The Bees Algorithms for the Vehicle Routing Problem

Michael J. Dinneen Aisha J.L. Fenton

Department of Computer Science
University of Auckland
Auckland, New Zealand

Abstract

We examine the Pickup and Delivery problem for Full truck load vehicles - an important problem in moving bulk freight. We provide background on the related problems VRP and PDP as well as providing a brief survey of the more common heuristics for these problems. We provide an algorithm based on adapting classic heuristics and provide a set of test instances to benchmark the problem against. We discuss implementation challenges and solutions and present a real-world case study.

Contents

*

Chapter 1

Introduction

TODO

1.1 Motivation

1.1.1 Road transport industry

Chapter 2

Background

This chapter provides a short history and background material on the Vehicle Routing Problem. In particular we review the solution methods that been brought to bear on the Vehicle Routing Problem and some of the classic results reported in the literature.

This chapter is laid out as follows. We start in section 2.1 by informally defining what the Vehicle Routing Problem is and by providing a timeline of the major milestones in its research. We also reviewing here a closely related problem, the Traveling Salesman Problem, which is a cornerstone of the Vehicle Routing Problem. We then review in section 2.2 the *Exact Methods* that have been developed to solve the Vehicle Routing Problem. These are distinguished from the other methods we review in that they provide exact solutions, where the globally best answer is produced. We follow this in section 2.3 by reviewing the *Classic Heuristics* that have been developed for the Vehicle Routing Problem. These methods aren't guaranteed to find the globally best answer, but rather aim to produce close to optimal solutions using algorithms with fast running times that are able to scale to large problem instances. In section 2.4 we review *Meta-heuristic* methods that have been adapted for the Vehicle Routing Problem. These methods provide some of the most competitive results available for solving the Vehicle Routing Problem and are considered the state-of-the-art currently. Lastly in section 2.5 we review a modern family of meta-heuristics called *Swarm Intelligence* that has been inspired by the problem solving abilities exhibited by some groups of animals and natural processes. These last methods have become a popular area of research recently and are starting to produce competitive results to many problems. This thesis uses a Swarm Intelligence method for solving the Vehicle Routing Problem.

2.1 VRP overview

The Vehicle Routing Problem (VRP) seeks to solve the problem of assigning and ordering work for a finite number of resources, such that the cost of undertaking that work is minimized. Often the context used is that of a fleet of vehicles delivering goods to a set of customers, although the problem can equally be applied across many different industries and scenarios (including non-logistics scenarios, such as microchip layout). The aim is split the deliveries between the vehicles and to specify an order in which each vehicle undertakes the work, such that the distance travelled by them is minimized and any pre-stated constraints are met. In the classic version of the VRP the constraints that must be met are:

1. Each vehicle must start and end its route at the depot.
2. All goods must be delivered.
3. The goods can only be dropped off a single time and by a single vehicle.
4. Each good requires a specified amount of capacity. However, each vehicle has a finite amount of capacity that can't be exceeded. This adds to the complexity of the problem as this necessary influences the selection of deliveries assigned to each vehicle.

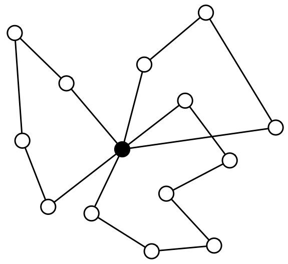

Figure 2.1: *An example of customers being assigned to three selected vehicle routes. The depot is the black dot in the centre.*

More formally the VRP can be represented as a graph (V, E). The vertices of the graph V represent all locations that can be visited, this includes each customers location and the location of the depot. For convenience let v^d denote the vertex that represents the depot. We denote the set of customers as $C = 1, 2, ..., n$. Now let the set of edges E corresponds to valid connections between customers and connections to the depot – in the typical case all connections are possible. Each edge $(i, j) \in E$ has a corresponding cost c_{ij}. The cost is typically the travel distance between the two locations.

A solution to the given VRP instance can be represented as a family of 'routes', denoted by $\mathfrak{S}$. Each route itself is a sequence of customer visits that are performed by a single vehicle, denoted by $R = [v_1, v_2, ..., v_k]$ such that $v_i \in V$, and v_1 and v_k are always the depot. Each customer has a demand $d_i, i \in C$. And q is the maximum demand permissible for any route (i.e. the maximum capacity). The cost of the solution, and the value we aim to minimize, is given by the following formula:

$$\sum_{R \in \mathfrak{S}} \sum_{v_i \in R} c_{v_i, v_{i+1}}$$

We can now formalize the VRP constraints as follows:

$$\bigcup_{R \in \mathfrak{S}} = V \tag{2.1}$$

$$(R_i - v^d) \cap R_j = \emptyset \qquad \forall R_i, R_j \in \mathfrak{S} \tag{2.2}$$

$$v_i = v_j \qquad \forall R_i \in \mathfrak{S}, \neg\exists v_i, v_j \in (R_i - v^d) \tag{2.3}$$

$$v_0, v_k \in R_i = v^d \qquad \forall R_i \in \mathfrak{S} \tag{2.4}$$

$$\sum_{v \in R_i} d_v < q \qquad \forall R_i \in \mathfrak{S} \tag{2.5}$$

Equation 2.1 specifies that all customers are included in at least one route. 2.2 and 2.3 ensure that each customer is only visited once, across all routes. 2.4 ensures that each route starts and end with at the depot. Lastly 2.5 ensures that each route doesn't exceed it's capacity. This version of the problem has come to be known as the Capacitated Vehicle Routing Problem (often appreciated to VRP in the literature). See chapter 3 for an alternative formation, which states the problem as an integer linear programming problem, as is standard in the VRP literature.

VRP was first formally introduced in 1959 by Dantzig and Ramser in [9] – the original name used in their paper was the Truck Scheduling Problem. The VRP has remained an important problem in logistics and transport, and is one of the most the studied of all combinatorial optimization problems. Hundreds of papers have been written on it over the 50 period. From the large number of implementations carried out over this period it has been shown that VRP has real benefits to offer transport and logistics companies. Anywhere from 5% to 20% savings have been reported where a vehicle routing procedure has been implemented [39].

From the VRP comes a family of related problems. These problems model other constraints that are encountered in real-world applications of the VRP. Classic problems include: VRP with Time Windows, (appreciated to VRPTW) that introduces a time window constraints against each customer, that the vehicle must arrive within. VRP with Multiple Depots (appreciated to MDVRP), where the vehicles are dispatched from multiple starting points. And finally the Pickup and Delivery Problem (appreciated to PDP), where goods are both picked up delivered during the course of the route (such as a courier would do).

== PIC OF TIMELINE ==

2.1.1 TSP introduction and history

The VRP is can be seen as a combination two combinatorial hard problems, the Traveling Salesman Problem (more precisely the MTSP), and the Bin Packing Problem, BPP.

The Traveling Salesman Problem (commonly abbreviated to TSP) can informally be defined as follows: given n points on a map provide a route through each of the n points such that each point is only used once and the total distance travelled is minimized. The problem's name, the *Traveling Salesman*, comes from the classic real world example of the problem. A salesman is sent on a trip to visit n cities. They must select which order to visit the cities in, such that the least amount of distance needs to be traversed.

Although the problem sounds like it might be easily solvable, it is in fact $\mathcal{NP}$-hard. The best known exact algorithms for solving the TSP still require a running time of $O(2^n)$. Karp's famous paper, Reducibility Among Combinatorial Problems, in 1972 showed that the Hamiltonian Circuit problem is $\mathcal{NP}$-complete. This implied the NP-hardness of TSP, and thus supplied the mathematical explanation for the apparent difficulty of finding optimal tours.

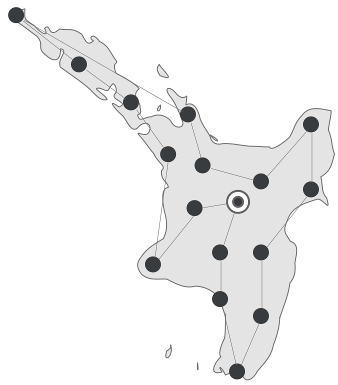

Figure 2.2: *Shown is an example of a 16 city TSP tour around New Zealand's North Island. This tour is one of 20,922,789,888,000 possible tours for these 16 cities.*

TSP has a history reaching back many years. It is itself related to another classic graph theory problem, the Hamiltonian circuit. Hamiltonian circuits have been studied since 1856 by both Hamilton [13] and Kirkman [18]. Whereas the Traveling Salesman Problem has been informally discussed for many years [34] it didn't become actively studied until after 1928, where Menger, Whitney, Flood and Robinson produced much of the early results in the field. Robinson's RAND report [32] is probably the first article to call the problem by the name it has since become known, the Traveling Salesman Problem.

> The purpose of this note is to give a method for solving a problem related to the traveling salesman problem. One formulation is to find the shortest route for a salesman starting from Washington, visiting all the state capitals and then returning to Washington. More generally, to find the shortest closed curve containing n given points in the plane.

An early result was provided by Dantzig, Fulkerson, and Johnson [8]. Their paper gave an exact method for solving a 49 city problem, a large number of cities for the time. Their algorithm used the cutting plane method to provide an exact solution. This approach has been the inspiration for many subsequent approaches, and is still the bedrock of algorithms that attempt to provide an exact solution.

A generalization of TSPis MTSP, where multiple routes are allowed to be constructed (i.e. multiple salesman can be used to visit the cities). The pure MTSP can trivially be turned into a TSP by constructing a graph G with $n-1$ additional copies of the starting node to the graph and by forbidding travel directly between these n starting nodes. Note however that the pure formulation of MTSP places no additional constraints on how the routes are constructed. Real life applications of the MTSP typically require additional constraints, such as limiting the size or duration of each route (i.e. one salesman shouldn't be working a 12 hour shift, while another has no work).

MTSP leads us naturally into the a family of problems given by the Vehicle Routing Problem (VRP). VRP– and it's family of related problems – can be understood as being a generalization of MTSP that incorporates additional constraints. Some of these constraints, such as introducing capacity limits, introduce other dimensions to the problem that are in themselves hard combinatorial problems.

2.2 Exact methods

The first efforts at providing solutions to the VRP were concerned with exact methods. These started by sharing many of the same techniques brought to bear on TSP. We follow Laporte and Nobert's survey [19] and classify exact algorithms for the VRP into three families: Direct tree search methods, Dynamic programming, and Integer linear programming.

The first classic Direct tree search results are due to Christolds and Ellison. Their 1969 paper in Operations Research Quarterly provided the first branch and bound algorithm for exactly solving the VRP[7]. Unfortunately it's time and memory requirements where such that it was only able to solve problems of up to 13 customers. This result was later improved upon by Christolds in 1976 by using a different branch model. This improvement allowed him to solve for up to 31 customers.

Christofides, Mingozzi, and Toth, [6] provide a lower bound method that is sufficiently quick (in terms of runtime performance) to be used to as a lower bound for excluding nodes from the search tree. They used this method to provide solutions as for a number of problems containing between 15 to 25 customers. Laporte, Mercure and Nobert [**?**] used MTSP as a relaxation of VRP within a branch and bound framework to provide solutions for 'realisticly' sized problems containing up to 250 customers.

The Dynamic Programming approach was first proposed for VRP by Eilon, Watson-Gandy and Christofides (1971). Their approach allowed them to solve exactly for problems of 10 to 25 customers. Since then, Christofides has made improvements to this algorithm to solve exactly for problems up to 50 vertices large.

A Set Partitioning algorithm was given by Balinski, and Quandt in 1964 to produce exact VRP solutions [3]. The problem sets they used where very small however, having only between 5 to 15 customers. And even despite this they weren't able to produce an solution for some of the problems. However, taking their approach as a starting point many authors have been able to produce more powerful methods. Rao and Zionts (1968), Foster and Ryan (1976), Orloff (1976), Desrosiers, Soumis and Desrochers (1984), Agarwal, Mathur and Salkin (1989), and Desrochers, Desrosiers and Solomon (1990), have all extended the basic set partitioning algorithm, using the Column Generation method, to produce more practically useful results.

Notwithstanding the above results, exact methods have been of more use in advancing theoretical understanding of VRP than to providing solutions to real life problems. This can mostly be attributed to the fact that real-life VRP instances often involve hundreds of customers, and involve richer constraints than are modeled in a simple VRP.

2.3 Classic heuristics

In this section we review the classic heuristic methods that have been developed for the VRP. These methods aren't guaranteed to find the globally best answer, but rather aim to produce close to optimal solutions using algorithms with fast running times that are able to scale to large problem instances. Classic heuristics for the VRP can be classified into three families. Constructive heuristics; Two-phase heuristics, which is again divided into two sub-families: cluster first and then route, and route first and then cluster; and Improvement methods.

2.3.1 Constructive heuristics

We start by looking at *constructive* heuristics. Constructive heuristics build a solutions from the ground up. They typically provide a recipe for building each route, such that the total cost of all routes is minimized.

A trivial but intuitive constructive heuristic is the *Nearest Neighbour* method. In this method routes are built-up sequentially. At each step the customer nearest to the last routed customer is chosen. This continues until the route reaches its maximum capacity, at which point a new route is started. In practice the Nearest Neighbour algorithm tends to provide poor results and is rarely used.

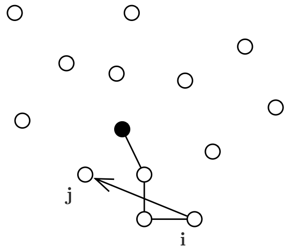

Figure 2.3: *Shown is an example of the Nearest Neighbour method being applied. A partially constructed route selects customer j to add, as it's closest to the last added customer, i*

An early and influential result was given by Clarke and Wright in their 1964 paper [11]. In their paper they presented a heuristic method that has since become known as the *Clarke Wright Savings* heuristic. The method improved upon Dantzig and Ramser's earlier work. The heuristic is based on the simple premise of iteratively combining routes in order of those pairs that provide the largest saving.

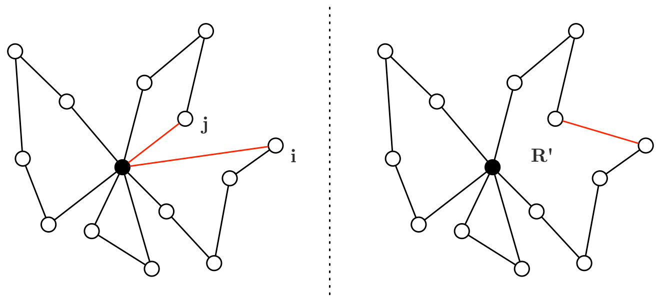

Figure 2.4: *Clark Wright Savings Algorithm. Customers i, j are selected candidate to merge. The merge results in a new route R'*

The algorithm works as follows:

Algorithm 1: Clark Write Savings Algorithm

initializeRoutes() M = savingsMatrix(V)
L = sortBySavings(SM) **for** $l_{ij} \leftarrow L$ **do**
  R^i, R^j = findRoutes(l_{ij}) **if** *feasibleMerge(R^i, R^j)* **then**
    combineRoute(R^i, R^j)
  end
end

The algorithm starts by initializing a candidate solution. For this it creates a route $R = [v_0, v_i, v_0]$ for all $v \in V$. It then calculates a matrix M that contains the savings $s_{ij} =$

$c_{i0} + c_{j0} - c_{ij}$ for all edges $(i, j) \in E$. It then produces a list L that enumerates each cell i, j of the matrix in descending order of the savings. For each entry in the list $l_{ij} \in L$ select the two routes, R^i, R^j that contain customers $i, j \in V$ and test to see if the merge is permissible. The merge is permissible iff:

1. $R^i \neq R^j$.
2. i, j are the first or last vertices (excluding the depot v^d) of their respective routes.
3. The combined demand of the two routes doesn't exceed the maximum allowed q.

The heuristic comes in two flavours, sequential and parallel. The sequential version adds the additional constraint the only one route can be constructed at a time. In this case one of R^i, R^j must be the current route under construction. If l_{ij} results in two new routes then this list item is ignored. If the merge is permissible then we merge routes R^i, R^j such that $R' = [v_0, ..., i, j, ..., v_k]$. In the parallel version, once the entire list of savings has been enumerated then the resulting candidate solution is returned as the answer. In the sequential version the *for loop* is repeated until no feasible merges remain.

The Clark Write Savings heuristic has been used to solve problems of up to 1000 customers with results often within 10% of optimal using only a 180 seconds of runtime [39]. The parallel version of the Clark Write Savings Algorithm outperforms the sequential version in most cases[12] and is typically the one employed.

The heuristic has proven to be surprising adaptable and has been extended to deal with more specialized vehicle routing problems where additional objectives and constraints must be factored in. Its flexibility is a result of its algebraic treatment of the problem [12]. Unlike many other VRP heuristics that exploit the problem's spatial properties (such as many of the two-phase heuristics 2.3.1), the savings formula can easily be adapted to take into consideration other objectives. An example of this is Solomon's equally ubiquitous algorithm [36] which extends the Clark Wright Savings algorithm to cater for time constraints.

This classic algorithm has been extended by Gaskell (1967), Yellow (1970) and Paessens (1988), who have suggested alternatives to the savings formulas used by Clarke and Wright. These approaches typically introduce additional parameters to guide the algorithm towards selecting routes with geometric properties that are likely to produce better combinations. Alkinkemer and Gavish provide an interesting variation on the basic savings heuristic [17]. They use a matching algorithm to combine multiple routes in each step. To do this they construct a graph such that each vertex represents a route, each edge represents a feasible saving, and the edge's weight represent the saving that can be realized by the merge of the two routes. The algorithm proceeds by solving a maximum cost weighted matching of the graph.

== IF MORE CONTENT NEEDED, THEN TALK ABOUT Sequential Insertion Heuristics ==

2.3.2 Two-phase heuristics

We next look at two-phase heuristics. We start by looking at the cluster-first, route second sub-family. One of the foundational algorithms for this method is due to Gillett and Miller who provided a new approach called the Sweep Algorithm in their 1974 paper[4]. This popularized the two-phase approach, although this method was suggested earlier by Wren in his 1971 book, and subsequently in Wren and Holliday's 1972 paper for Operations Research Quarterly. In this approach, a initial clustering phase is used to cluster the customers into a base set of routes. From here the routes are treated as separate TSP and optimized accordingly. The two-phase approach typically doesn't prescribe a method for how the TSP is solved and assumes that already developed TSP methods can be used. The classic sweep algorithm uses a simple geometric method to cluster the customers. Routes are built by sweeping a ray, centered at the depot, clockwise around the space enclosing the problem's locations. The Sweep method is surprising effective and has been shown to solve several benchmark VRP problems to within 2% to 9% of the best known solutions [39].

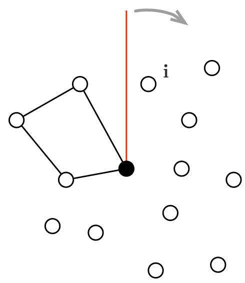

Figure 2.5: *This diagram shows an example of the Sweep process being run. The ray is swept clockwise around the geographic area. In this example one route has already been formed, and a second is about to start at customer i*

Fisher and Jaikumars's 1981 paper [**?**] builds upon the two-phase approach by providing a more sophisticated clustering method. They solve a General Assignment Problem to form the clusters instead. A limitation of their method is that the amount of vehicle routes must be fixed up front. Their method often produces results that are 1% to 2% better than similar results produced by the classic sweep algorithm [39].

Christofides, Mingozzi, and Toth expanded upon this approach in [**?**] and proposed a method that uses a truncated branch and bound technique (similar to Christofiedes Exact method). At each step it builds a collection of feasible routes containing a customer i for evaluation. It then evaluates each route and selects the route that the TSPwith the shortest distance can be produced from.

The Petal algorithm is a natural extension to the Sweep Algorithm. It was first proposed by Balinski and Quandt [**?**] and then extended by Foster and Ryan [**?**]. The basic process is to produce a collection of overlapping candidate routes (called petals) and then to solve a set partition problem to produce a feasible solution. As with other two-phase approaches it's assumed that the order of the customers within each routes is solved using any pick of existing TSPheuristics. It has produced some competitive results for small solutions, but quickly becomes impractical where the set of candidate routes that must be considered is large.

Lastly, there are route first, cluster second methods. The basic premise of these techniques are to first construct a "grand" TSP tour such that all customers are visited. The second phase is then concerned with splitting this tour into feasible routes. Route first, cluster second methods are generally thought to be less competitive than other methods [12], although interestingly Haimovich and Rinnooy Kan have shown that if all customers have unit demand then a simple shortest path algorithm (which can be solved in polynomial time) can be used to produce a solution from a TSP tour that is asymptotically optimal [23].

2.3.3 Iterative Improvement Heuristics

Iterative Improvement methods follow an approach where an initial candidate solution is iteratively improved by applying a single operation. The operation applied is typically fairly simple and only changes a small part of the candidate solution, such as the position of a single customer or edge within the solution. The set of solutions that is obtainable from a current candidate solution S by applying an operator Op is known as S's neighbourhood. In *Iterative Improvement* heuristics, a new solution S' is selected by exhaustively searching the entire neighbourhood of S for the best improvement possible. If no improvement can be found then the heuristic terminates. The initial candidate solution, that is the starting point, can be randomly selected or can be produced using another heuristic (for this the Constructive Heuristics are typically used, see 2.3).

Probably one of the best known improvement operators is $2-opt$. The $2-opt$ operator takes two edges $(i,j),(k,l) \in T$, where T are the edges traversed by a particular route $R = [v_1, ..., v_i, v_j, ..., v_k, v_l, ..., v_n]$, and removes these from the candidate solution. This splits the route into two disconnected components, $D_1 = [v_j, ..., v_k], D_2 = [v_1, ..., v_i, v_l, ..., v_n]$. A new candidate solution is produced by reconnecting D_1 to D_2 using the same vertices i, j, l, k but with alternate edges, such that $(i,j),(j,l) \in T$.

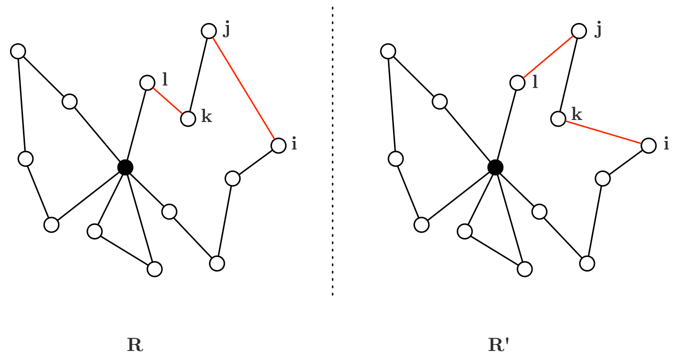

Figure 2.6: *Shows $2-Opt$ being applied to a candidate solution R and producing a new solution R'. In this example edges $(i,j),(k,l)$ are exchanged with edges $(i,k),(j,l)$*

The rational behind $2-Opt$ is that due to the triangle inequality edges that cross themselves are unlikely to be optimal. $2-Opt$ aims to detangle the route.

There are a number of other operations suggested in the literature. Christofides and Eilon give one of the earliest iterative improvement methods in their paper [7]. In this they made a simple change to $2-opt$ to increase the amount of edges removed to three - the operation fittingly being called $3-opt$. They found that their heuristic produced better results than the $2-opt$ could by itself.

In general, operations such as $3-opt$, that remove edges and then search for a more optimal recombination of components take $O(n^y)$ where y is the number of edges removed. A profitable strain of research has been on producing operations that reduce the amount of recombinations that must be searched. Or presents an operation that has since come to be known as $Or-Opt$ [27]. $Or-Opt$ is a restricted $3-opt$. It searches for a relocation of all sets of 3 consecutive vertices (i.e. chains), such that an improvement is made. If an improvement can't be made then it tries again with chains of 2 consecutive vertices, and so on. $Or-Opt$ has been shown to produce similar results to that of $3-Opt$, but with a running time of $O(n^2)$. More recently Renaud, Boctor, and Laptorte have presented a restricted version of $4-Opt$ (in a similar vain to $Or-Opt$) that has a running time of $O(wn^2)$ [16].

== CAN ADD SECTION ON OR-OPT HERE IF REQUIRED

Iterative improvement heuristics are often used in combination with the other techniques. In this case they are run on the candidate solution after the initial heuristic has completed. However, if used in this way there is often a fine balance between producing an operation that improves a solution, and one that is sufficiently destructive enough to escape a local minimum. Interest in Iterative Improvement heuristics has grown as the operations developed for them, such as $Or-Opt$, are directly applicable to more modern heuristics such as the family of meta-heuristic presented in the next section.

2.4 Meta-heuristics

Meta-Heuristics are a broad collection of methods that make few or no assumptions about the type problem being solved. They provide a framework that allows for individual problems to be modeled and 'plugged in' to the meta-heuristic. Typically meta-heuristics take an approach where a candidate solution (or solutions) is initially produced and then is iteratively refined towards the optimal solution. Intuitively meta-heuristics can be thought of searching a problem's search space. Each iteration searches the neighbourhood of the current candidate solution(s) looking for new candidate solutions that move closer to the goal.

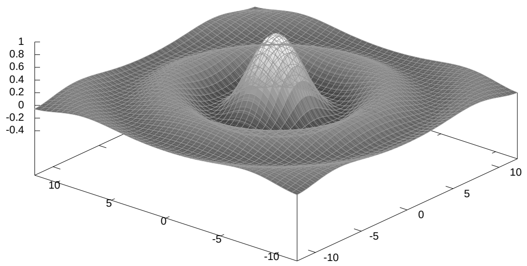

Figure 2.7: *This diagram shows an example of a search space that a meta-heuristic moves through. In this example the peak at the centre of the figure is the globally best answer, but there are also hills and valleys that the meta-heuristic may become caught in. These are known as local minima and maxima*

A limitation of meta-heuristics is that they aren't guaranteed to find an optimal solution (or even a good candidate!). Moreover the theoretical underpinnings of what makes one meta-heuristic more effective than another are still poorly understood. Meta-heuristics within the literature tend to be tuned for specific problems and then validated empirically.

There have been a number of meta-heuristics produced for the VRP in recent years. Many of the most competitive results produced in the last ten years have been from meta-heuristic approaches. We next review some of the more well known meta-heuristic results for VRP.

2.4.1 Simulated Annealing

Simulated Annealing is inspired by the annealing process used in metallurgy. The algorithm starts with a candidate solution (which can be randomly selected) and then moves to nearby solutions with a probability dependent on the quality of the solution and a global parameter T, which is reduced over the algorithm. In classic implementations the following formula is used to control the probability of a move:

$$e^{-\frac{f(s')-f(s)}{T}}$$

Where $f(s)$ and $f(s')$ represent the solution quality of the current solution, and the new solution respectively. By analogy to the metallurgy process T represents the current temperature of the solution. Initially T is high. This lets the algorithm free itself from any local minimum that it may be caught in. It is then cooled over time forcing the algorithm to converge to a new solution.

One of the first results was given by Robuste, Daganzo and Souleyrette [10]. They define the search neighbourhood all solutions that can be obtained from the current solution by applying one of three operations: relocating part of a route to another position within the same route, and exchanging customers between two routes. They tested their solution on some large real-world instances of up to 500 customers. They self-reported some success with their method but as their test cases are unique no direct comparison is possible.

Osman has given probably the best know Simulated Annealing results for VRP[28]. His algorithm expands upon many areas of the basic Simulated Annealing approach. The method start by using the Clark and Wright algorithm to produce a starting position. It defines its neighbourhood as candidate solutions that can be searched by applying a $\lambda - interchange$ operation.

$\lambda - interchange$ works by selecting two sequences (i.e. chains) of customers C_p, C_q from two routes, R_p and R_q, such that $|C_p|, |C_q| < \lambda$ (note the chains aren't necessary of the same length). The customers within each chain are then exchanged until an exchange produces an infeasible solution. As the neighbourhood produced by $\lambda - interchange$ is typically large Osman restricts λ to ≤ 2 and suggests that the first move that provides an improvement is used rather than exhaustively searching the entire neighbourhood.

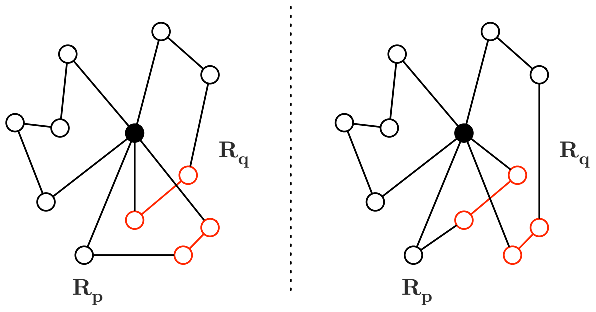

Figure 2.8: *This diagram shows an example of of $\lambda - interchange$ being applied to a candidate solution. Two sequences of customers C_p and C_q are selected from routes R_p, R_q respectively. Customers from C_p are then swapped with C_q where feasible.*

Osman also uses a sophisticated cooling schedule; his main change being that the temperature is cooled continuously while improvements are found, or if no improvement he resets the temperature (using $T_i = max(\frac{T_r}{2}, T_b)$, where T_r is the reset temperature, and T_b is temperature of the best solution found so far).

Although Simulated Annealing has produced some good results, and in many cases outperforms the classic heuristics (compare [12] with [22]), it is not competitive with the tabu search implementation.

2.4.2 Genetic Algorithms

Genetic Algorithms where first proposed in [14]. They've since been applied to many problem domains and are particularly well suited to applications that must work across a few distinct problems. In fact they were the first evolutionary inspired algorithms to be applied to combinatorial problems [31]. The basic operation of a GA is as follows:

Algorithm 2: Simple Genetic Algorithm

```
Generate the initial population
while termination condition not met do
    Evaluate the fitness of each individual
    Select the fittest pairs
    Mate pairs and produce next generation
    Mutate (optional)
end
```

In classic Genetic Algorithms each candidate solution is encoded as a binary string (i.e. chromosome). Each individual (i.e. candidate solution) is initially created randomly and used to seed the population. There are many techniques suggested in the literature for initially 'bootstraping' the population by making use of other heuristics to produce a stronger initial population, however special care must be taken with Genetic Algorithms to ensure that diversity is maintained within the population - otherwise there is a risk of premature convergence.

Next the fittest individuals are selected from the population, and reproduction (crossover) and mutation operations are applyed to them to produce the next generation. A classic crossover operation takes individuals encoded as binary strings and splits them at one or two points. The parts are then recombined to form a new candidate solution. The idea being that the crossover produces a new candidate solution from two successful partial solutions. This process is repeated until a termination condition is met (often a predetermined running time), or until the population has converged on a fitness.

Special consideration needs to be given to how discrete optimisation problems, such as the VRP, are represented and how their operators are constructed. For instance, consider how the classic crossover operation would work on a TSPpath. When two parts of two separate solutions are combined they are likely to contain duplicates. Therefore it is more common for the VRP (and the TSP as well) to use a direct representation, rather than a binary encoding. In this instance the VRP is represented as collection of sequences, each holdings a sequence of customers. The Genetic Algorithm then must use operators specially designed for this representation of the problem.

Two commonly used crossover operators are Order Crossover (OX) and Edge Assembly Crossover (EAX). OX [15] operates by placing two cuts points within each route. The substring between the two cut points is copied from the first parent directly into the offspring. The remainder of the string from the second parent is then copied to the offspring, but with any duplicates removed. This potentially leaves an incomplete solution, where not all the customers have been routed. The solutions is then *repaired* by inserting any remaining customers using an insertion heuristic.

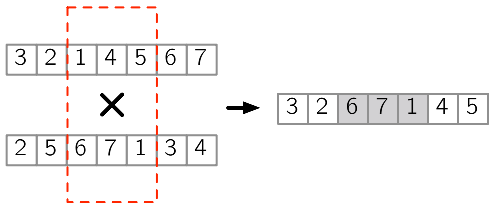

Figure 2.9: *This diagram shows the OX crossover operator being applied to two tours of the customers. A child is produced by taking customers at position* c_3, c_4, c_5 *from the second parent and injecting these into the same position in the first parent, removing any duplicates. This leaves customer* $4, 5$ *unrouted, so these are reinserted back into the child in the order that they appear in the first parent*

Another common crossover operator is EAX. EAX was originally designed for the TSP but has been adapted to the VRP by [25]. EAX operates using the following process:

1. Combine the two candidate solutions into a single graph by merging each solutions edge sets.
2. Create a partition set of the graph's cycles by alternately selecting an edge from each graph.
3. Randomly select a subset of the cycles.
4. Generate a (incomplete) child by taking one of the parents and removing all edges from the selected subset of cycles. Then add back in the edges from the parent that wasn't chosen.
5. Not all cycles in the child are connected to the route. Repair them by iteratively merging the disconnected cycles to the connected cycles.

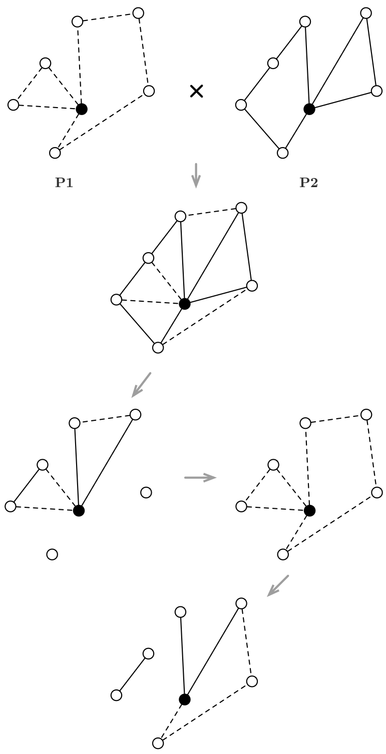

Figure 2.10: *This diagram shows an example of of EAX being applied on two parent solutions, $P1$ and $P2$. The parents are first merged together. Then a new graph is created by selecting alternate edges from each parent $P1$, $P2$. A subset of cycles are then taken and applied to $P1$, such that any edges from $P1$ are removed. The child solution produced is infeasible (it contains broken routes). These would need to be repaired.*

An alternative, and interesting, approach found in the literature is to instead encode a set of operations and parameters that are feed to another heuristic, that in turn produces a candidate solution. An early example of this was suggested by [**?**] who encoded a ordering of the customers. The ordering is then feed into an Insertion heuristic to produce the actual candidate solutions.

An influential adaption of Genetic Algorithms for solving VRPTW was given in [37] with their GIDEON algorithm. GIDEON uses an approach inspired by the sweep heuristic (an overview is provided in section 2.3.2). Routes are built by sweeping a ray, centered at the depot, clockwise around the space enclosing the problem's locations. Customers are collected into candidate routes based on a set of parameters that are refined by the GA. GIDEON uses the GA to evolve these parameters rather than to operate on the problem directly. Finally GIDEON uses a local search method to optimize customers with the routes, making use of the $\lambda - interchange$ operation (see section 2.4.3 for a description of this operator).

Generally speaking Genetic Algorithms haven't been as competitive in VRP as other meta-heuristics. However, more recently there have been two very promising application of Genetic Algorithms to VRP. Nagata [25] has adapted the EAX operator for with the VRP. And Berger and Barkaoui have presented a Hybrid Genetic Algorithm called HGA-VRP in [5]. HGA-VRP adapts a construction heuristic to be used as a crossover operator. The basic idea is that a set of routes are selected from each parent that are located close to each other. Customers are then removed from one parent and inserted into the second using an operation inspired by Solomon's construction heuristic for VRPTW[36].

Both methods have reached the best known solution for a number of the classic VRP benchmark instances by Christofides, Mingozzi and Toth [6] and are competitive with the best Tabu Search methods.

2.4.3 Tabu Search

Tabu Search follows the general approach shared by many meta-heuristics; it iteratively improving a candidate solution by searching for improvements within the current solution's neighborhood. Tabu search starts with a candidate solution, that may be generated randomly or by using another heuristic. Unlike Simulated Annealing, the best improvement within the current neighbourhood is always taken as the next move. This introduces the problem of cycling between candidate solutions. To overcome this Tabu Search introduces a list of solutions that have already been investigated and are forbidden as next moves (hence its name of Tabu List).

The first instance of Tabu Search being used for VRP is by Willard [40]. Willard's approach made use of the fact that VRP instances can be transformed into a MTSP instances and solved. The algorithm makes use of a combination of simple vertex exchange and relocate operations. Although opening the door for further research its results weren't competitive with the best classic heuristics.

Osman gives a more competitive use of Tabu Search in [28]. As with his Simulated Annealing method he makes use of the $\lambda - interchange$ operation to define the search neighbourhood. Osman provides two alternative methods to control how much of the neighbourhood is searched to select the next move - Best-Improvement (BI) and First-Improvement (FI). Best-Improvement searches the entire neighbourhood and selects the move that is the most optimal. First-Improvement searches only until a move is found that is more optimal than the current position. This heuristic produces some competitive results that often out perform the classic heuristics. However it has been refined and improved upon by newer Tabu Search methods.

Toth and Vigo introduced the concept of Granular Tabu Search (GTS) [29]. Their method makes use of a process that removes moves from the neighbourhood that are unlikely to produce good results. They reintroduce these moves back into the process if the algorithm is stuck in a local minimum. Their idea follows from an existing idea known as candidate lists. Toth and Vigo's method has produced many competitive results.

Taillard gave a very successful application of Tabu Search in [38]. Talliard's Tabu Search uses Or's $\lambda - interchange$ as it's neighbourhood structure. It borrows two novel concepts from [21], the use of a more sophisticated tabu mechanism - the duration (or number of iterations) that an item is tabu for is chosen randomly, and a diversification strategy, where vertices that are frequently moved without giving an improvement are penalised. An novel aspect of Taillard's algorithm is it's decomposing of the problem into sub-problems. The problem is split into regions using a simple segmentation of the region centred about the depot (Taillard also provides an alternative approach for problems where the customers are evenly distributed around the depot). From here each subproblem is solved individually, with customers being exchanged between neighboring segments periodically. Taillard observes that exchanging customer beyond neighboring segments is unlikely to produce an improvement, so can safely be ignored. Taillard's method has produced some of the currently best known results for the standard Christofides, Mingozzi and Toth problem sets [6].

2.4.4 Large Neighbourhood Search

Large Neighbourhood Search (commonly abbreviated to LNS) was recently proposed as a heuristic by Shaw [35]. Large Neighbourhood Search is a type heuristic belonging to the family known as Very Large Scale Neighbourhood search (VLSN)[1]. Very Large Scale Neighborhood search is based on a simple premise that rather than searching within a neighborhood of solutions that can be obtained from a single (and typically quite granular) operation, such as $2 - opt$, it might be profitable to consider a much broader neighborhood; A neighbourhood of candidate solutions that are obtained from applying many simultaneous changes to the current solution. What distinguishes these heuristics from others is that the neighborhood under consideration is typically exponentially large, and sometimes infeasible to search. Therefore much attention is given to providing practical methods to search these neighborhoods.

Large Neighbourhood Search uses a Destroy and Repair metaphor for how it searches within its neighbourhood. The basic operation is as follows.

Algorithm 3: Large Neighbourhood Search

```
x = an initial solution
while termination condition not met do
    x^t = x
    destroy(x^t)
    repair(x^t)
    if x_t better than current solution then
        x = x_t
    end
end
Result: x
```

[1]LNS is somewhat confusingly named given that it a type of VLSN, and not a competing approach

Firstly a starting position is generated. This can be done randomly or by using another heuristic. Then for each iteration of the algorithm a new position is generated by destroying part of the candidate solution and then by repairing it. If the new solution is better than the current solution, then this is selected as the new position and loop repeats. This can be seen as being a type of Very Large Scale Neighborhood because at each iteration the number of neighboring solutions that can be built from the partially destroyed solution is exponential on the size of the items removed (i.e. destroyed).

Obviously a key component of this approach are the functions used to destroy and repair the solution. Care must be given to how these functions are constructed. They must pinpoint an improving solution from a very large neighbourhood of candidates, while also providing enough degrees of freedom to escape local minimum.

Empiric evidence in the literature shows that even surprisingly simple functions can be effective (more effective in some cases) [35][33]. In applications of Large Neighbourhood Search to VRP a pair of simple operations are commonly used (alongside more complex ones) for the destroy and repair functions. Specifically, part of the candidate solution is destroyed by randomly selecting and removing n customers. Then it is repaired by finding the least cost reinsertion points back into the solution for the n customers.

Shaw applied Large Neighbourhood Search to VRP in his original paper introducing the method[35]. Shaw introduces a few novel approaches to the destroy and repair functions. His destroy function removes a set of 'related' customers. He defines related customer to be any two customers that share a similar geographic location, that are sequentially routed, or that share a number of similar constraints (such as overlapping time windows if times constraints are used). The idea of removing related customers, over simply removing random customers, is that related customers are more likely to be profitable exchanged (likewise unrelated customers are more likely to be reinserted back in their original positions). Shaw's repair function made use of a simply branch and bound method that finds the minimum cost reinsertion points within the partial solution. His results where immediately impressive and reached the many of the best known solutions on the Christofides, Mingozzi and Toth problems [6].

More recently Ropke proposed an extension to the basic Large Neighbourhood Search process in [33]. His method adds the concept of using a collection of destroy and repairs functions, rather than using a single pair. Which function to use is selected at each iteration based on it's previous performance. In this way the algorithm self adapts to using the most effective function to search the neighbourhood.

Ropke makes use of several destroy functions. He uses a simple random removal heuristic, Shaw's removal heuristic, and a worst removal heuristic, which removes the most costly customers (in terms of that customer's contribution the routes overall cost). Likewise for insertion he makes use of several different functions. These include a simple greedy insertion heuristic, and a novel insertion method he calls the 'regret heuristic'. Informally the regret heuristic reinserts those customers first who are most impacted (in terms of increased cost) by not being inserted into their minimum positions. Specifically let U be the set of customers to be reinserted. let $x_{ik} = \{1, ..., m\}$ be a variable that gives the k'th lowest cost for inserting customer $i \in U$ into the partial solution. Now let $c_i^* = x_{i2} - x_{i1}$, in other words the cost difference between inserting customer i into it's second best position and its first. Now in each iteration of the repair function choose a customer that maximizes:

$$\max_{i \in U} c_i^*$$

Ropke presents a series of results that show that his Large Neighbourhood Search is very competitive for solving the VRP and a large number of related problems, VRP, VRPTW, PDPTW, and DARP). Considering that Large Neighbourhood Search was only proposed in 1998 it has been very successful. In a short space of time it has attracted a large amount of research and has produced some of the most competitive results.

2.5 Swarm intelligence

A recent area of research is in producing heuristics that mimic certain aspects of Swarm behaviour. Probably the most well known heuristics in this class are Particle Swarm Optimisation (PSO) and Ant Colony Optimisation (ACO). Swarm Intelligence is interesting to combinatorial optimisation researches as it demonstrates a form of emergent intelligence, where individual members with limited reasoning capability and simple behaviours, are able to achieve collective goals that are remarkable sophisticated.

In the context of combinatorial optimisation these behaviours can be mimicked and exploited to produce algorithms that are able to produce solutions to complex problems by simulating a number of agents who in themselves only need to perform rudimentary operations. A feature of this class of algorithms is the ease with which they can be parallelized, making them more easily adaptable to large scale problems.

Swarm Intelligence algorithms have been employed to solve a number of problems. We look at two examples here, Ant Colony Optimisation, and the Bees Algorithm, which this thesis makes use of.

2.5.1 Ant Colony Optimisation

Ant Colony Optimization is inspired by how ants forage for food and communicate successful sites back to the colony. Real life initially forage for food randomly. Once they find a food source they return to the colony and in the process lay down a pheromone trail. Other ants that then stumble upon the pheromone trail follow it with a probability dependent on how strong (and old) the pheromone trail is. If they do follow it and find food, then they return too, strengthening the pheromone trail. The strength of the pheromone trail reduces over time meaning that younger and shorter pheromone trails, that do not take as long to traverse, attract more ants.

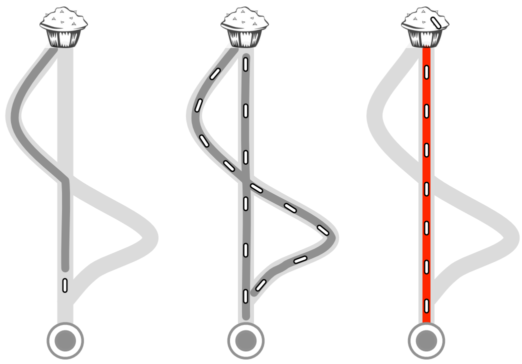

Figure 2.11: *This diagram depicts how ants make use of pheromone trails to optimise their exploitation of local food sources*

Ant Colony Optimisation mimics this behaviour on a graph by simulating ants marching along a graph that represents the problem being solved. The basic operation of the algorithm is as follows:

Algorithm 4: Ant Colony Optimisation

```
Data: A graph representing the problem
while termination condition not met do
    positionAnts()
    while solution being built do
        marchAnts()
    end
    updatePheromones()
end
```

At each iteration of the algorithm the ants are positioned randomly within the graph. The ants are then stochastically marched through the graph until they have completed a candidate solution (in the case of a TSPthis would be a tour of all vertices). At each stage of the march each ant selects their next edge based on the following probability formula:

$$p_{ij}^{k} = \frac{[\tau_{ij}^{\alpha}][\eta_{ij}^{\beta}]}{\sum_{l \in N^k} [\tau_{il}^{\alpha}][\eta_{il}^{\beta}]}$$

Where p_{ij}^{k} is probability that ant k will traverse edge (i,j), N^k is the set of all edges that haven't been traversed by ant k yet, τ is the amount of pheromone that has been deposited at an edge, η is the desirability of an edge (based on a priori knowledge specific to the problem), and α and β are global parameters that control how much influence each term has.

Once the march is complete and a set of candidate solutions have been constructed (by each ant k), update the pheromones deposited on each edge by the follow equation:

$$\tau_{ij} = (1-\rho)\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}$$

Where $0 < \rho \leq 1$ is the pheromone persistence, and $\Delta\tau_{ij}^{k}$ is a function that gives the amount of pheromone deposited by ant k. The function is defined as:

$$\Delta\tau_{ij}^{k} = \begin{cases} 1/C^{k} & \text{if edge } (i,j) \text{ is visited by ant } k \\ 0 & \text{otherwise} \end{cases}$$

Where C^k represents the total distance traveled through the graph by ant k. This ensures that shorter paths result in more pheromone being deposited.

As an example of Ant Colony Optimization's use in combinatorial problems, we show how it can be applied to the TSP. We build a weighted graph with $i \in V$ representing each customer and $(i,j) \in E$ and w_{ij} representing the cost of travel between each customer. And at each stage of the iteration we ensure that the following constraints are met:

- Each customer is visited at most once
- We set η_{ij} to be equal to w_{ij}

When the Ant Colony Optimizer starts it positions each ant at random customer within the graph. Each step of the ant's march then builds a tour through the customers. Once a ant has completed a tour then we have produced a candidate solution for the TSP. The solutions will be of variable quality. We use the length of the tours to ensure that more pheromone is deposited on shorter tours. At the end of n iterations the ants will have converged on an optimal solution (but like all meta-heuristics there's no guarantee that this the global optimum).

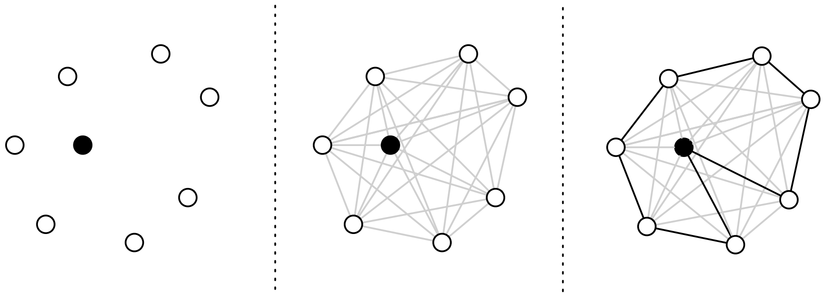

Figure 2.12: *Shown is an example of ACO being used to solve a TSP. Initially the ants explore the entire graph. At the end of each iteration the more optimal tours will have more pheromone deposited on them, meaning in the next iteration the ants are more likely to pick these edges when constructing their tour. Eventually the ants converge on a solution*

Ant Colony Optimization has been applied to VRP by Bullnheimer, Hartl, and Strauss in [1], [2]. They adapted the straight implementation used for the TSP (detailed immediately above) by forcing the ant to create a new route each time it exceeds the capacity or maximum distance constraint. They also use a modified edge selection rule that also takes into account the vehicle's capacity and it's proximity to the depot. The updated rule is now given by:

$$p_{ij}^{k} = \frac{[\tau_{ij}^{\alpha}][\eta_{ij}^{\beta}][s_{ij}][\kappa_{ij}]}{\sum_{l \in N^k}[\tau_{il}^{\alpha}][\eta_{il}^{\beta}][s_{il}][\kappa_{il}]}$$

Where s represents the proximity of customers i, j to the depot, and $\kappa = (Q^i + q^j)/Q$ (Q giving the max capacity, Q^i giving the capacity already used on the vehicle, and q^j is the additional load to be added). κ influences the ants take advantage of available vehicle capacity.

Bullnheimer et al.'s implementations of Ant Colony Optimization for VRP produces good quality solutions for the Christofides, Mingozzi and Toth problems [6], but is not competitive with the best modern meta-heuristics.

More recently Reimann, Stummer, and Doerner have presented a more competitive implementation of Ant Colony Optimization for VRP [24]. Their implementation operates on a graph where $(i, j) \in E$ representing the savings of combining two routes, as given by the classic Clark and Wright savings heuristic [11]. Each ant selects an ordering of how the merges are applied. This implementation is reported to be competitive with the best meta-heuristics [31].

2.5.2 Bees Algorithm

Over the last decade, and inspired by the success of Ant Colony Optimisation, there have been a numerous algorithms proposed that aim to mimic the behaviour of bees. These includes Bee colony optimization that has been applied to many different combinatorial problems, Marriage in honey Bees Optimization (MBO) that has been applied to propositional satisfiability problems, BeeHive that has been applied to timetabling problems, the Virtual Bee Algorithm (VBA) that is used for function optimisation, Honey-bee mating optimisation (HBMO) for cluster analysis, and finally the Bees Algorithm that is the focus of this thesis. See [20] for a bibliography and high level overview on many of these algorithms.

=== CAN EXTEND OUT IF NEEDED ===

The Bees Algorithm was first proposed by [30]. It is inspired by the foraging behaviour of honeys bees. Bee colonies must search a large geographic area around their hive in order to find sites with enough pollen that it can sustain a hive. Its essential that the colony makes the right choices in what sites are exploited and how much resource is expended on a particular site. They achieve this by sending scout bees out in all directions from the hive. Once a scout bee has found a promising site it returns to the hive and recruits hive mates to forage at the site too. The bee does this by performing a *waggle* dance. The dance communicates the location and quality of the site (i.e. fitness). Over time as more bees successfully forage at the site, more are recruited into exploiting that site. This last aspect is similar in process to how ants divert more resource to promising areas when foraging.

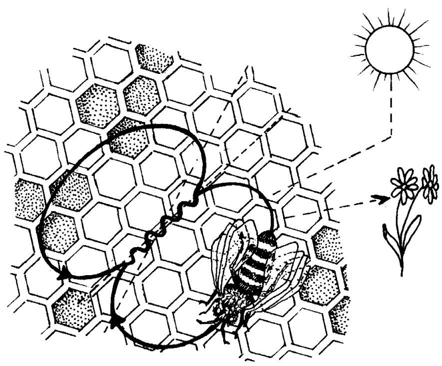

Figure 2.13: *Shown is the* waggle *dance performed a honey bee (image courtesy of [**?**]. The direction moved by the bee indicates the angle that the other bees must fly relative to the sun to find the food source. And the duration of the dance indicates its distance*

Informally the algorithm can be described as follows. Bees are initially sent out to random locations. The fitness of each site is then calculated. A proportion of the bees are reassigned to those sites that had the highest fitness values. Here each bee searches the local neighbor of the site looking to improve site's fitness. The remainder of the bees are sent out scouting for new sites, in other words they are reinitialized to a random position. This process repeats until some site reaches a satisfactory level of fitness – or the optimal solution is reached, if this is known in advance.

More formally the algorithm operates as follows:

Algorithm 5: Bees Algorithm

$B = \{b_1, b_2, ..., b_n\}$ setToRandomPosition(B)
while *termination condition not met* **do**
  sortByFitness(B)
  $E = \{b_1, b_2, ..., b_e\}$
  $R = \{b_{e+1}, b_{e+2}, ..., b_m\}$
  searchNeighbourhood($E \cup \{c_1, ..., c_{nep}\}$)
  searchNeighbourhood($R \cup \{d_1, ..., d_{nsp}\}$)
  setToRandomPosition($B - (E \cup R)$)
end

B is the set of bees that are used to search the search space. Initially the bees are set to random positions. Function $sortByFitness$ sorts the bees in order of maximum fitness. It then proceeds by taking the m most promising sites found by the bees. It does this by partitioning these into two sets, $E, N \subset B$. E is the first e best sites, and represents the so called *elite* bees. N is the $m - e$ next most promising sites. The $searchNeighbourhood$ function explores the neighbourhood around a provided set of bees. Each sites in E and N is explored. nep bees are recruited for the search of each $b \in E$, and nsp are recruited for the search of each $b \in N$. In practice this means that nep and nsp moves are explored, respectively, within the neighbourhoods of each site. These moves are typically made stochastically, but it is imaginable that a deterministic approach could be used too. The remaining $n - m$ bees (in other words, those not in E and N) are set to random positions. This is repeated until the termination conditions is met, which may be a running time threshold or a predetermined fitness level.

The promised advantage of the Bees Algorithm over other meta-heuristics is its ability to escape local minima and its ability to navigate search topologies with rough terrain (such as in figure 2.14). It achieves this by scouting the search space for the most promising sites, and then by committing more resource to those sites that are producing results.

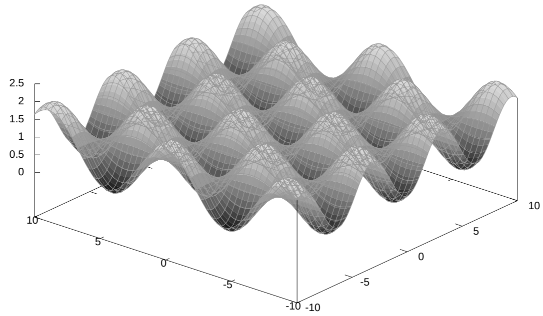

Figure 2.14: *Shown is a search space with many valleys and hills. These search spaces provide a challenge to meta-heuristic approaches as there are many local minima and maxima to get caught in. The Bees Algorithm ameliorates this by searching in many different areas simultaneously*

The Bees Algorithm has been applied to manufacturing cell formation, training neural networks for pattern recognition, scheduling jobs for a production machine, data clustering, and many others. See [**?**] for more examples and a comprehensive bibliography. However, as best as we know the Bees Algorithm hasn't been used on the Vehicle Routing Problem until now.

Chapter 3

Problem Definition

In this chapter we provide a formal definition of the VRP and briefly describe it's variants. The Capacitated Vehicle Routing Problem (VRP) is the more correct name for the VRPthat distinguished it from its variants. We start in section 3.1 by providing a formal definition of the VRP. We formulate it as an integer linear programming problem, as has become standard in the VRP literature. We follow this in section 3.2 by an overview of the VRP variants that are commonly used.

3.1 Capacitated Vehicle Routing Problem (VRP)

We formulate the VRP here as an integer linear programming problem. Although it is possible to solve the VRP using an integer programming solver, this is uncommon in practice as the best solvers are still only able to solve for small problem sizes. We provide this formulation as it has become the lingua franca of combinatorial problems.

We start the formation by specifying the variables used within it. We represent the VRP on a weighted graph, $G = (V, E)$. The vertices of the graph V represent all locations that can be visited, this includes each customers location and the location of the depot. For convenience we let v^d denote the vertex that represents the depot, and we denote the set of customers as $C = 1, 2, ..., n$. Thus the set of vertices is given by $V = v^d \cup C$. Now let the set of edges E corresponds to the valid connections between customers and connections to the depot. For the VRP all connections are possible, in other words we set G to be a clique. Each edge $(i, j) \in E$ has a corresponding cost c_{ij}. We let the cost be the euclidian distance between the two locations $c_{ij} = \sqrt{(x_j - x_j)^2 + (y_j - y_i)^2}$. Where $x_i, y_i, i \in V$ represents the coordinates of the customer's location.

We use R to denote the set of routes that comprise the solution. And we define q and t to be the a maximum capacity and the maximum duration, respectively, both of which can't be exceeded by a route. The demand (i.e. required capacity) for each customer is denoted by $d_i, i \in C$. Likewise we denote the service time required by each customer to be $t_i, i \in C$.

We then use the decision variable X_{ij}^r to denote if a particular edge $(i, j) \in E$ and $r \in R$ contains a trip between customers $i, j \in C$. We let $X_{ij}^r = 1$ where this is true, and $X_{ij}^r = 0$ where not.

We are now able to specify the problem as follows:

Minimize:

$$\sum_{r \in R} \sum_{(ij) \in E} c_{ij} X_{ij}^r \tag{1}$$

Subject to:

$$\sum_{r \in R} \sum_{j \in V} X_{ij}^r = 1 \qquad \forall i \in C \tag{2}$$

$$\sum_{i \in C} d_i \sum_{j \in C} X_{ij}^r \leq q \qquad \forall r \in R \tag{3}$$

$$\sum_{i \in C} t_i \sum_{j \in C} X_{ij}^r + \sum_{(ij) \in E} c_{ij} X_{ij}^r \leq t \qquad \forall r \in R \tag{4}$$

$$\sum_{j \in V} X_{v^0}^r = 1 \qquad \forall r \in R \tag{5}$$

$$\sum_{j \in V} X_{0^d}^r = 1 \qquad \forall r \in R \tag{6}$$

$$\sum_{i \in V} X_{ik}^r - \sum_{j \in V} X_{kj}^r = 0 \qquad \forall k \in C \text{ and } \forall r \in R \tag{7}$$

The objective function (1) minimizes the costs c_{ij}. Constraint (2) ensures that each customer can only be serviced by a single route. Constraint (3) enforces the capacity constraint; each route cannot exceed the maximum vehicle capacity q. Likewise (4) enforces the route's duration constraint. A route's duration is the sum of it's service times $t_i, i \in R$ and its travel time. By convention the travel time is taken to be equal to its total distance, which in turn is equal to the cost of the edges it traverses. Constraint (5) and (6) ensure that each route starts at the depot and finishes at the depot, and does this exactly once each. Lastly, constraint (7) is a flow constraint that ensures that the number of vehicles entering a customer is equal to the number of vehicles leaving.

Constraint 4, which enforces a maximum route duration, t, is often left out of the traditional VRP formation but is included here as it is present in the problem instances we use as benchmarks in chapter 5.

3.2 Variants

In this section we provide an overview of the common variations of the VRP that are used. These variations have arisen from real-world vehicle routing scenarios, where the constraints of are often more involved than is modelled in the VRP.

3.2.1 MDVRP

A simple extension to the VRP is to allow each vehicle to start from different depots. Part of the problem then becomes which customers are assigned to which depots, which in itself becomes a hard combinatorial problem. The VRP formation can easily be relaxed to allow this. There are two variations of the problem. One constrains each route to finish at the same depot that it started from. The other allows vehicles to start and finish at any depot, as long as the same number of vehicles return to the depot as left it.

3.2.2 Vehicle Routing with Time Windows (VRPTW)

The Vehicle Routing Problem with Time Windows (VRPTW) adds the additional constraint to the problem that each customer must be visited within a time window specified by the customer. More formally, for VRPTW each customer $i \in C$ also has a corresponding time window $[a_i, b_i]$ in which the goods must be delivered. The vehicle is permitted to arrive before the start time, a_i; however, in this case the vehicle must wait until a_i adding to the time it takes to complete the route. However it is not permitted for the job to start after time b_i.

An additional constraint is added to the formation of VRP to ensure that the time window constraint is meet: $a_i \leq S_i^r \leq b_i$ where the decision variable S_i^r provides the time that route $r \in R$ arrives at customer $i \in V$.

3.2.3 PDP

The Pickup and Delivery problem generalizes the VRP. In this problem goods are both picked up and delivered by the vehicle along its route. The vehicle's work now comes in two flavours: pickup jobs, $P = \{p_1, p_2, ..., p_k\}$, and delivery jobs, $D = \{d_1, d_2, ..., d_l\}$, such that $C = P \cup D$. Additional constraints are added to the VRP formation to ensure that

1. Pickup and deliver jobs are completed on the same route, that is $p_i \in r \Rightarrow d_i \in r$ for $r \in R$.
2. The pickup job, P_i, appears before its corresponding delivery job, D_i, in a route $r \in R$.

3. The vehicles capacity isn't exceeded as the goods are loaded and unloaded. This requires the use of an intermediate variable, $y_i, i \in V$ that represents the load of the route at customer i. It then applies constraints: $y_0 = 0$, $X^r_{ij} = 1 \Rightarrow y_i + d_i = y_j$, and $\sum_{r\in R}\sum_{j\in V} X^r_{ij} y_i \leq q$ to enforce this.

There is also a variation on PDP that adds time windows, called PDPTW. In this case the extra constraints from the VRPTW problem are merged with those above. PDP is a much harder problem computationally than VRP, as its extra constraints add new dimensions to the problem. Because of this PDP has only been actively researched in the last decade.

Chapter 4

Algorithm

This chapter describes the algorithm used within this thesis and provides a detailed description of its operation. We start by reviewing the goals that the algorithm set out to achieve in 4.1. We then provide in section 4.2 a description of how the algorithm internally represents the VRP problem and its candidate solutions. Next in section 4.3 we provide a detailed description of the operation of the algorithm. Finally in section 4.4 we describe the neighbourhood structures that are used by the algorithm to define its search space.

4.1 Goals

The algorithm presented in this thesis was built for use in a commercial setting. It was developed as part of a NZTE grant for the company *vWorkApp Inc.*'s scheduling and dispatch software. Accordingly more emphasis is given to runtime performance relative to optimisation performance, than is typically afforded in the VRP literature.

The algorithm aims to achieves the following goals (in order of priority):

1. All constraints meet. Specifically the maximum shift time for a route must be observed.
2. Robust and repeatable results. Many of the algorithms in the literature suffer from poor high variance of results between runs. Often the only the best solution from ten or more runs is reported.
3. Speed of execution. It was more important that the algorithm produces a result within 5% of optimal quickly, than it achieves 99% of optimal over a longer period of time. The algorithm has also been designed to allow for it to be easily parallelized, to take advantage of modern hardware.
4. Travel distance is minimized.

4.2 Problem representation

Our algorithm represents the problem in a direct and straight forward manner. It proceeds by directly manipulating the candidate solution $\mathfrak{S}$, where $\mathfrak{S}$ is a set of routes $R \in \mathfrak{S}$, and each route contains an ordered sequence of customers $v_i \in R$ starting and ending at the depot vertex v^d.

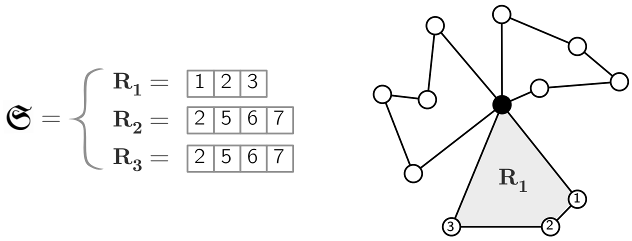

Figure 4.1: *Shown is an example of a simple VRP candidate solution as represented internally by the algorithm.*

More generic representations are sometimes used for meta-heuristics, as is commonly seen with Genetic Algorithms, as they allow the algorithm to be easily adapted to other combinatorial problems. This often comes at a cost of added complexity and often inferrer results[1]. This algorithm was designed specifically for solving instances of the VRP so the direct representation was chosen.

The algorithm makes use of a *fitness* concept, common to many meta-heuristics, to describe the cost of the solution. Our fitness function $f()$ includes terms for the distance (i.e. cost) of the solution and penalties for breaking the capacity and maximum route time constraints. Specifically $f()$ is defined as follows:

$$c(R) = \sum_{i \in R} c_{i,i+1} \tag{4.1}$$

$$d(R) = max(\sum_{i \in R} d_i - q, 0) \tag{4.2}$$

$$t(R) = max(\sum_{i \in R} t_i + c(R) - t, 0) \tag{4.3}$$

$$f(\mathfrak{S}) = \sum_{R \in \mathfrak{S}} (\alpha c(R) + \beta d(R) + \gamma c(R)) \tag{4.4}$$

Function $c(R)$ calculates the cost (i.e distance) of a given route, function $d(R)$ calculates how overcapacity that the given route is. We define overcapacity to be how much larger

[1]This occurs because the operators that act on the problem representation can no longer exploit information that is specific to the problem domain and must rely on general purpose operations instead

the route's summed demands $d_i, i \in R$ are than the stated maximum allowable capacity q. Likewise function $t(R)$ calculates the overtime of the given route. A route's duration is calculated as being the sum of its customer's service times $t_i, i \in R$ and its travel time. By convention the travel time is equal to the distance of the route. Function $t(R)$ then returns how much over the maximum allowable route duration t the duration is. Lastly the fitness function $f()$ is the weighted sum of these three terms. Parameters α, β, and γ are used used to control how influence each term has on determining the candidate solutions fitness.

For the purposes of benchmarking our algorithm (see chapter 5) we use a travel cost that is equal to the 2D Euclidian distance [2] between the two points. For real life problem instance we have found that using a manhattan distance[3] often provides superior results. This is presumably due to the manhattan distance better modeling the road system within Auckland, which although not a strict grid still doesn't allow line-of-site travel.

4.3 Enhanced Bees Algorithm

Our Algorithm is based on the *Bees Algorithm* (see section 2.5.2 for an overview of this algorithm). The algorithm makes some changes to adapt the Bees Algorithm to this domain. An interesting aspect of the Bees Algorithm is that it covers a broad search area, minimizing the risk of being stuck in a local optimum. It achieves this by randomly probing (or in the Bees Algorithm parlance, Scouting) many areas of the search space through its entire run. However, this approach isn't well suited to hard combinatorial problems, where a newly constructed solution, let alone a randomly constructed one, is often far from optimal (for instance, the Clark Wright Savings heuristic still produces solutions that are upto 15% from the best known and will require many operations to get close to optimal). We've adapted the Bees Algorithm such that many of its unique characteristics, such as its relative robustness, are maintained while working well with hard combinatorial problem, such as the VRP.

Our algorithm can be summarized at a high level as follows:

Algorithm 6: Enhanced Bees Algorithm

```
S = seedSites()
while termination condition not met do
    for s_i ∈ S do
        explore(s_i, d)
        if i < λ then
            removeWorstSite
        end
    end
end
```

The algorithm maintains a collection of sites S, and each site $s_i \in S$ maintains a collection

[2] In this case we define the cost function as $c_{ij} = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$

[3] Conversely we use define the cost function as $c_{ij} = (x_j - x_i)^2 + (y_j - y_i)^2$

of bees B_i. Each bee is a proxy to the problem domain that we're trying to solve. In our case this is the VRP problem representation covered above.

Initially each site is seeded, such that each site $s_i \in S$ contains a collection of bees B_i, and each bee has a corresponding VRP candidate solution $\mathfrak{S}$. Each candidate solution is initialized by seeding each route with a randomly chosen customer, and is then filled out using the insertion heuristic outlined in 4.4.2. Each site is then, in turn, improved upon. This is achieved by iteratively exploring the neighbourhood of each site. The process used to explore each site is where the majority of the algorithm's processing takes place, and where the interesting aspects of the algorithm are developed. The procedure is covered in detail in sections 4.3.1, 4.3.2, and 4.4.

The number of sites explored is reduced over the run of the algorithm. This borrows from the idea of a cooling schedule used by Simulated Annealing. Sites are reduced using the formula:

$$S = S - s_w \qquad \text{if } i \bmod \lambda = 0 \tag{4.5}$$

Where s_w represents the worst site, in terms of fitness, i represents the current iteration of the algorithm, and λ represents the period of iterations with which the number of sites are reduced. Once the algorithm is complete the $\mathfrak{S}$ with the best overall fitness is returned as the answer.

In the next section we review in more detail each aspect of the algorithm.

4.3.1 Bee movements

Bees are moved around the search space to look for improvements to the collection of candidate solutions being maintained. Each bee represents a current solution $\mathfrak{S}$, so a valid bee move is any new candidate solution $\mathfrak{S}'$ that can be reached within the neighbourhood of $\mathfrak{S}$ (see 4.4 for the operations under which the neighbourhood is defined).

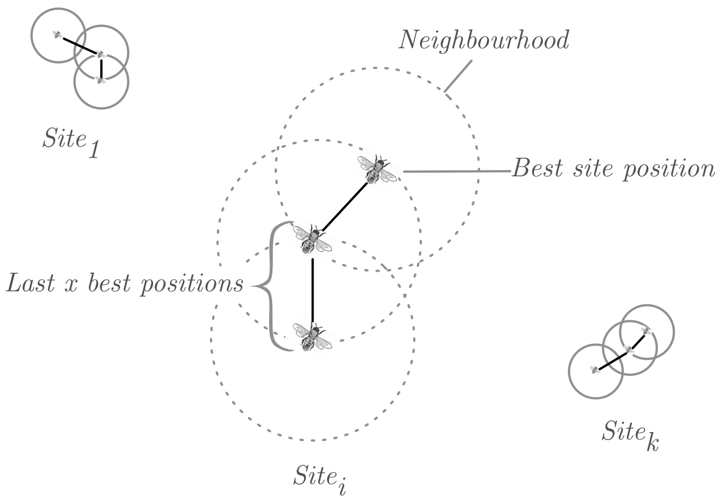

Figure 4.2: *TODO.*

A feature of our algorithm is that two Bees can't occupy the same position. The algorithm maintains a register of the current positions occupied by each bee. We use the current fitness, $f(\mathfrak{S})$, as a quick and simple representation of a bees current position[4]. If a bee tries to occupy the same position as another bee (i.e. they share the same candidate solution) then the bee trying to occupy that position is forced to explore the neighbourhood again and find another position.

Enforcing the constraint that each bee must occupy a unique position has two benefits. It forces diversification between the bees and sites hence encouraging a greater degree of the search space to be explored. It also has the benefit of increasing the chance of a local optimum being escaped, as a bee ensnared in the local optimum position now forces the remainder of the hive to explore alternative positions. This feature has a similar intent and purpose to the tabu lists used by Tabu Search.

Another feature of our algorithm is the role that sites play in concentrating exploration on certain areas of the search space. Each site maintains a list, $M = [\mathfrak{S}_1, ..., \mathfrak{S}_\epsilon]$, of the last ϵ best positions. Each $\mathfrak{S}_i \in M$ is then taken as a launching point for a site's bees to explore. θ bees are recruited for the exploration of each $\mathfrak{S}_i \in M$. Once all positions in M have been explored then the best ϵ positions are again taken and used as the launching points for the site's next round of exploration. This exploration method has two advantages. Firstly, it allows for a simple type of branching, as ϵ of the most promising positions there were traversed through on the way to the current position are also explored. Secondly, it prevents cycling between promising solutions that are in close vicinity to each other.

[4]This obviously won't work in circumstances where there is a reasonable likelihood of two candidate solutions, $\mathfrak{S}_i$ and $\mathfrak{S}_j$ having $f(\mathfrak{S}_i) = f(\mathfrak{S}_j)$. This isn't the case with the problem instances we've used in this thesis. However, this will need to be modified if the algorithm is to be used on more general problem instances

Conversely, sites don't interact with each other. Each maintains its own unique list of ϵ promising positions. Our constraint that no two bees can occupy the same position ensures that each sites in turn covers a area of the search space. In practice we've found that this is sufficient to encourage sites to diverge and explore separate areas of the search space.

4.3.2 Search space coverage

As mentioned above one of the unique aspects of the Bees Algorithm is its ability to produce robust results through probing a large area of the search space. However this doesn't work well with hard combinatorial problems, where it can't ascertained quickly if an area in the search space shows promise or not.

We use an alternative approach inspired by Simulated Annealing's use of a cooling schedule to overcome this limitation. Bees are initially divided equally between each site $s_i \in S$, ensuring that each site is explored equally. Then every λ period of iterations we reduce the number of sites maintained, such that $S = S - s_w$, where s_w is the site with lowest fitness. We measure each sites fitness simply from the fitness of it's best position found to date.

This process continues until a single site remains. We show experimentally in chapter 5 that this process improves the robustness of the algorithm and produces better results overall than if it were run using only a single site.

4.4 Search Neighbourhood

As discussed above each bee seeks to improve upon its current fitness by exploring the local neighbourhood of the solution it represents. In our algorithm it does this by applying a Large Neighbourhood Search (LNS) operator to its candidate solution $\mathfrak{S}$. The LNS operator differs from the more common VRP operators in that a single operation applies many changes to the candidate solution $\mathfrak{S}$. This widens the neighbourhood of $\mathfrak{S}$ to encompass exponentially many candidate solutions. LNS navigates through the vast space it opens by selecting only those changes that have a high likelihood of improving the solution.

The LNS operation is comprised of two-phases: a destroy phase, and a repair phase. In a VRP algorithm the destroy phase is typically used to remove customers from a solution's route. In our algorithm we undertake this using two destroy heuristics. A some-what intelligent heuristic that attempts to remove those customers that are more likely to be able to be recombined in a profitable way. And a simple random selection. These heuristics are covered more formally in section 4.4.1 below. The second phase is used to repair the partial solution. Our algorithm uses a simple insertion heuristic that inserts the customers into those locations that that have the lowest insertion cost. This heuristic is covered more formally in section 4.4.2 below.

4.4.1 Destroy heuristic

We employ two destroy heuristics. The first simply selects l customers randomly from a solution $\mathfrak{S}$ and removes these from their routes. The second is slightly more complicated and is due to Shaw [35]. Shaw's removal heuristic stochastically selects customers such that there is a higher likelihood of customers that are related to one another being removed. For our purposes related is defined as meaning that for any two customers $v_i, v_j \in V$ then either v_i, v_j are geographically close to one another, that is c_{ij} is small, or they share an adjacent position within the same route, $R = [..., v_i, v_j, ...]$.

The rational to removing related customers is that these customers are the most likely to profitable exchange positions with one another. Conversely unrelated customers are more likely to be reinserted back to the same positions they were removed from.

4.4.2 Repair heuristic

The repair heuristic that we use randomly selects one of the removed customers v_j, and calculates a cost for reinserting v_j between each pair of jobs $v_i, v_k \in R_i$ for all $R_i \in \mathfrak{S}$ (actually not all reinsertion positions are considered, see the next section, 4.4.3, for a description of which positions are considered). The reinsertion cost is calculated as follows:

$$c^* = (c_{ij} + c_{jk} - c_{ik})cost = c^* + (d(R') - d(R)) + (t(R') - t(R)) \tag{4.6}$$

Where c^* calculates the cost difference in terms of travel distance. R and R' are defined as the route before and after the customer is inserted, respectively. And functions $d(R)$ and $t(R)$ are all defined as they are in section 4.2. The final cost is the sum of the added travel distance and the two extra penalties, if the the route is now over capacity or time. The algorithm selects the position with the lowest insertion cost to reinsert the customer. This is repeated until all customers are reinserted into the solution.

The reason that the order in which customers are reinserted is done randomly is that it adds a beneficial amount of noise to the heuristic. This ensures a healthy diversity of solutions are generated from the heuristic.

4.4.3 Neighbourhood extent

We also use two techniques that adjust the extent of the neighbourhood being searched.

The first technique we use is to allow some flexibility in selecting infeasible solutions are intermediate candidate solutions. As can be seen from our formulation of the candidate solutions' fitness values (see section 4.2) violations the problem's capacity and duration constraints are penalized rather than forbidden. This allows for bees to navigate through infeasible solutions, where other aspects of that solution are sufficiently attractive enough to outweigh the penalties. However, only feasible solutions are allowed to be counted as the best overall solutions found by the algorithm. In chapter 5 we show experimentally how much impact different values of the penalty constraints, α, β, and γ, have on the performance of the algorithm.

Another technique that we use is to adjust the number of insertion positions considered as part of the repair heuristic. The number of insertion positions considered starts with both sides of the three closest customers and increases as the site ages. More formally, let $v_i \in V$ be the customer that is being inserted. An ordered sequence of candidate insertion points, $L_{v_i} = [v_1, ..., v_n]$ such that $v_j \in V - v_i$, is kept that lists customers in increasing geographic distance from v_i, that is $c_{ij} \forall j \in V - i$. The LNS repair operator tests μ positions from L_{v_i} to find the cheapest insertion point. The repair operator tests both possible insertion points represented by $v_j \in L_{v_i}$, that is, it tests both the insertion cost of inserting v_i immediately before and after v_j in the route R that contains v_j.

For each site $s_i \in S$ we also maintain an counter $a_i | i \in S$ that denotes the age of the site. A site's age is incremented for each iteration that a site doesn't improve upon it currently best known solution (as defined by the solutions fitness, $f(\mathfrak{S})$). Whenever a site improves upon its best known solution then the counter is reset such that $a_i = 0$.

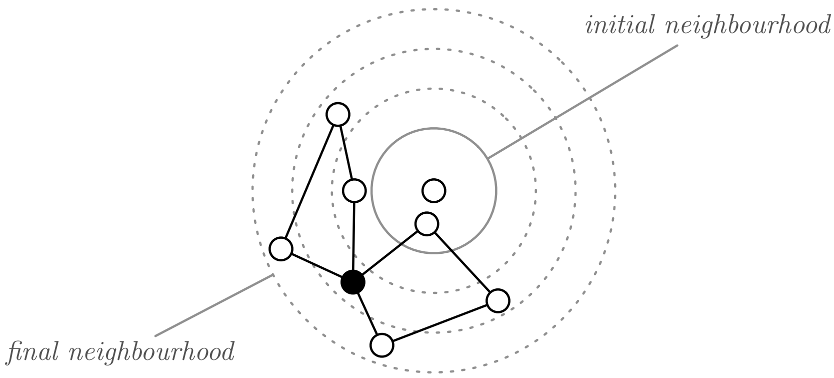

Figure 4.3: *TODO.*

We then use the following formula to increase how much of L_{v_i} is considered as the site ages.

$$\mu = |L_{v_i}| * min(\frac{a_i}{k}, 1)$$

Where k is a constant that controls the rate at which the search area is expanded.

As this process extends the number of insertion positions that are considered by the repair heuristic, this also serves to extend the neighbourhood of solutions surrounding a candidate solution $\mathfrak{S}$. In this way we dynamically extend the size of the neighbourhoods searched by a site $s_i \in S$ if site s_i has become stuck in a local optima.

Chapter 5

Results

In this chapter we provide a detailed breakdown of the results obtained by our algorithm. The algorithm is tested against the well known set of test instances due to Christofides, Mingozzi and Toth [6]. We start in section 5.1 by presenting the results from running the algorithm in its two standard configurations: the first configuration is optimised to produce the best overall results, regardless of its runtime performance; the second configuration is optimised to produce the best results possible within a 60 second time threshold. We follow this in section 5.2 by producing results, on the same problem instances, for a standard Bees Algorithm and a straight LNS local search. Here we aim to demonstrate that the enhancements suggested in this thesis do in fact improve the solution quality. Finally we end in section 5.3 by comparing and ranking how our algorithm performs compared to results in the literature.

5.1 Enhanced Bees Algorithm results

The results provided here show how the algorithm performs using its two standard configurations. The first standard configuration is optimised to produce the best overall results, that is, the minimum travel distance that meets all capacity and service time constraints. No consideration is made for its runtime performance. This configuration is left to run for 30 minutes before being terminated. This configuration is denoted as *Best* in the diagrams and tables below.

Conversely the second configuration is optimised to produce the best results possible within a 60 second runtime threshold. Again, the algorithm aims to minimize the travel distance, while meeting all capacity and service time constraints.

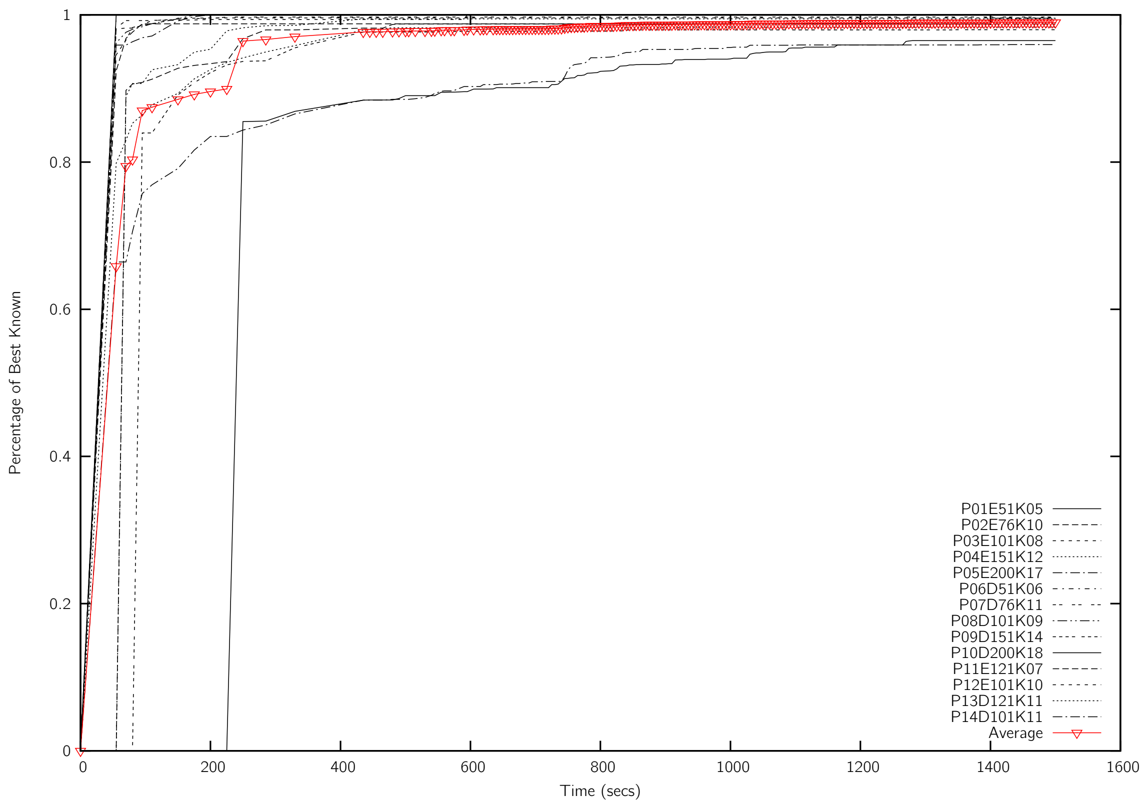

Figure 5.1: Best. *Shown are the results obtained on the standard 14 Christofides, Mingozzi and Toth instances when the algorithm is optimised for producing the best overall results. These instances where left to run for 30 minutes on a MacBook Pro 2.8 GHz Intel Core 2 Duo. The lefthand axis shows the relative percentage compared to the best known result for the same problem instance. The bottom axis shows the elapsed runtime in seconds. Infeasible solutions are shown as being 0% of the best known result. The average result obtained across all problem instances is shown in red*

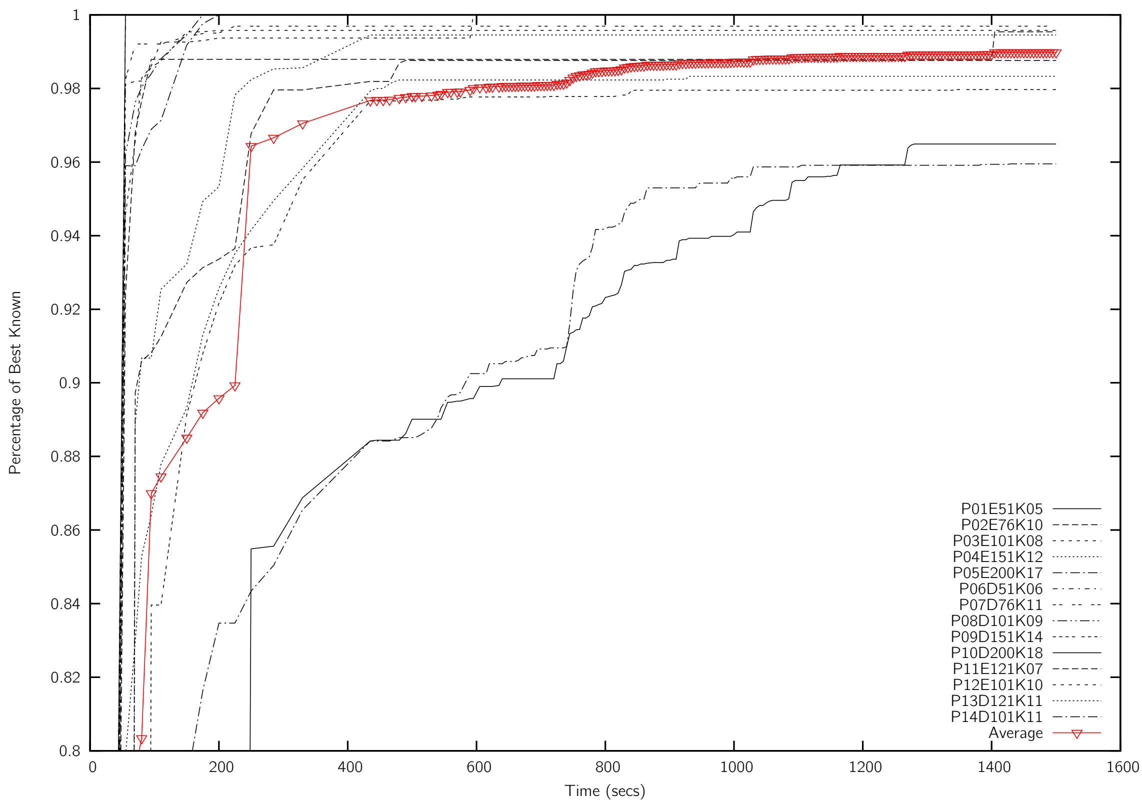

Figure 5.2: Best - Blowup. *The same graph as depicted in 5.1 is shown, but with the section between 0.8 and 1.0 of the left axis blown up.*

The results depicted in 5.1 and **??** were obtained on a MacBook Pro 2.8 GHz Intel Core 2 Duo, using the following parameters: The algorithm was set to start with $|S| = 100$ (i.e. 100 sites), and reduce this number each 50 iterations ($\lambda = 50$) by 1% until only $|S| = 3$. The number of promising solutions remembered by each site was set to $|M| = 5$. The algorithm was left to run for 30 minutes on each problem before being terminated. Infeasible solutions (i.e. solutions over their capacity or service time constraints) were allowed to be traversed through, but were scored as being 0% of the best known solution.

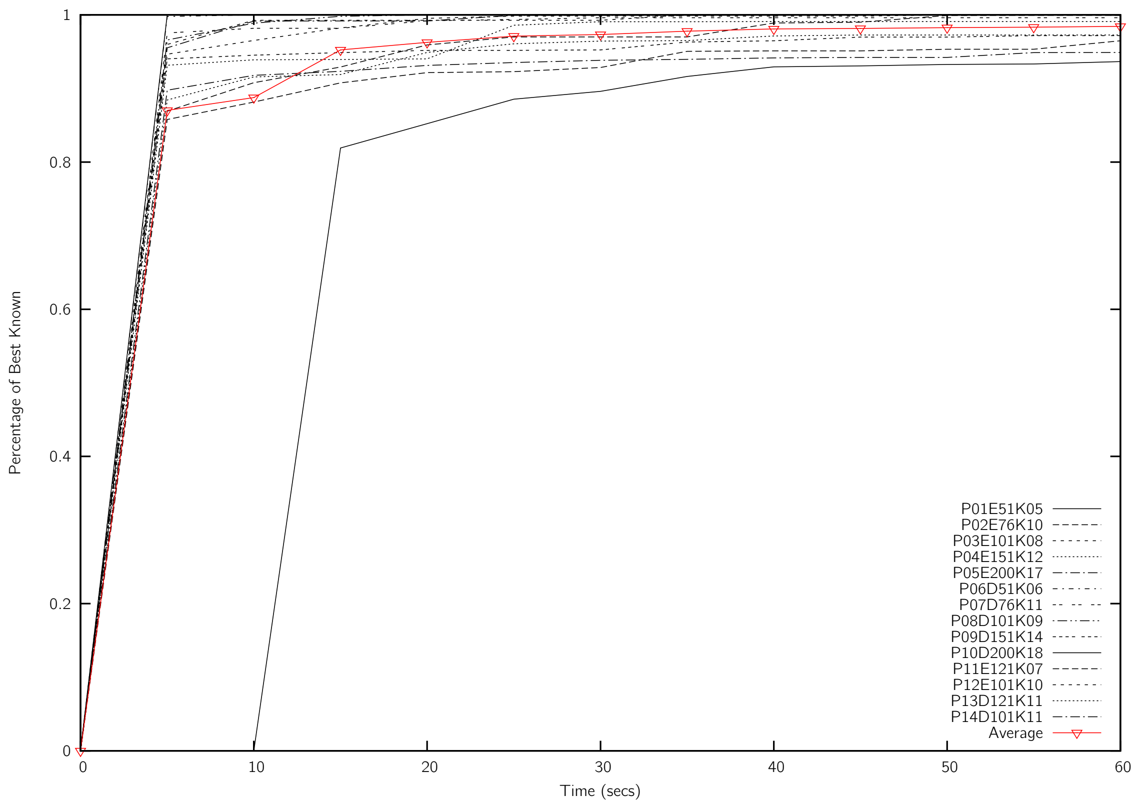

Figure 5.3: Fast. *Shown are the results obtained on the standard 14 Christofides, Mingozzi and Toth instances when the algorithm is optimised for producing the best results within a 60 second runtime limit. These instances where obtained on a MacBook Pro 2.8 GHz Intel Core 2 Duo. The lefthand axis shows the relative percentage compared to the best known result for the same problem instance. The bottom axis shows the elapsed runtime in seconds. Infeasible solutions are shown as being 0% of the best known result.*

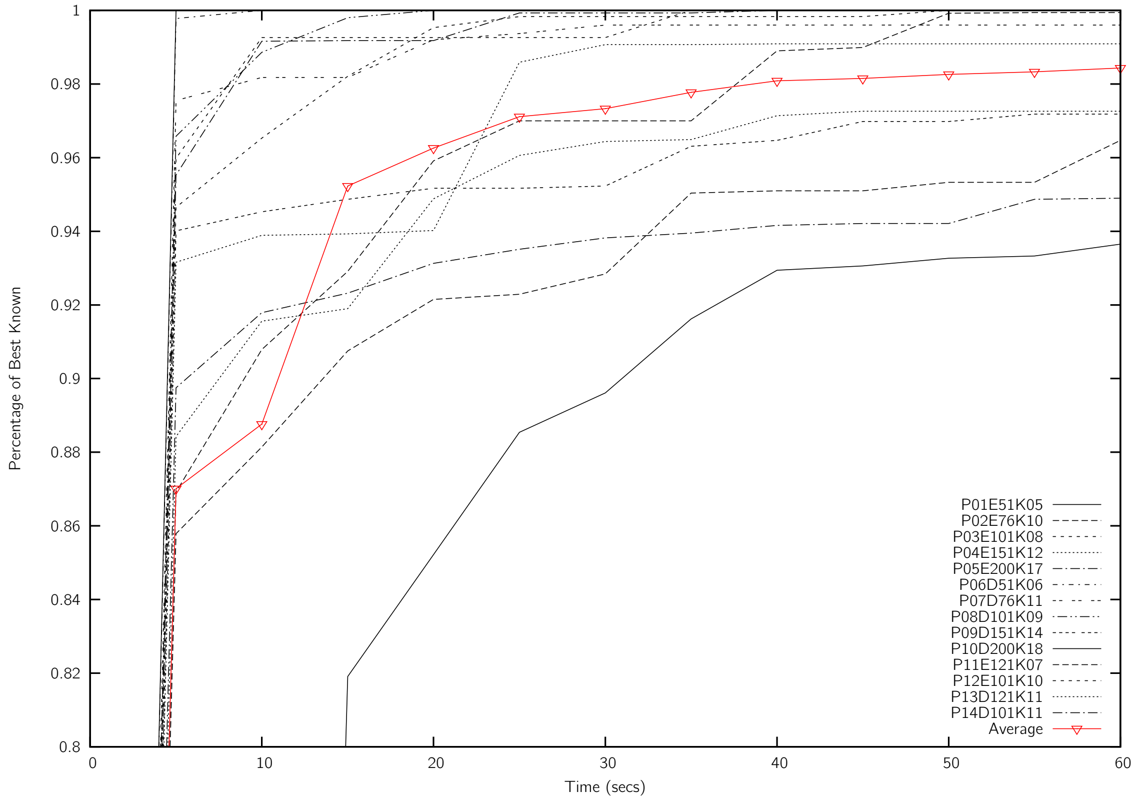

Figure 5.4: Fast - Blowup. *The same graph as depicted in 5.3 is shown, but with the section between 0.8 and 1.0 of the left axis blown up.*

The results depicted in 5.3 and 5.4 were obtained on a MacBook Pro 2.8 GHz Intel Core 2 Duo, using the following parameters: The algorithm was set to start with $|S| = 25$ (i.e. 25 sites), and reduce this number each iteration ($\lambda = 1$) by 1% until only $|S| = 1$. The number of promising solutions remembered by each site was set to $|M| = 5$. The algorithm was left to run for 60 seconds on each problem before being terminated. The LNS improvement heuristic was set to destroy between 0% and 80% (with a mean of 40%) of the solution at each step. The repair operator initially only considers the first 3 closest customers as reinsertion points, but it increases this to be 50% of the closest customers as the site ages. Infeasible solutions (i.e. solutions over their capacity or service time constraints) were allowed to be traversed through, but were scored as being 0% of the best known solution.

Table 5.1 summarizes these results.

5.2 Experiments

In this section we review the results obtained by implementing a standard Bees Algorithm and a LNS local search. The aim of these experiments is to prove that the algorithmic

Instance	Results				Best Known[3]
	Fast[1]		Best[2]		
P01E51K05	524.61	*100.00%*	524.61	*100.00%*	524.61
P02E76K10	835.77	*99.94%*	835.77	*99.94%*	835.26
P03E101K08	826.14	*100.00%*	826.14	*100.00%*	826.14
P04E151K12	1057.40	*97.26%*	1045.88	*98.33%*	1028.42
P05E200K17	1360.85	*94.90%*	1345.94	*95.95%*	1291.45
P06D51K06	555.43	*100.00%*	555.43	*100.00%*	555.43
P07D76K11	913.37	*99.60%*	912.47	*99.69%*	909.68
P08D101K09	865.94	*100.00%*	865.94	*100.00%*	865.94
P09D151K14	1196.32	*97.18%*	1186.65	*97.97%*	1162.55
P10D200K18	1490.49	*93.65%*	1446.67	*96.49%*	1395.85
P11E121K07	1080.20	*96.47%*	1055.21	*98.76%*	1042.11
P12E101K10	819.56	*100.00%*	819.55	*100.00%*	819.56
P13D121K11	1555.30	*99.09%*	1549.71	*99.45%*	1541.14
P14D101K11	866.37	*100.00%*	866.37	*100.00%*	866.37
Average		*98.44%*		*99.04%*	

[1] Run for 60 seconds
[2] Run for 30 minutes
[3] As reported by Gendreau, Laporte, and Potvin in [22]

Table 5.1: *Standard Results*

enhancements suggested in this thesis do in fact produce better results than are possible from the standard algorithms. We also demonstrate that the combination of the Bees Algorithm with the LNS local search produces better results than if either algorithm were used solely.

5.2.1 Bees Algorithm vs. Enhanced Bees Algorithm

We start by showing the results obtained by using the standard Bees Algorithm as described by Pham et al. in [30]. The same problem instances as section 5.1 are used; the results are directly comparable between the two.

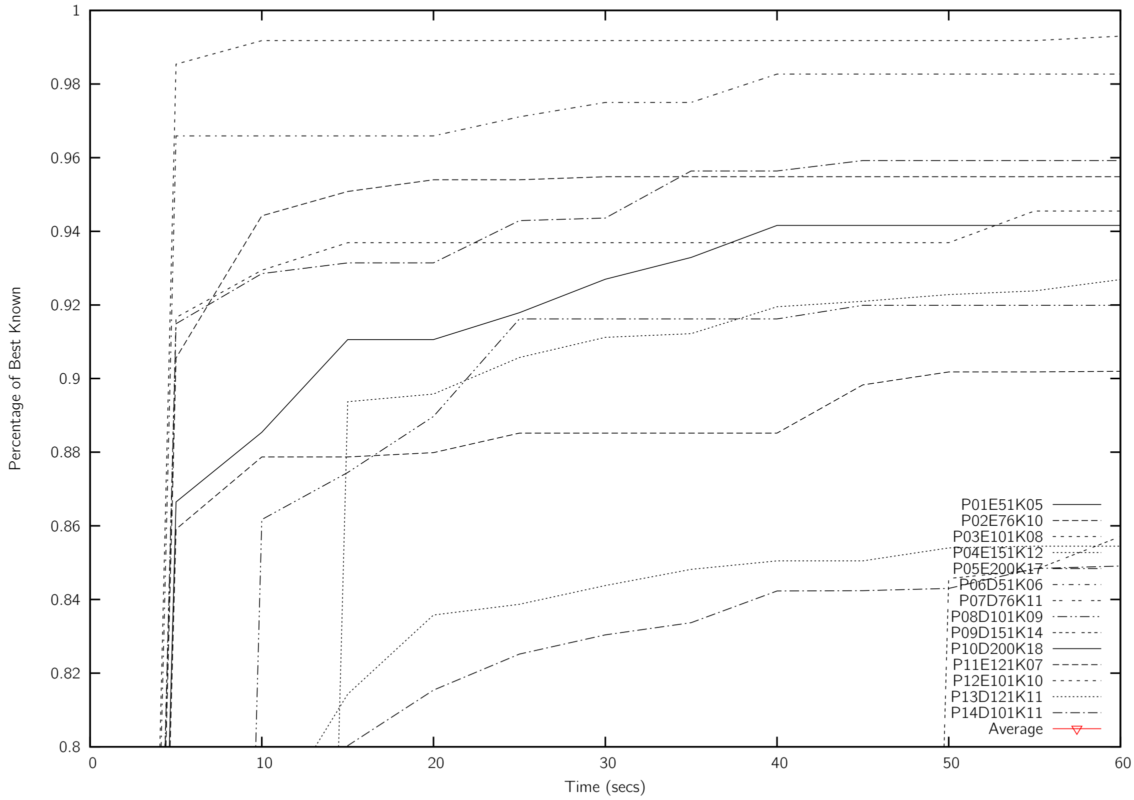

Figure 5.5: *Shown are the results obtained on the standard 14 Christofides, Mingozzi and Toth instances when the algorithm is optimised for producing the best results within a 60 second runtime limit. These instances where obtained on a MacBook Pro 2.8 GHz Intel Core 2 Duo. The lefthand axis shows the relative percentage compared to the best known result for the same problem instance. The bottom axis shows the elapsed runtime in seconds. Infeasible solutions are shown as being 0% of the best known result. The average result obtained across all problem instances is shown in red*

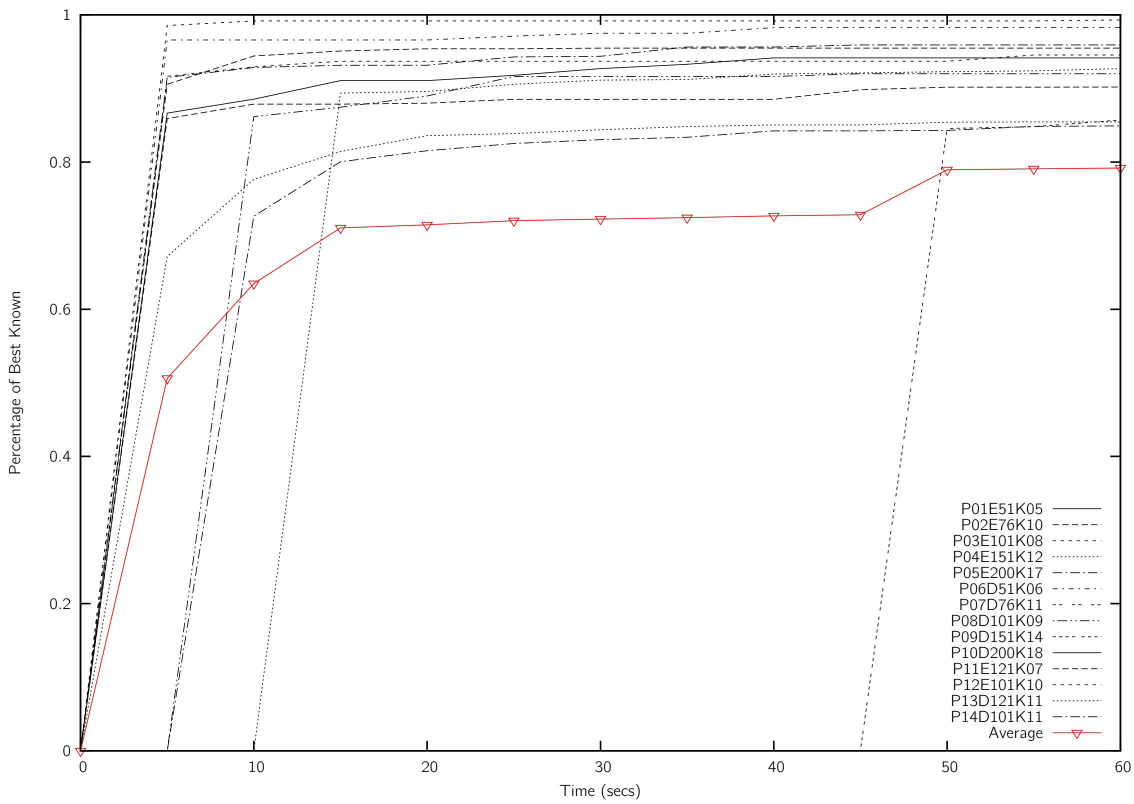

Figure 5.6: *Shown are the same results as depicted in* **??**, *but with the section between 0.8 and 1.0 of the left axis blown up. Note the average is not shown on the blow up because it is below 80% percent.*

The results depicted in **??** and **??** were obtained on a MacBook Pro 2.8 GHz Intel Core 2 Duo, using the following parameters: The algorithm used 25 sites. The best 6 sites were selected as being elite sites. Each elite site had 3 bees recruited for the search. Another 6 sites were selected as being non-elite, and had 2 bees recruited for the search. The bees from the remaining 13 sites were left to search randomly. The algorithm was left to run for 60 seconds on each problem before being terminated. Infeasible solutions (i.e. solutions over their capacity or service time constraints) were allowed to be traversed through, but were scored as being 0% of the best known solution. A $2-$*interchange* improvement heuristic was used for the improvement phase of each bee (see chapter **??** for an overview on how this heuristic works).

5.2.2 Large Neighbourhood Search

Next we show the results obtained by using a stand-alone LNS search embedded within a hill climb meta-heuristic. The LNS search is the one used and embedded within our algorithm. It should be noted that there are more sophisticated LNS algorithms available than the comparatively simple one used here. And these would most probably produce better results than the LNS results presented here. However, we believe one of the attractive features of our algorithm is that it is uses a fairly simple local search method. Moreover, part of what we aim to demonstrate here is that the limitations of our simple local method are offset by its combination with the Bees Algorithm.

The same problem instances as section 5.1 are used; the results are directly comparable between the two.

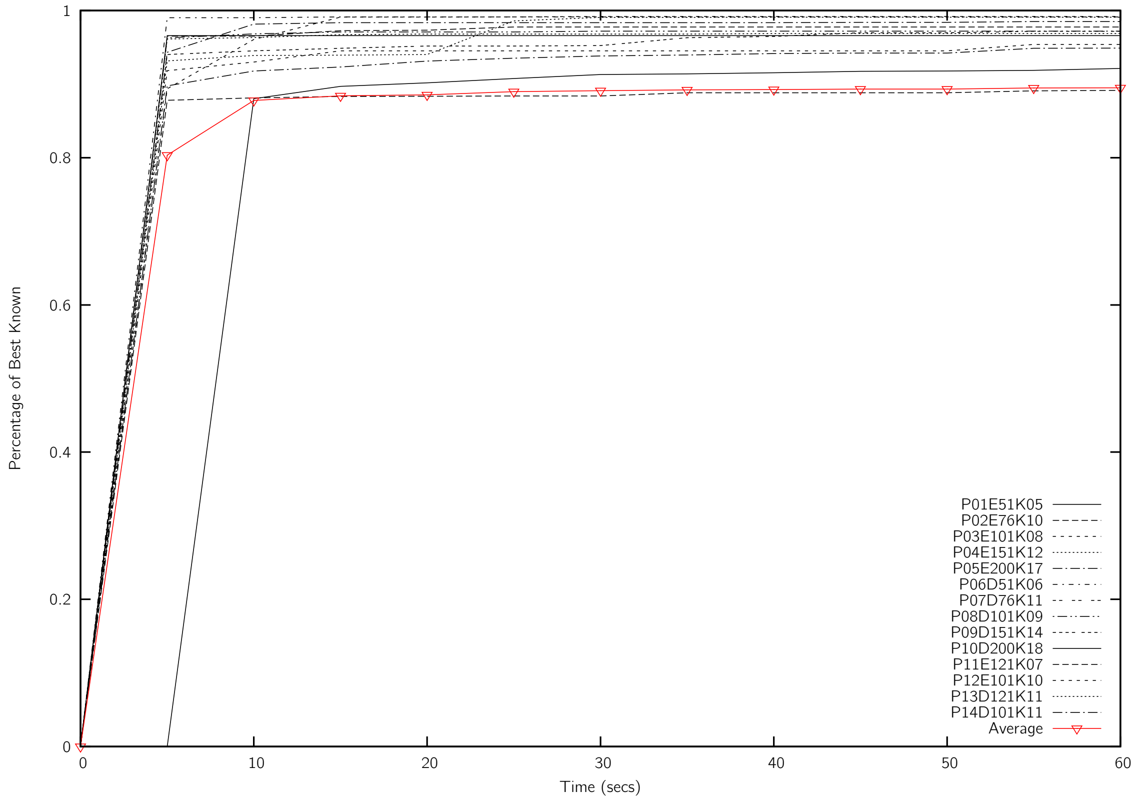

Figure 5.7: *Shown are the results obtained on the standard 14 Christofides, Mingozzi and Toth instances when the algorithm is optimised for producing the best results within a 60 second runtime limit. These instances where obtained on a MacBook Pro 2.8 GHz Intel Core 2 Duo. The lefthand axis shows the relative percentage compared to the best known result for the same problem instance. The bottom axis shows the elapsed runtime in seconds. Infeasible solutions are shown as being 0% of the best known result. The average result obtained across all problem instances is shown in red*

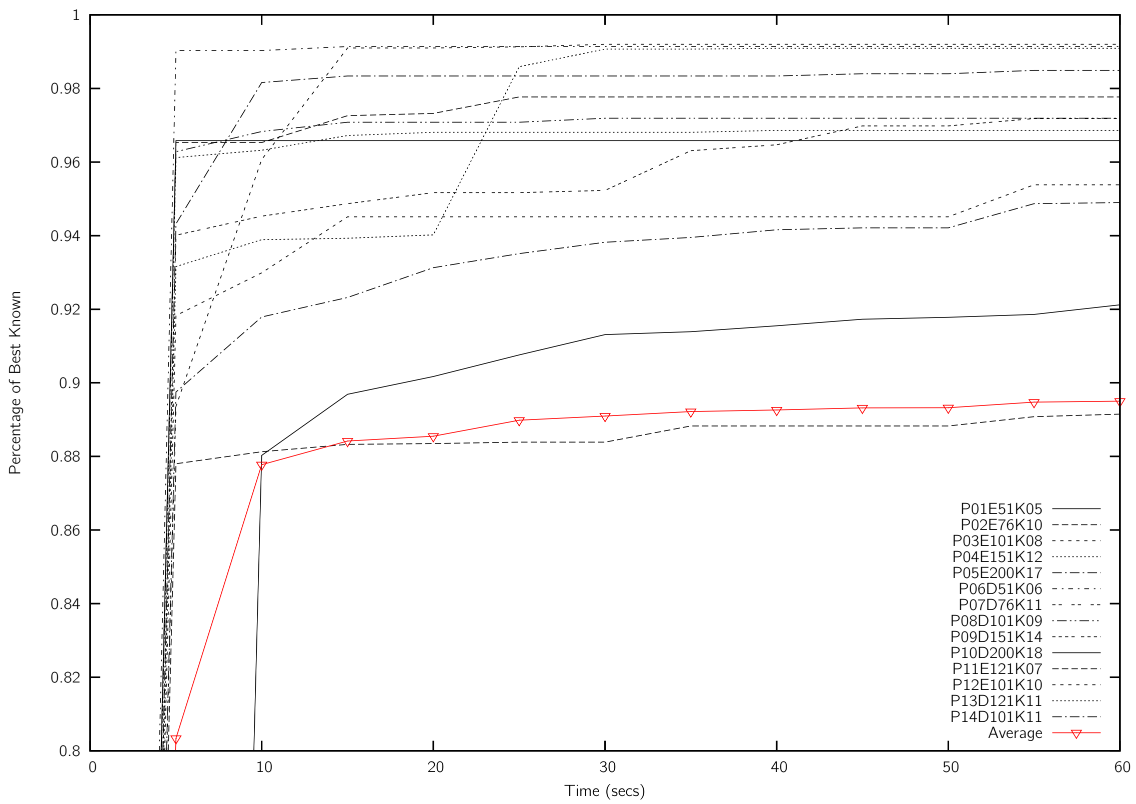

Figure 5.8: *Shown are the same results as depicted in 5.7, but with the section between 0.8 and 1.0 of the left axis blown up.*

The results depicted in 5.7 and 5.8 were obtained on a MacBook Pro 2.8 GHz Intel Core 2 Duo, using the following parameters: The algorithm starts at starting position generated by a simple insertion heuristic. The LNS heuristic was set to destroy between 0% and 80% (with a mean of 40%) of the solution at each step. The repair enumerates all customers when deciding the best reinsertion point. Infeasible solutions (i.e. solutions over their capacity or service time constraints) were allowed to be traversed through, but were scored as being 0% of the best known solution.

5.2.3 Summary

Table **??** summarises the results from the Bees Algorithm and LNS experiments, alongside the results obtained by our Enhanced Bees Algorithm. The Bees Algorithm is the worst of the three. This isn't surprising given that the standard operation of the Bees Algorithm is used to solve continuos problems rather than discrete ones. Where the Bees Algorithm has been used for discrete problems it has been adapted to incorporate more sophisticated local search techniques, much as the Enhanced Bees Algorithm has been. Also of note is that two of the problem instances didn't produce feasible solution at all within the 60 second threshold. We believe that given a longer running time the algorithm would most probably have found a feasible solution. However, one of the goals of this thesis was to produce robust results reliably; in this count the unaltered Bees Algorithm isn't competitive.

The LNS improvement heuristic produced much stronger results. This shows that the LNS improvement heuristic is a large part of the results obtained by our Enhanced Bees Algorithm. The LNS heuristic is a fairly new heuristic (in the VRP research at least) however it has produced some of the most competitive results. This is borne out by results obtained here and in section 5.1. However, it did fail to find a feasible solution for one of the problem instances. Again this is probably due to the limited runtime permitted. If that problem instance is removed, then its average result becomes 96.39%, getting us much closer to our Enhanced Bees Algorithm results.

Instance	Bees Algorithm[1]		LNS[2]		Enhanced Bees[3]	
P01E51K05	557.17	*94.16%*	543.16	*96.58%*	524.61	*100.00%*
P02E76K10	925.97	*90.20%*	854.35	*97.77%*	835.77	*99.94%*
P03E101K08	873.75	*94.55%*	866.13	*95.38%*	826.14	*100.00%*
P04E151K12	1203.47	*85.45%*	1061.75	*96.86%*	1057.40	*97.26%*
P05E200K17	1520.89	*84.91%*	1360.85	*94.90%*	1360.85	*94.90%*
P06D51K06	565.21	*98.27%*	560.24	*99.14%*	555.43	*100.00%*
P07D76K11	1153.57	*00.00%*	1215.12	*00.00%*	913.37	*99.60%*
P08D101K09	941.34	*91.99%*	890.93	*97.19%*	865.94	*100.00%*
P09D151K14	1356.43	*85.71%*	1196.31	*97.18%*	1196.32	*97.18%*
P10D200K18	2153.48	*00.00%*	1515.32	*92.12%*	1490.49	*93.65%*
P11E121K07	1091.42	*95.48%*	1168.91	*89.15%*	1080.20	*96.47%*
P12E101K10	825.38	*99.30%*	826.14	*99.20%*	819.56	*100.00%*
P13D121K11	1662.70	*92.69%*	1555.29	*99.09%*	1555.30	*99.09%*
P14D101K11	903.25	*95.92%*	879.69	*98.49%*	866.37	*100.00%*
Average		*79.18%*		*89.50%*		*98.44%*

[1] Run for 60 seconds and configured as described in 5.2.1
[2] Run for 60 seconds and configured as described in 5.2.2
[3] Run for 60 seconds and configured as described in 5.1

Table 5.2: *Results Summary*

5.3 Comparison

Lastly we provide a comparison of the Enhanced Bees Algorithm with other well known results from the literature.

== PIC OF COMPARISON GRAPH ==

As can be seen from table 5.3 some of the best results known are due to Taillard's Tabu Search heuristic. He reaches 12 of the best known solutions from the set of 14 problems. The Enhanced Bees Algorithm by comparison finds 8 of the 14 best known solution. However, the

Instance	CW[1]	Sweep[2]	Gen Assign[3]	$3-Opt$[4]	SA[5]	TS[6]	ACO[7]	EBA[8]	Best Known[9]
P01E51K05	584.64	532	524	578.56	528 (167)	**524.61** (360)	**524.61** (6)	**524.61** (5)	524.61
P02E76K10	900.26	874	857	888.04	838.62 (6434)	**835.26** (3228)	844.31 (78)	835.77 (60)	835.26
P03E101K08	886.83	851	833	878.70	829.18 (9334)	**826.14** (1104)	832.32 (228)	**826.14** (50)	826.14
P04E151K12	1133.43	1079	1014	1128.24	1058 (5012)	**1028.42** (3528)	1061.55 (1104)	1045.88 (470)	1028.42
P05E200K17	1395.74	1389	1420	1386.84	1378 (1291)	1298.79 (5454)	1343.46 (5256)	1345.94 (1430)	1291.45
P06D51K06	618.40	560	560	616.66	**555.43** (3410)	**555.43** (810)	560.24 (6)	**555.43** (5)	555.43
P07D76K11	975.46	933	916	974.79	909.68 (626)	**909.68** (3276)	916.21 (102)	912.47 (225)	909.68
P08D101K09	973.94	888	885	968.73	866.75 (957)	**865.94** (1536)	866.74 (288)	**865.94** (20)	865.94
P09D151K14	1287.64	1230	1230	1284.64	1164.12 (84301)	**1162.55** (4260)	1195.99 (1650)	1186.65 (845)	1162.55
P10D200K18	1538.66	1518	1518	1538.66	1417.85 (5708)	1397.94 (5988)	1451.65 (4908)	1446.67 (1270)	1395.85
P11E121K07	1071.07	1266	-	1049.43	1176 (315)	**1042.11** (1332)	1065.21 (552)	1055.21 (345)	1042.11
P12E101K10	833.51	937	824	824.42	826 (632)	**819.56** (960)	**819.56** (300)	**819.56** (35)	819.56
P13D121K11	1596.72	1776	-	1587.93	1545.98 (7622)	**1541.14** (3552)	1559.92 (660)	1549.71 (435)	1541.14
P14D101K11	875.75	949	876	868.50	890 (305)	**866.37** (3942)	867.07 (348)	**866.37** (40)	866.37

[1] Clark Write's Savings (Parallel) algorithm. Implemented by Laporte and Semet [12].
[2] Sweep Algorithm due to Gillett and Miller [4]. Implemented by Christofides, Mingozzi and Toth [6]. Reported in [12].
[3] Generalized Assignment due to Fisher and Jaikumar [**?**]. Reported in [12]
[4] $3-Opt$ local searched used after the Clark Write's Savings (Parallel) algorithm. First improvement taken. Implemented by Laporte and Semet [12].
[5] Simulated Annealing due to Osman [28]. Runtime duration is given in parentheses and is reported in seconds on a VAX 86000
[6] Tabu Search due to Taillard [38]. Runtime duration is given in parentheses and is reported in seconds on a Sillicon Graphics Workstation, 36Mhz
[7] Ant Colony Optimisation due to Bullnheimer, Hartl, Strauss [2]. Bullnheimer et al. provided two papers on Ant Colony Optimisation for VRP, the better of the two is used. Runtime duration is given in parentheses and is reported in seconds on a Pentium 100
[8] Enhanced Bees Algorithm. Results are shown from the *best* configuration in section 5.1. Runtime duration is given in parentheses and is reported in seconds on a MacBook Pro 2.8 GHz Intel Core 2 Duo.
[9] Best known results as reported by Gendreau, Laporte, and Potvin in [22]

Table 5.3: *Results Comparison*

Enhanced Bees Algorithm is still very competitive. The runtime duration required to find a best known solution is smaller than many of the other meta-heuristics, although a direct comparison is hard to make as the many of the reported results were run on significantly older machines. And the solution quality is within 1% of the best known solutions, which places it within the ranks of the best meta-heuristics for VRP.

Chapter 6

Conclusion

6.1 Future Directions

Bibliography

[1] C. Strauss B. Bullnheimer, R. F. Hartl. Applying the ant system to the vehicle routing problem. *Meta-heuristics: Advances and trends in local search paradigms for optimization*, page 285–296, 1999.

[2] C. Strauss B. Bullnheimer, R. F. Hartl. An improved ant system algorithm for the vehicle routing problem. *Annals of Operations Research*, 89:319–328, 1999.

[3] M.L. Balinski and R.E. Quandt. On an integer program for a delivery problem. *Operations Research*, 12(2):300–304, 1964.

[4] L.R. Miller B.E. Gillett. A heuristic algorithm for the vehicle dispatch problem. *Operations Research*, 22:340–349, 1974.

[5] Jean Berger and Mohamed Barkaoui. A hybrid genetic algorithm for the capacitated vehicle routing problem. In *Genetic and Evolutionary Computation — GECCO 2003*, volume 2723 of *Lecture Notes in Computer Science*, pages 198–198. 2003.

[6] Mingozzi A. Christofides, N. and P. Toth. Exact algorithms for the vehicle routing problem, based on spanning tree and shortest path relaxations. *Mathematical Programming*, 20:255–282, 1981a.

[7] N. Christofides and S. Eilon. An algorithm for the vehicle dispatching problem. *ORQ*, 20:309–318, 1969.

[8] Fulkerson Dantzig and Johnson. Solution of a large-scale traveling salesman problem. *Operations Research*, (2):393–410, 1954.

[9] G. B. Dantzig and J. H. Ramser. The truck dispatching problem. *Management Science*, 6(1):80–91, Oct, 1959.

[10] C.F. Daganzo F. Robuste and R. Souleyrette II. Implementing vehicle routing models. *Transportation Research*, 24B:263–286, 1990.

[11] J.W. Wright G. Clark. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12:568–581, 1964.

[12] Frederic Semet Gilbert Laporte. Classical heuristics for the vehicle routing problem. Technical report, Les Cahiers du Gerad, 1999.

[13] W.R. Hamilton. Memorandum respecting a new system of roots of unity (the icosian calculus). *Philosophical Magazine*, 12, 1856.

[14] J. H. Holland. Adaptation in natural and artificial systems. *The University of Michigan Press*, 1975.

[15] J.R.C Holland I.M Oliver, D.J. Smith. A study of permutation crossover operators on the traveling salesman problem. *Grefenstette - Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, pages 224–230, 1987.

[16] G Laporte J Renaud, F F Boctor. A fast composite heuristic for the symmetric traveling salesman problem. *INFORMS Journal on Computing*, 8:134–143, 1996.

[17] B. Gavish K. Altinkemer. Parallel savings based heuristic for the delivery problem. *Operations Research*, 39:456–469, 1991.

[18] T.P. Kirkman. On the representation of polyhedra. *Philosophical Transactions of the Royal Society of London Series A*, 146:413–418, 1856.

[19] G. Laporte and Y Nobert. *Surveys in Combinatorial Optimization*, chapter Exact algorithms for the vehicle routing problem. 1987.

[20] Chee Peng Lim, Lakhmi C. Jain, and Satchidananda Dehuri, editors. *Innovations in Swarm Intelligence*, volume 248 of *Studies in Computational Intelligence*. Springer, 2009.

[21] G. Laporte M. Gendreau, A. Hertz. A tabu search heuristic for the vehicle routing problem. *Management Science*, 40:1276–1290, 1994.

[22] J Potvin M Gendreau, G Laporte. Metaheuristics for the vehicle routing problem. Technical report, Les Cahiers du Gerad, 1998, revised 1999.

[23] A H G R Kan M Haimovich. Bounds and heuristics for capacitated routing problems. *Mathematics of Operations Research*, 10:527–542, 1985.

[24] K. Doerner M. Reimann, M. Stummer. A savings-based ant system for the vehicle routing problem. *Proceedings of the Genetic and Evolutionary Computation Conference*, page 1317–1325, 2002.

[25] Y. Nagata. Edge assembly crossover for the capacitated vehicle routing problem. *EvoCOP, LNCS*, 4446:142–153, 2007.

[26] Road Transport Forum NZ. Road transport forum nz - transport facts. Website.

[27] I Or. *Traveling salesman-type combinatorial problems and their relation to the logistics of regional blood banking.* PhD thesis, Evanston, IL: Northwestern University, 1976.

[28] I. H. Osman. Metastrategy simulated annealing and tabu search algorithm for the vehicle routing problem. *Annals of Operations Research*, 41:421–451, 1993.

[29] D. Vigo P. Toth. The granular tabu search (and it's applications to the vehicle routing problem). Technical report, 1998.

[30] Koc Otri S Rahim S Pham, Ghanbarzadeh and Zaidi M. The bees algorithm. Technical report, 2005.

[31] Jean-Yves Potvin. A review of bio-inspired algorithms for vehicle routing. In *Bio-inspired Algorithms for the Vehicle Routing Problem*, pages 1–34. 2009.

[32] J. Robinson. On the hamiltonian game (a traveling salesman problem). *Research Memorandum RM-303*, 1949.

[33] Stefan Ropke. *Heuristic and exact algorithms for vehicle routing problems*. PhD thesis, Department of Computer Science, University of Copenhagen (DIKU), 2005.

[34] Alexander Schrijver. On the history of combinatorial optimization (till 1960). *?*, ?:?, ?

[35] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. pages 417–431. 1998.

[36] M. M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Oper. Res.*, 35(2):254–265, 1987.

[37] P.L. Juell S.R. Thangiah, K.E. Nygard. Gideon: A genetic algorithm system for vehicle routing with time windows. *Proceedings of 7th IEEE Conference on Artificial Intelligence Applications*, pages 322–328, 1991.

[38] E. D. Taillard. Parallel iterative search methods for vehicle routing problems. *Networks*, 23:661–673, 1993.

[39] Paolo Toth and Daniele Vigo, editors. *The vehicle routing problem*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2001.

[40] J. A. G. Willard. Vehicle routing using r-optimal tabu search. m.sc. dissertation, 1989.