and preferences. The approach they use is a hybrid approach using heuristics to create and repair schedules. The Tabu search performs the main search processes (using an accent/decent search – accepting both positive and negative changes) with a heuristics to try and fill in gaps (e.g., scheduling of weekends is done ignoring some constraints – specific details of these heuristics are not given). The Tabu search only uses the first Tabu memory – recency and the Tabu list length is not revealed. The results are compared to those produced by hand and variations of their

search method and a steepest descent algorithm. They show that their heuristic is superior to the other methods and (when compared to manually scheduling the workforce) faster.

Verhoeven (1998) uses Tabu search to solve the resource constrained scheduling problem. Only recency is used and a simple neighbourhood function is defined. The heuristic also implements a simple restart method, which upon finding no improvements for a certain number of iterations, restarts from the best solution found so far. The results were competitive. No comparison between using the method with and without the restart was made, so its effectiveness can not be derived.

Tabu search Hyper-heuristics

Burke et al. use Tabu search again, but this time as a hyper-heuristic (Burke, Kendall and Soubeiga, 2003) to show that a Tabu search based hyper-heuristic can be used to solve different types of scheduling problem with good results. In their hyper-heuristic, the Tabu search is used to Tabu a low level heuristic when it performs badly. Associated with each low level heuristic is a rank and the rank is adjusted depending on how it performs. At each iteration, out of the non Tabu low level heuristics it selects the one with the highest rank. It actions the low level heuristic and,

- If it improves, its rank is increased
- If it makes no change, its rank is decreased and it is made Tabu
- If it makes worse the solution, its rank is decreased and the Tabu list emptied

Six low level heuristics are used which are 3 variations of a “swap” move and “move” move where which slots are chosen by different heuristics. The technique is applied to nurse rostering and university timetabling and compared to a Genetic Algorithm. For the rostering, the hyper-heuristic outperformed Genetic Algorithm in some criteria but not others; however the hyper-heuristic results were still competitive. In the time tabling

experiments, the Tabu search did very well and in the majority of instances, beat or equalled the Genetic Algorithm. Perhaps

The is a choice function like hyper heuristic, and in work extending this (Burke and Soubeiga, 2003), they compare it to a Choice Function hyper heuristic and use the methods to solve the nurse scheduling problems Different Tabu tenures and negative reinforcement learning methods for adjusting the rank of the low level heuristics are experimented with (Tabu Tenures or 2, 3 or 4 and negative reinforcement learning values of 2, 3 or 4). The resulting hyperheuristic was able to produce similar quality results with half the CPU time.

(Kendall and Mohd Hussin, 2005) use a method similar to the Tabu search hyper-heuristics described above. Instead of using a choice function or reinforcement learning mechanism to choose the next low level heuristic to apply at every iteration, every low level heuristic (that is not Tabu) is considered and the best improvement only solution is accepted. They believe that "by allowing the low level heuristics to compete at each iteration and selecting the heuristic with the best performance will help to balance intensification and diversification". They use their method on an examination timetabling problem (with standard datasets used in literature). Tabu durations between 0 and 4 (using around 13 low level heuristics) are tested for 10 minute runs. The Tabu duration of 2 is shown to be optimal and a 4 hour long run is done to compare improvement over a long period. The results are compared to literature and shown to be competitive, some of the runs coming very close to the best in literature, however some deviate by a large amount. The longer runs produced results approx 4-14% fitter than the short run. Such a varied range could be a result of incorrect low level heuristics – there are the correct ones to deal with some instances, but not others. In (Kendall and Hussin, 2005) they apply the above method to a timetabling problem in the MARA

University of Technology. In addition to the 0-4 Tabu tenure lengths they use a dynamic Tabu tenure length that is picked at random between 0-4 at every iteration. It is shown that the results are at least 80% better than manually created ones.

2.2.6. Variable Neighbourhood Search

Variable Neighbourhood Search (Mladenovic and Hansen, 1997) (VNS) is based on the idea of systematically changing the neighbourhood of a local search algorithm. Variable Neighbourhood Search enhances local search using a variety of neighbourhoods to "shake" the search into a new position after it reaches a local optimum. Several variants of VNS exist as extensions to the VNS framework (Hansen and Mladenovic, 2001) which have been shown to work well on various optimization problems. Variable neighbourhood search is relatively easy to implement. The shake moves can simply be "chained" random local search moves as in (Lin, 1965) but if this is not adequate, new shake moves may have to be implemented.

Reduced Variable Neighbourhood search (rVNS) (Hansen and Mladenovic, 2001) is an attempt to improve the speed of variable neighbourhood search (with the possibility of a worse solution). Usually, the most time consuming part of VNS is the local search. rVNS picks solutions randomly from neighbourhoods which provide progressively larger moves. rVNS is targeted at large problems where computational time is more important than the quality of the result. In combinatorial optimisation problems, local search moves like "swap two elements" are frequently used, and (Fleszar and Hindi, 2004) for RCPSP as well as others such as (Garcia et al., 2006) apply VNS by having the neighbourhoods make an increasing number of consecutive local search moves. Sevkli et. al. (2006) however defines only two neighbourhoods for

VNS applied to the Job Shop Scheduling Problem, a swap move and an insert move, which proves to be effective.

Interest in VNS is growing and a recent survey by Hansen et. al. (Hansen, Mladenović and Moreno Pérez, 2008) states "Interest in VNS is clearly increasing. This is evidenced by the increasing number of papers on that topic (just a few ten years ago, about a dozen five years ago, and about 50 in 2007".

2.4. Population Based Solution Methods

2.4.1. Genetic Algorithms

Since their introduction by Bremermann (1958) and Fraser (1957) and the seminal work done by Holland (1975), genetic algorithms (GAs) have been developed extensively to tackle problems including the travelling salesman problem (for example (Whitley, Starkweather and Shaner, 1991)), bin packing problems (for example (Falkenauer, 1996)) and scheduling problems (for example (Terashima-Marin, Ross and Valenzuela-Rendon, 1999)). A Genetic Algorithm tries to evolve a population to a higher level of fitness by a process analogous to evolution in nature. With GAs, the genotype of a problem to be solved are stored in a chromosome. Initially, a population is generated either randomly or with some knowledge (for example, for the RCPSP problem, random, precedence valid, chromosomes could be generated (Hartmann, 1997)).

At each generation (iteration) of the GA, the entire population is evaluated using the schedule evaluation function. A new population is created from the existing population using selection, crossover and mutation. The new population is then

recombined with the old population using a replacement strategy. The process is repeated until a termination condition which is usually a certain number of generations.

Crossover

Crossover takes two parent chromosomes and combines them to create one or more children. There are several crossover operators (Hartmann, 1997). The three generic popular crossover operators are described here.

Uniform Crossover - This is usually used to create two children. Each gene of the first (second) child is taken from the father (mother) with probability *p*, otherwise it comes from the mother.

Mother	a	b	c	d	e	f	g
Random	0.1	0.5	0.3	0.8	0.3	0.4	0.9
Father	A	B	C	D	E	F	G

With a p=0.4

Daughter	A	b	C	d	E	f	g
Random	**0.1**	0.5	**0.3**	0.8	**0.3**	0.4	0.9
Son	a	B	c	D	e	F	G

Changing the value of *p* changes the magnitude of effect the crossover has. Low values means that the son and daughter are similar to the mother and father and values close to 0.5 mean they are more of a combination. This is more suited to problems which do not have a clear structure in chromosome, for example Random Key or Priority Rule representations, or the representation for the knapsack problem used by (Chu and Beasley (1998).

One Point Crossover – One point crossover again creates two offspring. The mother and father are both split at the same random point, and then the different sections swapped to make two children.

Mother	a	b	c	d	e	f	g
Father	A	B	C	D	E	F	G

With a split after the 3rd gene:

Daughter	a	b	c	D	E	F	G
Son	A	B	C	d	e	f	g

This method keeps chains of genes together so is good for representations where order of the genes has an effect on solution quality.

Two Point Crossover – Two point crossover is similar to one point except that there are two points where the chromosome is split:

Mother	a	b	c	d	e	f	g
Father	A	B	C	D	E	F	G

With a split after the 3rd and 5th gene:

Daughter	a	b	c	D	E	f	g
Son	A	B	C	d	e	F	G

In a study done by Hartman (1997) on the RCPSP, the two point crossover was shown to be the most effective, with uniform and one-point coming second and third. Hartman notes that "The two-point crossover operator appears to be capable of inheriting building blocks that contributed to the parents' fitness (for much larger projects, even more than two cuts may probably be advisable)." indicating that a *k*-point crossover might be an interesting area of research as this allows more "blocks" to be considered.

Mutation

To help diversification, GAs have a fixed, small probability of mutation (P_m) of perhaps 0.01 or less. During mutation, each "bit" of the chromosome has P_m probability of being changed. With RK chromosomes, this could simply mean generating a new random key number for that bit. In a permutation based AL chromosome this becomes more tricky as it needs to keep a valid chromosome. This is usually overcome by P_m becoming the probability of swapping the activity with the one to the right of it. For all positions $i = 1,..., J-1$ I_i and I_{i+1} are swapped with probability P_m.

Lower mutation rates can lead to genetic drift (unless using replacement methods that favour diverse methods). This is where all the individuals' genes become the same and converge on an optima (not necessarily the global optima).

Selection (Replacement Strategies)

When the new pool of offspring has been created, the original population and the new offspring need to be combined and reduced to keep the population size fixed.

Full Replacement – the population is replaced by offspring and the parents discarded.

Ranked – the parent and the offspring population are combined and ranked by their fitness and we keep the best ones.

Proportional Selection – this is analogous to a roulette wheel, where all the population has a section of the wheel and the size of that section is proportional to the deviation from the best individual. The wheel is spun and the one it lands on is removed from the population. This is repeated until the population is at its correct size. More formally, $F(I)$ is the fitness of individual I and P is our population. If $f_{best} = \min(f(I) \mid I \in P)$,

the probability that an individual will die (and thus will not be moved to the next generation) is defined by $p_{death}(I) = \frac{(f(I) - f_{best} + 1)^2}{\sum_{I' \in P}(f(I') - f_{best} + 1)^2}$.

2-Tournament – two different individuals are chosen at random and the weaker of the two is removed until the population is back to normal size.

3-Tournament – three different individuals are chosen at random and the weakest of the three is removed until the population is back to normal size.

Elitism – elitism keeps a number of the best solutions. This is to make sure the population keeps good results and can be combined with another method of replacement.

The best parameters

Hartman experimentally tests the three main types of GA different mutation rates (0.01, 0.05, 0.10), selection methods and different crossover operators to see which combination works the best (Hartmann, 1997). He found that Permutation GAs using ranked selection, two-point crossover and a mutation rate of 0.05 worked best. The "best parameters" of course are problem dependant, which self adapting GAs try to overcome.

Self adapting Genetic Algorithms

Self adapting Genetic Algorithms are those which tune their parameters automatically to achieve better results for a given problem. The idea was first developed by Degris et al. (1999) (though not applied to the RCPSP). Hartman (2002) first developed this method for the RCPSP and used a GA to optimise the order of the activities scheduled. He also added a gene which represented a bit which if set to 1, a serial schedule generation scheme is used, otherwise a parallel schedule generation scheme was used and the

results are competitive with other state-of-the-art heuristics evaluated by Kolish and Hartmann (2006).

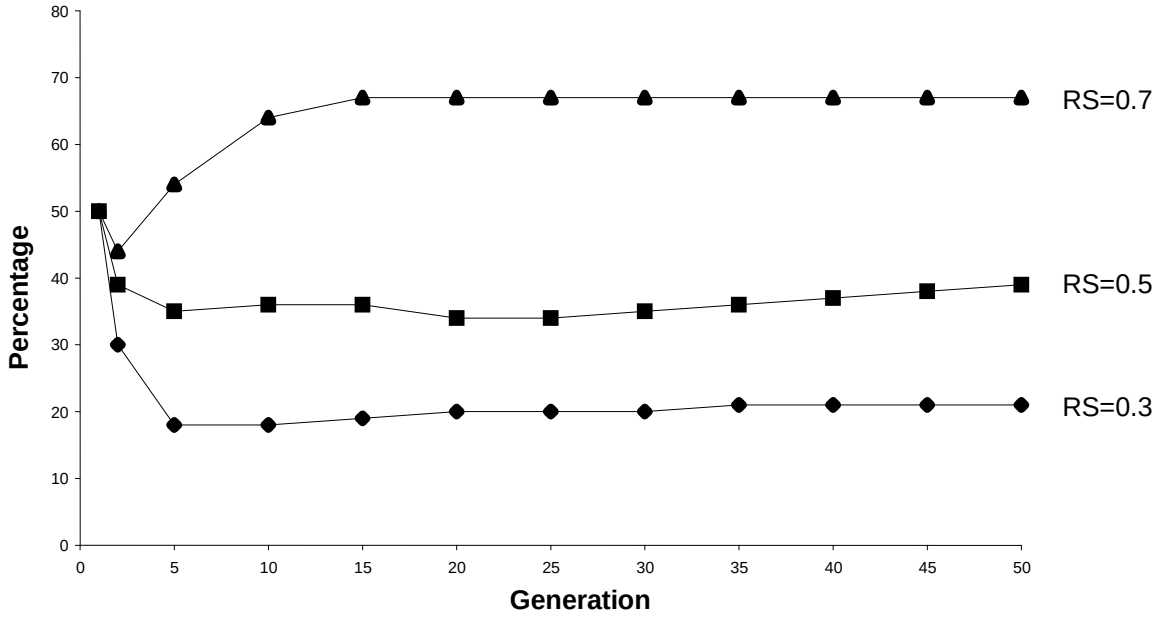


Figure 2.4. Average percentage of the population using Serial schedule generation scheme over generations (created from the data in (Hartmann, 2002))

Interesting to note were the statistics of the schedule generation scheme usage presented as a graph in Figure 2.4. From this graph we can see that with scarcer resources (lower values of RS) the parallel schedule generation scheme is favoured over the serial and with more resources available the serial schedule generation scheme is favourable. It is interesting to note that the only the RS=0.7 favours serial schedule generation scheme when it has been suggested in literature that serial is superior to parallel. This would imply that most of the literature has used problems with high resource availability.

Another method is adaptive crossover and mutation rates. With this method, the crossover and mutation rates adapt depending on the performance of the GA.

When there is no improvement in the average fitness over three generations, the mutation rate and crossover rate are modified as follows:

New Mutation Rate = (Old Mutation Rate + 1) / 2

New Crossover Rate = Old Crossover Rate / 2

If the average fitness rate over the last 3 generations has improved, they are modified as follows:

New Mutation Rate = Old Mutation Rate / 2

New Crossover Rate = (Old Crossover Rate + 1) / 2

This helps intensification and also stops the genetic drift by increasing the crossover rate when little improvements are found. This does not usually perform as well as fixed crossover and mutation rates. Having said this, finding the optimal crossover and mutation rates is hard and usually problem specific and having adaptive rates eliminates this time consuming process.

2.4.2. Scatter Search

Laguna et. al. (1999) discuses the basic scatter search heuristic and shows its application a 0-1 knapsack problem. To apply scatter search to an RCPSP one would need to use a RK. Like a GA, a scatter search works on a population (known as the reference set) but where a GA uses a larger populations, SS works best with a population of around 10 (the population size is known as b). The reference set is split into two, a good set and a diverse set. There are 5 basic functions that need defining to implement a scatter search.

Diversification Generation Method - This function is like the initial population generation of a GA. Its function is to create a diverse range of trial solutions from an arbitrary solution (or seed solution). In the 0-1 knapsack problem, random bits are used and then the complement of these is also used.

Improvement Method - This function should improve an individual. It also should be able to make invalid individuals valid. For example, in the knapsack problem, if the

knapsack was too full, items were removed one by one and if the knapsack wasn't full, items were added one by one.

Reference Set Update Method – this method builds and maintains the reference set. Solutions maybe considered good either because of their quality or their diversity. This is analogous to selection in GAs. In the 0-1 knapsack problem, the best solutions by their fitness are put in the best set. Diverse solutions are those defined as having as many bits set different to the best, and the most diverse are put in the diverse set. These are combined to create a new reference set.

Subset Generation Method – This method creates subsets of the reference set, which will be used in the solution combination method to make new solutions. Typically, all 2-element subsets of the reference set are picked. Then 3 element subsets are generated taking all the 2 element subsets and adding the next best solution (measured by the fitness value). 4 element subsets are then created in the same way and finally subsets consisting of the *I* best solutions for *I = 5 to b.*

Solution Combination Method – This method combines the elements in the subsets generated in the previous step to create a new solution(s). In the 0-1 knapsack problem, the solutions to be combined are combined with a weighted sum and then rounded to the nearest whole number.

For example $p_1=0,1,1,0$, $p_2=1,1,0,0$, $p_3=0,1,1,1$, $f(p_1)=3$, $f(p_2)=5$, $f(p_3)=2$

The new solution would be

$p_4=(0*3+1*5+0*2)/10, (1*3+1*5+1*2)/10, (1*3+0*5+1*2)/10, (0*3+0*5+1*2)/10$

$p_4=0.1, 1, 0.5, 0.2 = 0, 1, 1, 0$

Permutation based problems could use crossover operators as in the GA.

The following pseudo code shows how the heuristic works.

1. Initially use the Diversification Generation Method to generate a pool of solutions.

2. Use the Improvement Method to improve these solutions.
3. Use the Reference Set Update Method to sort these solutions.
4. Use the Subset Generation Method to create subsets of solutions.
5. Use the Solution Combination Method to create new solutions from the subsets.
6. Use the Improvement Method to improve the new solutions.
7. While the stopping criteria are not met, go to step 3.

2.4.3. Scatter Search/Electromagnetism Hybrid

The SS/EM hybrid (Debels et al., 2006) uses the scatter search hyper-heuristic with a few modifications. First of all it uses the Standardised Random Key representation to reduce the search space. The Solution combination method is modified for electromagnetism. The solution generation method creates pairs of solution for the "good" set, and pairs from the "good" and the "diverse" set. Solutions sets from the "good" set are combined using 2-point crossover as in GAs and solutions sets containing a solution from the "good" and one from the "diverse" are combined using EM.

In EM, a charge is associated with each solution. A fit solution will produce a positive attracting charge and an unfit solution will produce a negative repelling charge. In the SS/EM hybrid, only one force is used (this was found to be the best way experimentally). An attracting force is applied to the "diverse" solution from the "good" solution in an attempt to pull it into an area of the solution space that is fitter.

This method performs extremely well for the RCPSP problem and it should be noted that this method consistently came top three in the experimental investigation (Kolisch and Hartmann, 2006).

2.4.4. Sawing Evolutionary Algorithms

"SAWing" Evolutionary Algorithms are the evolutionary equivalent of Guided Local Search. "SAWing" or Stepwise Adaptation of Weights, takes an existing EA with a weighted sum fitness function alters the weights every *n* chromosome evaluations (usually *n* is the population size.). When applied to genetic programming for data mining (Eggermont, Eiben and van Hemert, 1999) mixed results were reported. SAWing EAs were then used to solve Constraint Satisfaction Problems (CSP) (Eiben and van Hemert, 2001) with more consistent results. Stepwise Adaptation of Weights were used in conjugation with standard EAs and compared results to the standard EAs. A weight was given to each constraint of the CSP and these were adjusted by the SAW EAs in response to the search process. This leads to worse results in the short term but is beneficial in the long run.

Precision SAW is introduced in (Eggermont and van Hemert, 2000) in an attempt to improve performance on problems with real valued variables (CSP use 1/0 for true/false) when applied to symbolic regression. Here the weights of are adapted in response to the error in an attempt to speed up the process and indeed this was the case. This work is further refined in (Eggermont and van Hemert, 2001).

2.5. Hyper Heuristics

Most meta-heuristics have to be tailored to a specific problem and are too problem specific or require a lot of knowledge and this can often be expensive and time consuming especially when trying to find the right parameters for these meta-heuristics. Hyper-heuristics raise the level of generality to a heuristic which allows it to be applied to many different optimisation tasks with little modification. Broadly speaking, the term

hyper-heuristics describes the process of (meta-)heuristics that choose (low level) (meta-)heuristics to the problem in hand.

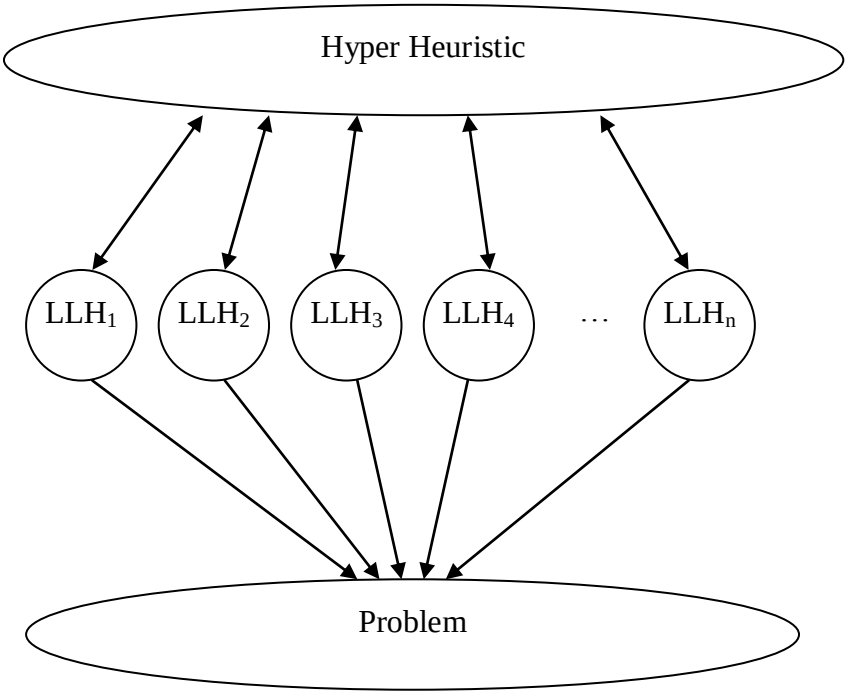


Figure 2.5. Hyper Heuristic Framework

The term hyper-heuristics was first coined in 1997 by Denzinger, Fuchs and Fuchs (1997) as a protocol that chooses and combines several AI methods. The term "hyper-heuristic" was reintroduced by Cowling et.al (2001) and is described as a heuristic which "… is able to choose between low-level heuristics without the need to use domain knowledge, by using performance indicators which are not specific to the problem each time a low-level heuristic is called, in order to decide which heuristic to use at a particular point in the search space."

Evolutionary approaches to scheduling would generally involve populations of schedules which are to be evolved over time into better, fitter schedules. A hyper-heuristic evolutionary approach to scheduling would use a population of low level heuristics and it is these which would evolve.

2.5.1. Evolving heuristic choice

Before the term hyper-heuristics was used there was work done in literature with the same principal idea. A genetic algorithm was developed which has hyper-heuristic characteristics to solve an open shop scheduling problem (OSSP) (Fang, Ross and Corne, 1994). They test three methods. OSSP involves a collection of machines and a collection of jobs and each job is comprised of operations. An operation is an ordered pair (a, b) in which a is the machine on which the operation must be performed, and b is the time it will take to process this operation on machine a. A feasible OSSP schedule assigns a start time to each operation, satisfying the constraints that a machine can only do one operation at once and that two or more operations of the same job cannot be processed at the same time.

They develop four GAs, the most significant of which is the fourth one which they call "evolving heuristic choice". It uses a chromosome double the length of the number of jobs, where the chromosome "abcd…" tells the schedule generation scheme to try to use the a-th heuristic to choose an operation from the b-th unscheduled job and insert it as early as possible, then use the c-th heuristic to choose an operation from the d-th unscheduled job and insert it as early as possible…, etc. In this heuristic the hyper heuristic is the GA and the low level heuristics are the operation choosing heuristics. This could be taken further to have another heuristic choose which job to pick as well.

2.5.2. Choice Function

The choice function (Cowling, Kendall and Soubeiga, 2001) is a method for choosing which low level heuristic to apply. Which LLH to use is determined by a weighted sum of various performance indicators of each LLH. The choice function is as follows:

$$f(h_j) = \alpha f_1(h_j) + \beta f_2(h_k, h_j) + \delta f_3(h_j)$$

where $f_1(h_j)$ is a measure of the improvement that was gained last time the heuristic *j* was used and maybe negative if the solution quality decreased. This helps favour LLH's that are improving the solution.

$f_2(h_k, h_j)$ is a measure of the improvement that was gained last time the heuristic j was used after the heuristic *k* and again may also be negative if the solution quality decreased.

$f_3(h_j)$ is the amount of time since the heuristic *j* was applied last. This helps when the search settles on a local optima and improvements are not being found. It could be that a heuristic that has not been applied for a while may help, but because it has only ever before been found to produce bad results, it would not normally be applied. This also can help with diversification, as sometimes making a bad move will help in the long run.

Cowling, Kendall and Soubeiga (2001) use the choice function with a local search and applied to a sales summit scheduling problem. They use 4 different methods for choosing the LLH depending on the choice function. Firstly, "StraightChoice" simply chooses the LLH with the highest value of *f*. Secondly, "RankedChoice" ranks the LLHs according to their Choice Function and evaluates a proportion of the best and apply the one which yields the best result. Thirdly, "RouletteChoice" uses a roulette style selection with the chances of each proportional to their relative Choice Function values. Fourthly, "DeCompChoice" tries (up to four) LLHs which maximise f_1, f_2, f_3 and F and then chooses the one which yields the greatest improvement.

For a benchmark to test these against, they use some other methods for picking the LLH. "SimpleRandom" simply picks an LLH at random each iteration. "RandomDecent" which picks an LLH and applies it until no further improvements can be made, then chooses another one and repeats the process. "RandomPermDecent" is

similar to "RandomDecent" except the permutation of LLHs is already determined. "Greedy" simply evaluates the improvement for using each heuristic at each iteration and then chooses the best. Their experimental results show that all the hyper heuristics beat the normal heuristics and that DeCompChoice provides the best results of the tested high level heuristics with over a 9% improvement on the worst hyper heuristic. These methods have been applied to personnel scheduling (Cowling, Kendall and Soubeiga, 2001) (Cowling, Kendall and Soubeiga, 2002) with similar results.

2.5.3. Well known Meta-Heuristics as High Level Heuristics

A lot of work recently has been done which uses well known meta-heuristics to decide which low level heuristic to use or whether or not to accept a solution. For example, (Bai and Kendall, 2003) uses simulated annealing to determine whether or not to accept a new solution. At each iteration a new candidate solution is generated by applying a low level heuristic. The low level heuristic used is determined by the choice function and then simulated annealing is used to decide if we should keep the new solution. They apply the hyper heuristic to a shelf space allocation problem (an extension of the multi-knapsack problem (Yang, 2001)) and it performs better the other hyper heuristics they tested, however they did not benchmark it against ordinary meta-heuristics so a comparison cannot be made. More recently (Burke et al., 2008) provided a study of simulated annealing based hyperheuristics with different applications.

Özcan et al. (2009) compare the Choice Function Simulated Annealing hyper heuristic to a new acceptance method called "Late Acceptance". This deffers the acceptance of a local move until k steps after it has been made. This allows they hyperheuristic to see what long term effects the application of a low level heuristic has.

In all of the test cases Late Acceptance performed better except in two of the problems where the methods were statistically equivalent.

Ayob (2003) uses a Monte Carlo based hyperheuristic to solve a Component placement sequencing for multi head placement optimisation problem. Like the Simulated Annealing based hyper heuristic above, the choice function is used to determine which low level heuristic to use then the Monte Carlo algorithm is used to determine whether or not to accept the new solution. Again the benchmarks in this paper only test very similar or simple hyper-heuristics against theirs and this paper shows that theirs is the superior.

2.5.4. Genetic Algorithm based Hyper-heuristics

Genetic Algorithm based hyper-heuristic developed in (Cowling, Kendall, and Han, 2002) called hyper-GA evolves a chromosome (usually a small length of about 10-20 genes) which consists of a set of genes which are numbers which in turn related to which low level heuristic to apply. This order is evolved over generations using normal GA methods including mutation and a two-point crossover. The best sequence is applied at each generation and the sequence will evolve depending on the state of the problem. They apply their method to a trainer scheduling problem and compare its results with other methods including a mimetic heuristic (a GA/Local search hybrid heuristic) and show that it outperforms these other methods. Like with any other hyper-heuristic, this method can be applied to any problem given the right set of low level heuristics.

This is developed further to include a variable length chromosome which they call ALChypher-GA (Han, Kendall and Cowling, 2002). In this hyperheuristic the same principles as the hyper-GA apply however the length of the chromosome can change or

"adapt" to suit the problem in hand. The ALChypher-GA uses specially designed crossover and mutation operators to insert or remove groups of genes. They also implement a penalty function to penalise the length of the chromosome where the increased length of the chromosome increases the run times due to additional calculation and evaluation required by these longer chromosomes. This is defined by (Length of chromosome * CPU time to evaluate chromosome) / (Improvement in objective function).

As performance of individual genes of the chromosome can be evaluated, a clever crossover operator is used which swaps the best parts of each chromosome over. For example, if in parent one, genes 4-7 make the most improvement, and in parent 2 genes 8-10 make the most improvement, these sets of genes will be swapped. Here you can see how the length of the chromosome may change. The hyper-heuristic uses two new crossover operators, removing-worst and inserting-good. As the names suggest, these remove bad genes that make little or negative improvement, and insert good genes that make good improvements respectively.

From their experimental results we see that the chromosome length changes quite vastly in the first 25 generations while it is finding the optimal length. We can see that this method does work as with the two of problems they illustrate different average lengths of chromosome are settled upon. This can be advantageous when the problems are of different sizes or unknown a priori.

Guided Operators for the hyper-GA are introduced in (Han and Kendall, 2003) which aim to aid the hyper-GA in finding the optimal length for the chromosome and to do this more effectively and efficiently. There are two ways it does this; if the chromosome is longer than the average length of the chromosomes in the population, the worst genes are removed during mutation. If the average length is less than the

average, good genes will be inserted. Experimental results show that the guided adaptive length chromosome hyper-GA performs better than the adaptive length chromosome hyper-GA. This new method also decreases CPU time. In all these papers, no comparison is made to meta-heuristics.

The Tabu assisted hyper-GA (hyper-TGA) (Han and Kendall, 2003) improves the hyper-GA by using Tabu methods. The problem with the hyper-GA is that determining the good and bad genes is computationally expensive. Instead of removing bad or poorly performing genes, they are made Tabu for a certain period of time. The idea is to Tabu genes which do not affect the objective function. For example if a gene 4's low level heuristic was applied and no (positive) change was observed, gene 4 would be made Tabu for n generations. At each generation, when a gene is tabooed the low level it is skipped when applying the chromosome and its tenure (the remaining Tabu time) is decreased by 1. An optimal Tabu length of five was found experimentally. The pseudo code for the hyper-TGA is outlined here:

1. Generate an initial solution S (randomly or greedily)
2. Generate P initial chromosomes; initialise the Tabu tenure t_{jk} (j is the position in chromosome k) for each gene to 0. Store these chromosomes in a pool.
3. For each chromosome k (0<=k<=P)
    a. Apply the low-level heuristic to S according to the order of the chromosome when t_{jk}=0
    b. Record the new solution S_k
    c. Record the change in the objective function each low level heuristic makes C_j
    d. If t_{jk}>0 then $t_{jk} = t_{jk} -1$
    e. If C_j =0 then t_{jk} =n (n is the Tabu length)
4. Compare each S_k to S: if S_k>S then S= S_k
5. Apply crossover and mutation
6. Add the new chromosomes and the 10 best chromosomes from the current pool to a new pool. If the stopping criteria are not met, go to step 3.

From their experimental results and comparing it to those of their previous work, the hyper-TGA is found to be an improvement on the hyper-GA not only in computational time but also it produces better solutions. Once again, this method has not been compared with any meta-heuristics and so its performance cannot be compared.

2.5.5 Adaptive Memory Programming

Various works have analysed similarities and differences of optimisation heuristics. (Taillard et al., 2001) describes recent developments in some of these heuristics and present a unified approach called Adaptive Memory Programming (AMP). The method tries to unify meta-heuristics with memory, specifically Tabu search, Genetic Algorithms, Ant Colony Optimisation and Scatter Search. All these methods have the same characteristics:

1. A set of solutions or a special data structure that aggregates the particularities of the solutions produced by the search is memorized
2. A provisional solution is constructed using the data in memory
3. The provisional solution is improved using local search algorithms or a more sophisticated meta-heuristic.
4. The new solution is included in the memory or is used to update the data structure that memorizes the search history.

The paper then goes on to explain how several hybrid methods can be fit into the AMP methodology and in turn shows that metaheuristics are evolving toward a unified problem solving approach. It also notes advantages of the AMP methodology:

- AMP is highly distributable. As the local search takes the bulk of the time, this can be parallelised with each processor working on a different provisional solution.

- AMP is also able to work with dynamic problems as well as it is able to adapt to modified data.
- AMP can offer a set of high quality solutions for the user to choose from.

AMP has similarities with the hyperheuristic framework. Hyperheuristics, in theory, can be used to solve any problem given the right set of heuristics. An "AMP" method can be used to solve a problem give heuristics to construct provisional solutions and use local search on them. In fact, a hyperheuristic can be expressed in the AMP format. Step 2 would create a new solution using a low level heuristic chosen using the data in memory. Step 3 would be ignored (or incorporated into the low level heuristic)

Meta/hyper heuristics and other methods of using existing heuristics to create better one, seems to be a very interesting area of research. Object-Orientated Programming and Design revolutionised programming by allowing reuse of code. These methods are doing the same for optimisation.

2.5.6 No Free Lunch Theorem

The no free lunch theorem (Wolpert and Macready, 1997) is informally described as stating that "any two algorithms are equivalent when their performance is averaged across all possible problems" (Wolpert and Macready, 2005). That is to say any two algorithms, when applied to all problems, will perform equally as bad or good as each other irrespective of the evaluation method and hence applying a fixed algorithm to each a problem is worse than matching a tailored solution to the problem.

Since hyperheuristics use a tailored set of low level heuristics to solve a problem, it may be argued that they may perform better (given the right set of low level heuristics for the problem.) There has been a variety of results that have refined the no

free lunch theorem (see (Whitley and Watson, 2005) for a comprehensive review). Poli and Graff (2009) use the connection between function closure and the NFL being if-and-only-if from (Schumacher, Vose and Whitley, 2001) to prove that the no free lunch theorem does not automatically apply to hyperheuristics. They make two key conclusions: "... in practice, when a hyper-heuristic approach is applied to finding a solver for a specific problem, there can be a free lunch" and "... in practice, when a hyper-heuristic approach is applied to finding a solver for a not-too-large set of problems, there will likely be a free lunch".

2.6. Enhancement Methods

There are some methods in literature which work with an existing optimisation method to improve the performance. Most of the metaheuristics described so far, take a standard local search and modify it in a problem specific way to help it escape local minima. These methods usually require knowledge of the problem to make them effective however not all of them do. For example, Guided Local Search requires a user to define features of problems which requires a good understanding of the problem. Sawing EAs on the other hand modify the penalty/fitness function which has the advantage that you don't need to understand the problem, however, this can only enhance an existing EA.

Forward-Backward Improvement (FBI) is a problem specific improvement algorithm. The idea was developed by (Valls, Ballestin and Quintanilla, 2003) and (Tormos and Lova, 2001) independently. Valls calls it (double) justification and Tormos and Lova call their method a forward backward improvement (FBI) pass. Valls et al. method is simple and they demonstrate that it improves the results of 15 out of the 22 state of the art heuristics they tested (Valls, Ballestin and Quintanilla, 2003). An active

schedule (see (Sprecher, Kolisch and Drexl, 1995) for definitions of active schedules) is justified to the right. That is, all activities, starting with the latest end times, are scheduled as late as possible, without violating resource, precedence or time constraints. Similarly the schedule is left justified, in which all activities, starting with the earliest start times, are scheduled as early as possible, without breaking and resource, precedence or time constraints. This results in what Valls calls a double justified schedule.

In (Tormos and Lova, 2003) they only apply FBI if the fitness is already better than the average fitness of the population before FBI. This helps prevent processing that is most likely unnecessary. In (Tormos and Lova, 2003) the process is refined further so that a backward-forward pass can be applied as well as a forward-backward pass. This can also be applied more than once depending on the quality of the schedule.

Forward-backward improvement can be applied to any solution and it will never make a solution worse (in terms of FBI's objective which is to minimise make span). This makes it ideal for using in combination with another method.

2.7. Summary

This chapter reviewed literature relating to the Resource Constrained Project Scheduling problem and its variants as it is a well studied problem and the most relevant to the scheduling problem we will study. Unfortunately it is less complex than the problem we study so many of the solution methods will not be directly applicable nor results comparable.

Different methods for evaluating solutions were shown and some methods which manipulate the evaluation method to better solve the problem.

The solution methods reviewed fell into 4 categories: Local Search Based, Population Based, Hyper-heuristics and Problem Specific. Local Search based approaches in general are fast, working on a single solution and quickly evaluating small changes to it in an attempt to find a change which will make an improvement. When local optima occur the search has to find a way out and this is where many of the solution methods differ.

Population based approaches do not generally have this problem as they keep a set of solutions which influence each other. These solutions try to keep diverse sets which aim to help other individuals to escape from local minima, whilst smaller mutations try and make the individuals fitter.

Hyperheuristics aim to abstract the problem specific information from the solution method. In theory, a Hyperheuristic could solve any problem given the right set of Low Level Heuristics to work on the problem. It is the hyperheuristics job to work out which Low Level Heuristic to use at a given point in time. These again can be classified into search based and population based approaches.

Finally a problem specific method was reviewed which exploits problem specific ideas to improve the solution.

Chapter 3

Problem Description

The scheduling problem we consider is complex and “messy”. It contains many of the features of scheduling problems in literature and thus has many of them as sub-problems. This model was created in collaboration with Trimble MRM Ltd to precisely model aspects of scheduling problems seen in the real world. It provides us with a hard to solve multi-objective problem which when simplified resembles many problems in literature.

It is split into two parts: the static part, which is concerned with build schedules meeting certain criteria, and the dynamic part, which involves responding to events that happen during the course of a schedule.

3.1. Static Scheduling Problem

The workforce scheduling problem that we consider consists of four main components: Tasks, Resources, Skills and Locations. A *task* T_i is a job or part of a job that needs to be completed. Each task must start and end at a specified location. Usually the start and end locations are the same but they may be different. Each task has one or more time windows. Some time windows which are an inconvenience for the customer have an associated penalty. We have a set $\{T_1,T_2,...,T_n\}$ of tasks to be completed. Each task is undertaken by one or more resources. We have set of resources $\{R_1,R_2,...,R_m\}$. A task requires resources with the appropriate skills. We have a set $\{S_1,S_2,...,S_k\}$ of skills. Task T_i requires skills $[\mathrm{TS}_1^i,TS_2^i,...,TS_{t(i)}^i]$ with work requirements $[w_1^i,w_2^i,...,w_{t(i)}^i]$where w_q^i is the amount of skill TS_q^i required. Task T_i also has an associated priority $p(T_i)$. Resources are the components that undertake the work and possess skills. Resource R_j possesses skills $[\mathrm{RS}_1^j,RS_2^j,...,RS_{r(j)}^j]$. A function $c(R,S)$ expresses the competence of resource R at skill S, relative to an average competency. Each resource R travels from location to location at speed $v(R)$. For tasks T_1, T_2, $d(T_1, T_2)$ measures the distance between the end location of T_1 and the start location of T_2. There are three main groups of hard constraints: task constraints, resource constraints and location constraints and they are described below.

3.1.1. Task constraints

- Each task can be worked on only within specified time windows.
- Some tasks require other tasks to have been completed before they can begin (*precedence* constraints).
- Some tasks require other tasks to be started at the same time (*assist* constraints).

- Tasks may be split across breaks within a working day. No tasks may take more than one day.
- For a task to be scheduled it must have exactly one resource assigned to it for each of the skills it requires.
- All assigned resources have to be present at a task for its whole duration regardless of their skill competency and task skill work requirement.
- If a task T_i with skill requirements $[\mathrm{TS}_1^i, TS_2^i, ..., TS_{t(i)}^i]$ and amounts $[w_1^i, w_2^i, ..., w_{t(i)}^i]$ is carried out by resources $[\mathrm{R}_1^i, R_2^i, ..., R_{t(i)}^i]$ then the time taken is

$$\max_{q \in \{1,2,...t(i)\}} \left(\frac{w_q^i}{c(R_q^i, TS_q^i)} \right)$$

i.e. the greatest time taken for any single resource to complete a skill requirement

3.1.2. Resource constraints

- A resource *R* travels from location to location at a fixed speed *v(R)*.
- Resources may only work during specified time windows.
- Resources can only work on one task at once and only apply one skill at a time.

3.1.3. Location constraints

- Resources must travel to the location of each task they work on, and are unavailable during this travel time.
- Resources must start and end each day at a specified “home” location and must have sufficient time to travel to and from their home location at the start and end of each day.

3.1.4. Objectives

When building a schedule many different and often contradictory business objectives are possible. In this thesis we consider three objectives. The first objective is Schedule Priority (*SP*), given by

$$SP = \sum_{\{i:T_i \text{ is scheduled}\}} p(T_i)$$

Maximising Schedule Priority maximises the value of the tasks scheduled (and implicitly minimises the value of tasks unscheduled).

The second objective measures Travel Time (*TT*) across all resources. Define *A={(i1,i2,j):*task T_{i1} comes immediately before T_{i2} in the schedule of resource R_j*}*. Then,

$$TT = \sum_{(i1,i2,j)\in A} \frac{d(T_{i1}, T_{i2})}{v(R_j)}$$

Travel to and from home locations is handled by considering dummy tasks fixed at the start and end of the working day, at the home location of each resource.

The third objective measures the inconvenience associated with completing tasks or using resources at an inconvenient time, which we have labelled Schedule Cost (*SC*). In order to express this accurately we express the time windows for Resource *R* using a function *τ* where *τ(R,t)* is the cost per unit time for resource *R* working at time *t*. We introduce a variable

$$\mathrm{X}(R,t) = \begin{cases} 1 & \text{if resource } R \text{ is busy (on a task or traveling) at time } t \\ 0 & \text{otherwise} \end{cases}$$

Similarly we introduce *τ'* where *τ'(T,t)* is the cost per unit time for task *T* being executed at time *t* and

$$\mathrm{X}'(T,t) = \begin{cases} 1 & \text{if task } T \text{ is being executed at time } t \\ 0 & \text{otherwise} \end{cases}$$

Then

$$SC = \sum_{i=1}^{n} \int_{t} X'(T_i,t)\tau'(T_i,t)dt + \sum_{j=1}^{m} \int_{t} X(R_j,t)\tau(R_j,t)dt$$

Other objectives are possible but these three objectives express most of the primary concerns of the users in this case, at a high level. Considering lower level objectives at regional and resource group level could, however, give in many more objectives for this problem.

3.2. Dynamic Repair Problem

This problem is "messy" (and realistic) as it contains dynamic events which disrupt the pre-planned schedule during its execution. In this work the dynamic events are stochastically generated, following a detailed study of the nature of the events that occur in practice, and simulation is used to pass the new problem information on to a rescheduling / schedule repair mechanism. *s(Ti)* gives the start time of a task *Ti* in the "static" predictive schedule generated before consideration of dynamic events and *e(Ti)* gives the start time of a task *Ti* at the end of the schedule's execution (after all schedule repair events have occurred). If a task is not scheduled the time representing the end of the schedule is used for both *s(Ti)* and *e(Ti)*. The four types of dynamic events considered in this work are perturbation of task time, task overruns, travel overruns and task on-site cancellations.

The perturbation of task time event models the fact that the forecast duration of each task is incorrect by some small amount (generated by a normal distribution with a mean of its current duration and a standard deviation of a few minutes). When a task's duration is to increase this event occurs when the task reaches the end of its planned

duration, if the task's duration is to decrease the event occurs when the task reaches its new (earlier) end time. The overrun event (for both travel and tasks) causes significant change in the travel or task duration (increase or decrease), and the on-site cancellation event is used to model a real-world phenomenon whereby an engineer (resource) arrives at a task location and discovers that the task cannot be completed for some reason (often due to no access). In this case the task ends after ten minutes and all resources pre-planned for work on that task are free to do other work.

3.2.1. Dynamic Objectives

Schedule stability (ΔST) is a standard objective and is measured by considering absolute change in start times of all tasks (Cowling and Johansson, 2002):

$$\Delta ST = \sum_{i=1}^{n} |s(T_i) - e(T_i)|$$

The equation above represents the absolute change in start times for all tasks, we wish to minimise this value.

As with the static aspects of the problem several other objectives are possible however we chose to include only change in start times as we believe it is of most concern to both businesses and customers. Other possible objectives studied for other problems are discussed in literature. Vieira (Vieira, Herrmann and Lin, 2003) discusses several stability / robustness measures including absolute change in start times as well as schedule nervousness which measures how likely things are to change in the future. Nervousness is not relevant here since tasks are communicated one by one to resources in the field. Considering these and other lower level objectives at regional and resource group level is realistic and useful and would give many more objectives for this problem.

3.3. Summary

The problem we study has been formed with our industrial sponsor and shares many of the features of literature reviewed in the previous chapter, however it encompasses more features than any of the single problems. As such, certain parts of the problem can be excluded to reduce our problem into those studied in literature.

Generating test instance for a problem is done using the problem generator of (Cowling et al., 2006), which is explained in more detail in (Colledge, 2009). This provided us with an easy way to produce problem instances with specific parameters. Generating problems by hand of the size required to involve all complexities of the problem would be impossible.

The problem was formulated with Nicolas Colledge and is also published in his thesis (Colledge, 2009). Both our works use this problem as a case study however the solution methods and the research areas investigated are different. Colledge investigates the effects of multi-objective algorithms on diversity and quality of solutions. He also develops hyperheuristic solutions to the dynamic aspect of the problem including the low level heuristics used later in my thesis. However, since his work is multi-objective and he is interested in studying a diverse set of pareto optimal solutions, no direct comparison between the methods he uses and the results obtained in this thesis can be made.

Chapter 4

Search in Scheduling

Our initial work looked at local search with respect to scheduling. Initially we took schedules produced by a genetic algorithm and tried to improve them using exact search on small sub problems. It was clear the schedules produced by the genetic algorithm were not optimal and so we went on to produce constructive scheduling heuristics from scratch. This chapter details this process and the results and conclusions found.

4.1. Improving GA Generated Schedules with Exact Search

The Genetic Algorithm of our initial work (Cowling et al., 2006) for the Workforce Scheduling Problem we study, was quick and provided good results. The reason it is quick is that it uses a naive method of inserting tasks.

The chromosome represents the order of tasks to be scheduled, like an Activity List for the RCPSP. However, tasks in the Workforce Scheduling problem may require more than one resource and rarely are there homogenous resources, so selecting the appropriate resources is not as straight forward. The genetic algorithm uses a simple method for selecting resources. For each skill required by the resource it draws up a list of resources that possess the skill. It then calculates for each resource the available time multiplied by the resource's skill competency and chooses the resources with the highest availability. In effect, this weights those that are good at the skill higher. The GA then finds the time the selected resources have in common and insert the task as early as possible. This provides good results in a reasonable amount of time.

We then took this idea of enumerating resources further to see how good the GA was at choosing resources. We took schedules generated by the genetic algorithm and tried to improve them, similar to how a mimetic algorithm does, although we only use local search after the evolution has stopped. To improve them, we went through each scheduled task and tried to assign different resources to it with the aim of improving the quality of the schedule. Observed results showed that there was plenty of room for improvements in the genetic algorithm, but more likely that a constructive local search would be a better place to start.

4.2. Constructing Schedules using Local Search

In this section, a hyperheuristic framework is developed for the workforce scheduling problem and is used in many subsequent chapters and publications (Remde et al., 2007) (Dahal et al., 2008).

Most meta-heuristics have to be tailored to a specific problem and are problem specific or require a lot of knowledge and this can often be expensive and time consuming especially when trying to find the right parameters for these meta-heuristics. Hyper-heuristics raise the level of generality of a heuristic which allows it to be applied to many different optimisation tasks with little modification.

For any hyper-heuristic, low level heuristics (LLHs) need to be developed to be used by the hyper-heuristic. The first LLH we created simply inserted a task at the most optimal time into a schedule using selected resources (low level heuristics would be made from this by having lots of different ways to select a task to be scheduled, and lots of ways of choosing the resources to assign to the task). To test this insertion heuristic, we will use an exhaustive search to calculate the change in fitness observed after scheduling a task, for all combinations of resources. Once the best assignment of resources has been found the task will be scheduled using those resources. This exhaustive search will then be used to schedule as many tasks as possible.

```
current, test, best : Solution


while (Some tasks can be scheduled)
        for each Unscheduled Task T
                for each Resource Combination R
                        test:=Insert T into current using the resources R
                        if (test is fitter than best) then best:=test
                end for
                current:=best
        end for
end while
```

Figure 4.1. Local Search Heuristic

4.2.1. Experimental Results

The problem instance has 1000 tasks, 200 resources and 20 skills. Four methods were used to solve the problem. The first was a Genetic Algorithm (similar to the one used in (Cowling et al., 2006)) which evolves an order of tasks to be scheduled (using good parameters from the paper with crossover rate 5%, population size 50 and 100 generations). The other 3 are variations of the exhaustive local search starting from an empty schedule and use variations of the heuristic in Figure 4.1. "LS one pass 2 skills" tries to insert all tasks with a maximum of 2 skills once and stops. "LS multi pass 2 skills", tries to insert all tasks with a maximum of 2 skills and keeps trying until no more tasks can be inserted into the schedule. "LS multi pass 3 skills", tries to insert all tasks with a maximum of 3 skills and keeps trying until no more tasks can be inserted into the schedule. The local search is limited to scheduling tasks with 3 skills as exhaustively searching 4 (or more) skills would take too long. The fitness function that

will be used is *Fitness=SP-4SC-2TT* (as described in (Cowling et al., 2006)), where *SP* is the sum of the priorities of all the scheduled tasks, *SC* is sum of all the costs associated with scheduled tasks and *TT* is the total travel time on the schedule. The results of a single run are shown in Table 4.1 and Figure 4.1 shows a graph of the results.

Table 4.1. Test Results and Approximate CPU Time Used

Method	FITNESS	SP	SC	TT	CPU TIME
Genetic Algorithm	10891	17357	1528	177	8 h
LS one pass 2 skill	15145	20663	1344	71	2 m
LS multi pass 2 skill	19055	26313	1766	97	4 m
LS multi pass 3 skill	19308	30356	2703	118	9 m

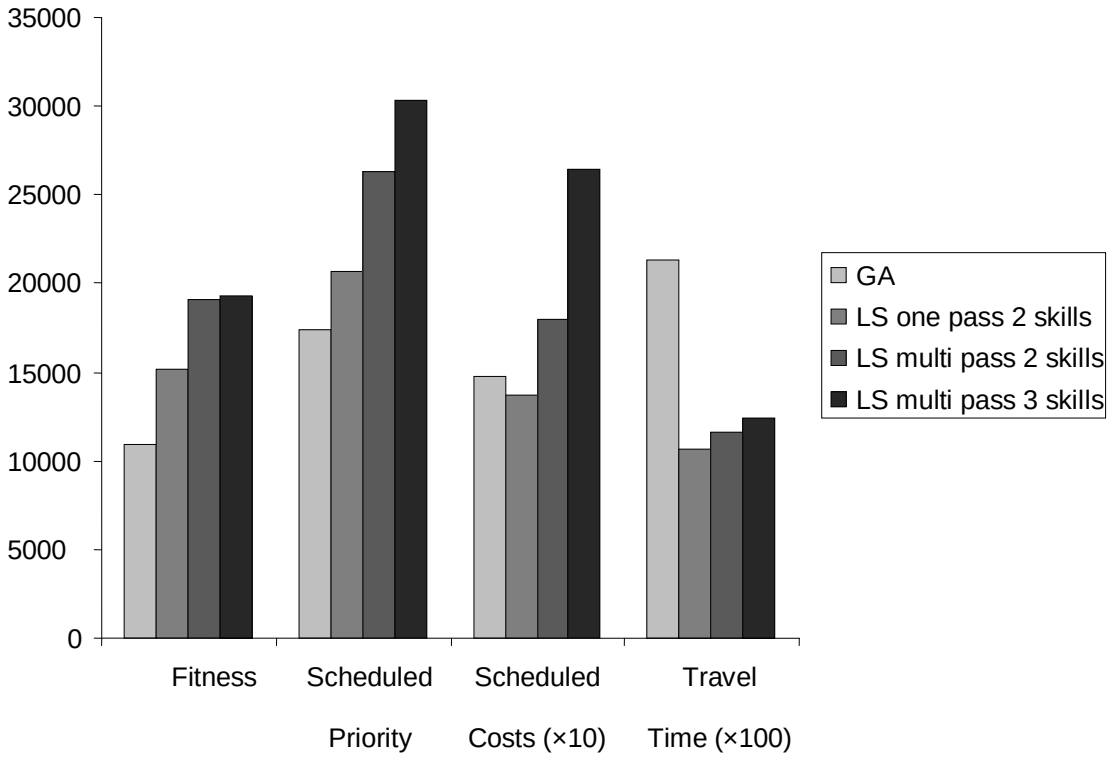


Figure 4.2. Graph of results from Table 4.1

4.2.2. Analysis

From the results we can clearly see that the schedules produced from the local search are far superior to those of the GA and were produced in far less time. The results show that there is little change between “LS multi pass 2 skills” and “LS multi pass 3 skills”. This may be due to the fact that priority values of tasks are not proportionate to the number of resources needed (this means inserting tasks which require fewer resources generally leads to greater increases in fitness than inserting those requiring more resources). The difference between “LS multi pass 2 skills” and “LS one pass 2 skills” indicates that multiple passes are beneficial. This is because after one pass, improvements will have been made (freeing up resources including time) and so further improvements can be made in later passes.

Even though there is no statistical significance measure, the results suggest the local search techniques are much more favourable than the GA. The GA uses a very simple heuristic to insert tasks which often makes less than optimal decisions. This quick approach is needed because of the large number of times it is done when evaluating an individual.

4.3. Experimental Framework

In the previous section a modest amount of CPU time was required to obtain the results. To get statistically meaningful results heuristics must be run multiple times on multiple problem instances and an average taken. These individual runs are isolated and can be executed in parallel on multiple machines. Henceforward a parallel framework for running multiple experiments will be designed and implemented and used throughout the rest of the thesis. Similarly when parameter tuning the optimal value for

multiple different parameters needs to be determined. We often end up with a scenario shown in Figure 4.3. This code parallelises very well as each run is independent of the others.

```
for (run=1..10)
  for (a in A)
    for (b in B)
      for (c in C)
        Solve(run,a,b,c)
```

Figure 4.3. Pseudo code for parameter tuning

To run this code on multiple machines a method of dealing out work to each machine is required. Without writing complex network code, this can be achieved by using a shared file system. For this method we map each set of parameters onto a filename. Since file systems are designed for allowing exclusive access to files it is easy to use this to deal out the work. Figure 4.4 shows how we determine if we should work on a file and Figure 4.5 shows how this method can be used to parallelise the pseudo code in Figure 4.3.

The implementation and presentation on this framework has been made freely available at (http://inf.brad.ac.uk/~smremde/parallel).

```
ShouldIWorkOn(file)
  if (FileExists(file))
    if (FileIsReadOnly(file))
      return false
    else if (Lock(file))
      return true
    else
      return false
    end if
  else if (CreateAndLock(file))
    return true
  else
    return false
   end if
```

Figure 4.4. Determining if the computer should work on a specific file.

```
for (run=1..10)
  for (a in A)
    for (b in B)
      for (c in C)
        file = MakeFilename(run,a,b,c)
        if (ShouldIWorkOn(file))
          Solve(run,a,b,c)
          MarkAsReadOnly(file)
          Unlock(file)
        end if
```

Figure 4.5. Parallelising Figure 4.3

4.3.1. Speed-Ups

Although no numerical data on the speed ups can be given it can be estimated and approximations made from experience. When the part being parallelised takes a decent amount of time (for example most of the experiments in this chapter took over 20 minutes) then the speed up is close to linear. A good approximation can be calculated using:

$$Single\ Total\ Time = n\bar{x}$$

$$Parallelized\ Total\ Time = n\bar{x} + cn\delta$$

$$Speed\ up = \frac{Single\ Total\ Time}{Parallel\ Total\ Time\ \cdot c^{-1}} = \frac{n\bar{x}c}{n\bar{x} + cn\delta} = \frac{\bar{x}c}{\bar{x} + c\delta}$$

Where δ is the time it takes to check is an experiment needs doing, n is the number of experiments to be done, $\bar{x}$ is the average time required to do an experiment and c is the number of computers to be parallelised on. When δ is very small compared to x (in our case a fraction of a second compared to 30-60 minutes of experiments), δ becomes insignificant:

$$\lim_{\delta \to 0} Speed\ up = c$$

4.4. Using Exact Search in Local Moves

In this section we propose a method of splitting the complex real-world workforce scheduling problem into smaller parts and solving each part using exhaustive search. These smaller parts comprise a combination of choosing a method to select a task to be scheduled and a method to allocate resources, including time, to the selected task. We use reduced Variable Neighbourhood Search (rVNS) and hyperheuristic approaches to decide which sub problems to tackle. The resulting methods are compared to local search and Genetic Algorithm approaches. Parallelisation is used to perform nearly one CPU-year of experiments. The results show that the new methods can produce results fitter than the Genetic Algorithm in less time and that they are far superior to any of their component techniques. The method used to split up the problem is generalisable and could be applied to a wide range of optimisation problems.

4.4.1. Introduction

The fitness of a schedule used here is given by one of the single weighted objective functions used in (Cowling et al., 2006) and previously, $f = SP - 4SC - 2TT$, where SP is the sum of the priority of scheduled tasks, SC is the sum of the time window costs in the schedule (both resource and task) and TT is the total amount of travel time. This objective is to maximise the total priority of tasks scheduled while minimising travel time and cost. In this section we will compare the Genetic Algorithm method with a new reduced Variable Neighbourhood Search and hyperheuristic methods.

We propose a method to break down this “messy” problem by splitting it into smaller parts and solving each part using exact enumerative approaches. Hence each part consists of finding the optimal member of a local search neighbourhood. We then

design ways to decide which part to tackle at each stage in the solution process. These smaller parts are the combination of a method to select a task and a method to select resources for the task. We will take these smaller parts and use reduced Variable Neighbourhood Search and hyperheuristics to decide the order in which to solve them.

4.4.3. Heuristic Solution Methods

Our proposed framework splits the problem into (1) selecting a task to be scheduled and (2) selecting potential resources for that task. A task is randomly chosen from the top two tasks which we have not tried to schedule ranked by the task order, to make the search stochastic, to ensure that running it multiple times will produce different results. We have implemented 8 task selection methods given in table 4.2. Note that some of our task orders are deliberately counterintuitive to give us a basis for comparison.

PriorityDesc, *PriOverReq*, *PriOverMaxReq* and *PriOverAvgReq* are attempts to identify the tasks which will give us the most reward and schedule them first. They estimate the task duration differently and use this estimate to calculate priority per hour. *PrecedenceDesc* attempts to schedule those tasks with the largest number of succeeding tasks first. *PrecedenceAsc*, *PriorityAsc* and *Random* give us some indication of the effect of task orders since intuition would suggest that they should give poor results.

We then define Resource Selectors which select a set of potential resources for each skill required by the selected task. The Resource Selectors first sort the resources by their competencies at the skill required and then select a subset of them. This could be, for example, the top five or the top six to ten etc. The subsets of resources are then enumerated and exhaustive search used to find the insertion which will yield the lowest time window and travel penalties subject to precedence constraints. Figure 4.6 illustrates this.

Table 4.2. Task sorting methods

Method	Description
Random	Tasks are ordered at random.
PriorityDesc	Tasks are ordered by their priority in descending order
PriorityAsc	Tasks are ordered by their priority in ascending order
PrecedenceAsc	Tasks are ordered by their number of precedences ascending
PrecedenceDesc	Tasks are ordered by their number of precedences descending
PriOverReq	Tasks are ordered by their estimated priority per hour assuming the task will take as long as the total skill requirement
PriOverMaxReq	Tasks are ordered by their estimated priority per hour assuming the task will take as long as the maximum skill requirement
PriOverAvgReq	Tasks are ordered by their estimated priority per hour assuming the task will take as long as the average skill requirement

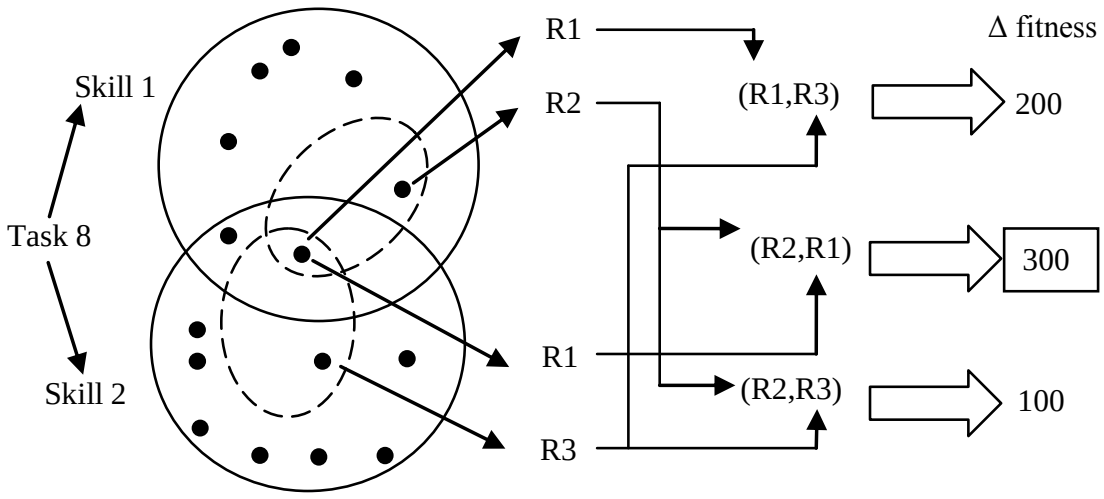


Figure 4.6. Resource Selector. The dotted subset of resources possessing the required skill is chosen by a Resource Selector. The assignment (R2, R1) is chosen as the best insertion.

The neighborhoods of our rVNS insert tasks selected by a task order using a given resource selector. If an insertion is not possible, because of resource or task constraints, we try the next resource selector and so on. We consider several sequences of resource selection neighborhoods, or “chains”, as shown in Figure 4.7.

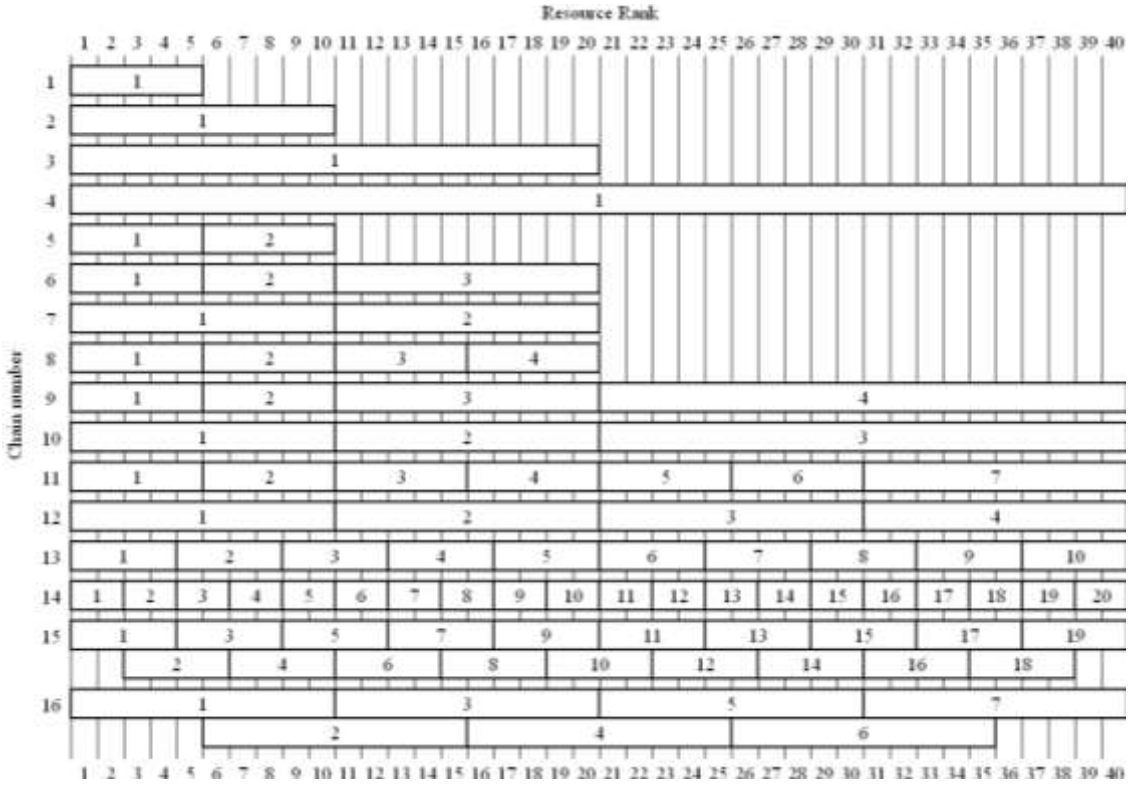


Figure 4.7. Resource selection "chains" for the rVNS. For example, chain number 6 implies we will consider the top 5 resources first, then the top 6-10, then the top 11-20.

```
k is the index of the resource selector in use
(N1, N2, … Nkmax ) is our chain of resource selectors
Sort tasks using the chosen task order
k:=1
while (k<kmax)
        for each Unscheduled Task T
                Select Sets of Resources Using Nk for Task T
                Exhaustively Search the selected sets of resources to find the optimal
insertion I which maximises the change in fitness.
                Insert task T into the schedule using I
        next
        if some tasks were inserted then k:=1
        else k:=k+1
        end if
end while
```

Figure 4.8. Pseudo code for our rVNS method.

These neighbourhoods show a progression of increasingly larger ranges and smaller splits. Figure 4.8 shows the pseudo code for our rVNS method. Allowing search to restart at the start of the chain allows the search to retry insertions that may have

failed before because of resource or task constraints. With the 16 resource selection chains and the 8 task orders we have defined, we have 128 different rVNS methods.

Our first hyperheuristic, *HyperRandom*, selects at random a Low Level Heuristic (i.e. a (task order, resource selector) pair) to use at each iteration and applies it if its application will result in a positive improvement. This continues until no improvement has been found for a certain number of iterations. The second, *HyperGreedy*, evaluates all the Low Level Heuristics at each iteration and applies the best if it makes an improvement. This continues until no improvement is found. The low level heuristics are the combination of a task selector and a resource selector.

The genetic algorithm we will use is that of (Cowling et al., 2006). The chromosome represents an order of tasks to be scheduled by a serial scheduler. The initial population is generated randomly and the task order is evolved. The way in which the tasks are inserted into the schedule is a fast naïve approach as schedule must be generated many times per generation. The serial scheduler takes the next task from the chromosome and allocates resources to it greedily skill by skill. A resource is selected by finding the resource which has the greatest amount of available time in common with the task's time windows and any other resources already selected. After each skill has been allocated a resource, it is inserted into the schedule as early as possible. We use a population size of 50, mutation rate of 1%, and a crossover rate of 25% using Uniform Crossover. The GA is run for 100 generations (or for a maximum of 2.5 hours) and the result is the fittest individual in the final population.

4.4.4. Computational Experiments

To compare the methods for solving the problem, we use each method (one Genetic Algorithm, 128 rVNS and two hyperheuristics) on five different problem instances. The

five new problem instances require the scheduling of 400 tasks using 100 resources over one day using five different skills. Tasks require between one and three skills and resources possess between one and five skills. The problems are made to reflect realistic problems Trimble have identified and are generated using the problem generator used in (Cowling et al., 2006).

Each method is used for five runs of the five instances and an average taken of the 25 results. Five runs were chosen to give some statistical significance within a reasonable amount of time. To ensure fairness, each method is also run for a 2.5 hour "long-run" where the 25 results are repeatedly generated and the best average over all their repeated runs is reported. As these experiments require nearly a CPU year to complete (five runs of five instances using 131 different methods lasting 2.5 hours each = 8187.5 CPU hours) they were run in parallel on 60 identical 3.0 GHz Pentium 4 machines. Implementation was in C# .NET under Windows XP.

Figure 4.9 shows the results of the 2.5 hour "long run" for each rVNS approach. Results for a single run of each approach were 1-4% worse on average. The intuitively "bad" task orders, *PriorityAsc* and *Random* are clearly shown to be worse than the intuitively reasonable orders such as *PriorityDesc*. It would also appear that the good chain is more important than a good order. Measures based on decreasing priority or priority per hour (*PriorityDesc*, *PriOverReq*, *PriOverAvgReq*, *PriOverMaxReq*) are superior to other measures. Figure 4.10 compares the best approaches in detail. Chain 12 produces the best results for all task orders. It is clear to see the correlation between results with common chains or task orders. Chain 4 demonstrates that trying to estimate priority per hour is superior to *PriorityDesc*. This is probably because with a limited amount of free time in the schedule, using tasks that have lower priority but can be completed in a shorter time is more beneficial.

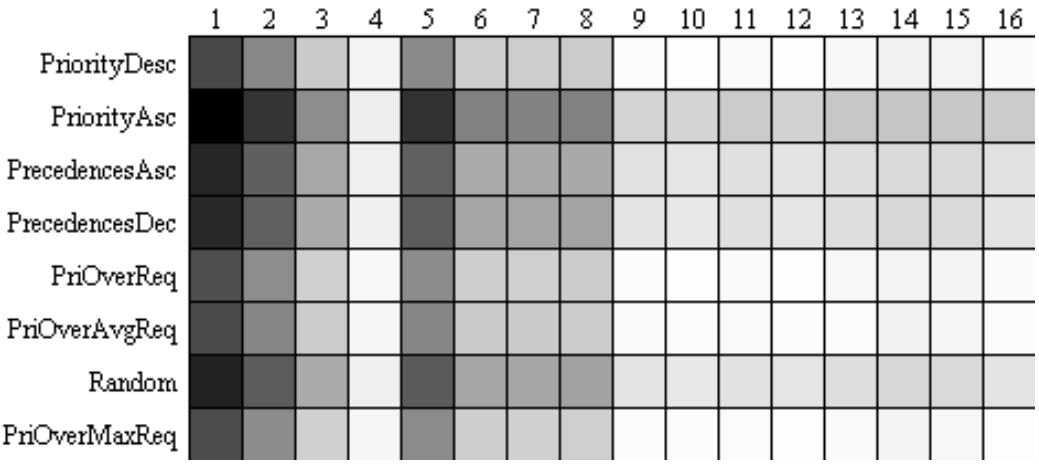


Figure 4.9. Heat graph of the performance of rVNS methods for 2.5 hour "long run". Black = 4472, White =26525

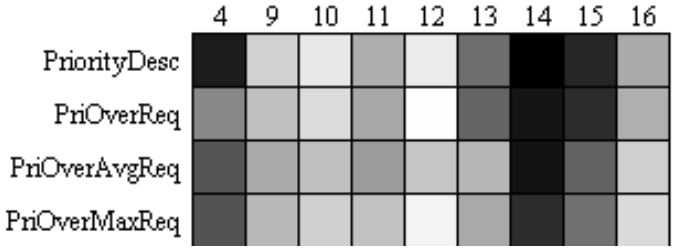


Figure 4.10. Heat graph of the performance of selected rVNS methods for 2.5 hour "long run". Black = 25398, White =26525

Figure 4.11 compares the CPU time for a single run of rVNS using each chain. It is clear that the approach would scale to very large problems using small resource selection sets such as for chains 1, 5, 6, 8, 11 and 13. Moreover, it appears that little solution quality is lost when covering the resources with small subsets rather than larger ones as in chain 4, but the CPU times are significantly reduced. Chain 12 yields the best results of the chains which take reasonable amounts of CPU time, and clearly outperform chain 2 and chain 7 which do not consider the whole set of resources. It seems that resources of poor competence must be considered to get the best possible results.

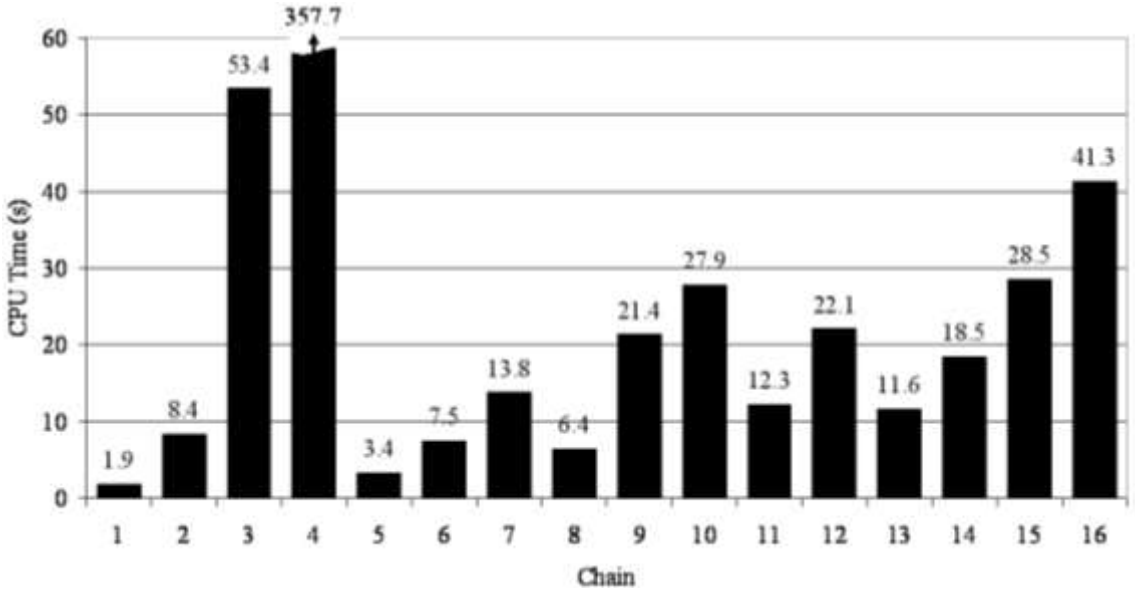


Figure 4.11. Average CPU time taken by each chain used in the rVNS methods. Average of 25 runs.

Table 4.3 shows the best result from the rVNS (Chain 12, Task Order *PriOverReq*) compared with GA and the hyperheuristic methods. They quite clearly show that *HyperGreedy* provided the fittest results on average while using more CPU time. The GA provided the worst result and in the slowest time. This may result from its insertion heuristic, however implementing a better one would make it even slower. The rVNS is the fastest method we have tested and provides results nearly 20% better than the GA in less than 1/350 of the CPU time required. Exactly solving small sub problems appears very effective in this case.

Table 4.3. GA, rVNS and Hyper-Heuristic Results for one run and long run (average of 25 runs).

Method	Fitness (single run average)	CPU Time (s)	Fitness (after 2.5 hours)
GA	21401.3	9000.0	21401.3
rVNS (Best)	25662.5	25.1	26215.1
HyperRandom	24525.4	78.3	25645.4
HyperGreedy	26523.6	419.2	27103.1

HyperRandom performs poorly compared to the best rVNS method. rVNS task selectors and resource selectors are s ensible guesses which significantly improve on the random approach of *HyperRandom*. The resource selectors of the rVNS tend to select resources which are of similar competence, so that a high competence resource is not combined with a low-competence resource (which might tie up the time of a high-competence resource).

The *HyperRandom*, and the *HyperGreedy* heuristics try significant numbers of bad low level heuristics which make local improvements which in the long run are far from optimal. In the case of the *HyperGreedy* method, the bad low level heuristics are evaluated every iteration which wastes CPU time. Analysis of the low level heuristics used in the *HyperGreedy* method was performed and show that 19 (26.4%) of the low level heuristics were never used and 56 (77.7%) of the low level heuristics were used less than one percent of the time. Figure 4.12 analyses the low level heuristics (LLHs) used. It shows the top 20 LLHs used together with when they are used in schedule generation. First third, middle third and last third show the usage at different stages in the scheduling process – from when the schedule is empty and unconstrained to when the schedule is almost full and inserting a task is more difficult. From these results it is clear that different LLHs contribute at different stages of the solution process, and that many different LLHs provide a contribution. For example, LLH 32 is more effective at the start of scheduling, LLH 12 is more effective in the middle and LLH 64 is more effective at the end. Without access to a large number of LLHs it seems that solution quality would be much reduced.

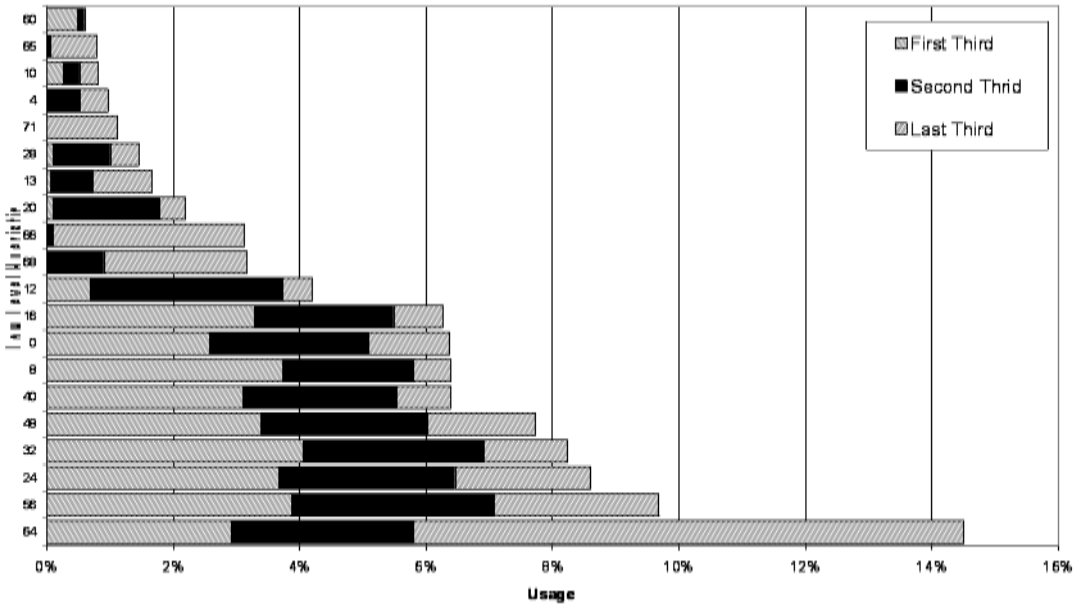


Figure 4.12. Usage of low level heuristics throughout the *HyperGreedy* search

4.4.5. Analysis

In this section we have compared a large number (128) of reduced Variable Neighbourhood Search (rVNS) approaches to hyperheuristics and Genetic Algorithm approaches for workforce scheduling problem. We have demonstrated the effectiveness of heuristic/exact hybrids which find optimal sub problem solutions using an enumerative approach. Our rVNS method can produce good results to large problems in low CPU time. Our hyperheuristics produce even better results using more CPU time and we showed that the hyperheuristic uses a range of low level heuristics throughout the search process.

The hyperheuristics we used are simple and learning could potentially decrease CPU time and increase fitness. In the next section we intend to implement a learning mechanism. We have seen from the analysis that many low level heuristics were never used and some used mainly at the end or beginning. Learning the low level heuristics behaviour could potentially lead to better solutions in less time.

4.5. Binary Exponential Back Off

Hyperheuristics (Chakhlevitch and Cowling, 2008) (Burke, 2003) reflect problem knowledge using a number of (usually simple) low level heuristics and objective measure(s). The hyperheuristic uses information about the performance of each low level heuristic (CPU time and objective measures) to determine which low level heuristics to apply at each decision point. The hypothesis of the hyperheuristic method is that some combination of these low level heuristics will prove effective in escaping any poor quality local optimum/basin of attraction (Chakhlevitch and Cowling, 2008). However, the method which decides which low level heuristics to choose need not be problem specific, and hence a single hyperheuristic method can work generally across many problem models and instances. In some cases low level heuristics are parameterised, or composed by "multiplying" together components (Remde et al., 2007) (Chakhlevitch and Cowling, 2005), which can give rise to hundreds or even thousands of heuristics (Cowling and Chakhlevitch, 2003). In such a case, deciding in reasonable time which heuristics to use may be difficult. However, there is evidence that having such a rich selection of low level heuristics may yield better results for complex problems in the long run (Chakhlevitch and Cowling, 2005).

Using the hyperheuristic framework introduced earlier, we create more low level heuristics and implement a new Tabu based hyperheuristic with dynamic Tabu tenures designed with large neighbourhoods in mind. *HyperRandom* and *HyperGreedy* heuristics try significant numbers of bad low level heuristics and hence waste CPU time, which may be highly significant for complex instances requiring CPU-hours or -days to solve. Analysis of the low level heuristics used in the *HyperGreedy* method showed that 26.4% of the low level heuristics were never used and 51.2% of the low level heuristics were used less than one percent of the time. An approach which more

aggressively prunes poor low level heuristics could result in large CPU savings in this case, with little or no impact on solution quality. This impact would increase with increasing numbers of low level heuristics.

The method we propose is a Tabu based hyperheuristic with dynamically adapting Tabu tenures designed for very large neighbourhoods. In a variety of computer networks, binary exponential back off or truncated binary exponential back-off is a randomized protocol for regulating transmission on a multiple access broadcast channel (Metcalfe and Boggs, 1976). This algorithm is used to spread out repeated retransmissions of the same block of data and to increase overall efficiency. Data needs to be retransmitted when a collision occurs. This happens when two (or more) computers try to transmit information on the same medium (a wire, a wireless frequency, etc) at the same time. When a computer transmits information, it also "listens" to see whether received information is what has been transmitted. If it detects anomalies it assumes that there is interference (probably from another computer trying to transmit at the same time). When a collision occurs, the computer will increase its *backoff* value by 1, and wait a random amount of time between 0 and $2^{backoff}$-1 before trying to retransmit. If the transmission is successful, the back off value is reset to 0, otherwise the back-off value is increased by 1 again and the process is repeated. Truncated binary exponential back-off (Kwak, Song and Miller, 2005) works in a similar way, but also sets a ceiling on the maximum back-off time. This is the industry standard for many computer networks including Ethernet.

Binary exponential back off seems like a promising approach to adjust individual Tabu tenures in a Tabu-search hyperheuristic (Burke, Kendall and Soubeiga, 2003) or metaheuristic, where the neighbourhood size is large. For low level heuristics that perform poorly, the Tabu tenure would increase exponentially with each poor

application and thus minimise wasted CPU time. For low level heuristics that perform badly at the start of the search and well at the end, little time would be wasted at the start but when the low level heuristic starts performing well, it will not be penalised for doing badly at the start. This could focus a very large set of low level heuristics down to a smaller set of low level heuristics which are effective at a particular point in the search.

Tabu search (Burke, Kendall and Soubeiga, 2003) can be used to make undesirable moves unusable for a certain number of iterations (the Tabu tenure). This is usually used to stop poorly performing moves being tried in succession or to stop the undoing of good moves. The optimal duration of a Tabu tenure has been tested in several papers and it is most likely a function of the neighbourhood size and the problem size (Laguna, Marti and Campos, 1999). Random Tabu tenures are used in (Rolland, Schilling and Current, 1996) where a move is made Tabu for a period randomly chosen between 1 and the maximum Tabu tenure and was found to be superior to fixed Tabu tenures. Much work has been done on Tabu search and related choice function methods (Chakhlevitch and Cowling, 2008).

Several papers have investigated Tabu-based and related choice function (Cowling, Kendall and Soubeiga, 2001) based hyperheuristics. (Kendall and Mohd Hussin, 2005) uses a simple Tabu mechanism where good and bad low level heuristics are made Tabu for a fixed Tabu tenure. A small number of low level heuristics are used (13) with short Tabu tenures (1-4 iterations) and good results are obtained in a large amount of CPU time. This is extended in (Kendall and Hussin, 2005) where the low level heuristic is repeated until no further improvements can be found before being made Tabu and random Tabu tenures are utilised. Random Tabu tenures provide results of similar quality to those with fixed tenure equal to the expected random tenure on the

two problem instances they consider. The repeated application of a low level heuristics does not increase solution quality considerably. (Burke, Kendall and Soubeiga, 2003) uses a ranking system for non Tabu low level heuristics. When a non-Tabu low level heuristic performs well its rank is increased, when it does not make a positive change its rank is decreased and the low level heuristic is put in the Tabu list on a first in first out basis. If it makes a negative change the Tabu list is emptied as it may have reached. At each iteration the low level heuristic with the highest rank that is not Tabu is used. The number of low level heuristics is again small, and the maximum size of the Tabu list is between 2 and 4. (Cowling and Chakhlevitch, 2003) use a Tabu hyperheuristic to manage a large set (95) of low level heuristics. The hyperheuristic allows the use of Tabu low level heuristics if it makes the best improvement (and then stops the low level heuristic from being Tabu). If no improving low level heuristics are available, a non improving non Tabu low level heuristic is used and made Tabu. Fixed Tabu tenures of 10, 30, 60 and 100 and adaptive Tabu tenures are investigated, but results provide no clear advantage of using adaptive Tabu tenures over fixed ones.

4.5.1. Hyperheuristic Approaches

Papers such as (Remde et al., 2007) (Chakhlevitch and Cowling, 2005) generate possible LLHs by considering separately (1) selecting a task to be scheduled and (2) allocating potential resources (including time) for that task. The task selector chooses a task and the resource allocator assigns resources for each skill required by the task, so that the total number of LLHs is the number of task selectors multiplied by the number of resource allocators. In the previous section resource selectors order the resources by their competency at the skill (as more competent resources can complete the task quicker) and then pick a range of these resources (Top 5, Top 10, etc). In addition to

these approaches, here we add more low level heuristics in an attempt to yield better results, by improving the likelihood of there being at least one good LLH in every situation. Table 4.4 describes the new resource allocators. Combining each of the 9 task selector with each of the 27 resource allocators gives a total of 243 Low Level Heuristics. Note that for this problem it is usually better to choose a group of uniformly poor competence resources for a task (so that they complete at about the same time) rather than a heterogeneous set (where fast, effective, resources have to wait for slower resources to finish when they could be completing other tasks).

Table 4.4. New Resource Selectors.

Name	Description
Deviation *x*	Resources complete a skill in a time dependent upon their competence. This selector attempts to find resources that will complete the different skills of task in the same amount of time by selecting resources with competencies that deviate *x={50%, 25%, 12.5%, 6.25%}* from the task's skill requirement.
x^{th} Quarter	This picks the *x={1,2,3,4}* quarter of task ranked by skill. Unlike the "Top x" task selectors, the number chosen is proportionate to the number of resources who can do the task.
x^{th} Eighth	This picks the *x={1…8}* eighth of task ranked by skill.
Dynamic *x*	This selector picks larger sets of resources for the skills requiring more effort and less to those requiring less effort. It will create *x={10, 50, 100, 1000}* combinations when enumerating the resulting sets.
All Resources	Considers all possible resources (and hence is very slow).

The hyperheuristic *HyperRandom*, selects at random a Low Level Heuristic (i.e. a (task order, resource selector) pair) to use at each iteration and applies it if the application will result in a positive improvement. This continues until no improvement has been found for a certain number of iterations. *HyperGreedy* evaluates all the Low Level Heuristics at each iteration and applies the best if it makes an improvement. This continues until no improvement is found. As might be expected, *HyperGreedy* is very

CPU-intensive, and generates good quality results, but is inefficient. For example, over one quarter of the low level heuristics were never applied in experimental trials in the previous section and over half of them were only applied once.

Here we propose a Tabu based hyperheuristic with dynamically adapting Tabu tenures designed for very large neighbourhoods, inspired by the binary exponential back-off algorithm used in networking (Metcalfe and Boggs, 1976). We use an analogous backing off method to exponentially increase the Tabu tenure of low level heuristics which repeatedly yield no improvements, meaning the time between trials of bad heuristics gets exponentially greater. The heuristic is given in Figure 4.13. We use two methods to decide which of the low level heuristics, which were tried, to back off (those "deemed bad"):

1) "Best *x*": only the best *x* improving low level heuristics are not backed off.

2) "Prop *x*": all non improving low level heuristics and those improving low level heuristics not in the top *x*% of the range of the fitness are backed off.

Define:
backoff_min is the minimum backoff value (we choose 4)
LLH_i is Low level heuristic *i*
$Tabu_i$ is the Tabu value of LLH_i ($0 \leq Tabu_i \leq Backoff_i$)
$backoff_i$ is the backoff value of LLH_i ($Backoff\_min \leq Backoff_i$)
$Eligible = \{LLH_i: Tabu_i=0\}$
$\Delta(S,LLH_i)$ is the change in the objective function which would result from applying low level heuristic LLH_i to solution S.
$apply(S,LLH_i)$ is the new solution we get after applying low level heuristic LLH_i to solution *S*.

Initialise:
create an initial solution *S* (often the solution *S* is the empty solution).
for all *i*:
    $backoff_i \leftarrow backoff\_min$
    choose $Tabu_i$ uniformly at random in $\{0,1,2,\ldots, backoff_i\}$
Iterate:
    while $(Eligible \neq \{\})$
        $best\Delta = 0$
        for each low level heuristic $LLH_i \in Eligible$
            if $\Delta(S,LLH_i) > 0$
                $backoff_i \leftarrow backoff\_min$
                if $\Delta(S,LLH_i) > best\Delta$
                    $best\Delta \leftarrow \Delta(S,LLH_i)$
                    $besti \leftarrow i$
            else
                if LLH_i is "deemed bad" (see text)
                    $backoff_i \leftarrow 2 * backoff_i$
                    choose $Tabu_i$ uniformly at random in $\{0,1,2,\ldots, backoff_i\}$
        for each low level heuristic $LLH_i \notin Eligible$
            $Tabu_i \leftarrow Tabu_i - 1$
        if $best\Delta > 0$
            $S \leftarrow apply(S,LLH_{besti})$
Terminate:
        for each low level heuristic LLH_i
            if $\Delta(S,LLH_i) > 0$
                $S \leftarrow apply(S,LLH_{besti})$
                go to **Initialise**

Figure 4.13. The Binary Exponential Back Off (BEBO) hyperheuristic.

4.5.2. Computational Experiments

We compare several hyperheuristic methods (*rVNS*, *HyperGreedy*, *HyperGreedyMore*, *HyperRandom*, *HyperRandomMore*, 9 BEBO *"Best x"*, 7 BEBO *"Prop x",* and *15* "standard" *Tabu Hyperheuristics*) ten times on five different problem instances and averaged the 50 results. The five problem instances require the scheduling of 400 tasks using 100 resources over one day using five different skills. Tasks require between one and three skills and resources possess between one and five skills. The problems reflect realistic problems Trimble MRM have identified and are generated using the problem generator used in (Cowling et al., 2006). The complexity of dealing with these real-world problem instances means that these experiments require over 155 CPU days to complete, so they were run in parallel on 88 cores of 22 identical 4 core 2.0 GHz Machines. Implementation was in C# .NET under Windows.

rVNS is best rVNS method taken from the previous section. *HyperRandom* and *HyperRandomMore* are the random hyperheuristics from the previous section with the latter including the additional low level heuristics introduced in this section. *HyperGreedy* and *HyperGreedyMore* are the greedy hyperheuristics from the previous section with the latter including the additional low level heuristics introduced in this paper. *HyperGreedyMore* will be the benchmark for all the tests as this is the most CPU-intensive approach and produces the best result.

The *BEBO* hyperheuristics are described in the above section. We try both of the proposed back-off methods with various sets of parameters. We also compare with a "standard" *Tabu* hyperheuristic, setting the Tabu tenure to *t=5, 7, 10, 25, 50* each time a low level heuristic is tried and fails to give an improvement. We also investigate different methods of deciding which LLHs to make Tabu. *TabuBest y t=x* signifies that all but the top *y* improving low level heuristics will not be made Tabu with tenure *x* at

each iteration. This is similar to the method used in (Burke, Kendall and Soubeiga, 2003) however we experiment with larger Tabu tenures as we use more low level heuristics. We also investigated making all non improving low level heuristics Tabu however the results for these were very poor in terms of CPU time (as nearly all of the low level heuristics make a positive improvement early in the search even if this improvement is very small) and these results are not reported below. In addition to these fixed tenures, we try random tenures as used in (Kendall and Hussin, 2005): *rTabu Best y t=x* is similar to *Tabu Best y t=x*, but with a random tenure between 0 and *x* each time a low level heuristic is made Tabu.

The results are presented in Table 4.5. We see that the availability of additional LLHs significantly improves the performance of *HyperGreedyMore* relative to *HyperGreedy* and *HyperRandomMore* relative to *HyperRandom*. The best BEBO method in terms of fitness is *BEBO Best 20*, which is also one of the slowest since it maintains a relatively large set of *Eligible* low level heuristics. Even so, the worst performing hyperheuristic *BEBO Best 1* got an average fitness of 97.64% of the average fitness of *HyperGreedyMore* in only 18.52% of the CPU time. Unsurprisingly, the size of the set of heuristics considered bad appears to determine the trade-off between solution quality and time reduction, although the reduction in solution quality is modest given the large reduction in CPU time. In all cases, random Tabu tenures improved Tabu search, further supporting previous work (Rolland, Schilling and Current, 1996). The fastest "standard" Tabu hyperheuristic *rTabu* was not as quick as the fastest BEBO method and also resulted in poorer quality solutions. The best *rTabu* hyperheuristic (in terms of solution quality) took much more CPU time than BEBO methods which gave similar quality.

To see how the time reduction scales with the number of low level heuristics, *HyperGreedy* and *Best 10* experiments were repeated 10 times with randomly chosen subsets of low level heuristics. The results, shown in Figure 4.14, compare *BEBO Best 10* with *HyperGreedy* with different numbers of low level heuristics. We can see that fitness of the two approaches remains very close (none of the results for *BEBO Best 10* dropped below 99.3% of the *HyperGreedy* fitness). As the number of low level heuristics decreases, the time savings for *BEBO Best 10* reduce. When 80 or 90% of heuristics have been removed, results are erratic (since the small set of low level heuristics is not guaranteed to be rich enough to yield good results). In this case *BEBO Best 10* deals better with the erratic nature and produces better fitness that *HyperGreedy* on average.

Table 4.5. Fitness and Time of individual instances for new hyperheuristic (average of 50 runs).

Method	Average Fitness	Average Time (s)	Fitness % of *HyperGreedy More*	Time % of *HyperGreedy More*
rVNS	21974.9	19.3	88.21%	0.25%
HyperRandom	19326.7	18.5	77.58%	0.24%
HyperGreedy	22084.0	233.0	88.65%	2.98%
HyperRandomMore	20553.8	176.3	82.51%	2.26%
HyperGreedyMore	24911.3	7807.2	100.00%	100.00%
BEBO Best 1	24324.6	1446.1	97.64%	**18.52%**
BEBO Best 2	24588.6	1775.4	98.70%	22.74%
BEBO Best 3	24774.3	2043.6	99.45%	26.18%
BEBO Best 4	24693.9	2077.5	99.13%	26.61%
BEBO Best 5	24734.6	2209.5	99.29%	28.30%
BEBO Best 10	24782.8	2572.5	99.48%	32.95%
BEBO Best 15	24869.3	2825.4	99.83%	36.19%
BEBO Best 20	24993.8	3150.9	**100.33%**	40.36%
BEBO Best 25	24927.1	3261.1	100.06%	41.77%
BEBO Prop 0.01%	24756.3	2341.1	99.38%	29.99%
BEBO Prop 0.05%	24737.2	2260.3	99.30%	28.95%
BEBO Prop 0.1%	24670.5	2295.6	99.03%	29.40%
BEBO Prop 0.5%	24753.3	2434.1	99.37%	31.18%
BEBO Prop 1%	24685.3	2278.6	99.09%	29.19%
BEBO Prop 5%	24543.7	2453.7	98.52%	31.43%
BEBO Prop 10%	24429.5	2507.3	98.07%	32.12%
rTabu Best 5 t=5	24420.5	4818.5	98.03%	61.72%
rTabu Best 5 t=7	24307.0	4151.0	97.57%	53.17%
rTabu Best 5 t=10	24121.2	3572.1	96.83%	45.75%
rTabu Best 5 t=25	23976.3	2663.8	96.25%	34.12%
rTabu Best 5 t=50	22872.2	2271.5	91.81%	29.10%
rTabu Best 10 t=5	24415.1	5305.6	98.01%	67.96%
rTabu Best 10 t=7	24459.0	4834.1	**98.18%**	61.92%
rTabu Best 10 t=10	24235.2	4251.3	97.29%	54.45%
rTabu Best 10 t=25	24149.5	3448.7	96.94%	44.17%
rTabu Best 10 t=50	24014.8	3047.5	96.40%	39.03%
Tabu Best 5 t=5	18104.2	2641.1	72.67%	33.83%
Tabu Best 5 t=7	19141.0	2577.4	76.84%	33.01%
Tabu Best 5 t=10	19364.5	2419.1	77.73%	30.99%
Tabu Best 5 t=25	18714.3	1968.9	75.12%	25.22%
Tabu Best 5 t=50	19139.1	1784.8	76.83%	**22.86%**
Tabu Best 10 t=5	20085.0	3443.8	80.63%	44.11%
Tabu Best 10 t=7	19246.5	3028.4	77.26%	38.79%
Tabu Best 10 t=10	19648.4	2675.3	78.87%	34.27%
Tabu Best 10 t=25	20143.4	2534.1	80.86%	32.46%
Tabu Best 10 t=50	20087.3	2351.1	80.64%	30.12%

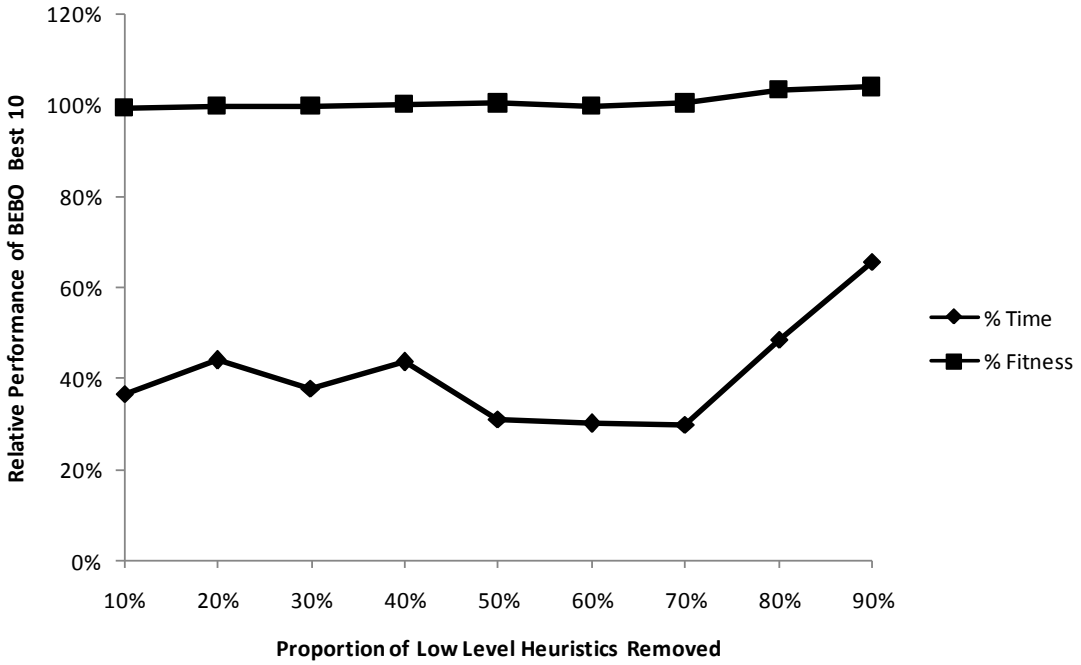


Figure 4.14. Graph showing the relative performance of *BEBO Best 10* to *HyperGreedyMore* with different neighborhood sizes.

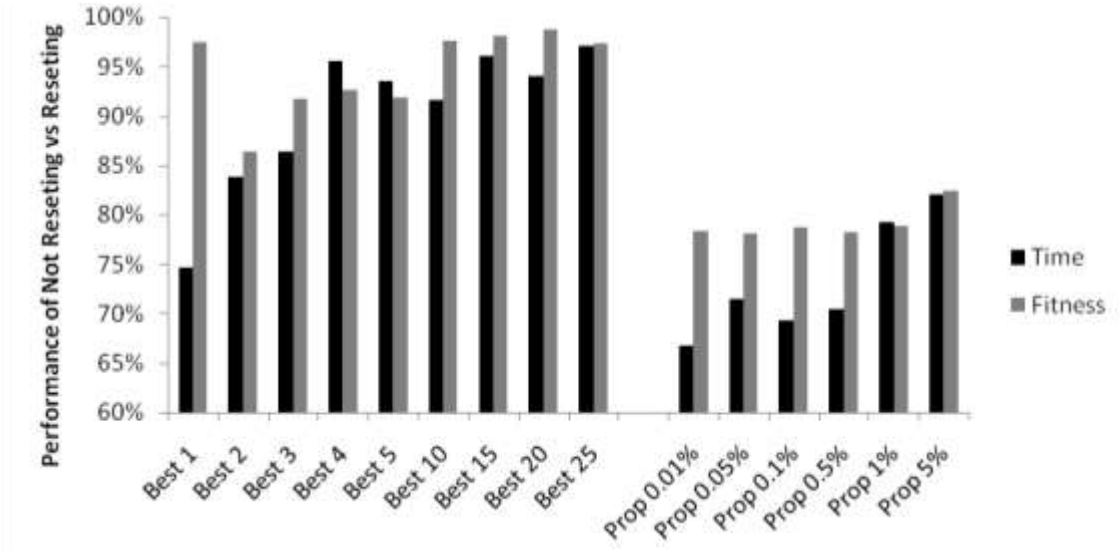


Figure 4.15. Graph showing the relative performance of each hyperheuristic without resetting compared to the performance of using the same hyperheuristic with resetting.

When the search finds no positive moves in the *Eligible* set of low level heuristics the Tabu tenures of the low level heuristics are reset and all low level heuristics are tried again. This is potentially a waste of time, if after resetting and trying

all the low level heuristics again no improvement is found. To see the impact this has on fitness and CPU time, the methods were tried with and without the reset. Figure 4.15 shows the relative performance of each hyperheuristic without resetting compared to the performance of using the same hyperheuristic with resetting in terms of time and fitness. Here 100% would denote that not resetting was equally as good as resetting – since all values are below 100%, and some are well below 100% these results indicate that resetting is essential, and its effects on the time used are modest.

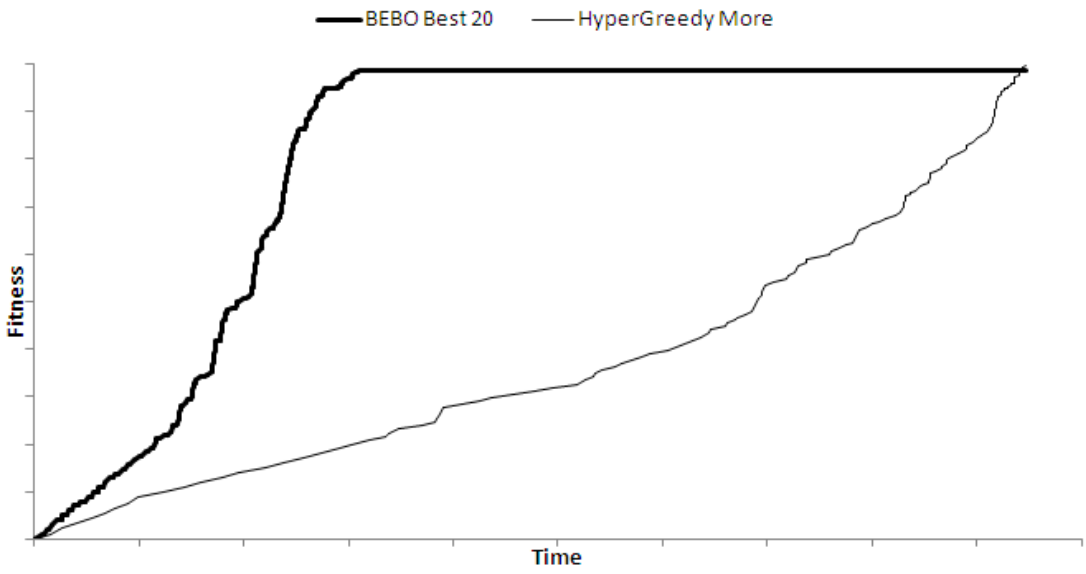


Figure 4.16. Graph showing the fitness against CPU time of the *BEBO Best 20* and *HyperGreedyMore* heuristics.

Figure 4.16 shows the difference in the evolution of fitness through time for *BEBO Best 20* and *HyperGreedy More*. The curve for BEBO Best 20 stays well ahead of the corresponding curve for *HyperGreedy More*, until very late in the search process. There is good evidence here of the efficiency savings resulting from exponential back-off, which waste far less function calls to poor LLHs.

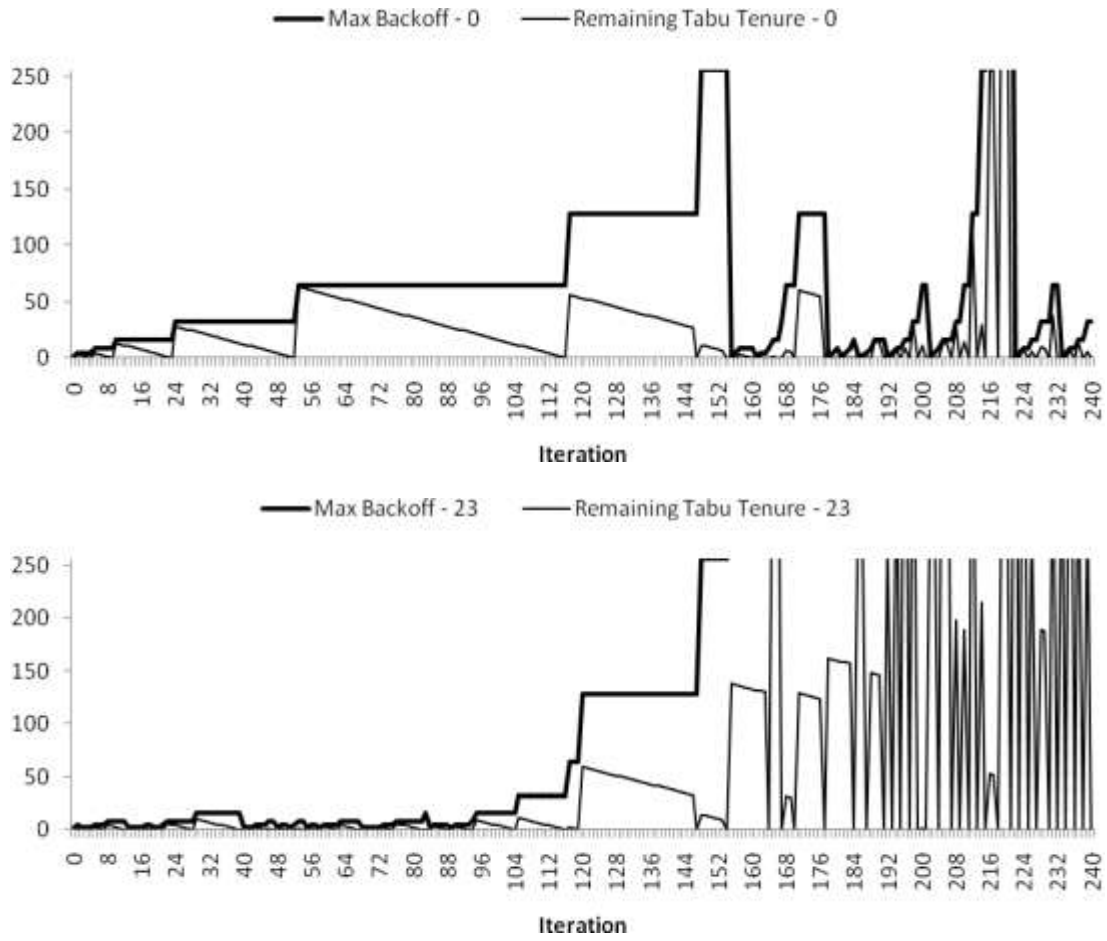


Figure 4.17. **Graph showing the Tabu Tenure and Backoff of a two different low level heuristics over the iterations of the search.**

Figure 4.17 shows how the BEBO Tabu tenures change over the search for two typical low level heuristics with very different properties. The plot shows $Backoff_i$ and $Tabu_i$ at each iteration of a *BEBO Best 20* run for LLH_0 and LLH_{23}. LLH_0 is used towards the end of the search and as we can see, BEBO is very efficient only trying the low level heuristic about 7 times in the first 150 iterations. Later in the search LLH_0 maintains a low back-off value, and produces useful solution improvements. LLH_{23} performs well at the beginning of the search but after about 96 iteration it no longer serves a useful purpose. Hence Figure 4.17 clearly shows the back off value and the Tabu tenure increasing so that little time is wasted with the low level heuristic later in

the search. These behaviours are typical of the BEBO approach, as is the (less interesting, and very common) behaviour where a consistently poor heuristic is called only rarely right throughout the search.

4.5.3. Analysis

This section investigates the use of Binary Exponential Back-Off (as used in computer and telecommunications networks) to set the Tabu tenure for low level heuristics in a hyperheuristic framework. The approach has been empirically tested on a complex, real-world workforce scheduling problem. The results have shown the potential of the new method to generate good solutions much more quickly than exhaustive (greedy) approaches and standard Tabu approaches. In particular, the benefits of the approach increase with increasing neighbourhood size (i.e. with an increased number of low level heuristics). Binary Exponential Back-Off is able to produce results very close to a highly CPU intensive greedy heuristic which investigates a much larger set of low level heuristics at each iteration, in terms of fitness, and is able to do so in a fraction of the CPU time. BEBO performs much better than “standard” fixed and random Tabu tenure hyperheuristics. Different method for deciding which heuristics to back off were tested, since adjusting the number of low level heuristics backed off determines the trade-off between CPU time used and solution quality. We have shown that modifying the number of low level heuristics backed off may be used to adjust the search and trade off time available against solution quality.

In principle our exponential back-off methods could be used in any Tabu implementation with large neighbourhoods, which provides a promising possible direction for further research.

4.6. Summary

In this chapter, first search based heuristic were used to improve schedules produced by a genetic algorithm. The results showed that solutions generated by the simple construction heuristic in the genetic algorithm were poor and could easily be improved with local search. Various methods were used and the results compared showing local search as a promising technique for the problem.

Then, a hyperheuristic framework was investigated and low level heuristics were created using exact/heuristic hybrids. These low level heuristics worked on a specific part of the problem and solved it exactly. The heuristic part picked a task to be inserted next and the potential resources to do the task. The exact part then enumerated every possible insert using that task and a subset of the potential resources to find the best insertion.

Experimentation was done using over 100 of these low level heuristics. Results showed that using them in a greedy search, though time consuming, produced better results than a reduced Variable Neighbourhood Search method. The reduced Variable Neighbourhood search was the best of many hand crafted heuristics created using selected low level heuristics as neighbourhood functions.

Analysis of the low level heuristic usage in the greed hyperheuristic showed that about a quarter of the low level heuristics were never used and 50% were used less that 1% of the time. This suggests that the greedy hyper heuristic could be sped up by not wasting time trying these poorly performing low level heuristics (if they could be identified).

Binary Exponential Back-off was used as a method to control Tabu tenures in a Tabu hyperheuristic based on the greedy hyperheuristic. The aim was to exponentially decrease the time you spend trying bad heuristics without penalising them if they do

start performing well. Many variations were tried and the results compared to the unmodified greedy hyperheuristic and standard Tabu based hyperheuristics. The results showed that using binary exponential back-off gave the ability to trade of solution quality for CPU time, depending on the parameters used. Even so, the loss in solution quality was extremely small compared to the savings in CPU time. Selected parameters produced results of 97.64% fitness using 18.52% of the CPU time and 100.33% fitness in 40.36% of the CPU time when compared to the greedy hyperheuristic.

Analysis of the back-off of some of the low level heuristics showed that the back-off was working as intended and that low level heuristics which performed well were not backed off and these that performed repeatedly poor were backed of exponentially until they started performing well. These results show Binary Exponential Back-Off to be a very useful method for decreasing CPU time in hyperheuristics and potentially other search based heuristics with large neighbourhoods.

Chapter 5

The Variable Fitness Function

In this chapter a new search technique called the Variable Fitness Function will be defined and analysed. The Variable Fitness Function can be used to enhance a search based optimisation method if the method uses a weighted sum fitness function (or can be made to do so) and extra CPU time is available.

5.1. Motivation

When modelling a real-world decision problem as a problem of combinatorial optimisation, it is usually assumed that there is a single underlying objective (fitness) measure to allow automatic comparison between candidate solutions. In real problems, this objective measure is almost always a function of several underlying sub-objectives relating to revenue, cost, staff and customer satisfaction, sustainability etc. (Viana and de Sousa, n.d.) (Thiagarajan and Rajendran, 2005), and solution heuristics are often highly tailored to deal with complex problem-specific decision rules. In commercial computerised decision support systems a weight is usually assigned to each sub-objective to reflect its relative importance, and the objective consists of a weighted sum of sub-objectives (Thiagarajan and Rajendran, 2005). Allowing the user to make these choices of relative importance up front often works well in practice, since it allows (and empowers) users to make difficult a priori decisions of importance as a decision support system is implemented, and potentially to engage in "what if?" analysis of different sub-objective weights, when time allows (MacCarthy and Wilson, 2001). The weighted sum objective reflects the relative importance of sub-objectives in a "finished" solution. This is not an issue for *exact* approaches which guarantee to find an optimal solution (e.g. (Albiach, Sanchis and Soler, n.d.)), but since heuristic approaches are usually used for hard combinatorial optimisation problems (Kolisch and Hartmann, 2006), the objective weights for a "finished" solution may be a poor reflection of the objective weights which would provide the best local decision for a heuristic, working on a partial or poor-quality solution. Designing an objective function which varies from iteration to iteration, and takes account of the features of the current solution and the heuristic method(s) used to solve it appears to be a very difficult task.

Local search and other search based heuristics have the problem of local minima. That is solutions to which there are no better neighbouring solution but which the solution is not globally optimal. Metaheuristics attempt to overcome this, generally by giving the search the ability to accept a worse solution in the hope of finding a better solution, however due to the nature of the fitness function, they may fail. Consider Figure 5.1 where the grey areas are infeasible solutions and the star marks the incumbent solution.

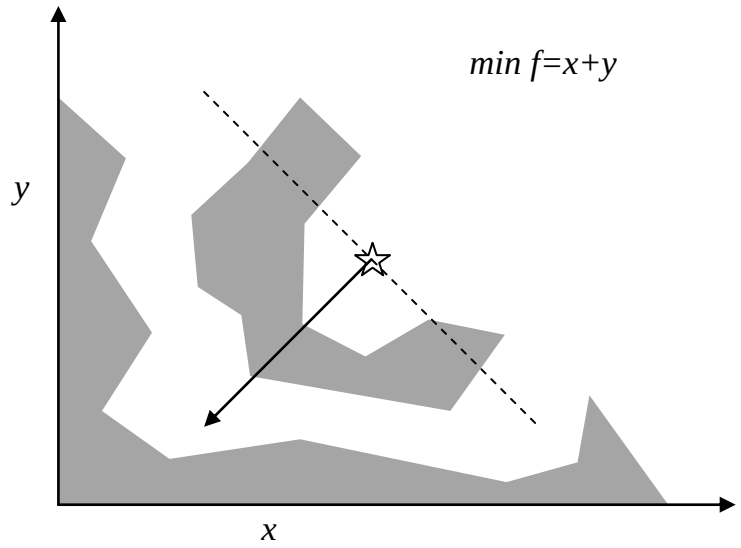


Figure 5.1. Sample minimisation problem for a greedy steepest descent search

The arrow here indicates the search direction and the dotted line shows the tangent to the search direction. A greedy hill climbing/or steepest decent search would only accept solutions to the left of this line, so it is clear from this example that a local search using one of these techniques would fail to find the global optima, as it could never leave the local optima it is heading for. Metaheuristics provide a solution for this by allowing the acceptance of inferior solutions. For example, VNS makes kick moves when local optima are reached. These kicks get increasingly bigger. But if these kicks get too big,

then it becomes like a random restart, and if too small, the local optima can't be escaped. Figure 5.2 shows an example of this.

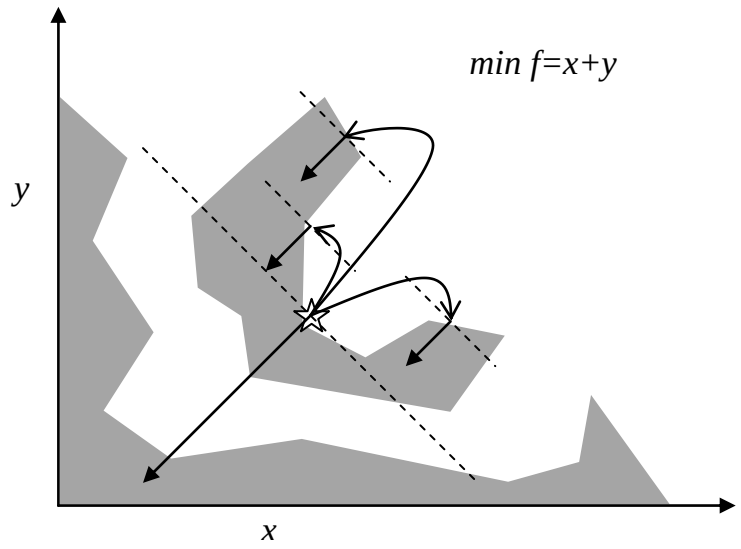


Figure 5.2. Example moves the kick method of a Variable Neighbourhood Search may make

Designing neighbourhood functions, or modifying existing ones to try and overcome these problems (to which identifying the problem may be half the task), may be impossible, hard or require a lot of work. A simple way to achieve the same task is to modify the problem. Instead of modifying the problem, which could be as hard as designing new heuristics/neighbourhoods and is specific to the given problem, we decide to modify something common in all optimisation problems: the fitness function. Because of this, the Variable Fitness Function can also be used to enhance searches that are bespoke or too complex to improve and this can be done without problem knowledge. This could potentially save a lot of expert time trying to create new heuristics for a new problem and indeed from our experimental results we have evolved variable fitness functions which have observed behaviour similar to well known heuristics.

If in the problem given above, the fitness function was modified to *min f = y – x* at the start and then changed back to *min f = x + y*, we might see a search move as in figure 5.3. In general, we cannot see the shape of the fitness landscape and so defining these changes becomes the problem. For this we use an evolutionary algorithm defined in the next section.

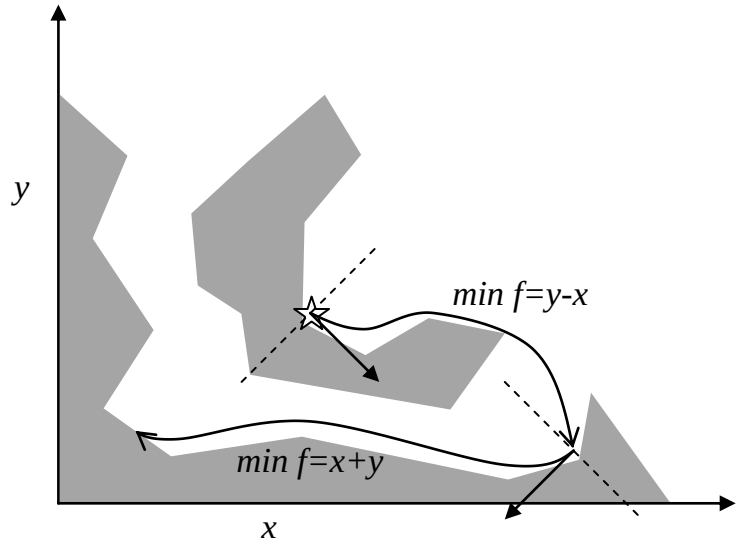


Figure 5.3. Example of how changing the fitness function can lead to escaping a local optima

5.2. Definition

The Variable Fitness Function approaches describe how the weights of a weighted sum fitness function change over the iterations of a search process, although other approaches are possible where the VFF modifies a nonlinear combination of objectives, or adjusts weights dependent upon other factors than time. The Variable Fitness Function in this case is piecewise linear, describing the relative importance of objectives at each iteration. We consider two alternatives: the standard Variable Fitness Function fixes the number of discontinuities and the number of iterations between them. The

adaptive Variable Fitness Function allows the points of discontinuity to evolve along with the Variable Fitness Function objective weights (These two methods will be described in detail in the next sub sections.)

5.2.1. Standard Variable Fitness Function

We define a set of weights $\{W_{a,b}\}$ where a indexes the weight set ($a=0...A-1$) and b indexes the objective ($b=1...B$). We define I, the number of iterations between the weight sets. The Variable Fitness Function is now defined as:

$$f(s,i) = \sum_{b=1}^{B} O_b(s) W_b(i)$$

where s is the solution to be evaluated and i is the iteration, $O_b(s)$ is the value of objective b for solution s, and

$$W_b(i) = W_{b,\lfloor i/I \rfloor} + \frac{i \bmod I}{I}(W_{b,\lfloor i/I \rfloor + 1} - W_{b,\lfloor i/I \rfloor})$$

(i.e. the linear interpolation of the weight of objective b for iteration i)

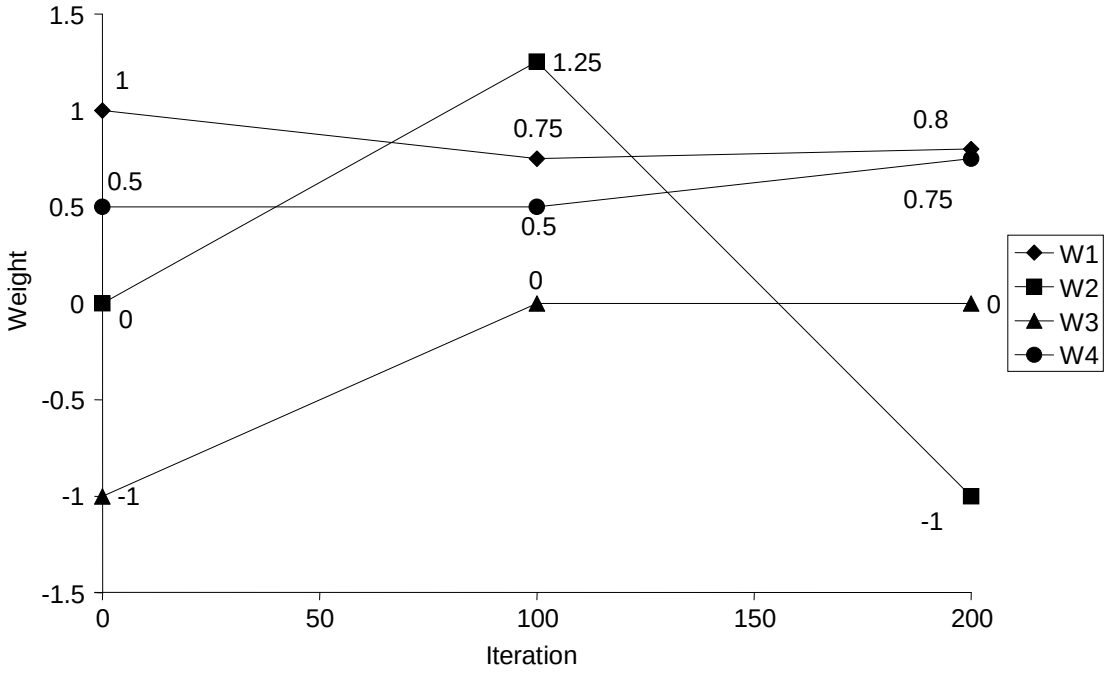


Figure 5.4. An example standard Variable Fitness Function. The number of weight sets (3) and the number of iterations between them (100) are fixed.

Figure 5.4 shows how the weights of an example Variable Fitness Function change over iterations. The final set of weights are evolved like the rest of the weights and are not fixed as the global fitness function, however, observations of evolved variable fitness functions show that these weights are often similar to the global fitness function.

5.2.2. Adaptive Variable Fitness Function

Initial experiments with the standard Variable Fitness Function quickly showed its weakness. If the number of iterations between discontinuities was too small, the solution quality would suffer due to the unstable nature of the objective weights. If the number of iterations was to large, CPU time would be wasted while stuck in local optima waiting for a change of objectives. The adaptive Variable Fitness Function does not require an effective number of weight sets and iterations between them to be known. These are evolved along with the weight data to find appropriate values, which can lead

to more complex Variable Fitness Functions. For the adaptive version we define a set of "gaps" I_a such that weight set a starts at I_{a-1} iterations after the previous weight set (or at iteration 0 if it is the first weight set). We then redefine $W_b(i)$ as

$$W_b(i) = W_{b,c} + \frac{i-j}{k-j}(W_{b,c+1} - W_{b,c})$$

where iteration i occurs in the range from weight set c (starting at iteration j) to weight set $c+1$ (starting at iteration k).

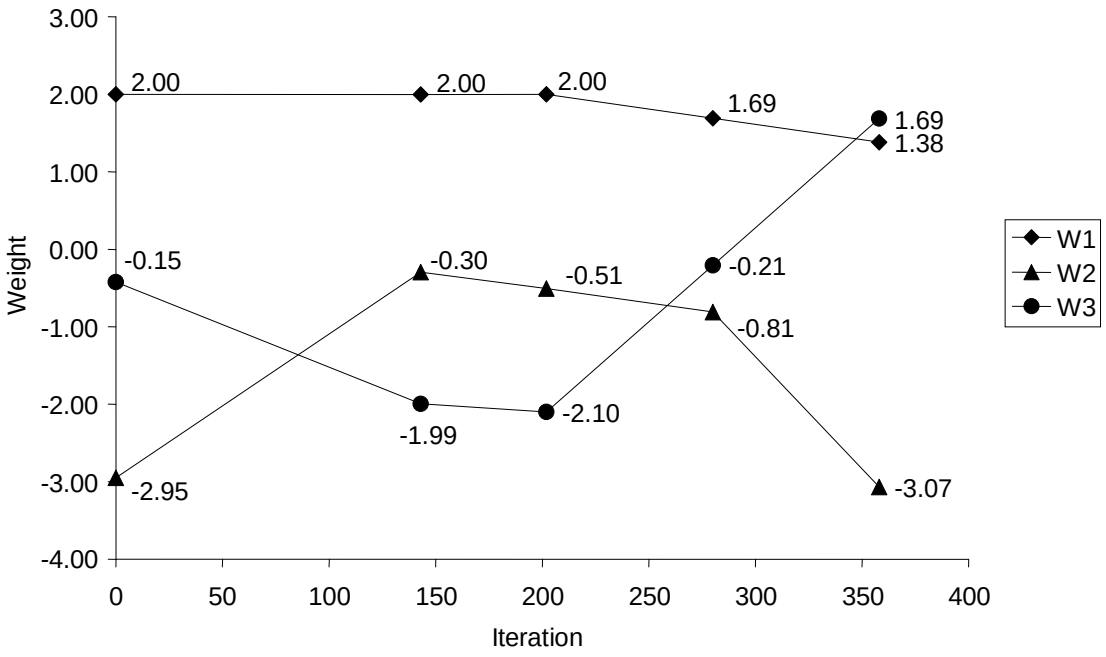


Figure 5.5. An example adaptive Variable Fitness Function. In this example, the number of iterations between the weight sets and the number of weight sets may vary.

Figure 5.5 illustrates an adaptive Variable Fitness Function, showing how the weights change over the iterations, for example, that the weight of objective 1 (W1) starts at 2 (hence the objective is to be maximized) and then after iteration 200 its importance

starts to decrease. This if often useful for objectives which are high level (such as the priority of scheduled tasks) but which become less important later, during schedule refinement Objective 3 (that has weight W3) is to be minimized, and its importance is higher at the start and end of the search process. This can be useful for objectives related to schedule stability, when repairing a schedule, which must be taken into account while major changes are being made, early in repair, but can then assume less importance during a period of diversification to avoid local optima, before being "fixed" at the end of the process.

5.2.3. Evolution

Little work has been done in encoding piecewise linear functions such as these into chromosomes. A complex encoding for polynomial expressions is proposed in (Potgieter and Engelbrecht, 2007), where an encoding is used to optimize a curve to fit a function described by a set of data points and is not an appropriate method in this case. The evolution here is actually more similar to work done on tuning of parameters for another algorithm using genetic algorithms (Shimojika, Fukuda and T., 1995).

When optimizing the weights of the Variable Fitness Function, each weight in the Variable Fitness Function appears as a gene in a GA chromosome. When the adaptive Variable Fitness Function is used, the iterations between the weight sets are also included. Figure 5.6 shows how the weight sets are mapped to the genes of a chromosome.

A modified version of 1 point crossover (Reeves, 1995) will be used, where the crossover point may only be on a weight set boundary so as to keep mutually compatible weight sets together. The thick lines in Fig 5.6 show these crossover points. Each gene will have a chance to be mutated with a probability of p_{mut}, the mutation rate.

Mutation will simply perturb the value of the gene by adding a random variable normally distributed around 0 and with the standard deviation defined for that weight. Hence $W_{a,b}$ is perturbed by a value from the normal distribution $N(0, V_b)$ with probability p_{mut}. Where V_b is the standard deviation of mutation associated with objective b and p_{mut} is the probability of mutation, which is the same for all alleles. This is similar to work done on mutation of artificial neural network weights evolved using GAs (Yao, 1999) where the network weight is mutated by a random number selected from a normal distribution.

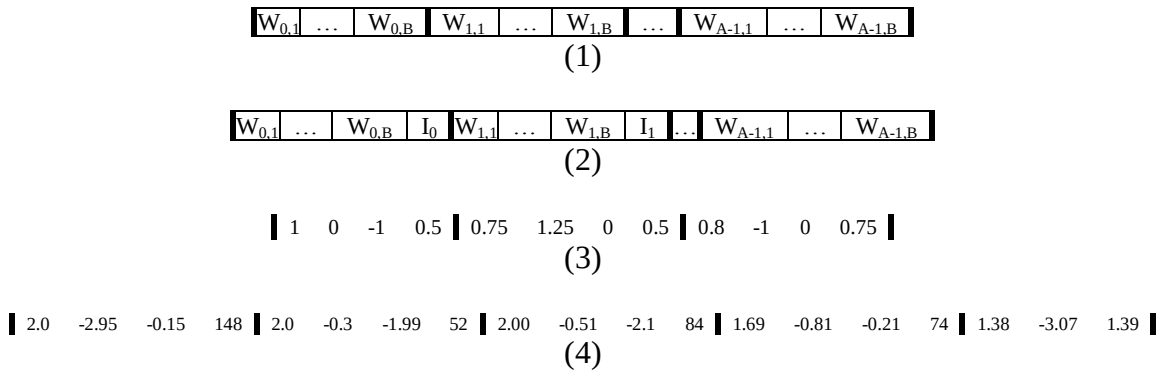


Figure 5.6. Mapping the weights to a chromosome for a standard Variable Fitness Function (1) and an adaptive Variable Fitness Function (2). (3) shows chromosome representing the fixed length Variable Fitness Function in Figure 5.4. (4) shows chromosome representing the adaptive length Variable Fitness Function in Figure 5.5.

The initial population of Variable Fitness Functions is generated at random. We may also seed the initial population with the global fitness function. These seeds are Variable Fitness Functions where the weights are constant over all iterations and equivalent to the global fitness function, and we insert suitable break points to allow for later mutation. This may give the genetic algorithm a good individual to work from or provide good genetic material to create other individuals. For the random individuals $W_{a,b}$ is picked uniformly at random out of the interval $[L_b, U_b]$, where L_b and U_b are

parameters which may be different for each objective *b*. Seeding the initial population with the global fitness function and using an elitist replacement scheme would ensure that in a worst case scenario, the best individual of the final population is a Variable Fitness Function representing the global fitness function. In cases where the global fitness function is seeded we often see the evolution process building off these seeded individuals.

In the adaptive VFF there is also a p_{adapt} probability that the chromosome will change length to either increase or decrease the complexity of the variable fitness function. If a chromosome is to change length there is an equal probability it will either shrink or grow by one weight set. If it is to shrink, a random weight set is chosen and removed from the chromosome. If it is to grow, a new weight set is inserted between two randomly chosen adjacent weight sets. The inserted weight set does not immediately change the shape of the Variable Fitness Function as it is inserted exactly half way between the two adjacent weight sets and has weight values that are the mean of the bordering weight sets. The new weight set is then mutated to introduce slight variations. The I_a genes also have a p_{mut} probability of being mutated by a normal distribution with a variance of 5% of their initial value. This gives the chromosomes a chance to get more and less complex and to also expand to more or less iterations.

We have introduced several parameters in this section but our experiments using the TSP and workforce scheduling problems show that the performance of VFF is not very sensitive to these parameters, so L_b and U_b can be set to -1 and 1 respectively without losing any information as any weighted sum objective function can be normalized so that weights will generally lie within this range. Normalization of a vector of weights from a weighted sum fitness function is done by dividing the vector by the maximum absolute value of the vector's component values. We have found that

V_b set to any value around 5% of the range ($V_b = 0.05(U_b - L_b)$) also works well. All the variable fitness function experiments in this thesis have used these default values unless stated otherwise. In the experiments carried out $p_{mut} = p_{adapt} = 0.05$ unless stated otherwise. This lack of sensitivity could be due to the higher level nature of the Variable Fitness Function, and a similar lack of parameter sensitivity has been observed in the study of hyperheuristics which also operate at a higher level (Chakhlevitch and Cowling, 2008).

5.3. Comparison with other Meta-heuristics

The Variable Fitness Function is a metaheuristic as it works with another heuristic but at a higher level and can be applied to any search based optimization heuristic without modification so long as two conditions are satisfied: (i) The objective function can be expressed in terms of two or more sub-objectives (which is almost always the case in our experience of practical problems) or additional objectives can be found for single objective problems; and (ii) We have enough CPU time to run the heuristic many times. More importantly, no modification of the underlying heuristic is needed.

Several metaheuristics have been described in literature that use the same method of changing the Fitness Function to aid the heuristic process. Generally speaking these metaheuristics require problem specific knowledge and modifications to the local search process. If the search heuristic is bespoke and complex (as it often is for real-world problems), some metaheuristics may prove hard to implement. Here we describe three local search metaheuristics that are effective and have the practical advantage that they are easy to implement for complex, practical problems (although

even for these three heuristics, we believe the VFF approach is likely to be easier to implement in practice).

Simulated annealing (Aarts and Korst, 1989) works by changing the acceptance criteria of a local search operator (an artificial way of changing the fitness function). It will always accept moves which lead to a better solution, however it also has a chance to accept moves that make the solution worse (according to a global fitness function). This probability of accepting a worse move is controlled by a cooling scheme and is inversely proportional to how bad the move is and how long the search has been conducted. High early acceptance probability helps diversity at the beginning and reducing probability helps intensify the search toward the end. In the survey done by Kolisch and Hartmann (Kolisch and Hartmann, 2006) the heuristic was shown to be competitive and performed well ranking about midway of the tested heuristics for the Resource Constrained Project Scheduling Problem (RCPSP). Simulated Annealing has the great advantage for complex, practical problems, that it is usually easy to implement, Given a local search heuristic, all that is needed is a simple change the acceptance criteria. However, tuning the cooling scheme so that it is effective for most/all problem instances can be tricky and is usually a matter of trial and error.

Guided Local Search (GLS) (Tsang and Voudouris, 1997) modifies the fitness function to change the search direction the search heads when search is stuck in a local optimum. Features of a solution are identified and penalties for solutions exhibiting these features are increased when the solution is stuck in a local optimum. Guided Local Search redefines the objective function thus:

$$f'(s) = f(s) + \lambda \sum_{i=1}^{F} p_i I_i(s)$$

where λ is the weighting for the guided local search, F is the number of features, p_i is the penalty value for the i-th feature and $I_i(s)=1$ when s exhibits feature i, 0 otherwise. When the search settles in a local optimum s^* the utility of penalizing feature i is defined by:

$$util_i(s^*) = I_i(s^*) \times \frac{c_i}{1+p_i}$$

The feature or features with the largest utility will be penalized by increasing their penalty values. This has the effect of changing the fitness function and forces the search to move in another direction. An evolutionary variation of GLS is given by the Stepwise Adaptation of Weights (SAW) evolutionary algorithm (eggermont, 1999). SAW also uses changes of weights to escape local minima, but in this case the weights are applied to an evolutionary algorithm rather than a local search algorithm. The SAWing algorithm of (Eiben et al., 1998) was applied to Constraint Satisfaction Problems, where each constraint is given a weight. After every n evaluations, the weights of the fitness function are adjusted depending on the constraints that are the most violated in the population. For a Constraint Satisfaction Problem, identifying "features" is easy for both the GLS and SAW EA, however it may not always be straight forward for more complex problems. The VFF approach differs from the SAWing and GLS approach essentially in that the VFF approach finds weights for objectives which are already present, so that there is a much "lighter touch" in terms of modification of the solution. Later we will show that this light touch can be highly effective even though it requires only very limited change to the solution heuristic.

Variable Neighborhood Search (Mladenovic and Hansen, 1997) (VNS) is based on the idea of systematically changing the neighborhood of a local search algorithm. Variable Neighborhood Search enhances local search using a variety of neighborhoods to "shake" the search into a new position after it reaches a local optimum. Several

variants of VNS exist as extensions to the VNS framework (Hansen and Mladenovic, 2001) which have been shown to work well on various optimization problems. Variable neighborhood search is relatively easy to implement. The shake moves can simply be "chained" random local search moves as in (Lin, 1965) but if this is not adequate, new shake moves may have to be implemented. These shake moves ignore the fitness function and are just accepted.

These local search metaheuristics all require modification of the local search. In the case of simulated annealing, only a minor change to the criteria of accepting a neighbouring solution is needed, however in guided local search and variable neighbourhood search, much larger changes are needed. The methods may also require extensive tuning and may be sensitive to the addition of spurious objectives and constraints. The Variable Fitness Function requires no modification of the underlying local search and hence can easily be used to enhance any local search method. This becomes particularly important when trying to solve complex, real-world problems with a wide range of objectives and a detailed model, where the VFF approach provides a straightforward way to further enhance an existing approach.

5.4. Summary

This chapter introduces and motivates the Variable Fitness Function as a general-purpose approach to enhance a wide variety of search methods. The Variable Fitness Function defines precisely how the weights of a Weighted Sum Fitness Function change over the course of the search with the aim of evolving new heuristics Unlike any of the literature, these changes are defined before the search starts and does not use local information to decide on the path. A simple evolutionary algorithm is described which

can be used to evolve Variable Fitness Functions which vary over the iterations of a search. The approach is compared to related methods from the literature.

Chapter 6

Variable Fitness Function Case Studies

In this section the Variable Fitness Function is implemented for various problems with unique and interesting characteristics. Experimental investigations show the effectiveness of the Variable Fitness Function to produce superior results.

6.1. Application to the Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is a well studied optimization problem which usually has the single objective to minimize tour length (Lawler et al., 1985). We study a multi-objective variant of the TSP (MO-TSP) and use the Variable Fitness Function to guide a 2-opt local search (Croes, 1958) to find better solutions than 2-opt alone.

The TSP consists of a set of n cities, and a cost matrix c_{ij} ($1 \leq i,j \leq n$) that defines the cost of travelling from city i to city j. The aim of the TSP is to determine a tour of minimum length visiting each city only once and returning to the starting city. The Symmetric TSP (STSP) adds a further constraint that $c_{ij} = c_{ji}$ for all i,j. We study a variant of this such that each (unordered) pair of cities has B uncorrelated objectives associated with it. The global objective is to find a tour which minimizes a weighted sum of the B objectives. Such a problem can easily be converted into an STSP (and in fact we do to solve them exactly). This way of creating multi objective TSP problems is related to the work of Jaszkiewicz et al (Jaszkiewicz, 2002) who use uncorrelated objectives to make the symmetric TSP into a multi-objective problem. We simply split each edge weight at random to demonstrate the effectiveness of VFF in this (uncorrelated) case.

Figure 6.1 shows an example problem. In the problems we look at, all costs are generated uniformly at random between 0 and 1. "Splitting" each edge in this way provides a way to convert a single-objective TSP to a multiple objective problem, which is readily applicable to other problems. However, in most of the real-world problems which we study, including the workforce scheduling problem, there are already many (often too many!) objectives.

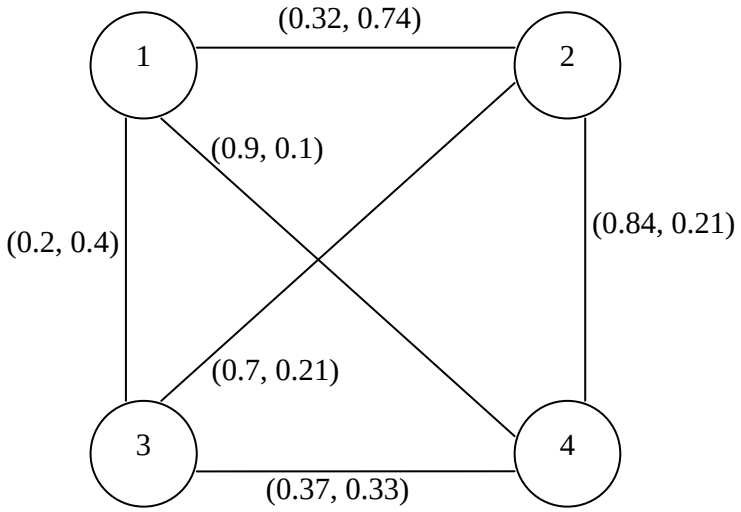


Figure 6.1. Example MO-TSP with 4 cities and 2 objectives

6.1.1. Variable Fitness Function Usage

We investigate 3 different constructive solution generation methods and the 2-opt local search improvement heuristic. We will then use the Variable Fitness Function to enhance these methods in two ways. Firstly, just enhancing the 2-opt part and secondly enhancing the initial solution generation method as well. The methods are shown in Table 6.1. All of these heuristics for the TSP are well known – see (Lawler et al., 1985) for details.

The three initial solution generation methods we use are an arbitrary random solution (*AR*), and solutions generated using the nearest neighbor (*NN*) and multiple fragment (*MF*) constructive heuristics. The arbitrary solution is simply the tour where all the nodes are visited in a fixed, randomly chosen order. Nearest neighbor starts the tour at a given point and repeatedly adds the nearest unvisited city to the city at a fixed end of the current partial tour until a complete tour is found. Multiple fragments is similar to nearest neighbor as at each iteration it adds an edge between the two closest

unconnected cities whose connection does not form a cycle (unless it is the last edge to be added).

Table 6.1. Heuristics and Variable Fitness Function enhanced versions used in our experiments

Heuristic	Description
AR	Solution is an arbitrary solution
NN	Solutions are generated using a stochastic nearest neighbor algorithm
MF	Solutions are generated using a stochastic multiple fragment algorithm
AR + 2opt	Solutions are generated using AR and improved using a stochastic 2-opt
NN + 2opt	Solutions are generated using NN and improved using a stochastic 2-opt
MF + 2opt	Solutions are generated using MF and improved using a stochastic 2-opt
AR + VFF(2opt)	Solutions are generated using AR then improved using 2-opt where the fitness function for 2-opt is evolved
NN + VFF(2opt)	Solutions are generated using NN then improved using 2-opt where the fitness function for 2-opt is evolved
MF + VFF(2opt)	Solutions are generated using MF then improved using 2-opt where the fitness function for 2-opt is evolved
VFF(NN + 2opt)	Solutions are generated using NN and improved using 2-opt where the fitness function for both the NN and the 2-opt algorithm is evolved
VFF(MF + 2opt)	Solutions are generated using MF and improved using 2-opt where the fitness function for both the MF and the 2-opt algorithm is evolved

Table 6.2 shows the parameters used in the evolution. Picking the weights between -1 and 1 gives us the possibility to start the search in every direction, including those negatively correlated with the global fitness function.

2-opt is a simple local search heuristic which improves a TSP solution by finding edges *(i, j)* and *(k, l)* in the current tour such that $c_{ij} + c_{kl} > c_{ik} + c_{kl}$ and replacing edges (i, j) and (k, l) with (i, k) and (j, l). For each of the NN, MF and 2-opt heuristics we consider stochastic versions where instead of greedily choosing the best at each iteration, we choose peckishly (Corne and Ross, 1995), so that we choose one of the best two

possibilities at each iteration, with equal probability, so generating a different solution in each run and allowing us to use extended CPU time effectively when doing multiple restarts.

Table 6.2.Parameters used to evolve the Variable Fitness Functions for the multiobjective TSP problems

Objective $b_.$	Initial Value $L_b \ldots U_b$	Standard Deviation V_b
1	-1…1	0.1
2	-1…1	0.1

6.1.2. Computational Experiments

Each of our ten methods was run 5 times for each of 5 problem instances. They were given the same CPU time (15 minutes on a 3Ghz Pentium 4, with all code implemented in C#) in which to find a solution and were restarted if they completed before the allotted time. The optimal solution of each of the 5 instances was found using CONCORDE (Applegate et al., n.d.). The five instances have 100 cities and 2 objectives, equally weighted in the global fitness function. The quality of a method will be assessed by the average deviation from the optimal tour, measured by this global objective, over the 25 runs.

To tune the genetic algorithm parameters for the Variable Fitness Function evolution, the genetic algorithm was run for 1000 fitness evaluations with different population sizes of 10, 20 and 40 in an attempt to find the best parameters. The population was seeded with *0, 1* and *All* global fitness functions to see the difference. The parameter tuning experiments show that the genetic algorithm was not sensitive to the parameters. A population size of 20, and seeding with no global fitness functions were among the best set of parameters and were used for the rest of the experiments.

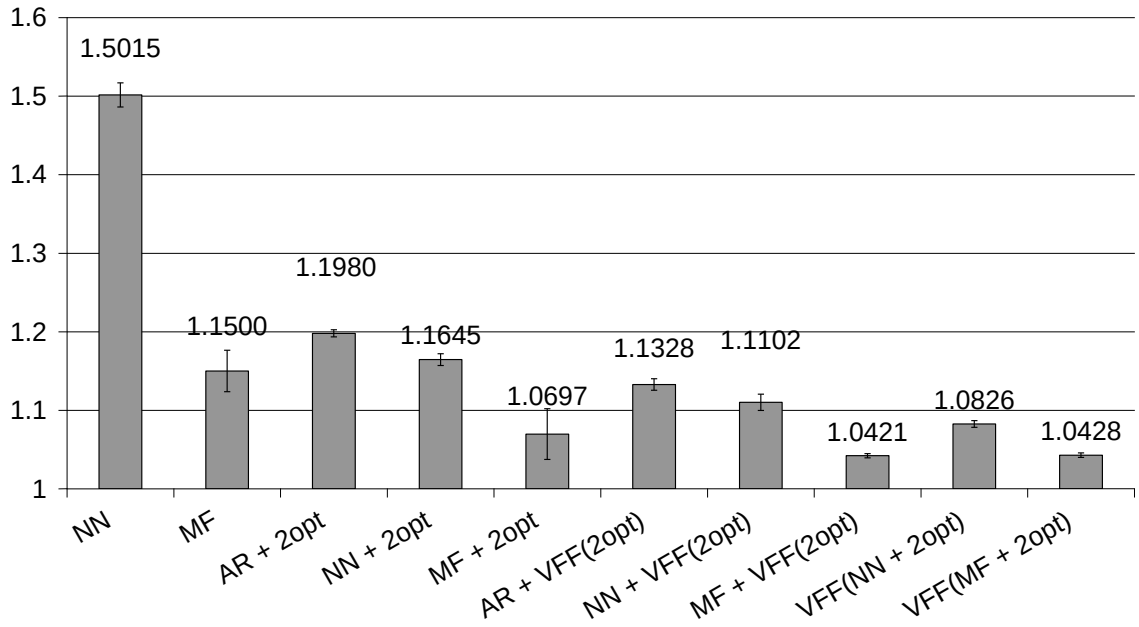


Figure 6.2. Average Deviation of the ten tested methods from the optimal solution. Error bars show 90% confidence intervals averaged over 5x5 runs.

The comparative results are shown in Figure 6.2, showing each method's average deviation from the global optimum over 5 runs of 5 problems instances with 90% confidence intervals. We can see that NN provides the weakest result. 2-opt can be seen to improve the NN and MF heuristics considerably as expected. When we enhance the 2-opt with the Variable Fitness Function, we can see significant further improvements (at 90% confidence). Overall the results demonstrate that the *MF + VFF(2opt)* and *VFF(MF + 2opt)* perform the best. For *every* heuristic H in our experiments, *VFF(H)* performs much better than *H*. MF + 2opt has a wide 90% confidence interval, but for all other heuristics *H*, *VFF(H)* is significantly better at the 90% confidence level . These results provide good evidence that the Variable Fitness Function can be used to enhance a simple local search without using additional problem knowledge or modification of the search technique.

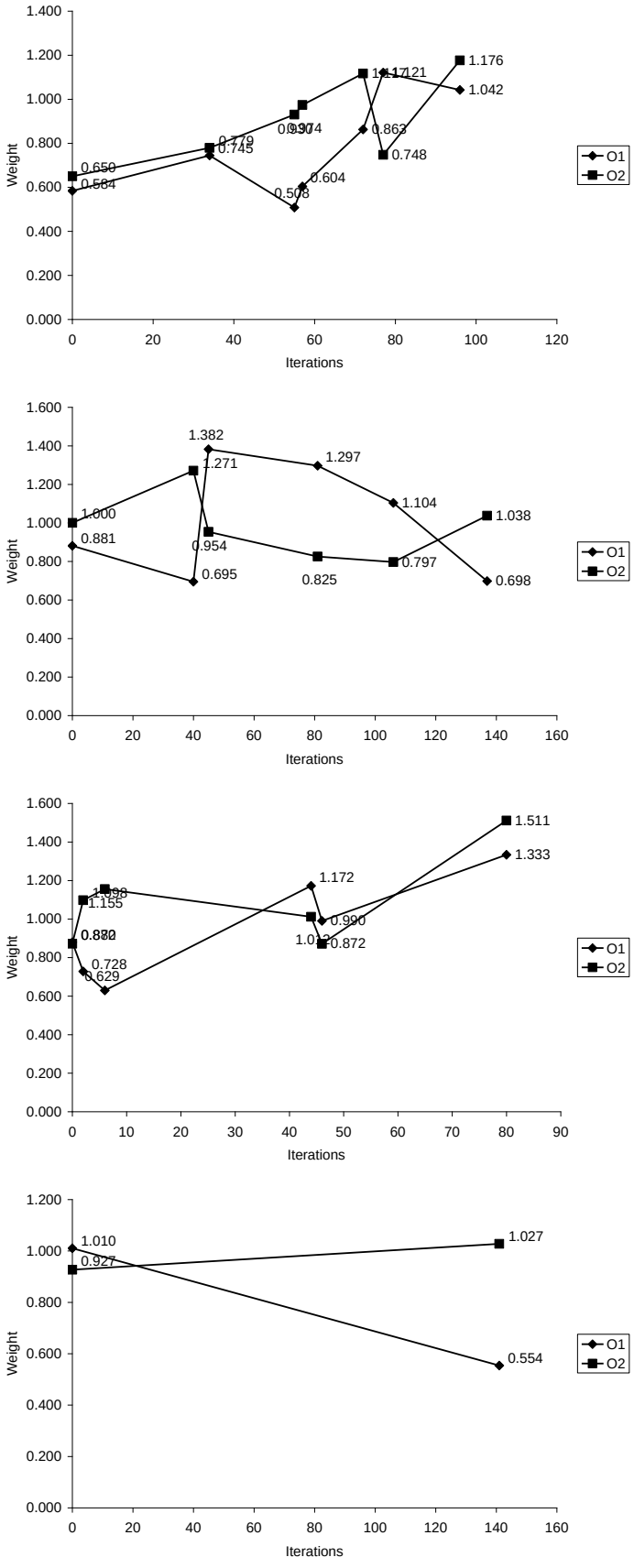


Figure 6.3. Sample of the best Variable Fitness Functions evolved for the *VFF(NN + 2Opt)* heuristic for different MO-TSP problems.

Figure 6.3 show a sample of the best Variable Fitness Functions evolved for different MO-TSP problem instances. They are quite different and have very little in common. Each one is quite different because the problem instances have nothing in common. This is as expected because the objectives were generated randomly and uncorrelated, and we observe less heterogeneity for the two other problems which we study in this chapter, where objectives are correlated. This implies, not surprisingly, that there is not a single good Variable Fitness Function for this set of uncorrelated MO-TSP problem instances. This can be seen where the weights of Variable Fitness Function change priority (for example at approximately iteration 75 of the first graph). This could be because the search has reached a local optimum with respect to one objective and changing the direction toward optimizing the other objective avoids or escapes it.

To show that the evolved Variable Fitness Functions exploit the characteristics of the search operators acting on individual problem instances' characteristics rather than the characteristics of the TSP itself, we used each of the evolved Variable Fitness Functions for *VFF(NN + 2opt)* on the other problem instances for which it was not evolved. Figure 6.4 shows this comparison and indicates that using an incorrect Variable Fitness Function is worse than using the global fitness function (comparing *Mismatched VFF(NN + 2 opt)* to *NN + 2 opt*). Lack of correlation between problem instances means that this is not possible, but for related instances, this may be possible. For problems where there is significant correlation between the objectives in different instances, a VFF generated using historical problem instances is likely to show good performance on future problem instances as with the scheduling case study presented in this chapter.

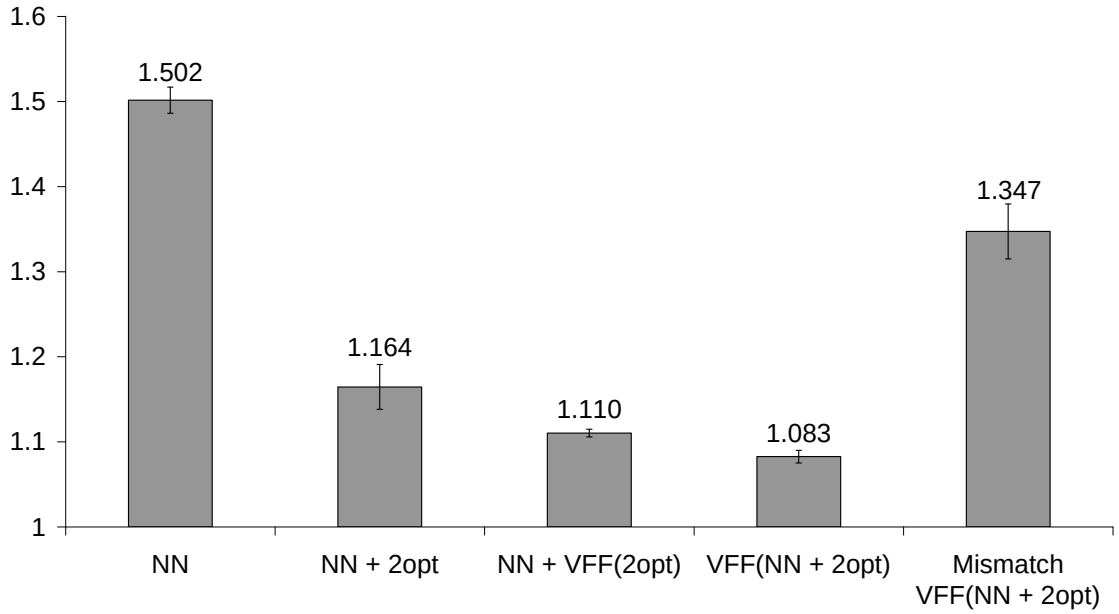


Figure 6.4. ***Mismatched VFF(NN + 2 opt)*** **represents the average deviation from the optimal solution when using Variable Fitness Function evolved for other problem instances. 90% confidence intervals averaged over 25 runs)**

Figure 6.5 shows a visualization of the search process of a Variable Fitness Function for an *AR + VFF(2opt)* search. The top plot shows the evolved Variable Fitness Function and the plot below it shows the behaviour during a search. *Current Solution Fitness* shows the fitness of the solution at each iteration (as measured by the global fitness function). *Moves to Local Optimum* and *Local Optimum Fitness* show the number of 2-opt moves the current solution is to the 2-opt local optimum that would be found if the weights were fixed (at the VFF values) and the fitness of that local optimum. When the *Moves to Local Optima* reaches zero the search is at a local optimum. When it increases after being zero it has changed search direction and escaped a local optimum. While *Local Optimum Fitness* stays constant, the search is heading for a fixed local optimum, and when it changes it has changed direction. When the *Local Optimum Fitness* is worse than the *Current Solution Fitness* the search is heading in a non intuitive way (in terms of the global fitness function), away from a local optimum.

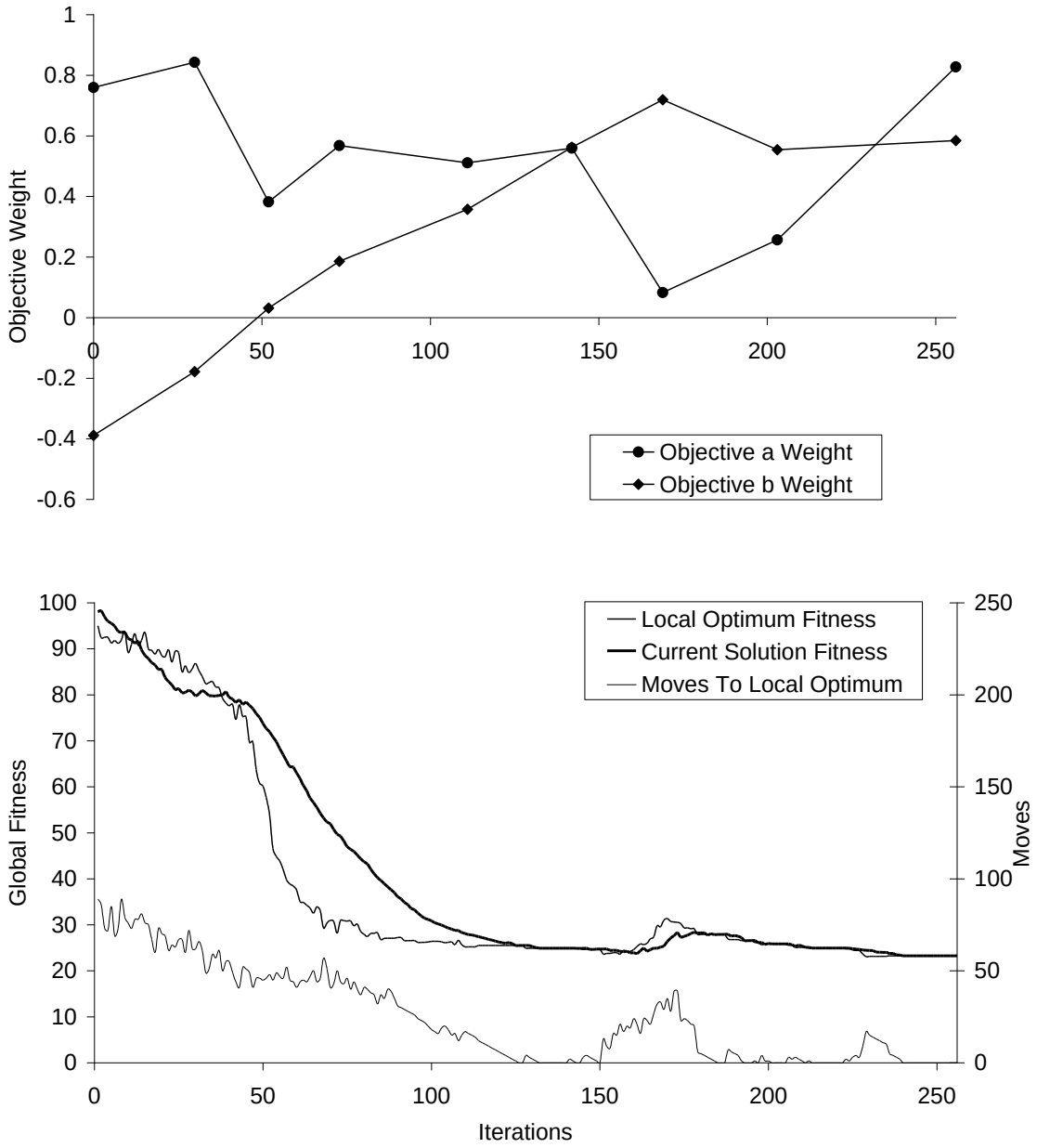


Figure 6.5. Visualising the search. Fitness uses the right axis and are measured in term of the global fitness function.

From Figure 6.5 we see that the search is both escaping local optima and changing direction to avoid local optima throughout the evolution of the VFF. Until a good local optimum is reached the *Moves to Local Optimum* stays high, keeping the search away from poor local optima (iterations 1-120). After a good local optimum is reached, the Moves to Local Optimum is increased by more radical changes in the objective weights (iterations 150-170). During the beginning of the search the *Current Solution Fitness*

does not improve much and the *Local Optimum Fitness* varies a lot. This appears to indicate that search is moving to a different area of the search space, with respect to both fitness functions (iterations 1-70). We then see a period of intensification where there is a large improvement in the solution quality and the *Local Optimum Fitness* varies less and the *Moves to Local Optimum* steadily decreases indicating it is heading toward the same local optima (iterations 50-120) At around iteration 140, the search has reached a local optima for both the VFF and the global fitness function after which, we see a change in the priority of weights in the Variable Fitness Function which leads the search to another, slightly better, local optimum at around iteration 200. During the period of "diversification" we can see that the *Local Optimum Fitness* is worse than the *Current Solution Fitness* (iterations 16-170). This is because the objective *a weight* has become very small and the search is probably pushing toward the other objective at the expense of this objective.

6.2. Application to the Virus Board Game

Virus is a two player zero-sum board game of complete information (Cowling, 2005). In previous work (e.g. (Kendall and Whitwell, 2001) for Chess and (Lucas and Runarsson, 2006) for Othello), the weights of a board evaluation function have been tuned using evolutionary techniques. It is intuitive that the ideal tactics at the beginning of a game will be different from those in the middle and end of a game, although defining the precise move where each stage stops and the next one starts may be difficult. In this section we show that the Variable Fitness Function can learn a strategy which changes over the course of the game, without needing information as to the nature of the game or the approach being used to select moves, and that it is competitive against hand crafted

AI players. Since it is clear that tactics should change over the course of a game, but it is far from clear when these changes should take place, the Variable Fitness Function would seem a promising approach. This problem is quite different from the TSP problem above and the scheduling problem below, since given the very complex discontinuous and non-linear nature of the objectives, this should provide a stern test for the VFF approach.

6.2.1. Problem Description and Variable Fitness Function Usage

The virus game is played on an 8x8 square board, and was first seen in the video game "Seventh Guest" (Matthews, n.d.) (Cowling, Naveed and Hossain, 2006). The objective is to control more squares than your opponent at game end, through "infecting" your opponent's pieces with one-space "grow" moves and two-space "jump" moves. It has some similarities to the better known "Othello" (Lucas and Runarsson, 2006) (Hingston and Masek, 2007) but has a much higher branching factor. Anecdotal evidence of over 100 people that have played the game suggests that it is tactically rich, and difficult for a human to play well. See (Cowling, 2005) for a complete description of how the game is played. The Variable Fitness Function approach will be used to determine the weights of a board evaluation function used within an alpha-beta minimax search framework. The board evaluation function is made up of 18 objectives. These are formed from 9 different measures of the board from each player's perspective. The measures are described in Table 6.3 and are chosen to give a good representation of the state of play from both the player and the opposition's point of view. It should be noted that finding the "true" evaluation function is extraordinarily hard to compute (Iwata and Kasai, 1994).

Table 6.3. Virus Board game objective measures.

Measure	Description
Square Count	The number of squares a player has captured.
Safe Square Count	The number of squares the enemy can never capture.
Biggest Grow Move	The biggest amount of pieces a "grow" move can capture from the board position.
Biggest Jump Move	The biggest amount of pieces a "jump" move can capture from the board position.
Squares 1	Number of empty squares at distance 1 from the player.
Squares 2	Number of empty squares at distance 2 from the player.
Squares 3	Number of empty squares at distance 3 from the player.
Sum of Distances	Sum of the distance to all empty squares on the board.
Capture Potential	Measures how vulnerable a board position is to the player.

In (Cowling et al., 2004), Virus was used as a teaching aid for AI where 50 students were asked to write AI players for the game. Final year undergraduate and Masters students wrote a board evaluation function for a Virus player that used minimax search to a depth of 3 ply and alpha-beta pruning (Knuth and Moore., n.d.). Such players use limited CPU time, but play a strong game as judged by anecdotal evidence from human opponents. Our Variable Fitness Function player will use the same mini-max approach and will play against these hard coded players, as well as the top evolved players from the previous generations.

6.2.2. Computational Experiments

The 50 hand coded players played in a tournament to determine their relative performance. The top two are used to provide the fitness of evolved players in training and the second top two will be used to test the evolved players after training to make

sure that the evolved players are learning tactics rather than "overfitting" (Mitchell, 1997) to beat the top two players.

The individuals evolved will be played against the best two hand crafted players and the three best evolved players found from previous generations. Each of these 5 opponents will be played twice (once as the black player and once as the white player) and 3 points will be awarded for a win, 1 point for a draw and 0 for a loss, with 0.0001 points being added for each square on the board that belongs to the player at the end of the game to break ties and reward clear victory more highly than marginal victory. The fitness of an individual is the sum of its points for each of the 10 matches hence the maximum score for an individual is just over 30. After evolution, the best evolved players will be played against the third and fourth best hand coded players.

Table 6.4. Parameters used to evolve the Variable Fitness Functions for the Virus problem

Objective b.	Initial Value $L_b \ldots U_b$	Standard Deviation V_b
all	-1…1	0.1

Evaluation of an individual is time consuming taking up to 3 minutes and 20 seconds of CPU time and so the experiments will be run in parallel on 30-60 3.0 Ghz Pentium 4 machines.

Since we have not done any experimentation with the weights and the fitness function associated with this game is likely PSPACE-hard to calculate, we cannot seed the initial population, so all experiments will start from populations of randomly generated individuals. Table 6.4 shows the parameters used for the initial population generation and mutation, these values are standard values as defined earlier for use when good parameters are not known. Population sizes of 10 and 20 were tested with a

fixed 500 individual evaluations. Ten runs of the VFF approach are undertaken and an average taken.

Figure 6.6 plots the average fitness of the population in each generation over 10 runs. The fitness is further broken down to show the points obtained from games played against the two hand crafted players (the dotted line at 12 indicates 4 wins). From these graphs it can be seen that a population size of 20 appears to be slightly better, gaining, on average, 8.8 points (7.8 against the hand crafted players), although the small difference lends credibility to the lack of sensitivity to the population size. Recall that the fitness function itself is evolving as the new "best" evolved players are added. To see the effects of this we must look in detail at individual runs and in particular, the best individual of the population. The points obtained against the hand crafted players steadily increases showing that the population is evolving to beat the hand crafter players more and more.

Figure 6.7 shows a typical run of the genetic algorithm. As well as the population averages we can see the best individual's performance. From this plot it is clearer to see what is going on. Initially, the best individual in the population has scored 6 points. This means it has won 2 of its games (or possibly won one and drawn three). Generation 10 is where we first evolve a player that is able to beat both of the hand crafted players. After this we see a series of spikes. These occur when an individual is found that beats the previously best evolved players, as well as the hand-crafted ones. The Best fitness then drops back to 12 after better players are evolved because these best players are then used to test future individuals. In this way we would hope that evolved players early on a run would be beaten by later evolved players, which we explore below.

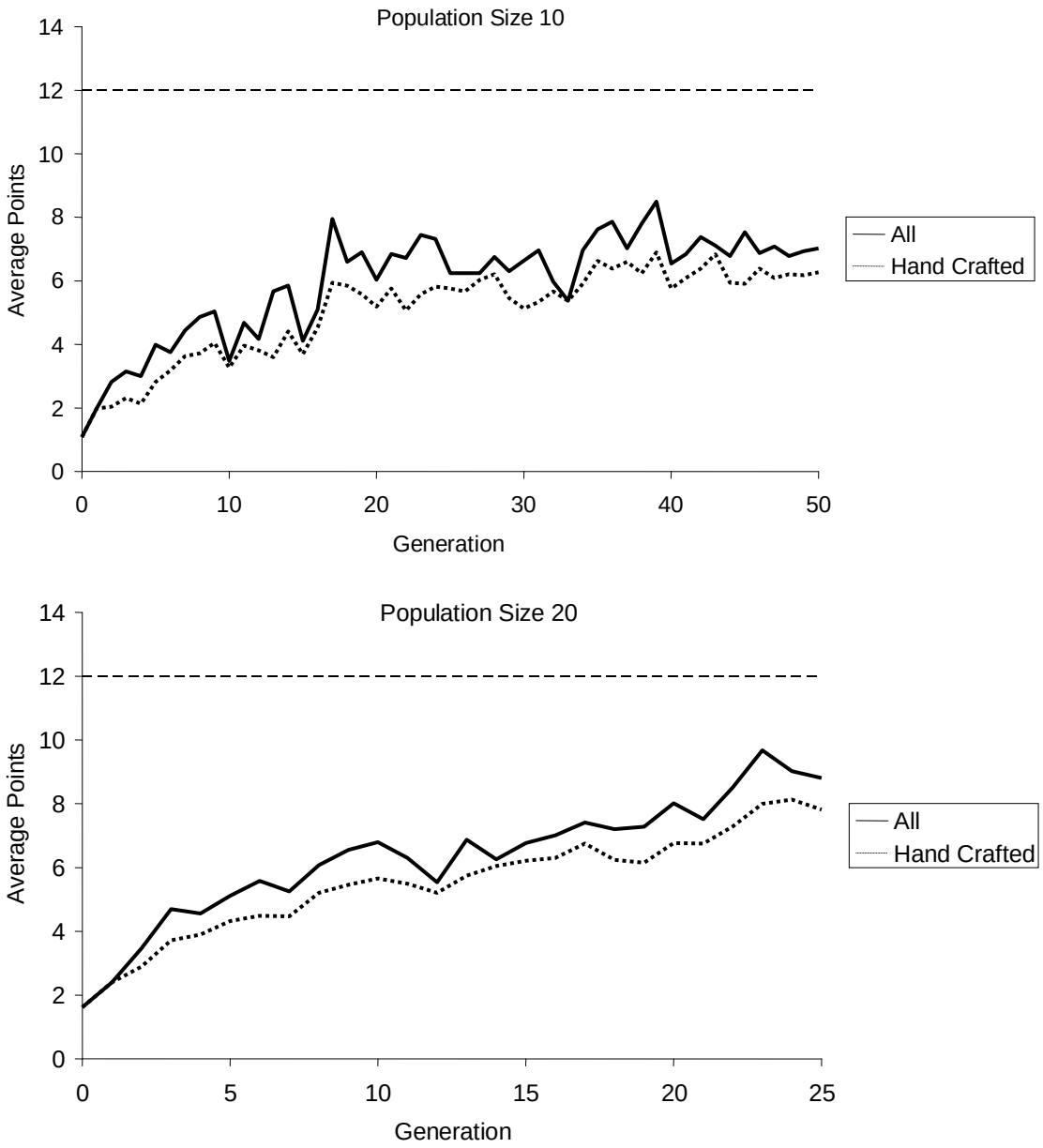


Figure 6.6. Average fitness of the population during evolution of Virus players for a population size of 10 and 20

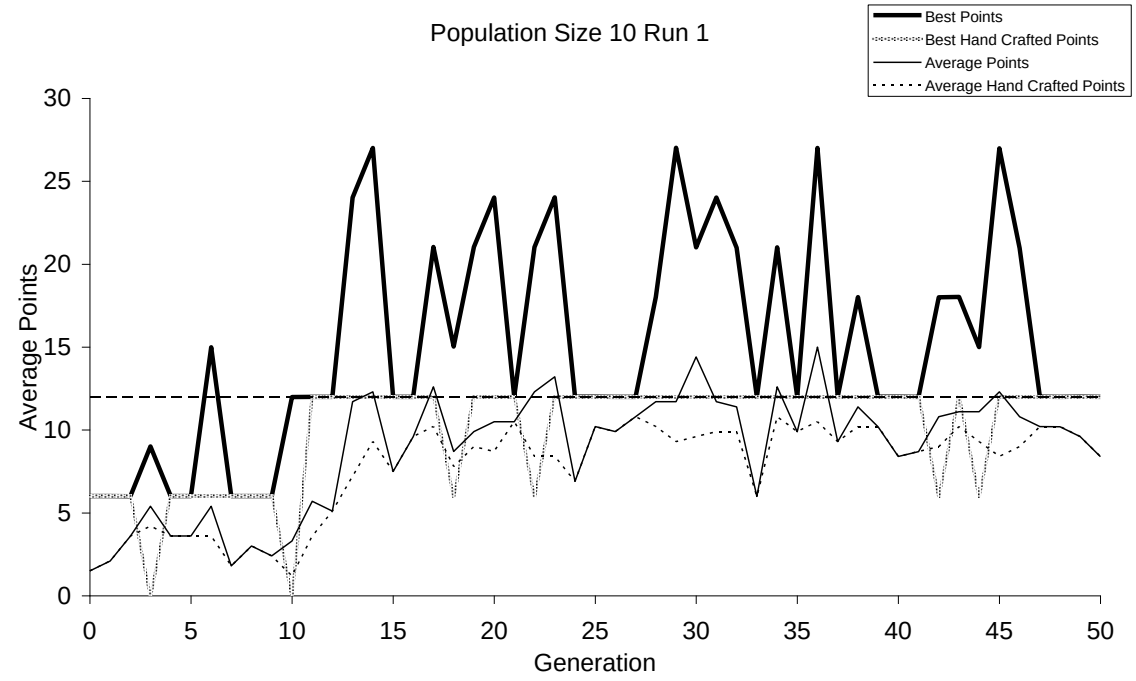


Figure 6.7. Plot of the evolution of a single run of the GA with population size 10

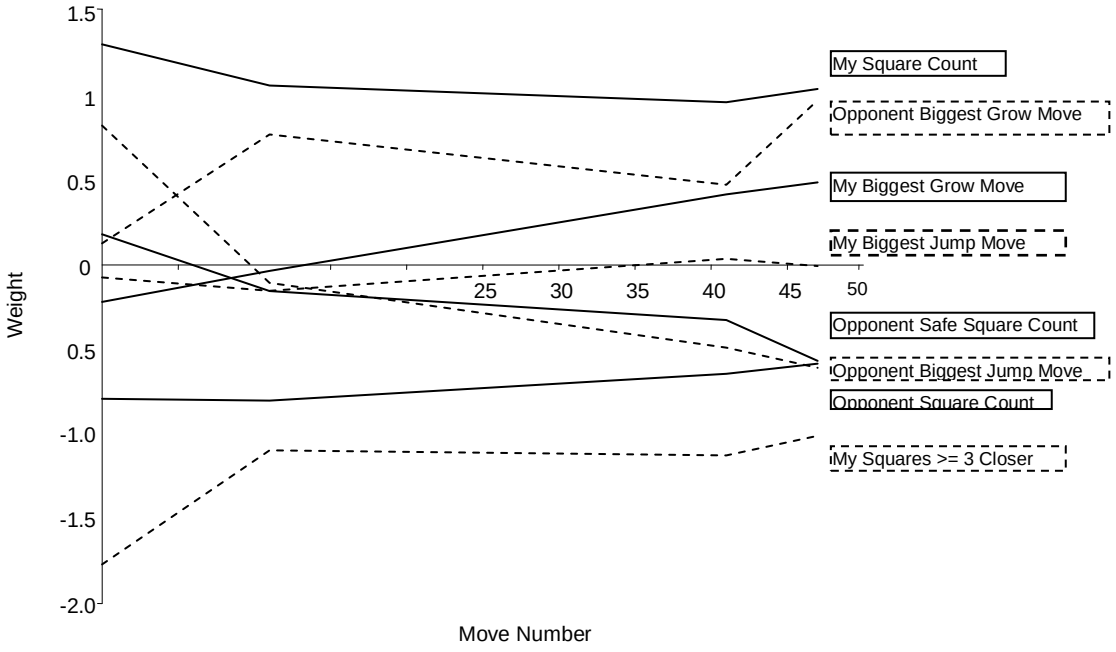


Figure 6.8. Plot of the VFF of the best individual of an evolved population of size 10.

Figure 6.8 shows the VFF plot of some of the weights of an evolved individual having high fitness. It can be seen that there are 3 stages to this tactic, one for the first ten moves, one for the middle and one for the end. Some of these weights may seem

counterintuitive, for example, throughout the VFF, Opponents Biggest Grow Move is weighted high indicating that if the opponent can make a grow move that will capture a large amount of pieces, then we should put ourselves in this position. Analysis of the actual move made may reveal that it is a good tactic to let your opponent capture some vulnerable pieces early in the game, which may be taken back later. The VFF also appears to consider "My Biggest Jump Move" to be of low importance later in the game. It is a good feature of the VFF for complex problems that fitness measures which arguably add little more than noise are given low weighting after evolution.

When playing the best evolved player of each run against the top four hand crafted players, 8 of the 10 scored 24 (eight wins), with an average score of 22.9. This shows that the evolved players were not over fitting to beat the hand crafted players they were trained against and the effect of playing against the previously best found makes the evolved players good in general, managing to also beat the 3rd and 4th best hand crafted players.

6.3. Application to the Workforce Scheduling Problem

Next we study a large, complex real-world scheduling problem as studied in (Cowling et al., 2006) (Remde et al., 2007). For this problem the Variable Fitness Function uses objectives other than those defined in the global fitness function (provided to us by the problem owners) to find a better solution. We investigate whether the Variable Fitness Function evolved offline using training instance data may be used online for unseen test instances. This seems possible, and even likely in this case, due to the common features

of each problem instance and the similarities in the complex interactions between resources (including time) when finding a solution to a problem.

When building a schedule many different and often contradictory business objectives are possible. In this section we consider three objectives. The first objective is Schedule Priority (*SP*). Maximizing Schedule Priority maximizes the value of the tasks scheduled (and implicitly minimizes the value of tasks unscheduled). The second objective measures Travel Time (*TT*) across all resources. The third objective measures the inconvenience associated with completing tasks or using resources at an inconvenient time, which we have labeled Schedule Cost (*SC*). Other objectives are possible but these three objectives express most of the primary concerns of the users in this case, at a high level, which would be suitable for global optimization. We also introduce other measures below which are used within the VFF for this problem, but which would not be appropriate to include in a global fitness function.

6.3.1. Variable Fitness Function Usage

The global fitness function we use to assess the fitness of a complete solution is $f = SP$, where *SP* is the sum of the priority of work done. This objective function will aim to build schedules with as many high priority tasks in it as possible. This is a good indication of the fitness of the schedule (as we are trying to maximize the number of tasks we schedule), but when assessing an incomplete schedule, or the change a move in a local search will make, it may not be as effective in the long run. A single immutable fitness function is really most suited to the case where there is an algorithm guaranteed to find the optimum, which is not the case here, or for the majority of large, complex problems.

Other measures which are considered are: *SC*, the sum of the costs associated with conducting tasks and using resources at particular times in the schedule and *TT*, the sum of the travel time on the schedule. We also introduce a metric which tries to estimate the difficulty of scheduling the remaining tasks by analyzing bottlenecks. The shifting bottleneck heuristic (Adams, Balas and Zawack, 1988) improves a solution by identifying the bottleneck resource and reordering work to relieve the bottleneck, repeating until no more improvements can be made. In our approach, we will keep track of the supply and demand for each skill, in order to identify potential bottlenecks during the solution construction process. Table 6.5 shows an example of how this approach helps to identify bottlenecks.

Table 6.5. Example Skill supply and demand

Skill	Supply	Demand	Difference
1	100	50	50
2	100	100	0
3	100	150	-50

From the example, we can see that although there is an equal supply of all skills, skill 3 is the most constrained as there is more demand than supply. When comparing two supply and demand tables (in effect two potential solutions), we look for the minimum value of (demand-supply) over all skills (in our example -50 for skill 3). Solutions are ranked according to the lowest difference. To break ties we would go to the second lowest difference and so on.

Encapsulating bottleneck information into a single number appropriate for a fitness function would be impractical and so we will use the above bottleneck measure to compare solutions or moves that are of similar quality when assessed by the other objectives. We introduce ε, a value which specifies how fit a solution or move must be

compared to the fittest of the set being considered. These "good enough" solutions or moves are then judged using the supply and demand tables. For example, in a local search we may have ten moves available of which the best move may make a +100 change to the fitness function. If ε was 0.9 (moves have to be within 10% of the best move to be considered), then out of the 10 solutions, only those which made a +90 or better improvement to the fitness of a solution would be considered, and the supply and demand tables used to determine the best one in this *ε-good* set. Similar *ε-good* ideas are used in reinforcement (Reinforcement Learning: An Introduction, 1998) learning and in peckish (Corne and Ross, 1995) heuristics. The value of ε provides a fourth measure to be evolved by the VFF approach, along with SP, SC and TT objective measures.

We used the greedy hyperheuristic to solve different instances with varying values of ε. The results shown in Figure 6.9 show that there may be a "magic number" for ε around 0.995, where the average fitness increased slightly, however this is likely to be just noise. It seems likely that a fixed value of ε does not yield the best solutions for a given heuristic/problem instance pair as intuitively, potential bottlenecks are more important near the beginning of the search than at the end.

For this problem, we now have four parameters in our Variable Fitness Function: the weight of SP, SC, TT and the value of ε. The Variable Fitness Function will be used with the greedy improvement only hyperheuristic used previously. Table 6.6 shows the weights of the weighted sum fitness function, as determined by problem experts, and using the simple "Priority Only" fitness measure.

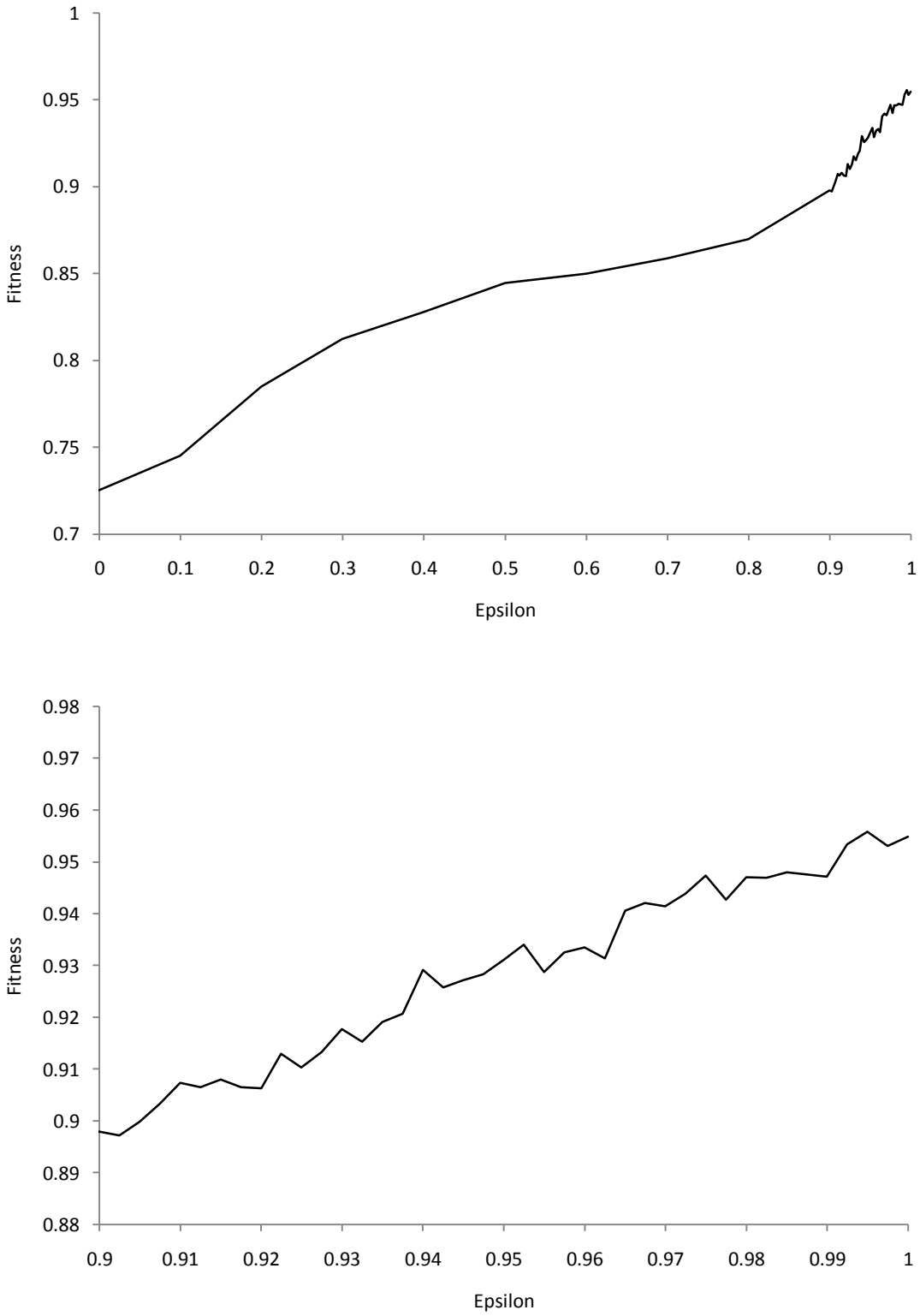


Figure 6.9. Parameter tuning experiments for epsilon (ε). Plots show various values of epsilon and the corresponding average relative fitness of 10 runs compared to not using epsilon.

Table 6.6. Weighted Sum Fitness weights.

Objective *b*	Global Fitness Weight	Expert's Fixed Weights
1 (SP)	1	1
2 (SC)	0	-4
3 (TT)	0	-2
4 (ε)	0	n/a

Table 6.7. Normalized Variable Fitness Function evolution parameters.

Objective $b_.$	Initial Value $L_b...U_b$	Standard Deviation V_b
1 (SP)	-0.25 … 0.25	0.025
2 (SC)	-1 … 1	0.1
3 (TT)	-0.5 … 0.5	0.05
4 (ε)	0 … 1	0.1

Table 6.7 shows the variables used for the evolution of the Variable Fitness Function. These values were generated automatically using the method outlined in chapter 5. ε is clamped between 0 and 1 during VFF evolution.

6.3.2. Computational Experiments

The Variable Fitness Function was evolved on 5 problem instances (the training data). These represent 5 examples of what an individual workforce may experience from day to day. Schedule quality was assessed by the sum of the fitness measured by the global fitness function (*f=SP*). The five problem instances require the scheduling of 400 tasks using 100 resources over one day using five different skills. Tasks require between one and three skills and resources possess between one and five skills. The problems are made to reflect appointment based problems Trimble MRM have identified and are

generated using a problem generator developed in collaboration with Trimble MRM. These problems are different from the ones in previous chapters. The evolved Variable Fitness Function was not only compared against the global fitness function for the training data, but also against another 5 test instances to see if evolved Variable Fitness Functions could be used online for test data instances. If we can show that evolved Variable Fitness Functions work well on data it was not trained on, then we have demonstrated the ability of VFFs to learn characteristics of problems and the heuristics used to solve them. This further shows the potential of using lots of CPU time to evolve Variable Fitness Functions offline using training data, and then using the evolved VFFs on a day to day basis to improve solution quality with no overhead in terms of CPU time, and very little overhead in terms of development effort. Training runs were repeated 10 times and averages taken. Each evaluation takes about 3.5 minutes and so the experiments will take over 1 month of CPU time to complete so were run in parallel on 30-60 3.0 GHz machines, using idle cycles from a lab containing standard Windows PCs.

To tune the parameters of the GA, we tested population sizes of *10* and *20* (with a fixed 500 chromosome evaluations to ensure similar CPU time across runs), and seeded the initial populations with *0*, *1* and *All* global fitness functions. Figure 6.10 shows the results for population sizes *10* and *20*. These results show that the GA is fairly insensitive to the population size. Choosing *1* or *All* of the initial population to be seeded with the global fitness function gives results which are statistically not distinguished at the 90% confidence level, although both of these results are significantly better than no seeding (at the 90% level). Figure 6.10 also shows that the population size is not statistically significant at this level. A population size of *10* and

seeding the initial population with *1* global fitness functions was chosen for the subsequent experiments.

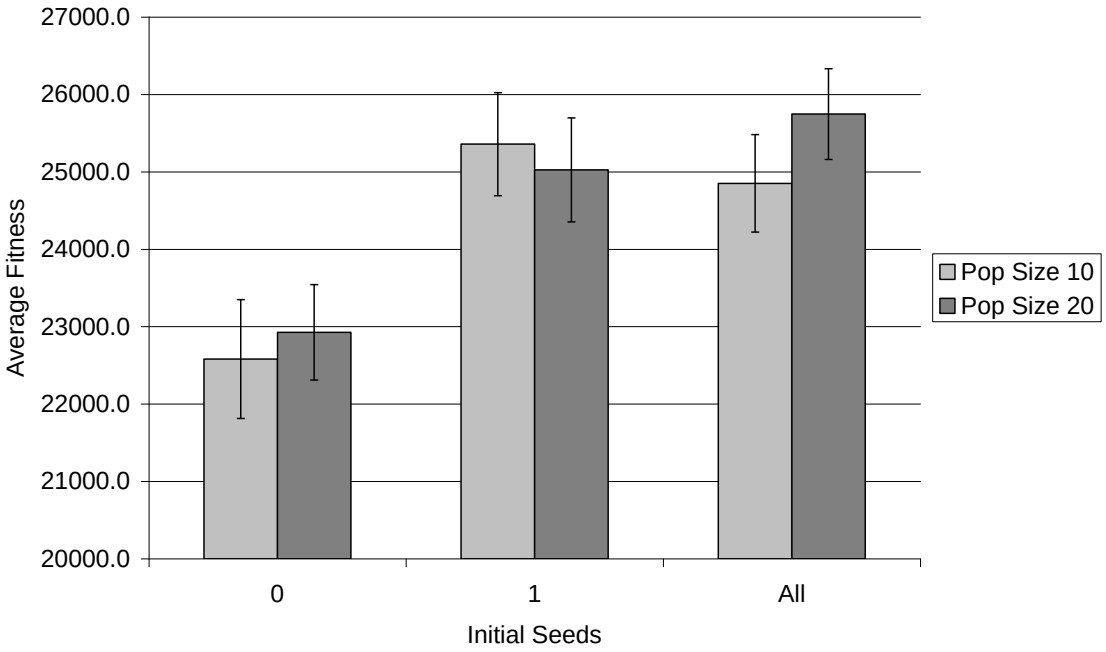


Figure 6.10. Parameter tuning on a single problem instance experimental results with 90% confidence intervals (Average of 10 runs).

We took the evolved Variable Fitness Functions from each of the 10 runs using the best parameters and compared their performance using the global fitness function on the training data and on 5 previously unseen test data instances. Also included was a "Random" heuristic, which was given the same amount of time as the Variable Fitness Function evolution. This heuristic randomly picks one of the top two moves as measured by the global fitness function at each iteration. Figure 6.11 shows the comparison. The results for the training data show that the evolved Variable Fitness Functions were very much better than the global fitness function. This VFF appears to have learned to use the other objectives to aid the search process, specifically to "tune" weights to the time-varying characteristics of the current schedule and the heuristic used to improve it, and found solutions which are on average 23.8% fitter, and which are significantly better than a randomized restart strategy given the same amount of CPU

time. Looking at the test data we can see that the VFF approach is also able to find a significantly better solution, at the 90% confidence level, which is on average 16.8% fitter. The CPU time for the original heuristic, with the "Ordinary" fitness measure, and the VFF-enhanced heuristic, are identical. It is particularly notable that when compared to the "Random" heuristic, the VFF achieves a performance which is better using $1/500^{th}$ of the CPU time.

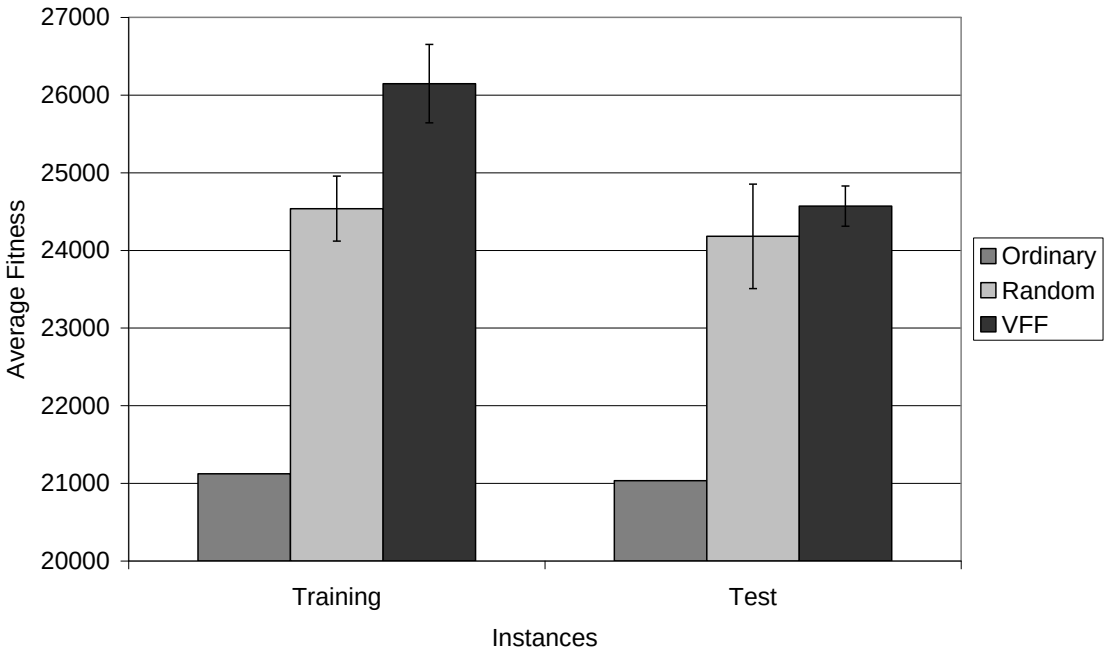


Figure 6.11. Comparison of the average fitness using the evolved Variable Fitness Functions and the global fitness function. Note that the Random heuristic on the Test data takes over 500 times as much CPU time as Ordinary or VFF approaches (average of 50 runs).

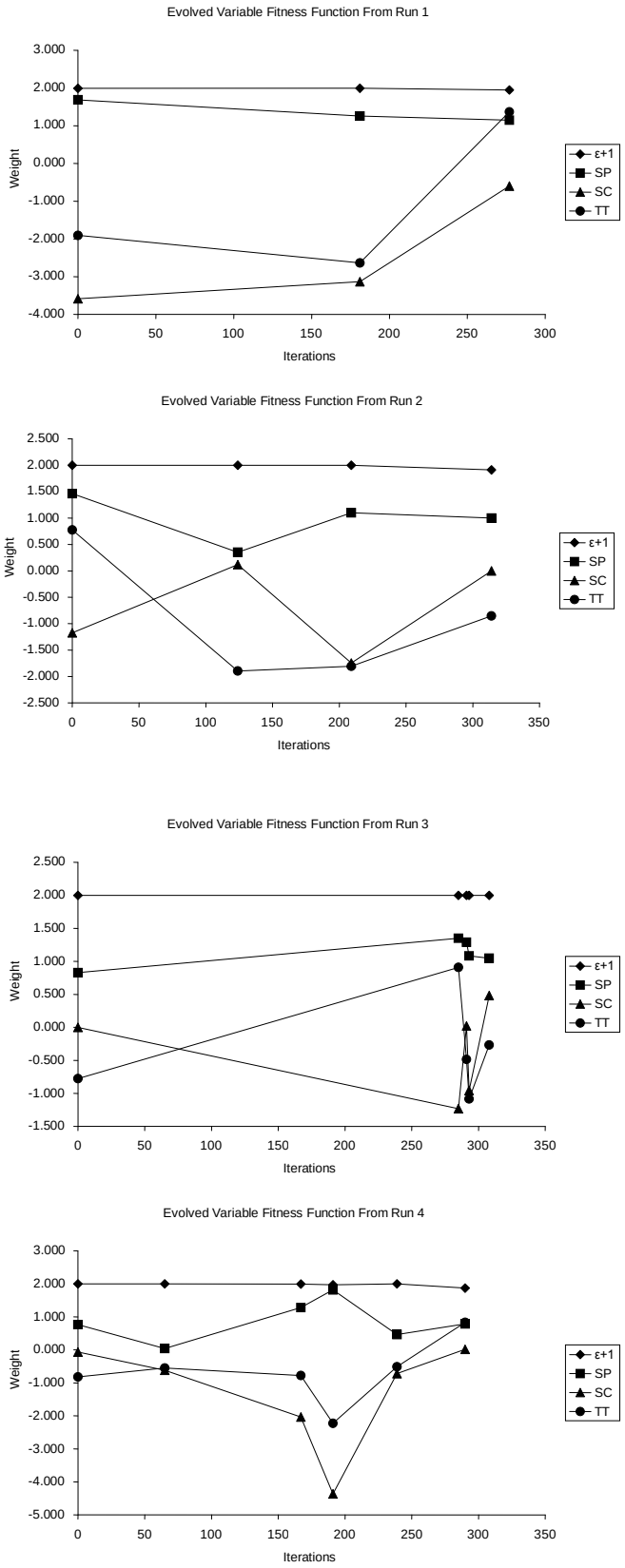


Figure 6.12. Selected best individuals from the final populations of individual runs. (ε is the bank consideration threshold, SP is the weight of sum of scheduled priority, SC is the weight of the sum of scheduled cost and TT is the weight of the travel time). ε+1 is shown to separate the line from SP.

Figure 6.12 shows a sample of the evolved fitness functions. While these show similar trends they are quite different, and their form is sufficiently unusual that it is not plausible that they might have been created simply by hand-tuning of weights. The graphs show that the Variable Fitness Function approach has learnt that ε should be around 1 throughout the search with a slight decrease toward the end. Run 1 shows quite a simple Variable Fitness Function. From this graph we can see that that TT and SC objectives are to be minimized at the start, but are far less important towards the end of search. This intuitively makes sense, as if we can minimize the costs and the travel time on the schedule, we can fit more tasks in. However, toward the end of the schedule optimization process when the schedule is fuller, we don't care as much about costs and travel as we just want to fit as many remaining tasks in as possible. Run 2 is similar to run 1 with additional detail at the end. Run 3 is interesting because toward the end we observe tight changes in weights. However, it is most likely that that part of the graph is not used and that the scheduling process has reached its final local optimum before this point. Run 4 shows quite a complex graph, but we can still see trend from the other runs: ε starts at 1 and slowly decreases, SP always stays positive and dominates SC and TT which start low and increase toward the end.

6.4. Summary

In this chapter, the Variable Fitness Function has demonstrated its ability to enhance heuristic search across three very different problem domains, the multiobjective Travelling Salesman Problem, position evaluation function for the Virus board game, and a complex, real-world workforce scheduling problem. In all cases, we have shown that given a optimization heuristic *H* and a sufficiently large quantity of CPU time we

can produce a better heuristic *VFF(H)*. Our results show that the evolved VFFs are able to exploit features of both the problem instance and the heuristic used to solve it. We have also provided evidence that the Variable Fitness Functions are learning to move the search to different parts of the search space when local optima are encountered (particularly for the multiobjective TSP). The generation of a solution using VFF takes longer due to the evolutionary process. However, if the time is available, this method appears to be better than random perturbations to a local search with multiple restarts (across the multiobjective TSP and the workforce scheduling problem) and requires no modification of the local search heuristic, unlike most common metaheuristics. The case studies showed that the Variable Fitness Function has great potential. The three quite different optimization examples we have used to show this method can be used on a wide range of problems. Moreover, our experiments show that evolution is insensitive to parameter choices, as has also been observed for the hyperheuristic methods (Chakhlevitch and Cowling, 2008), which also works at a higher level of generality, dealing with solution methods rather than directly with problems. We provide an automatic method for setting most parameters. The VFF approach is highly CPU-intensive, with the work in this section taking over 13 CPU-months. In this work we have harvested idle CPU cycles of large labs of PCs in overnight and weekend runs. We can use such a cheap source of CPU cycles in other VFF applications, especially when solving very large, complex real-world problems.

Where there exist similarities between training instances of a problem and unseen test instances (for the Virus board game and the real-world scheduling problem), the VFF has been shown to perform well on these test instances, *given no additional CPU time*. Hence we may use large numbers CPU cycles offline to yield a VFF which improves the solutions generated by a heuristic online, without modifying that heuristic

or taking additional CPU time. The performance of the VFF in the experimental case studies in this paper gives us optimism for the broad application of the VFF approach across a wide variety of scheduling and optimization problems, particularly complex, real world problems where existing systems, models and heuristics have been developed at great cost, when the VFF may yield improvements which require very little modification to existing systems, and which do not require additional CPU time.

Chapter 7

Enhancing Metaheuristics with Variable Fitness Functions

Our initial case studies look at local search and constructive heuristics. This section uses the Variable Fitness Function to enhance metaheuristic performance. A Variable Neighbourhood search for a static scheduling problem and a greedy look ahead search hyperheuristic for the dynamic scheduling problem will be studied. Studying the Variable Fitness Functions ability to enhance metaheuristics will be interesting as it removes the Variable Fitness Function further away from the search process and with a hyperheuristic it is arguably even further away.

Finally, the results from this chapter and chapter 6 will be analysed to identify what situation is required for the evolved Variable Fitness Functions to generalise across different problem instances.

7.1. Variable Neighbourhood Search for A Complex Workforce Scheduling Problem

In this section we study the real world workforce scheduling problem and apply constructive search and variable neighbourhood search (VNS) metaheuristics and enhance these methods by using a Variable Fitness Function. The Variable Fitness Function (VFF) uses an evolutionary approach to evolve weights for each of the (multiple) objectives. We show that the VFF significantly improves performance of constructive and VNS approaches on training problems, and "learns" features of problems, problem instances and heuristics used to solve them which enhance the performance on unseen test problem instances. Using real problems requires large amounts of CPU however this is justified as this gives the results plausibility of applicability.

7.1.1. Variable Fitness Function Application

In the problem instances we study, 10 resources possess between 1 and 5 skills of which there are various bottlenecks in the availability. The resources travel at varying speeds. There are 300 tasks requiring between 0.5 and 1 hours to complete to be scheduled over 3 days and each has a 4 hour time window in which it must be completed. A task requires a resource to possess a certain skill, of which some skills are in more demand than others. Tasks are to be completed as early in the 4 hour time window as possible. Tasks have precedence constraints such that some tasks may not be started before another has completed. A chain of tasks is a subgraph of the precedence digraph of maximum indegree 1 where the indegree (outdegree) of a task in the subgraph is one if the indegree (outdegree) in the precedence digraph is greater than zero. This extra

constraint makes the problem similar to one Trimble encounter. This arguably makes the problem easier as smaller precedence trees are constructed and adds a new objective we can measure (completed chains).

For this real world complex problem there are many objectives. Table 7.1 lists some of the principal objectives we have identified. The global fitness function is defined as *f = 5 (Scheduled High) + 2 (Scheduled Low) + (Complete Chains) – 0.1 (Overrun)*, following reflection with our industrial collaborator. This objective function rewards the scheduling of high priority tasks more than low priority tasks and also the completion of chains of tasks whilst trying to minimise the amount of overrun (and hence inconvenience to the clients) and ignoring travel costs.

Table 7.1. Objectives used for the workforce scheduling problem.

Objective	Description
Scheduled High	The number of high priority tasks scheduled.
Scheduled Low	The number of low priority tasks scheduled.
Complete Chains	The number of task chains that have been completed.
Travel Distance	The total distance travelled by the resources.
Travel Time	The total time spent travelling by the resources
Overrun	The total number of hours the task have overrun.

We use a constructive heuristic, *CON*, to build an initial schedule then an improvement metaheuristic, *IMP*, to improve it. For the improvement heuristic we have decided to use Variable Neighbourhood Search (VNS). VNS is relatively simple to implement and we have seen that this kind of method can work well for scheduling problems (Remde et al., 2007). We use a local search heuristic where we define the neighbourhood as schedules which result from optimally reinserting a task, i.e. placing a task optimally in the schedule in terms of the current (variable) fitness function. If the task is not yet scheduled, this means allocating the resources and time to it that yield the best

improvement in fitness. If the task is already scheduled, this may mean moving the task in time, allocating new resources or a combination of the two (Figure 7.1). At each iteration of the local search, the entire neighbourhood is sampled and the best solution accepted. When the local search of the VNS reaches a local optimum, the search is kicked into a new area of the search space. We define these kicks as removing between 1 and 4 scheduled tasks chosen at random and all their dependent tasks. We remove dependent tasks so that precedence constraints are not broken. The pseudo code is given in Figure 7.2.

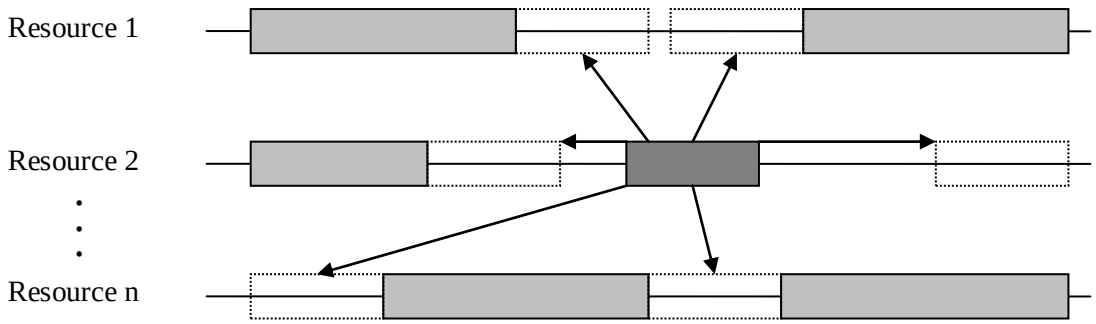


Figure 7.1. Task reinsertion. The task is moved to the resource and time in the schedule which provides the best change in fitness according to the Variable Fitness Function. The light grey boxes represent other tasks, the dark grey is the task being optimized and the dotted boxes are the positions being considered. For simplicity this process is illustrated for a task requiring only one skill.

Table 7.2. Various methods to be used and their VFF enhanced versions.

Heuristics	Description
CON	Construction heuristic using the global fitness function.
CON + IMP	Construction heuristic and improvement heuristic using the global fitness function.
VFF(CON)	Construction heuristic using a Variable Fitness Function.
VFF(CON + IMP)	Construction heuristic and improvement heuristic using a Variable Fitness Function.
CON + VFF(IMP)	Construction heuristic using the global fitness function and improvement heuristic using a Variable Fitness Function.

The construction method, *CON*, uses the local search operator of the VNS and terminates when a local optimum is found. The improvement metaheuristic, *IMP*, has a stopping criterion of 10,000 iterations as this gave a good trade off between CPU time and solution quality. Table 7.2 lists the methods we will try. We can see the first two are normal heuristics and the last three are enhanced using VFF.

```
k:=1 - the kick magnitude
s : Solution – initially empty

while (k<5)
        s' := LocalSearch(s)

        if improvement found then
                k := 1
                s := s'
        else
                k := k+1
                Remove k tasks and dependants from s
        end if
end while
```

Figure 7.2. VNS Pseudo Code

7.1.2. Computational Experiments

The five methods are used ten times on each of the five training instances and averages taken to produce statistically significant results. These five training instances are chosen

to contain variations that a workforce would see on a day to day basis following consultation with Trimble. By using multiple problem instances to evolve Variable Fitness Functions we are trying to ensure that Variable Fitness Functions learn characteristics of the problem through learning multiple problem instances. For the methods enhanced with the Variable Fitness Function, a test set of five previously unseen problems instances will be solved using VFFs which were evolved using the training instances. Good performance on the test data will imply that a lot of CPU time could be used to train a "general purpose" Variable Fitness Function, then that Variable Fitness Function could be used very quickly in "real time".

The methods requiring the evolution of a Variable Fitness Function will be given 50 generations with a population size of 10 (equivalent to 500 evaluations as we have seen this works well in the previous chapter). Methods without a Variable Fitness Function will also be given 500 evaluations and the best one taken to give them the same amount of CPU time. The *CON* heuristic takes approximately 30 seconds to construct the five schedules and the *IMP* heuristic takes approximately 25 minutes on a 3.0 GHz PC. As these experiments shall take over 260 CPU days to complete they will be run in parallel on approximately 95 computers.

Table 7.3 shows the results of the individual methods and their standard deviations and Figure 7.3 graphs these with 90% confidence intervals. From the results we can see that the Variable Fitness Functions were indeed able to enhance the standard methods significantly in all cases. We see very large variations in fitness when the Variable Fitness Function is used on the constructive part of the search (*VFF(CON)* and *VFF(CON + IMP)*). Detailed investigation of individual runs leads us to believe that this is because when the Variable Fitness Function affects the constructive part of the search, it has the possibility to move a great distance in the search space from the

constructive algorithm using fixed fitness function weights. The best Variable Fitness Function enhancement for the *CON + IMP* method was to just enhance the improvement part (CON + VFF(IMP)). The variation of *CON + IMP* is extremely low and the solutions that it has found are far from optimal. This may be because the random kick method of the VNS we have chosen was not sufficiently disruptive however a more disruptive kick mechanism could have a negative effect.

Table 7.3. Average fitness and standard deviation of ten runs of each method assessed using the global fitness function.

Method	Average Fitness	Standard Deviation
CON	3189.6774	N/A
VFF(CON)	3308.0253	150.3236
CON + IMP	3617.4517	8.4949
VFF(CON + IMP)	3689.9001	111.2555
CON + VFF(IMP)	3770.2434	35.4282

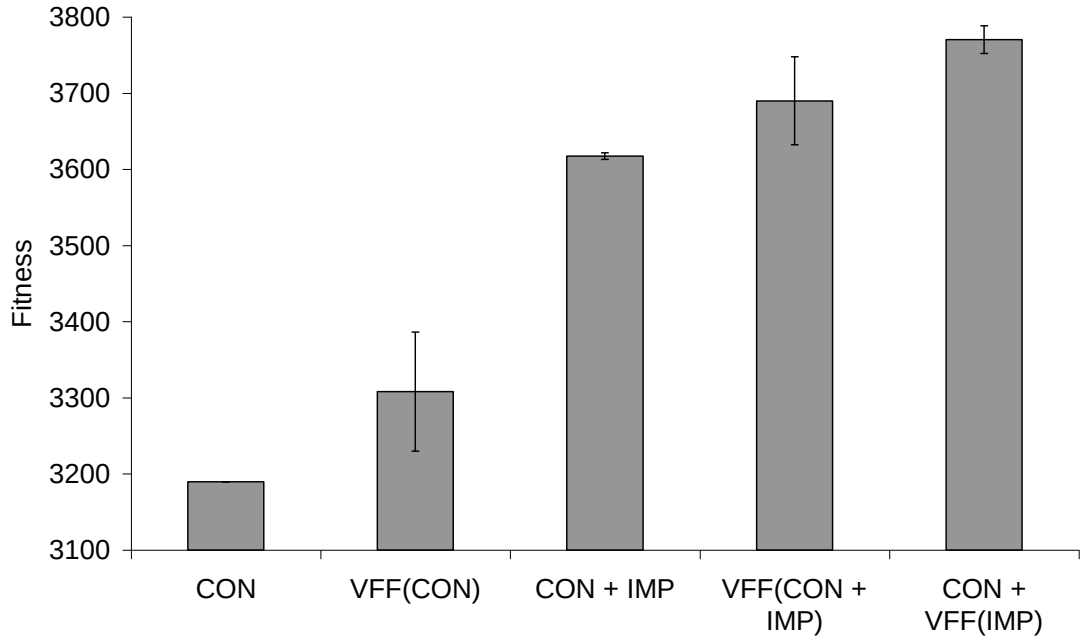


Figure 7.3. Graph of the results in Table 7.3 with 90% confidence intervals averaged over 10 runs.

Figure 7.4 shows the breakdown of the individual objectives and Table 7.4 shows the difference in the average objective measures the enhanced methods have produced. This chart and table show that the VFF approach has found a way to obtain improvements to high priority (highly weighted) objectives at the expense of low priority ones (low or zero weight).

In all of the cases where the method is enhanced by the Variable Fitness Function, the number of scheduled tasks, both low priority and high priority, has increased. This intuitively makes sense as these are the highest weighted objectives in the global fitness function. Travel time was decreased in both the cases where the Variable Fitness Function was used to enhance the metaheuristic. The increase in travel time and other penalty objectives for the CON approach is not surprising as CON has no way to optimize these objectives by reinserting. Travel time is not included in the global fitness function, however, it would appear that when task reinsertion is permitted, the VFFs have learnt that less time spent travelling means more time can be spent doing tasks showing that the VFF is capable of using objectives which go not directly affect the quality of the solution or which were deemed low importance by the expert. In all cases, overrun increased, indicating that tasks were not scheduled as close to their start time as possible.

Table 7.4. Average change in objectives as a result of Variable Fitness Function enhancement over the 25 problem instances.

Base Method	CON	CON + IMP	
Improvement Using	VFF(CON)	VFF(CON + IMP)	CON + VFF(IMP)
Scheduled High (max)	20.20	4.30	17.10
Scheduled Low (max)	40.50	37.10	43.20
Travel Distance (min)	13.79	-24.03	-67.69
Travel Time (min)	14.91	-24.85	-64.34
Overrun (min)	603.52	220.51	203.08
Complete Chains (max)	-3.30	-1.20	1.20

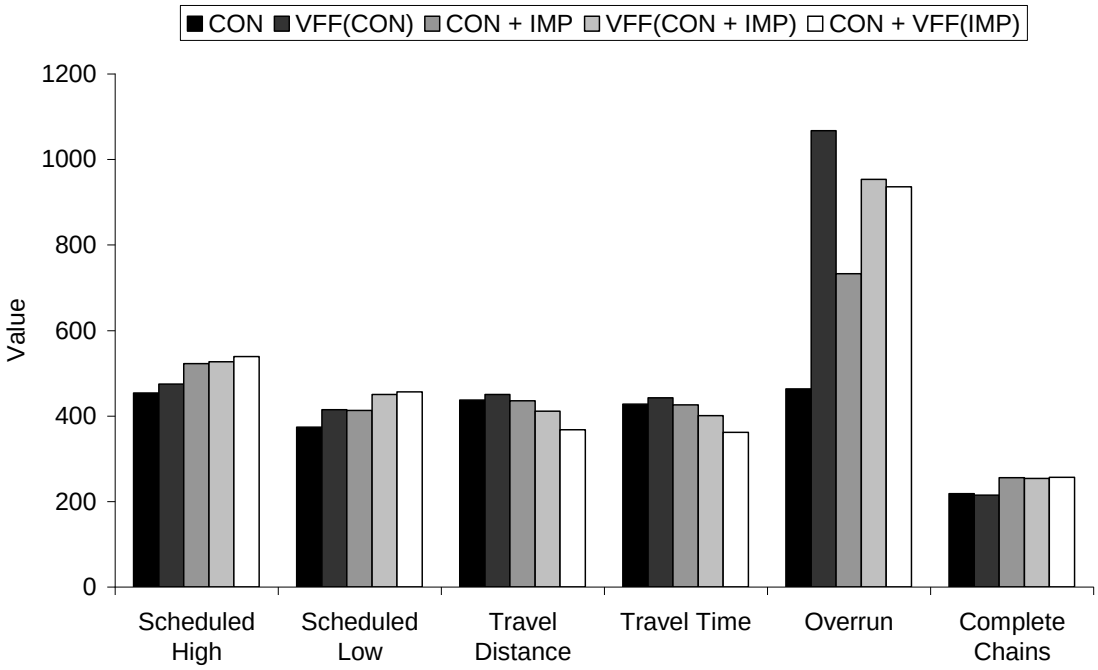


Figure 7.4. Individual objective breakdown for each method. Note: *f = 5 (Scheduled High) + 2 (Scheduled Low) + (Complete Chains) – 0.1 (Overrun)* so Scheduled High, Scheduled Low and Completed Chains are to be maximized, Overrun is to be minimized and the others are not considered when evaluating fitness.

Figure 7.4 shows the evolution process in action. Not only do these graphs show that the evolution process is working, and that the populations are evolving VFFs that lead to better schedules, but it shows the difference between randomly generated Variable Fitness Functions (those in the initial population at generation 0) and evolved ones (those in the final population at generation 50). The plot showing the evolution of *VFF(CON)* method shows a greater increase in fitness from random Variable Fitness Functions to evolved Variable Fitness Functions than that of *CON + VFF(IMP)* (note the difference in "fitness" scale between the graphs). This is because the *CON + IMP* is a better method than *CON*, and hence there is less room for improvement.

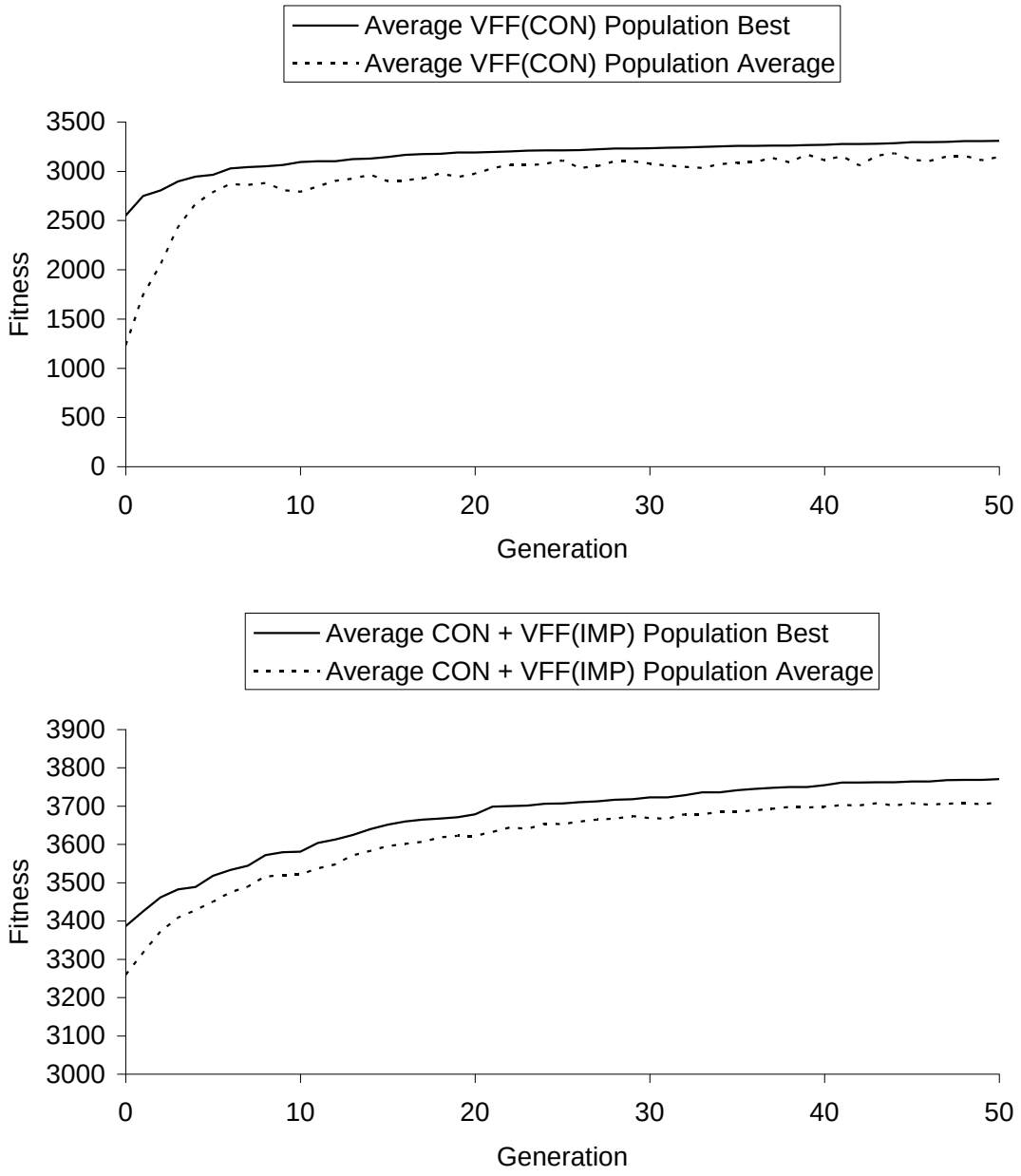


Figure 7.5. Average population fitness and best of the population's fitness at each generation showing the evolution for *VFF (CON)* and *CON + VFF(IMP)* methods.

Figure 7.6 shows an example of how the Variable Fitness Function is working. The top plot shows a typical (good quality) evolved Variable Fitness Function from the final evolved population and how the objective weights change over the iterations when using this VFF. The two very interesting objectives that are highlighted are Travel Time and Overrun. Plotted below are the objective values obtained at each iteration from a

single run using this Variable Fitness Function. A correlation can be seen between the weight of overrun and the average overrun observed. When the weight is positive, overrun increases and when the weight is negative it decreases. This Variable Fitness Function has in fact learnt a type of right-left shift heuristic (Valls, Ballestin and Quintanilla, 2003), which is frequently used in schedule repair. This is a schedule improvement heuristic which works by shifting tasks to the right and then packing them as early as possible. We can see toward the end of this VFF, the weights of the objectives used in the global fitness function are emphasised providing one last push toward the global fitness optima.

Figure 7.7 shows the improvement gained versus the global fitness function from using the Variable Fitness Function enhanced methods over the standard methods, for both the training data and the test data. Note that for test data instances, the amount of CPU time for the VFF and standard approaches are the same. As seen in the chart, the Variable Fitness Function enhanced methods are still significantly better than their standard versions on the test data (with the exception of the *VFF(CON + IMP)* method whose 90% confidence interval takes it below 0%). This is a good indication that Variable Fitness Functions trained for the *VFF(CON)* and *CON + VFF(IMP*) could be reused on different problem instances with good performance, and that they have "learned" generalisable information about the problem as well as specific information about the training instances.

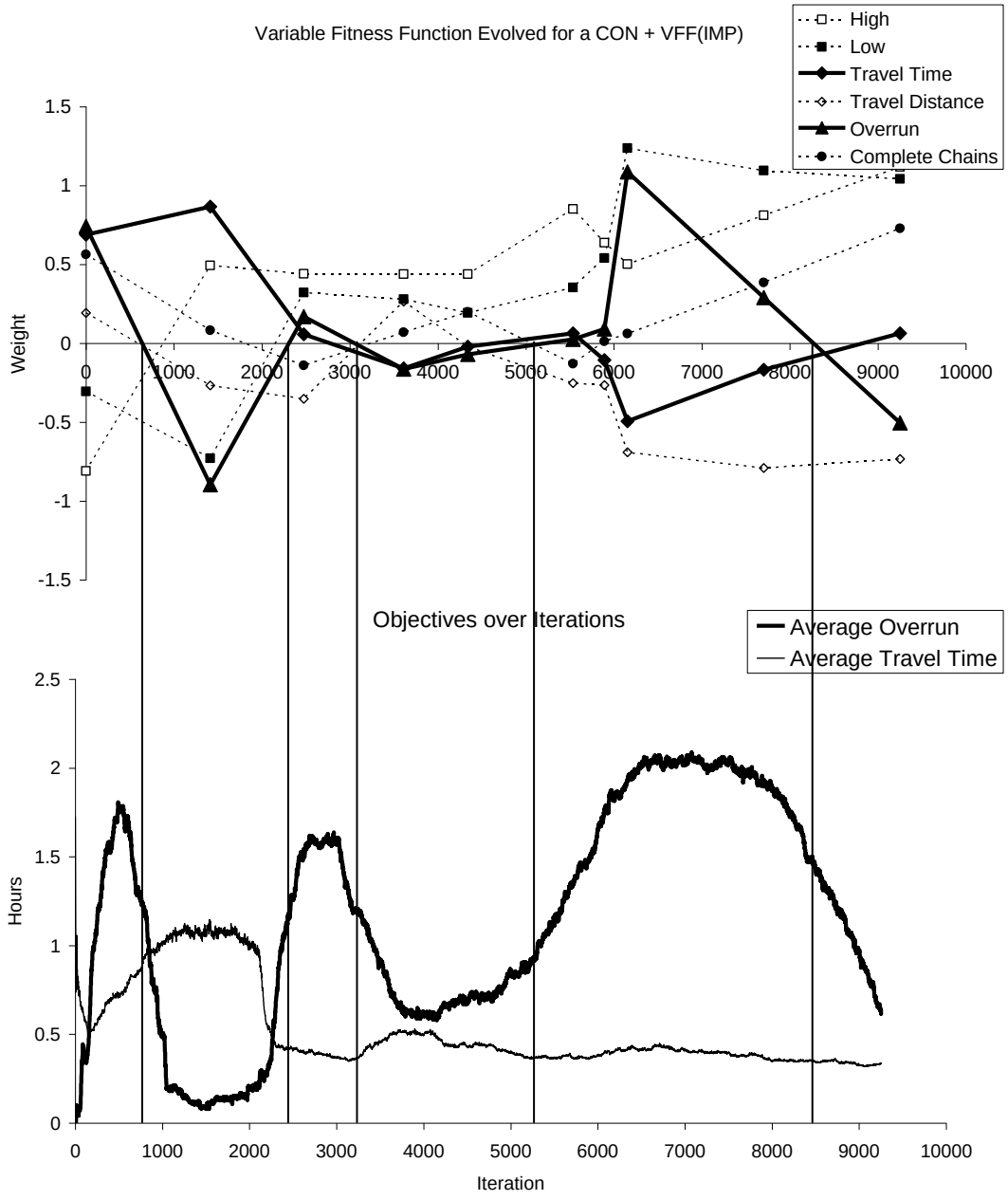


Figure 7.6. A selected evolved VFF shown above and a plot below showing how two selected objective measures change over the course of a search.

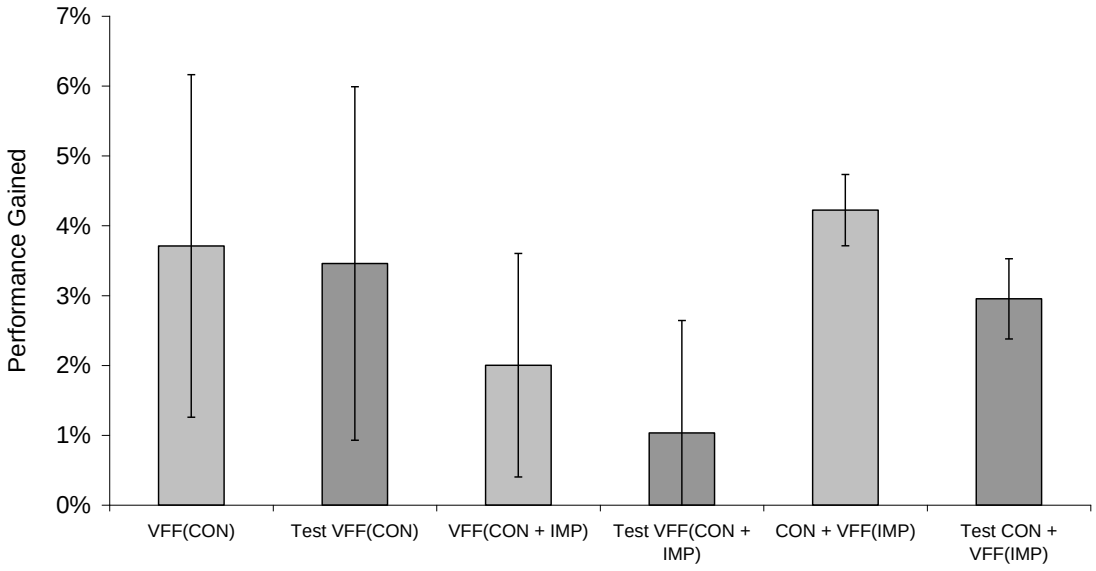


Figure 7.7. Average method performance gained using Variable Fitness Function on test data compared to training data.

7.1.3. Summary

In this section we have demonstrated the application of an evolutionary Variable Fitness Function to a constructive heuristic and a metaheuristic for a complex, real-world workforce scheduling problem. We have shown that statistically significant increases in heuristic and metaheuristic performance can be gained by using the Variable Fitness Function, for a range of test problems. We have also seen that evolution plays a key role in getting these gains. To show the reusability of the evolved Variable Fitness Functions they were used on another set of problem instances and showed gains of nearly equal magnitude. This is a strong indicator that a Variable Fitness Function could be evolved offline and then the evolved Variable Fitness Function be used in a real time situation. Arguably, the Variable Fitness Function can be used for any optimization problem where multiple objectives can be defined.

7.2. Hyper-heuristics for Dynamic Workforce Scheduling

The problem we consider consists of a schedule and a set of events that disrupt the schedule. Heuristics are used to repair a schedule after the events have disrupted it. A heuristic's fitness will be assessed by using the heuristic to repair the schedule following all the dynamic events individually on five problem instances. The events are responded to individually as the process of repairing an event changes the schedule and has knock on effect when repairing future events. This can lead to a very noisy optimisation environment making the process more difficult (Colledge, 2009). Repairing each event in isolation and then summing the changes in fitness will aid the optimisation process by eliminating a lot of the noise without the cost of CPU time.

The repair heuristics we will use are hyper heuristics using eight different low level heuristics and a greedy look-ahead search to determine which low level heuristic to use. The greedy look-ahead search creates a decision tree of certain depth where the nodes are solutions and the branches are the application of a low level heuristic. At each iteration, it takes a step toward the leaf that leads to the best outcome by applying the low level heuristic for the branch it traverses, and then rebuilds the tree back to the maximum depth. Eight low level heuristics are used as this gives a good trade off between solution quality and CPU time required (Colledge, 2009). The hyper heuristic solution has thousands of low level heuristics to choose eight from. This gives rise to a situation where we can create different quality heuristics by the selection of low level heuristics. To find "good", "ok", and "bad" sets, we tested over 10,000 different combinations of low level heuristics on five different problems. This data provided us with a method to normalise each instance, as different size problems have been used and

these can result in large variations in fitness. For each of the problem instances we found the range in fitness that these 10,000 heuristics scored. These ranges allow us to normalise the instance's fitness to the range [0 1] so we can compare a heuristic's performance between the five instances and take an average. The heuristics were ranked according to their minimum normalised fitness as we wanted heuristics that perform well across all the instances, not just well on average. "good" was chosen as the heuristic with rank number 1 (ranked 2nd on averages), "ok" was chosen as rank 33 (rank 38 on averages) and "bad" was chosen as rank 66 (rank 104 on averages).

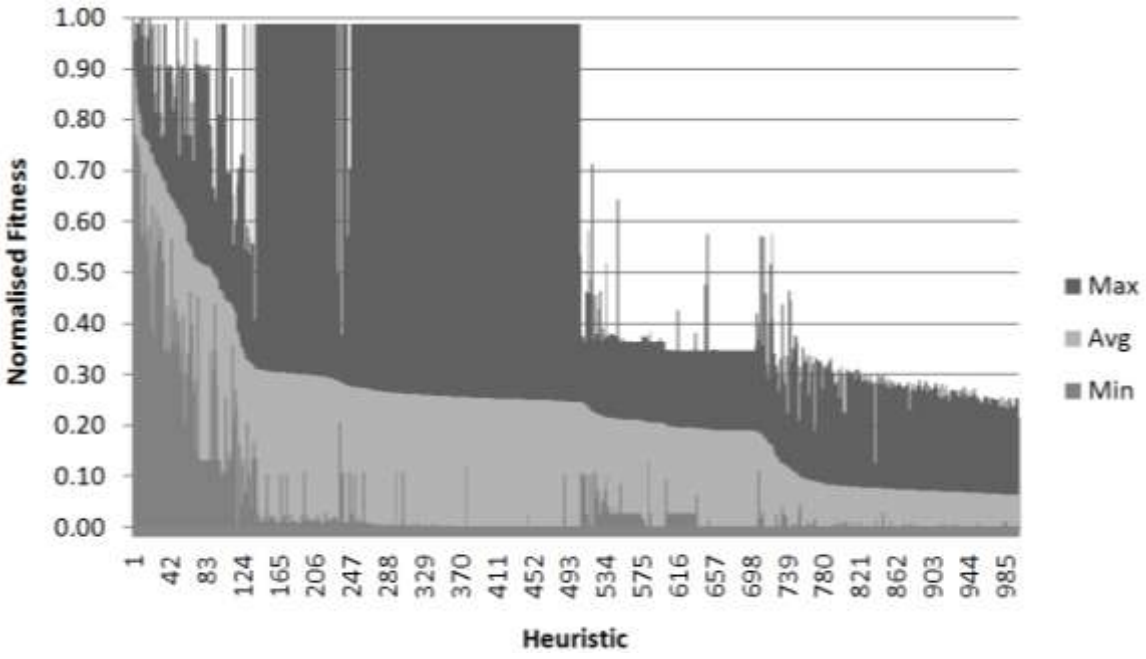


Figure 7.8. Top 1000 randomly created repair heuristics and their average performance on 5 problem instances. "Good" was chosen as rank 2, "ok" as rank 38 and "bad" as 104.

Figure 7.8 shows the average performance of the top 1000 heuristics (a collection of 8 randomly chosen low level heuristics combined with the greed look ahead search) on the five problems ranked by average fitness and shows some interesting features. The stepped increases may indicate features of the solution methods. A lot of the low level

heuristics are similar in nature, doing similar things but with slightly different parameters (for example move a task right by 1, 2, 5, 10 minutes). Perhaps to reach a certain plateau requires a certain type of low level heuristic in the combination, and the small changes in quality come from the variations in parameters. Surprisingly, most of the heuristics failed in at least one instance (note that 0 is slightly above the axis for clarity). This is indicated where Min is equal to 0. The block of heuristics where max is nearly 1.0 may indicate that one of the problems is easy to solve if you have a given LLH in the selection. However, as the average increases the Max decreases which may indicate that this heuristic is only good for that given instance and thus to increase average performance we have to sacrifice the LLH that is good for that problem.

The three chosen heuristics ("good", "ok", and "bad"), artificially created of different quality, will be enhanced using the Variable Fitness Function. The Variable Fitness Function will be used to guide the search each time it is used to repair an event with the aim of fixing the event with as little disruption and change to the schedule as possible. During the learning phase, Variable Fitness Functions will be trained on the five previously mentioned problems. The fitness of a Variable Fitness Function will be measured as the sum of the normalised sum of fitness of the five problem instances (hence it usually lies between 0 and 5, 5 being better).

As only very small search depths have been used to date (Colledge, 2009) and it is intuitive that the Variable Fitness Function would be more effective with more iterations to use, we will run tests with tree search depths of 10, 30 and 50 to see if there is an advantage in doing so. The results in Figure 7.9 show that increasing the search depth to 30 yielded significant increases and increasing it even more to 50 provided slight further increases. Increasing search depth increases the CPU time required and so 30 iterations was chosen as a good trade off between solution quality and CPU time

required. The “Good” and “Bad” heuristic provide some very interesting results. The results would indicate that using a Variable Fitness Function enhanced search with depth 10 would provide better results than a non Variable Fitness Function enhanced search with depth 30. In practice, this would mean better results at approximately 1/3 of the computational power. This is also the case with the ”Bad” heuristic which seems to have reached results as good as is possible with the set of low level heuristics as increasing the search depth makes no change. “OK” with a 10 Deep VFF seems to be the only case where the VFF could not enhance the quality to that of the no VFF 30 deep version, however at the same depth, VFF enhanced the performance of all the heuristics.

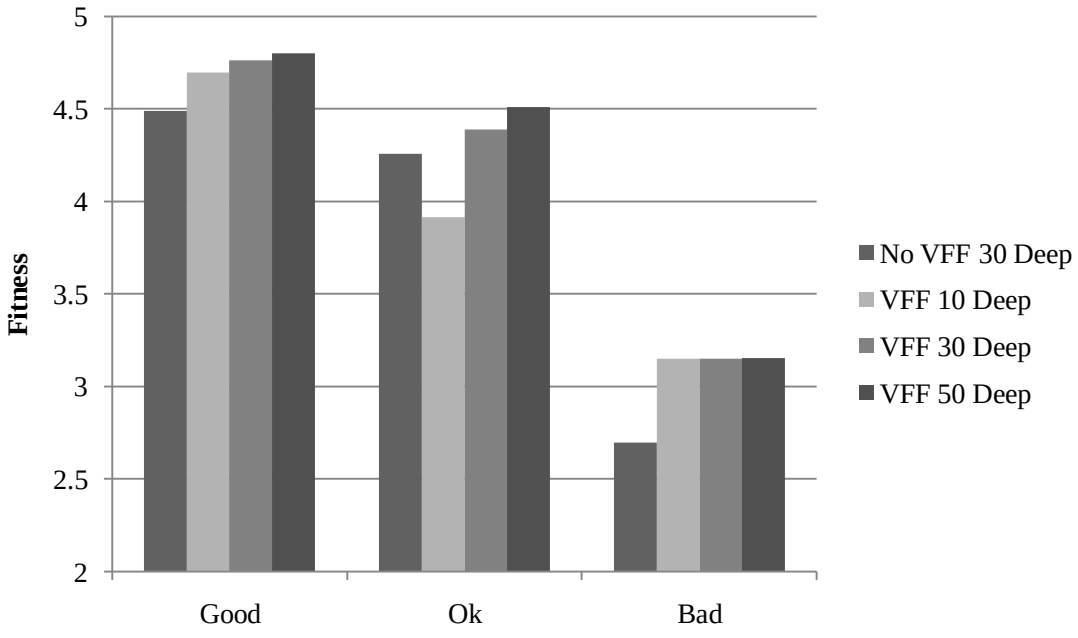


Figure 7.9. Performance of Variable Fitness Functions with different search depths compared to a fixed fitness function with a 30 deep search depth for a single run.

The Variable Fitness Function has only been used with linear functions to interpolate the weights between the weight sets. Here, we run tests with stepped VFFs and with linear interpolation. These results in figure 7.10 show an average of 10 runs (population size of 10, 50 generations each) with 90% confidence intervals and that there is no

statistically significant increase in performance with or without linear interpolation. This is probably due to the shortness of the Variable Fitness Functions and the complexity of the low level heuristics. Even so, they both show significant increases in solution quality to the non Variable Fitness Function enhanced methods.

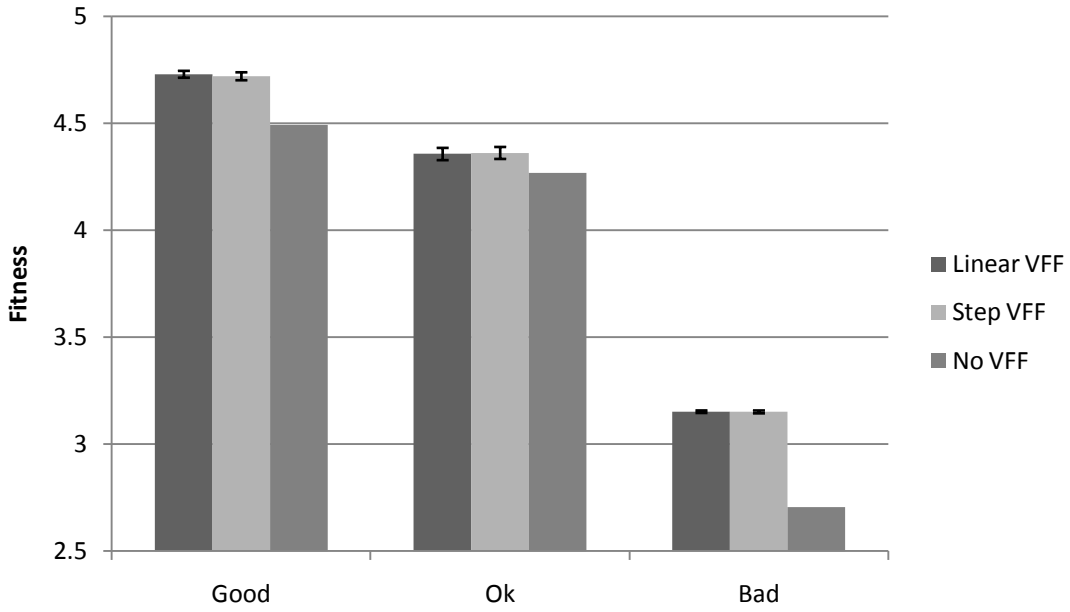


Figure 7.10. Stepped vs Linear Variable Fitness Functions. 90% error intervals are shown averaged over the 10 runs.

Generalisability, or the ability of the Variable Fitness Function to perform as well on unseen data, is a desirable factor. The evolved linear Variable Fitness Functions were used to test generalisability in three different ways: A) same schedules, unseen set of events; B) similar schedules, same set of events; and C) similar schedules, unseen set of events. The graph below shows the raw fitness values obtained from each of these experiments using the "Good" hyperheuristic. No normalisation is possible in this case without running thousands of heuristics on these new generalisation problems.

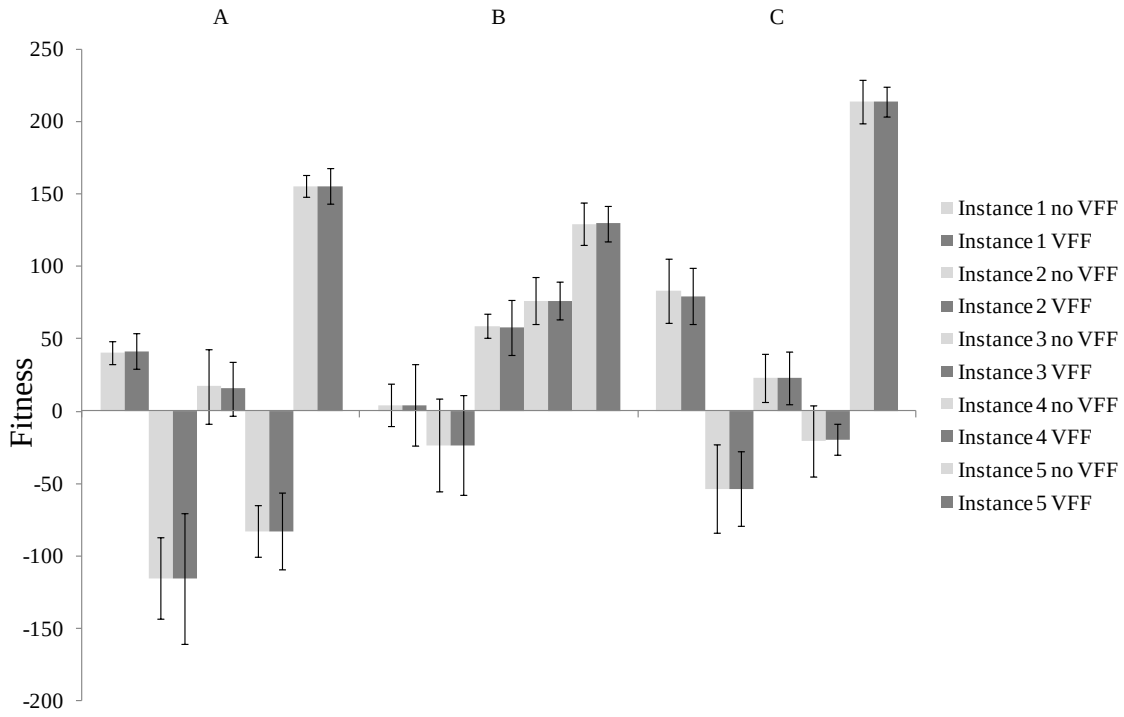


Figure 7.11. Using the evolved Variable Fitness Functions on unseen data.

Figure 7.11 shows only small (and not statistically significant) differences when using the best evolved Variable Fitness Function from each of the 10 runs on unseen data. This indicates that the Variable Fitness Functions evolved here would not make significant improvements to unseen data. The results for the "Ok" and "Bad" heuristic show similar results. The most likely cause of this is that too few instances were used in the evolution of the Variable Fitness Functions and so the Variable Fitness Functions are not general enough, finding ways to exploit the training data's features and not the problem itself, a problem known as "over-fitting".

The evolved Variable Fitness Functions look dissimilar however there are probably commonalties amongst them. Where problems have many objectives such as the ones evolved in this work, it is often hard to see trends between different evolved Variable Fitness Functions. We use scatter plot matrix (Becker and Cleveland, 1987) to try and visualise multiple Variable Fitness Functions. Figure 7.12 (top) shows the

scatter plot matrix which plots the final populations of the 10 runs of the "Good" heuristic. Each large box shows a Cartesian scatter plot of the weights assigned by VFF to two different objectives (the objective taken from the row and column of the matrix). Each point in the box represents a pair of objective weights found in one of the iterations of one of the Variable Fitness Functions plotted. Furthermore the colour of the point represents which iteration it was found in, dark at iteration 0, light at iteration 30, which helps to see time trends. Figure 7.11 (bottom) shows the same scatter plots, with our observed trends marked with arrows. This representation makes it easier to see trends as it lets us see a good deal of information all at once.

Here we see some definite trends in the data. If we look at the TT/TT, DST/DST and STA/STA plots, we can see that these objectives seem to change over the course of the iterations, TT starting strongly negative and ending weakly negative/positive, DST seems to be a lot noisier however a positive to negative shift can be seen. STA is noisy like DST however it starts strongly negative and becomes positive. In SP/SC two trends can be seen: a big sweeping shift and smaller, seemingly opposite shift. This is probably because multiple strategies produce good results, so multiple different trends can be seen. Several other big sweeping shifts can be seen and are highlighted.

This information can be used to better understand how the Variable Fitness Functions are improving the heuristic and can be better explained to non experts who can be a bit wary when told the precise numbers as they might find it difficult to understand why or how this should work. This information could also be used to build new types of heuristics which mimic the VFF's behaviour. This would provide users with a new interesting way to explore heuristics and aid in the design process of new heuristics. It may also give new insight into the problem itself.

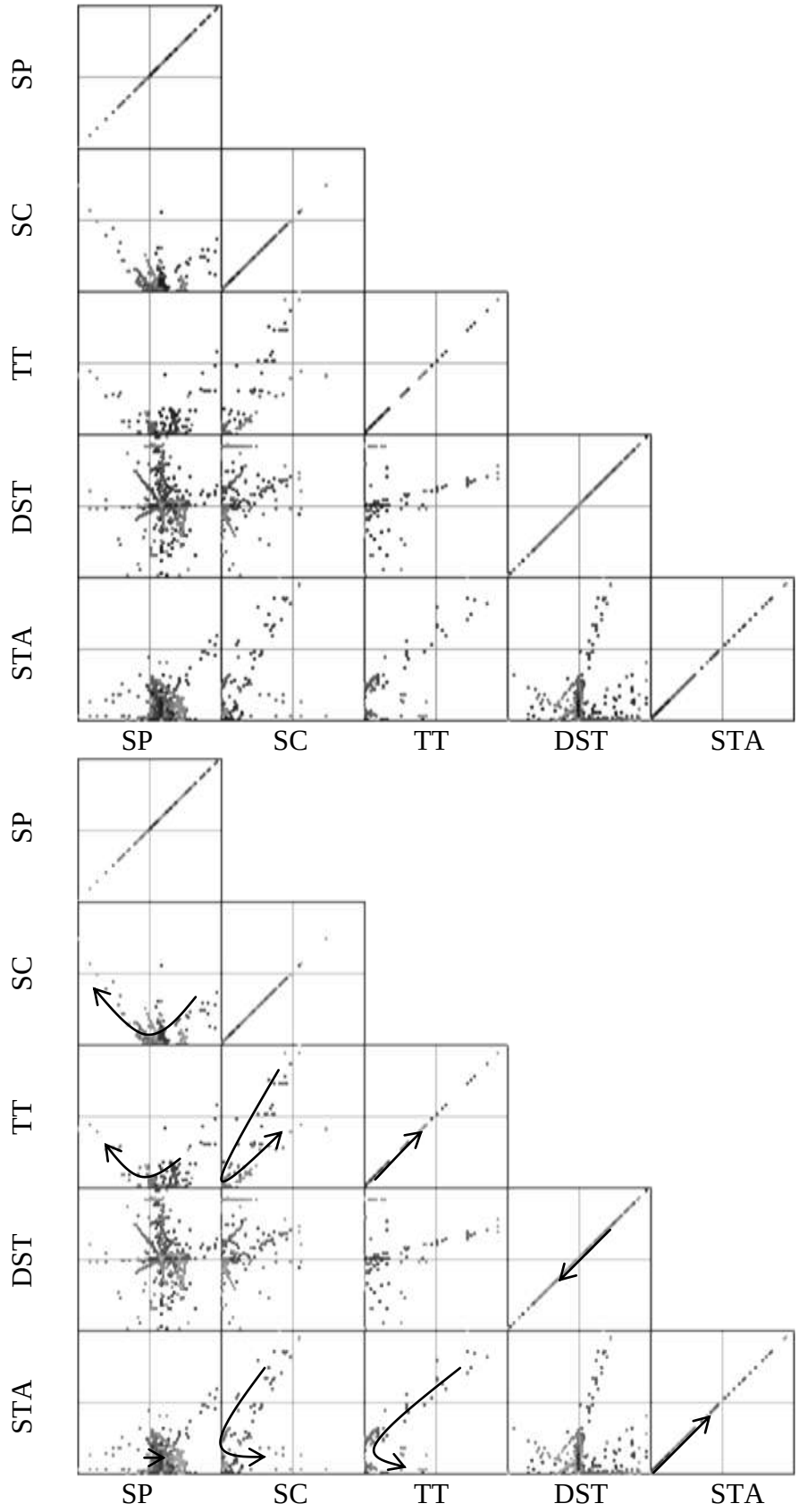


Figure 7.12. Scatter plot matrices for the evolved Variable Fitness Functions with identified trends highlighted below.

7.2.1. Summary

Large amounts of experimentation with the Variable Fitness Function applied to a hyperheuristic for the dynamic workforce repair problem have been undertaken. These experiments have shown several things

- Using a Variable Fitness Function in certain situations can produce results of higher quality with less iterations of search. In practice this means that potentially better results can be obtained in less CPU time.
- In this solution method, there is little to choose between linearly interpolated and stepped Variable Fitness Functions.
- The Variable Fitness Functions evolved were no better than the non Variable Fitness Function enhanced versions when used on unseen data. The most likely cause of this is the noisy problem. The Variable Fitness Function used in Section 6.3 and 7.1 to enhance the building of schedules similar to the ones in this section found five problem instances were enough to evolve generalisable Variable Fitness Functions, however more may be required in this noisy nature of the dynamic environment.

Furthermore, scatter plot matrices were used to analyse the evolved Variable Fitness Functions. This proved to be an easy way to make this very complex data easily readable to a non expert and could potentially be used in heuristic design.

7.3. Summary and Generalisation Ability of Variable Fitness Functions

A large amount of experimentation with the Variable Fitness Function has been applied in this thesis with varying results. Table 7.5 summarises key elements we have

identified which may determine whether or not evolved Variable Fitness Functions will be generalisable.

Table 7.5. Indicators which may lead to evolved Variable Fitness Functions being generalisable.

[1] Scheduling see Section 6.2 and 7.1
[2] Virus board game see Section 6.2
[3] TSP see Section 6.1
[4] Schedule Repair see Section 7.2

Indicator	Generalisable	Not Generalisable
Iterations	Very Large [1]	Very Few [4]
	Medium [2]	Few [3]
Complexity of Local Moves	Low [1,2]	High [4]
		Low [3]
Correlation of Objectives	High [1,2]	None [3]
		Medium [4]
Number of Objectives	Medium (6) [1]	Very Low (2) [3]
	High (18) [2]	Medium (4) [4]
Noise in Problem	Low [1]	Low [3]
	Medium [2]	High [4]
Heterogeneity of Training Data	High [1,2]	Medium [4]
		Very Low [3]
Similarity of Test Data to Training Data	High [1,2]	Very Low [3]
		Medium [4]

Iterations: It would appear that larger iteration counts are required. In the very generalisable cases (where evolved Variable Fitness Functions worked will on unseen data), static scheduling and virus, 10,000 and ~60 iterations were used respectively, compared to the 30 used in repair. Interestingly, the results seen from the different depths of repair show that increasing the search depth further is unlikely to increase performance. It is also infeasible to increase the search depth to the order of these in the static scheduling as too much CPU time would be required. The exception here is the TSP application, however the aim of that investigation was not to produce generalisable results (and the reason for no generalisability is clearly explained in "Similarity of Training Data" and "Correlation of Objectives").

Complexity of Local Moves: Both virus and static scheduling used very simple moves compared to the complex low level heuristics used in the repair. This gives the Variable Fitness Function more control over what is happening as it can more directly affect the results. The hyperheuristic in the repair raises the Variable Fitness Function's abstraction level from the problem. Intuitively, the closer the Variable Fitness Function can operate on the problem the better the results.

Correlation of Objectives: Correlation of objectives seems to be slightly important. In the scheduling problems, the objectives had indirect effects on the other objectives (an increase in travel time increased travel cost, decrease in travel time would mean more tasks could be scheduled increasing scheduled priority etc.) In virus, a lot of the board evaluation functions were similar and complementary and of course had objectives from the view of both players made the objectives have a direct effect on other objectives. In the TSP there was no correlation between the objectives, within the problem instances, nor between different problem instances. This would make it hard for the Variable Fitness Function to a) manipulate the solution and b) learn information that is common to different instances.

Number of Objectives: The data seems to indicate that many objectives are better than none. In fact identifying extra objectives unused in the global fitness function is advantageous as seen in section 7.1. Adding more objective measures, no matter trivial, is a good idea because there is the possibility that the VFF will learn to use one of these to good effect.

Noise in Problem: Low noise is best. Noisy environments have been observed to affect GAs negatively in literature and would appear to be supported by this work. Rattray and Shapiro (Rattray and Shapiro, 1997) state that it is possible to overcome the effects of noise by using a larger population, although this is impractical for our computationally intensive studies.

Similarity of Training Data and *Similarity of Test Data to Training Data*: There seems to be a direct correlation here as the similarity ranges from none in the TSP, to some in repair, to high in static scheduling and virus, the generalisability increases from none to high. This is expected as training on more similar problems is likely to lead to generalisable Variable Fitness Functions.

Chapter 8

Conclusions, Observations and Future Work

This chapter concludes the thesis summarising the work done, reflecting on the undertaking of the work and identifies interesting areas of work arising from the thoughts presented in this thesis.

8.1. Conclusions

In this thesis very large amounts of CPU time have been used to empirically test new heuristic methods on various problems. The methods have been analysed in detail in order to ascertain how they work and give a greater understanding of the methodology.

In Chapter 4, local search heuristics were developed and tested in order to create a hyperheuristic framework for a detailed model to capture important features of Trimble MRM's dynamic workforce scheduling problem. Initially we showed that the solutions generated by a simple genetic algorithm could be enhanced using exact search methods. We then introduced a method for splitting the dynamic workforce scheduling problem up using the smaller problem of scheduling a single task optimally. Hundreds of low level heuristics were created to solve these smaller parts of the problem using exact methods. Variable Neighbourhood Search and hyperheuristics were used to control the order in which these smaller problems were solved on realistic problem instances our industrial sponsor identified and produced schedules vastly superior to the genetic algorithm showing that exact/heuristics hybrids are an effective optimisation tool in this case. Furthermore the results of greedy hyperheuristic could be analysed to see which methods to use at what point in the scheduling process to create a solution faster. In business terms this equates to more cost effective solutions, more quickly. Analysis of the low level heuristics showed that hyperheuristic used many of the low level heuristic and there was no single "silver bullet". It did however reveal that many were used in less than 1% of the search indicating that learning which ones to avoid could potentially lead to less CPU time being used. The hyperheuristics produced fitter results than the variable neighbourhood search, however used a much larger CPU time.

This work was published in:

S. M. Remde, P. I. Cowling, K. P. Dahal, N. J. Colledge, "**Exact/Heuristic Hybrids Using rVNS and Hyperheuristics for Workforce Scheduling**" in Proceedings of EvoCOP 2007, Springer LNCS 4446, 2007, pp. 188-197. (One of 3 papers nominated for the best in conference nomination out of 81)

We introduced learning in the form of a Binary Exponential Back Off based Tabu Hyperheuristic in Section 4.4. The Tabu based algorithm was used to learn which low level heuristics were performing badly and not use them. The time between trying repeatedly poor performing low level heuristics was increased exponentially minimising the time wasted on the low level heuristics that never perform well or only perform well at the end. The heuristic was shown to be better than fixed and random Tabu tenures and managed to produce results within 99% of the best, very CPU intensive, hyperheuristic in one third of the CPU time required.

This work was published in:

S. M. Remde, P. I. Cowling, K. P. Dahal, N. J. Colledge. "**Binary Exponential Back Off for Tabu Tenure in Hyperheuristics**" in Proceedings of EvoCOP 2009, Springer LNCS 5482, 2009.

In Chapter 5 we introduce the Variable Fitness Function framework and a method of evolution, which we used in chapter 6 and 7 with various local search and metaheuristics respectively. The Variable Fitness Function provides and intelligent way to control the direction of any local search-based heuristic in order to yield better results. Extra CPU time is needed to evolve these "search directions" and if available

this methodology provides a very simple yet powerful way of enhancing a search heuristic without the need to modify the underlying search heuristic. This is useful when the heuristic is bespoke and crafting more intelligent search heuristics require a large amount of work or expert knowledge. Usually the cost of such is greater than the cost of extra CPU cycles used to evolve a Variable Fitness Function which in most of the cases was reusable. If the CPU time used to evolve a Variable Fitness Function can be reused because the Variable Fitness Function is exploiting the problem and not the problem instance, then the cost of evolving the Variable Fitness Function can be amortized by multiple uses.

In Chapter 6.1 the Variable Fitness Function was used with a multiobjective TSP. This work showed how it is easy to create an artificially multiobjective problem out of a single objective problem for use with the Variable Fitness Function. In this section the Variable Fitness Function enhanced methods were superior to the original heuristics. The search was visualised and indicators of the Variable Fitness Function moving the search to different areas of the search space and out of local optima were seen. This section was published in

This work was published in:

S. M. Remde, P. I. Cowling, K. P. Dahal, N. J. Colledge, "**Evolution of Fitness Function to Improve Heuristic Performance**" in proceedings of Learning and Intelligent Optimization (LION) II, Springer LNCS 5313, 2008, pp 206-219.

In Chapter 6.2 the Variable Fitness Function was used to create players for a tactically rich board game. It was hypothesised that tactics at the beginning, middle and end of the game would differ and the Variable Fitness Function provided a great way of

capturing this. Indeed, Variable Fitness Functions evolved for this game showed such behaviour and the evolution in this case was an adaptive one, evaluating the population against two hand crafted players and the three best evolved players. This lead to some interesting plots of fitness over the generations which provided strong evidence that the players were getting better.

In Chapter 6.3 the Variable Fitness Function was used with a constructive scheduling process. "The Bank" was developed to try and anticipate bottlenecks when using constructive scheduling. The Bank keeps track of the supply and demand of each type of resource to try and identify potential bottlenecks during the scheduling process. It can be used to compare to potential moves to see which one results in the most favourable situation. The Variable Fitness Function here was used to control not only the weights of the fitness function, but also the a variable related to the bank. This shows that the Variable Fitness Function could be used in other situations where parameters need to change over time and are not directly considered in a fitness function. This has been submitted to European Journal of Heuristics.

In Chapter 7.1 the Variable Fitness Function was used on a Variable Neighbourhood Search for the scheduling problem. One such evolved Variable Fitness Function was shown to work similarly to a right-left shift. We showed evidence that the Variable Fitness Function was able to create this well-known schedule improvement which shows the potential to make new heuristics for problems.

This work was published in:

K. P. Dahal, S. M. Remde, P. I. Cowling, N. J. Colledge, "**Improving Metaheuristic Performance by Evolving a Variable Fitness Function**" in Proceedings of EvoCOP 2008, Springer LNCS 4972, 2008, pp. 170-181

In chapter 7.2 multiple evolved Variable Fitness Functions for the dynamic side of the scheduling problem were visualised simultaneously using scatter-plot matrices. This helped the user to see trends between multiple Variable Fitness Functions that would have otherwise been hard to identify. These higher level trends give a greater understanding of how the Variable Fitness Functions are working. The Variable Fitness Functions evolved for the dynamic problem were unable to generalise across unseen instances however they were able to improve the performance on instances they were evolved on.

The ability of the evolved Variable Fitness Functions to perform well on unseen data was always tested and Table 7.5 gives a summary of the information obtained in this thesis, however, this needs to be investigated more extensively.

8.2. Observations

I see general approaches to solving problems as a research direction of continuing importance. When a new problem needs to be solved, general methods can be quickly adapted to solve it to a high quality whereas tailored methods may be much harder to apply. With generality comes a loss of quality of course, and that is why I see methods such as the ones described in this thesis as an enabler to understand problems better. The Variable Fitness Function discovered the right-left heuristic in days, whereas it was published by experts in 2003 – years after the RCPSP was formulated. Admittedly this pattern was identified as a right-left shift – if we had no idea what a right-left shift was it might not have been so straight forward. There could be many more efficient heuristics identified in the Variable Fitness Functions we have evolved in

this thesis that do not have a name – yet. Potentially this approach could be used with existing and new problems to identify new and interesting areas to look at.

Working with an industrial partner has really aided the work and added credibility to the results. Trimble MRM Ltd. has had great input into the project and the discussions with them have always led to interesting ideas. Furthermore they helped us identify a real world problem and develop a model that captures the problems faced by many different types of their customers.

Working with a real world problem has provided us with a rich multiobjective problem landscape to test complex heuristic methods. The disadvantage of this is the CPU time required to solve the problems. Sometimes this became an issue when debugging problems. For testing purposes we found it useful to use the TSP which can be solved much quicker.

Due to the complex problem and the many runs we did to average results, a very large amount of CPU time was needed (over 3.25 CPU years of experiments actually made it into the thesis). The Genetic Algorithms were parallelised using the method in (Colledge, 2009) and the other experiments were parallelised at the experiment level on up to 100 machines. We used University computer labs over nights and weekends which meant that machines could be turned off (by other students) and so the software had to be resilient to failure.

8.3. Future Work

This section identifies some areas of interest for future work arising from this thesis.

8.3.1. Exact/Heuristic Hybrids

In Chapter 4 Exact/Heuristic hybrid methods were used on the workforce scheduling problem. It would be interesting to see if the techniques used in this chapter to make lots of low level heuristics could be applied to other problems, and if the used of the rVNS and hyperheuristics would produce as significant results. The analysis of the results should also be extended in 3 ways:

1) Each low level heuristic should be used in isolation to show that the result truly is a combination of multiple heuristics and not a single one.

2) The pattern of low level heuristics used by the greedy search could be applied to other problems to see if it provides good results on unseen data. If so, the greedy hyperheuristic could be used to learn a pattern of low level heuristics offline and then that pattern used (many times quicker) online. This is similar to the HyperGA approach however patterns would be found by trial, not by evolution.

3) Analysis was done to see at which part of the search low level heuristics were used. This could be taken further to build rVNS neighbourhoods based on the analysis of these results. This rVNS would be a lot faster and could potentially provide nearly as good results as the greedy hyperheuristic with the speed of the rVNS.

8.3.2. Binary Exponential Back Off

The Binary Exponential Back Off algorithm used in chapter 4 to control the Tabu tenures of individual low level heuristics looks very promising. The idea of parameterless hyperheuristic that “just works” is interesting, and this certainly takes a step in that direction. BEBO Tabu search introduced a new parameter however this can can be used to determine the trade-off between CPU time and solution quality.

Further work should be done experimentally testing the BEBO heuristic with different problems and parameters to see that the parameters affect the trade-off in other problems also. It is very likely that a good estimation of solution quality and CPU time required can be calculated as a function of the proportion of low level heuristic deemed bad. If this is the case it could provide an invaluable tool for testing what-if scenarios, where the user may want quick responses in the short run while trying scenarios, but a better quality solution once the ideal scenario has been found.

BEBO uses an existing mechanism to exponentially avoid trying bad moves. Similar statistical approaches which try to focus on good areas exist such as UCB (Kocsis and Szepesvari, 2006). It would be interesting to see if these ideas could be used to control search.

8.3.3. Variable Fitness Function

Numerous areas of research arise from the Variable Fitness Function.

1. More extensive work needs to be carried out to try and identify factors which make an evolved Variable Fitness Function work across multiple problem instances. This means investigating different problems with different characteristics, the number and diversity of the problems instances used to evolve the Variable Fitness Functions, and the similarities and differences between the ones where generalisation works and does not.
2. An investigation of why the evolution process is so insensitive to the evolutionary parameters should be carried out. This has been observed many times in the thesis and it is likely to be because of the higher level nature of the evolution, however more evidence of this would be nice and further investigation and evidence is needed.

3. Variable Fitness Functions that vary over scalars other than time or iterations is also an interesting idea. Variable Fitness Functions could be made to vary depending on a characteristic of the problem (such as the availability of resources, the time of day, the month of the year, etc) in order to learn from and exploit features of these characteristics. Multi-dimensional Variable Fitness Function could mean the weighted sum fitness function could vary according to two or more measures as this would give a greater depth to the characteristics of a problem the Variable Fitness Functions could learn.
4. So far we have only used genetic algorithms to optimise Variable Fitness Functions. These have the advantage of being tried and tested and it has worked out well as it seems very insensitive to parameters. Other way of creating Variable Fitness Functions should be investigated. Combining this with point 3 could also provide interesting results. For example, say we used schedule workload (some measure of how under/over subscribed the schedule is) as the dimension and then asked an expert to define weights for different scenarios. The Variable Fitness Function could be used to fill in the gaps. This could be combined with Valuated State Space ideas to add an easier interface for the expert to define these weights.
5. So far piecewise linear and stepping functions have been used. It would be interesting to try different functions like Bezier curves or sine functions to connect the discontinuities. The work so far suggests different types of functions do not have an effect, but this was on very short searches and it is more than likely to have a larger effect on longer searches.

Bibliography

Aarts, E. and Korst, J. (1989) *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Computing*, Chichester: Whiley.

Aarts, E. and Lenstra, J. (2003) *Local Search in Combinatorial Optimization*, John Wiley & Sons.

Abdullah, S., Ahmadi, S., Burke, E., Dror, M. and McCollum, B. (2007) 'A Tabu based large neighbourhood search methodology for the capacitated examination timetabling problem', *Journal of Operational Research Society*, vol. 58, pp. 1494 - 1502.

Adams, J., Balas, E. and Zawack, D. (1988) 'The Shifting Bottleneck Procedure for Job Shop Scheduling', *Management Science* , vol. 34, no. 3, pp. 391-401.

Albiach, J., Sanchis, J.M. and Soler, D. (n.d) '(2008) An asymmetric TSP with time windows and with time-dependent travel times and costs: An exact solution through a graph transformation', *Eur. J. Op. Res.*, vol. 189, no. 3, pp. 789-802.

Applegate, D., Bixby, R., Chvatal, V. and Cook, W. *Concorde TSP Solver*, [Online], Available: http://www.tsp.gatech.edu/concorde.html [12 Jun 2007].

Ayob, M. and Kendall, G. (2003) 'A Monte Carlo Hyper-Hueristic To Optimise Component Placement Sequencing For Multi Head Placement Machine', Proceedings of the International Conference on Intelligent Technologies, pp. 132-141.

Bai, R. and Kendall, G. (2003) 'An Investigation of Automated Planograms Using a Simulated Annealing Based Hyperheuristics', Meta-heuristics: Progress as Real Problem Solvers, Selected Papers, pp. 87-108.

Becker, O. (1967) 'Das HelmstaK dtersche Reihenfolgeoproblem die Effzienz verschiedener NaKherungsverfahren', Computer uses in the social sciences. Berichteiner Working Conference, Wien.

Becker, R. and Cleveland, W. (1987) 'Brushing Scatterplots', *Technometrics*, vol. 29, no. 2, pp. 127-142.

Boctor, F.F. (1996) 'Resource-constrained project scheduling by Simulated Annealing', *International Journal of Production Research*, vol. 34, no. 11, pp. 2335-2351.

Bouleimen, K. and Lecocq, H. (2003) 'A new efficient simulated annealing algorithm for the resource-constrained project scheduling and it's multiple mode version.', *European Journal of Operational Research*, vol. 149, no. 2, pp. 268-281.

Bremermann, H.J. (1958) 'The evolution of Intelligence. The Nervous System as a Model of it's environment.', *Technical Report No. 1, Dept. Of Mathematics, Univ. of Seattle, Washington*, vol. 477, no. 17.

Brucker, P., Drexl, A., Mohring, R., Neumann, K. and Pesch, E. (1999) 'Resource-constrained Project Scheduling: Notation, classification, models and methods', *European Journal of Operational Research*, vol. 112, pp. 3-41.

Burke, E. (2003) 'Hyper-Heuristics: An Emerging Direction in Modern Search Technology', in *Handbook of Metaheuristics*, Springer.

Burke, E., Cowling, P. and Keuthen, R. (2001) 'Effective Local And Guided Variable Neighbourhood Search Methods for the Asymetric Travelling Salesman Problem', Proceedings of Applications Of Evolutionary Computing, pp. 203-212.

Burke, E., De Causmaecker, P. and Berghe, G. (1999) 'A hybrid tabu search algorithm for the nurse rostering problem" Simulated Evolution And Learning', *Lecture Notes In Artificial Intelligence* , vol. 1585, pp. 187-194.

Burke, E.K., Kendall, G., Misir, M. and Özcan., E. (2008) 'A study of simulated annealing hyperheuristics. ', Proceedings of the 7th International Conference on the Practice and Theory of Automated Timetabling, pp. 1-4.

Burke, E., Kendall, G. and Soubeiga, E. (2003) 'A Tabu-Search Hyperheuristic for Timetabling and Rostering', *Journal of Heuristics*, vol. 9, December, pp. 451-470.

Burke, E. and Soubeiga, E. (2003) 'Scheduling nurses using a tabu-search hyper heuristic', Proceedings of the 1st Multi-disciplinary International Scheduling conference: Theory and Applications, pp. 197-218.

Carter, M. and Laporte, G. (1996) 'Recent developments in practical examination timetabling', *Practice and theory of automated timetabling*, pp. 3-21.

Chakhlevitch, K. and Cowling, P. (2005) 'Choosing the Fittest Subset of Low Level Heuristics in a Hyperheuristic Framework.', Evolutionary Computation in Combinatorial Optimization, Lecture Notes in Comp. Science 3448, pp. 23-33.

Chakhlevitch, K. and Cowling, P. (2008) 'Hyperheuristics: Recent Developments.', in Cotta, C., Sevaux, M. and Sorensen, K. *Adaptive and Multilevel Metaheuristics*, Studies in Computational Intelligence, vol 136, Springer.

Chanas, S. and Kobylanski, P. (1996) 'A new heuristic algorithm solving the linear ordering problem.', *Computational Optimization and Applications*, vol. 6, no. 2, pp. 191-205.

Chu, P. and Beasley, J. (1998) 'A Genetic Algorithm for the Multidimensional Knapsack Problem', *Journal of Heuristics*, vol. 4, no. 1, pp. 63-86.

Colledge, N. (2009) *Thesis: Evolutionary approaches to dynamic mobile workforce scheduling*, Bradford: University of Bradford.

Corne, D. and Ross, P. (1995) 'Peckish Initialisation Strategies for Evolutionary Timetabling', Proceedings of PATAT, pp. 227-240.

Cowling, P. (2005) 'Board Evaluation For The Virus Game', IEEE Symposium on Computational Intelligence in Games, pp. 59-65.

Cowling, P. and Chakhlevitch, K. (2003) 'Hyperheuristic for managing a large collection of low level heuristics to schedule personnel', Proc. of the 2003 IEEE Congress on Evolutionary Computation, IEE Press, pp. 1214-1221.

Cowling, P., Colledge, N., Dahal, K. and Remde, S. (2006) 'The Trade Off between Diversity and Quality for Multi-objective Workforce Scheduling', Evolutionary Computation in Combinatorial Optimization, pp. 13-24.

Cowling, P., Colledge, N., Dahal, K. and Remde, S. (2006) 'The Trade-Off between diversity and quality for multiobjective scheduling', EvoCOP 2006, Springer LNCS 3906, pp. 13-24.

Cowling, P., Fennell, R., Hogg, R., King, G., Rhodes, P. and Sephton, N. (2004) 'Using Bugs And Virus to Teach Artificial Intelligence', In the proceedings of 5th Game-on International Conference on Computer Games: Artificial Intelligence, Design and Education, pp. 360-364.

Cowling, P. and Johansson, M. (2002) 'Using real time information for effective dynamic scheduling', *European Journal of Operational Research*, vol. 139, no. 2, pp. 230-244.

Cowling, P., Kendall, G. and Han, L. (2002) 'An Investigation of a Hyper-heuristic Genetic Algorithm Applied to a Trainer Scheduling Problem', Proceedings of Congress on Evolutionary Computation (CEC2002), Hawaii, pp. 1185-1190.

Cowling, P., Kendall, G. and Soubeiga, E. (2001) 'Hyperheuristic Approach to Scheduling a Sales Summit.', Selected papers from the 3rd International Conference on the Practice and Theory of Automated Timetabling (PATAT 2001), pp. 12-19.

Cowling, P., Kendall, G. and Soubeiga, E. (2002) 'Hyperheuristics: A Tool for Rapid Prototyping in Scheduling and Optimisation', Applications of Evolutionary Computing: Proceedings of Evo Workshops, Lecture Notes in Comp. Sci. 2279, pp. 1-10.

Cowling, P.I. and Keuthen, R. (2005) 'Embedded local search approaches for routing optimization.', *Computers & OR*, vol. 32, pp. 465-490.

Cowling, P., Naveed, M. and Hossain, M. (2006) 'A Coevolutionary Model for the Virus Game', IEEE Symposium on Computational Intelligence and Games, pp. 45 - 51.

Cowling, P., Remde, S., Dahal, K. and Colledge, N. (Submitted) 'Evolution of Fitness Functions to Improve Optimisation', *Journal of Heuristics*.

Croes, G.A. (1958) 'A Method for Solving Traveling Salesman Problems', *Operations Research*, vol. 6 , pp. 791-812.

Dahal, K., Remde, S., Cowling, P. and Colledge, N. (2008) 'Improving Metaheuristic Performance by Evolving Variable Fitness Functions', Evolutionary Computation in Combinatorial Optimization Proceedings, pp. 70-181.

Dahal, K., Tan, K. and Cowling, P. (2007) *Evolutionary scheduling*, Springer SCI.

Deb, K. (2004) *Multi-Objective Optimisation using Evolutionary Algorithms*, Wiley.

Debels, D., Reyck, B., Leus, R. and Vanhoucke, M. (2004) 'A Hybrid Scatter Search/electromagnetism Meta-Heuristic for Project Scheduling.', Working Papers of Faculty of Economics and Business Administration, Ghent University, Belgium, p. 04/237.

Debels, D., Reyck, B.D., Leus, R. and Vanhoucke, M. (2006) 'A hybrid scatter search/electromagnetism meta-heuristic for project scheduling', *European Journal of Operational Research*, vol. 169, no. 2, pp. 638-653.

Dell'Amico, M. and Trubian, M. (1993) 'Applying tabu search to the job-shop scheduling problem', *Annals of Operational Research*, vol. 41, no. 3, pp. 231-252.

Denzinger, J., Fuchs, M. and Fuchs, M. (1997) 'High Performance ATP Systems by Combining Several AI Methods', Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI '97), pp. 102-107.

Derigs, U., Kabath, M. and Zils., M. (1999) 'Adaptive genetic algorithms: A methodology for dynamic autoconfiguration of genetic search algorithms', in Voss, S., Martello, S., Osman, I. and Roucairol, C. *Meta-heuristics: Advances*

and trends inlocal search paradigms for optimization, Kluwer Academic Publishers.

Dorndorf, U. and Pesch, E. (1995) 'Evolution Based Learning in a Job Shop Scheduling Environment', *Computers & Operations Research*, vol. 22, pp. 25-40.

Drexl, A. and Gruenewald, J. (1993) 'Nonpremptive multi-mode Resource Constrained Project Scheduling', *IIE Transactions*, vol. 25, pp. 74-81.

eggermont, x. (1999) 'Adapting the fitness function in GP for data mining', Proceedings of Second European Workshop on Genetic Programming.

Eggermont, J., Eiben, A. and van Hemert, J. (1999) 'Adapting the Fitness Function in GP for Data Mining', *Genetic Programming*, pp. 195-204.

Eggermont, J. and van Hemert, J. (2001) 'Adaptive Genetic Programming Applied to New and Existing Simple Regression Problems', *Genetic Programming*, pp. 23-35.

Eggermont, J. and van Hemert, J. (n.d) 'Stepwise Adaptation of Weights for Symbolic Regression with Genetic Programming', *Proceedings of the Twelfth Belgium/Netherlands Conference on Artificial Intelligence (BNAIC'00)*, pp. 259-266.

Eiben, A. and van Hemert, J. (2001) 'SAW-ing EAs: adapting the fitness function for solving constrained problems', *New ideas in optimization*, pp. 389-402.

Eiben, A.E., van Hemert, J.I., Marchiori, E. and Steenbeek, A.G. (1998) 'Solving binary constraint satisfaction problems using evolutionary algorithms with an adaptive fitness function', Proceedings of the 5th Conference on Parallel Problem Solving from Nature, pp. 196-205.

Ernst, A.T., Jiang, H., Krishnamoorthy, M. and D., S. (2004) 'Staff scheduling and rostering: A review of applications, methods and models', *Eur. J. Op. Res*, vol. 153, p. 3–27.

Falkenauer, E. (1996) 'A Hybrid Grouping Genetic Algorithm for Bin Packing', *Journal of Hueristics*, vol. 2, no. 1, pp. 5-30.

Fang, H., Ross, P. and Corne, D. (1994) 'A Promising Hybrid Ga/Heuristic Approach for Open-Shop Scheduling Problems', 11th European Conference on Artificial Intelligence, pp. 590-594.

Fleszar, K. and Hindi, K. (2004) 'Solving the resource-constrained project problem by a variable neighbourhood scheduling search', *European Journal of Operational Research*, vol. 155, no. 2, Jun, pp. 402-413.

Fox, B. (1996) 'An Algorithm for schedule improvement by schedule shifting'.

Fraser, A. (1957) 'Simulation of genetic systems by automatic digital computers - Effects of linkage on rates under selection.', *Australian Journal of Biological Science*, vol. 10, pp. 492-499.

Garcia, C., Perez-Brito, D., Campos, V. and Marti, R. (2006) 'Variable neighborhood search for the linear ordering problem', *Comp. & Oper. Research*, vol. 33 , no. 12, Dec, pp. 3549-3565.

Garey, M.R. and Johnson, D.S. (1979) *Computers and intractability: A Guide to NP-Completeness*, Freeman.

Glover, F. (1986) 'Future paths for integer programming and links to artificial intelligence', *Computers & Operations Research*, vol. 13, no. 5, pp. 533-549.

Glover, F. (1998) 'A Template for Scatter Search and Path Relinking', in *Artificial Evolution*, LNCS 1363, Springer.

Glover, F. and Taillard, E. (1993) 'A user's guide to tabu search', *Annals of Operations Research*, vol. 4, no. 1, pp. 1-28.

Glover, F., Taillard, E. and De Werra, D. (1993) 'A User's Guide to Tabu Search', *Annals of Operational Research*, vol. 41, no. 1, pp. 3-28.

Han, L. and Kendall, G. (2003) 'An investigation of a tabu assisted hyper-heuristic genetic algorithm', The 2003 Congress on Evolutionary Computation, pp. 2230-2237.

Han, L. and Kendall, G. (2003) 'Guided Operators for a Hyper-Heuristic Genetic Algorithm', Australian Conference on Artificial Intelligence, pp. 807-820.

Han, L., Kendall, G. and Cowling, P. (2002) 'An Adaptive Length Chromosome Hyper-Heuristic Genetic Algorithm for a Trainer Scheduling Problem', Proceedings of the 4th Asia-Pacific Conference on Simulated Evolution And Learning (SEAL'02), pp. 267-271.

Hansen, P. (1986) 'The steepest ascent/mildest decent heuristic for combinatorial programming', Congress on Numerical Methods in Combinatorial Optimistation, Capri, Italy.

Hansen, P. and Mladenovic, N. (2001) 'Variable neighborhood search: Principles and applications', *European Journal of Operational Research*, vol. 130 , no. 3, May , pp. 449-467.

Hansen, P., Mladenović, N. and Moreno Pérez, J.A. (2008) 'Variable neighbourhood search: methods and applications', *4OR: A Quarterly Journal of Operations Research*, vol. 6, no. 4, December, pp. 319-360.

Hartmann, S. (1997) 'A Competitive Genetic Algorithm for RCPSP', *Naval Research Logistics*, vol. 45, pp. 733-750.

Hartmann, S. (1999) *Project Scheduling under Limited Resources: Model, methods and applications*, New York: Springer-Verlag.

Hartmann, S. (2002) 'A Self-Adapting Genetic Algorithm for Project Scheduling under Resource Constraints', *Naval Research Logistics*, vol. 49, no. 5, pp. 433 - 448.

Hartmann, S. and Kolisch, R. (2000) 'Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem', *European Journal of Operational Research*, vol. 127, no. 2, pp. 394-407.

Harwig, J., Barnes, J. and Moore, J. (n.d) 'An Adaptive Tabu Search Approach for 2-Dimensional Orthogonal Packing Problems', *Military Operations Research*, p. 2006.

Hingston, P. and Masek, M. (2007) 'Experiments with Monte Carlo Othello', *IEEE Congress on Evolutionary Computation*, pp. 4059-4064.

Holland, J.H. (1975) *Adaptation in Natural and Artificial Systems*, Michigan: University of Michigan Press .

Iwata, S. and Kasai, T. (1994) 'The Othello game on an n × n board is PSPACE-complete', *Theoretical Computer Science* , vol. 123, no. 2, pp. 329 - 340.

Jaszkiewicz, A. (2002) 'Genetic Local Search for Multi-Objective Combinatorial Optimisation', *European Journal of Operational Research*, vol. 137, no. 1, pp. 50-71.

Josephson, J.R. (1998) 'An Architecture for Exploring Large Design Spaces', proc Nat. Conf. AAAI, pp. 143-150.

Jozefowska, J., Mika, M., Rozycki, R., Waligora, G. and Weglarz, J. (2000) 'Solving the discrete-continuous project scheduling problem via its discretization', *Math. Methods Operat. Res.*, vol. 52, pp. 489-499.

Kendall, G. and Hussin, N.M. (2005) 'A tabu search hyper-heuristic approach to the examination timetabling problem at the MARA University of Technology', Practice And Theory of Automated Timetabling V, Lecture Notes In Computer Science 3616 , pp. 270-293.

Kendall, G. and Mohd Hussin, N. (2005) 'An Investigation of a Tabu-Search-Based Hyper-heuristic for Examination Timetabling', Multidisciplinary Scheduling: Theory and Applications, pp. 309-328.

Kendall, G. and Whitwell, G. (2001) 'An Evolutionary Approach for the Tuning of a Chess Evaluation Funtion using Population Dynamics', Proceedings of the IEEE Congress on Evolutionary Computation.

Kendell, G., Soubeiga, E. and Cowling, P. (n.d) 'Choice Function and Random Hyperheuristic'.

Knuth, D. and Moore., R. (n.d) 'An analysis of alpha-beta pruning', *Artificial Intelligence* , vol. 6, pp. 293-326.

Kocsis, L. and Szepesvari, C. (2006) 'Bandit Based Monte-Carlo Panning', *Proceedings of European Conference on Machine Learning 2006*, vol. LNCS 4212, pp. 282-293.

Kolisch, R. (1996) 'Efficient priority rules for the resource-constrained project scheduling problem.', *Journal of operations management*, vol. 14, no. 3, p. 179.

Kolisch, R. (1996) 'Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation', *European Journal of Operational Research*, vol. 90, no. 2, April , pp. 320-333.

Kolisch, R. and Hartmann, S. (1999) 'Heuristic Algorithms for Solving the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis'.

Kolisch, R. and Hartmann, S. (2006) 'Experimental Investigations of Heuristics for RCPSP: An Update', *European Journal of Opererational Research*, vol. 174, no. 1, pp. 23-37.

Kwak, B.J., Song, O.N. and Miller, L.E. (2005) 'Performance Analysis of Exponential Backoff', *IEE-ACM Transactions on Networking*, vol. 13, no. 2, pp. 343-355.

Laguna, M., Marti, R. and Campos, V. (1999) 'Intensification and Diversification with elite tabu search solutions for the linear ordering problem', *Computers & Operations Research*, vol. 26, no. 12, OCT, pp. 1217-1230.

Lawler, E., Lenstra, J., Rinnooy Kan, A. and Shmoys, D. (1985) *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Chichester: Wiley.

Leyuan, S. and Yunpeng, P. (2005) 'An efficient search method for job-shop scheduling problems', *IEEE Transactions on Automation Science and Engineering*, vol. 2, no. 1, pp. 73 - 77.

Lin, S. (1965) 'Computer solutions of the traveling salesman problem', *Bell Systems Technical Journal*, vol. 44, pp. 2245-2269.

Lucas, S.M. and Runarsson, T.P. (2006) 'Temporal Difference Learning Versus Co-Evolution for Acquiring Othello Position Evaluation', IEEE Symposium on Computational Intelligence and Games, pp. 52-59.

MacCarthy, B.L. and Wilson, J.R. (2001) *Human Performance in Planning and Scheduling*, Taylor & Francis.

Matthews, J. *Virus Game Project*, [Online], Available: http://www.generation5.org/content/2000/virus.asp [27 June 2008].

Metcalfe, R.M. and Boggs, D.R. (1976) 'Ethernet: Distributed Packet Switching for Local Computer Networks', *Coms. of the ACM*, vol. 19, no. 5, pp. 395-404.

Mitchell, T. (1997) 'Machine Learning' McGraw Hill.

Mladenovic, N. and Hansen, P. (1997) 'Variable Neighborhood Search', *Computers & Operational Research 24 (11)*, pp. 1097-1100.

Mori, M. and Tseng, C.C. (1997) 'A genetic algorithm for multi-mode resource constrained project scheduling problem', *European Journal of Operational Research*, vol. 100, pp. 134-141.

Oppen, J. and Lokketangen, A. (2008) 'A tabu search approach for the livestock collection problem', *Comp. & Ops. Res.* , vol. 35, no. 10, pp. 3213-3229.

Özcan, E., Birben, M., Bykov, Y. and Burke., E. (2009) 'Examination timetabling using late acceptance hyper-heuristics.', Proceedings of the 2009 IEEE Congress on Evolutionary Computation, pp. 997-1004.

Ozdamar, L. (1999) 'A Genetic Algorithm Approach to a General Category Project Scheduling', *IEEE Transactions on Systems, Man and Cybernetics*, vol. 29, no. 1, pp. 44-59.

Pinedo, M. and Chao, X. (1999) *Operations scheduling with applications in manufacturing and services.*, McGraw-Hill Companies.

Poli, R. and Graff, M. (2009) 'There Is a Free Lunch for Hyper-Heuristics, Genetic Programming and Computer Scientists', Proceedings of the European

Conference on Genetic Programming, Lecutre Notes in Computer Science 5481, pp. 195-207.

Potgieter, G. and Engelbrecht, A.P. (2007) 'Genetic Algorithms for the Structure Optimisation of learned Polynomial Expressions.', *Applied Mathematics and Computation*, vol. 186, no. 2, pp. 1441-1466.

Rattray, M. and Shapiro, J. (1997) 'Noisy Fitness Evaluation in Genetic Algorithms and the Dynamics of Learning', Foundations of Genetic Algorithms 4, pp. 117-139.

Reeves, C. (1995) 'Genetic Algorithms and Combinatorial Optimization.', in Rayward-Smith, V. *Applications of Modern Heuristic Methods*, Henley-on-Thames: Alfred Waller.

Remde, S., Cowling, P., Daha, K. and Colledge, N. (2009) 'Binary Exponential Back Off for Tabu Tenure in Hyperheuristics', Proceedings of EvoCOP 2009, Springer LNCS 5482, pp. 109-120.

Remde, S., Cowling, P., Dahal, K. and Colledge, N. (2007) 'Bottleneck Avoidance by Fitness Function', Informatics Workshop, University of Bradford, pp. 44-48.

Remde, S., Cowling, P., Dahal, K. and Colledge, N. (2007) 'Exact/Heuristic Hybrids using rVNS and Hyperheuristics for Workforce Scheduling', Proceedings of Evolutionary Computation in Combinatorial Optimization, Lecture Notes in Computer Science 4446, pp. 188-197.

Remde, S., Cowling, P., Dahal, K. and Colledge, N. (2008) 'Evolution of Fitness Function to Improve Heuristic Performance"', Proceedings of Learning and Intelligent Optimization (LION) II, Springer LNCS 5313, pp. 206-219.

Remde, S., Cowling, P., Dahal, K. and Colledge, N. (Submitted) 'Binary Exponential Back Off Experimental Investigation and Comparison', *Journal of the Operational Research Society on Heuristic Optimisation*.

Rolland, E., Schilling, D. and Current, J. (1996) 'An efficient tabu search procedure for the p-median problem', *European Journal of Operational Research*, vol. 96, pp. 329-342.

Schumacher, C., Vose, M. and Whitley, L. (2001) 'The no free lunch and problem description length.', Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), p. 565–570.

Sevkli, M. and Aydin, M. (2006) 'A variable neighbourhood search algorithm for job shop scheduling problems', *Lecture Notes in Computer Science*, vol. 3906, pp. 261-271.

Shimojika, K., Fukuda and T., H.Y. (1995) 'Self-Tuning Fuzzy Modeling with adaptive membership function, rules, and hierarchical structure-based on Genetic Algorithm', *Fuzzy Sets And Systems* , vol. 71, no. 3, pp. 295-309.

Sprecher, A., Kolisch, R. and Drexl, A. (1995) 'Semi-active, active, and non-delay schedules for the resource-constrained project scheduling problem', *European Journal of Operational Research*, vol. 80, no. 1, pp. 94-102.

Sutton, R.S. and Barto, A.G. (1998) 'Reinforcement Learning: An Introduction' Cambridge: MIT Press.

Taillard, E. (1991) 'Robust taboo search for the quadratic assignment problem', *Parallel computing*, vol. 17, p. 443.

Taillard, E., Gambardella, L., Gendreau, M. and Potvin, J. (2001) 'Adaptive memory programming: A unified view of metaheuristics"', *European Journal of Operational Research*, vol. 135, no. 1, NOV, pp. 1-16.

Terashima-Marin, H., Ross, P. and Valenzuela-Rendon, M. (1999) 'Evolution of Constraints Satisfaction Strategies in Examination Timetabling', *Proceedings of the Genetic and Evolutionary Computationally Conference*, pp. 635-642.

Thiagarajan, S. and Rajendran, C. (2005) 'Scheduling in dynamic assembly job-shops to minimize the sum of weighted earliness, weighted tardiness and weighted flowtime of jobs', *Comp. Ind. Engrg.*, vol. 49, no. 4, pp. 463-503.

Thompson, P.M. and Orlin, J.B. (1989) *The theory of cyclic transfers.*, No. OR 200-89, Operations Research Center, MIT, Cambridge, MA.

Tormos, P. and Lova, A. (2001) 'A competitive heuristic solution techinique for resource constrained project scheduling', *Annal of Operational Research*, vol. 102, pp. 65-81.

Tormos, P. and Lova, A. (2003) 'An efficient multi-pass heuristic for project scheduling with constrained resources', *International Journal of Production Research*, vol. 51, no. 5, pp. 1071-1086.

Tormos, P. and Lova, A. (2003) 'Integrating Hueristics for Resource Constrained Project Scheduling: One Step Further', Technical report, Department of Statistics and Operations Research, Universidad Politécnica de Valencia, Valencia.

Tsang, E. and Voudouris, C. (1997) 'Fast local search and guided local search and their application to British Telecom's workforce scheduling problem', *Operations Research Letters*, vol. 20, no. 3, pp. 119-127.

Ulusoy, G. and Ozdamar, L. (1989) 'Heuristic Performance and Network/Resource Characteristics in Resource-Constrained Project Scheduling', *Journal of the Operational Research Society*, vol. 40, pp. 1145-1152.

Valls, V., Ballestin, F. and Quintanilla, S. (2003) 'Justification and RCPSP: A technique that pays', *European Journal Of Operational Research*, vol. 165, no. 2, pp. 375-386.

Vanesian, T., Kreutz-Delgado, K., Burgin, G. and Fogel, D. (2007) 'Algorithmic Tools for Adversarial Games: Intent Analysis Using Evolutionary Computation', IEEE Symposium on Computational Intelligence in Security and Defense Applications, pp. 108-115.

Verhoeven, M. (1998) 'Tabu search for resource-constrained scheduling', *European Journal Of Operational Research*, vol. 106, no. 2-3, APR, pp. 266-276.

Viana, A. and de Sousa, J. (2000) 'Using metaheuristics in multiobjective resource constrained project scheduling', *Eur. J. Op. Res.*, vol. 120, no. 2, pp. 359-374.

Viana, A. and de Sousa, J. (n.d) 'Using metaheuristics in multiobjective resource constrained project scheduling', *Eur. J. Op. Res.* , vol. 120, no. 2, pp. 359-374.

Vieira, G.E., Herrmann, J.W. and Lin, E. (2003) 'Rescheduling manufacturing systems: a framework of strategies, policies and methods', *Journal of Scheduling*, vol. 6, no. 1, pp. 39-62.

Voudouris, C. and Tsang, E. (1999) 'Guided Local Search and its application to the traveling salesman problem', *European Journal of Operational Research*, vol. 113, no. 2, pp. 469-499.

Whitley, D., Starkweather, T. and Shaner, D. (1991) 'The travelling salesman and sequence scheduling: Quality solutions using genetic edge recombination.', in *Handbook of Genetic Algorithms*, New York: Van Nostrand Reinhold.

Whitley, D. and Watson, J. (2005) 'Complexity theory and the no free lunch theorem', in Burke, E. and Kendall, G. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, Springer, US.

Wolpert, D. and Macready, W. (1997) 'No free lunch theorems for optimization. ', *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, p. 67–82.

Wolpert, D. and Macready, W. (2005) 'Coevolutionary free lunches', *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 6, pp. 721-735.

Yang, M. (2001) 'An Efficient Algorithm to Allocate Shelf Space', *European Journal of Operational Research*, vol. 131, pp. 107-118.

Yao, X. (1999) 'Evolving artificial neural networks', *Proceedings Of The IEEE* , vol. 87, no. 9, pp. 1423-1447.