

University of Southampton Research Repository
ePrints Soton

Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination

UNIVERSITY OF SOUTHAMPTON

**Polynomially Searchable Exponential
Neighbourhoods for Sequencing Problems
in Combinatorial Optimisation.**

by Richard K. Congram

submitted for the degree of Doctor of Philosophy
in Operational Research

Faculty of Mathematical Studies

April 2000

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF MATHEMATICAL STUDIES

MATHEMATICS

Doctor of Philosophy

POLYNOMIALLY SEARCHABLE EXPONENTIAL
NEIGHBOURHOODS FOR SEQUENCING PROBLEMS IN COMBINATORIAL
OPTIMISATION.

by Richard K. Congram

In this thesis, we study neighbourhoods of exponential size that can be searched in polynomial time. Such neighbourhoods are used in local search algorithms for classes of combinatorial optimisation problems. We introduce a method, called dynasearch, of constructing new neighbourhoods, and of viewing some previously derived exponentially sized neighbourhoods which are searchable in polynomial time. We produce new neighbourhoods by combining simple well-known neighbourhood moves (such as swap, insert, and k-opt) so that the moves can be performed together as a single move.

In dynasearch neighbourhoods, the moves are combined in such a way that the effect of the combined move on the objective function is equal to the sum of the effects of the individual moves from the underlying neighbourhood. Dynasearch neighbourhoods can be formed using dynamic programming from underlying moves:

- nested within each other;
- disjoint from each other;
- and in the case of the TSP overlapping one another.

Our dynasearch neighbourhoods made from underlying disjoint moves are successfully implemented within well-known local search methods to form competitive algorithms for the travelling salesman problem and state-of-the-art algorithms for the total weighted tardiness problem and linear ordering problem.

By viewing moves from some known travelling salesman problem neighbourhoods as a combination of underlying moves, each reversing a section of the tour, greater insight into the structure of the neighbourhoods may be obtained. This insight has both enabled us to calculate the size of a number of neighbourhoods and demonstrate how some neighbourhoods are contained within others.

Contents

1	Introduction	1
1.1	Sequencing problems and their solution	1
1.2	Local search	2
1.3	Sequencing problems studied	3
1.4	An outline of the thesis	4
2	Combinatorial Optimisation Problems	5
2.1	Introduction	5
2.2	Methods for solving combinatorial optimisation problems	5
2.2.1	Dynamic programming	6
2.2.2	Branch and bound	6
2.3	Complexity theory	8
2.4	Heuristics	12
2.4.1	The computational complexity of local search	13
3	Traditional Local Search Methods	15
3.1	Introduction	15
3.2	Descent	15
3.3	Threshold accepting	17
3.4	Simulated annealing	18
3.5	Tabu search	21
3.6	Genetic algorithms	23
3.7	Iterated local search	25
3.8	Guided local search (GLS)	27
3.9	Neural networks	29
3.10	Ant systems	31
3.11	Greedy randomised adaptive search procedure (GRASP)	33

3.12 Variable depth search	35
4 Polynomially Searchable Exponential Neighbourhoods	38
4.1 Introduction	38
4.2 Literature review	39
4.2.1 Overview	39
4.2.2 Pyramidal neighbourhoods	42
4.2.3 Twisted sequence neighbourhood	44
4.2.4 Edge ejection neighbourhoods	45
4.2.5 Balas's neighbourhood	45
4.3 Some possible issues in the practical application of a PSEN	47
4.4 Dynasearch	51
4.4.1 The dynasearch neighbourhood	51
4.4.2 A basic dynasearch algorithm	53
4.5 Traditional local search heuristics that appear to combine well with dynasearch	53
4.5.1 Iterated local search	53
4.5.2 Guided local search	54
5 The Total Weighted Tardiness Problem	56
5.1 Problem definition and literature review	56
5.2 Dynasearch	57
5.2.1 Swap and dynasearch swap neighbourhoods	57
5.2.2 Finding the best set of independent swaps	60
5.2.3 An alternative presentation of the dynamic program	61
5.2.4 Speed-ups	62
5.3 Implementation of iterated dynasearch	65
5.4 Computational experience	67
5.4.1 Experimental design	67
5.4.2 Multi-start and iterated dynasearch vs. first-improve and best- improve descent	70
5.4.3 Iterated dynasearch vs. tabu search	71
5.5 Conclusions	73

6 The Linear Ordering Problem	74
6.1 Problem definition and literature review	74
6.2 Dynasearch	77
6.2.1 Insert and dynasearch insert neighbourhoods	77
6.2.2 Finding the best set of independent insert moves	79
6.2.3 An alternative presentation of the dynamic program	81
6.2.4 Speed-ups	82
6.3 Implementation of iterated dynasearch	83
6.4 Computational experience	84
6.4.1 Experimental design	84
6.4.2 Multi-start and iterated dynasearch vs. first-improve and best-improve descent	87
6.4.3 Iterated dynasearch vs. tabu search and other local search methods	89
6.4.4 Using a hybrid acceptance criterion in iterated dynasearch . .	91
6.5 Concluding remarks	92
7 Travelling Salesman Problem	94
7.1 Problem definition and literature review	94
7.2 Competitive local search techniques	96
7.2.1 Speed-ups for edge exchanging descents	97
7.3 Dynasearch	99
7.3.1 2-opt and ds-2-opt	100
7.3.2 $2\frac{1}{2}$ -opt and ds- $2\frac{1}{2}$ -opt	102
7.3.3 3-opt and ds-3-opt	105
7.3.4 An alternative presentation of the dynamic program	106
7.3.5 Speed-ups	108
7.4 Implementation of iterated and GLS dynasearch	108
7.5 Computational experience	110
7.5.1 Experimental design	110
7.5.2 A comparison of a range of local search algorithms	112
7.6 Conclusions	114

8 Polynomially Searchable Exponential Neighbourhoods for the Linear Ordering Problem	115
8.1 The concept giving rise to the new PSEN	115
8.2 A method for calculating the size of large neighbourhoods that are formed by combining simple neighbourhood moves.	116
8.3 Introduction to PSEN for the linear ordering problem	119
8.4 Some block definitions used to define the neighbourhoods	119
8.5 Disjoint insert neighbourhood	121
8.6 Nested insert neighbourhood	123
8.7 Nested insert inside disjoint insert neighbourhood	126
8.8 Disjoint insert inside nested insert neighbourhood	129
8.9 Combined disjoint insert and nested insert neighbourhood	133
8.10 Conclusion	136
9 Polynomially Searchable Exponential Neighbourhoods for the Travelling Salesman Problem	137
9.1 Introduction	137
9.2 Some new notation and definitions	139
9.3 An alternative diagrammatic representation of some neighbourhood moves	142
9.4 Some block definitions used to define neighbourhoods	144
9.4.1 The blocks used to define the non-rotational strict reversal neighbourhoods	144
9.4.2 The blocks used to define the rotational strict reversal neighbourhoods	145
9.4.3 The blocks used to define the non-rotational reversal neighbourhoods	146
9.4.4 The blocks used to define the rotational reversal neighbourhoods	147
9.5 PSEN based on strictly disjoint and strictly nested reversals	148
9.5.1 Disjoint 2-opt	148
9.5.2 Rotational disjoint 2-opt	150
9.5.3 Rotational nested 2-opt	152
9.5.4 Restricted nested 2-opt	154

9.5.5	Nested 2-opt inside disjoint 2-opt	155
9.5.6	Nested 2-opt inside rotational disjoint 2-opt	157
9.5.7	Disjoint 2-opt inside rotational nested 2-opt	159
9.5.8	Rotational combined disjoint 2-opt and nested 2-opt	161
9.6	PSEN based on disjoint and nested reversals.	164
9.6.1	Disjoint reverse	164
9.6.2	Rotational disjoint reverse	166
9.6.3	Rotational nested reverse	167
9.6.4	Nested reverse inside disjoint reverse	170
9.6.5	Nested reverse inside rotational disjoint reverse	173
9.6.6	Disjoint reverse inside rotational nested reverse	174
9.6.7	The rotational twisted sequence neighbourhood	177
9.6.8	The rotational twisted sequence neighbourhood and k -opt neighbourhoods	180
9.6.9	Pyramidal (Sarvanov and Doroshko 1981a)	184
9.6.10	Rotational pyramidal (Carlier and Villon 1990)	187
9.6.11	Nested 2-opt inside pyramidal neighbourhood	189
9.7	Some 3-opt dynasearch neighbourhoods.	191
9.7.1	Rotational disjoint pure 3-opt	191
9.7.2	Rotational disjoint 3-opt (with 2-opt sub-neighbourhood)	194
9.7.3	Rotational nested pure 3-opt	196
9.7.4	Rotational nested 3-opt (with 2-opt sub-neighbourhood)	197
9.8	PSEN in which each neighbour is reachable by a set of k -opt moves but not by a set of reversals.	198
9.8.1	Overlapping 2-opt	198
9.8.2	Overlapping 3-opt	200
9.8.3	Other Overlapping neighbourhoods	203
9.9	Other PSEN neighbourhoods	203
9.10	Conclusion	204
10	Conclusions and Extensions	206

Acknowledgements

My special thanks go to my supervisor Chris Potts for his expert advice and guidance. In particular his excellent hands off approach to supervision allowing me great academic freedom to develop to my own ideas is appreciated.

I would like to thank: Steef van de Velde for organising a stimulating and very enjoyable two month visit to Erasmus University, Rotterdam and for his supervision and support whilst there; Leon Peeters and his friends for all patiently speaking English; and Rotterdam kajak club making my stay in the Netherlands so much easier, lending me a bike, canoe and paddles.

I am grateful to Gregory Gutin for some informative interactions by email and Thomas Stützle for lending me his iterated 3-opt code for the travelling salesman problem.

I would like to thank my room and pure mathematics colleagues in Southampton for many entertaining lunch times, and my canoeing friends at Woodmill for providing an environment in which to relax away from mathematics. Finally, I am grateful to my parents and family for their support.

Chapter 1

Introduction

1.1 Sequencing problems and their solution

Many real life problems involve the sequencing of discrete objects or events. Common examples include routing delivery vehicles and scheduling manufacturing and computer systems. It is often easy to come up with a solution to these types of problem if there are no constraints and the quality of the solution is not of interest. However, it is usually quite difficult to find a good solution, and sometimes almost impossible to find the best solution. There are generally a very large finite number of possible solutions from which to chose.

It is theoretically possible to calculate the value of all the possible solutions and select the best; this is known as complete enumeration. However, from a practical perspective, it is infeasible to follow such a strategy, particularly with large problems. Generally the number of possible solutions grows very quickly as the problem size increases. An example of a simplified real life problem with this characteristic is known as the travelling salesman problem, a description of which is as follows. Given a number of cities and the distance between any pair of cities, the travelling salesman problem consists of finding the shortest round-trip, visiting each city exactly once. If a particular travelling salesman problem contains 5 cities, there exist only 12 possible solutions. However, the number of possible solutions grows rapidly as the number of cities in the problem increases. For a problem containing 10 cities there already exist 181,440 possible solutions and for a travelling salesman problem containing 50 cities there exist over 3×10^{62} possible solutions! Clearly for problems over a certain size, it is not going to be possible to follow a strategy of complete enumeration.

A considerable amount of work has been done over the last five decades, developing methods which are capable of finding the best solution, without resorting to

the explicit examination of each possible alternative solution (i.e. complete enumeration). However, for some large problems in industry, even these methods take an impractical length of time. It is in this situation that methods known as heuristics are used. Heuristic algorithms often find good solutions in practice but do not guarantee to find the best solution. In fact, many heuristic algorithms give no guarantee on solution quality.

1.2 Local search

Local search refers to a group of heuristics which, when properly implemented, have been shown to be particularly effective in finding good solutions to both large randomly generated test problems and real industrial problems in reasonable time periods. Their popularity, particularly with practitioners in industry, has been aided by their easy implementation in modern computer languages, and the rapid improvement in computer power on which they are reliant.

Local search in some way mirrors the process by which a manager works in industry. When seeking to improve the operation of an existing plant, a manager may produce a new strategy from scratch without reference to previous strategies. However, following this approach is unlikely to yield a better strategy than the current one. A better and more widely used approach is as follows. Make a small change to the current strategy to form a new strategy. Trial the new strategy, and if it proves better than the previous strategy, then keep it; otherwise, return to the previous strategy. This process is normally repeated continually.

A local search algorithm starts with an initial solution and then continually tries to find better solutions by making small predefined changes to the current solution. For example, a predefined defined change may be swapping the positions of any two jobs in a sequence of jobs to be processed by a single machine. The solutions reachable by small predefined changes to the current solution are said to be in the current solution's neighbourhood. The local search algorithm searches the current solution's neighbourhood for a better solution. If a better solution exists, then one such solution is selected as the current solution and the search is repeated. Eventually, all of the solutions in the neighbourhood of the current solution will be worse than the current solution, and the algorithm terminates. The solution found may not be the best solution to the problem, but it is often quite a good one.

Above is a description of the simplest local search procedure. Many more complex local search procedures have been produced which, while often taking longer to run, usually produce much better quality solutions.

In designing a local search procedure, the size of the neighbourhood used is very influential in determining its effectiveness. A larger neighbourhood is generally capable of finding a better quality solution, at the expense of taking longer to search. This thesis is concerned with the study of large neighbourhoods which can be searched quickly, as well as demonstrating how with the right implementation some of these neighbourhoods can be of practical interest. The neighbourhoods in question are very large (in fact they are of exponential size) and yet they can be searched in polynomial time, which is necessary if they are to be of practical use. Many of the new polynomially searchable exponential neighbourhoods (PSEN) studied have a unique characteristic, in that a single move in their neighbourhood is often equivalent to a series of moves in other widely used neighbourhoods.

The effectiveness of one particular PSEN studied has been demonstrated, by its involvement in producing state-of-the-art local search algorithms for two types of problems, namely the single machine total weighted tardiness problem and the linear ordering problem. The neighbourhood's effectiveness is also demonstrated on the well known travelling salesman problem, which is described earlier in this chapter.

1.3 Sequencing problems studied

In this thesis we study in detail three types of sequencing problems. These problems are all widely studied in the operational research literature, and hence are suitable for testing our new local search algorithms. The single machine total weighted tardiness problem is a scheduling problem in which a set of jobs must be processed by a single machine. Each job has a date by which it should preferably be completed. If a job is not completed on time, it costs a set amount for each time period it is late. The objective is to minimise the total cost caused by late jobs.

The linear ordering problem can be viewed as a set of tasks, where between each pair of tasks there is a preference as to which task is performed first. The strength of preference is measured by a weight. The objective is to find the order of the tasks which maximises the sum of weights of the individual preferences that are satisfied. Although the above description makes the linear ordering problem sound

of little practical interest, this is not the case; its applications include triangulating input and output matrices in economics, ordering archaeological objects in time, and suppressing feedback in electrical circuits.

The travelling salesman problem (TSP) specifies a number of cities and the distance between any pair of cities. It is required to find the shortest round-trip visiting each city exactly once.

1.4 An outline of the thesis

The remainder of this thesis is structured as follows. Chapter 2 gives a general introduction to combinatorial optimisation problems and the types of methods, both exact and heuristic, that can be applied to them. Chapter 3 gives an introduction to local search heuristics. Chapter 4 reviews some previous work done on polynomially searchable exponential neighbourhoods and introduces one of our new neighbourhoods, disjoint dynasearch. The next three chapters give accounts of dynasearch applied to the total weighted tardiness problem, the linear ordering problem and the travelling salesman problem, respectively. Chapters 8 and 9 introduce a number of new polynomially searchable exponential neighbourhoods for the linear ordering problem and the travelling salesman problem, respectively. Finally in Chapter 10 we draw some conclusions, and comment on some possible extensions and interesting loose ends.

Chapter 2

Combinatorial Optimisation Problems

2.1 Introduction

A combinatorial problem is one where discrete choices must be made, and there exist a finite or a countably infinite number of feasible solutions to the problem. A set of choices is referred to as feasible, if it gives a valid solution. Each feasible solution has a value which is normally determined by a function called the objective function. The solution to the optimisation problem is the feasible solution which has the best objective function value. The best objective function value is the smallest objective function value for a minimisation problem and the largest for a maximisation problem. There may be more than one optimal solution. For example, in a single machine scheduling problem, with penalties for jobs finished late, the first two jobs in an optimal schedule A, B may be early and remain early if their order is reversed B, A ; the other jobs remaining unaffected by the swap.

2.2 Methods for solving combinatorial optimisation problems

For some combinatorial optimisation problems, rules have been discovered which quickly and efficiently construct an optimal solution. By quickly and efficiently we really mean in polynomial time, a term which will be explained shortly in the section on computational complexity: for example, consider a scheduling problem where the objective is to minimise the maximum lateness of n jobs. Each job takes a given time to be processed by the single machine, and has a due date by which it should preferably be completed. A job's lateness is defined as its completion time minus its

due date. An optimal solution can be obtained by processing the jobs in the order of their due dates, earliest to latest. It takes polynomial time to order the jobs in this way. More precisely, it requires $O(n \log n)$ time (see computational complexity section 2.3), which is a relatively small time interval.

For many combinatorial optimisation problems, no polynomial time algorithm is known, and it is in this situation that mathematical programming is often used. Mathematical programming methods include dynamic programming, branch and bound, as well as more general (mixed) integer programming methods. General methods, which do not make use of any specific properties of the class of problems, tend to be more time consuming.

A simple, but inefficient, method for solving combinatorial optimisation problems is complete enumeration; where every possible solution is calculated in turn and the solution with the best objective function is returned as the optimal. Implicit enumeration methods such as branch and bound and dynamic programming attempt to rule out groups of solutions as non-optimal without calculating them explicitly.

2.2.1 Dynamic programming

At each stage, sub-problems of increasing complexity are solved recursively using knowledge obtained from subproblems solved in previous stages of the algorithm. The subproblems determine more and more features of the optimal solution(s) until the problem is solved. Obviously, the method hinges on the ability to break the problem down into stages, which enable the subproblems in each stage to be solved by an efficient recursive algorithm from the solutions in the previous stage. Although dynamic programming is generally much more efficient than complete enumeration, it is still not in general a polynomial time algorithm.

2.2.2 Branch and bound

A branch and bound approach can in theory be used to solve virtually all combinatorial optimisation problems. However, the problem instance should not be too large, as the approach does not have polynomial time complexity.

To solve a combinatorial optimisation problem, we must find the best of a large number of possible solutions. Suppose we are dealing with a minimisation problem (an equivalent method exists for a maximisation problem). The method separates the possible solutions into subsets. It is possible to calculate a lower bound on the

objective function values in any subset (this will be discussed later). If the lower bound of a subset is greater than the best objective function value found so far, the optimal solution of the problem cannot lie in the subset and the subset is referred to as fathomed. No further work needs to be done on a fathomed subset.

The search of subset can be represented as a tree. The set of all possible solutions is represented by the first node. The set of solutions represented by a node N may be partitioned into subsets, which are also represented as nodes, joined by edges to node N .

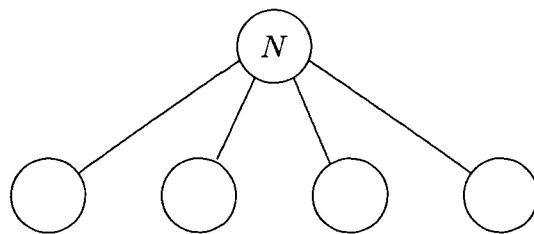


Figure 2.1: A general set of solutions in the tree and its subsets in the next level of the tree

At any point in time a decision must be made either to:

- a) partition the set of solutions at an existing node forming new nodes, or
- b) calculate a lower bound for the set at an existing node in an attempt to fathom the node.

At each successive level of the tree, the sets of solutions represented by the nodes become smaller, until at the final level each node represents a single solution. Eventually, the method terminates with all nodes fathomed.

Generally the tighter/better the lower bound at a node, the longer it takes to calculate, but the greater the chance of the node being fathomed. So there is usually a trade off between the speed of calculation and the quality of the bounds to be used in any branch and bound procedure.

2.3 Complexity theory

As indicated previously, in principle it is possible to find the optimal solution to any combinatorial problem. However, in practice to solve some large problems to optimality may take an unacceptable length of time, by all known algorithms. Very informally, complexity theory classifies problems in terms of the relative time frame in which optimal solutions can be found for large problem instances. (A problem *instance* is obtained by specifying particular values for all the problem parameters.) More significantly, computational complexity has also shown that the hardest combinatorial optimisation problems are in some sense all equivalent. In fact if an algorithm could be found capable of efficiently solving all instances of just one such problem, it would enable all of the difficult problems to be solved efficiently. The claims made will become clearer as a slightly more technical non-rigorous introduction to computational complexity is now given. Firstly it is useful to define some terms.

- The formal measure of the size of an instance is the *input length*. The *input length* for an instance I of a problem Π is roughly defined to be the number of symbols in the description of I obtained from a reasonable encoding scheme for Π .
- *Algorithms* are general, step by step procedures for solving problems.
- The *time complexity function* for an algorithm expresses its time requirements by giving, for each possible input length, the largest amount of time needed by the algorithm to solve a problem instance of that size.
- Let $f(n)$, $g(n)$ be functions from the positive integers to the positive reals.
 - (a) $f(n) = O(g(n))$ if there exists a constant $c > 0$ such that, for large enough n , $f(n) \leq cg(n)$
 - (b) $f(n) = \Omega(g(n))$ if there exists a constant $c > 0$ such that, for large enough n , $f(n) \geq cg(n)$
 - (c) $f(n) = \Theta(g(n))$ if there exist constants $c, c' > 0$ such that, for large enough n , $cg(n) \leq f(n) \leq c'g(n)$
- A *polynomial time algorithm* is one whose time complexity function is $O(p(\text{inputlength}))$ for some polynomial function p . Any algorithm that is

not a *polynomial time algorithm* is defined to be an *exponential time algorithm*; this includes non-exponential functions of the input length such as $O((\text{inputlength})!)$ and $O((\text{inputlength})^{\log(\text{inputlength})})$.

- A problem is referred to as *intractable* if it is so hard that no known polynomial time algorithm can solve it to optimality.
- A decision problem is a problem to which the solution is either yes or no.

Every optimisation problem has a related decision problem as illustrated by a general minimisation problem and a specific combinatorial optimisation problem, the travelling salesman problem below.

Minimisation problem

a) Optimisation problem

What is the optimal solution?

b) Decision problem

Is there a solution with an objective function value less than k ?

(where k is a given constant)

Travelling salesman problem (TSP)

where a salesman has to find a route visiting each of n cities once and only once.

a) Optimisation problem:

Which route minimises the total distance travelled?

b) Decision problem:

Is there a route for which the total distance travelled is less than k ?

The interest in decision problems stems from the following characteristic. Given a candidate that is supposed to prove that “yes” is the correct answer to a decision problem, one has only to calculate the candidate’s objective function value to verify the claim. For many problems this calculation can be performed in polynomial time. However, verifying the optimality of a candidate solution to an optimisation problem is as difficult in terms of computational complexity as it would be to find the optimal solution without any prior knowledge.

There is a class of decision problems called NP, which can be ‘solved’ by a polynomial time nondeterministic algorithm. Informally a problem can be ‘solved’

by such an algorithm if, for every instance of the problem, it is possible to guess a solution from which one can verify in polynomial time, that “yes” is the correct answer to the decision problem. Obviously, this is not the same as solving a problem in polynomial time because in practice it is impossible to guess the desired solution.

The class of decision problems for which there is a known polynomial time algorithm capable of solving any problem instance is known as P. Clearly, P is a subclass of NP.

No polynomial time algorithm is known for a large number of combinatorial problems, and it is a wide spread belief that it is impossible to solve many of these problems in polynomial time and therefore $P \neq NP$. Although no progress has been made on (dis)proving the conjecture $P \neq NP$, insight into the class NP has been gained through dividing the class into subclasses. Any problem in the subclass of NP called NP-complete problems has the property that all NP problems can be polynomially reduced to it. The significance of the class NP-complete is explained below. If problem Π can be reduced to problem Π' in polynomial time and a polynomial algorithm for solving problem Π' is found, then problem Π can also be solved in polynomial time. Problem Π can be solved in polynomial time as follows: Problem Π is reduced to problem Π' in polynomial time and then solved by the polynomial time algorithm for problem Π' . This means that if a polynomial time algorithm is found for any NP-complete problem then all problems in the class NP can be solved in polynomial time, i.e. $P=NP$.

Unary NP-complete (sometimes called strongly NP-complete) and binary NP-complete (sometimes known as NP-complete in the ordinary sense) form a partition of NP-complete problems. These concepts were introduced by Garey & Johnson 1978, and Lageweg, Lawler, Lenstra & Rinnooy Kan 1978. If a problem is unary NP-complete, then it is NP-complete even when the encoding scheme is allowed to represent numbers using *unary* notation (a string of n 1's representing the number n). The point of this differentiation is that binary NP-complete problems may have pseudo-polynomial algorithms that only display *exponential behaviour* with instances containing *exponentially large numbers*, and instances of this sort may be rare for the application of interest.

If $P \neq NP$ then it has been shown that there must exist problems that are neither in P nor NP-complete, this class is known as NPI (I standing for ‘intermediate’).

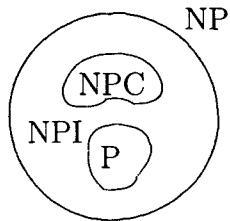


Figure 2.2: A simple diagram of NP (assuming $P \neq NP$)

So far we have only discussed the decision version of a problem. The term NP-hard is used to describe the corresponding optimisation problem of an NP-complete decision problem.

As indicated in this section, computational complexity gives some insight to the relative difficulty of a problem. However, time complexity is a worst-case measure, and therefore may give a misleading indication as to the difficulty of solving commonly faced real life instances of the problem. Only one instance of a given problem need have exponential time complexity for the problem to have exponential time complexity. There may be a corresponding discrepancy between an algorithm's time complexity and its expected running time in practice.

2.4 Heuristics

Given that many problems are NP-hard, see Garey and Johnson (1979), and it is widely believed that NP-hard problems cannot be solved to optimality in polynomial time (discussed in previous section), heuristics that find near-optimal solutions in reasonable running times are of great practical use.

A heuristic (sometimes known as an approximation algorithm) is defined by Reeves & Beasley (1995) as “a technique which seeks good (i.e. near-optimal) solutions at a reasonable computational cost without being able to guarantee either feasibility or optimality, or even in many cases to state how close to optimality a particular feasible solution is.”

Heuristics can be divided into two broad classes; constructive heuristics and local search methods. This thesis is concerned with the latter, although simple constructive heuristics are often used as the starting solutions for local search to improve.

A local search heuristic superimposes a neighbourhood structure on the solutions of an optimisation problem, that is, one specifies for each solution a set of ‘neighbouring’ solutions. Given a neighbourhood, a local search heuristic generally works as follows

A general local search heuristic

- 1) Obtain an initial solution to use as the first current solution,
(this may be randomly chosen or from a constructive heuristic).
- 2) Generate one or more solutions from current solution’s neighbourhood.
- 3) Use a given criteria to decide whether to:
 - a) accept a neighbour as the new current solution and restart at step 2;
 - b) continue searching by returning to step 2;
 - c) stop the heuristic giving a solution, often the best found.

The simplest local search is descent (sometimes known as iterative local improvement). This heuristic only allows improving moves, i.e. a neighbour is only accepted if it has a better objective function value. The algorithm keeps moving to a better neighbour, as long as one exists, until finally it terminates at a locally optimal solution (one that does not have a better neighbour).

Many local search techniques have evolved which allow uphill (worsening) moves under certain conditions. Even if a particular local search heuristic has been selected, there are often many further decisions to be made. There may be parameter values which control the timing, frequency and size of uphill moves. The size/type of neighbourhood must be determined, and in some cases how much of it is to be searched before considering making a move.

2.4.1 The computational complexity of local search

Although using local search is a practical way of obtaining reasonable solutions to NP-hard optimisation problems in a reasonable time, the time taken by the local search algorithms to find a local minimum is rarely known to be polynomially bounded. Formally, a local search problem is obtained from an optimisation problem, by defining a neighbourhood on the solution set. The problem is then as follows: given an instance, compute a locally optimal solution (i.e. a solution that does not have a strictly better neighbour). To produce theoretical results, we also require a standard local search algorithm to use on our local search problem. The standard local search algorithm starts from an initial solution and then moves iteratively from a solution to a better neighbouring solution, as long as there is one, until it finally terminates at a locally optimal solution. Note that if there is more than one better neighbouring solution, a selection rule is used to choose the particular neighbour to which the search should move. The selection rule used in the search may affect the computational complexity of the local search problem.

For a local search heuristic to be effective on a particular optimisation problem it must be possible, in polynomial time, to:

- i) generate an initial solution;
- ii) evaluate the cost of solutions;
- iii) search the neighbourhood.

The above is true for all well-known local search problems. Formally, local search problems for which the above properties hold are members of the PLS (polynomial time local search) class. Within the class PLS lies a class of problems called PLS-complete. A problem is said to be PLS-complete if every problem in PLS can be reduced to it (note the similarity with the classes NP-complete and NP). So, if a

PLS-complete problem is found that can be solved in polynomial time, it will enable all PLS-complete problems to be solved in polynomial time. Sadly, it is not even known whether many of the famous local search problems such as the travelling salesman problem (TSP) under the standard a) Lin Kernighan b) 3-opt or c) 2-opt neighbourhoods (using the standard selection rules) are PLS-complete.

Under a particular selection rule, the 2-opt heuristic for the Euclidean TSP problem can take an exponential number of iterations (Lucker 1976). The Euclidean TSP local search problem can be solved in $O(n^3)$ time with a sub-neighbourhood of 2-opt, where only 2-opt moves that involve edges that currently cross on the plane are considered (van Leeuwen & Schoone 1980). Thus the standard local search algorithm only requires $O(n^3)$ time to reach a local minimum. In this sub-neighbourhood any tour which has no crossing edges on the plane is a local minimum.

Finally, for practical purposes the average running time may be far more significant than the computational complexity of the running time (which is only a measure of the worst-case running time). The simplex local search method for solving linear programming problems is a prime example. Note that the simplex method for linear programming problems is an exact neighbourhood which means that every local optimum is also a global optimum. Therefore, the solution to the local search problem is the solution to the optimisation problem. Under most selection rules the simplex problem has been shown to take exponential time; however, in practice it is a very effective, widely-used algorithm that appears to have a very good average running time. Further, non-local search methods exist that are known to take a polynomially bounded time (ellipsoid Khachiyan 1979; interior point method Karmarkar, 1984, Grötschel, Lovasz & Schrijver, 1988) which on some types of problem instance are outperformed by simplex local search method. This is not to devalue the ellipsoid or interior point method's contribution to the field of computational complexity. The papers showed that the linear programming decision problem is in the class P, and not as Garey and Johnson in 1979 conjectured in NPI.

Chapter 3

Traditional Local Search Methods

3.1 Introduction

In this chapter we will describe some of the most widely used local search methods; in addition we will introduce some of the less established local search methods. For the less established local search methods we quote a few results from the literature in an attempt indicate their current ability to compete with other local search methods.

Throughout the chapter we will used $f(s)$ to denote the objective function of the solution s .

3.2 Descent

Most local search heuristics superimpose a neighbourhood structure on the solutions of an optimisation problem, that is, one specifies for each solution a set of *neighbouring* solutions. Descent (iterative local improvement) is the simplest local search procedure, which as the name suggests just descends to a local minimum by moving to a neighbouring solution with a lower objective function than the current solution at every step of the algorithm. Note: In some applications of descent using the acceptance criterion $f(s') \leq f(s)$, which allows neutral moves, may be more effective as long as the search is not allowed to cycle.

First improve descent algorithm

begin

 Get initial solution : s_0

$s = s_0$

repeat

 generate solution i.e neighbour s' from s

 if acceptance criterion: $f(s') < f(s)$ is satisfied then $s = s'$

```

until  $f(s') \geq f(s)$  for all neighbours  $s'$  of  $s$ 
end

```

With such a simple algorithm there are few choices to be made. We need to decide, as for all neighbourhood heuristics, how to obtain the initial solution and define the procedure by which neighbours are produced.

The initial solution can be chosen at random or produced by a constructive heuristic. For fast simple local search heuristics, like first improve descent, the choice of the initial starting solution may have a large effect on the quality of the solution returned by the algorithm. As can be seen in this thesis, for more powerful local search techniques involving longer searches the quality of the final solution given by the algorithm may be virtually independent of the initial solution. Generally for single runs of less powerful fast local search algorithms, such as descent, it is well worth using a constructive heuristic to find a good initial solution. Even for multiple runs of descent, it is usually more effective to use a constructive heuristic with a random element, than to use a series of totally random starting solutions.

The neighbourhood used is quite problem specific. A particular solution can often be represented as an assignment, partition, sequence or graph. For example, a particular solution of a single machine scheduling problem can usually be represented as a sequence, giving the order in which the jobs should be processed. There are two basic neighbourhoods for single machine scheduling problems: swap and insert. The smallest swap neighbourhood is *transpose*, sometimes known as *adjacent pairwise interchange*, where any two adjacent jobs in the current permutation may be swapped. For example, A, B, D, C, E is a neighbour of A, B, C, D, E because C and D can be swapped. A generalisation of this is the *swap* or *general pairwise interchange* neighbourhood where any two jobs may be swapped. For example, A, D, C, B, E is a neighbour of A, B, C, D, E because B and D can be swapped. The second basic neighbourhood is the *insert* neighbourhood, where a job is removed from one position in the schedule and inserted in another. For example, A, D, B, C, E is a neighbour of A, B, C, D, E because D can be removed from the sequence and inserted between jobs A and B . A generalisation of this is *block insert*, where a subsequence of jobs may be removed from one position in the sequence and inserted in another. For example, C, D, A, B, E is a neighbour of A, B, C, D, E because A, B can be removed from the sequence and inserted between jobs D and

E.

There are other types of descent algorithms. Instead of selecting the first improving move that is found, a subset of the neighbourhood can be searched before a move is selected. In a best improve descent algorithm, the whole neighbourhood is searched, and the algorithm then moves to the best solution in the neighbourhood. The method is often referred to as *steepest descent* for obvious reasons. The first improve variant of the algorithm is more widely used because, although it generally settles in a slightly worse local minimum, it is usually much faster.

3.3 Threshold accepting

Threshold algorithms, first introduced by Dueck & Scheuer (1990), can be thought of as a generalisation of descent in which uphill moves can be accepted.

In a threshold acceptance algorithm, neighbouring solutions are produced in a similar fashion to first improve descent. However, unlike in the descent algorithm, uphill moves may be accepted if the increase in the objective function δ is less than a threshold value τ_k . The threshold value τ_k is traditionally a non-increasing function of k , the number of iterations performed by the algorithm. i.e $\tau_k \geq \tau_{k+1}$. The threshold values are gradually lowered eventually becoming equal to 0, from which point only improving moves are accepted. Once the threshold value is 0, the remainder of the search is identical to a first improve descent.

Threshold accepting algorithm

begin

 Get initial solution : s_0

$s = s_0$

$k = 0$

repeat

 generate solution s' from s

 calculate the increase in the objective function $\delta = f(s') - f(s)$

 if acceptance criterion: $\delta < \tau_k$ is satisfied then $s = s'$

$k = k + 1$

until stopping criterion is satisfied

end

Another way of viewing threshold accepting is as a simplification of the more widely used method of simulated annealing; (the subject of the next section) where the probabilistic acceptance criterion used in simulated annealing is replaced by a deterministic criterion as described above. Although only a small amount of research has been performed on threshold accepting particularly in comparison with simulated annealing, because of the connection between the two methods many of the results from research performed on simulated annealing are directly relevant to threshold accepting. Some of the discussions on cooling schemes in the next section, are particularly relevant to the corresponding schemes by which the threshold values may be lowered.

3.4 Simulated annealing

Simulated annealing can be viewed as a generalisation of threshold acceptance, although it was introduced earlier by Kirkpatrick, Gelatt & Vecchi (1983) and Černý (1985). The idea, as the name suggests, originates from physics and from an attempt to simulate the evolution of a solid as it is slowly cooled from a liquid by Metropolis, Rosenbluth, Rosenbluth, Teller & Teller (1953). The particles, if cooled slowly enough, eventually settle in their ground state arranged in a highly structured lattice which minimises the energy of the system.

Kirkpatrick, Gelatt & Vecchi (1983) and Černý (1985) independently showed that the Metropolis algorithm could be applied to combinatorial optimisation problems by mapping the elements of the physical cooling process onto the elements in the optimisation problem, summarised by table 3.1 (Dowsland 1995).

Thermodynamic simulation	Combinatorial optimisation
System states	Feasible solutions
Energy	Cost
Change of state	Neighbouring solution
Temperature	Control parameter
Frozen state	Heuristic solution

Table 3.1: Mapping elements from the physical cooling process

As mentioned earlier simulated annealing can be described as a generalisation of threshold accepting, where the deterministic acceptance criterion is replaced by

a non deterministic one. The probability of acceptance p_k is given by :

$$p_k = \begin{cases} 1 & \text{if } \delta \leq 0 \\ \exp(-\delta/t_k) & \text{if } \delta > 0, \end{cases}$$

where δ is the increase in the objective function due to the candidate move. Here the temperature t_k is gradually lowered until only improving moves are accepted, and a local minimum is found and returned by the algorithm.

The behaviour of the simulated annealing algorithm with varying temperature can be modelled as a non-homogeneous Markov chain. A large amount of research has been done on the statistical behaviour of simulated annealing. One interesting paper by Hajek (1988) shows a connection between the shape of the objective function landscape, in particular the depth of the deepest local minimum, and the rate of cooling to guarantee asymptotic convergence to the global minimum. Unfortunately, the time taken to achieve the required convergence grows exponentially in terms of problem size, and often requires more iterations than an exhaustive search. However, simulated annealing has often been shown to be effective in a practical context, where considerably less time is allowed. With a sensible cooling scheme, good local minima can often be obtained.

Simulated annealing algorithm

begin

 Get initial solution : s_0

$s = s_0$

$k = 0$

repeat

 generate solution s' from s

 calculate the increase in the objective function $\delta = f(s') - f(s)$

 accept solution s' i.e $s = s'$ with probability p_t

 where $p_k = \begin{cases} 1 & \text{if } \delta \leq 0 \\ \exp(-\delta/t_k) & \text{if } \delta > 0 \end{cases}$

$k = k + 1$

until stopping criterion is satisfied

end

The idea in any cooling scheme is to obtain the best quality solution in a given time. Any simulated annealing algorithm can give better quality solutions if more iterations are allowed at each temperature. After all, even random sampling over the

solution space will eventually produce the optimal solution. So it is only meaningful, as with other heuristics, to compare algorithms over the same time scale.

The choice, therefore, is what fraction of the time available to spend at different temperatures. How high should the cooling scheme start? One idea is to save time by starting at a moderately low temperature but from a good initial solution obtained from a constructive heuristic. If the cooling scheme starts at a high temperature, the more traditional approach, there appears little point wasting time constructing a good initial starting solution which will soon be destroyed by the large uphill moves allowed early in the algorithm, leaving little correlation between the quality of the initial solution and the quality of the final local minimum in which the algorithm terminates. There is clearly a grey area in the choice of the quality of initial solutions for starting temperatures between the two extremes.

Obviously if all the objective functions in a problem instance were multiplied by a large constant the underlying problem would be unchanged but the temperatures required in an effective simulated annealing cooling scheme would need to be scaled up. So the temperatures used in the scheme are in some ways dependent on the instance.

The cooling scheme can be solely determined before the execution of the algorithm, which is a *static cooling scheme*, or adaptively changed during the execution of the algorithm, which is a *dynamic cooling scheme*.

Even for static cooling schemes, there are endless combinations of temperature value and number of iterations at each temperature. The gaps between temperature values could be smaller at lower temperatures. More iterations could be performed at the lower temperature values. There could be many temperature values with only one iteration performed at each, or a few temperature values with many iterations at each.

In dynamic cooling schemes, the temperature may even be cycled in the cooling scheme, starting high, cooling and then raising the temperature again. This may effectively produce successive periods of diversification and intensification (much as Glover (1990) advocates in tabu search). It may be worth recording the local minimum found at the end of each intensification stage as the final local minimum may not be the best visited.

Chained local search, a variant of iterated local search, introduced by Martin, Otto & Felten (1991,1992) uses simulated annealing to move around a neighbour-

hood of local minima (see subsection 3.7). This method also uses successive periods of diversification to move away from the current local minimum by means of a ‘kick’ and intensification to attempt to find a new local minimum.

3.5 Tabu search

Tabu search was formalised by Glover (1986), but the basic ideas were also sketched by Hansen (1986). There is less theoretical interest in the technique; unlike simulated annealing no proof of convergence exists. However, its proven effectiveness on many problems has made it popular with the local search practitioner.

A significant difference between simulated annealing and tabu search is their use of memory. Simulated annealing has no memory, whereas tabu search consults information collected about previous moves before each new move is made.

Tabu search algorithm

begin

 Get initial solution : s_0

$s = s_0$

repeat

 find s' the best legal neighbour of s

 update tabu list and aspiration criterion which define legal neighbours.

$s = s'$

until stopping criterion satisfied

end

In many early implementations of tabu search the whole neighbourhood is searched and the best neighbouring solution found is moved to. So initially the search descends like a best improve descent to a local minimum. If the best solution in the current solution’s neighbourhood was always selected, without restriction, the search would be likely to just cycle, repeatedly returning to the local minimum. This is not allowed because the tabu list restricts the search returning to recently visited solutions. So, at every point in the search, the best legal move is selected, even if this move results in an increase in the objective function. A move is illegal if an attribute of the move or the solution to be moved to is on the tabu list, unless the aspiration criterion is also satisfied. The aspiration criterion is satisfied if the

solution has particularly good characteristics. In this way, the tabu list forces the search out of the current local minimum and into other basins of attraction.

As indicated previously, the main purpose of a tabu list is to avoid cycling back into previously searched local minima. A decision to be made is what should be held on the tabu list. The objective function values of previous solutions visited could be held on the list, this may be too restrictive if a problem has large flat areas of solution space, and therefore a large number of solutions with the same value. Holding the solution itself in the tabu list would overcome this problem, but produces a greater memory requirement and a longer checking time. An attribute of the solution may have a much lower memory requirement. Holding a solution attribute on the tabu list can be effective, particularly if an aspiration criterion is used. Alternatively the inverse of moves previously made or moves with certain attributes may be made tabu, where again the effectiveness of the search may be improved if an aspiration criterion is used in an attempt to prevent good solutions being missed. The reasoning behind holding the inverse of moves recently made on the tabu list is to stop previous moves made being reversed and thereby hopefully avoid cycling.

If there is a very good solution in the neighbourhood, for example one with the best objective function found so far, an aspiration criterion attempts to ensure that a move is made to this solution, even if an attribute of the solution is tabu.

The length of the tabu list may be a crucial factor in determining the effectiveness of a search. The shorter the tabu list the more likely the search is to cycle. The longer the tabu list the more likely good solutions are to be missed even with an aspiration criterion. The length of an effective tabu list is much shorter than may be expected; often a length of only 7 is quite adequate.

The most common aspiration criterion is based on quality, often defined by the objective function of a candidate neighbour. The aim is to intensify the search in a promising area. However, an allowance can be made for the influence of the candidate neighbour. A candidate neighbour may be influential if it has a significantly different structure to the current solution, thus diversifying search into new areas of the solution space.

Smaller neighbourhoods than those used in simulated annealing may be most effective in tabu search, due to the time requirement to search the whole neighbourhood before the best legal move is chosen.

Generally, it appears to be a good idea to use a constructive heuristic to find an initial solution because of the time saved descending to the first local minimum. However, the quality of the final solution found by tabu search, for reasonable run times, appears to show little correlation with the quality of the initial solution. Therefore, for long runs, the use of a constructive heuristic becomes unimportant as, the time saved descending to the first local minimum becomes a negligible fraction of the total search time.

3.6 Genetic algorithms

Genetic algorithms are modelled on reproduction in nature, each individual in the population represents a solution to the combinatorial optimisation problem. The *survival of the fittest principle* is used to attempt to guide the search towards an ever improving population of solutions. The quality of a particular solution is determined by a monotonic transformation of the objective function, which is usually referred to as the *fitness score*, a term from theoretical biology.

Although the ideas behind genetic algorithms stem from sexual reproduction in living things, in the context of producing high quality solutions to combinatorial problems there appears little reason to restrict algorithms to mimic nature in every characteristic. The genetic local search algorithm is an example of a hybrid of a pure genetic algorithm and local search (Mühlenbein, Georges-Schleuter, and Krämer 1988). In the recombination part of the algorithm, two or more solutions are recombined to produce an offspring. Mutation may be used to produce further variation. In nature, mutations are normally caused by randomly occurring copying errors in the replication of sections of chromosomes. Mutations in the algorithm may be produced by randomly selecting a neighbour of the offspring solution. In the next part of the algorithm, local search is used to improve the offspring solutions. Finally a selection procedure is used to reduce the population back to its original size (in nature, this is the so called *survival of the fittest*, since only the proportion of the population which are best adapted survive, a cold winter for example, to reproduce in the following spring).

Genetic local search algorithm**begin***Initialise.* Construct an initial population of M solutions.*Improve.* Use local search to replace the M solutions in the population by M local optima.**repeat***Recombine.* Augment the population by adding m offspring solutions; the population size now equals $M + m$.*Improve.* Use local search to replace the m offspring solutions in the population by m local optima.*Select.* Reduce the population to its original size by selecting M solutions from the current population.**until** stopping criterion satisfied**end**

To implement a genetic algorithm, decisions must be made, including: producing the initial population, size of the population, selecting parents from the population, recombining the parents, size and frequency of mutations, and stopping criteria.

There is some evidence that using a constructive heuristic to produce a good initial population (instead of producing the population randomly) can reduce the running time of the algorithm without a significant deterioration in solution quality. There seems to be little consensus on population size. Effective algorithms with population sizes ranging from 20 to several thousand have been produced. A large population may improve the quality of the solution but at the cost of a slower rate of convergence. Selection of parents for recombination is normally at random from the population, but with a bias towards fitter individuals. While mutations diversify the search, their size and frequency is specific to the particular implementation, and are often partially determined by trial and error. Possible stopping criteria include: stop after a given number of generations, and stop if the improvement in average fitness of the population from the preceding generation is smaller than a given amount.

3.7 Iterated local search

A simple but effective procedure to explore multiple local minima, which can be implemented in any type of local search algorithm, is to perform multiple runs with the algorithm, each using a different starting solution. In the simplest form of this approach, these starting solutions are generated randomly, and do not rely on the results of previous runs of the local search algorithm. Typically, the starting solutions are chosen randomly, or by applying some constructive heuristic with varying parameter values or a randomised element. We refer to this approach as *multi-start* local search.

An intuitively appealing idea is to restart near a local minimum, rather than from a randomly generated solution. Under this approach, the next starting solution is obtained from the current local optimum (where the current local optimum is usually either the best local optimum found thus far, or the most recently generated local optimum) by applying a prespecified type of random move to it. We refer to such a move as a *kick*, and to the approach as *iterated* local search. In this way, not all the good characteristics from previously found solutions are lost. As early as 1981, Baxter (1981) used a kick (temporarily penalising a set of solution characteristics by a fixed amount), to dislodge the search away from the current local optimum to the next starting solution. The idea of using random kicks traces back to Baum (1986A, 1986B), who presents iterated 2-opt and 3-opt algorithms for the traveling salesman problem, in which a single random 2-opt move is used as a kick to move away from the current local optimum to the next starting solution. Although his computational results are discouraging, more recent research (for example, see Johnson (1990), Johnson & McGeoch (1997), Johnson, Bentley, McGeoch & Rothbergh (1998), Martin, Otto & Felten (1991, 1992), Storer, Wu & Vaccari (1992), Charon and Hudry (1993), Martin & Otto (1996), Brucker, Hurink & Werner (1996, 1997), Lourenço (1995), Lourenço & Zwijnenburg (1996) and Stützle (1998a)) shows that iterated local search can be extremely competitive for a range of local search problems.

Essentially, this type of algorithm is performing a local search on the local optima. Following the generation of an initial current solution, a traditional descent or other local search algorithm is applied to find a solution which is a local optimum, set as the current solution. If the local optimum found is not the first generated, a

decision is made as to whether to leave the current solution unchanged or to replace the current solution with the new local optimum.

Having decided on a current solution, a kick is applied. If the solution resulting from the kick has some unsatisfactory features, then it can be rejected before applying a local search algorithm to find a local optimum. In this case, another kick is executed, and the process is repeated until the kick is accepted. Then, a local optimum is found, and the entire procedure is repeated until a stopping criterion (usually based on computation time or the total number of iterations) is satisfied.

Various criteria for accepting S are possible. Martin, Otto & Felten (1990, 1991) in their *chained local improvement algorithm* suggested a simulated annealing acceptance criterion, where the new move is accepted with a probability $e^{-\delta/t_k}$, where δ is equal to the new local optimum minus the current local minimum. Later Johnson (1990) simplified the chained local improvement algorithm. In particular, he made the decision as to whether to move to a new local optimum deterministic: accept new local optimum if it is better than the current solution. Johnson refers to the algorithm as iterated local optimisation, perceiving it as a genetic algorithm with a population of one, the kick corresponding to a mutation. When applied to the TSP, Johnson's method with no simulated annealing criterion still often successfully gives the global optimum.

Regarding the choice of kick, there is ample computational evidence that iterated local search is competitive only if the kick is sufficiently large to move to a solution that is not too close to the local optimum. If it is not, then the effect of the kick might be reversed in a single or small number of iterations, and the kick would literally lead nowhere—this is most likely explanation of Baum's disappointing results. On the other hand, the kick should not be too large, or else the good characteristics of the previous local optimum are lost, and the procedure is then effectively multi-start rather than iterated local search. For a k -exchange neighbourhood, an effective kick would then be a single (or several) $(k + 1)$ -exchange, or several k -exchanges. As an example, for the travelling salesman problem, both Johnson (1990) in his iterated 3-opt and iterated Lin-Kernighan algorithms, and Martin, Otto & Felten (1991, 1992) in their iterated simulated annealing algorithm with the 3-opt neighborhood structure, use a specific 4-opt move (a double bridged 4-opt move) as a kick. However, Martin, Otto & Felten (1991, 1992) had a length restriction on the edges added by the kick whereas Johnson (1990) removed the

length restriction finding it had little effect on the algorithm.

Stützle & Hoos (1999) analysed the run time behaviour of iterated local search for the TSP and found that for long runs on large problems, there was some stagnation in improvements. For very long runs, an improvement in performance can be found by restarting even from a randomly generated solution, although periodically using a larger kick is more efficient approach. This requirement for periodic diversification to avoid stagnation is not surprising as it is found to be effective in many other local search methods.

3.8 Guided local search (GLS)

Guided local search (Voudouris 1997, Voudouris & Tsang 1998, 1999) originates from work done by Wang and Tsang on Constraint Satisfaction Problems (Wang & Tsang 1991).

It is unfortunate that the name Guided local search has already been used by (Balas & Vazacopoulos 1994, 1998) which could cause confusion. Balas and Vazacopoulos produced a research report in 1994 before publishing the work in the journal *Management science* in 1998. In this thesis, any future reference to guided local search (GLS) refers to the method of Voudouris and Tsang.

The method can be applied to any combinatorial optimisation problem. We will just describe GLS in relation to the TSP problem. The first stage of the method is to find a local minimum using any local optimiser (e.g. 2-opt descent, 3-opt descent or Lin-Kernighan which will be described in section 3.12). Then a solution feature must be penalised; the obvious solution feature to penalise in the TSP problem is long edges in the current local minimum. Let p_{ij} denote the current penalty for each edge i, j , where all penalties are initially set to zero. The edge (or edges) i, j of the current tour with the largest value of $d_{ij}/(1 + p_{ij})$ is (are) penalised further by increasing p_{ij} by one. The augmented landscape, which is defined by the distances $D_{ij} = d_{ij} + \lambda p_{ij}$ where λ is a parameter, is then searched using the local optimiser to find a local minimum. The true value of each local minimum found in the augmented landscape is recorded. The process is repeated until a given stopping criteria is reached.

The parameter λ determines the relative effect of the penalty. Obviously it is the size of λ relative to the general size of the edges around a local minimum that is important. λ is quite heavily dependent on the particular problem instance. λ

can reliably be determined by multiplying a parameter a by the average edge length in the first local minimum found. The choice of parameter a is dependent mainly on the local optimiser and appears much less dependent on the particular problem instance than λ . A low value of a intensifies the search, slowly allowing the algorithm to search the current area, before being forced by penalties to search other areas. Conversely, a high a diversifies the search, quickly allowing it to search a larger area in a given time. A high value of a causes GLS decisions to be in full control of the local search, overriding any local gradient information. A low a leaves the local optimiser in more control, but leaves GLS requiring many penalty cycles before the local minimum is escaped and a move is executed.

GLS has been successfully applied to the TSP (Voudouris & Tsang 1999), with the version using 2-opt as the local optimiser found to be the most effective, although even GLS 2-opt appears unable to compete with the best implementations of iterated Lin-Kernighan. Voudouris (1997) claims that GLS is competitive for the quadratic assignment problem. Moreover Voudouris & Tsang (1998) claim that GLS is state-of-the-art for the frequency assignment problem seeming to outperforming the best methods resulting from the European wide CALMA project (reviewed by Tiourine, Hurkens and Lenstra 1995). Based on the results given in the respective papers it appears that GLS also outperforms the more recent implementation of an ant system algorithm of Maniezzo, Carbonaro & Montemanni (1999).

3.9 Neural networks

The first neural network architecture was due to McCulloch and Pitts in 1943. However, it was not until 1985 that a neural network was used to solve a combinatorial optimisation problem (Hopfield & Tank 1985).

Neural networks are mainly computationally inspired and in some sense attempt to mimic the biological neural networks inside the brains of animals. Traditionally computers are structured and used to sequentially process a set of predetermined instructions in order to solve a given problem. As far as we know, biological neural networks solve problems very differently, not only in the way they encode problem instances, but also in the way that they effectively use billions of simple processors (neurons) many working in parallel to find a solution. The biological neurons, whilst being individually capable of very little, collectively form a brain which is both incredibly powerful and adaptable.

We will now describe some of the basic elements of an artificial neural network. They consist of a network of neurons, each of which normally takes a value within the interval $[0,1]$ (or $[-1,1]$). The value of a particular neuron is determined by the inputs it receives from other neurons to which it is connected. The value of each input is a given by an interaction between the value of the particular neuron giving rise to the input and the weight of the connection.

There are two main kinds of neural network architecture, feed-forward and feed-back. Feed-forward networks have successfully been applied to feature recognition problems. Feed-back networks have been applied to optimisation, and so it is this type of neural network on which we will focus. In an attempt to give a feel for how a neural network may work, we will describe a well-known application of a feed-back network: Hopfield and Tank's network for the TSP.

Hopfield & Tank (1985) encoded the TSP as a n by n network of neurons x_{ij} . The neurons x_{ij} take on arbitrary values in the interval $[0, 1]$. Each row i of the network corresponds to a city and each column j to a relative position in the tour. The assignment $x_{ij} = 1$ is taken to mean that city i is in the j th position in the tour. So for the network to correspond to a valid tour there must be only a single neuron in each row and column with the value 1, and all other neurons must have the value 0. This validity criteria results in negative weights (inhibition) between nodes in the same row, and between nodes in the same column. The objective in

the TSP to find the shortest tour is encoded by weights between neurons in adjacent columns x_{ij} and $x_{i+1,k}$ for $j \neq k$ which are inversely proportional to the length of the edge between the two nodes represented by the rows j and k .

Since Hopfield and Tank's paper in 1985 a great deal of research has been conducted on neural network applications to combinatorial optimisation problems. Peterson & Söderberg (1989) found an alternative encoding often to be more effective; they used multi-state so called *Potts neurons*. The effect on the encoding of the TSP is that the n neurons representing the position in the tour of a particular city in the Hopfield & Tank (1985) network are replaced by a single n dimensional *Potts neuron*. Hybrid neural network algorithms have been suggested which appear to be particularly effective on the TSP such as 'deformable template' algorithms (Durbin & Willshaw 1987).

Although Peterson & Söderberg (1997) remain optimistic stating: 'with respect to quality of the solutions, the ANN (artificial neural network) methods produce very competitive results compared to other approximation schemes' before referring to Peterson (1990) for details of comparisons for the TSP. This is in direct conflict with the assessment by Johnson and McGeoch of neural networks in their review of local search heuristics for the TSP, including the paper by Peterson (1990). Johnson & McGeoch (1997) state that 'no neural network algorithm can produce tours even as good as those we have reported for single run 3-opt (first improve descent), and most take substantially more time, at least on sequential machines'. One glimmer of light was the extended GENET of Boyce, Dimitropoulos, vom Scheidt & J.G Taylor (1995) for the frequency assignment problem which, although no longer a competitive approach, in 1995 could compete with the best algorithms known including simulated annealing, tabu search and genetic algorithms implemented by creditable European research groups. (see the review by Tiourine, Hurkens & Lenstra 1995). Generally, however, neural networks appear currently to be far from competitive when compared with other local search techniques for combinatorial optimisation problems.

3.10 Ant systems

Ant systems have their roots in the PhD thesis of Dorigo (1992), and the papers of Dorigo, Maniezzo & Colomi (1991, 1996).

Real ants, although almost blind, are capable of finding the shortest path from a food source to the nest. They also have the ability to adapt to changes in the environment, for example finding a new shortest path, if an existing route is blocked by an obstacle. The primary means by which ants communicate information about paths is by pheromone trails. Ants deposit pheromone while walking, and each ant probabilistically prefers to follow a direction rich in pheromone. So while an isolated ant moves essentially at random, an ant encountering a previously laid trail can detect it and with a probability determined by the strength of the particular pheromone trial follow it, thus reinforcing the trail with its own pheromone. Finally, it should be noted that the pheromone forming the trails continually evaporates, so that the strength of a trail reduces, if its use diminishes.

The following simple example gives insight into how, by following the above behaviour, the shortest path can be found and maintained even when conditions change. It is assumed that all ants move at the same speed. Imagine a straight pheromone trail between a food source and an ants nest. Now consider how the ants following the trail will react to a post put in the ground directly on the line of the trail. Since the ants will have to form a new trail round the post, probabilistically half will go left and half right. Given that it is a longer distance round the post left than right, the ants that select right will make more journeys over their trail per unit time than those ants that choose left. Therefore, the pheromone trail right of the post will become stronger and more ants will begin to follow this route. The greater number of ants following the trail right will make the trail even stronger, and even more ants will follow this trail. Soon all the ants will be following the trail right.

When using an artificial ant colony as an optimisation tool, known as an ant system, rather than as a simulation to learn more about ant behaviour, there is no need for their abilities to be constrained to those of real ants or for the environment in which the ants work to be similar. Artificial ants in ant systems can see, have a memory, and live in an environment where time is discrete. For example, in the ant system for the TSP proposed by Dorigo, Maniezzo & Colomi (1996), each ant can

see the distance from one city to another. Each ant can therefore decide whether to use an edge based not only on the amount of pheromone trail on it, but also the edge's length. In addition, each ant has a memory enabling it to form legal tours. A tabu list stops the ant from revisiting cities until the tour is complete.

In a simple ant system for the TSP about 10 ants are placed at random cities. At each time step the ants move to new cities and modify the pheromone trail on the edges used. The new cities to be moved to are selected from the cities which are not part of the particular ant's current partial tour, with a probability that is a function both of the amount of pheromone trail on, and length of, the connecting edge. Once all of the ants have formed complete tours, extra pheromone is added to the shortest tour formed, and the ants start building new tours as before. In an attempt to mimic the way in which the real ant formed a new route around the post, the amount of extra pheromone added is inversely proportional to the length of the tour.

Two of the most effective ant systems appear to be the approximate non-deterministic tree search (ANTS) ant system introduced by Maniezzo (1998) and the Min-Max ant system by Stützle & Hoos (1998a, 1998b). The ANTS algorithm differs from a tree search algorithm in that it lacks a complete backtracking mechanism, forming an incomplete exploration of the search tree. The Min-Max ant system is a hybrid ant system, allowing some ants to improve their solution by using a local search heuristic. We refer to Dorigo & Di Caro & Gambardella (1999) for a list of ant systems proposed.

Maniezzo, Carbonaro & Montenmanni (1999) claim that their ANTS local search algorithm for the frequency assignment problem is at the level of state-of-the-art. However, they appear unaware of the guided local search method by Voudouris & Tsang (1998). Both Voudouris & Tsang (1998) and Maniezzo, Carbonaro & Montenmanni (1999) compare their methods with the best results from the European wide CALMA project, the results of which are reviewed by Tiourine, Hurkens & Lenstra (1995), and from these comparisons it appears that Voudouris & Tsang's guided local search algorithm remains the state-of-the-art method for the frequency assignment problem.

The Min-Max algorithm of Stützle & Hoos (1998a, 1998b) and ANTS local search algorithm of Maniezzo (1998) both appear to be competitive for the quadratic assignment problem. The Min-Max ant system has also been shown to be competitive

for flow shop scheduling Stützle (1998c). Generally, the most effective algorithms based on ant systems appear to be able to compete with other local search methods.

3.11 Greedy randomised adaptive search procedure (GRASP)

GRASP is a multi-start procedure which consists of two parts, a construction phase and a local search procedure. It was first introduced by Feo & Resende (1989). The construction phase is normally a randomised greedy construction heuristic which due to its random element yields a different initial solution each time it is run. The construction phase is therefore capable of producing a diverse set of starting solutions from which the local search procedure finds a local minimum. The local search procedure in the basic GRASP is a simple descent algorithm. GRASP has some similarities with iterated local search in the way it repeatedly descends to local minima from different start points. However, basic GRASP has no memory, making no use of information gathered in previous iterations. In contrast, the standard iterated local search ‘kicks’ from the best solution currently found so that the new initial solution shares some characteristics with the best solution currently known. The fact that the basic GRASP algorithm has no memory makes it perfectly suited for parallelisation, and only as a final step do the solutions found by the different parallel processors need to be compared to return the best solution found overall by the procedure.

The basic GRASP construction phase is similar to the semi-greedy heuristic (Hart & Shogan 1987) which predates it. At each construction iteration a candidate list is formed, which lists all of the elements which can be added to the current partial solution in order of their myopic benefit, as measured by a so called ‘greedy’ function. A ‘random’ procedure selects an element from the candidate list to be added to the partial solution. The construction heuristic is ‘adaptive’ because the myopic benefit as measured by the greedy function for each candidate is updated at each iteration of the construction phase, thereby reflecting changes due to the selection of the previous element.

Although GRASP requires randomness in the construction phase to diversify the search, (without any randomness the same local minimum would be found at each restart) the more randomness in the procedure the worse the average quality of the

initial solution constructed.

Many enhancements to GRASP seem to attempt to incorporate some form of memory, in order to make use of information gathered in previous iterations. Prais & Ribeiro (1998) proposed a scheme where the parameters determining the level of randomness in the construction phase are periodically adjusted in response to the quality and variation of local minima found so far by the search. Also, Fleurent & Glover (1998) maintain a set of elite solutions to be used in the constructive phase.

Surprisingly, Resende (1998) in his otherwise excellent preprint review paper makes no mention of the effectiveness of GRASP as compared to other local search techniques. Many authors of GRASP algorithms appear to prefer to restrict the comparisons of their results to those of other GRASP algorithms. Maniezzo (1999) compares three algorithms for the quadratic assignment problem, his own ANTS search, the GRASP algorithm of Li, Pardalos & Resende (1994) and the tabu search algorithm of Taillard (1991). Whilst he claims that the Li, Pardalos & Resende algorithm is the best GRASP algorithm currently known for the quadratic assignment problem, he finds the GRASP algorithm produces considerably worse results than the other two algorithms in the CPU time allowed. We suspect that findings of Maniezzo (1999) are not unique, and that the best GRASP algorithms are probably not competitive with other widely used local search heuristics.

3.12 Variable depth search

Variable depth search really describes a type of neighbourhood search strategy which can be used within an algorithm of one of the types described in the previous sections. Although the variable depth search strategy is of particular interest to us because polynomially searchable exponential neighbourhoods (PSEN) can be formed by this strategy, our main focus in this section is where the strategy has been used to form effective neighbourhoods of polynomial size.

The term *variable depth search* was used by Papadimitriou & Steiglitz (1982) to describe local search methods which produce an improving move formed by a sequence of connected changes to the current solution. The length of the sequence of changes is variable. Each change in the sequence is found by a search and the decision as to whether to perform another search is dependent on the potential for further improvement indicated by the previous search.

The first successful algorithm of this type was probably the Kernighan & Lin (1970) algorithm for the uniform graph partitioning problem where a move comprises of a sequence of swap moves. Arguably the most successful and corresponding well known is the Lin-Kernighan (1973) algorithm for the TSP the basic methodology of which we will outline below.

Before describing the whole Lin-Kernighan algorithm we will focus on an underlying property on which the method is based. Specifically, starting with a Hamiltonian path and adding an edge which does not form a tour (and cannot be immediately ejected), there is a unique edge whose ejection will produce a Hamiltonian path.



Figure 3.1: The current Hamiltonian path.

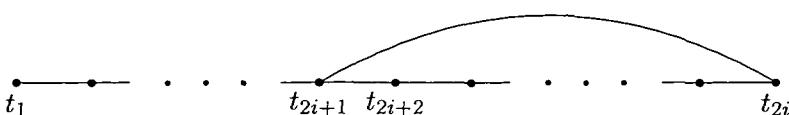


Figure 3.2: The one tree formed.

Let the current Hamiltonian path at the i th step be from city t_1 to t_{2i} as displayed in figure 3.1. Let the edge added be $\{t_{2i}, t_{2i+1}\}$, where $t_{2i+1} \neq t_1$ (in order not to form a tour.) The *one tree* (spanning tree plus one edge) displayed in figure 3.2 is formed. The unique edge whose ejection will form a Hamiltonian path is adjacent

to city t_{2i+1} . By labelling the ejected edge as $\{t_{2i+1}, t_{2i+2}\}$, the Hamiltonian path in figure 3.3 is obtained. This concludes the i th step, leaving the Hamiltonian path displayed in figure 3.3, which is the current Hamiltonian path at the $i + 1$ th step as displayed in figure 3.1.

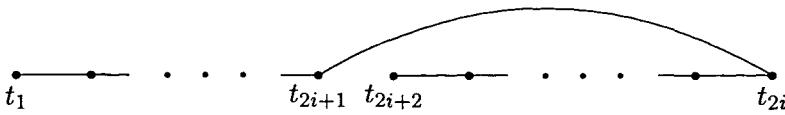


Figure 3.3: The Hamiltonian path produced.

The Lin Kernighan algorithm

begin

L_1 = the length of the initial tour.

Let the length of the shortest tour found $L = L_1$

Select a city t_1 and an edge adjacent to it $\{t_1, t_2\}$ to remove, thus forming the Hamiltonian path displayed in (Figure 3.1 where $i = 1$)

$P_1 = L_1 - d_{t_1, t_2}$ the length of the Hamiltonian path.

The candidate edge to be added $\{t_2, t_3\}$ is the shortest edge involving city t_2 which was not part of the original path

$i = 1$

While there exists a candidate edge to be added **and** $P_i + d_{t_{2i}, t_{2i+1}} < L$ **do**

begin

Add edge $\{t_{2i}, t_{2i+1}\}$ to form the *one tree* displayed in Figure 3.2.

Remove edge t_{2i+1}, t_{2i+2} to form a Hamiltonian path (Figure 3.3)

$P_{i+1} = P_i + d_{t_{2i}, t_{2i+1}} - d_{t_{2i+1}, t_{2i+2}}$

$L_{i+1} = P_{i+1} + d_{t_1, t_{2i+2}}$

$L = \min\{L, L_{i+1}\}$

$i = i + 1$ (Figure 3.1 displays current Hamiltonian path.)

The candidate edge to be added $\{t_{2i}, t_{2i+1}\}$ (if one exists) is the shortest edge involving city t_{2i} which was not part of the original or current path, such that either (a) a tour is formed if the edge is added or (b) the edge to be ejected $\{t_{2i+1}, t_{2i+2}\}$ has not been previously added.

end;

Return the shortest tour found (length L)

end.

The number of steps with the while loop condition satisfied determines the depth of the search and the corresponding length of the sequence of changes.

The exact rules on edges which are allowed to be ejected and added vary in different implementations. Johnson & McGeoch (1997), unlike in the implementation of Lin & Kernighan (1973) given above, allow edges in the original Hamiltonian path which have since been ejected to be added.

In Lin-Kernighan, once a Hamiltonian path has been formed the algorithm iteratively selects a edge to add which causes the ejection of another edge, until some stopping criteria are met. Ejection chains are a generalisation of Lin-Kernighan, where the basic moves for transitioning from one solution to another are compound moves composed of a sequence of paired steps. The first component of each creates a *dislocation* (i.e. an inducement for further change), while the second component creates a change designed to restore the system. In Lin-Kernighan, the first component is the addition of an edge and the second restoring component is the ejection of another edge. In general, a node, edge or path may be the element added or ejected by an ejection chain method.

Numerous papers have been published on ejection chain local search algorithms, both developing new algorithms and showing them to be effective on a range of problems including the TSP, clustering, the vehicle routing problem and the generalised assignment problem (See Glover 1992, Dondorf & Pesch 1994, Glover & Pesch 1995, Rego & Roucairol 1996, Glover & Punnen 1997, Punnen & Glover 1997, Pesch & Glover 1997, Rego 1998a, Rego 1998b, Cavique, Rego & Themido 1999 and Glover, Ibaraki & Yagiura 1999). In particular, the ejection chain algorithm for the TSP produced by Pesch & Glover 1997 appears to be competitive with their implementations of Lin-Kernighan when used in a multi-start framework.

Most ejection chains local search methods are based on edge ejections. However, Rego & Roucairol 1996 use vertex ejection for the vehicle routing problem, and Rego 1998a uses path ejection for the TSP.

Some edge ejection local search methods form moves which are better than an exponential number of other possible moves, whilst only requiring a polynomial search time (Glover 1996, Glover & Punnen 1997, Punnen & Glover 1997, Punnen 1996, Gutin 1999). A discussion of edge ejection algorithms and other algorithms with this characteristic, which we refer to as polynomial searchable exponential neighbourhoods (PSEN), can be found in the next chapter.

Chapter 4

Polynomially Searchable Exponential Neighbourhoods

4.1 Introduction

Although the first polynomially searchable exponential neighbourhood (PSEN) was introduced to the USSR research fraternity as early as 1981, it was not until 1990 that a PSEN was presented in the West.

The known PSENs are almost exclusively for the traveling salesman problem (TSP) and most appear to be unsuitable for other combinatorial optimisation problems. Only a few of the known PSENs have been implemented, and sadly those that have, whilst showing some potential, appear to require more research if they are to be relevant within a practical context. It appears that few researchers within local search are aware of the PSEN and those which are perceive them likely to be only ever of theoretical interest.

One spin off from theoretical research in this area has been the discovery of a lower bound on the number of solutions dominated by some successful construction heuristics. For example, the well known vertex insertion heuristics (Rosenkrantz, Stearns & Lewis II 1977), under any insertion rule, have been show to produce a solution that is better or as good as $\Omega((n - 2)!)$ others, even though they run in only $O(n^2)$ time (Punnen & Kabadi 1999).

In this chapter, we first review some of the results for PSENs in the literature in section 4.2. In section 4.3, we comment on some factors which we feel affect the successful implementation of PSEN. In section 4.4, we introduce our PSEN, dynasearch, in its general form. Finally in section 4.5, we suggest the local search algorithms in which we feel dynasearch may be an effective neighbourhood.

In other chapters, we show how the dynasearch neighbourhood can be used within algorithms for the TSP, and also show how the dynasearch neighbourhood can be used within iterated local search to form state of the art local search algorithms for the single machine total weighted tardiness scheduling problem and the linear ordering problem.

4.2 Literature review

4.2.1 Overview

The first PSEN appear to have originated from polynomially solvable special cases of the TSP. For example, some of the well-solved special cases reviewed by Gilmore, Lawler & Shmoys (1985) in a widely read book on the TSP (Lawler, Lenstra, Rinnooy Kan & Shmoys 1985) have subsequently been used to form PSENs. It should be noted that some of these PSENs had been implemented earlier in the USSR, although researchers in the West only recently became aware of the existence of the papers, which are written in Russian. Neighbourhoods which have been derived from polynomially solvable special cases of the TSP include the pyramidal neighbourhood and the Balas neighbourhood (see subsections 4.2.2 and 4.2.5 respectively for details).

As far as we are aware, all PSENs have been applied to the TSP except for an algorithm by Hurink (1999) for the single machine batching problem. Hurink's algorithm is essentially the same as dynasearch. Although Hurink comments on the similarities between the algorithms, he describes his algorithm as one of finding a shortest path.

Table 4.1 attempts to summarise some of the PSENs for the TSP in the literature. The table lists the polynomial exponential neighbourhoods, giving both the size of the neighbourhood and the complexity of the algorithm which has been proposed to search it.

Where the underlying structure was not originally proposed as a neighbourhood, we reference both the paper that introduced the structure, and the one which first formed a neighbourhood from the structure. For example the data structure known as twisted sequences was first introduced by Aurenhammer (1988) but it was not until 1997 that Deïneko and Woeginger used the structure to form a neighbourhood and derived a dynamic program to search the neighbourhood formed in polynomial

time. As discussed earlier in this overview, other structures were first proposed as polynomially solvable special cases of the TSP before being used as neighbourhoods. Examples include the pyramidal neighbourhood and the Halin graph neighbourhood.

Type of neighbourhood	Complexity of algorithm	Size of neighbourhood	log of size of neighbourhood	Reference
pyramidal	$O(n^2)$	$\Theta(2^n)$	$\Theta(n)$	Klyaus (1976) Sarvanov & Doroshko (1981a)
assign	$O(n^3)$	$\Theta((\frac{n}{2})!)$	$\Theta(n \log n)$	Sarvanov & Doroshko (1981b)
rotational pyramidal	$O(n^3)$	$\Theta(n2^n)$	$\Theta(n)$	Carlier & Villon (1990)
assign-Gutin	$O(n^3)$	$\Omega((\frac{n}{2})!(1+\varepsilon)^{\sqrt{n}})$	$\Theta(n \log n)$	Gutin (1997)
shortest path ejection chains	$O(n^2)$	$\Theta(n2^n)$	$\Theta(n)$	Glover (1996) Punnen & Glover (1997)
ejection chains	$O(n^3)$	$\Omega((n/2)!)!$	$\Theta(n \log n)$	Glover (1996)
Halin	$O(n)$	$\Theta(2^{\frac{n}{4}})$	$\Theta(n)$	Cornuejols, Naddef & Pulleyblank (1983) Glover & Punnen (1997)
single ee *	$O(n)$	$\Theta(6^{\frac{n}{3}})$	$\Theta(n)$	Glover & Punnen (1997)
double ee *	$O(n)$	$\Theta(12^{\frac{n}{3}})$	$\Theta(n)$	Glover & Punnen (1997)
dynasearch 2-opt	$O(n^2)$	unknown, calc in section 9.5.1	$\Theta(n)$	Potts & van de Velde (1995)
dynasearch 3-opt	$O(n^3)$	unknown, calc in section 9.7.2	$\Theta(n)$	Potts & van de Velde (1995)
PQ-tree	$O(n^3)$	$\Theta(2^{n \log \log n})$	$\Theta(n \log \log n)$	Burkard, Dēneko & Woeginger (1996)
matching ee *	$O(n^4)$	$\Theta(n(n/2)!)!$	$\Theta(n \log n)$	Punnen (1996)
twisted sequence	$O(n^7)$	unknown, calc in section 9.6.7	$\Theta(n)$	Aurenhammer (1988) Dēneko & Woeginger (1997)
matching ee *	$O(n^{3+k})$	$\Theta(n^{k-\frac{1}{4}}e^{\sqrt{\frac{n}{2}}}(\frac{n}{2})!)$	$\Theta(n \log n)$	Gutin (1999)
matching ee *	$O(n^{1+\epsilon})$	$\Theta(2^{n \log n})$	$\Theta(n \log n)$	Gutin (1999)
matching ee *	$O(k^5 n)$	$\Omega(k^n)$	$\Theta(n)$	Gutin (1999)
diameter<4			$\Theta(n \log n)$	Gutin & Yeo (1999a)
matching ee *	$O(n^3)$	$\Omega(\frac{n!}{c^n})$ for $c > 3.15$	$\Theta(n \log n)$	Yeo (1997)
Balas	$O(k^2 2^{k-2} n)$	$\Omega((\frac{k-1}{e})^n)$	$\Theta(n)$	Balas (1999) Balas & Simonetti (1998)

Table 4.1: Some of the PSEN for the TSP in the literature (ee= edge ejection *)

The significance of some of the results in table 4.1 stems from the fact that $O(\log(n!)) = O(n \log n)$. So the order of the log of some of the neighbourhood sizes in table 4.1 is equal to the order of the log of the size of the solution space! Gutin (1999) noted that $O(\log((\frac{n}{k})!))$ and $O(\log((n - k)!))$ for a constant k are equivalent to $O(n \log n)$ by Stirling's theorem. Stirling's theorem states for all positive integers m that

$$\sqrt{2\pi m} \left(\frac{m}{e}\right)^m e^{(12m+1)^{-1}} < m! < \sqrt{2\pi m} \left(\frac{m}{e}\right)^m e^{(12m)^{-1}}$$

The following lines give some insight as to how these results can be obtained.

$$\begin{aligned}
 O\left(\log\left(\sqrt{2\pi m}\left(\frac{m}{e}\right)^m e^{(12m+1)^{-1}}\right)\right) &= O\left(\frac{1}{2}\log(2\pi m) + m\log\left(\frac{m}{e}\right) + (12m+1)^{-1}\right) \\
 &= O\left(\frac{1}{2}\log(2\pi) + \frac{1}{2}\log m + m\log m - m\log e\right. \\
 &\quad \left.+ (12m+1)^{-1}\right) \\
 &= O(m\log m)
 \end{aligned}$$

In subsequent subsections, we will describe some of the more practical PSEN in table 4.1 which have been incorporated into local search heuristics.

Construction algorithms with exponential domination number which run in polynomial time.

We now summarise some of the results from the closely linked area of construction algorithms with exponential domination number which run in polynomial time. It should be noted that an impractical large PSEN which can be quickly searched, but whose solutions contain a large number of neighbours in common, may form an effective construction heuristic. This situation is made more likely due to construction heuristics being a slightly less competitive field than local search heuristics.

The domination number of an approximation algorithm, A , for the TSP is the maximum integer k such that for every instance I of the TSP of size n , A produces a tour T which is not worse than at least k tours in I including T itself.

type of heuristic	complexity of algorithm	domination number	log of domination number	reference
matching	$O(n^4)$	$\Omega\left(\frac{n!}{c^n}\right)$ for $c > 1.5$	$\Theta(n \log n)$	Gutin & Yeo (1998a)
greedy expectation	$O(n^3)$	$\Omega((n-2)!)$	$\Theta(n \log n)$	Gutin & Yeo (1998b)
vertex insertion	$O(n^2)$	$\Omega((n-2)!)$	$\Theta(n \log n)$	Rosenkrantz, Stearns & Lewis (1977) Punnen & Kabadi (1999)
Karp	$O(n^3)$	$\Omega((n-2)!)$	$\Theta(n \log n)$	Karp (1979) Punnen & Kabadi (1999)

Table 4.2: Some heuristics with large domination number for the TSP in the literature

Table 4.2 contains some polynomial time construction heuristics for the TSP which are known to have large domination numbers. The greedy expectation algorithm (Gutin & Yeo 1998b) in addition to having a record domination for a polynomial algorithm for the TSP, was also shown to have an exponential domination number for the quadratic assignment problem, whilst still running in polynomial

time. Punnen & Kabadi (1999) used some ideas contained in the paper by Gutin & Yeo (1998b) to show that the well known Karp heuristic and vertex insertion heuristics had exponential domination numbers.

4.2.2 Pyramidal neighbourhoods

The pyramidal neighbourhood, as far as we know, is the first PSEN that was discovered. In the USSR, it was shown that the TSP can be minimised over the set of pyramidal permutations in $O(n^2)$ by Klyaus (1976) and the pyramidal neighbourhood was subsequently introduced by Sarvanov and Doroshko (1981a). The West was about nine years behind! In 1985 Gilmore Lawler and Shmoys discovered Klyaus's result and as far as we are aware it was at this point that the permutations where first referred to as pyramidal. Subsequently, the pyramidal neighbourhood was introduced by Carlier and Villon (1990). Carlier and Villon also introduced the rotational pyramidal neighbourhood in their paper.

A permutation is pyramidal if it first goes through a subset of the cities in order of increasing value of label and then goes through the remaining cities in decreasing value of label. Mathematically, a permutation σ' is pyramidal in relation to the permutation $\sigma = (\sigma(1), \sigma(2), \dots, \sigma(n))$ if σ' is of the form

$$(\sigma(i_1), \sigma(i_2), \dots, \sigma(i_k), \sigma(n), \sigma(j_1), \sigma(j_2), \dots, \sigma(j_{n-k-1})),$$

where $0 \leq k \leq n - 1$, $i_1 < i_2 < \dots < i_k$ and $j_1 > j_2 > \dots > j_{n-k-1}$

The pyramidal neighbourhood consists of all the pyramidal tours for a given permutation representing the tour. Without loss of generality we relabel the cities so that the permutation defining the current tour is $(1, \dots, n)$.

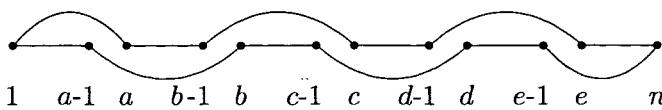


Figure 4.1: A pyramidal neighbourhood move breaking 5 edges.

All moves involve breaking between 2 and $n - 2$ edges, effectively producing a k -opt move, where $2 \leq k \leq n - 2$, according to the number of edges broken. All moves in the pyramidal neighbourhood break edge $1, n$. No moves in the pyramidal neighbourhood break edges $\{1, 2\}$ or $\{n - 1, n\}$. Carlier and Villon introduced the

rotational pyramidal neighbourhood, to remove this characteristic, in an attempt to produce a neighbourhood of more practical interest.

The rotational pyramidal neighbourhood consists of all the pyramidal tours that correspond to any of the n permutations representing the current tour. (Note a permutation, and the permutation obtained by reversing the permutation are counted as a single permutation here as they have been for the pyramidal neighbourhood.)

The pyramidal neighbourhoods have impressive theoretical characteristics. Carlier & Villon (1990) proved that the pyramidal neighbourhood graph has diameter $d_n \leq \log n$ (where the diameter of a graph is defined as the least positive integer d , such that at most d edges need to be traversed to travel between any pair of nodes in the graph), which was the smallest diameter polynomially searchable neighbourhood graph known at the time. This is an impressive result, since most neighbourhood graphs can only be shown to have diameters $d_n < kn$ for a constant k . Gutin and Yeo (1999a) have since shown that the pyramidal neighbourhood graph has diameter $d_n = \Theta(\log n)$, and have introduced a PSEN graph for the TSP which they prove notably to have diameter $d_n \leq 4$. Deineko & Woeginger (1997) prove that the rotational pyramidal neighbourhood contains over 75% of the 3-opt neighbourhood moves.

Practically, Carlier & Villon (1990) found that the pyramidal neighbourhood was ineffective and although they produced results showing the rotational pyramidal neighbourhood can compete with the algorithm of Lin & Kernighan (1973) in terms of solution quality, the effective run times they quote appear unacceptably high at $O(n^3)$. Modern k -opt and Lin-Kernighan implementations with neighbourhood lists have effective run times $O(n)$ (see subsection 7.2.1 for details). The rotational pyramidal neighbourhood is probably only of academic interest unless speedups to improve its effective runtime complexity are found.

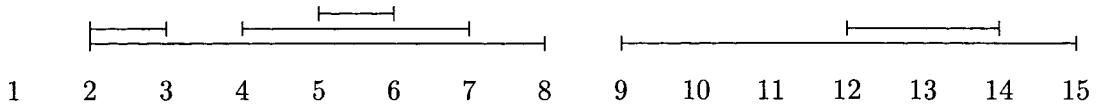
More information about both the pyramidal and rotational pyramidal is given in subsections 9.6.9 and 9.6.10, respectively. In these subsections, we show how the pyramidal and rotational pyramidal neighbourhoods are contained within the twisted sequence neighbourhood.

4.2.3 Twisted sequence neighbourhood

Twisted sequences were introduced by Aurenhammer (1988). However, it was not until 1997 that Deřineko & Woeginger (1997) proposed using the sequences to form a neighbourhood, and they derived a dynamic programming algorithm to search the neighbourhood in $O(n^7)$.

Any solution in the twisted sequence neighbourhood of a given permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$ can be obtained by reversing (twisting) a set of intervals over $(\sigma(1), \sigma(2), \dots, \sigma(n))$ in which a pair of intervals in the set are either disjoint or one is nested within the other (nested in this context means inclusively contained within).

For example, the permutation $(1,8,4,6,5,7,2,3,15,12,13,14,11,10,9)$ lies in the twisted sequence neighbourhood of the permutation $(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)$ as illustrated in the figure below. Note that each line indicates the reversal of the cities below it. It is probably easiest to start by performing the reversals indicated by the highest lines and move down, so the cities 5 and 6 are the first to be reversed, followed by 2,3, then 4,6,5,7, then 12,13,14, and so on.



Deřineko and Woeginger bound the size of the twisted sequence neighbourhood by $\Omega(2^n)$ and $O(6^n)$. In subsection 9.6.7, we calculate the size of the twisted sequence neighbourhood and in subsection 9.6.8 show that 4-opt is a sub-neighbourhood. In the remainder of section 9.6 and in section 9.5 a number of new sub-neighbourhoods are introduced. In particular, the pyramidal neighbourhoods are shown to be sub-neighbourhoods of the twisted sequence.

4.2.4 Edge ejection neighbourhoods

Edge ejection algorithms are a type of ejection chain where, as their name suggests, the elements ejected are edges. Ejection chains are a class of variable depth methods which are discussed in (section 3.12). In the variable depth methods section, there is a description of undoubtedly the most famous edge ejection neighbourhood of Lin and Kernighan (1973).

In this section, we will restrict our discussion to edge ejection algorithms which form PSENs. The first such method was introduced by Glover (1992, 1996), which he refers to as *shortest path edge chains*, and provides a PSEN of size $\Theta(n2^n)$ in linear time. Shortest path edge chains have been implemented within a tabu search framework by Punnen & Glover (1997) who show that the neighbourhood may have some potential in a practical context.

Many other PSEN edge ejection neighbourhoods are yet to be implemented. Glover & Punnen (1997) introduced the *single* and *double* edge ejection methods which they introduce as both methods for solving special cases of the TSP and as methods capable of forming PSEN. Punnen (1996) introduced a large matching edge ejection neighbourhood. Using the results of Punnen (1996), Gutin (1999) formed new matching edge ejection neighbourhoods with some impressive theoretical characteristics. For example, Deĭneko & Woeginger (1997) had showed that if an exponential neighbourhood for the TSP is searchable in time $f(n)$, then its neighbourhood size must be less than $(2f(n)/n)^n$. Their result implies that any neighbourhood searchable in $O(n)$ time must be smaller than $O(2^n)$. Impressively, one of the matching edge ejection algorithms in Gutin (1999) is capable of searching a neighbourhood of size $O(2^{n \log n})$ in $O(n^{1+\epsilon})$ time for every $0 < \epsilon \leq 2$.

4.2.5 Balas's neighbourhood

Balas (1999) introduced this linearly solvable special case of the TSP containing precedence constraints and commented how a dynamic program could be used to form a PSEN. Subsequently, Balas and Simonetti (1998) implemented the Balas neighbourhood.

In Balas's first paper published in 1999, he proposed a linear time dynamic programming algorithm to find the optimal (shortest possible) tour of the following restricted TSP: Given an initial ordering of the n cities and an integer $k > 0$, find a minimum cost tour such that if city i precedes city j by at least k positions in the

initial ordering, then city i precedes city j in any optimal tour. More formally, given an integer $k > 0$, and an ordering $(\sigma = \sigma(1), \dots, \sigma(n))$, an optimal tour, satisfying $\sigma(i) < \sigma(j)$ for $1 \leq i, j \leq n$ such that $i + k \leq j$ can be found in $O(k^2 2^{k-2} n)$ time. When applied as a heuristic to the TSP, the dynamic programming algorithm finds the best tour in a neighbourhood of size $O(((k - 1)/e)^{n-1})$. We refer to this neighbourhood as the Balas neighbourhood.

Recent computational work described in the PhD thesis of Simonetti (1998) and by Balas and Simonetti (1998) demonstrates the potential of the Balas neighbourhood. Simonetti and Balas found that the solution quality obtained by their algorithm was very dependent on the starting tour used, in fact more dependent than is the case for the traditional k -opt and Lin-Kernighan heuristics. Therefore, they suggest using the best tour produced by another local search algorithm as the initial solution for the Balas local search algorithm.

Balas and Simonetti used as a test bed four large problem instances containing between 1304 and 1889 cities. To find good initial solutions to these large problem instances, they performed 10,000 iterations of the state-of-the-art iterated Lin-Kernighan code (Applegate, Bixby, Chvatal and Cook 1999). Even starting with such good initial solutions, the Balas neighbourhood was able to quickly find small improvements. However, to continue to improve the solutions, further iterations of iterated Lin Kernighan are required.

The Balas neighbourhood is most effective on geographic problems where cities tend to cluster in metropolitan areas. If the clustering in a problem is sufficiently pronounced it can be solved by this procedure.

We feel that more research is required to find the best way to use the Balas neighbourhood.

4.3 Some possible issues in the practical application of a PSEN

Although a large number of PSENs are known, few appear to be of practical relevance. The practical relevance of some seems to be immediately ruled out, particularly for use on large instances, due to the prohibitive time complexity of searching them. Others may at first sight appear show great potential, with vast neighbourhoods searchable in reasonably low time complexity. However, as we will discuss in this section, these characteristics may not be enough to guarantee effectiveness in a practical context.

Some characteristics which may indicate a PSEN is unlikely to be of any more than theoretical interest

(a) Neighbouring solutions have a large number of neighbours in common
 Consider the extreme case where a large PSEN has the property that any pair of neighbouring solutions have all bar two neighbours in common (Not counting the current solution as part of its own neighbourhood). Implementing the PSEN from an initial solution S_1 , the best solution S_2 out of an exponential number of neighbours is chosen. We already know that S_2 is a better solution than all of S_1 's other neighbours and S_1 itself, so searching S_2 's PSEN in effect finds out whether a single solution is better than the value of S_2 . This is true for all moves in the PSEN neighbourhood bar the first one. So although the first iteration may be extremely effective, searching the PSEN for all successive iterations is likely to be an extremely inefficient way of comparing the current solution with one other solution.

If in a large PSEN, a neighbouring solution does not contain an exponential number of neighbours that are not contained in the current solution's neighbourhood, the PSEN is unlikely to be an effective neighbourhood but may be an effective construction heuristic. It is obviously a desirable feature of a constructive heuristic to dominate an exponential number of solutions. In fact the well-known construction heuristics for the TSP random/farthest/nearest node insertion heuristics and the algorithm of Karp(1979) all dominate $(n - 2)!$ solutions (Punnen & Kabadi 1999). The node insertion heuristics algorithms require only $O(n^2)$ time and Karp's algorithm requires $O(n^3)$ time. (No TSP algorithm is currently known with a lower time complexity than $O(n^2)$ which is capable of dominating $(n - 2)!$ solutions.)

(b) **The effective complexity of searching the neighbourhood is high.**

The computational complexity is of course a worst-case measure and the effective running times of even poorly implemented neighbourhood search algorithms are generally lower than their computational complexity would suggest. How much lower may be a crucial component in determining a neighbourhood's competitiveness.

To prove their worth, a given PSEN must be able to compete in terms of the quality of solution obtainable in some time frame with the state-of-the-art traditional algorithm. Competitive implementations of traditional neighbourhoods almost always involve at least one type of speedup. For further information, see subsections 5.2.4 and 7.3.5 which describe the speedups used in implementations of our local search algorithm on the total weighted tardiness problem and TSP, respectively. The effect of speedups on the effective running times of the well known 2-opt and 3-opt neighbourhoods for the TSP have been commented on by Johnson & McGeoch (1997). It appears that many competitive implementations of traditional local search for the TSP crucially rely on some form of the following speedups, *Locality searches* using neighbourhood lists (Steiglitz & Weiner 1968) and *don't look bits* (Bentley 1992). A description of each is given in subsection 7.2.1. Locality searches do not effect the move chosen in the neighbourhood whereas the other two have a slightly detrimental effect on the quality of the local minimum found. Neighbourhood lists alone reduce the computational complexity of searching 2-opt and 3-opt neighbourhoods from $O(n^2)$ and $O(n^3)$ respectively to $O(n)$. Their effect on the speed of the local search in practice is very marked, on Euclidean instances. Johnson & McGeoch (1997) observed that the running times (time taken to find a local minimum) were reduced in practice to less than $O(n^{1.2})$, with 3-opt only taking a factor of three times longer than 2-opt! Lin & Kernighan (1973) quoted running times for their Lin-Kernighan heuristic on Euclidean instances of $O(n^{2.2})$, even before the introduction of *don't look bits* by Bentley (1992).

It is in this context of low effective running times which PSENs must compete. It is not enough for a PSEN to compete with poorly implemented traditional algorithms which barely run faster than the order of their worst-case computational complexity, or to imagine that because the computational complexities of two neighbourhoods are the same that their effective running times will be. The rotational pyramidal neighbourhood (Carlier & Villon 1990) is a classic example. Although

the neighbourhood has impressive theoretical characteristics and can compete with Lin-Kernighan (1973) in terms of solution quality, its run times are effectively $O(n^3)$. To be of practical interest speedups would need to be found for the rotational pyramidal neighbourhood which are as effective as those known for Lin-Kernighan. This does not seem likely.

(c) The neighbourhood has a rugged solution landscape

The importance of a neighbourhood's solution landscape has been observed for traditional neighbourhoods. If the solution landscape is not very smooth, it may contain many local optima and make the global optimum more difficult to find.

An example of the importance of landscape may be seen in the total weighted tardiness scheduling problem where even though the complexity of searching the swap neighbourhood may be greater than the insert neighbourhood, the swap neighbourhood may be more effective because of the shape of its landscape.

The size of both the swap and insert neighbourhoods is $O(n^2)$, and for many problems both naively take $O(n^3)$ to search, the extra $O(n)$ due to calculating the effect on each of the jobs scheduled in between the two jobs being swapped or the job's point of removal and insertion, respectively. However, as demonstrated for the linear ordering problem (see sub-section 6.2.2), if the insert neighbourhood is searched in a particular order the complexity of the search can be reduced to $O(n^2)$.

The effect of inserting a job in a schedule will commonly make sections of the schedule much earlier or later, disrupting the schedule and inhibiting insertion moves. Swap moves may have a much less pronounced effect on the jobs not directly involved in the swap move. If both jobs directly involved in the swap have similar processing times, the section of schedule in between the two jobs swapped is not greatly disrupted as a result of the swap.

Although formal research has not been done on importance of the shape of the solution landscape of PSENs, it seems reasonable to conjecture that given the importance of the shape of the solution landscape in helping to determine the effectiveness of traditional neighbourhoods, this is very likely to also be the case for PSENs. The shape of the neighbourhood appears to have been neglected in the study of PSENs with most emphasis being placed on the size of the neighbourhood.

PSEN are probably best utilised as part of a local search algorithm

Implementing PSENs in simple descent algorithms from a random initial solution

may not give an accurate indication of their potential as part of other local search techniques.

The difference in the speed of finding a move in a first and best improve descent are most pronounced when the search is a long distance from a local minimum. Improving moves are quickly found early in a first improve descent, particularly from a random starting solution, when the proportion of improving moves in the neighbourhood is often high. The nearer a first improve descent is to a local minimum, the lower the proportion of improving moves in the neighbourhood and the longer the search takes to find an improving move. A best improve descent normally searches the whole neighbourhood independently of the proportion of improving moves in the neighbourhood. If inequalities are used in an attempt to rule out non-improving moves, without implicitly calculating the effect of each move on the objective function, then the time taken to find the best improving move may actually decrease as the search approaches a local minimum. This effect can be due to the reduction in the number of improving moves, each of which requires implicit calculation.

Given that PSENs find the best move in an exponential neighbourhood, one would expect their characteristics to be more similar to a best improve descent than a first improve descent. Therefore the time taken to search the PSEN when used near a local minimum, may seem more reasonable in comparison with a first improve descent. Comparing the performance of PSEN's with other local search heuristics starting from good solutions, effectively searching neighbourhoods near local minima, may show them in a more flattering light than in a simple descents from random starting solutions. It may therefore be difficult to make a judgement on the effectiveness of a neighbourhood outside of the context in which it is being used.

Many of the more complex local search heuristics search near a local minimum for large proportions of their run time. We feel that many PSENs are probably best utilised within other local search heuristics at times when the search is near local minima or continuously within local search heuristics that predominantly search close to local mimima.

One can never expect the best move to be found in an exponential neighbourhood as fast the first improving move in a traditional neighbourhood. However, the quality of local minima found may make up for the longer search time.

4.4 Dynasearch

4.4.1 The dynasearch neighbourhood

Dynasearch is a PSEN that uses dynamic programming to combine a set of *independent* individual moves from a traditional neighbourhood (e.g. swap, insert, 3-opt, Lin-Kernighan). The set of independent moves form a single dynasearch move that can be performed as a single iteration.

The way in which dynasearch combines traditional neighbourhood moves enables all of the speedups to be used that have been shown to be effective for the underlying neighbourhood. In addition, the neighbourhood formed can be expected to be at least as well behaved as the underlying traditional solution landscape.

The definition of independence must be restrictive enough to ensure that the moves do not interfere with one another, in the sense that the reduction in the objective function produced by the combined move is equal to the sum of the reductions produced by the individual moves. An example of a combined move made up of independent moves in the neighbourhood is given below.

Objective function of solution = k

Improving moves in neighbourhood	A	B	C	D	E	F	G	H
Reductions in objective function	a	b	c	d	e	f	g	h

Sets of independent moves	A, C, D	A, C, E, F	A, H	B, D, E, F, H	G
Corresponding reduction	$a + c + d$	$a + c + e + f$	$a + h$	$b + d + e + f + h$	g

Largest reduction = $a + c + e + f$

Best independent subset of moves = A, C, E, F

Objective function by performing moves A, C, E, F is equal to $k - (a + c + e + f)$

Problems whose solutions can be represented as permutations of elements are most naturally suited to dynasearch. In the simplest implementations (referred to as disjoint k -opt in Chapter 9 but earlier just referred to as dynasearch) moves are only defined as independent if the last element involved in one move comes before the first element in the next. This allows a dynamic programming algorithm to iterate through the ordering recursively finding the best set of independent moves for larger and larger subsets of elements. Eventually, the best set of independent moves over the whole neighbourhood is found. For example, if the permutation

$$(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)$$

represents a solution, a dynasearch move could be represented by

$$1, 2, 3(4, 5, 6)7, 8(9, 10, 11, 12, 13)(14, 15, 16, 17)18, 19, 20.$$

The dynasearch move combines 3 moves in the original neighbourhood, the three moves involve the subsets $\{4, 5, 6\}$, $\{9, 10, 11, 12, 13\}$ and $\{14, 15, 16, 17\}$.

The last move may be allowed to involve elements occurring in the permutation before the first move in some graph problems such as the TSP. This is because there are n permutations representing the same tour in a TSP instance size n . So any one of these permutations could be chosen to represent the current solution. For example, if the permutation

$$(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)$$

represents a solution, a legal move could be represented by

$$3(4, 5, 6)7, 8(9, 10, 11, 12, 13)(14, 15, 16, 17)18(19, 20, 1, 2).$$

The move combines 4 moves in the original neighbourhood the four moves involve the elements $\{4, 5, 6\}$ $\{9, 10, 11, 12, 13\}$ $\{14, 15, 16, 17\}$ and $\{19, 20, 1, 2\}$.

In a permutation problem, where overlapping moves are not allowed, dynasearch can find the best set of independent moves by the following simple general algorithm.

Let $\Delta(j)$ be the reduction given by the best set of independent moves involving the elements currently in the positions 1 to j inclusively, possibly with some constraints on the new ordering (such as element j occupying the last position).

Let $M(i, j)$ be the reduction resulting from a move involving the elements currently in the positions i and j and possibly elements currently in positions between the positions i and j .

Let σ denote the current permutation. It should be possible to compute $M(i, j)$ from the sets $\{\sigma(1), \dots, \sigma(i - 1)\}$, $\{\sigma(j + 1), \dots, \sigma(n)\}$, and the new configuration of elements $\sigma(i), \dots, \sigma(j)$; otherwise dynasearch cannot be applied.

Set $\Delta(0) = 0$ and $\Delta(1) = 0$, all other $\Delta(j)$ can be calculated using the recursive formula below:

$$\Delta(j + 1) = \min_{1 \leq i \leq j+1} \{\Delta(i - 1) + M(i, j + 1)\} \text{ for } j = 1, \dots, n - 1$$

The best reduction achievable by independent moves is $\Delta(n)$ and these moves are found by backtracking.

4.4.2 A basic dynasearch algorithm

In a straightforward implementation of the dynasearch algorithm, we start with an initial permutation $\sigma^{(0)}$ as the current solution. During iteration t , $\sigma^{(t-1)}$ is the current permutation, which we attempt to improve by making a move in the dynasearch neighbourhood. Using the dynamic program of section 4.4, we compute the values $\Delta(\sigma_j^{(t-1)})$ for $j = 1, \dots, n$, and then apply a backtracking procedure to find a corresponding permutation $\sigma^{(t)}$. Therefore, we obtain the set of independent moves from the underlying neighbourhood which achieves the best reduction in objective function value.

The solution defined by $\sigma^{(t)}$ is a local optimum in the dynasearch neighbourhood if $\Delta(\sigma_n^{(t-1)}) = 0$. In this case, the algorithm terminates. On the other hand, if $\Delta(\sigma_n^{(t-1)}) > 0$, then a further iteration with $\sigma^{(t)}$ as the current permutation is executed.

4.5 Traditional local search heuristics that appear to combine well with dynasearch

4.5.1 Iterated local search

‘Kick’ methods were first introduced by Baum (1986a, 1986b), and latter referred to as iterated local search. The method only comes into play once the local optimiser has become trapped in a local minimum. The local optimiser, dynasearch may therefore be implemented as a simple descent algorithm often enabling non-improving moves to be quickly ruled out through the use of inequalities. These speedups are particularly important in choose best algorithms, such as dynasearch, where the whole neighbourhood must be search before a move is performed.

The kick in iterated local search is a random move or group of moves that hopefully allows the search to escape the local minimum without losing many of the good properties of the local minimum’s solution (for more details see section 3.7).

When applying the iterated local search algorithm we introduce a new backtracking procedure. As far as we are aware, backtracking does not appear to have previously been used in iterated local search. In a backtracking step, the current solution is set to be the best solution found thus far. The motivation for backtracking is to ensure that most of the search is performed in ‘interesting’ regions of the solution space. Thus, backtracking can help overcome decisions that direct the

search towards inferior local optima.

The main design features of our iterated local search algorithm with backtracking are illustrated in figure 4.2. Essentially, this type of algorithm is performing a local search on the local optima. Following the generation of an initial current solution, a traditional descent or other local search algorithm is applied to find a solution S which is a local optimum. If S is the first local optimum that is generated, then we define $S_C = S$ to be the current solution. When appropriate, the current best solution found thus far, which we denote by S_B , is updated. A decision is made as to whether to adopt the new local optimum S as the current solution, or to retain S_C .

4.5.2 Guided local search

The guided local search of Voudouris & Tsang (1999) only comes into play once the local optimiser has become trapped in a local minimum. So that as with iterated local search, the dynasearch neighbourhood can be implemented in an efficient descent algorithm.

The guided local search method tries to escape the current local minimum by changing the landscape which is being searched. This is done by penalising all solutions containing a given undesirable feature of the current solution. In this way, the augmented cost of the current solution is increased making neighbouring solutions that do not include the feature comparatively more desirable. The objective function is calculated when a local minimum is reached in the augmented cost function. Note that a local minimum in the augmented cost function landscape may not be a local minimum with respect to the original cost function of the problem (for more details see section 3.8).

We have introduced a backtracking procedure that appears to improve the effectiveness of GLS, at least on short runs. Backtracking does not appear to have previously been used within GLS. In our implementation, the search is periodically restarted from the best solution so far. This solution may no longer be a local minimum in the current augmented objective function landscape so the method descends to a local minimum before penalising any more edges. The modification focuses the local search around the best solution found so far adjusting the landscape in the basin of attraction within which this solution lies.

Within both iterated and guided local search methods dynasearch intensively

searches the region until it can improve the solution no further. The search is then diversified in an attempt to escape the local minimum.

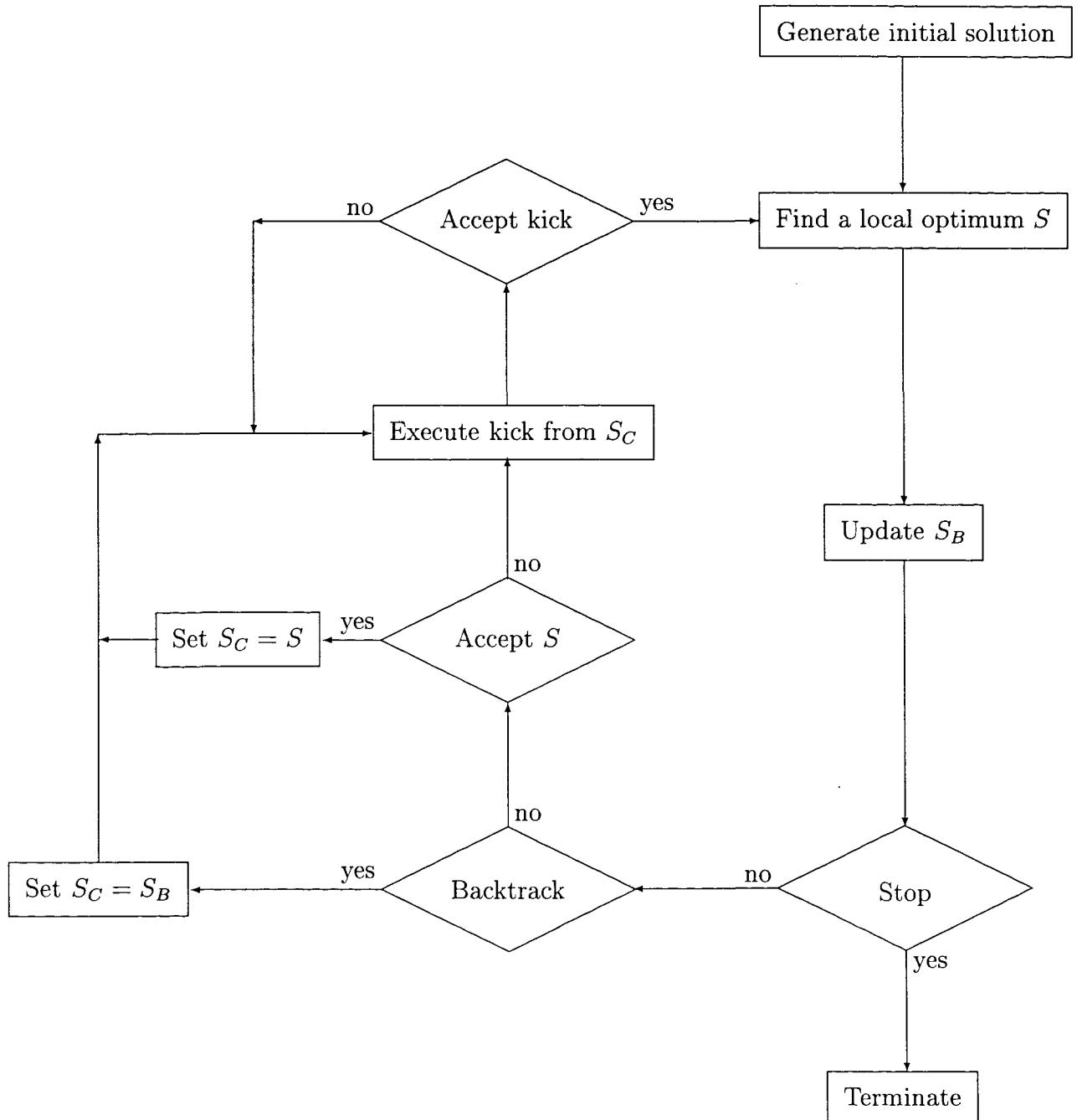


Figure 4.2: Overview of iterated local search with backtracking.

Chapter 5

The Total Weighted Tardiness Problem

5.1 Problem definition and literature review

The single-machine total weighted tardiness scheduling problem can be stated as follows. Each of n jobs (numbered $1, \dots, n$) is to be processed without interruption on a single machine that can handle no more than one job at a time. Job j ($j = 1, \dots, n$) becomes available for processing at time zero, requires processing during an uninterrupted positive processing time p_j , has a positive weight w_j , and has a due date d_j by which time it should ideally be completed. For a given processing order of the jobs, the earliest completion time C_j and the tardiness $T_j = \max\{C_j - d_j, 0\}$ of each job j can readily be computed. The problem is to find a processing order of the jobs with minimum total weighted tardiness $\sum_{j=1}^n w_j T_j$.

This total weighted tardiness problem is not only NP-hard in the strong sense (see Lawler (1977) and Lenstra, Rinnooy Kan and Brucker (1977)), but is also very difficult from a practical point of view: the state-of-the-art branch and bound algorithm of Potts and Van Wassenhove (1985) runs into severe trouble when trying to solve instances with more than 50 jobs to optimality.

In a computational study, Crauwels, Potts and Van Wassenhove (1998) compare the performance of multi-start versions of simulated annealing, threshold accepting, tabu search and genetic algorithms. For each algorithm, one variant uses the ‘natural’ representation of solutions as a permutation of the integers $1, \dots, n$ to specify the processing order of jobs, and another variant uses a binary representation in which jobs are indicated as early or late, from which a decoding heuristic constructs a processing order of the jobs. Their results show that the best-quality solutions are

provided by a multi-start tabu search algorithm with the permutation representation. Moreover, this is the champion local search algorithm, since it is superior to the simulated annealing algorithm of Matsuo, Suh and Sullivan (1987), and to the descent and simulated annealing algorithms of Potts and Van Wassenhove (1991).

5.2 Dynasearch

Potts and Van Wassenhove (1991) and Crauwels, Potts and Van Wassenhove (1998) find that for the natural permutation representation, the swap neighbourhood is preferred to other neighbourhoods for the total weighted tardiness problem. In this section, we present the principles of our dynasearch algorithm. We compare the swap neighbourhood with its dynasearch counterpart (section 5.2.1), present a dynamic programming algorithm to search this neighbourhood (section 5.2.2), give an alternative presentation of the dynamic programming algorithm to search this neighbourhood (section 5.2.3) and finally introduce some speed-ups that reduce the empirical running time of the dynasearch algorithm (section 5.2.4).

5.2.1 Swap and dynasearch swap neighbourhoods

For many combinatorial optimisation problems whose solutions can be represented as sequences, partitions, or assignments, some type of k -exchange neighbourhood structure ($k \geq 2$) is usually adopted, since it is both effective and easy to search. The k -exchange neighbourhood contains all solutions that can be obtained by exchanging k elements in the sequence, partition, or assignment. For a sequencing problem, *swap* is a 2-exchange neighbourhood that interchanges any two elements, irrespective of whether they are adjacent. As an example, the permutation $(3, 2, 1, 4, 5, 6)$ is a swap neighbour of $(1, 2, 3, 4, 5, 6)$, obtained by swapping elements 1 and 3. Verifying local optimality for a k -exchange neighbourhood requires $\Omega(n^k)$ time, where n is the total number of elements. For small values of k , a k -exchange neighbourhood can be searched quickly but, when used in a traditional descent algorithm, the resulting solutions are only of average quality. As k increases, the computational effort required to search the neighbourhood grows quickly, so that selecting larger values of k is often impractical. Therefore, a common choice is $k = 2$, which for sequencing problems corresponds to the swap neighbourhood.

Let $\sigma = (\sigma(1), \dots, \sigma(n))$ be a permutation or sequence that defines the current

processing order of the jobs, where $\sigma(j)$ is the job in position j , for $j = 1, \dots, n$. The swap neighbourhood of a given permutation σ comprises all sequences that can be obtained by interchanging any two jobs $\sigma(i)$ and $\sigma(j)$, where $1 \leq i < j \leq n$. The size of the swap neighbourhood is $n(n - 1)/2$.

The *dynasearch swap* neighbourhood of σ allows a new permutation to be obtained by a series of swaps. To specify which swaps are allowed, we need a definition. The two moves that swap job $\sigma(i)$ with job $\sigma(j)$, and job $\sigma(k)$ with job $\sigma(l)$, respectively, are said to be *independent* if $\max\{i, j\} < \min\{k, l\}$ or $\min\{i, j\} > \max\{k, l\}$. The dynasearch swap neighbourhood consists of all solutions that can be obtained from σ by a series of pairwise independent swap moves.

The dynasearch swap neighbourhood has size $2^{n-1} - 1$, as outlined below. Every dynasearch swap move can be defined by a distinct binary string of length n . The value of each bit defines the involvement of the job in the corresponding position of the sequence, a 1 or a 0 indicating that a job is involved or is not involved in a swap, respectively. Given that there are $2t$ 1's in the binary string, the $2s - 1$ th and $2s$ th 1 in the binary string are swapped for $s = 1, 2, \dots, t$. Clearly there must be an even number of 1's in the sequence, so given the values of the bits in the first $n - 1$ positions the value in the n th position is determined. Given that all of the bits in the string are not 0 which would signify not performing any move, any of the bits in the first $n - 1$ positions can be either a 0 or a 1. Therefore $2^{n-1} - 1$ different strings can be formed corresponding to $2^{n-1} - 1$ different neighbourhood moves.

Dynasearch uses a best-improve strategy: a dynasearch swap move is equivalent to a *best* series of independent swaps. Consequently, if a solution is a local optimum with respect to the dynasearch swap neighbourhood, then it is also a local optimum with respect to the swap neighbourhood, and vice versa. We conjecture that the dynasearch swap neighbourhood produces larger basins of attraction for good local minimum than the swap neighbourhood.

We now present an example to illustrate the difference between the swap and dynasearch swap neighbourhoods.

Example. Consider the 6-job instance that is specified in Table 6.1. Suppose the initial sequence is $(1, 2, 3, 4, 5, 6)$.

Job j	1	2	3	4	5	6
Processing time p_j	3	1	1	5	1	5
Weight w_j	3	5	1	1	4	4
Due date d_j	1	5	3	1	3	1

Table 5.1: Data for the problem instance.

Table 5.2 shows the moves that are made by a best-improve descent algorithm with the swap neighbourhood. With this traditional approach, we observe that the search becomes trapped in a local minimum with a total weighted tardiness of 70.

Iteration	Current sequence	Total weighted tardiness
	1 2 3 4 5 6	109
1	1 2 3 5 4 6	90
2	1 2 3 5 6 4	75
3	5 2 3 1 6 4	70

Table 5.2: Swaps made by best-improve descent.

Iteration	Current sequence	Total weighted tardiness
	1 2 3 4 5 6	109
1	1 3 2 5 4 6	89
2	1 5 2 3 6 4	68
3	5 1 2 3 6 4	67

Table 5.3: Dynasearch swaps.

Table 5.3 shows the ability of dynasearch to make multiple independent swaps to move to the next solution. The example may suggest that, by definition, dynasearch necessarily yields a better result than best-improve (or first-improve) descent, for any instance. This is not the case: there may be instances for which traditional descent is better. Our claim, which is substantiated by the computational results in section 5.4.2, is that dynasearch is better on average.

5.2.2 Finding the best set of independent swaps

To find the best set of independent swaps that can be obtained from a permutation σ , we employ a dynamic programming algorithm. This algorithm uses a forward enumeration scheme in which jobs are appended to the end of the current partial sequence and are possibly swapped. We define a partial sequence to be in state (j, σ) , for $j = 0, 1, \dots, n$, if it can be obtained from the partial sequence $(\sigma(1), \dots, \sigma(j))$ by applying a series of independent swaps. Of course, to find the best possible sequence in the dynasearch swap neighbourhood of σ , which by definition must be in state (n, σ) , we only need to consider a sequence that has minimum objective value among all sequences in this state.

Let σ_j be a partial sequence with minimum total weighted tardiness for jobs $\sigma(1), \dots, \sigma(j)$ among partial sequences in state (j, σ) . Further, let $F(\sigma_j)$ be the total weighted tardiness for jobs $\sigma(1), \dots, \sigma(j)$ in σ_j . This partial sequence must be obtained from a partial sequence σ_i that has minimum objective value from all partial sequences in some previous state (i, σ) , where $0 \leq i < j$, by appending job $\sigma(j)$ if $i = j - 1$, or by first appending jobs $\sigma(i + 1), \dots, \sigma(j)$ and then interchanging jobs $\sigma(i + 1)$ and $\sigma(j)$ if $0 \leq i < j - 1$. These two possibilities are considered in detail below.

- $i = j - 1$. In this case, job $\sigma(j)$ is not involved in any swap, and $\sigma(j)$ is simply appended to a partial sequence σ_{j-1} ; hence, $\sigma_j = (\sigma_{j-1}, \sigma(j))$. Accordingly,

$$F(\sigma_j) = F(\sigma_{j-1}) + w_{\sigma(j)}(P_{\sigma(j)} - d_{\sigma(j)})^+,$$

where $(x)^+ = \max\{x, 0\}$ for any real x , and $P_{\sigma(j)} = \sum_{i=1}^j p_{\sigma(i)}$.

- $0 \leq i < j - 1$. Here, jobs $\sigma(j)$ and $\sigma(i + 1)$ are swapped, so that σ_j can be written as $\sigma_j = (\sigma_i, \sigma(j), \sigma(i+2), \dots, \sigma(j-1), \sigma(i+1))$, and the total weighted tardiness $F(\sigma_j)$ is readily computed as

$$\begin{aligned} F(\sigma_j) &= F(\sigma_i) + w_{\sigma(j)}(P_{\sigma(i)} + p_{\sigma(j)} - d_{\sigma(j)})^+ \\ &\quad + \sum_{k=i+2}^{j-1} w_{\sigma(k)}(P_{\sigma(k)} + p_{\sigma(j)} - p_{\sigma(i+1)} - d_{\sigma(k)})^+ \\ &\quad + w_{\sigma(i+1)}(P_{\sigma(j)} - d_{\sigma(i+1)})^+. \end{aligned}$$

We are now ready to present our dynamic programming algorithm. The initialization is

$$\begin{aligned} F(\sigma_0) &= 0, \\ F(\sigma_1) &= w_{\sigma(1)}(p_{\sigma(1)} - d_{\sigma(1)})^+, \end{aligned}$$

and the recursion for $j = 2, \dots, n$ is

$$F(\sigma_j) = \min \left\{ \begin{array}{l} F(\sigma_{j-1}) + w_{\sigma(j)}(P_{\sigma(j)} - d_{\sigma(j)})^+, \\ \min_{0 \leq i \leq j-2} \left\{ F(\sigma_i) + w_{\sigma(j)}(P_{\sigma(i)} + p_{\sigma(j)} - d_{\sigma(j)})^+ \right. \\ \quad \left. + \sum_{k=i+2}^{j-1} w_{\sigma(k)}(P_{\sigma(k)} + p_{\sigma(j)} - p_{\sigma(i+1)} - d_{\sigma(k)})^+ \right. \\ \quad \left. + w_{\sigma(i+1)}(P_{\sigma(j)} - d_{\sigma(i+1)})^+ \right\}. \end{array} \right.$$

The optimal solution value is then equal to $F(\sigma_n)$, and the corresponding sequence can be found by backtracking.

This dynamic programming algorithm runs in $O(n^3)$ time and requires $O(n)$ space. Hence, dynasearch requires $O(n^3)$ time to verify local optimality, which is the same time requirement as for a traditional descent algorithm.

Finally, we note that a backward dynamic programming algorithm can be derived. However, it does not offer any computational advantages, since the backward algorithm has the same time and space requirements as the forward scheme.

5.2.3 An alternative presentation of the dynamic program

Let $\delta(i+1, j)$ be the reduction in total weighted tardiness of the sequence due to swapping the job in position $i+1$ with the job in position j . Then

$$\begin{aligned} \delta(i+1, j) &= (C_{\sigma(j)} - C_{\sigma(i)}) - (w_{\sigma(j)}(P_{\sigma(i)} + p_{\sigma(j)} - d_{\sigma(j)})^+ \\ &\quad + \sum_{k=i+2}^{j-1} w_{\sigma(k)}(P_{\sigma(k)} + p_{\sigma(j)} - p_{\sigma(i+1)} - d_{\sigma(k)})^+ \\ &\quad \quad \quad + w_{\sigma(i+1)}(P_{\sigma(j)} - d_{\sigma(i+1)})^+), \end{aligned}$$

$$\text{where } C_{\sigma(j)} = \sum_{i=1}^j w_{\sigma(j)}(P_{\sigma(j)} - d_{\sigma(j)})^+.$$

The dynamic program

Let σ_j be a partial sequence with minimum weighted tardiness for the jobs $\sigma(1), \dots, \sigma(j)$ among partial sequences in state (j, σ) . Further, let $\Delta(\sigma_j)$ be the reduction in the weighted tardiness due to the independent swap moves performed on the jobs $\sigma(1), \dots, \sigma(j)$ to form σ_j .

Initialisation

$$\Delta(\sigma_i) = 0 \text{ for } i = 0, 1$$

Recursion

Calculate $\Delta(\sigma_j)$ for $j = 2, \dots, n$:

$$\Delta(\sigma_j) = \max \left\{ \begin{array}{l} \Delta(\sigma_{j-1}) \\ \max_{1 \leq i \leq j-1} \{\Delta(\sigma_i) + \delta(i+1, j)\} \end{array} \right\}.$$

$\Delta(\sigma_n)$ is the largest improvement obtainable from a move in the neighbourhood. The move corresponding to the improvement $\Delta(\sigma_n)$ is found by backtracking.

Note how this presentation highlights the simplicity of the dynamic program and shows how most of the work involved in searching the neighbourhood is contained in calculating the effect of moves in the underlying swap neighbourhood.

5.2.4 Speed-ups

There are several tricks that can be employed to reduce the computation time for traditional descent and dynasearch algorithms. We refer to a swap move that strictly reduces the total weighted tardiness as *improving*; otherwise, it is *non-improving*. Speed-ups that reduce the computation time for determining whether a move is non-improving, or is dominated by some other move, are useful for both descent and dynasearch. (However, they would not be useful for a simulated annealing algorithm.) Another category of speed-ups applies to the dynamic programming algorithm that is used in dynasearch. These two categories of speed-ups are discussed in the following subsections.

Speed-ups for total weighted tardiness comparisons

Before providing details of our speed-ups, we first present some preprocessing that allows these speed-ups to be implemented efficiently. As indicated in section 5.2.2, we compute for the current sequence σ the partial sums of processing times

$$P_{\sigma(j)} = \sum_{i=1}^j p_{\sigma(i)},$$

for $j = 1, \dots, n$. Additionally, we compute the partial sums of weighted tardiness values

$$V_{\sigma(j)} = \sum_{i=1}^j w_{\sigma(i)}(P_{\sigma(i)} - d_{\sigma(i)})^+,$$

and the partial sums of weights for late jobs

$$W_{\sigma(j)} = \sum_{i=1}^j w_{\sigma(i)}U_{\sigma(i)},$$

for $j = 1, \dots, n$, where

$$U_{\sigma(i)} = \begin{cases} 1 & \text{if } P_{\sigma(i)} > d_{\sigma(i)}, \\ 0 & \text{otherwise.} \end{cases}$$

We now describe some tests which, if successful, indicate that interchanging jobs $\sigma(i+1)$ and $\sigma(j)$, where $0 \leq i \leq j-2$, of the current permutation σ , cannot reduce the total weighted tardiness by more than some specified value Δ . In descent, such a test is used to reject a potential move that swaps jobs $\sigma(i+1)$ and $\sigma(j)$. Specifically, for first-improve descent, we set $\Delta = 0$, so that non-improving moves are rejected. For best-improve descent, we set $\Delta = 0$ at the start of the neighbourhood search for σ , and then update Δ as appropriate so that Δ is the best improvement found thus far. In this case, the test rejects non-improving moves, and moves that cannot achieve an improvement that exceeds Δ . For dynasearch, the computation of $F(\sigma_j)$ by dynamic programming uses minimization to compare candidate values that are obtained for different values of i . From the recursion, an initial candidate value is $\hat{F}(\sigma_j) = F(\sigma_{j-1}) + w_{\sigma(j)}(P_{\sigma(j)} - d_{\sigma(j)})^+$. The candidate value for i , where $0 \leq i \leq j-2$, is equal to $F(\sigma_i)$ plus the total weighted tardiness for jobs $\sigma(i+1), \dots, \sigma(j)$ that results from swapping jobs $\sigma(i+1)$ and $\sigma(j)$. We set $\Delta = F(\sigma_i) + (V_{\sigma(j)} - V_{\sigma(i)}) - \hat{F}(\sigma_j)$, where $\hat{F}(\sigma_j)$ is the best candidate value found thus far. Thus, the test rejects the candidate value corresponding to i when it cannot provide a better candidate value than $\hat{F}(\sigma_j)$.

Under the most straightforward implementation, to evaluate the interchange of the jobs in positions $i + 1$ and j requires the total weighted tardiness of $j - i$ jobs in positions $i + 1, \dots, j$ to be computed. We describe below a pre-testing procedure that, although avoiding the computation of $j - i$ weighted tardiness values, may indicate that the swap move cannot improve the total weighted tardiness by more than Δ . For cases in which the result of this pre-testing is inconclusive, we present a method of reducing the computational effort in evaluating the total weighted tardiness of the relevant $j - i$ jobs.

We now present our pre-testing procedure which, if successful, guarantees that interchanging jobs $\sigma(i + 1)$ and $\sigma(j)$ cannot reduce the total weighted tardiness by more than Δ . First, suppose that $p_{\sigma(j)} \geq p_{\sigma(i+1)}$. If the combined weighted tardiness of jobs $\sigma(i + 1)$ and $\sigma(j)$ increases, or decreases by an amount that does not exceed Δ as a result of the swap, then the required improvement is not achieved. Alternatively, suppose that $p_{\sigma(j)} \leq p_{\sigma(i+1)}$. An upper bound on the reduction in the total weighted tardiness of jobs $\sigma(i + 2), \dots, \sigma(j - 1)$ is given by $\min\{V_{\sigma(j-1)} - V_{\sigma(i+1)}, (p_{\sigma(i+1)} - p_{\sigma(j)})(W_{\sigma(j-1)} - W_{\sigma(i+1)})\}$. Therefore, if the combined weighted tardiness of jobs $\sigma(i + 1)$ and $\sigma(j)$ increases by an amount that exceeds or is equal to this upper bound minus Δ as a result of the swap, then the required improvement is not achieved.

We now describe our method of computing the total weighted tardiness of jobs $\sigma(i + 1), \dots, \sigma(j)$. At the start of each iteration, we partition the current sequence σ into runs of non-late and late jobs, and then treat each run as a group. Our motivation is that if $p_{\sigma(j)} \leq p_{\sigma(i+1)}$, then all the jobs in positions $i + 2, \dots, j - 1$ are completed at the same time or earlier after swapping jobs $\sigma(i + 1)$ and $\sigma(j)$. Hence, there is no delay to any run of non-late jobs within positions $i + 2, \dots, j - 1$, so that their contribution to the total weighted tardiness remains zero. Similarly, if $p_{\sigma(j)} \geq p_{\sigma(i+1)}$, then the total weighted tardiness of any run of late jobs within positions $i + 2, \dots, j - 1$ increase by $(p_{\sigma(j)} - p_{\sigma(i+1)})W$ as a result of the swap, where W is the total weight of jobs in the run.

To use the above speed-ups for runs of non-late and late jobs, we associate a string with the current sequence of jobs, where the string is constructed as follows. If the first run of jobs is late, then the string starts with a 1; otherwise, it starts with a 0. Whenever a new run starts, the position of the first job in the run is added to the string. The final number in the string is $n + 1$. For example, for an instance

with 40 jobs, the string 0 2 7 40 41 means the job in position 1 is non-late, the jobs in positions 2 to 6 are late, the jobs in positions 7 to 39 are non-late, and the job in the last position is late.

Speed-ups for the dynamic program in dynasearch

Consider the computation of $F(\sigma_j^{(t-1)})$ for $j = 1, \dots, n$. Let h_t denote the largest index such that

$$\sigma^{(t-1)}(j) = \sigma^{(t-2)}(j) \quad \text{for } j = 1, \dots, h_t.$$

Then we must have that

$$F(\sigma_j^{(t-1)}) = F(\sigma_j^{(t-2)}) \quad \text{for } j = 1, \dots, h_t.$$

Accordingly, in iteration t of the dynasearch algorithm, we need to perform the recursion only for $j = h_t + 1, \dots, n$.

5.3 Implementation of iterated dynasearch

In this section, we present our iterated dynasearch algorithm for the total weighted tardiness problem. A general introduction to iterated local search is given in section 3.7. Our description refers to the notation in section 4.5.1 and in particular figure 4.2.

To fully evaluate the effectiveness of the dynasearch concept, we compare the empirical performance of iterated dynasearch not only with the performance of the state-of-the-art multi-start tabu search algorithm of Crauwels, Potts and Van Wassenhove (1998), but also with iterated first-improve and iterated best-improve descent.

In our implementation of these three iterated algorithms, a kick is a series of random swap moves (that in general are not independent) from a previously generated local minimum. Even though the same underlying neighbourhood is used for the kicks, for a sufficiently long series of moves, the random swaps invariably produce a solution that is outside the region of attraction of the local minimum. Moreover, the local minimum that is obtained as a result of the kick is often different from the one from which the kick is performed. We use the parameter α to denote the number of random swap moves in a kick. Based on the results of initial experiments, we use the parameter value $\alpha = 6$.

Our algorithms backtrack to the best solution found thus far every β iterations, where an iteration refers to finding a local optimum following a kick, and β is a

parameter. For iterations in which backtracking does not occur, we always accept the new solution S to be the current solution. Our initial experiments indicate that $\beta = 5$ is a suitable value.

The performance of the algorithms is not particularly sensitive to small changes in the values of parameters α and β , and in some of the other design features. For instance, by varying α , we observe that using a kick with six, rather than five or seven random swap moves, produces only slightly better results. Moreover, for kicks based on random 3-exchanges, a single random 3-exchange is not very effective, but a kick comprising two random 3-exchanges is a reasonable alternative to six random swaps.

We have not yet exploited a known property of the total weighted tardiness problem: there exists an optimal solution in which non-late jobs are sequenced in non-decreasing order of due dates, or EDD order. In an attempt to reduce the computation time for dynasearch descents, we transpose adjacent non-late jobs if they are not in EDD order, during a number of iterations at the beginning of the algorithm. Then, for the remaining iterations, we diversify the search by transposing non-late jobs so that they no longer appear in EDD order. Specifically, for the first 100 iterations, a series of transposes of adjacent non-late jobs that are not in EDD order is performed, starting at the beginning of the sequence. After the first 100 iterations, a similar procedure transposes adjacent non-late jobs, either if they are not currently in EDD order, or with probability $1/3$ if they are currently in EDD order and their interchange would not cause either one to become late. All transposes are performed immediately prior to the kick.

Finally, we discuss the heuristic that is used to produce a starting solution for the iterated local search algorithms. For non-iterated versions of descent or dynasearch in which a single local minimum is to be generated, then a better starting solution tends to lead to a local minimum that is at least as good. However, when multiple local minima are generated, as in iterated local search algorithms, the quality of the starting solution has less effect on the quality of the final solution.

In our implementation, all searches start from the heuristic solution that is generated by the so-called Apparent Urgency (AU) rule. The AU rule is a constructive heuristic that is presented by Morton, Rachamadugu and Vepsalainen (1984). It is selected for two reasons: first, it is used by Crauwels, Potts and Van Wassenhove (1998) so that using the AU rule makes comparisons easier; second, Potts and Van

Wassenhove (1991) and Morton and Pentico (1993) find that the AU rule compares favorably with other constructive heuristics for the total weighted tardiness problem. The AU rule is a dynamic list scheduling heuristic that selects an unscheduled job j with the smallest AU_j value to occupy the first unfilled position of the sequence, where AU_j is defined by

$$AU_j = \frac{w_j}{p_j} \exp\left(-\frac{\max\{0, d_j - t - p_j\}}{k\bar{p}}\right).$$

In this expression, t is the sum of the processing times of the scheduled jobs, \bar{p} is the average processing time of the jobs, and k is the so-called lookahead parameter, which is preset according to the tightness of the due dates. In our implementation, we follow Potts and Van Wassenhove (1991) by using $k = 0.5$ for $TF = 0.2$, $k = 0.9$ for $TF = 0.4$, and $k = 2.0$ for $TF > 0.4$, where $TF = 1 - \sum_{j=1}^n d_j / (n \sum_{j=1}^n p_j)$ is a parameter of the problem instance that is known as the tardiness factor (see section 5.4.1 for the values of TF that are used in our test instances).

5.4 Computational experience

This section reports on our computational experience with randomly generated instances to evaluate and compare the empirical performance of the various local search algorithms. In section 5.4.1, we discuss the design of the computational experiments, including the random instance generator. Section 5.4.2, compares the performance of multi-start first-improve descent, multi-start best-improve descent, and multi-start dynasearch, as well as iterated versions of these algorithms. Finally, in section 5.4.3, we compare iterated dynasearch with the multi-start tabu search algorithm of Crauwels, Potts and Van Wassenhove (1998).

5.4.1 Experimental design

We use exactly the same set of randomly generated problem instances as Crauwels, Potts and Van Wassenhove (1998), who adopted the generation scheme proposed by Potts and Van Wassenhove (1985). Instances with $n = 40$, $n = 50$ and $n = 100$ were randomly generated, and for each job j ($j = 1, \dots, n$), an integer processing time p_j was generated from the uniform distribution $[1, 100]$ and an integer processing weight w_j was generated from the uniform distribution $[1, 10]$. Different instance classes of different ‘hardness’ were generated by using different uniform distributions for generating the due dates. For a given *relative range of due*

dates RDD (RDD = 0.2, 0.4, 0.6, 0.8, 1.0) and a given *average tardiness factor* TF (TF = 0.2, 0.4, 0.6, 0.8, 1.0), an integer due date d_j was randomly generated from the uniform distribution $[P(1-TF-RDD/2), P(1-TF+RDD/2)]$, where $P = \sum_{j=1}^n p_j$. Five instances were generated for each of the 25 pairs of values of RDD and TF, yielding 125 instances for each value of n . These instances are available electronically at the OR library that is run by Beasley (1990).

In their study, Crauwels, Potts and Van Wassenhove (1998) attempt to solve the instances with $n = 40$ and $n = 50$ by applying the branch and bound algorithm of Potts and Van Wassenhove (1985). The algorithm successfully solves 124 and 103 problems out of 125 for the 40- and 50-job instances, respectively, on an HP 9000-G50 computer within a time limit of two minutes for each instance. There were no attempts to solve the 100-job instances since prohibitively large computation times were anticipated. Where the optimal solution value is unknown (which is the case for all of the 100-job instances), the best known solution value is used as though it was the optimal.

All of our descent and dynasearch algorithms were coded in C and run on a SPARC 5/110 server station. However, Crauwels, Potts and Van Wassenhove (1998) ran their algorithms, which were coded in C, on a HP 9000-G50 computer. The information collected by the Standard Performance Evaluation Corporation (1992) indicates that the HP 9000-G50 is a factor 1.276 faster than the SPARC 5/110, and accordingly we use this conversion factor to make a direct comparison of computation times. As with any comparison between algorithms that are run on different machines, conversion factors for computation times only give a rough guide. Consequently, any conclusions that use the computation times obtained with our conversion factor of 1.276 are only valid if performance differences are sufficiently large.

We compare the performance of the various local search algorithm on basis of the following statistics:

ARPD = the average relative percentage deviation of the solution value found by the local search algorithm from the optimal (or best known used in) solution value;

MRPD = the maximum relative percentage deviation of the solution value found by the local search algorithm from the optimal (or best known, used in Crauwels, Potts and Van Wassenhove (1998)) solution value;

NO = the number of optimal (or best known, used in Crauwels, Potts and Van Wassenhove (1998)) solution values found out of 125;

ACT:SPARC = the average computation time in seconds on a SPARC 5/110 server station;

ACT:HP = the average computation time in seconds on a HP 9000-G50 computer;

NI = the number of iterations performed in multi-start and iterated descent and dynasearch, where an iteration refers to descending to a local minimum, and the number of moves performed for each start in the multi-start tabu search algorithm of Crauwels, Potts and Van Wassenhove (1998).

During our experimental work, we obtained better solutions for some 100-job instances than those generated by Crauwels, Potts and Van Wassenhove (1998). Nevertheless, to facilitate a direct comparison of results, the best known solution values of Crauwels, Potts and Van Wassenhove are used when computing the above statistics. When we obtain a solution value that is better than the previous best known value, it is then treated as if the optimum of Crauwels, Potts and Van Wassenhove has been found, except that its relative percentage deviation is not zero but negative—so, such a solution actually reduces the average relative percentage deviation statistics for the local search algorithm. A further consequence of obtaining new best known solution values is that the numbers of optimal or best known solution values, as listed by Crauwels, Potts and Van Wassenhove, are sometimes higher than would be the case if they were recalculated with respect to the new values.

5.4.2 Multi-start and iterated dynasearch vs. first-improve and best-improve descent

This subsection compares the empirical performance of multi-start versions and iterated versions of first-improve descent, best improve descent and dynasearch. All results are obtained with the parameter values $\alpha = 5$ and $\beta = 5$ (see section 5.3 for the definition of α and β), and without the transpose of adjacent jobs according the EDD criteria, as described in section 5.3. Note that the value of $\alpha = 5$ gives good results for iterated dynasearch and iterated descent, although the best choice for iterated dynasearch was found to be $\alpha = 6$ as used in section 5.4.3. Also, identical speed-ups (see section 5.2.4) are applied in each algorithm, but they appear to be most effective in reducing the computation time of multi-start first-improve descent. All multi-start algorithms are run for 2, 4 and 20 seconds for the 40-, 50- and 100-job instances, respectively, and for the iterated algorithms these times are halved. In each case, the results are obtained from the average of 10 independent runs.

Table 5.4 gives our computational results for multi-start versions of first-improve descent, best-improve descent, and dynasearch. Although multi-start dynasearch provides the best quality solutions, its superiority over multi-start best-improve descent is not substantial. Each descent in first-improve executes on average only about 6, 7 and 10 moves for the 40-, 50- and 100-job instances, respectively. By contrast, corresponding numbers of best-improve descent moves are 32, 40 and 86, and corresponding numbers of dynasearch moves are 15, 18 and 31, respectively. Note that a dynasearch move may correspond to several descent moves, so the numbers of moves for dynasearch are not directly comparable to those for descent. The small number of moves per descent for first-improve indicates that the search often becomes trapped in a local minimum fairly quickly, which explains the relatively poor performance of this algorithm. However, for best-improve descent and dynasearch, the numbers of moves tends to be sufficiently large to allow the search to find better quality local minima.

n	ACT: SPARC	Multi-start first-improve				Multi-start best-improve				Multi-start dynasearch			
		NI	ARPD	MRPD	NO	NI	ARPD	MRPD	NO	NI	ARPD	MRPD	NO
40	2	117	0.245	7.756	77.8	33	0.165	6.617	99.6	70	0.087	5.289	116.9
50	4	133	0.503	11.197	65.8	32	0.270	6.750	83.1	74	0.162	5.394	95.1
100	20	87	0.573	14.640	33.4	13	0.437	20.070	44.6	39	0.360	20.070	47.3

Table 5.4: Computational results for multi-start local search algorithms.

For iterated versions of the algorithms in which the restarts are not from random

starting sequences but are obtained from kicks, the results are shown in Table 6.10. These results exhibit different characteristics from those in Table 5.4. Specifically, iterated dynasearch can be seen to outperform iterated first-improve and iterated best-improve descent on all performance measures: it achieves significantly lower ARPD and MRPD values, and it finds considerably more optimal (or best) solutions than the two iterated descent algorithms.

n	ACT: SPARC	Iterated first-improve				Iterated best-improve				Iterated dynasearch			
		NI	ARPD	MRPD	NO	NI	ARPD	MRPD	NO	NI	ARPD	MRPD	NO
40	1	109	0.080	2.350	95.0	115	0.090	6.018	108.3	119	0.049	5.338	121.2
50	2	111	0.386	10.686	74.2	140	0.181	6.800	88.9	143	0.044	2.776	111.5
100	10	78	0.447	15.851	36.0	123	0.291	18.391	47.1	122	0.041	1.834	82.5

Table 5.5: Computational results for iterated local search algorithms.

For all iterated algorithms and all instance sizes, the average numbers of moves per descent are between 5 and 8, which is slightly more than the number of random moves in the kick ($\alpha = 5$). The average numbers of moves per descent are less for best-improve than for dynasearch, which suggests that dynasearch allows more searching before becoming trapped in a local minimum, and is consistent with the better quality solutions found by dynasearch. Although the average number of moves per descent for first-improve is more than for best-improve and dynasearch for the 100-job instances, the higher quantity of moves cannot compensate for their lower quality.

One reason for the impressive performance of iterated dynasearch is its apparent ability to move between local minima. The dynasearch swap neighbourhood is much larger than the traditional swap neighbourhood, containing different basins of attraction around the same set of local minima. For the instances with 100 jobs, iterated dynasearch detects about six times more local minima with distinct objective values than iterated first-improve or iterated best-improve descent.

5.4.3 Iterated dynasearch vs. tabu search

In this subsection, we report on our computational results that compare the empirical performance of iterated dynasearch with that of the state-of-the-art local search algorithm, namely the multi-start tabu search method of Crauwels, Potts and Van Wassenhove (1998). All information regarding the performance of the tabu search algorithm with five starts is taken directly from the paper of Crauwels, Potts and Van Wassenhove. Table 5.6 provides results for the two algorithms. The iterated

dynasearch algorithm is run with our recommended parameter values $\alpha = 6$ and $\beta = 5$, and each of the results is obtained from the average of 10 independent runs. To allow a fair comparison with tabu search using a similar computation time, iterated dynasearch is run for different numbers of iterations. The rows of the table are aligned according to the average computation time values. Note that our computation times are converted to equivalent times on the HP 9000-G50 computer by multiplying by a factor 1.276 to account for our slower computer.

	Iterated dynasearch					Multi-start tabu search					
n	NI	ARPD	MRPD	NO	ACT:HP	NI	ARPD	MRPD	NO	ACT:HP	
40	50	0.0150	1.8594	123.7	0.20		$2n^2/5$	0.00	0.33	118	1.32
	100	0.0001	0.0166	124.8	0.40						
	150	0.0000	0.0000	125.0	0.62						
50	100	0.0021	0.1738	122.3	0.64		$4n^2/5$	0.00	0.17	123	2.64
	200	0.0006	0.0754	124.1	1.33						
	450	0.0002	0.0193	124.8	2.86						
	900	0.0000	0.0000	125.0	5.76						
100	100	0.0077	0.4117	107.2	2.80		$2n^2/5$	0.01	0.28	113	2.95
	200	0.0031	0.2357	116.6	5.72						
	500	0.0012	0.1410	120.9	13.95						
	1300	0.0001	0.0534	123.2	36.40						

Table 5.6: Computational results for iterated dynasearch and multi-start tabu search.

Table 5.6 clearly shows that iterated dynasearch generates better solutions than the tabu search algorithm, in considerably shorter computation times. The speedups of section 5.2.4 contribute significantly to the reduced computation times. From our computational results, we conclude that iterated dynasearch is preferred to all other known local search algorithms for the total weighted tardiness problem.

Even without the speedups, we claim that dynasearch is preferred. Typically, the speedups reduce the run times of dynasearch for the instances with 40, 50 and 100 jobs by factors of 2.6, 3.4 and 5.5, respectively. However, a tabu search algorithm with the appropriate speedups must yield smaller reduction factors, since some of the savings in computation time can only be achieved with a search for improving moves. Even allowing for reduced computation times for tabu search, the quality of solutions produced by tabu search remains lower than that for dynasearch under similar run times.

5.5 Conclusions

The contribution of this chapter is threefold. First, it gives evidence of the effectiveness of a new local search technique known as dynasearch.

The second contribution relates specifically to the application of dynasearch to the single-machine total weighted tardiness scheduling problem. An iterated dynasearch algorithm for this problem performs significantly better than all other known local search algorithms, including the state-of-the-art tabu search algorithm of Crauwels, Potts and Van Wassenhove (1998), with respect to both solution quality and computation time. The explanation for iterated dynasearch finding better solutions is that, after performing a sufficiently large number of random moves from a local minimum, dynasearch stands a much better chance of descending into a different local minimum than traditional descent algorithms. In this way, iterated dynasearch is able to explore more distinct local minima. One explanation for iterated dynasearch being faster is the use various speed-ups that sometimes reject a move without a complete evaluation of the total weighted tardiness.

The third contribution relates to the study of iterated local search. We have shown in our iterated local search algorithm for the total weighted tardiness problem that multiple random moves from the underlying local search neighbourhood can form an effective kick.

Chapter 6

The Linear Ordering Problem

6.1 Problem definition and literature review

Consider a sequence of elements, where between any pair of elements there is a preference (the strength of which is given by a weight) as to which element occurs earlier in the sequence. How should the elements be ordered? This in essence is the linear ordering problem. More formally the problem may be defined as follows. Given a matrix $E = (e_{ij})$ of weights with n rows and n columns, the linear ordering problem (sometimes known as the triangulation problem in economics) consists of finding a simultaneous permutation of the columns and rows, such that the sum of entries above the main diagonal (of the permuted matrix) is as large as possible.

More rigorously, let $\sigma = (\sigma(1), \dots, \sigma(n))$ be a permutation or sequence that defines the current position of the particular row and column of elements in relation to their position in the original matrix. For example, a permutation starting with $\sigma(1) = 4$ indicates that the column and row that were originally in position 4 are now in position 1. The objective function value for the simultaneous permutation σ is then given by $\sum_{i=1}^n \sum_{j=i}^n e_{\sigma(i), \sigma(j)}$

Before we outline how the linear ordering problem is polynomially related to the acyclic subgraph problem, we show how the acyclic subgraph problem is trivially equivalent to the feedback arc set problem. Let $D = (V, A)$ be a weighted digraph with arc weights c_{ij} for every $(i, j) \in A$.

The acyclic subgraph problem of D consists of finding an acyclic subgraph $D' = (V, B)$, where $B \subseteq A$, such that $\sum_{(i,j) \in B} c_{ij}$ is as large as possible.

The feedback arc set problem of D consists of finding a set of edges C containing at least one arc from every directed cycle such that $\sum_{(i,j) \in C} c_{ij}$ is as small as possible.

Clearly removing at least one arc from every directed cycle within a weighted digraph leaves an acyclic subgraph. Further removing the set of edges C containing at least one arc from every directed cycle such that $\sum_{(i,j) \in C} c_{ij}$ is as small as possible, leaves an acyclic subgraph $D' = (V, A \setminus C)$ such that $\sum_{(i,j) \in A \setminus C} c_{ij}$ is as large as possible (Kaas 1981).

We now show how the linear ordering problem and acyclic subgraph problem are polynomially related (Grötschel, Jünger & Reinelt 1984a, 1985a). Let a tournament be a digraph $E = (V, A)$ such that A contains exactly one arc with end nodes u and v for all $u, v \in V$. A tournament on n nodes is then an orientation of the complete graph K_n .

An algorithm for solving the acyclic subgraph problem, can be used to find the optimal solution to a linear ordering problem on the weighted digraph D , as follows: Let $M = \max\{-c_{ij} : (i, j) \in A\} + 1$ and $c'_{ij} = c_{ij} + M$. Now that all edges are positive solving the acyclic subgraph problem with edge weights c'_{ij} finds an acyclic tournament, which defines the optimal ordering for the linear ordering problem.

Similarly an algorithm for solving the linear ordering problem can be used to find the optimal solution to an acyclic subgraph problem on the weighted digraph D , as follows: Let $c'_{ij} = \max\{c_{ij}, 0\}$ for all $(i, j) \in A$. Form a complete digraph by adding any missing edges, giving added edges weight $c'_{ij} = 0$. The complete digraph with edge weights c'_{ij} now defines a linear ordering problem for which an optimal ordering can be found. Adding all positive forward arcs to the optimal ordering, forms an acyclic tournament which determines an optimum acyclic subgraph of the original problem D .

Since the linear ordering problem is polynomially related to the acyclic subgraph problem and the acyclic subgraph optimisation problem is NP-hard, the linear ordering optimisation problem must also be NP-hard (Karp 1972).

Before discussing some research conducted on the linear ordering and equivalent problems, there is one more important equivalent version of the linear ordering problem which should be mentioned, the problem of minimising total weighted (or average weighted) completion time in one machine scheduling (Boenchenhendorf 1982).

Many applications have been proposed for the linear ordering problem including multiple criteria decision making, ranking elements in paired comparison experiments, suppressing feedback in electrical networks, ordering archaeological objects in time and triangulating input-output matrices (Glover, Klastorin & Klingman

1974, Reinelt 1985). An input-output matrix, in the triangulation problem, is an $n \times n$ matrix where each entry e_{ij} denotes the monetary value of goods moving from sector i of the economy of a region to sector j . An optimal solution then orders the sectors in such a way that sectors that tend to produce materials for use in other sectors come before sectors that tend to produce goods for end consumers.

The acyclic subgraph problem has been shown to be polynomially solvable on a number of classes of digraph. The acyclic subgraph problem is known to be solvable in polynomial time for planar digraphs (Lucchesi 1976). The problem has also been shown to be polynomially solvable for the more general classes of $K_{3,3}$ -free graphs (Penn & Nutov 1993), reducible flow graphs (Ramachandran 1988), and weakly acyclic graphs (Grötschel, Jünger & Reinelt 1985a).

Kaas (1981) introduced a branch and bound method using a Lagrangean relaxation for the lower bound (based on the Korte and Oberhofer method of Lenstra (1973)), which could handle (34,34) input-output matrices and (25,25) random matrices. Grötschel, Jünger & Reinelt 1984a, 1984b combined branch and bound techniques with a linear cutting plane procedure based on their studies of the acyclic subgraph polytope (Grötschel, Jünger & Reinelt 1985b) to solve structured input-output matrices with dimension up to 60. Grötschel, Jünger & Reinelt conclude that their algorithm compares favourably with existing codes. A subsequent branch and bound algorithm by Flood (1990) for the feedback arc set problem appears uncompetitive. Flood comments that randomly generated problems above dimension 20 remain formidable for his code but fails to mention the earlier algorithms of Kaas (1981) or Grötschel, Jünger & Reinelt (1984a) and the fact that even Kaas's code can solve random problems of dimension 25.

Becker (1967) proposed a simple fast constructive heuristic based on a simple ranking statistic. Becker's heuristic iteratively finds the element with the highest value of his statistic $q_i = \sum_{k=1}^n e_{ik} / \sum_{k=1}^n e_{ki}$, ranking it next. It then repeats the procedure with the element removed, continuing until all elements are ranked. The procedure requires $O(n^3)$ time.

Chanas & Kobylanski (1996) produced a local search procedure which makes use of the property of the linear ordering problem that if a given permutation of the elements is an optimal solution to the maximisation problem then the permutation reversed is an optimal solution to the minimisation problem. In particular once their *sort* procedure is stuck in a local minimum, they reverse the current solution and use

their *sort* procedure again, until two successive local minima have the same value. The sort procedure constructs a solution by iteratively inserting the i th element into the partial solution containing the first $i - 1$ elements, to form a partial solution containing the first i elements, for $i = 1, \dots, n$. The local search method is guaranteed at every stage to obtain a local minimum which is at least as good as the last.

Laguna, Marti & Campos (1998) introduce a complex tabu search for the linear ordering problem which is based on insert moves. The insert moves they use are neither first improve nor best improve as will be outlined below. Firstly as a preprocessing step, a measure of influence for each element is calculated, which measures the relative size of the weights associated with each element. This measure is used during the intensification phase to bias the selection of elements to be removed and inserted elsewhere, to those whose movement is likely to have most effect on the objective function. Given an element to be removed from its current position the best position for it to be inserted is found. Periodically diversification and extra intensification by path relinking occurs, and elite solutions are used in both.

Laguna, Marti & Campos (1998) review the work previously done by Becker (1967) and Chanas & Kobylanski (1996) on the linear ordering problem describing their algorithms in detail as well as comparing them to their own multi-start Greedy (multi-start first improve decent) and Tabu search algorithms. Laguna, Marti & Campos conclude from their computational results that their tabu search is the state-of-the-art local search algorithm for the linear ordering problem.

6.2 Dynasearch

6.2.1 Insert and dynasearch insert neighbourhoods

Unlike in many scheduling problems swapping a pair of elements in the linear ordering problem is likely to disrupt the solution more than moving just one. There appears to be no corresponding effect to the one in some scheduling problems, where the swapping of two jobs with similar processing times leaves the jobs in between virtually unaffected. Therefore given that the insert neighbourhood for the linear ordering problem can be searched in $O(n^2)$ time as compared to $O(n^3)$ for the swap neighbourhood, we only consider the insert neighbourhood.

Example. The matrix below represents an instance of size 5. Leaving the matrix in its current order represents a solution, the permutation 1,2,3,4,5 , with an

objective function of $6+3+1=10$ (the sum of weights in the upper diagonal).

$$\begin{pmatrix} 0 & 0 & 6 & 0 & 3 \\ 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 10 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 \end{pmatrix}$$

Table 6.1 shows the moves that are made by a best improve descent algorithm, with an insert neighbourhood, starting from the initial permutation (1,2,3,4,5).

The two moves that insert job $\sigma(i)$ after job $\sigma(j)$, and job $\sigma(k)$ after job $\sigma(l)$, respectively, are said to be *independent* if $\max\{i, j\} \leq \min\{k-1, l\}$ or $\min\{i-1, j\} \geq \max\{k, l\}$. The dynasearch insert neighbourhood consists of all solutions that can be obtained from σ by a series of pairwise independent insert moves.

Table 6.2 shows the ability of dynasearch to make multiple *independent* insert moves in a single iteration. The dynasearch descent starts from the same permutation (1,2,3,4,5) as the best improve descent, but for this particular instance terminates in a better local optimum. This is not meant to imply that Dynasearch always finds better local optimum than best improve descent; for some instances it will descend to a worse local optimum. The computational results in section 6.4. give an appraisal of the power of the dynasearch insert neighbourhood.

Iteration	Current sequence	Objective function
	1 2 3 4 5	10
1	1 3 4 2 5	20
2	1 5 3 4 2	24

Table 6.1: Insert moves made by best-improve descent.

Iteration	Current sequence	Objective function
	1 2 3 4 5	10
1	2 1 3 4 5	23
2	4 2 1 5 3	32

Table 6.2: Dynasearch Insert moves.

6.2.2 Finding the best set of independent insert moves

It is assumed that the matrix defining the problem instance contains no negative values and that the main diagonal contains zeros. If this is not the case, the matrix may be adjusted as a preprocessing step, and a constant added to the objective function value to account for any changes made.

To find the best set of independent insert moves that can be obtained from a permutation σ , we employ a dynamic programming algorithm. This algorithm uses a forward enumeration scheme in which elements are appended to the end of the current partial permutation, are possibly inserted at an earlier point in the sequence, or have an earlier element in the sequence inserted after them. We define a partial sequence to be in state (j, σ) , for $j = 0, 1, \dots, n$, if it can be obtained from the partial sequence $(\sigma(1), \dots, \sigma(j))$ by applying a series of independent insert moves. Of course, to find the best possible sequence in the dynasearch insert neighbourhood of σ , which by definition must be in state (n, σ) , we only need to consider a sequence that has the maximum objective value among all sequences in this state.

Let σ_j be a partial sequence with the maximum objective function value for the elements $\sigma(1), \dots, \sigma(j)$ among partial sequences in state (j, σ) . Further, let $\Delta(\sigma_j)$ be the increase in the objective function value due to the insert moves performed on the elements $\sigma(1), \dots, \sigma(j)$ to form σ_j . This partial sequence must be obtained from a partial sequence σ_i that has maximum objective value from all partial sequences in some previous state (i, σ) , where $0 \leq i < j$, by appending element $\sigma(j)$ if $i = j - 1$, or by backward or forward insertion if $0 \leq i < j - 1$. A backward insertion is performed by inserting $\sigma(j)$ before $\sigma(i + 1)$ and a forward insertion involves first appending element $\sigma(j)$ and then removing element $\sigma(i+1)$ from its current position and appending it after $\sigma(j)$. These three possibilities are considered in detail below.

- $i = j - 1$. element $\sigma(j)$ is simply appended to a partial sequence σ_{j-1} ; hence, $\sigma_j = (\sigma_{j-1}, \sigma(j))$. Accordingly,

$$\Delta(\sigma_j) = \Delta(\sigma_{j-1})$$

- Backward insertion. Here, element $\sigma(j)$ is inserted before $\sigma(i + 1)$, so that σ_j can be written as $\sigma_j = (\sigma_i, \sigma(j), \sigma(i + 1), \dots, \sigma(j - 1))$, and the increase in

objective function $\Delta(\sigma_j)$ is readily computed as

$$\Delta(\sigma_j) = \Delta(\sigma_i) + \sum_{k=i+1}^{j-1} (e_{\sigma(j), \sigma(k)} - e_{\sigma(k), \sigma(j)}) \quad \text{for } 0 \leq i < j - 1$$

- Forward insertion. Here, element $\sigma(i+1)$ is removed from its current position and reinserted after $\sigma(j)$, so that σ_j can be written as $\sigma_j = (\sigma_i, \sigma(i+2), \dots, \dots, \sigma(j), \sigma(i+1))$, and the increase in objective function $\Delta(\sigma_j)$ is readily computed as

$$\Delta(\sigma_j) = \Delta(\sigma_i) + \sum_{k=i+2}^j (e_{\sigma(k), \sigma(i+1)} - e_{\sigma(i+1), \sigma(k)}) \quad \text{for } 0 \leq i < j - 1$$

Note: A forward insertion with $i = j - 2$ is the same move as a backwards insertion with $i = j - 2$. Both moves swap the elements $\sigma(j-1)$ and $\sigma(j)$. Therefore in the recursion below only forward insertions with $i \leq j - 3$ are considered.

We are now ready to present our dynamic programming algorithm. The initialization is

$$\Delta(\sigma_0) = 0,$$

$$\Delta(\sigma_1) = 0,$$

and the recursion for $j = 2, \dots, n$ is

$$\Delta(\sigma_j) = \max \begin{cases} \Delta(\sigma_{j-1}), \\ \max_{0 \leq i \leq j-2} \left\{ \Delta(\sigma_i) + \sum_{k=i+1}^{j-1} (e_{\sigma(j), \sigma(k)} - e_{\sigma(k), \sigma(j)}) \right\}, \\ \max_{0 \leq i \leq j-3} \left\{ \Delta(\sigma_i) + \sum_{k=i+2}^j (e_{\sigma(k), \sigma(i+1)} - e_{\sigma(i+1), \sigma(k)}) \right\}, \end{cases}$$

The increase in objective function produced by the best set of independent insert moves is then equal to $\Delta(\sigma_n)$, and the corresponding sequence can be found by backtracking.

If as is implied by the summations $\sum_{k=i+1}^{j-1} (e_{\sigma(j), \sigma(k)} - e_{\sigma(k), \sigma(j)})$ and $\sum_{k=i+2}^j (e_{\sigma(k), \sigma(i+1)} - e_{\sigma(i+1), \sigma(k)})$ it takes $O(n)$ time to calculate the effect of each individual insert move on the objective function, the dynamic programming algorithm runs in $O(n^3)$ time. However, if for any given j , possible insert moves are searched in order of decreasing values of i , the change in objective function value caused by each move can be calculated iteratively in constant time as shown below.

Let $\delta(s, t)$ be the effect on the objective function of removing the element currently in position s and inserting it after the element currently in position t .

All values of $\delta(s, t)$ can be calculated in $O(n^2)$ time using the following equations:

$$\begin{aligned}\delta(j-1, j) &= \delta(j, j-2) = e_{\sigma(j), \sigma(j-1)} - e_{\sigma(j-1), \sigma(j)} \\ \delta(j, i) &= \delta(j, i+1) + e_{\sigma(j), \sigma(i+1)} - e_{\sigma(i+1), \sigma(j)} \quad \text{for } i \leq j-3 \text{ (backward insertion)} \\ \delta(i+1, j) &= \delta(i+1, j-1) + e_{\sigma(j), \sigma(i+1)} - e_{\sigma(i+1), \sigma(j)} \quad \text{for } i \leq j-3 \text{ (forward insertion)}\end{aligned}$$

So the recursion for $j = 2, \dots, n$ becomes

$$\Delta(\sigma_j) = \max \begin{cases} \Delta(\sigma_{j-1}), \\ \max_{0 \leq i \leq j-2} \{\Delta(\sigma_i) + \delta(j, i)\}, \\ \max_{0 \leq i \leq j-3} \{\Delta(\sigma_i) + \delta(i+1, j)\}. \end{cases}$$

This dynamic programming algorithm runs in $O(n^2)$ time. Hence, dynasearch requires $O(n^2)$ time to verify local optimality, which is the same time requirement as for a reasonable implementation of a traditional descent algorithm (reasonable indicating the use of an iterative process as above to calculate the effect of a move on the objective function $\delta(i, j)$). The apparent reduction in time complexity from $O(n^3)$ to $O(n^2)$ is possible on many permutation problems with an insert neighbourhood.

It is noted that the algorithms proposed by Laguna, Marti & Campos (1998) appear to search the insert neighbourhood in $O(n^3)$. They comment that it takes $O(n)$ time to evaluate the effect of a particular move. The practical effect of the reduction in computational complexity is investigated in Subsection 6.4.3 by comparing the time the implementations take to perform a best improve descent.

Finally, we note that a backward dynamic programming algorithm can be derived. However, it does not offer any computational advantages, since the backward algorithm has the same time and space requirements as the forward scheme.

6.2.3 An alternative presentation of the dynamic program

Let σ_j be a partial sequence with the maximum objective function value for the elements $\sigma(1), \dots, \sigma(j)$ among partial sequences in state (j, σ) . Further, let $F(\sigma_j)$ be the contribution to the objective function value for elements $\sigma(1), \dots, \sigma(j)$ in σ_j .

As a preprocessing step calculate $F'(j) = \sum_{i=1}^j \sum_{k=i}^n e_{\sigma(i)\sigma(k)}$ which can be calculated for $j = 1, \dots, n$ in $O(n^2)$ as follows:

Let $F'(0) = 0$

$$F'(j) = F'(j-1) + \sum_{k=j+1}^n e_{\sigma(j)\sigma(k)}$$

Initialisation

$$F(\sigma_i) = 0 \text{ for } i = 0, 1$$

Recursion

Calculate $F(\sigma_j)$ for $j = 2, \dots, n$.

$$F(\sigma_j) = \max \begin{cases} F'(j) - F'(j-1) + F(\sigma_{j-1}), \\ \max_{0 \leq i \leq j-2} \left\{ F'(j) - F'(i) + F(\sigma_i) + \sum_{k=i+1}^{j-1} (e_{\sigma(j), \sigma(k)} - e_{\sigma(k), \sigma(j)}) \right\}, \\ \max_{0 \leq i \leq j-3} \left\{ F'(j) - F'(i) + F(\sigma_i) + \sum_{k=i+2}^j (e_{\sigma(k), \sigma(i+1)} - e_{\sigma(i+1), \sigma(k)}) \right\}. \end{cases}$$

The recursion above takes $O(n^3)$ time but with some preprocessing this can be reduced to $O(n^2)$ as shown below. Recall that $\delta(j, i)$ is defined as in subsection 6.2.2 and calculated as before recursively in $O(n^2)$.

Initialisation

$$F(\sigma_i) = 0 \text{ for } i = 0, 1$$

Recursion

Calculate $F(\sigma_j)$ for $j = 2, \dots, n$.

$$F(\sigma_j) = \max \begin{cases} F'(j) - F'(j-1) + F(\sigma_{j-1}), \\ \max_{0 \leq i \leq j-2} \left\{ F'(j) - F'(i) + F(\sigma_i) + \delta(j, i) \right\}, \\ \max_{0 \leq i \leq j-3} \left\{ F'(j) - F'(i) + F(\sigma_i) + \delta(i+1, j) \right\}. \end{cases}$$

6.2.4 Speed-ups

As a consequence of iteratively calculating the effect of almost all moves based on the effect of others (see section 6.2.2), there appears to be little scope for avoiding calculating changes in the objective function through the use of inequalities. Even if suitable inequalities could be found, their effect would seem to be limited as there is no multiplication, division or evaluation of powers required in the calculation of the objective function. The scope for speed-ups in local search appears to be very problem specific: For example, the use of pretesting with inequalities is extremely effective in speeding up local search on the total weighted tardiness problem (see section 5.2.4).

Some time savings can be made for the local search on linear ordering problem based on preprocessing. The number of operations in the calculation of $\delta(s, t)$ can

be reduced by preprocessing. If a matrix $E' = \{e'_{st}\}_{m \times m}$ is produced before the search is started and used in place of $E = \{e_{st}\}_{m \times m}$ where $e'_{st} = e_{st} - e_{ts}$.

Then $\delta(j-1, j) = \delta(j, j-2) = e'_{\sigma(j)\sigma(j-1)}$

and other values can be calculated iteratively:

$$\delta(j, i) = \delta(j, i+1) + e'_{\sigma(j)\sigma(i+1)} \quad \text{for } i+1 \leq j \text{ (backward insertion)}$$

$$\delta(i+1, j) = \delta(i+1, j-1) + e'_{\sigma(j)\sigma(i+1)} \quad \text{for } i+1 \leq j \text{ (forward insertion)}$$

The change in the objective function caused by moving to each of the $(n-1)^2$ neighbours can be calculated in a total of only $(n-1)^2$ additions. It would be hard to conceive of a more efficient way to calculate the change due to a move, than a single addition!

6.3 Implementation of iterated dynasearch

In this section, we present our iterated dynasearch algorithm for the linear ordering problem. To fully evaluate the effectiveness of the dynasearch concept, we compare the empirical performance of iterated dynasearch not only with the performance of the state-of-the-art tabu search by Laguna, Marti & Campos (1998), but also with iterated first-improve and iterated best-improve descent.

In our implementation of these three iterated algorithms, a kick is a series of random insert moves (that in general are not independent) from a previously generated local minimum. Even though the same underlying neighbourhood is used for the kicks, for a sufficiently long series of moves, the random inserts invariably produce a solution that is outside the neighbourhood of the local minimum. Moreover, the local minimum that is obtained as a result of the kick is often different from the one from which the kick is performed. We use the parameter α to denote the number of random insert moves in a kick.

Two basic algorithms are used, one where the acceptance rule is “accept all new local minima S as the new current sequence S_C ”, the other where the acceptance rule is “only accept a new local minimum S if it is better than $S_B = S_C$ ” which we refer to as *accept all* and *accept better* respectively. The accept all rule produces a search in which all kicks are performed from the most recently found local minimum. The effect of using the accept better rule is that kicks are always performed from the best local minimum found by the local search so far.

The use of backtracking effectively produces an algorithm with characteristics

lying between these two extremes. Note that accepting all new local minimum is equivalent to backtracking every infinitely many iterations and only accepting new local minimum S that are better than $S_C = S_B$ is equivalent to backtracking every time. Although there is evidence that backtracking a few times in the search gives slightly better results than either of the two extreme (in terms of backtracking) algorithms for the randomly generated instances, backtracking was not investigated further.

Six basic iterated algorithms are investigated, the two acceptance criteria being used with the three types of descent: first improve, best improve and dynasearch. The best number of random insert moves to use in a kick (the best value for α), was determined for each algorithm in turn.

Random starting solutions were used, because when large numbers of local minima are generated, as is the case in reasonably long runs of iterated local search algorithms, the quality of the starting solution appears to have little effect on the quality of the final solution.

6.4 Computational experience

6.4.1 Experimental design

We use exactly the same set of randomly generated problem instances as Laguna, Marti & Campos (1998). Three sets of problem instance are used.

The first set of problem instances are of input-output tables from sectors in the European economies available electronically at the OR library that is run by Beasley (1990). The set consists of 49 problem instances of size ranging from 44 to 60 elements. These problem instances were first solved to optimality by Grötschel, Jünger & Reinelt (1984b).

The second set of problem instances are also of input-output tables but from sectors in the United States economies which can be obtained from the Stanford graph base Knuth (1993). The set of Stanford graph base problems consists of 25 instances for each size of 40, 60 and 75. We are not aware of any optimal solutions for these problem instances.

The third set of problem instances, unlike the first two problem sets, have no structure; the matrices containing random values were produced by Laguna, Marti & Campos (1998). The random values come from a uniform distribution defined on

the interval (0,25000). The set consists of 25 instances for each size of 75, 150 and 200. Again, we are not aware of any optimal solutions for these problem instances.

All of our descent and dynasearch algorithms were coded in C and run on either a SPARC 5/110 server station or Power Challenge R10000. We compare the performance of the various iterated local search algorithms on the basis of the following statistics:

ARPD = the average relative percentage deviation of the solution value found by the local search algorithm from the optimal (or best known) solution value;

MRPD = the maximum relative percentage deviation of the solution value found by the local search algorithm from the optimal (or best known) solution value;

NO = the number of optimal (or best known) solution values found for the test instances in the problem set;

ACT:SPARC = the average computation time in seconds on a SPARC 5/110 server station;

ACT:R10000 = the average computation time in seconds on a Power Challenge R10000 computer;

NI = the number of iterations performed in multi-start and iterated descent and dynasearch, where an iteration refers to descending to a local minimum.

Laguna, Marti & Campos (1998) give computational results for their greedy (multi-start first improve) and tabu search algorithms as well as the constructive heuristic of Becker (1967) and a multi-start version of the algorithm of Chanas & Korylanski (1996). They perform their computational results on a Pentium containing an Intel LT430TX 166MH motherboard with MMX. The information collected by the Standard Performance Evaluation Corporation (1995) indicates that the Pentium with an Intel LT430TX 166MH motherboard with MMX is a factor 4.044 faster

than the SPARC 5/110 and that the Power Challenge R10000 is a factor 1.534 faster than the Pentium containing an Intel LT430TX 166MH motherboard with MMX. Accordingly we use these conversion factors to make a direct comparison of computation times.

We compare iterated dynasearch with the various local search algorithms reviewed by Laguna, Marti & Campos (1998) using the performance statistics used in their paper, namely:

Value = the average objective function value found by the local search algorithm;

Deviation = the deviation of the average objective function value of the set of problem instances found by the local search algorithm from the optimal (best known) average objective function value of the set of problem instances;

Num. of Opt. (Best) = the number of optimal (or best known) solution values found out of the total number of problem instances in the test set;

equivalent ACT = the average computation time (equivalent for iterated dynasearch) in seconds on a Pentium containing an Intel LT430TX 166MH motherboard with MMX. Additionally for iterated dynasearch the time on a SPARC 5/110 server station for the structured problems and on a Power Challenge R10000 for the random problems is given in parentheses.

The tables also contain the value of α which is given in parentheses beside or beneath the type of search (first improve, best improve, dynasearch). Note that where there are three values within the parentheses, a different value of α was used for each problem size 75, 150, 200, respectively.

During our experimental work, we obtained better solutions for some of the Stanford graph base problems and for many of the random problems than those known to Laguna, Marti & Campos (1998). However, due to the availability of the spreadsheet containing the individual results of Laguna, Marti & Campos, we were able to recalculate the above statistics to allow for the new best solutions known.

6.4.2 Multi-start and iterated dynasearch vs. first-improve and best-improve descent

OR library problems (49 instances)										
	Multi-start			Iterated local search						
Size					accept all			accept better		
		First	Best	Dyna	First (5)	Best (15)	Dyna (20)	First (10)	Best (25)	Dyna (40)
range	ARPD	0.002	0.003	0.001	0.001	0.006	0.000	0.000	0.003	0.000
44-60	MRPD	0.063	0.125	0.015	0.034	0.138	0.015	0.006	0.138	0.000
	No. of Opt	44	44	45	47	44	47	47	47	49
	ACT : Sparc	2.22	2.17	2.01	2.03	1.97	1.98	1.97	2.04	1.22
	NI	35	65	75	205	220	215	130	160	85

Table 6.3: Comparison of iterated and multi-start algorithms for OR library problems

Stanford Graph Base problems (75 instances)										
	Multi-start			Iterated local search						
Size					accept all			accept better		
		First	Best	Dyna	First (5)	Best (15)	Dyna (20)	First (10)	Best (25)	Dyna (40)
40	ARPD	0.001	0.000	0.003	0.000	0.000	0.000	0.000	0.000	0.000
	MRPD	0.015	0.009	0.031	0.002	0.003	0.000	0.000	0.000	0.000
	No. of Best	16	21	17	23	23	25	25	25	25
	ACT : Sparc	0.42	0.42	0.42	0.40	0.41	0.28	0.28	0.44	0.26
	NI	13	17	25	55	50	35	15	35	20
60	ARPD	0.002	0.001	0.001	0.000	0.002	0.000	0.007	0.003	0.000
	MRPD	0.040	0.025	0.012	0.001	0.042	0.002	0.071	0.059	0.000
	No. of Best	11	15	17	19	19	24	20	22	25
	ACT : Sparc	2.75	2.44	2.44	2.39	2.37	2.42	2.44	2.38	1.44
	NI	20	30	45	90	125	130	45	75	45
75	ARPD	0.001	0.001	0.000	0.000	0.002	0.000	0.002	0.003	0.000
	MRPD	0.013	0.009	0.003	0.001	0.016	0.000	0.035	0.035	0.000
	No. of Best	9	8	9	16	14	21	22	22	25
	ACT : Sparc	14.70	14.31	14.36	14.71	14.70	14.23	14.57	14.36	10.88
	NI	50	85	130	340	470	510	190	330	255

Table 6.4: Comparison of iterated and multi-start algorithms for Stanford G.B. problems

We first discuss the value for the parameter α that is selected. Our results indicate that the best value of α depends on the type of acceptance criterion used: searches where the accept better rule is used always prefer a larger value of α than those where the accept all rule is used. The best value for α also appears to depend on the type of descent that is used. For example first improve always prefers a smaller kick than either dynasearch or best improve. There was no evidence amongst the input-output table problem sets of the best value of α depending on the size of instance for any of the six algorithms. However, the reverse is true amongst the random problems, since all six algorithms perform better with larger values of α for larger instances. It must be noted that very much smaller computational times

Random (0,25000) problems (75 instances)										
Size		Multi-start			Iterated local search					
					accept all			accept better		
		First	Best	Dyna	First	Best	Dyna	First	Best	Dyna
					(10,20,25)	(25,80,120)	(20,35,45)	(25,40,45)	(80,120,150)	(40,75,80)
75	ARPD	0.157	0.200	0.185	0.043	0.041	0.018	0.064	0.069	0.032
	MRPD	0.314	0.292	0.306	0.196	0.164	0.078	0.216	0.194	0.183
	No. of Best	0	0	0	2	4	5	0	4	7
	ACT : R10000	1.21	1.01	1.05	1.04	1.03	0.99	1.01	1.00	0.97
	NI	25	40	65	70	90	170	35	60	100
150	ARPD	0.293	0.300	0.340	0.073	0.067	0.043	0.112	0.114	0.100
	MRPD	0.451	0.388	0.470	0.224	0.213	0.118	0.253	0.272	0.206
	No. of Best	0	0	0	0	1	2	1	0	0
	ACT : R10000	9.99	10.51	9.84	10.92	10.46	10.01	10.80	11.93	10.00
	NI	30	50	85	55	70	180	30	40	95
200	ARPD	0.319	0.330	0.358	0.089	0.091	0.047	0.118	0.124	0.103
	MRPD	0.372	0.423	0.502	0.184	0.259	0.165	0.230	0.275	0.241
	No. of Best	0	0	0	0	0	0	0	0	0
	ACT : R10000	20.55	21.64	20.55	20.99	20.01	20.03	23.45	21.64	20.17
	NI	16	35	80	35	45	170	25	40	100

Table 6.5: Comparison of iterated and multi-start algorithms for Random problems(0,25000)

are used relative to the difficulty of the problems for the random instances, and that the smallest size of instances for the random problems is equal to the largest instances amongst the input-output table problems. The best value for α for iterated dynasearch appears to be the same for the input-output table instances of size 75 as for the random instances of size 75. However, for traditional iterated descent, larger values of α appear to be preferred in the local searches performed on the random instances size 75 than for the local searches performed on the input-output tables of size 75.

Finally, it should be noted that the performance of the algorithms is not particularly sensitive to small changes in the value of the parameter α .

For a given acceptance criterion, iterated dynasearch appears significantly to out-perform iterated first improve and iterated best improve descent in terms of the quality of solutions it is able to find in a given time across all problem sizes and types. There does not appear to be a significant difference in the level of performance between iterated first improve and best improve descent for any given acceptance criterion on any of the sets of test instance.

Multi-start dynasearch appears to be slightly more effective than multi-start first improve and best improve descent on the more structured input-output table problem sets, but performs less well on the random problem sets than the traditional multi-start descent versions. On the random problem sets, multi-start first improve

descent appears to be the most effective of the multi-start algorithms. In contrast, on the input-output table problems, first improve seems to be the least effective multi-start algorithm.

For all problems, the accept all criterion yields lower ARPD values for iterated first improve and iterated best improve descent algorithms than for the accept better criterion. However, on the input-output table problems the traditional descents using the accept better acceptance criterion appear to find more best solutions. It is difficult to comment on the number of best solutions found for the random problems, as the time allowed was really too short to find them. The iterated dynasearch using the accept better acceptance criterion, appears to find all the best solutions known for all the input-output table problems in less than the allotted time, and therefore appears to significantly outperform the other eight algorithms in the table both in terms of time and quality of solution obtained. The second most effective algorithm for these problems is iterated dynasearch with the accept all acceptance criterion.

For the random instances, all three types of descent perform better in their iterated version, and in particular with the accept all criterion. On random instances iterated dynasearch appears to outperform the iterated first improve and iterated best improve algorithms for a given acceptance criteria.

To summarise, iterated dynasearch outperforms all forms of multi-start descent and for any given acceptance criterion appears to outperform iterated first improve or best improve descent.

6.4.3 Iterated dynasearch vs. tabu search and other local search methods

It has already been noted that Laguna, Marti & Campos (1998) appear to use an $O(n^3)$ time procedure to search insert neighbourhoods for improving moves in their local search implementations. An $O(n^2)$ time procedure can not be implemented on their particular insert neighbourhood where elements are selected by their *measure of influence* (see introduction) for possible removal from their current position and insertion elsewhere. Therefore their tabu search neighbourhood could not be searched in $O(n^2)$. However, in their initial experiments when they compare their special insert neighbourhood with the simple best improve neighbourhood, they appear to search the best improve neighbourhood in $O(n^3)$ time.

The effect on the computation times merits discussion. Laguna, Marti & Campos

(1998) quote an average CPU time of 0.04 seconds (on their Pentium) to perform a single best improve descent from a random starting solution on each of the 49 OR library problems. Our $O(n^2)$ implementation appears to be approximately five times faster, taking an average CPU time equivalent to about 0.008 seconds on their Pentium (0.033 seconds on our SPARC) to perform a single best improve descent from a random starting solution on each of the 49 OR library problems.

OR library problems (49 instances)						
	Greedy 10	Becker	CK 10	TS LOP	iter ds accept all	iter ds accept better
					(20)	(40)
Value	20375556.16	22038090.39	22040892.14	22041261.51	22041261.76	22041263.82
Deviation	0.02%	8.95%	0.02%	0.00%	0.00%	0.00%
No. of Optimal	22	0	27	47	47	49
equivalent ACT	0.08	0.02	1.06	0.93	0.49 (1.98)	0.30 (1.22)

Table 6.6: Comparison of iterated dynasearch algorithm with other algorithms in the literature on OR library problem instances

Stanford Graph Base problems (75 instances)						
	Greedy 10	Becker	CK 10	TS LOP	iter ds accept all	iter ds accept better
					(20)	(40)
Value	5909898.24	6022126.63	6032591.57	6033124.09	6033168.427	6033169.08
Deviation	0.01%	2.04%	0.01%	0.00%	0.00%	0.00%
No. of Best	19	0	17	59	71	75
equivalent ACT	0.55	0.20	16.33	4.09	1.98 (8.01)	1.57 (6.34)

Table 6.7: Comparison of iterated dynasearch algorithm with other algorithms in the literature on Stanford graph base problem instances

Random (0,25000) problems (75 instances)						
	Greedy 10	Becker	CK 10	TS LOP	iter ds accept all	iter iter ds accept better
					(20,35,45)	(40,75,80)
Value	128981141	125587971.7	128919838	129269367.5	129439318	129374397.3
Deviation	0.40%	3.02%	0.45%	0.18%	0.05%	0.10%
No. of Optimal	0	0	0	4	5	7
equivalent ACT	1.21	0.80	108.44	20.19	14.44 (9.41)	16.21 (10.56)

Table 6.8: Comparison of iterated dynasearch algorithm with other algorithms in the literature on random problem instances

The results from Laguna, Marti & Campos (1998) given in the three tables, appear to indicate that their tabu search is better than other algorithms known at that time, namely multi-start greedy with 10 restarts (multi-start first improve decent), the constructive heuristic of Becker (1967) and multi-start CK with 10 restarts (CK referring to the algorithm by Chanas & Kobylanski (1996)), and therefore is the state-of-the-art algorithm for the linear ordering problem. However, we consider a fairer comparison would have been produced, if the number of restarts was varied to produce a multi-start first improve descent which took the same amount of time

as the tabu search. We used this approach when we compared iterated dynasearch with our implementation of multi-start first improve descent. Tables 6.6, 6.7 and 6.8 using the measures used in Laguna's paper clearly show that iterated dynasearch is considerably more effective than their state-of-the-art tabu search, whichever of the two acceptance criteria is used. For example on the Stanford graph base problems, using the accept better criterion, iterated dynasearch finds a larger number of best solutions known and a higher average objective function value than the state-of-the-art tabu search. This is achieved in far less time than that used by the tabu search. Iterated dynasearch with the accept better criterion is capable of finding all 49 of the optima for the OR library problems and all 75 of the best solutions known for the Stanford graph base problems in less than the corresponding times allowed for the tabu search.

6.4.4 Using a hybrid acceptance criterion in iterated dynasearch

It appears that some of the differences can be explained by the length of run used in obtaining the results in comparison with the average time required to find the best local minimum known. If backtracking is used early in a search the effect can be detrimental. Imagine the situation where a good but not optimal local minimum has been found, the local minimum found may be the best in the vicinity that kicking from the best solution known is likely to find. Kicking from the previous local minimum can be an efficient way to diversify the search without resorting to time consumingly large kicks.

If no backtracking is used in a search, although low ARPD are quickly obtained it may take a long time to find the optimal solution. If it is desirable to find optimal solutions rather than just very good solutions, a hybrid approach appears to be more effective than either individual strategy, where the local search is started using the 'accept all' acceptance criterion, continuing with this strategy long enough so that near optimal solutions are expected, before then changing the acceptance criterion to 'accept better'.

Tables 6.9 and 6.10 demonstrate the benefit of using a hybrid approach. The columns in each table give the results for a particular version of iterated dynasearch. The first column for the version of iterated dynasearch uses the accept all criterion, the second column uses the accept better criterion, and the third starts by using the

accept all criterion, but after a predetermined number of kicks uses the accept better criterion. In the first table, containing results for the Stanford graph base problems, the first 120 kicks, each comprising of 20 insert moves, are performed from the previous local minimum, and then the rest of the kicks, each comprising of 40 insert moves, are performed from the best currently known local minimum. The stopping criterion is to stop the search if there is no improvement in the value of the best known local minimum for 140 successive iterations. The hybrid method in the third column of the second table, containing the random problems, is identical except that the first phase using the accept all criterion lasts for the first 4000 iterations and the stopping criterion is to stop the search if 5000 successive iterations result in no improvement in the objective function.

Stanford Graph Base problems (25 instances)				
size		iter ds accept all	iter ds accept better	iter ds accept all /iter ds accept better
		(20)	(40)	(20)/(40)
75	ARPD	0.000	0.000	0.000
	MRPD	0.000	0.000	0.000
	No. of Best	21	25	25
	ACT : Sparc	14.23	10.88	6.16
	NI	510	255	120/140

Table 6.9: Comparison of different versions of iterated dynasearch on Stanford G.B problems

Random (0,25000) problems (25 instances)				
size		iter ds accept all	iter ds accept better	iter ds accept all /iter ds accept better
		(20)	(40)	(20)/(40)
75	ARPD	0.000	0.009	0.000
	MRPD	0.002	0.096	0.000
	No. of Best	24	18	25
	ACT : R10000	66.10	74.30	24.92
	NI	20000	15000	4000/5000

Table 6.10: Comparison of different versions of iterated dynasearch on Rand(0,25000) problems

6.5 Concluding remarks

The contribution of this chapter is three fold. First, it gives further evidence of the effectiveness of a new local search technique known as dynasearch.

The second contribution is showing how the time complexity of searching the linear ordering problem insert neighbourhood can be reduced to $O(n^2)$ time by searching the neighbourhood in a particular order. This result can be mirrored in many insert neighbourhoods particularly those on problems of finding an optimal permutation.

The third contribution relates specifically to the application of dynasearch to the linear ordering problem. An iterated dynasearch algorithm for this problem performs significantly better than all other known local search algorithms, including the state-of-the-art tabu search algorithm of Laguna, Marti & Campos (1998), with respect to both solution quality and computation time. An iterated dynasearch algorithm has already been shown to be the state-of-the-art algorithm for the single-machine total weighted tardiness problem (see Chapter 5).

Our iterated local search algorithm for the chapter on the total weighted tardiness problem shows that the random moves used in the kick can still be effective even if they belong to the underlying local search neighbourhood. In this chapter we have given more evidence of the ability of multiple random moves from the underlying neighbourhood to form productive kicks.

Chapter 7

Travelling Salesman Problem

7.1 Problem definition and literature review

It is convenient to state the (symmetric) travelling salesman problem as a minimization problem in a complete graph $G = (V, E)$. The set of vertices V correspond to the cities, and the set E contains an edge $\{i, j\}$ of length d_{ij} for each pair of cities i and j ($i = 1, \dots, n$, $j = 1, \dots, n$), where $d_{ji} = d_{ij}$. The TSP is the problem of finding a Hamiltonian cycle of minimum total length that contains each vertex of V exactly once. Any Hamiltonian cycle can be represented as a permutation of the cities; accordingly, the length of a Hamiltonian cycle that is induced by the permutation σ is then

$$\sum_{j=1}^n d_{\sigma(j), \sigma(j+1)},$$

where $\sigma(j)$ is the city in the j 'th position of σ ($j = 1, \dots, n$), and where by definition $\sigma(n + 1) = \sigma(1)$. Henceforth, we call a Hamiltonian cycle simply a *tour*.

The symmetric travelling salesman problem has many applications from large scale integration chip fabrication (Korte 1989) to X-ray crystallography (Bland & Shallcross 1989).

In recent years the record for the largest nontrivial TSP instance solved to optimality has increased substantially from 318 cities in 1980 (Crowder & Padberg 1980) up to 13,509 in 1998 (Applegate, Bixby, Chvatal & Cook 1998). Although the advances seen have been due in part to the increase in computer power, much of the improvement is due to the major developments in the branch and cut algorithms used.

The travelling salesman problem has received more attention than any other intractable combinatorial optimization problem. It has always served as the bench-

mark problem on which many new techniques have been tested.

An excellent overview, is given by Lawler, Lenstra, Rinnooy Kan & Shmoys (1985).

The best descent local search algorithms are the so-called *edge exchanging* methods: these are based on a neighbourhood in which up to k edges in a feasible tour are exchanged for new edges to make a new feasible tour. Among these algorithms, we find the famous k -opt algorithms (Croes 1958, Lin 1965), Or-opt (Or 1976), and the Lin-Kernighan algorithm (Lin & Kernighan, 1973). The latter is widely considered to be the champion of TSP descent heuristics.

The superiority of edges exchanging methods over other descent methods is partly due to the effectiveness of some well known speedups (see section 7.2.1). In fact well implemented k -opt and Lin-Kernighan are so powerful that, as far as we are aware, they currently form a part of all competitive local search algorithms.

Before discussing the state of the art local search algorithms for the TSP in the next section, we will review some of the less competitive local search techniques in the literature. It is particularly interesting to note how some well known techniques, forms of which are competitive on many other problems, cannot compete with well implemented edge exchanging descent methods.

Tabu search algorithms in the literature appear to be inferior to either a single Lin-Kernighan descent or multi-start Lin-Kernighan in terms of the solution quality obtainable within a given time frame. Despite the leap in performance of genetic algorithms since the influential work of Mühlenbein, George-Scheuter & Krämer (1988), it's only recently that they have appeared competitive. Two new papers Nagata & Kobayashi (1997) and Watson et al. (1998) yield encouraging results. Simulated annealing algorithms are unable to compete with a single run of Lin-Kernighan in terms of the solution quality obtainable within a given time frame. However, over longer time periods, simulated annealing algorithms can outperform multi-start Lin-Kernighan on some instances. Neural networks appear to be far from competitive. They struggle to complete with the quality of solution obtainable by a single 2-opt descent, even when very long time periods are allowed. A detailed review of the research undertaken on the TSP using the above types of local search algorithm is given by Johnson & McGeoch (1997).

Finally the reader is referred to section 4.2 for a review of search procedures for the TSP involving PSEN. Of these, only Balas's neighbourhood has been shown

to have the potential in the short term (Balas & Simonetti 1998) to be used in a state-of-the-art local search procedure (Balas & Simonetti 1998). The rotational pyramidal neighbourhood (Carlier & Villon 1990) seems to be only of academic interest, unless suitable speedups are found to reduce the effective run time complexity drastically from $O(n^3)$. Many PSEN's are obviously not of practical interest for reasons explained in section 4.3 and others are yet to be implemented.

We now turn our attention back to more competitive local search techniques containing phases of edge exchanging descent.

7.2 Competitive local search techniques

Johnson & McGeoch (1997) review a large number of methods and believe the most effective method to be the “kick” method, using Lin-Kernighan as the local optimiser which they refer to as Iterated Lin-Kernighan (ILK) (see section 3.7). They state: “It is now widely believed that the Martin-Otto-Felten approach, in particular the ILK variant, is the most cost-effective way to improve on Lin-Kernighan, at least until one reaches stratospheric running times.”

The other seemingly most promising method is guided local search (GLS) as proposed by Voudouris & Tsang (1999). GLS augments the cost function of the problem to include a set of penalty terms for the edges and passes this problem, instead of the original one, for minimisation by the local optimiser. The local optimiser’s search is directed by the penalty terms and focuses attention on promising regions of the search space. Iterative calls are made to a local optimiser. Each time the local optimiser gets caught in a local minimum, the penalties are modified and the local optimiser is called again to minimise the modified cost function (see section 3.8).

The most effective variant of GLS known uses 2-opt as the local optimiser method and can easily outperform multi-start Lin-Kernighan. Although the implementation of GLS-2-opt by Voudouris & Tsang (1999) appears unable to compete with the state-of-the art implementation of ILK (by Applegate, Bixby, Chvatal & Cook 1999), it does have the advantage of being far simpler to implement. The Voudouris & Tsang (1999) implementation of GLS-2-opt is, according to the results in this Chapter, more effective than the best implementation of iterated 3-opt (Stuetzle 1998b) of which we are aware.

No other local search heuristic in the literature appears to be able to compete with the best iterated local search algorithms or with GLS-2-opt. It should be noted that, under some authors' classifications, iterated local search is seen as a form of genetic algorithm (Johnson & McGeoch 1997) and possibly more reasonably GLS is viewed as a type of tabu search, thus invalidating some of the comments above.

As stated before, all of the competitive local search algorithms contain phases of edge exchange descent, the effectiveness of which is partially dependent on some well known speed-ups which are described in the next sub-section.

7.2.1 Speed-ups for edge exchanging descents

Many efficient implementations of k -opt and Lin-Kernighan local search algorithms use the same well known tricks. Unlike the speed-ups which we used on the TWTP that did not effect the dynasearch move found, these speed-ups do affect the moves made, and consequently the local minimum found. However, in practice the speed-ups, when implemented correctly, are capable of producing impressive reductions in the time taken to search a given neighbourhood, at the cost of a small reduction in the average quality of local optimum found. Such speedups include *Locality searches* using neighbourhood lists and *don't look bits*.

Locality searches, introduced by Steiglitz & Weiner (1968), are based on the observation that, for at least one city a in a 2-opt swap, the edge removed from the tour that was connected to a is larger than the new edge in the tour that is now connected to a . So the 2-opt neighbourhood can be searched in the following way. For each city a , we consider each of its current neighbours as city b . We then consider as city c , all of the cities on a 's neighbourhood list that are closer than city b . There then only exists one current neighbour d to c that could produce a legitimate 2-opt move. The 2-opt move considered breaks edges $\{a, b\}$ and $\{c, d\}$ replacing them with $\{a, c\}$ and $\{b, d\}$.

The equivalent 3-opt search, first used by Lin & Kernighan (1973), requires two locality searches. The first is identical to the locality search for 2-opt, so we obtain cities a, b, c and d . Edges $\{a, b\}$ and $\{c, d\}$ are broken and edge $\{a, c\}$ is added. The second locality search is centered at d (Bentley (1992) implies that the search needs to be centred at c rather than d , but we think that this is a misprint). We now consider as city e , all cities on d 's neighbourhood list that are less than a distance of $d_{ab} + d_{cd} - d_{ac}$ away. There then exists only one current neighbour f to e that could

produce a legitimate 3-opt move. The 3-opt move considered breaks edges $\{a, b\}$, $\{c, d\}$ and $\{e, f\}$ replacing them with $\{a, c\}$, $\{d, e\}$ and $\{b, f\}$.

The effect of the use of a neighbourhood list, of a constant length k , on a locality search centered on a given city a is to reduce the number of moves considered to a constant. In a 2-opt descent, the locality search centered at a considers less than $2k$ moves. A brief explanation is that b is one of the two cities adjacent to a , city c is one of the k cities on a 's neighbourhood list and given cities a , b and c , then city d is determined. In a 3-opt descent, the locality search centered at a considers less than $2k^2$ moves. The explanation is similar to that for the corresponding 2-opt descent; city b is one of two cities adjacent to a and cities c and e must be selected from neighbourhood lists. As indicated from the above observations, using a locality search with a neighbourhood list of constant length reduces the computational complexity of searching both the 2-opt and 3-opt neighbourhoods to $O(n)$.

Locality searches are generally implemented using neighbourhood lists which were introduced concurrently by Steiglitz & Weiner (1968). They propose storing for each city c a list of the remaining cities in order of increasing distance from c . We use an implementation of neighbourhood lists suggested by Johnson and McGeoch (1997), where only the nearest 20 neighbours are stored, for each city c , in order of increasing distance from c . This makes the search significantly faster and saves memory space at a small cost in terms of missed improving moves.

Don't look bits, introduced by Bentley (1992), effectively save time by indicating cities that are not thought worth considering as city a , for the above locality searches in the current iteration. Obviously, even if a city's bit is coded 0 (often referred to as closed) so it will not be considered as city a in the search, it may still be considered as any of the other cities involved in the 2 or 3-opt move. The idea is that if a previous search centered at a city a has not resulted in the discovery of an improving move and neither of the edges adjacent to a have changed since the search, it is unlikely that a further search centered at a city a will result in the discovery of an improving move. The setting of bits both at the start and within the descent, is very influential in the trade-off between the quality of the local minima found and the speed of the descent required to find it.

The standard rules for updating the status of bits within a descent are:

- If a search centred at city a does not result in the discovery of an improving move then close city a 's bit (set city a 's bit to 0).

- If a new edge $\{a, b\}$ (the edge connecting cities a and b) is introduced to the tour then open city a and b 's bits (set cities a and b 's bits to 1).

In the next section, we show how to embed 2-opt, and 3-opt neighbourhoods into dynasearch algorithms ds-2-opt, and ds-3-opt, respectively.

7.3 Dynasearch

A k -opt algorithm ($k = 2, 3, \dots$) for the TSP starts with any tour, and attempts to improve it by removing up to k edges that are not all adjacent, and replacing them with a corresponding number of new edges so that a new tour is generated. Note that, in the new tour, some segments of the original tour may be reversed. Thus, the k -opt neighbourhood comprises all tours that can be obtained by replacing up to k edges in a tour by new edges. The solutions that are obtained by replacing exactly k non-adjacent edges form the *pure k -opt* neighbourhood.

A search of the k -opt neighbourhood of the current tour is performed with the aim of finding a tour which is shorter than the current tour. If no shorter tour exists in the neighbourhood, then the current tour is a *local optimum* and the algorithm terminates. In a k -opt algorithm, it is usual to adopt a *first improve* strategy. Thus, a new tour is adopted as the current tour if the total length of the edges that are removed exceeds the total length of the new edges.

The computation time required by an iterative local improvement algorithm is related to the size of the neighbourhood. For the k -opt neighbourhood, the size is $O(n^k)$. Thus, the time requirement to check for local optimality is $O(n^k)$.

Suppose that σ is a permutation which defines the current tour, and let $\sigma(n+1) = \sigma(1)$. Consider two pure k -opt neighbours that are obtained by removing edges $\{\sigma(i_1), \sigma(i_1+1)\}, \dots, \{\sigma(i_k), \sigma(i_k+1)\}$ and $\{\sigma(j_1), \sigma(j_1+1)\}, \dots, \{\sigma(j_k), \sigma(j_k+1)\}$, respectively, where $1 \leq i_1 < \dots < i_k \leq n+1$ and $1 \leq j_1 < \dots < j_k \leq n+1$. These neighbours are *independent* if either $i_k < j_1$ or $j_k < i_1$. Informally, the segments of the tour $(\sigma(i_1), \dots, \sigma(i_k+1))$ and $(\sigma(j_1), \dots, \sigma(j_k+1))$ that are rearranged under these moves to produce the two neighbours, contain no common edges. Independence of non-pure k -opt neighbours is defined analogously.

In the corresponding dynasearch algorithm, the ds- k -opt neighbourhood comprises all solutions that can be obtained from the current solution by a series of independent k -opt moves. Moreover, we adopt a *best improve* strategy that selects

the shortest tour in this neighbourhood. A tour is a local optimum in the $ds\text{-}k\text{-opt}$ neighbourhood if, and only if, it is a local optimum in the $k\text{-opt}$ neighbourhood.

In the following subsections, we show, for particular choices of k , how a best $ds\text{-}k\text{-opt}$ neighbour can be found by dynamic programming.

7.3.1 2-opt and $ds\text{-}2\text{-opt}$

The 2-opt neighbourhood of the solution defined by permutation σ contains all tours that are obtained by selecting a pair of distinct non-adjacent edges, say $\{\sigma(i), \sigma(i+1)\}$ and $\{\sigma(j), \sigma(j+1)\}$ where $1 < i+1 < j \leq n$, and replacing them by the edges $\{\sigma(i), \sigma(j)\}$ and $\{\sigma(i+1), \sigma(j+1)\}$; see Figure 7.1 and 7.2. This is the only replacement that creates a feasible tour. In the 2-opt algorithm, the new tour is adopted as the current tour if, and only if, it is shorter; that is, if

$$d_{\sigma(i),\sigma(i+1)} + d_{\sigma(j),\sigma(j+1)} > d_{\sigma(i),\sigma(j)} + d_{\sigma(i+1),\sigma(j+1)}. \quad (7.1)$$

Note that such an edge exchange reverses a part of the tour. The new tour is represented by the permutation $(\sigma(1), \dots, \sigma(i), \sigma(j), \sigma(j-1), \dots, \sigma(i+1), \sigma(j+1), \dots, \sigma(n))$. Observe that a given tour has $n(n-3)/2$ neighbours, so the size of the 2-opt neighbourhood is $O(n^2)$. Since checking whether inequality (7.1) holds takes constant time, verifying local optimality requires $O(n^2)$ time overall.

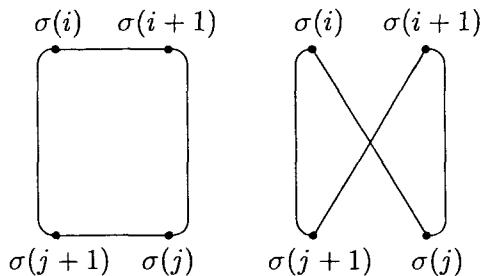


Figure 7.1: A 2-opt move involving the edges $\{\sigma(i), \sigma(i+1)\}$ and $\{\sigma(j), \sigma(j+1)\}$; left the old tour, right the new tour.

As for 2-opt, the input for our dynasearch algorithm $ds\text{-}2\text{-opt}$ is any tour that is defined by some permutation σ of the cities. We now present a dynamic programming algorithm that finds a shortest tour in the $ds\text{-}2\text{-opt}$ neighbourhood. Let $F(j:\sigma)$ be the length of a shortest Hamiltonian path with two fixed endpoints, namely city $\sigma(1)$ and city $\sigma(j)$, that is obtained by applying independent 2-opt moves to the path defined by $(\sigma(1), \dots, \sigma(j))$. In the computation of $F(j+1:\sigma)$, there are then two possible ways to transform this path to include city $\sigma(j+1)$:

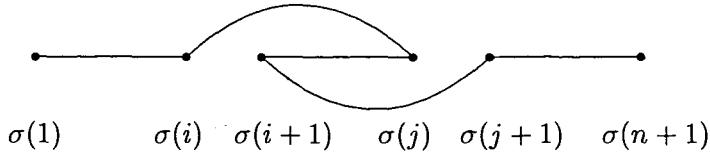
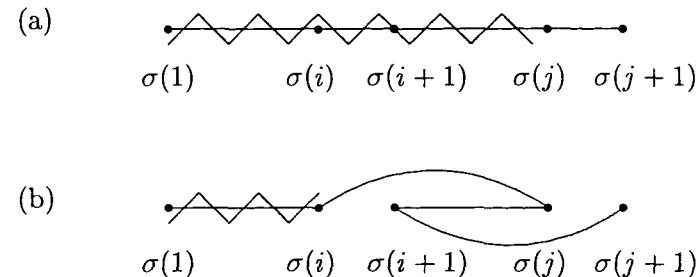


Figure 7.2: An alternative diagram displaying the new tour after the 2-opt move, involving the edges $\{\sigma(i), \sigma(i+1)\}$ and $\{\sigma(j), \sigma(j+1)\}$, which makes use of the fact that $\sigma(1) = \sigma(n+1)$

- (a) go directly from city $\sigma(j)$ to city $\sigma(j+1)$ by adding the edge $\{\sigma(j), \sigma(j+1)\}$ and leave the original path unaltered;
- (b) go to city $\sigma(j+1)$ by adding the edge $\{\sigma(j), \sigma(j+1)\}$ and then perform a 2-opt move involving the edges $\{\sigma(i), \sigma(i+1)\}$ and $\{\sigma(j), \sigma(j+1)\}$ for some i ($i = 1, \dots, j-2$).

In either case, we obtain a Hamiltonian path starting in city $\sigma(1)$ and ending in city $\sigma(j+1)$ that visits all other cities $\sigma(2), \dots, \sigma(j)$, in some order, in between; see Figure 7.3.



where indicates that edges may be involved in other moves

Figure 7.3: The two possibilities in the computation of $F(j+1:\sigma)$.

The initialization of our dynamic programming algorithm is as follows:

$$\begin{aligned} F(1:\sigma) &= 0, \\ F(2:\sigma) &= d_{\sigma(1), \sigma(2)}, \\ F(3:\sigma) &= d_{\sigma(1), \sigma(2)} + d_{\sigma(2), \sigma(3)}, \end{aligned}$$

and the recursion for $j = 3, \dots, n$ is

$$F(j+1:\sigma) = \min \left\{ \begin{array}{l} F(j:\sigma) + d_{\sigma(j),\sigma(j+1)}, \\ \min_{1 \leq i \leq j-2} \left\{ F(i:\sigma) + d_{\sigma(i),\sigma(j)} + D_{\sigma(i+1),\sigma(j)} + d_{\sigma(i+1),\sigma(j+1)} \right\}, \end{array} \right.$$

where $D_{\sigma(i+1),\sigma(j)} = \sum_{k=i+1}^{j-1} d_{\sigma(k),\sigma(k+1)}$. The first expression in the recursion corresponds to possibility (a) and the second expression to possibility (b). Accordingly, $F(n+1:\sigma)$ is the length of the shortest tour in the neighbourhood of our starting solution; the corresponding tour is found by backtracking.

The running time of this dynamic program depends on the time to compute the $D_{\sigma(i+1),\sigma(j)}$ values. Note that

$$\begin{aligned} D_{\sigma(i+1),\sigma(j)} &= \sum_{k=i+1}^{j-1} d_{\sigma(k),\sigma(k+1)} \\ &= D_{\sigma(1),\sigma(j)} - D_{\sigma(1),\sigma(i+1)}, \end{aligned}$$

and that all $D_{\sigma(1),\sigma(j)}$ values ($j = 1, \dots, n$) can be computed and stored in $O(n)$ time in a preprocessing step. Accordingly, we can retrieve each value $D_{\sigma(i+1),\sigma(j)}$ in constant time, and the recursion requires $O(n^2)$ time. Verifying whether a tour is a local optimum in the ds-2-opt neighbourhood requires $O(n^2)$ time, which matches the time requirement to check whether a tour is a local optimum in the 2-opt neighbourhood.

Finally, we note that a *backward* dynamic programming algorithm of the same complexity can be derived. It uses values $G(j:\sigma)$ ($j = 1, \dots, n$) which denote the length of a shortest Hamiltonian path, starting in city $\sigma(n+1)$ and ending in city $\sigma(j)$, that is obtained by applying independent 2-opt moves to the path defined by $(\sigma(n+1), \dots, \sigma(j))$. The worst-case complexity of the backward algorithm is the same as that of the forward algorithm.

7.3.2 $2\frac{1}{2}$ -opt and ds- $2\frac{1}{2}$ -opt

The $2\frac{1}{2}$ -opt neighbourhood refers to the 2-opt neighbourhood in combination with *vertex re-insertion*. Vertex re-insertion is performed by deleting one vertex from the tour and reinserting it in another place. Accordingly, if permutation σ defines the current tour, and we move vertex $\sigma(j)$ between vertices $\sigma(i)$ and $\sigma(i+1)$, as shown in Figure 7.3.2, then the resulting tour is shorter if and only if

$$d_{\sigma(i),\sigma(i+1)} + d_{\sigma(j-1),\sigma(j)} + d_{\sigma(j),\sigma(j+1)} > d_{\sigma(i),\sigma(j)} + d_{\sigma(j),\sigma(i+1)} + d_{\sigma(j-1),\sigma(j+1)}. \quad (7.2)$$

Note that the 3-opt neighbourhood contains the $2\frac{1}{2}$ -opt neighbourhood. However, verifying local optimality in the $2\frac{1}{2}$ -opt neighbourhood only requires $O(n^2)$ time.

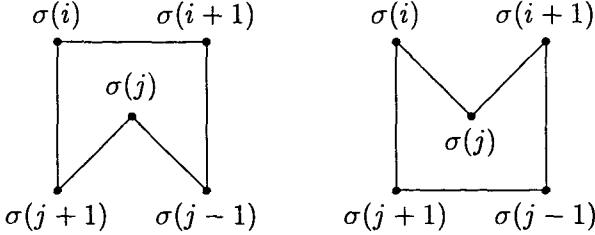


Figure 7.4: Vertex re-insertion where city $\sigma(j)$ is relocated between $\sigma(i)$ and $\sigma(i+1)$; left the old tour, right the new tour.

We now show how to incorporate $2\frac{1}{2}$ -opt in a dynasearch algorithm. It is sufficient to present the underlying dynamic programming algorithm, since other features of the ds- $2\frac{1}{2}$ -opt algorithm are the same as for ds-2-opt.

The algorithm proceeds in the same fashion as for ds-2-opt. We start with any given permutation σ of the cities. Let $F(j : \sigma)$ be the length of a shortest Hamiltonian path with city $\sigma(1)$ and city $\sigma(j)$ as the endpoints, that is obtained by applying independent $2\frac{1}{2}$ -opt moves to the path defined by $(\sigma(1), \dots, \sigma(j))$. There are *four* possible ways to augment this path to include city $\sigma(j+1)$. The first two possibilities, (a) and (b), are identical to those for ds-2-opt in section 7.3.1, while the other two are:

- (c) go to city $\sigma(j+1)$ by adding the edge $\{\sigma(j), \sigma(j+1)\}$ and then reinsert city $\sigma(j)$ between cities $\sigma(i)$ and $\sigma(i+1)$ for some i ($i = 1, \dots, j-2$);
- (d) go to city $\sigma(j+1)$ by adding the edge $\{\sigma(j), \sigma(j+1)\}$ and then reinsert city $\sigma(i+1)$ between cities $\sigma(j)$ and $\sigma(j+1)$ for some i ($i = 1, \dots, j-2$).

Moves (c) and (d) effectuate vertex re-insertion, where move (c) can be regarded as a *backward* re-insertion and (d) as a *forward* re-insertion. They are depicted in Figure 7.5. In all cases, we obtain a Hamiltonian path starting in city $\sigma(1)$ and ending in city $\sigma(j+1)$, and visiting all other cities $\sigma(2), \dots, \sigma(j)$, in some order, in between.

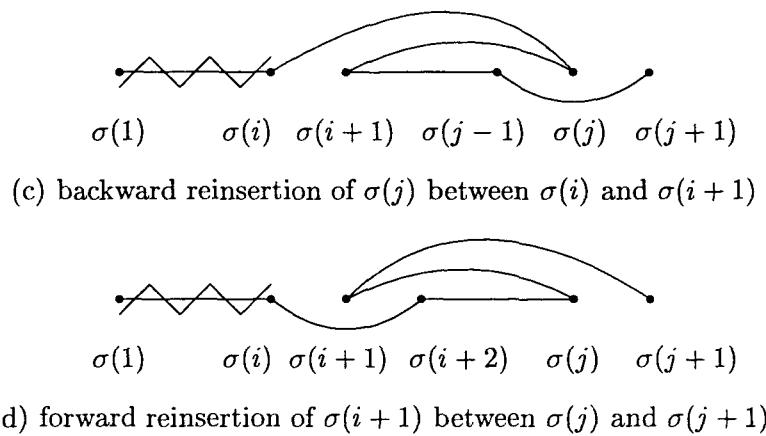


Figure 7.5: The two possible vertex reinsertions to reach state $F(j+1:\sigma)$.

The initialization of our dynamic programming algorithm is as follows:

$$\begin{aligned} F(1:\sigma) &= 0, \\ F(2:\sigma) &= d_{\sigma(1),\sigma(2)}, \\ F(3:\sigma) &= d_{\sigma(1),\sigma(2)} + d_{\sigma(2),\sigma(3)}, \end{aligned}$$

and the recursion for $j = 3, \dots, n$ is

$$F(j+1:\sigma) = \min \left\{ \begin{array}{l} F(j:\sigma) + d_{\sigma(j),\sigma(j+1)}, \\ \min_{1 \leq i \leq j-2} \left\{ F(i:\sigma) + d_{\sigma(i),\sigma(j)} + D_{\sigma(i+1),\sigma(j)} + d_{\sigma(i+1),\sigma(j+1)} \right\}, \\ \min_{1 \leq i \leq j-3} \left\{ F(i:\sigma) + d_{\sigma(i),\sigma(j)} + d_{\sigma(j),\sigma(i+1)} \right. \\ \quad \left. + D_{\sigma(i+1),\sigma(j-1)} + d_{\sigma(j-1),\sigma(j+1)} \right\}, \\ \min_{1 \leq i \leq j-3} \left\{ F(i:\sigma) + d_{\sigma(i),\sigma(i+2)} + D_{\sigma(i+2),\sigma(j)} \right. \\ \quad \left. + d_{\sigma(j),\sigma(i+1)} + d_{\sigma(i+1),\sigma(j+1)} \right\}. \end{array} \right.$$

The last two expressions in the recursion correspond to vertex re-insertion. The length of a shortest tour in the $ds-2\frac{1}{2}$ -opt neighbourhood of our starting solution is $F(n+1:\sigma)$, and the corresponding tour is found by backtracking. This dynamic programming algorithm also runs in $O(n^2)$ time and $O(n)$ space.

Note that, by dropping the second expression in the recursion, which performs the 2-opt moves, we obtain an algorithm of the same complexity for checking whether a tour is a local optimum in the ds-vertex-re-insertion neighbourhood.

7.3.3 3-opt and ds-3-opt

A 3-opt search strategy is expensive, since each move replaces two or three edges in the current tour by two or three others. Suppose that permutation σ defines the current tour, and that two or three of the distinct edges $\{\sigma(h), \sigma(h+1)\}$, $\{\sigma(i), \sigma(i+1)\}$, and $\{\sigma(j), \sigma(j+1)\}$ are replaced. If all three edges are to be replaced and they are non-adjacent, then there are four triples of new edges that will produce a tour:

$$\{\sigma(h), \sigma(j)\}, \{\sigma(i+1), \sigma(h+1)\} \text{ and } \{\sigma(i), \sigma(j+1)\};$$

$$\{\sigma(h), \sigma(i+1)\}, \{\sigma(j), \sigma(h+1)\} \text{ and } \{\sigma(i), \sigma(j+1)\};$$

$$\{\sigma(h), \sigma(i)\}, \{\sigma(h+1), \sigma(j)\} \text{ and } \{\sigma(i+1), \sigma(j+1)\};$$

$$\{\sigma(h), \sigma(i+1)\}, \{\sigma(j), \sigma(i)\} \text{ and } \{\sigma(h+1), \sigma(j+1)\}.$$

If all three edges are replaced and exactly two of them are adjacent, then 3-opt becomes vertex re-insertion and the new edges are uniquely determined. Also, each pair of non-adjacent edges can be replaced to give a 2-opt neighbour. Since a tour has $O(n^3)$ neighbours under a 3-opt search, verifying local optimality takes $O(n^3)$ time.

The dynamic programming algorithm is of the forward type and proceeds in the same manner as before. There are five possible ways to augment the Hamiltonian path so that city $\sigma(j+1)$ is included. These are:

- (a) go directly from city $\sigma(j)$ to city $\sigma(j+1)$ by adding the edge $\{\sigma(j), \sigma(j+1)\}$ and leave the original path unaltered;
- (b) to (e) go to city $\sigma(j+1)$ by adding the edge $\{\sigma(j), \sigma(j+1)\}$ and then perform a 3-opt move by deleting the edges $\{\sigma(h), \sigma(h+1)\}$, $\{\sigma(i), \sigma(i+1)\}$ and $\{\sigma(j), \sigma(j+1)\}$, with $1 \leq h \leq i-2 \leq j-4$, and replacing them in the four ways that are described above.

In all cases, we obtain a Hamiltonian path starting in city $\sigma(1)$ and ending in city $\sigma(j+1)$, that visits all other cities $\sigma(2), \dots, \sigma(j)$, in some order, in between.

The initialization of our dynamic programming algorithm is as follows:

$$F(1:\sigma) = 0,$$

$$F(2:\sigma) = d_{\sigma(1), \sigma(2)},$$

$$F(3:\sigma) = d_{\sigma(1), \sigma(2)} + d_{\sigma(2), \sigma(3)},$$

$$F(4:\sigma) = d_{\sigma(1), \sigma(2)} + d_{\sigma(2), \sigma(3)} + d_{\sigma(3), \sigma(4)},$$

and the recursion for $j = 4, \dots, n$ is

$$F(j+1:\sigma) = \min \begin{cases} F(j:\sigma) + d_{\sigma(j),\sigma(j+1)}, \\ \min_{1 \leq h \leq i-2 \leq j-4} \left\{ \Delta_{hij} + d_{\sigma(h),\sigma(j)} + d_{\sigma(i+1),\sigma(h+1)} + d_{\sigma(i),\sigma(j+1)} \right\}, \\ \min_{1 \leq h \leq i-2 \leq j-4} \left\{ \Delta_{hij} + d_{\sigma(h),\sigma(i+1)} + d_{\sigma(j),\sigma(h+1)} + d_{\sigma(i),\sigma(j+1)} \right\}, \\ \min_{1 \leq h \leq i-2 \leq j-4} \left\{ \Delta_{hij} + d_{\sigma(h),\sigma(i)} + d_{\sigma(h+1),\sigma(j)} + d_{\sigma(i+1),\sigma(j+1)} \right\}, \\ \min_{1 \leq h \leq i-2 \leq j-4} \left\{ \Delta_{hij} + d_{\sigma(h),\sigma(i+1)} + d_{\sigma(j),\sigma(i)} + d_{\sigma(h+1),\sigma(j+1)} \right\}, \end{cases}$$

where $\Delta_{hij} = F(h:\sigma) + D_{\sigma(h+1),\sigma(i)} + D_{\sigma(i+1),\sigma(j)}$. The length of a shortest tour in the ds-3-opt neighbourhood of our starting solution (excluding 2-opt and $2\frac{1}{2}$ neighbours) is $F(n+1:\sigma)$, and the corresponding tour is found by backtracking. This dynamic program runs in $O(n^3)$ time, but requires only $O(n)$ space. Finally, note that a backward dynamic programming algorithm can be developed in a similar fashion.

7.3.4 An alternative presentation of the dynamic program ds-2-opt neighbourhood

Let $\delta(i, j+1)$ be the improvement obtained due to performing a 2-opt move involving removing the edges $\{\sigma(i), \sigma(i+1)\}$ and $\{\sigma(j), \sigma(j+1)\}$ on the current tour σ :

$$\delta(i, j+1) = d_{\sigma(i),\sigma(i+1)} + d_{\sigma(j),\sigma(j+1)} - d_{\sigma(i),\sigma(j)} - d_{\sigma(i+1),\sigma(j+1)}.$$

Let $\Delta(j : \sigma)$ be the largest reduction in the length of the Hamiltonian path with two fixed endpoints, namely city $\sigma(1)$ and city $\sigma(j)$, that is obtained by applying independent 2-opt moves to the path defined by $(\sigma(1), \dots, \sigma(j))$.

Initialisation

$$\Delta(i:\sigma) = 0 \text{ for } i = 1, 2, 3$$

Recursion

Calculate $\Delta(j:\sigma)$ for $j = 3, \dots, n$:

$$\Delta(j+1:\sigma) = \max \left\{ \Delta(j:\sigma), \max_{1 \leq i \leq j-2} \{ \Delta(i:\sigma) + \delta(i, j+1) \} \right\}.$$

$\Delta(n+1:\sigma)$ is the largest improvement obtainable from a move in the neighbourhood. The move corresponding to the improvement $\Delta(n+1:\sigma)$ is found by backtracking.

ds- $2\frac{1}{2}$ -opt neighbourhood

Let $\delta(i, j + 1)$ be defined as for ds-2-opt. Let $\delta'(j, i)$ be the improvement obtained due to performing a $2\frac{1}{2}$ -opt move involving removing the vertex $\sigma(j)$ from its current position and reinserting it between vertices $\sigma(i)$ and $\sigma(i + 1)$ in the current tour σ , where $j < i - 1$ or $j > i + 2$

$$\delta'(j, i) = d_{\sigma(j-1), \sigma(j)} + d_{\sigma(j), \sigma(j+1)} + d_{\sigma(i), \sigma(i+1)} - d_{\sigma(j), \sigma(i)} - d_{\sigma(j), \sigma(i+1)} - d_{\sigma(j-1), \sigma(j+1)}.$$

Let $\Delta(j : \sigma)$ be the largest reduction in the length of the Hamiltonian path with two fixed endpoints, namely city $\sigma(1)$ and city $\sigma(j)$, that is obtained by applying independent $2\frac{1}{2}$ -opt moves to the path defined by $(\sigma(1), \dots, \sigma(j))$.

Initialisation

$$\Delta(i : \sigma) = 0 \text{ for } i = 1, 2, 3$$

Recursion

Calculate $\Delta(j : \sigma)$ for $j = 3, \dots, n$:

$$\Delta(j + 1 : \sigma) = \max \begin{cases} \Delta(j : \sigma), \\ \max_{1 \leq i \leq j-2} \{\Delta(i : \sigma) + \delta(i, j + 1)\}, \\ \max_{1 \leq i \leq j-3} \{\Delta(i : \sigma) + \delta'(j, i)\}, \\ \max_{1 \leq i \leq j-3} \{\Delta(i : \sigma) + \delta'(i + 1, j)\}. \end{cases}$$

$\Delta(n + 1 : \sigma)$ is the largest improvement obtainable from a move in the neighbourhood. The move corresponding to the improvement $\Delta(n + 1 : \sigma)$ is found by backtracking.

ds-3-opt neighbourhood

Let $\delta(h, i + 1, j + 1)$ be the improvement obtained due to performing a 3-opt move involving removing the edges $\{\sigma(h), \sigma(h + 1)\}$, $\{\sigma(i), \sigma(i + 1)\}$ and $\{\sigma(j), \sigma(j + 1)\}$ on the current tour σ . Then

$$\begin{aligned} \delta(h, i + 1, j + 1) &= d_{\sigma(h), \sigma(h+1)} + d_{\sigma(i), \sigma(i+1)} + d_{\sigma(j), \sigma(j+1)} \\ &\quad - \min \begin{cases} d_{\sigma(h), \sigma(j)} + d_{\sigma(i+1), \sigma(h+1)} + d_{\sigma(i), \sigma(j+1)} \\ d_{\sigma(h), \sigma(i+1)} + d_{\sigma(j), \sigma(h+1)} + d_{\sigma(i), \sigma(j+1)} \\ d_{\sigma(h), \sigma(i)} + d_{\sigma(h+1), \sigma(j)} + d_{\sigma(i+1), \sigma(j+1)} \\ d_{\sigma(h), \sigma(i+1)} + d_{\sigma(j), \sigma(i)} + d_{\sigma(h+1), \sigma(j+1)}. \end{cases} \end{aligned}$$

Let $\Delta(j : \sigma)$ be the largest reduction in the length of the Hamiltonian path with two fixed endpoints, namely city $\sigma(1)$ and city $\sigma(j)$, that is obtained by applying independent 3-opt moves to the path defined by $(\sigma(1), \dots, \sigma(j))$.

Initialisation

$$\Delta(i:\sigma) = 0 \text{ for } i = 1, 2, 3, 4$$

Recursion

Calculate $\Delta(j:\sigma)$ for $j = 4, \dots, n$.

$$\Delta(j+1:\sigma) = \max \left\{ \begin{array}{l} \Delta(j:\sigma), \\ \max_{\substack{1 \leq h \leq i-2 \\ 2 \leq j-4}} \{\Delta(h:\sigma) + \delta(h, i+1, j+1)\}. \end{array} \right.$$

$\Delta(n+1:\sigma)$ is the largest improvement obtainable from a move in the neighbourhood.

The move corresponding to the improvement $\Delta(n+1:\sigma)$ is found by backtracking.

7.3.5 Speed-ups

A dynasearch descent is a type of edge exchange descent method so all of the well known edge exchange descent speed-ups can be implemented (See sub-section 7.2.1).

An outline of how the speed-ups may be implemented in a descent.

begin

Select bits to be open at the start of the descent.

repeat

Select a city a with an open bit.

Perform a locality search centered on city a

using a neighbourhood list of size 20.

If an improving move is found perform the move.

Update the cities bits that have been affected by the search

using the standard rules.

until All cities in the current tour's bits are closed.

end

7.4 Implementation of iterated and GLS dynasearch

In this section, we present our iterated dynasearch and guided local search dynasearch algorithms for the travelling salesman problem problem. A general introduction to iterated local search and guided local search is given in sections 4.5.1 and 4.5.2 respectively.

We used the *nearest neighbour* construction heuristic to produce our initial solution. In this construction heuristic, initially a city is selected at random, $\sigma(1)$. The salesman then visits the nearest city to $\sigma(1)$, which we will label $\sigma(2)$, producing a path $\sigma(1), \sigma(2)$. Iteratively a longer and longer path is built up, until the path contains all n cities, as follows. Given a constructed path $\sigma(1), \dots, \sigma(i - 1)$ the nearest yet unvisited city to $\sigma(i - 1)$ is chosen as city $\sigma(i)$. Finally, $\sigma(n)$ is connected to $\sigma(1)$ producing a tour.

The iterated dynasearch algorithm is simple in terms of its implementation, which uses the criteria as Johnson (1990). All kicks are accepted, there is no backtracking and a new local minimum S is only accepted as the current local minimum S_c if it is better than or equal to the current local minimum.

If the neighbourhood is initially restricted to a dynasearch $2\frac{1}{2}$ -opt neighbourhood in the descent phase, the competitiveness of dynasearch 3-opt combined with kick methods appears to improve significantly. When a local minimum in the $2\frac{1}{2}$ -opt neighbourhood has been found, the full dynasearch 3-opt is searched. Two random 3-opt moves appear to make a more effective kick than a single double bridge (*DB* in the table) 4-opt move. The diagram below of a DB 4-opt move given here for clarity is repeated in section 9.9 where some consequences of the characteristic that one pair of edges broken in the move may be arbitrarily far from another pair of edges broken is discussed.

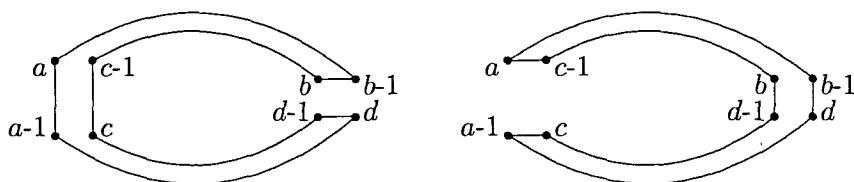


Figure 7.6: A double bridge 4-opt move

As with most recent implementations, in particular the Applegate-Bixby-Chvatal-Cook “Concorde” distribution and Johnson’s later version produced in wake of the “Concorde” distribution, after the kick has been performed only the cities adjacent to an edge directly involved in the kick had their *don’t look bits* open. It should be noted that in some of the first implementations all of the cities had their *don’t look bits* open after a kick (eg Johnson (1990)).

We found the effectiveness of GLS dynasearch 2-opt or 3-opt search were not very sensitive to the size of the control parameter a that determines the size of penalty.

In particular we found $a = \frac{1}{7}$ generally gives the best results for both GLS ds-2-opt and GLS ds-3-opt. The most effective use of “don’t look bits” appears to have the bits open on all of the nodes on the neighbourhood lists of the nodes at either end of the penalised edge.

7.5 Computational experience

7.5.1 Experimental design

The test problem instances used are from the TSPLIB of (Reinelt 1991). They have been used as test problem instances by many authors in the past, and in particular were used by Voudouris and Tsang (1999) with whom we are attempting to compare our computational results.

We used 20 symmetric travelling salesman problem instances ranging in size (number of cities in instance) between 48 and 1002. The distances between a pair of cities, sometimes referred to as the edge weight, are not calculated the same way in all of the problem instances. In fact, the the problem instances are of three distinct edge weight types namely, 2-dimensional Euclidean, 2-dimensional pseudo Euclidean and geographical.

The position of each city in all the test problem instances are described in terms of coordinates. Let x_i and y_i be the coordinates of city i . Descriptions of the way in which the distance d_{ij} between each pair of nodes i and j is calculated for the given type of test problem instance are given below.

2 dimensional Euclidean distance

$$d_{ij} = \left\lfloor \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} + 0.5 \right\rfloor$$

2 dimensional pseudo-Euclidean distance

$$d_{ij} = \left\lceil \sqrt{((x_i - x_j)^2 + (y_i - y_j)^2) / 10} \right\rceil$$

Geographical distance

If the instance has edge weights of the geographical type then x_i refers to the latitude and y_i the longitude, both in degrees and minutes, of a point on an idealised sphere representing the earth. Firstly we will calculate the latitude and longitude in radians, lat_i and long_i for all i .

Let

$$\text{PI} = 3.141592$$

$$\text{deg}^{\text{a}}_i = \lfloor x_i \rfloor$$

$$\text{min}^{\text{a}}_i = x_i - \text{deg}^{\text{a}}_i$$

$$\text{lat}_i = (\text{deg}^{\text{a}}_i + \frac{5}{3}\text{min}^{\text{a}}_i)\text{PI}/180$$

$$\text{deg}^{\text{o}}_i = \lfloor y_i \rfloor$$

$$\text{min}^{\text{o}}_i = y_i - \text{deg}^{\text{o}}_i$$

$$\text{long}_i = (\text{deg}^{\text{o}}_i + \frac{5}{3}\text{min}^{\text{o}}_i)\text{PI}/180$$

Note: min_i is the number of minutes divided by 100 so $\text{deg}_i + \frac{5}{3}\text{min}_i$ gives the angle in degrees. For example, if the angle of latitude is $13^{\circ}45'$ then $x_i = 13.45$, $\text{deg}^{\text{a}}_i = 13$ and $\text{min}^{\text{a}}_i = 0.45$, $0.45 \times \frac{5}{3} = 0.75$, so the angle of latitude is 13.75° . Multiplying by PI/180 converts an angle in degrees into one in radians.

Now that the angles are in radians, it is much easier to calculate the distance between two cities i and j .

let

$$q_1 = \cos(\text{long}_i - \text{long}_j)$$

$$q_2 = \cos(\text{lat}_i - \text{lat}_j)$$

$$q_3 = \cos(\text{lat}_i + \text{lat}_j)$$

$$\text{then } d_{ij} = \lfloor 1 + 6378.388 \arccos(((1 + q_1)q_2 - (1 - q_1)q_3)/2) \rfloor$$

where \arccos is the inverse of the cosine function.

Particular test problem instances used

Each of the names of twenty test instances indicate the size of the particular instance. Fourteen of the test instances used are of the 2-dimensional Euclidean distance edge



type, namely, eil76, kroa100, bier127, kroa150, u159, kroa200, gil262, lin318, pcb442, u574, rat575, u724, rat783, pr1002. Test instances att48, att532 have 2-dimensional pseudo Euclidean distance edge type. Finally four of the twenty test problems have geographical distance edge type (gr202, gr229, gr431 and gr666).

All of our dynasearch algorithms were coded in C and run on our SPARC 5/110 server station. Stuetzle's code was also written in C and run on our SPARC 5/110 server station. Voudouris and Tsang ran their algorithms on a DEC 3000 model 600 computer. The information collected by the Standard Performance Evaluation Corporation (1992) indicates that the DEC 3000 model 600 is a factor 1.452 faster than the SPARC 5/110, and accordingly we use this conversion factor to make a direct comparison of computation times.

We compare the performance of the various local search heuristics using the average relative percentage deviation (ARPD) of the solution value found by the local search heuristic from the optimal.

7.5.2 A comparison of a range of local search algorithms

The table compares GLS and kick methods with a range of traditional and dynasearch local optimisers. The results for GLS with 2-opt, 3-opt and Lin-Kernighan are from Voudouris & Tsang (1999). The results for iterated 3-opt are from code written by Stuetzle (1998b). Stuetzle's code appears to significantly outperform Martin, Otto and Felten's (1991,1992) original implementation of iterated 3-opt, and the implementation of the method by Voudouris & Tsang (1999), and is the most competitive implementation of iterated 3-opt that we are aware of. The values in the table are an average of 10 runs. Voudouris and Tsang's runs were given 5 minutes of computer time on DEC 3000 model 600, the dynasearch runs and Stuetzle's code were given a comparable time of 435 seconds on our SPARC station/server 5/110.

Our results suggest that GLS with restarts (R-GLS) as described in section 4.5.2 is the most effective way to use the local optimisation descent procedure 3-opt dynasearch.

We found that the use of 3-opt dynasearch in the GLS procedure is more effective than 2-opt dynasearch. As can be seen from the results in the table above, Voudouris and Tsang (1999) found 2-opt to be more effective than 3-opt, which in turn was

Problem	Mean Excess (%) over 10 runs										
	GLS LK $a = \frac{1}{10}$	GLS 3-opt $a = \frac{1}{8}$	GLS 2-opt $a = \frac{1}{6}$	iter 3-opt DB	iter 3-opt DB	GLS ds-3-opt $a = \frac{1}{7}$	R-GLS ds-3-opt $a = \frac{1}{7}$	GLS ds-2-opt $a = \frac{1}{7}$	iter ds-3-opt DB	iter ds-3-opt 2x3opts	
att48	0	0	0	0	0	0	0	0	0	0	
ei176	0	0	0	0	0	0	0	0	0	0	
kroA100	0	0	0	0	0	0	0	0	0	0	
bier127	0.207	0.002	0	0	0	0	0	0	0	0	
kroA150	0.002	0.001	0	0	0	0	0	0	0	0	
u159	0	0	0	0	0	0	0	0	0	0	
kroA200	0.089	0.007	0	0	0	0	0	0	0	0	
gr202	0.253	0.012	0	0	0	0	0	0.002	0.009	0	
gr229	0.153	0.015	0.004	0.024	0.005	0	0	0.010	0.016	0.030	
gil262	0.084	0.046	0.004	0.017	0	0	0	0	0	0.013	
lin318	0.583	0.129	0.026	0.164	0.241	0.005	0	0.039	0.245	0.089	
gr431	0.564	0.134	0.024	0.290	0.222	0.022	0.001	0.036	0.179	0.142	
pcb442	0.388	0.038	0.044	0.278	0.082	0.008	0	0.079	0.267	0.246	
att532	0.386	0.225	0.090	0.233	0.082	0.050	0.015	0.072	0.217	0.161	
u574	0.581	0.279	0.141	0.114	0.092	0.026	0.087	0.066	0.231	0.176	
rat575	0.288	0.171	0.099	0.191	0.097	0.031	0.025	0.090	0.276	0.196	
gr666	0.855	0.498	0.206	0.283	0.176	0.201	0.133	0.291	0.270	0.141	
u724	0.613	0.337	0.168	0.232	0.167	0.115	0.072	0.139	0.250	0.256	
rat783	0.511	0.285	0.161	0.452	0.153	0.052	0.012	0.129	0.405	0.308	
pr1002	1.042	0.945	0.621	1.315	0.446	0.384	0.292	0.612	0.583	0.497	
Average Excess	0.330	0.156	0.079	0.180	0.088	0.045	0.032	0.078	0.147	0.113	

Table 7.1: Computational results for a number of local search heuristics

found to be more effective than their implementation of Lin-Kernighan in the GLS procedure. Overall the most effective local optimiser for GLS appears to be 3-opt dynasearch.

The results in the table above suggest that our implementation of iterated 3-opt dynasearch using two random 3-opt moves as a kick, outperforms our implementation of iterated 3-opt dynasearch using a random double bridged 4-opt move as a kick, which in turn outperforms Stuetzle's implementation of iterated 3-opt. It appears that our implementation of iterated 3-opt dynasearch is the most effective iterated local search procedure which does not involve Lin-Kernighan as a local optimiser. This in itself may make our search procedure of interest to many researchers as many authors including Johnson and McGeoch (1997) and Voudouris and Tsang (1999) have commented on how time consuming and difficult it is to produce a reasonable implementation of Lin-Kernighan.

GLS 3-opt appears to outperform all implementations of all local search procedures except for iterated Lin-Kernighan, for the TSP.

In addition to outperforming the implementation of iterated Lin-Kernighan by

Voudouris and Tsang (1999) given in the table, our implementation of GLS dynasearch 3-opt appears to be able to significantly outperform the results from Johnson's early implementation of iterated Lin-Kernighan in Johnson (1990), which are quoted in Johnson and McGeoch (1997).

More recent implementations of iterated Lin-Kernighan appear to be considerably more effective. Johnson says his current implementation of iterated Lin-Kernighan can get within 0.0876 % of the optimal of Pr1002 in 5 minutes on his SGI R10000 (private communication 07/98). Our implementation of GLS dynasearch can on average find the optimal to Pr1002 in 21.43 minutes on SPARCstation/server 5/110 which is equivalent to less than 4 minutes on Johnson's computer. Applegate-Bixby-Chvatal-Cook's version that they released in their "Concorde" distribution appears to be the best implementation of iterated Lin-Kernighan. Johnson says it is typically much faster than his for a given quality tour. (private communication 07/98) Applegate-Bixby-Chvatal-Cook's version using the Delaunay graph finds the optimal for Pr1002 in 98.65 secs on average on their 500 Mhz Dec Alpha 21164 (private communication 08/98) equivalent to approximately 3.5 minutes on Johnson's computer. However their version using the 3-quad nearest graph finds the optimal for Pr1002 even more quickly, averaging 60.53 secs on the same computer (private communication 08/98) equivalent to approximately 2 minutes on Johnson's computer.

7.6 Conclusions

The ultimate competitiveness of GLS dynasearch with the best implementations of iterated Lin-Kernighan, is far from clear. Our implementation of guided local search dynasearch appears more effective than the initial implementations of Iterated Lin-Kernighan by Martin, Otto and Felten (1991, 1992), Johnson (1990), competitive with Johnson (1997), but less effective than the apparent state-of-the-art implementation by Applegate, Bixby, Chvatal and Cook (1999). It is still possible that a fine-tuned version of GLS Dynasearch could become the state-of-the-art method for the TSP.

Given the detailed and time consuming coding required to implement Lin-Kernighan, GLS with dynasearch 3-opt may have some value even if it is slightly less effective than iterated Lin-Kernighan.

Chapter 8

Polynomially Searchable Exponential Neighbourhoods for the Linear Ordering Problem

8.1 The concept giving rise to the new PSEN

Initially, before we introduce PSENs for the linear ordering problem, we introduce some general ideas and methodology which are as relevant to the chapter on PSENs for the TSP as they are to the current chapter. In the current section, we will discuss the underlying concept through which our new PSEN has been formed. In section 8.2 we introduce the methodology by which the sizes of many of the PSENs introduced in this chapter and the next can be found.

The idea behind the whole thesis is combining simple well-known neighbourhood moves, so that the moves can be performed together as a single move. The underlying neighbourhoods have been swap, insert and k -opt. Let a combined move be referred to as a dynasearch move, if its effect on the objective function is equal to the sum of the effects of the individual moves from the underlying neighbourhood. So far in the thesis we have concentrated on dynasearch neighbourhood moves formed by combining disjoint moves. Although we are unable to find other dynasearch PSENs for the TWTP due to the form of its objective function, we have successfully derived a number of other dynasearch PSENs for the LOP and TSP. If a move in a TWTP instance affects a job j , moving it from one position in the solution to another will in general alter the effect on the objective function of any other move in the neighbourhood which involves removing or adding one or more jobs lying between the original and new position of j . Therefore, even if a pair of moves do not directly involve the same jobs, unless the moves are disjoint the move produced by combining

the two moves will not be a dynasearch move. A pair of moves in either of the LOP and TSP that do not directly involve the same elements can be combined to form a dynasearch move if one of the underlying moves is completely contained within the other, in future this will be referred to as *nested*.

In this chapter and the next, we focus on other PSENs for the LOP and TSP which are formed by nested moves from the underlying neighbourhood and combinations of nested and disjoint moves from the underlying neighbourhood. We have also looked at combinations of moves from the underlying neighbourhood which interact with each other (do not only contain dynasearch moves). This means that the effect on the objective function of the combined move is not equal to the sum of the effects of the individual moves from the underlying neighbourhood.

8.2 A method for calculating the size of large neighbourhoods that are formed by combining simple neighbourhood moves.

In this chapter and the next we will use the definition of a neighbourhood given by Aarts & Lenstra (1997) where the current solution is included as a member of its own neighbourhood. Aarts and Lenstra state in their definition of a neighbourhood: “The set $N(i)$ is the neighbourhood of solution i , and each $j \in N(i)$ is a neighbour of i . We shall assume that $i \in N(i)$, for all $i \in \vartheta$ ”, where ϑ has previously been defined as the set of all feasible solutions of the combinatorial optimisation problem.

All of the new PSEN in this chapter and the next are built up by combining one or more of the following simple underlying neighbourhood moves, k -opt, swap, insert and reversal (where a section of the solution is reversed).

Consider the case where any pair of underlying neighbourhood moves have the following characteristic: Given that a pair of underlying neighbourhood moves effect the elements in the sequence in sets S_1 and S_2 respectively, one of the following is true:

- $S_1 \subset S_2$ the elements in set S_1 must be nested within the elements in set S_2 ,
- $S_2 \subset S_1$ the elements in set S_2 must be nested within the elements in set S_1 ,
- $S_1 \cap S_2 = \emptyset$ the elements in sets S_1 and S_2 must be disjoint.

Forming a recurrence relationship

The above characteristic allows us to form a recurrence relationship as follows. Suppose that underlying moves exclusively involve a pair of elements and that no two moves can involve the same element. Given the sequence of elements $(\sigma(1), \sigma(2), \dots, \sigma(n))$. Let the size of the neighbourhood be U_n (using the definition of a neighbourhood given above where the current solution is included as part of the neighbourhood). Let us condition on the involvement of $\sigma(n)$ in any underlying move.

- Given that there is no underlying move involving the element $\sigma(n)$, only the $U_{n-1} - 1$ possible moves on the sequence of elements $(\sigma(1), \sigma(2), \dots, \sigma(n-1))$ can be performed. (The neighbourhood move consisting of not performing a move, has not been included.)
- Suppose that there is an underlying move M involving the elements $\sigma(i)$ and $\sigma(n)$ and that there are no underlying moves involving element $\sigma(n)$ and any element from the partial sequence $\sigma(1), \sigma(2), \dots, \sigma(i-1)$. The number of different combined moves that can be formed depends on the type of underlying moves with which M can be combined (either nested, disjoint or both). If nesting is allowed, then M can be combined with any combination of underlying moves exclusively involving any of the elements $\sigma(i+1), \dots, \sigma(n-1)$. If disjoint moves are allowed, then M can be combined with any combination of underlying moves exclusively involving the elements $\sigma(1), \sigma(2), \dots, \sigma(i-1)$. So the number of possible neighbourhood moves can be calculated in terms of U for every possible value of i . By multiplying the number of combinations of underlying moves which are allowed to be nested in the partial sequence $\sigma(i+1), \dots, \sigma(n-1)$ by the number of combinations of underlying moves which are allowed to exclusively involve $\sigma(1), \sigma(2), \dots, \sigma(i-1)$, given there is an underlying move involving the elements $\sigma(i)$ and $\sigma(n)$.

Finally, the different mutually exclusive sets of neighbours can be added together to produce a recurrence relationship for U_n .

Calculating the order of the neighbourhood size from the recurrence relationship

The next step is to find the generating function $U(x) = \sum_{n=0}^{\infty} U_n x^n$ from which we can find the order of the size of the neighbourhood. Given the generating function

the size of the neighbourhood in terms of the size of the instance n can be found by finding the n th term. For most of the neighbourhoods considered the generating function $U(x)$ is of the form:

$$U(x) = \frac{p(x)}{q(x)}$$

If only the order of the neighbourhood size is required it is sufficient to find the root with the smallest absolute value, $1/|k|$, of the polynomial $q(x)$, as we demonstrate in the example below. Note that k is always positive, a negative value would imply that the size of the neighbourhood is negative for odd size instances. Note that i represents $\sqrt{-1}$ and a, b, c, d, e, f, g and h are real numbers, with $b, d, g > 0$.

As an illustration suppose that

$$U(x) = \frac{a}{1 - bx} + \frac{c}{1 - dx} + \frac{e + fi}{1 - (g + hi)x} + \frac{e - fi}{1 - (g - hi)x}$$

Using a Taylor series expansion, for $|x| < 1$ we can write

$$U(x) = \sum_{n=0}^{\infty} (a(bx)^n + c(dx)^n + (e + fi)((g + hi)x)^n + (e - fi)((g - hi)x)^n)$$

Let $\theta = \arctan(h/g)$ and $r = \sqrt{g^2 + h^2}$. Then we deduce that

$$\begin{aligned} U_n &= ab^n + cd^n + (e + fi)(g + hi)^n + (e - fi)(g - hi)^n \\ &= ab^n + cd^n + (e + fi)(r(\cos \theta + i \sin \theta))^n + (e - fi)(r(\cos \theta - i \sin \theta))^n \\ &= ab^n + cd^n + (e + fi)r^n(\cos n\theta + i \sin n\theta) + (e - fi)r^n(\cos n\theta - i \sin n\theta) \\ &= ab^n + cd^n + 2er^n \cos n\theta - 2fr^n \sin n\theta \\ &= ab^n + cd^n + 2r^n(e \cos n\theta - f \sin n\theta) \end{aligned}$$

For large n , U_n is approximately equal to the term in the equation containing k^n where $k = \max\{b, d, r\}$. Therefore the size of the neighbourhood is $O(k^n)$. For the neighbourhoods studied in this thesis, only one of the variables raised to the power n is greater than one, and in this case the other terms become negligible for a large enough n .

For a few of the neighbourhoods considered the generating function $U(x)$ is of the form:

$$U(x) = \frac{p(x) \pm \sqrt{q(x)}}{r(x)}$$

We conjecture that the order of the neighbourhood size is given by the reciprocal of the smallest root of $q(x)$ based on the form of the expansion of $\sqrt{q(x)}$.

In this chapter and the next when giving the order of a neighbourhood's size as a decimal we will give it to 3 decimal places.

8.3 Introduction to PSEN for the linear ordering problem

The remainder of the chapter is concerned with new polynomially searchable exponential neighbourhoods for the linear ordering problem. All of the neighbourhoods for the LOP are formed by combinations of insert moves. It should be noted that corresponding neighbourhoods could be formed by using combinations of swap moves. For each exponential size neighbourhood, a dynamic program capable of searching the neighbourhood in polynomial time is given. In addition, the order of the size of the neighbourhood is given along with an outline of how it can be calculated.

Throughout this chapter we will use $\delta(s, t)$, as defined in sub-section 6.2.2, to represent the effect on the objective function of removing the element currently in position s and inserting it after the element currently in position t . We also use $M(s, t)$ to refer to the move in which the element currently in position s is inserted after the element currently in position t .

8.4 Some block definitions used to define the neighbourhoods

Let $S_{i,j} = \{i, \dots, j\}$ for $1 \leq i \leq j \leq n$.

In order to form neighbourhoods using the concept of blocks and block operations we need to define both an initial block on which we can perform block operations and the block operations themselves.

We will use the block B^0 defined below as the initial block. The block B_0 which represents an insert move from the underlying neighbourhood is such that:

$$B^0 = B_1^0 \cup B_2^0 \cup B_3^0 \text{ where}$$

$$B_1^0 = (\sigma(i), \dots, \sigma(j)) \text{ for } i \leq j$$

$$B_2^0 = (\sigma(i), \dots, \sigma(j-2), \sigma(j), \dots, \sigma(k), \sigma(j-1), \sigma(k+1), \dots, \sigma(l))$$

for $1 \leq i < j \leq k \leq l \leq n$ (a forward insertion move).

$$B_3^0 = (\sigma(i), \dots, \sigma(j-1), \sigma(k+1), \sigma(j), \dots, \sigma(k), \sigma(k+2), \dots, \sigma(l))$$

for $1 \leq i \leq j \leq k < l \leq n$ (a backward insertion move).

Note that if $k = l$ then $B_2^0 = (\sigma(i), \dots, \sigma(j-2), \sigma(j), \dots, \sigma(k), \sigma(j-1))$ and correspondingly if $i = j$ then $B_3^0 = (\sigma(k+1), \sigma(j), \dots, \sigma(k), \sigma(k+2), \dots, \sigma(l))$

Let $B^C \xrightarrow{D} (B^A, B^B)$ refer to the disjoint block operation combining disjoint blocks B^A and B^B to form a single block of type B^C . Then given the particular blocks B^A , B^B and B^C involve the sets of elements $S_{j,k}$, $S_{l,m}$ and $S_{i,p}$, respectively, the relations $S_{j,k} \subset S_{i,p}$, $S_{l,m} \subset S_{i,p}$, $S_{j,k} \cap S_{l,m} = \emptyset$, must hold.

If the particular blocks B^A and B^B forming B^C are $B^A = (\sigma'(j), \dots, \sigma'(k))$,

$B^B = (\sigma'(l), \dots, \sigma'(m))$ and $i \leq j \leq k < l \leq m \leq p$, then

$$\begin{aligned} B^C = & (\sigma(i), \dots, \sigma(j-1), B^A, \sigma(k+1), \dots, \sigma(l-1), B^B, \sigma(m+1), \dots, \sigma(p)) \\ = & (\sigma(i), \dots, \sigma(j-1), \sigma'(j), \dots, \sigma'(k), \sigma(k+1), \dots, \sigma(l-1), \sigma'(l), \dots, \sigma'(m), \sigma(m+1), \dots, \sigma(p)). \end{aligned}$$

Let $B^B \xrightarrow{N} (B^A)$ refer to a nested block operation on block B^A to form B^B . Then given the particular blocks B^A and B^B involve the sets of nodes $S_{j,k}$ and $S_{i,l}$, respectively, the relation $S_{j,k} \subseteq S_{i,l} - \{l\}$ or $S_{j,k} \subseteq S_{i,l} - \{i\}$ must hold. Suppose that the particular block B^A used to form B^B is $B^A = (\sigma'(j), \dots, \sigma'(k))$,

For a forward insertion: $i < j \leq k \leq l$ then

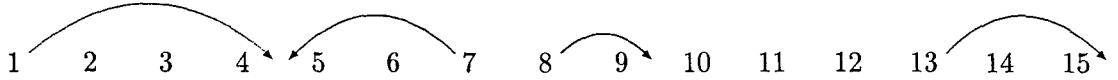
$$B^B = (\sigma(i), \dots, \sigma(j-2), \sigma'(j), \dots, \sigma'(k), \sigma(j-1), \sigma(k+1), \dots, \sigma(l)),$$

and for a backward insertion: $i \leq j \leq k < l$ then

$$B^B = (\sigma(i), \dots, \sigma(j-1), \sigma(k+1), \sigma'(j), \dots, \sigma'(k), \sigma(k+2), \dots, \sigma(l)).$$

8.5 Disjoint insert neighbourhood

The disjoint neighbourhood consists of a combination of disjoint moves (as implemented in Chapter 6), and can be formed by exclusively using disjoint block operations.



A disjoint block may be formed by combining two disjoint blocks with the disjoint block operator.

$$B^D = B^0$$

$$B^D \xrightarrow{D} (B^D, B^D)$$

Dynamic programming algorithm

The disjoint insert neighbourhood can be searched in $O(n^2)$ time. Let $\Delta(j)$ be the maximum increase in objective function achievable by performing a disjoint insert move on the partial sequence $(\sigma(1), \dots, \sigma(j))$.

Initialisation

$$\Delta(0) = \Delta(1) = 0$$

Recursion

Calculate $\Delta(j)$ for $j = 2, \dots, n$:

$$\Delta(j) = \max \left\{ \begin{array}{l} \max_{1 \leq i \leq j-1} \{\Delta(i-1) + \delta(i, j)\} \\ \max_{0 \leq i \leq j-3} \{\Delta(i) + \delta(j, i)\} \\ \Delta(j-1). \end{array} \right.$$

As the effect of inserting $j-1$ after j is the same as inserting j after $j-2$ (both effectively swap the positions of $j-1$ and j) we only consider the former in the equation above. Therefore the limits in the second line of the equation above are $1 \leq i \leq j-3$ not $1 \leq i \leq j-2$.

$\Delta(n)$ is the largest improvement obtainable from a move in the neighbourhood. The move corresponding to the improvement $\Delta(n)$ is found by backtracking.

Neighbourhood size

The size of the neighbourhood is approximately $\Theta(2.206^n)$. We now give an outline of how the size U_n can be calculated.

Starting with the identity permutation $\sigma(1), \sigma(2), \dots, \sigma(n)$, we condition on insert moves involving element $\sigma(n)$.

- Given $\sigma(n)$ remains in position n , then $U_{n-1} - 1$ neighbourhood moves are possible.

To obtain an element other than $\sigma(n)$ in position n , either $\sigma(n)$ must be the element removed from its current position to be inserted elsewhere in the sequence, or another element $\sigma(i)$ must be removed from its current position and inserted after $\sigma(n)$.

- Given $\sigma(n)$ is removed from its current position to be inserted after $\sigma(i)$ in the sequence $(M(n, i))$, the number of possible neighbourhood moves is U_i , where $0 \leq i \leq n - 2$.
- Given element $\sigma(i + 1)$ is removed from its current position to be inserted after $\sigma(n)$ in the sequence $(M(i + 1, n))$, the number of possible neighbourhood moves is U_i , where $0 \leq i \leq n - 3$. Note that if $i = n - 2$, then this is equivalent to swapping elements $\sigma(n - 1)$ and $\sigma(n)$, and therefore equivalent to inserting $\sigma(n)$ after element in position $\sigma(n - 2)$, which has already been counted.

Summing up the different neighbourhood moves and adding the possibility of performing no moves, we obtain

$$U_n = U_{n-1} + \sum_{i=0}^{n-2} U_i + \sum_{i=0}^{n-3} U_i \quad \text{for } n \geq 1,$$

where $U_0 = 1$.

We now outline how the generating function $U(x)$ may be calculated.

$$\begin{aligned} U(x) &= \sum_{i=0}^{\infty} U_i x^i \\ &= U_0 + U_1 x + U_2 x^2 + U_3 x^3 + U_4 x^4 + U_5 x^5 + \dots \\ &= 1 + x(U_0) \\ &\quad + x^2(U_1 + U_0) \\ &\quad + x^3(U_2 + U_1 + U_0 + U_0) \\ &\quad + x^4(U_3 + U_2 + U_1 + U_0 + U_1 + U_0) \\ &\quad + x^5(U_4 + U_3 + U_2 + U_1 + U_0 + U_2 + U_1 + U_0) + \dots \end{aligned}$$

$$\begin{aligned}
 &= 1 + (x + x^2 + x^3 + \dots)U(x) + (x^3 + x^4 + \dots)U(x) \\
 &= 1 + \frac{x}{1-x}U(x) + \frac{x^3}{1-x}U(x)
 \end{aligned}$$

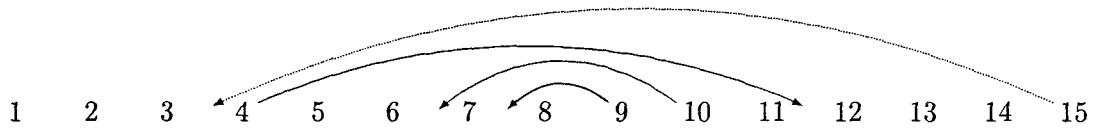
assuming that $|x| \leq 1$. Therefore,

$$U(x) = \frac{1-x}{1-2x-x^3}$$

Since the smallest root of $1 - 2x - x^3 = 0$ is approximately $1/2.206$, the neighbourhood size is approximately $\Theta(2.206^n)$.

8.6 Nested insert neighbourhood

The nested neighbourhood consists of a combination of moves nested within each other, and can be formed by exclusively using nested block operations.



A nested block may be formed by performing a nested block operation on a nested block.

$$B^N = B^0$$

$$B^N \xrightarrow{N} (B^N)$$

Dynamic programming algorithm

The nested insert neighbourhood can be searched in $O(n^2)$ time. Let $\Delta(i, j)$ be the maximum increase in objective function achievable by performing a nested insert move on the partial sequence $(\sigma(i), \dots, \sigma(j))$.

Initialisation

$$\Delta(i, i) = 0 \text{ for } i = 1, \dots, n.$$

Recursion

Calculate $\Delta(i, j)$ for $j - i, \dots, n - 1$, where $1 \leq i < j \leq n$:

$$\Delta(i, j) = \max \begin{cases} \Delta(i+1, j) + \delta(i, j) \\ \Delta(i, j-1) + \delta(j, i-1) \\ \Delta(i+1, j) \\ \Delta(i, j-1). \end{cases}$$

$\Delta(1, n)$ is the largest improvement obtainable from a move in the neighbourhood. The move corresponding to the improvement $\Delta(1, n)$ is found by backtracking.

Neighbourhood size

The size of the neighbourhood is $\Theta((2 + \sqrt{3})^n)$. We now give an outline of how the size U_n can be calculated.

Starting with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$, we condition on insert moves involving element $\sigma(n)$.

- Given $\sigma(n)$ remains in position n , then $U_{n-1} - 1$ neighbourhood moves are possible.

To obtain an element other than $\sigma(n)$ in position n , either $\sigma(n)$ must be the element removed from its current position to be inserted elsewhere in the sequence, or another element $\sigma(n-i)$ must be removed from its current position and inserted after $\sigma(n)$.

- Given element $\sigma(n)$ is removed from its current position to be inserted after element $\sigma(n-i-1)$ in the sequence $(M(n, n-1-i))$, the number of possible neighbourhood moves is U_i , where $1 \leq i \leq n-1$.
- Given element $\sigma(n-i)$ is removed from its current position to be inserted after element $\sigma(n)$ in the sequence $(M(n-i, n))$, and $\sigma(n-i)$ appears in position n (i.e no element in the partial sequence $\sigma(1), \dots, \sigma(n-i-1)$ is inserted after element $\sigma(n)$), the number of possible neighbourhood moves is U_i , where $2 \leq i \leq n-1$. Note that if $i=1$ then this is equivalent to swapping elements $\sigma(n-1)$ and $\sigma(n)$ and therefore equivalent to inserting element $\sigma(n)$ after $\sigma(n-2)$, which has already been counted.

Summing up the different neighbourhood moves and adding the possibility of performing no moves, we obtain

$$U_n = U_{n-1} + \sum_{i=1}^{n-1} U_i + \sum_{i=2}^{n-1} U_i \quad \text{for } n \geq 2,$$

where $U_1 = 1$.

We now outline how the generating function $U(x)$ may be calculated. Note that for convenience the term u_0 is omitted from the generating function.

$$\begin{aligned}
 U(x) &= \sum_{i=1}^{\infty} U_i x^i \\
 &= U_1 x + U_2 x^2 + U_3 x^3 + U_4 x^4 + U_5 x^5 + \dots \\
 &\quad x + x^2(U_1 + U_1) \\
 &\quad + x^3(U_2 + U_2 + U_1 + U_2) \\
 &\quad + x^4(U_3 + U_3 + U_2 + U_1 + U_3 + U_2) \\
 &\quad + x^5(U_4 + U_4 + U_3 + U_2 + U_1 + U_4 + U_3 + U_2) + \dots \\
 &= x + xU(x) + 2(x + x^2 + x^3 + \dots)U(x) - (x + x^2 + x^3 + \dots)xU_1 \\
 &= x + xU(x) + \frac{2x}{1-x}U(x) - \frac{x^2}{1-x}
 \end{aligned}$$

assuming that $|x| \leq 1$. Therefore,

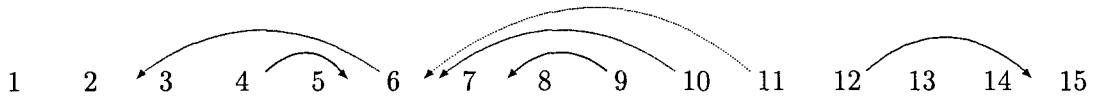
$$U(x) = \frac{x(1-2x)}{1-4x+x^2}$$

Since the smallest root of $1 - 4x + x^2 = 0$ is $2 - \sqrt{3}$, the neighbourhood size is $\Theta((2 + \sqrt{3})^n)$.

8.7 Nested insert inside disjoint insert neighbourhood

The idea behind the nested insert inside disjoint insert neighbourhood is that even though a disjoint neighbourhood move contains the set of disjoint moves which produce the maximum increase in the objective function value, a further increase may be possible by performing moves that are nested within the disjoint moves. The choice of disjoint moves may change in light of the allowance of the additional nested moves.

Only nested moves are allowed within the disjoint moves, thus enabling the best nested neighbourhood move to be found for each partial sequence as a preprocessing step. Consequently, there is a reduction in computational complexity as compared with the more general combined disjoint and nested neighbourhood (see section 8.9).



A nested inside disjoint block may be formed by combining a nested inside disjoint block with a nested block using the disjoint block operator.

$$B = B^N$$

$$B \xrightarrow{D} (B, B^N)$$

Dynamic programming algorithm

The nested insert inside disjoint insert neighbourhood can be searched in $O(n^2)$ time. Firstly, the improvements for the nested insert neighbourhood is calculated in $O(n^2)$ time. Specifically, we compute $\Delta^N(i, j)$, the maximum increase in objective function achievable by performing a nested insert move on the partial sequence of elements $(\sigma(i), \dots, \sigma(j))$, for $1 \leq i \leq j \leq n$.

Let $\Delta(j)$ be the maximum increase in objective function achievable by performing a nested insert inside disjoint insert move on the partial sequence $(\sigma(1), \dots, \sigma(j))$.

Initialisation

$$\Delta(0) = \Delta(1) = 0$$

Recursion

Calculate $\Delta(j)$ for $j = 2, \dots, n$:

$$\Delta(j) = \max_{1 \leq i \leq j} \{\Delta(i - 1) + \Delta^N(i, j)\}.$$

$\Delta(n)$ is the largest improvement obtainable from a move in the neighbourhood. The move corresponding to the improvement $\Delta(n)$ is found by backtracking.

Neighbourhood size

The size of the neighbourhood is approximately $\Theta(4.06^n)$. We now give an outline of how the size U_n can be calculated.

Let N_n be the size of the nested insert neighbourhood for a problem instance of size n (as calculated in section 8.6). Starting with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$, we condition on insert moves involving element $\sigma(n)$.

- Given $\sigma(n)$ remains in position n , then $U_{n-1} - 1$ neighbourhood moves are possible.

To obtain an element other than $\sigma(n)$ in position n , either $\sigma(n)$ must be the element removed from its current position to be inserted elsewhere in the sequence, or another element $\sigma(i)$ must be removed from its current position and inserted after $\sigma(n)$.

- Given element $\sigma(n)$ is removed from its current position to be inserted after $\sigma(i)$ in the sequence $(M(n, i))$, the number of possible neighbourhood moves

is $U_i N_{n-i-1}$, where $0 \leq i \leq n - 2$.

- Given element $\sigma(i+1)$ is removed from its current position to be inserted after $\sigma(n)$ in the sequence $(M(i+1, n))$, and $\sigma(i+1)$ appears in position n , the number of possible neighbourhood moves is $U_i N_{n-i-1}$, where $0 \leq i \leq n - 3$. Note that if $i = n - 2$, then this is equivalent to swapping $\sigma(n-1)$ and $\sigma(n)$, and therefore equivalent to inserting element $\sigma(n)$ after $\sigma(n-2)$, which has already been counted.

Summing up the different neighbourhood moves and adding the possibility of performing no moves, we obtain

$$U_n = U_{n-1} + U_{n-2} + 2 \sum_{i=0}^{n-3} U_i N_{n-1-i} \quad \text{for } n \geq 1,$$

where $U_0 = 1$.

We now outline how the generating function $U(x)$ may be calculated.

$$\begin{aligned} U(x) &= \sum_{i=0}^{\infty} U_i x^i \\ &= U_0 + U_1 x + U_2 x^2 + U_3 x^3 + U_4 x^4 + U_5 x^5 + \dots \\ &= 1 + x(U_0) \\ &\quad + x^2(U_1 + U_0) \\ &\quad + x^3(U_2 + U_1 + 2U_0 N_2) \\ &\quad + x^4(U_3 + U_2 + 2U_1 N_2 + 2U_0 N_3) \\ &\quad + x^5(U_4 + U_3 + 2U_2 N_2 + 2U_1 N_3 + U_0 N_4) + \dots \\ &= 1 + (x + x^2)U(x) + 2x(x^2 N_2 + x^3 N_3 + \dots)U(x) \\ &= 1 + (x + x^2)U(x) - 2x^2 N_1 U(x) + 2x N(x)U(x) \\ &= 1 + xU(x) - x^2 U(x) + 2x \frac{x(1 - 2x)}{1 - 4x + x^2} U(x) \end{aligned}$$

assuming that $|x| \leq 1$. Therefore,

$$U(x) = \frac{1 - 4x + x^2}{(1 - x)(1 - 4x - x^3)}$$

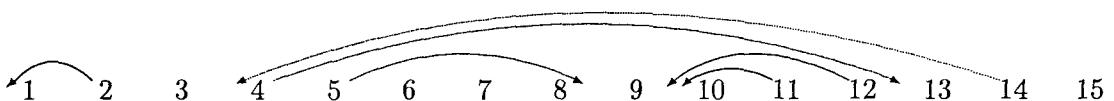
Since the smallest root of $(1 - x)(1 - 4x - x^3) = 0$ is approximately $1/4.06$, the neighbourhood size is approximately $\Theta(4.06^n)$.

8.8 Disjoint insert inside nested insert neighbourhood

The disjoint inside nested neighbourhood is an extension of the nested neighbourhood in exactly the same way as the nested inside disjoint neighbourhood is an extension of the disjoint neighbourhood.

The idea behind this neighbourhood is that, even though a nested neighbourhood move contains the set of nested moves which produce the maximum increase in the objective function value, a further increase may be possible by performing disjoint moves on the partial sequences which are unaffected by the nested moves. The choice of nested moves may change in light of the allowance of the additional moves.

Only disjoint moves are allowed on the partial sequences which are unaffected by the nested moves, although these moves may be nested within future blocks, thus enabling the best disjoint neighbourhood move to be found for each partial sequence as a preprocessing step. Consequently, there may be (although there is not in this particular case) a reduction in computational complexity as compared with the more general combined disjoint and nested neighbourhood see (section 8.9).



A disjoint inside nested block may be formed either by combining a disjoint inside nested block with a disjoint block using the disjoint block operator, or by performing a nested block operation on a disjoint inside nested block.

$$B = B^0$$

$$B \xrightarrow{N} (B)$$

$$B \xrightarrow{D} (B, B^D)$$

Dynamic programming algorithm

The disjoint insert inside nested insert neighbourhood can be searched in $O(n^3)$ time. Let $\Delta^D(i, j)$, the maximum increase in objective function achievable by performing a disjoint insert move on the partial sequence of elements $(\sigma(i), \dots, \sigma(j))$, for $1 \leq i \leq j \leq n$.

Let $\Delta(i, j)$ be the maximum increase in objective function achievable by performing a disjoint insert inside nested insert neighbourhood move on the partial sequence of elements $(\sigma(i), \dots, \sigma(j))$.

As seen below the calculation of $\Delta^D(i, j)$ for all i, j where $1 \leq i \leq j \leq n$ is required in order to calculate $\Delta(i, j)$.

Initialisation

$$\Delta^D(i, i) = 0 \text{ for } i = 1, \dots, n$$

$$\Delta(i, i) = 0 \text{ for } i = 1, \dots, n$$

Recursion

Calculate $\Delta^D(i, j)$ for $i = 1, \dots, n - 1$ and $j = i + 1, \dots, n$:

$$\Delta^D(i, j) = \max \begin{cases} \max_{i \leq k \leq j-2} \{\Delta^D(i, k) + \delta(j, k)\} \\ \max_{i \leq k \leq j-3} \{\Delta^D(i, k) + \delta(k+1, j)\} \\ \Delta^D(i, j-1) \end{cases}$$

Now we are ready to calculate $\Delta(i, j)$ for $j - i, \dots, n - 1$, where $1 \leq i < j \leq n$.

$$\Delta(i, j) = \max \begin{cases} \max_{i+1 \leq k \leq j} \{\Delta(i, k-1) + \Delta^D(k, j)\} \\ \max_{i+1 \leq k \leq j} \{\Delta^D(i, k-1) + \Delta(k, j)\} \\ \Delta^D(i, j-1) + \delta(j, i-1) \\ \Delta^D(i+1, j) + \delta(i, j) \end{cases}$$

$\Delta(1, n)$ is the largest improvement obtainable from a move in the neighbourhood. The move in the disjoint insert inside nested insert neighbourhood corresponding to the improvement $\Delta(1, n)$ is found by backtracking.

Neighbourhood size

The size of the neighbourhood is approximately $\Theta(4.22^n)$. We now give an outline of how the size U_n can be calculated.

Let D_n be the size of the disjoint insert neighbourhood for a problem instance of size n (as calculated in section 8.5). Starting with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$, we condition on insert moves involving element $\sigma(n)$.

- Given $\sigma(n)$ remains in position n , then $U_{n-1} - 1$ neighbourhood moves are possible.

To obtain an element other than $\sigma(n)$ in position n , either $\sigma(n)$ must be the element removed from its current position to be inserted elsewhere in the sequence, or another element $\sigma(i)$ must be removed from its current position and inserted after $\sigma(n)$.

- Given element $\sigma(n)$ is removed from its current position to be inserted after $\sigma(i)$ in the sequence $(M(n, i))$ and this move does not create any nesting (the partial sequence $(\sigma(i+1), \dots, \sigma(n-1))$ appears at the end of the sequence), the number of possible neighbourhood moves is U_i , where $0 \leq i \leq n-2$.
- Given element $\sigma(n)$ is removed from its current position to be inserted after $\sigma(i)$ in the sequence $(M(n, i))$ and there are nested moves in the partial sequence $(\sigma(i+1), \dots, \sigma(n-1))$, the number of possible neighbourhood moves is $D_i(U_{n-i-1} - 1)$, where $0 \leq i \leq n-3$ (for $i = n-2$, there are no non-trivial nested moves).
- Given element $\sigma(i+1)$ is removed from its current position to be inserted after $\sigma(n)$ in the sequence $(M(i+1, n))$ and this move does not create any nesting (the partial sequence $(\sigma(i+2), \dots, \sigma(n), \sigma(i+1))$ appears at the end of the sequence), the number of possible neighbourhood moves is U_i , where $0 \leq i \leq n-3$. Note that if $i = n-2$, then this is equivalent to swapping $\sigma(n-1)$ and $\sigma(n)$ and therefore equivalent to inserting element $\sigma(n)$ after $\sigma(n-2)$ which has already been counted.
- Given element $\sigma(i+1)$ is removed from its current position to be inserted after $\sigma(n)$ and so that it appears in position n in the sequence $(M(i+1, n))$ and there are nested moves in the partial sequence $(\sigma(i+2), \dots, \sigma(n))$, the number of possible neighbourhood moves is $D_i(U_{n-i-1} - 1)$. i can range between 0 and

$n - 3$.

Summing up the different neighbourhood moves and adding the possibility of performing no moves, we obtain

$$U_n = U_{n-1} + U_{n-2} + 2 \sum_{i=0}^{n-3} U_i + 2 \sum_{i=0}^{n-3} D_i(U_{n-1-i} - 1) \quad \text{for } n \geq 2,$$

where $U_0 = U_1 = 1$.

We now outline how the generating function $U(x)$ may be calculated.

$$\begin{aligned} U(x) &= \sum_{i=0}^{\infty} U_i x^i \\ &= U_0 + U_1 x + U_2 x^2 + U_3 x^3 + U_4 x^4 + U_5 x^5 + \dots \\ &= 1 + x(U_0) \\ &\quad + x^2(U_1 + U_0) \\ &\quad + x^3(U_2 + U_1 + 2U_0) + 2(U_2 - 1)D_0 \\ &\quad + x^4(U_3 + U_2 + 2U_1 + 2U_0 + 2(U_2 - 1)D_1 + 2(U_3 - 1)D_0) \dots \\ &= 1 + (x + x^2)U(x) + 2(x^3 + x^4 + \dots)U(x) \\ &\quad + 2x(x^2(U_2 - 1) + x^3(U_3 - 1) + \dots)D(x) \\ &= 1 + (x + x^2)U(x) + \frac{2x^3}{1-x}U(x) \\ &\quad + 2xU(x)D(x) - 2x(U_0 + U_1 x)D(x) - \frac{2x^3}{1-x}D(x) \\ &= 1 + (x + x^2)U(x) + \frac{2x^3}{1-x}U(x) + 2xU(x)D(x) - \frac{2x}{1-x}D(x) \\ &= 1 + (x + x^2)U(x) + \frac{2x^3}{1-x}U(x) + \frac{2x(1-x)}{1-2x-x^3}U(x) - \frac{2x}{1-2x-x^3} \end{aligned}$$

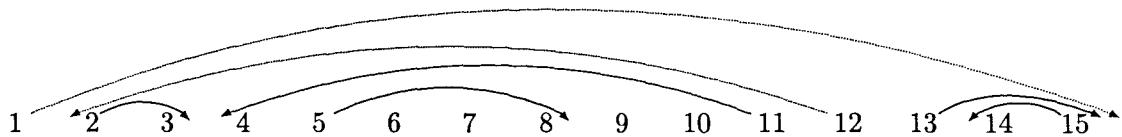
assuming that $|x| \leq 1$. Therefore,

$$U(x) = \frac{1 - 4x - x^3}{1 - 5x + 3x^2 + x^3 + x^4 + x^5}$$

Since the smallest root of $1 - 5x + 3x^2 + x^3 + x^4 + x^5 = 0$ is approximately $1/4.22$, the neighbourhood size is approximately $O(4.22^n)$.

8.9 Combined disjoint insert and nested insert neighbourhood

This neighbourhood simply consists of any set of disjoint and nested moves in any combination. There are no restrictions; any moves are allowed to be nested within any others, or disjoint to (outside of) any others.



Any block of moves is allowed to be nested within any other block of moves and any block of moves is allowed to be disjoint from any other block of moves. So a disjoint and nested block may be formed either by combining two disjoint and nested blocks using the disjoint block operator, or by performing a nested block operation on a disjoint and nested block.

$$\begin{aligned} B &= B^0 \\ B &\xrightarrow{N} (B) \\ B &\xrightarrow{D} (B, B) \end{aligned}$$

Dynamic programming algorithm

The combined disjoint insert and nested insert neighbourhood can be searched in $O(n^3)$ time. Let $\Delta(i, j)$ be the maximum increase in objective function achievable by performing a combined disjoint insert and nested insert move on the partial sequence $(\sigma(i), \dots, \sigma(j))$

Initialisation

$$\Delta(i, i) = 0 \text{ for } i = 1, \dots, n$$

Recursion

Calculate $\Delta(i, j)$ for $j - i = 1, \dots, n - 1$: where $1 \leq i < j \leq n$.

$$\Delta(i, j) = \max \begin{cases} \max_{i \leq k \leq j-1} \{\Delta(i, k) + \Delta(k+1, j)\} \\ \Delta(i, j-1) + \delta(j, i-1) \\ \Delta(i+1, j) + \delta(i, j). \end{cases}$$

$\Delta(1, n)$ is the largest improvement obtainable from a move in the neighbourhood. The move corresponding to the improvement $\Delta(1, n)$ is found by backtracking.

Neighbourhood size

The size of the neighbourhood is approximately $\Theta(5.36^n)$. We now give an outline of how the size U_n can be calculated.

Starting with the identity permutation $\sigma(1), \sigma(2), \dots, \sigma(n)$, we condition on insert moves involving element $\sigma(n)$.

- Given $\sigma(n)$ remains in position n then $U_{n-1} - 1$ neighbourhood moves are possible.

To obtain an element other than $\sigma(n)$ in position n , either $\sigma(n)$ must be the element removed from its current position to be inserted elsewhere in the sequence, or another element $\sigma(i)$ must be removed from its current position and inserted after $\sigma(n)$.

- Given $\sigma(n)$ is removed from its current position to be inserted after $\sigma(i)$ in the sequence $(M(n, i))$, the number of possible neighbourhood moves is $U_i U_{n-1-i}$, where $0 \leq i \leq n-2$.
- Given element $\sigma(i+1)$ is removed from its current position to be inserted after $\sigma(n)$ in the sequence $(M(i+1, n))$ and $\sigma(i+1)$ appears in position n , the number of possible neighbourhood moves is $U_i U_{n-1-i}$, where $0 \leq i \leq n-3$. Note that if $i = n-2$, then this is equivalent to swapping elements $\sigma(n-1)$ and $\sigma(n)$ and therefore equivalent to inserting $\sigma(n)$ after $\sigma(n-2)$, which has already been counted.

Summing up the different neighbourhood moves and adding the possibility of performing no moves, we obtain

$$U_n = U_{n-1} + U_{n-2} + 2 \sum_{i=0}^{n-3} U_i U_{n-1-i} \quad \text{for } n \geq 2,$$

where $U_0 = U_1 = 1$.

We now outline how the generating function $U(x)$ may be calculated.

$$\begin{aligned} U(x) &= \sum_{i=0}^{\infty} U_i x^i \\ &= U_0 + U_1 x + U_2 x^2 + U_3 x^3 + U_4 x^4 + U_5 x^5 + \dots \\ &= 1 + x(U_0) \\ &\quad + x^2(U_1 + U_0) \end{aligned}$$

$$\begin{aligned}
& + x^3(U_2 + U_1 + 2U_2U_0) \\
& + x^4(U_3 + U_2 + 2U_2U_1 + 2U_3U_0) \\
& + x^5(U_4 + U_3 + 2U_2U_2 + 2U_3U_1 + 2U_4U_0) + \dots \\
= & \quad 1 + xU(x) + x^2U(x) + 2xU^2(x) - 2xU(x)(U_0 + U_1x)
\end{aligned}$$

assuming that $|x| \leq 1$. Therefore,

$$2xU^2(x) - (1 + x + x^2)U(x) + 1 = 0$$

giving $U(x) = \frac{1 + x + x^2 \pm \sqrt{1 - 6x + 3x^2 + 2x^3 + x^4}}{4x}$

As explained in section 8.2, given the generating function above, we conjecture that the neighbourhood size is approximately $\Theta(5.36^n)$ since the smallest root of $1 - 6x + 3x^2 + 2x^3 + x^4 = 0$ is approximately equal to $1/5.36$.

8.10 Conclusion

We have shown how the neighbourhoods for the linear ordering problem given in Table 8.1 can be formed. Also we have provided a dynamic programming algorithm enabling each neighbourhood to be searched in polynomial time, and calculated the order of the size of each neighbourhood.

Type of insert neighbourhood	Complexity of DP	Size of neighbourhood	Section number
Disjoint	$O(n^2)$	$\Theta(2.21^n)$	8.5
Nested	$O(n^2)$	$\Theta((2 + \sqrt{3})^n)$	8.6
Nested inside disjoint	$O(n^2)$	$\Theta(4.06^n)$	8.7
Disjoint inside nested	$O(n^3)$	$\Theta(4.22^n)$	8.8
Combined disjoint and nested	$O(n^3)$		8.9

Table 8.1: PSEN for the LOP

It seems that the nested inside disjoint neighbourhood is worthy of further investigation, given that it contains both the disjoint and nested neighbourhoods and yet only requires $O(n^2)$ time to search. The other neighbourhood which may warrant future research is the combined disjoint and nested neighbourhood which whilst searchable in the same time complexity, $O(n^3)$, contains the disjoint inside nested neighbourhood.

The computational complexity of the dynamic program used to search the linear ordering neighbourhood is likely to be a reasonably accurate guide to the run time in practice because we are unaware of any effective speedups (see sub-section 6.2.4). It should be noted that this is in sharp contrast to the TSP for which there are a number of very effective speedups (see sub-section 7.2.1) which work on a range of neighbourhoods. The potential for the implementation of these speedups to the search algorithm for a new neighbourhood may therefore be a crucial component in determining the practical significance of the neighbourhood.

Chapter 9

Polynomially Searchable Exponential Neighbourhoods for the Travelling Salesman Problem

9.1 Introduction

This chapter is concerned with new polynomially searchable exponential neighbourhoods for the travelling salesman problem. A dynamic program is given for each neighbourhood, and in most cases the size of the neighbourhood is calculated. Within the sections we show that all bar the overlapping neighbourhoods are sub-neighbourhoods of the twisted sequence neighbourhood. A particular achievement is the calculation of the size of the twisted sequence neighbourhood. The non-rotational neighbourhood size is equal to the number of twisted sequences of a permutation; the value of which has remained open since the introduction of the sequences by Aurenhammer (1988).

Before introducing any new neighbourhoods, we use the next three sections to introduce the background material, by which we define and display many of our new neighbourhoods. In section 9.2 we introduce some new notation and give some definitions. In section 9.3 we give an alternative diagrammatic representation which aims to make the underlying structure of many of the new neighbourhoods more visible. In section 9.4 we define some blocks that will help us to define many of the neighbourhoods in sections 9.5 and 9.6. For further background we refer back to the first two sections of Chapter 8. An introduction to the concept giving rise to our new PSENs is given in section 8.1 and an introduction to the methodology by which we calculate the sizes of many of the new PSENs given in section 8.2.

Sections 9.5 and 9.6 of the current chapter contain neighbourhoods which can

be formed from simple combinations of nested and/or disjoint reversals. In section 9.5 the neighbourhoods are formed from independent reversals and are therefore dynasearch neighbourhoods (See section 4.4 in Chapter 4). Given a move M from a dynasearch neighbourhood is made up of reversals r_1, \dots, r_j , if the effect in terms of the change in objective function value, of any particular reversal r_i , when it is performed alone, is δ_{r_i} , then the effect of the combined move M is $\sum_{i=1}^j \delta_{r_i}$. The reversals are referred to as independent because the contribution of an individual reversal to the overall effect of the neighbourhood move is independent of any other reversals with which it may combine to form the move. In contrast, the reversals forming neighbourhoods in section 9.6 are not independent. The effect of any individual reversal on the objective function is dependent on the other reversals (if any) with which it is combined to form the neighbourhood move.

Sections 9.7 and 9.8 contain dynasearch neighbourhoods. Section 9.7 deals with neighbourhoods made up of nested or disjoint 3-opt moves. In section 9.8 we study overlapping dynasearch neighbourhoods where combinations of 2-opt and 3-opt moves are neither nested nor disjoint.

9.2 Some new notation and definitions

We define the sets:

$$S_{i,j} = \{i, \dots, j\} \text{ for } i \leq j \text{ and } S_{i,j} = \{i, \dots, n, 1, \dots, j\} \text{ for } i > j.$$

$$S_{i,j}^- = \{i+1, \dots, j-1\} \text{ for } i < j \text{ and } S_{i,j}^- = \{i+1, \dots, n, 1, \dots, j-1\} \text{ for } i > j.$$

Note that $S_{i,j}^-$ can be empty if $j = i + 1$

The rotations of a given permutation can be formed by iteratively removing the current first element of the permutation and adding it to the end of the permutation. There are n rotations of the permutation: $(\sigma(1), \sigma(2), \dots, \sigma(n))$, the current permutation itself and $n - 1$ others of the form:

$$(\sigma(i), \dots, \sigma(n), \sigma(1), \dots, \sigma(i-1)) \text{ for } i = 2, \dots, n.$$

The non-rotational neighbourhood of a given permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$ contains the subset of moves from the corresponding rotational neighbourhood which do not reverse the order of $\sigma(n)$ and $\sigma(1)$ (at least not without reversing the whole permutation which obviously has no effect on the tour).

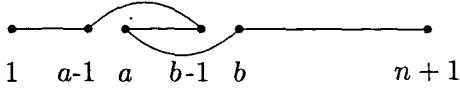
The moves contained in a rotational neighbourhood of a given permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$ are the superset of all of the moves in the non-rotational neighbourhoods of the n rotations of the permutation.

It is conventional to derive dynamic programming algorithms for the travelling salesman problem in terms of F , the length of the partial tour. However, for neighbourhoods based on k -opt moves we found it more convenient to use Δ , the reduction in length of the partial tour, although the computational complexity of the dynamic programming algorithms are the same. See section 7.3 for examples of dynamic programs written in both forms.

We now outline the notation used in this chapter, some of which has been used previously in Chapter 7.

Let $d_{\sigma(i), \sigma(j)}$ be the length of the edge joining city $\sigma(i)$ to city $\sigma(j)$ and $D_{i,j} = \sum_{k=i}^{j-1} d_{\sigma(k), \sigma(k+1)}$.

Let $M(a-1, b)$ refer to the 2-opt move, illustrated below, which breaks the edges $\{\sigma(a-1), \sigma(a)\}$ and $\{\sigma(b-1), \sigma(b)\}$, and let $\delta(a-1, b)$ be the reduction in tour length it produces. We can write

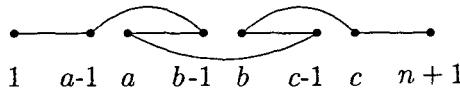


$$\delta(a-1, b) = d_{\sigma(a-1), \sigma(a)} + d_{\sigma(b-1), \sigma(b)} - d_{\sigma(a-1), \sigma(b-1)} - d_{\sigma(a), \sigma(b)}$$

for $|S_{a-1,b}| \geq 3$ and $\delta(a-1, b) = 0$ otherwise.

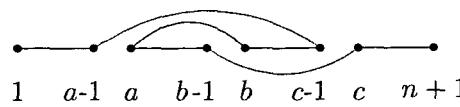
In the rest of the section, let $b \in |S_{a,c}^-|$.

Let $M(1; a-1, b, c)$ refer to the pure 3-opt move illustrated below, and let $\delta(1; a-1, b, c)$ be the reduction in tour length it produces.



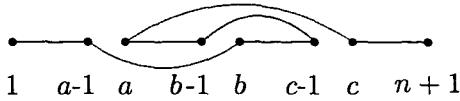
$$\begin{aligned} \delta(1; a-1, b, c) &= d_{\sigma(a-1), \sigma(a)} + d_{\sigma(b-1), \sigma(b)} + d_{\sigma(c-1), \sigma(c)} \\ &\quad - d_{\sigma(a-1), \sigma(b-1)} - d_{\sigma(a), \sigma(c-1)} - d_{\sigma(b), \sigma(c)} \end{aligned}$$

Let $M(2; a-1, b, c)$ refer to the pure 3-opt move illustrated below and let $\delta(2; a-1, b, c)$ be the reduction in tour length it produces.



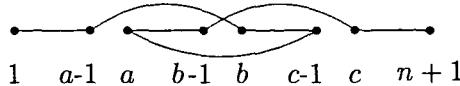
$$\begin{aligned} \delta(2; a-1, b, c) &= d_{\sigma(a-1), \sigma(a)} + d_{\sigma(b-1), \sigma(b)} + d_{\sigma(c-1), \sigma(c)} \\ &\quad - d_{\sigma(a-1), \sigma(c-1)} - d_{\sigma(b), \sigma(a)} - d_{\sigma(b-1), \sigma(c)} \end{aligned}$$

Let $M(3; a - 1, b, c)$ refer to the pure 3-opt move illustrated below and let $\delta(3; a - 1, b, c)$ be the reduction in tour length it produces.



$$\begin{aligned}\delta(3; a - 1, b, c) = & d_{\sigma(a-1), \sigma(a)} + d_{\sigma(b-1), \sigma(b)} + d_{\sigma(c-1), \sigma(c)} \\ & - d_{\sigma(a-1), \sigma(b)} - d_{\sigma(c-1), \sigma(b-1)} - d_{\sigma(a), \sigma(c)}\end{aligned}$$

Let $M(4; a - 1, b, c)$ refer to the pure 3-opt move illustrated below and let $\delta(4; a - 1, b, c)$ be the reduction in tour length it produces.



$$\begin{aligned}\delta(4; a - 1, b, c) = & d_{\sigma(a-1), \sigma(a)} + d_{\sigma(b-1), \sigma(b)} + d_{\sigma(c-1), \sigma(c)} \\ & - d_{\sigma(a-1), \sigma(b)} - d_{\sigma(c-1), \sigma(a)} - d_{\sigma(b-1), \sigma(c)}\end{aligned}$$

Let $\delta(a - 1, b, c)$ be the maximum improvement obtained by performing a pure 3-opt move involving removing the edges $\{\sigma(a - 1), \sigma(a)\}$, $\{\sigma(b - 1), \sigma(b)\}$ and $\{\sigma(c - 1), \sigma(c)\}$ on the current tour. Let

$$\begin{aligned}\delta(a - 1, b, c) = & d_{\sigma(a-1), \sigma(a)} + d_{\sigma(b-1), \sigma(b)} + d_{\sigma(c-1), \sigma(c)} \\ & - \min \left\{ \begin{array}{l} d_{\sigma(a-1), \sigma(b-1)} + d_{\sigma(a), \sigma(c-1)} + d_{\sigma(b), \sigma(c)} \\ d_{\sigma(a-1), \sigma(c-1)} + d_{\sigma(b), \sigma(a)} + d_{\sigma(b-1), \sigma(c)} \\ d_{\sigma(a-1), \sigma(b)} + d_{\sigma(c-1), \sigma(b-1)} + d_{\sigma(a), \sigma(c)} \\ d_{\sigma(a-1), \sigma(b)} + d_{\sigma(c-1), \sigma(a)} + d_{\sigma(b-1), \sigma(c)} \end{array} \right\} \\ = & \max_{1 \leq i \leq 4} \delta(i; a - 1, b, c)\end{aligned}$$

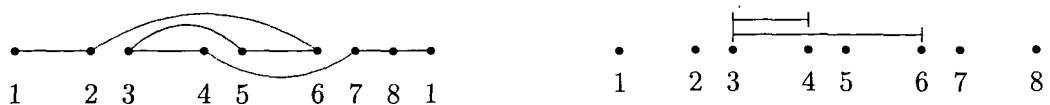
for $|S_{a-1,c}| \geq 5$ and $\delta(a - 1, b, c) = 0$ otherwise.

9.3 An alternative diagrammatic representation of some neighbourhood moves

We now introduce a useful alternative way of representing a 2-opt or 3-opt move. The 2-opt move illustrated below removes the edges $\{\sigma(a-1), \sigma(a)\}$ and $\{\sigma(b-1), \sigma(b)\}$ and replaces them with edges $\{\sigma(a-1), \sigma(b-1)\}$ and $\{\sigma(a), \sigma(b)\}$. The effect of this move is to reverse the section of tour $(\sigma(a), \sigma(a+1), \dots, \sigma(b-2), \sigma(b-1))$. The alternative representation consists of a list of the cities in their current order, with a line above the cities whose order is reversed by the 2-opt move. Both representations of the 2-opt move are illustrated below:



For clarity, before giving an example of how a more general 3-opt move can be represented, we give a specific example. The effect of the $M(2; 2, 5, 7)$ move illustrated below is equivalent to reversing two sections of tour, one contained within the other. Consider a tour with the order of the cities corresponding to the permutation σ : $(\sigma(1), \sigma(2), \sigma(3), \sigma(4), \sigma(5), \sigma(6), \sigma(7), \sigma(8))$. Firstly the order of the cities $(\sigma(3), \sigma(4))$ is reversed producing the permutation: $(\sigma(1), \sigma(2), \sigma(4), \sigma(3), \sigma(5), \sigma(6), \sigma(7), \sigma(8))$. Then the order of the cities $(\sigma(4), \sigma(3), \sigma(5), \sigma(6))$ is reversed forming the corresponding tour to the permutation: $(\sigma(1), \sigma(2), \sigma(6), \sigma(5), \sigma(3), \sigma(4), \sigma(7), \sigma(8))$.



The $M(2; 2, 5, 7)$ move is illustrated above both by the traditional diagram and the alternative diagram where the two lines signify the sections of tour which are reversed by the move.

We now give a more general example of how a 3-opt move may be represented. The effect of the $M(2; a - 1, b, c)$ move illustrated below is equivalent to reversing two sections of tour, one contained within the other. Given a tour with the order of the cities corresponding to the permutation:

$$(\sigma(1), \dots, \sigma(a-1), \sigma(a), \sigma(a+1), \dots, \sigma(b-2), \sigma(b-1), \sigma(b), \sigma(b+1), \dots \\ \dots, \sigma(c-2), \sigma(c-1), \sigma(c), \dots, \sigma(n)).$$

Firstly the order of the cities $(\sigma(a), \sigma(a+1), \dots, \sigma(b-2), \sigma(b-1))$ is reversed producing the permutation:

$$(\sigma(1), \dots, \sigma(a-1), \sigma(b-1), \sigma(b-2), \dots, \sigma(a+1), \sigma(a), \sigma(b), \sigma(b+1), \dots \\ \dots, \sigma(c-2), \sigma(c-1), \sigma(c), \dots, \sigma(n)).$$

Then the order of the cities $(\sigma(b-1), \sigma(b-2), \dots, \sigma(a+1), \sigma(a), \sigma(b), \sigma(b+1), \dots, \sigma(c-2), \sigma(c-1))$ is reversed producing the permutation:

$$(\sigma(1), \dots, \sigma(a-1), \sigma(c-1), \sigma(c-2), \dots, \sigma(b+1), \sigma(b), \sigma(a), \sigma(a+1), \dots \\ \dots, \sigma(b-2), \sigma(b-1), \sigma(c), \dots, \sigma(n)).$$



The $M(2; a - 1, b, c)$ move is illustrated above both by the traditional diagram and the alternative diagram where the two lines signify the sections of tour which are reversed by the move.

In diagrams where lines are used to indicate the sections of tour reversed, we follow the convention of placing shorter lines above the longer lines. To determine the effect on the order of the cities in the tour of a move represented by the diagram, we then perform the reversals indicated by the highest (shortest) lines first and systematically move down. The effect of nested reversals is in practice easier to determine by following this procedure.

9.4 Some block definitions used to define neighbourhoods

9.4.1 The blocks used to define the non-rotational strict reversal neighbourhoods

In order to form neighbourhoods using the concept of blocks and block operations, we need to define both an initial block on which we can perform block operations and the block operations themselves.

We will use the block B_S^0 defined below as the initial block. The block B_S^0 representing a strict reversal move from the underlying neighbourhood is such that:

$$B_S^0 = (\sigma(i), \dots, \sigma(j-1), \sigma(k), \sigma(k-1), \dots, \sigma(j+1), \sigma(j), \sigma(k+1), \dots, \sigma(l))$$

for $1 \leq i < j < k < l \leq n$.

Let $B^C \xrightarrow{D_S} (B^A, B^B)$ refer to the strictly disjoint block operation combining disjoint blocks B^A and B^B to form a single block of type B^C . Given the particular blocks B^A and B^B are $B^A = (\sigma'(j), \dots, \sigma'(k))$, $B^B = (\sigma'(l), \dots, \sigma'(m))$

$$\begin{aligned} B^C &= (\sigma(i), \dots, \sigma(j-1), B^A, \sigma(k+1), \dots, \sigma(l-1), B^B, \sigma(m+1), \dots, \sigma(p)) \\ &= (\sigma(i), \dots, \sigma(j-1), \sigma'(j), \dots, \sigma'(k), \sigma(k+1), \dots, \sigma(l-1), \sigma'(l), \dots, \sigma'(m), \sigma(m+1), \dots, \sigma(p)). \end{aligned}$$

for $1 \leq i \leq j \leq k \leq l \leq m \leq p \leq n$.

Let $B^B \xrightarrow{N_S} (B^A)$ refer to the strictly nested block operation on block B^A to form B^B . Then given $B^A = (\sigma'(j), \dots, \sigma'(k))$,

$$\begin{aligned} B^B &= (\sigma(i), \dots, \sigma(j-1), & B^A \text{ reversed} &, \sigma(k+1), \dots, \sigma(l)) \\ &= (\sigma(i), \dots, \sigma(j-1), \sigma'(k), \sigma'(k-1), \dots, \sigma'(j+1), \sigma'(j), \sigma(k+1), \dots, \sigma(l)) \\ \text{for } 1 \leq i < j &\leq k < l \leq n. \end{aligned}$$

9.4.2 The blocks used to define the rotational strict reversal neighbourhoods

We will use the block B_{SR}^0 defined below as the initial block. The block B_{SR}^0 representing a strict reversal move from the underlying neighbourhood is such that:

$$B_{SR}^0 = B_{SR}^1 \cup B_{SR}^2 \cup B_{SR}^3 \cup B_{SR}^4$$

where

$$B_{SR}^1 = (\sigma(i), \dots, \sigma(j-1), \sigma(k), \sigma(k-1), \dots, \sigma(j+1), \sigma(j), \sigma(k+1), \dots, \sigma(l))$$

for $1 \leq i < j \leq k < l \leq n$.

$$B_{SR}^2 = (\sigma(i), \dots, \sigma(n), \sigma(1), \dots, \sigma(j-1), \sigma(k), \sigma(k-1), \dots, \sigma(j+1), \sigma(j),$$

$$\sigma(k+1), \dots, \sigma(l))$$

for $1 \leq j \leq k < l \leq i \leq n$.

$$B_{SR}^3 = (\sigma(i), \dots, \sigma(j-1), \sigma(k), \sigma(k-1), \dots, \sigma(1), \sigma(n), \dots, \sigma(j+1), \sigma(j),$$

$$\sigma(k+1), \dots, \sigma(l))$$

for $1 \leq k < l \leq i < j \leq n$.

$$B_{SR}^4 = (\sigma(i), \dots, \sigma(j-1), \sigma(k), \sigma(k-1), \dots, \sigma(j+1), \sigma(j), \sigma(k+1), \dots, \sigma(n),$$

$$\sigma(1), \dots, \sigma(l))$$

for $1 \leq l \leq i < j \leq k \leq n$.

Let $B^C \xrightarrow{D_{SR}} (B^A, B^B)$ refer to the strictly disjoint rotational block operation combining disjoint blocks B^A and B^B to form a single block of type B^C . Then given the particular blocks B^A , B^B and B^C involve the sets of cities $S_{j,k}$, $S_{l,m}$ and $S_{i,p}$, respectively, the relations $S_{j,k} \subset S_{i,p}$, $S_{l,m} \subset S_{i,p}$, $S_{j,k} \cap S_{l,m}^- = \emptyset$, must hold.

Let $B^B \xrightarrow{N_{SR}} (B^A)$ refer to the strictly nested rotational block operation on block B^A to form B^B . Then given the particular blocks B^A and B^B involve the sets of cities $S_{j,k}$ and $S_{i,l}$, respectively, the relation $S_{j,k} \subseteq S_{i,l}^-$ must hold.

9.4.3 The blocks used to define the non-rotational reversal neighbourhoods

We will use the block B^0 defined below as the initial block. The block B^0 representing a reversal move from the underlying neighbourhood is such that:

$$B^0 = (\sigma(i), \dots, \sigma(j-1), \sigma(k), \sigma(k-1), \dots, \sigma(j+1), \sigma(j), \sigma(k+1), \dots, \sigma(l))$$

for $1 \leq i \leq j \leq k \leq l \leq n$.

Let $B^C \xrightarrow{D} (B^A, B^B)$ refer to the disjoint block operation combining disjoint blocks B^A and B^B to form a single block of type B^C . Given the particular blocks B^A and B^B are $B^A = (\sigma'(j), \dots, \sigma'(k))$, $B^B = (\sigma'(l), \dots, \sigma'(m))$

$$B^C = (\sigma(i), \dots, \sigma(j-1), B^A, \sigma(k+1), \dots, \sigma(l-1), B^B, \sigma(m+1), \dots, \sigma(p))$$

$$= (\sigma(i), \dots, \sigma(j-1), \sigma'(j), \dots, \sigma'(k), \sigma(k+1), \dots, \sigma(l-1), \sigma'(l), \dots, \sigma'(m), \sigma(m+1), \dots, \sigma(p))$$

for $1 \leq i \leq j \leq k < l \leq m \leq p \leq n$.

Let $B^B \xrightarrow{N} (B^A)$ refer to the nested block operation on block B^A to form B^B .

Then given $B^A = (\sigma'(j), \dots, \sigma'(k))$,

$$B^B = (\sigma(i), \dots, \sigma(j-1), \quad \text{\scriptsize B^A reversed} \quad , \sigma(k+1), \dots, \sigma(l))$$

$$B^B = (\sigma(i), \dots, \sigma(j-1), \sigma'(k), \sigma'(k-1), \dots, \sigma'(j+1), \sigma'(j), \sigma(k+1), \dots, \sigma(l))$$

for $1 \leq i \leq j \leq k \leq l \leq n$.

9.4.4 The blocks used to define the rotational reversal neighbourhoods

We will use the block B_R^0 defined below as the initial block. The block B_R^0 representing a reversal move from the underlying neighbourhood is such that:

$$B_R^0 = B_R^1 \cup B_R^2 \cup B_R^3 \cup B_R^4$$

where

$$B_R^1 = (\sigma(i), \dots, \sigma(j-1), \sigma(k), \sigma(k-1), \dots, \sigma(j+1), \sigma(j), \sigma(k+1), \dots, \sigma(l))$$

for $1 \leq i \leq j \leq k \leq l \leq n$.

$$B_R^2 = (\sigma(i), \dots, \sigma(n), \sigma(1), \dots, \sigma(j-1), \sigma(k), \sigma(k-1), \dots, \sigma(j+1), \sigma(j),$$

$$\sigma(k+1), \dots, \sigma(l))$$

for $1 \leq j \leq k \leq l \leq i \leq n$.

$$B_R^3 = (\sigma(i), \dots, \sigma(j-1), \sigma(k), \sigma(k-1), \dots, \sigma(1), \sigma(n), \dots, \sigma(j+1), \sigma(j),$$

$$\sigma(k+1), \dots, \sigma(l))$$

for $1 \leq k \leq l \leq i \leq j \leq n$.

$$B_R^4 = (\sigma(i), \dots, \sigma(j-1), \sigma(k), \sigma(k-1), \dots, \sigma(j+1), \sigma(j), \sigma(k+1), \dots, \sigma(n),$$

$$\sigma(1), \dots, \sigma(l))$$

for $1 \leq l \leq i \leq j \leq k \leq n$.

Let $B^C \xrightarrow{D_R} (B^A, B^B)$ refer to the disjoint rotational block operation combining disjoint blocks B^A and B^B to form a single block of type B^C . Then given the particular blocks B^A , B^B and B^C involve the sets of cities $S_{j,k}$, $S_{l,m}$ and $S_{i,p}$, respectively, the relations $S_{j,k} \subset S_{i,p}$, $S_{l,m} \subset S_{i,p}$, $S_{j,k} \cap S_{l,m} = \emptyset$, must hold.

Let $B^B \xrightarrow{N_R} (B^A)$ refer to the nested rotational block operation on block B^A to form B^B . Then given the particular blocks B^A and B^B involve the sets of cities $S_{j,k}$ and $S_{i,l}$, respectively, the relation $S_{j,k} \subseteq S_{i,l}$ must hold.

9.5 PSEN based on strictly disjoint and strictly nested reversals

In the strict neighbourhoods the following two statements must always be true:

- Between any two disjoint reversals there must be a gap of at least two edges (i.e. at least one city which is not part of either reversal).
- Any nested reversal must be completely contained by and not start or finish at the same position as the reversal in which it is nested.

9.5.1 Disjoint 2-opt

The non-rotational disjoint 2-opt neighbourhood consists of a combination of strictly disjoint reversals on the given permutation representing the tour. The neighbourhood is non-rotational so no single reversal can involve both $\sigma(1)$ and $\sigma(n)$ and because it is strictly disjoint the sets of cities involved in any pair of reversals must have at least one city between them which is not involved in either reversal.

This neighbourhood has previously been implemented in Chapter 7 and referred to as dynasearch 2-opt or ds-2-opt.

The figure below contains an example of a disjoint 2-opt move consisting of the three 2-opt moves: $M(1, 7)$, $M(8, 13)$ and $M(13, 1)$. The effect of the three moves is to reverse the sections of tour $(\sigma(2), \dots, \sigma(6))$, $(\sigma(9), \dots, \sigma(12))$ and $(\sigma(14), \sigma(15))$. Note that all of the sections of tour reversed have at least one city between them and none of them contain both $\sigma(1)$ and $\sigma(15)$.



The disjoint 2-opt (non-rotational strictly disjoint) neighbourhood can be formed by exclusively applying strictly disjoint block operations on strictly disjoint blocks.

$$B_S^D = B_S^0$$

$$B_S^D \xrightarrow{D_S} (B_S^D, B_S^D)$$

where the block B_S^0 and the block operation $\xrightarrow{D_S}$ are defined in section 9.4.1.

Dynamic programming algorithm

The disjoint 2-opt neighbourhood can be searched in $O(n^2)$ time. Let $\Delta(j)$ be the maximum reduction in length of the path from city $\sigma(1)$ to $\sigma(j)$ achievable by performing disjoint 2-opt moves on the path $(\sigma(1), \dots, \sigma(j))$.

Initialisation

$$\Delta(i) = 0 \text{ for } i \leq 3.$$

Recursion

Calculate $\Delta(j)$ for $j = 4, \dots, n + 1$:

$$\Delta(j) = \max \left\{ \begin{array}{l} \Delta(j-1), \\ \max_{1 \leq i \leq j-3} \{\Delta(i) + \delta(i, j)\} \end{array} \right.$$

where $\delta(i, j)$ is defined in section 9.2.

Optimal reduction in solution value

$$\Delta(n+1).$$

Neighbourhood size

The size of the neighbourhood is $\Theta(1.755^n)$. We now give an outline of how the size U_n of the neighbourhood can be calculated.

When conditioning on a reversal it is useful to consider one city after the reversal to be connected to it. Starting with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$, we condition on reversals if any connected to $\sigma(n)$:

- Given $\sigma(n)$ is not connected to a reversal $U_{n-1} - 1$ different neighbourhood moves are possible.
- Given the reversal of the section of tour $(\sigma(i+1), \dots, \sigma(n-1))$ the number of possible neighbourhood moves is U_i , where $0 \leq i \leq n-3$. (No nested moves are allowed, so no reversals can occur within the section of the tour $(\sigma(i+1), \dots, \sigma(n-1))$.)

Summing up the different neighbourhood moves and adding the null move of performing no reversals, we obtain

$$U_n = U_{n-1} + \sum_{i=0}^{n-3} U_i \quad \text{for } n \geq 1,$$

where $U_0 = 1$.

The corresponding generating function is $U(x) = \sum_{i=0}^{\infty} U_i x^i = \frac{(1-x)}{1-2x+x^2-x^3}$

Since the smallest root of $1 - 2x + x^2 - x^3 = 0$ is approximately $1/1.755$, the neighbourhood size is $\Theta(1.755^n)$

9.5.2 Rotational disjoint 2-opt

The rotational disjoint 2-opt neighbourhood consists of a combination of strictly disjoint reversals, and can be formed by exclusively using strictly disjoint rotational block operations.

The figure below contains an example of a rotational nested 2-opt move consisting of the three 2-opt moves: $M(5,8)$, $M(8,13)$ and $M(13,4)$.



A strictly disjoint rotational block may be formed by performing a strictly disjoint rotational block operation on a pair of strictly disjoint rotational blocks.

$$B_{SR}^D = B_{SR}^0$$

$$B_{SR}^D \xrightarrow{D_{SR}} (B_{SR}^D, B_{SR}^D)$$

where the block B_{SR}^0 and the block operation $\xrightarrow{D_{SR}}$ are defined in section 9.4.2.

Dynamic programming algorithm

The rotational disjoint 2-opt neighbourhood can be searched in $O(n^3)$ time. This can be done either by implementing the algorithm for the disjoint 2-opt neighbourhood n times where each of the cities is fixed in turn; or by the following dynamic programming algorithm.

Let $\Delta(i, j)$ be the maximum reduction in length achievable by performing disjoint 2-opt moves exclusively involving the set of cities $S_{i,j}$, where cities $\sigma(i)$ and $\sigma(j)$ remain in positions i and j respectively.

Initialisation

$$\Delta(i, i) = 0 \text{ for } i = 1, \dots, n.$$

Recursion

Calculate $\Delta(i, j)$ for increasing $|S_{i,j}|$ until $\Delta(i+1, i)$ is known for $1 \leq i \leq n$.

$$\Delta(i, j) = \max_{k \in S_{i,j} - \{j\}} \{\Delta(i, k) + \delta(k, j)\}.$$

Optimal reduction in solution value

$$\max_{1 \leq i \leq n-1} \Delta(i+1, i).$$

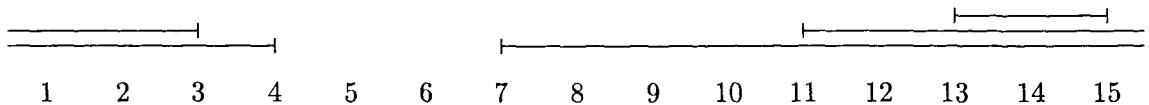
Neighbourhood size

The rotational disjoint 2-opt neighbourhood is at least as large, and at most a factor of n larger than the disjoint 2-opt neighbourhood. Thus, its size is $\Omega(2^n)$ and $O(n2^n)$.

9.5.3 Rotational nested 2-opt

The rotational nested 2-opt neighbourhood consists of a combination of reversals strictly nested within each other, and can be formed by exclusively using strictly nested rotational block operations.

The figure below contains an example of a rotational nested 2-opt move consisting of the three 2-opt moves: $M(6, 5)$, $M(10, 4)$ and $M(12, 1)$.



A strictly nested rotational block may be formed by performing a strictly nested rotational block operation on a strictly nested rotational block.

$$B_{SR}^N = B_{SR}^0$$

$$B_{SR}^N \xrightarrow{N_{SR}} (B_{SR}^N)$$

where the block B_{SR}^0 and the block operation $\xrightarrow{N_{SR}}$ are defined in section 9.4.2.

The computational complexity of searching the nested 2-opt neighbourhood is the same as computational complexity of searching the rotational nested 2-opt neighbourhood. So only the the rotational nested 2-opt neighbourhood is considered.

Dynamic programming algorithm

The rotational nested 2-opt neighbourhood can be searched in $O(n^2)$ time. Let $\Delta(i, j)$ be the maximum reduction in length achievable by performing nested 2-opt moves exclusively involving the set of cities $S_{i,j}$, where cities $\sigma(i)$ and $\sigma(j)$ remain in positions i and j respectively.

Initialisation

$$\Delta(i, i) = \Delta(i, i + 1) = 0 \text{ for } i = 1, \dots, n.$$

Recursion

Calculate $\Delta(i, j)$ for increasing $|S_{i,j}|$ until $\Delta(i + 1, i)$ is known for $1 \leq i \leq n$.

$$\Delta(i, j) = \max \begin{cases} \Delta(i, j - 1) \\ \Delta(i + 1, j) \\ \Delta(i + 1, j - 1) + \delta(i, j). \end{cases}$$

Optimal reduction in solution value

$$\max_{1 \leq i \leq n} \Delta(i+1, i).$$

Neighbourhood size

The size of the neighbourhood is $\Omega(2^n)$ and $O(n2^n)$. We now give an outline of how the size U_n of the non-rotational version of the neighbourhood can be calculated.

As before with other neighbourhoods built up of 2-opt moves when conditioning on a reversal it is useful to consider one city after the reversal to be connected to it.

Starting with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$, we condition on reversals connected to $\sigma(n)$:

- Given $\sigma(n)$ is not connected to a reversal, $U_{n-1} - 1$ different neighbourhood moves are possible.
- Given the reversal of the section of tour $(\sigma(n-i-1), \dots, \sigma(n-1))$ the number of possible neighbourhood moves is U_i , where $1 \leq i \leq n-2$. (Only nested moves are allowed so reversals can only occur within the section of the tour $(\sigma(n-i-1), \dots, \sigma(n-1))$.

Summing up the different neighbourhood moves and adding the null move of performing no reversals, we obtain

$$U_n = \sum_{i=1}^{n-1} U_i \quad \text{for } n \geq 2,$$

where $U_1 = 1$.

The corresponding generating function is $U(x) = \sum_{i=1}^{\infty} U_i x^i = \frac{x(1-x)}{1-2x}$

Since the only root of $1-2x=0$ is $1/2$, the non-rotational nested neighbourhood size is $\Theta(2^n)$. Therefore the rotational nested neighbourhood size is $\Omega(2^n)$ and $O(n2^n)$.

9.5.4 Restricted nested 2-opt

All moves in a Restricted nested 2-opt neighbourhood must be centred around the edge $n, 1$ which can't be broken. Only 2-opt moves of the form $M(n - i + 1, i)$ can be performed; in effect any section $n - i + 2, i - 1$ can be reversed where $i \leq n/2$.

The figure below contains an example of a restricted nested 2-opt move consisting of the four 2-opt moves: $M(9, 7)$, $M(10, 6)$, $M(11, 5)$ and $M(13, 3)$.



Dynamic programming algorithm

The restricted nested 2-opt neighbourhood can be searched in $O(n)$ time. Let $\Delta(j)$ be the maximum reduction in length achievable by performing restricted nested 2-opt moves on the path $(\sigma(n - j + 1), \dots, \sigma(j))$.

Initialisation

$$\Delta(1) = 0.$$

Recursion

Calculate $\Delta(i)$ for $i = 2, \dots, \lfloor \frac{n}{2} \rfloor$.

$$\Delta(i) = \Delta(i-1) + \max \begin{cases} \delta(n-i+1, i) \\ 0. \end{cases}$$

Optimal reduction in solution value

$$\Delta(\lfloor \frac{n}{2} \rfloor).$$

Neighbourhood size

A move in the neighbourhood is made up of any subset of $\lfloor n/2 \rfloor - 1$ possible reversals, and therefore the size of the neighbourhood is $\Theta(2^{n/2})$.

9.5.5 Nested 2-opt inside disjoint 2-opt

The nested 2-opt inside disjoint 2-opt consists of a combination of strictly nested reversals within strictly disjoint reversals. Only nested moves are allowed within the disjoint moves, thus enabling the best nested neighbourhood move to be found for each partial sequence as a preprocessing step. Consequently, there is a reduction in computational complexity as compared with the more general combined disjoint and nested neighbourhood (see section 9.5.8).

The figure below contains an example of a nested 2-opt inside disjoint 2-opt move consisting of the five 2-opt moves: $M(1, 7)$, $M(3, 6)$, $M(8, 13)$, $M(9, 12)$ and $M(13, 1)$.



Firstly we need to form a non-rotational strictly nested block.

$$\begin{aligned} B_S^N &= B_S^0 \\ B_S^N &\xrightarrow{N_S} (B_S^N) \end{aligned}$$

where the block B_S^0 and the block operation $\xrightarrow{N_S}$ are defined in section 9.4.1. Now using the non-rotational strictly nested block we can form the non-rotational strictly nested inside disjoint block as follows. The strictly nested inside disjoint block can be formed using the strictly disjoint block operation on a strictly nested inside disjoint block and a strictly nested block.

$$\begin{aligned} B &= B_S^N \\ B &\xrightarrow{D_S} (B, B_S^N) \end{aligned}$$

where the block operation $\xrightarrow{D_S}$ is defined in section 9.4.1.

Dynamic programming algorithm

The nested 2-opt inside disjoint 2-opt neighbourhood can be searched in $O(n^2)$ time. The first step is to search the Nested 2-opt neighbourhood in $O(n^2)$ (see section 9.5.3) as a preprocessing step. Finding $\Delta^N(i, j)$ for $1 \leq i < j \leq n$ the maximum improvement possible due to a nested 2-opt move on the path $(\sigma(i), \dots, \sigma(j))$, where

cities $\sigma(i)$ and $\sigma(j)$ remain in their original positions. Then the nested 2-opt inside disjoint 2-opt can be found in $O(n^2)$ time as follows.

Let $\Delta(j)$ be the maximum reduction in length achievable by performing nested 2-opt inside disjoint 2-opt moves on the path $(\sigma(1), \dots, \sigma(j))$.

Initialisation

$$\Delta(1) = 0.$$

Recursion

Calculate $\Delta(j)$ for $j = 1, \dots, n + 1$

$$\Delta(j) = \max_{1 \leq i \leq j-1} \{\Delta(i) + \Delta^N(i, j)\}.$$

Optimal reduction in solution value

$$\Delta(n + 1).$$

Neighbourhood size

The size of the nested 2-opt inside non-rotational disjoint 2-opt neighbourhood is $\Theta(2.206^n)$. Let N_n be the size of the nested 2-opt neighbourhood for a problem instance of size n (see section 9.5.3). So $N_n = \sum_{i=1}^{n-1} U_i$ and the generating function of N_n is $N(x) = x(1-x)/(1-2x)$

As before with other neighbourhoods built up of 2-opt moves, when conditioning on a reversal it is useful to consider one city after the reversal to be connected to it. We now give an outline of how the size U_n of the neighbourhood can be calculated.

Starting with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$, we condition on reversals connected to $\sigma(n)$:

- Given $\sigma(n)$ is not connected to a reversal $U_{n-1} - 1$ different neighbourhood moves are possible.
- Given the reversal of the section of tour $(\sigma(i+1), \dots, \sigma(n-1))$ the number of possible neighbourhood moves is $U_i N_{n-i-2}$, where $0 \leq i \leq n-3$.

Summing up the different neighbourhood moves and adding the null move of performing no reversals, we obtain

$$U_n = U_{n-1} + \sum_{i=0}^{n-3} U_i N_{n-i-2} \quad \text{for } n \geq 1,$$

where $U_0 = 1$.

The corresponding generating function is $U(x) = \sum_{i=0}^{\infty} U_i x^i = \frac{1 - 2x}{(1 - x)(1 - 2x - x^3)}$

Since the smallest root of $(1 - x)(1 - 2x - x^3) = 0$ is approximately 1/2.206, the nested 2-opt inside non-rotational disjoint 2-opt neighbourhood size is $\Theta(2.206^n)$.

9.5.6 Nested 2-opt inside rotational disjoint 2-opt

The nested 2-opt inside rotational disjoint 2-opt consists of a combination of strictly nested rotational reversals within strictly disjoint rotational reversals. This neighbourhood is the rotational version of the nested 2-opt inside disjoint 2-opt neighbourhood (section 9.5.5). Only rotational nested moves are allowed within the rotational disjoint moves, thus enabling the best nested neighbourhood move to be found for each partial sequence as a preprocessing step. Consequently, there may be (although there is not in this case) a reduction in computational complexity as compared with the more general combined disjoint and nested neighbourhood (see section 9.5.8).

The figure below contains an example of a nested 2-opt inside rotational disjoint 2-opt move consisting of the five 2-opt moves: $M(5, 8)$, $M(8, 13)$, $M(9, 12)$, $M(13, 4)$ and $M(14, 3)$. The rotational strictly nested inside disjoint block can be



formed using the rotational strictly disjoint block operation on a rotational strictly nested inside disjoint block and a rotational strictly nested block. The rotational strictly nested block was defined in section 9.5.3.

$$B = B_{SR}^N$$

$$B \xrightarrow{D_{SR}} (B, B_{SR}^N)$$

where the block B_{SR}^N and the block operation $\xrightarrow{D_{SR}}$ are defined in sections 9.5.3 and 9.4.2, respectively.

Dynamic programming algorithm

The nested 2-opt inside rotational disjoint 2-opt neighbourhood can be searched in $O(n^3)$ time. This can be done either by implementing the algorithm for the nested 2-opt inside disjoint 2-opt neighbourhood n times, starting with a different rotation of the original permutation each time, or by the following dynamic programming algorithm. Firstly we evaluate $\Delta^N(i, j)$ for $1 \leq i, j \leq n$ by searching the rotational nested 2-opt neighbourhood as in section 9.5.3. Then the nested 2-opt inside rotational disjoint 2-opt can be found in $O(n^3)$ time as follows:

Let $\Delta(i, j)$ be the maximum reduction in length achievable by performing nested 2-opt inside rotational disjoint 2-opt moves exclusively involving the set of cities $S_{i,j}$, where cities $\sigma(i)$ and $\sigma(j)$ remain in positions i and j respectively..

Initialisation

$$\Delta(i, i) = 0 \text{ for } i = 1, \dots, n.$$

Recursion

Calculate $\Delta(i, j)$ for increasing $|S_{i,j}|$ until $\Delta(i+1, i)$ is known for $1 \leq i \leq n$.

$$\Delta(i, j) = \max_{k \in S_{i,j} - \{j\}} \{\Delta(i, k) + \Delta^N(k, j)\}.$$

Optimal reduction in solution value

$$\max_{1 \leq i \leq n} \Delta(i+1, i).$$

Neighbourhood size

The size of the nested 2-opt inside rotational disjoint 2-opt neighbourhood is at least as large as, and at most a factor of n larger than the nested 2-opt inside disjoint 2-opt neighbourhood. Hence, the size of the nested 2-opt inside rotational disjoint 2-opt neighbourhood is $\Omega(2.206^n)$ and $O(n2.206^n)$.

9.5.7 Disjoint 2-opt inside rotational nested 2-opt

The disjoint 2-opt inside rotational nested 2-opt neighbourhood is an extension of the nested 2-opt neighbourhood in exactly the same way as the nested 2-opt inside disjoint 2-opt neighbourhood is an extension of the disjoint 2-opt neighbourhood.

Only strictly disjoint reversals are allowed on the parts of the tour which are unaffected by the strictly nested reversals, although these disjoint reversals may be nested within future blocks, thus enabling the best disjoint 2-opt neighbourhood move to be found for each section of tour as a preprocessing step. Consequently, there may be (although there is not in this particular case) a reduction in computational complexity as compared with the more general combined disjoint 2-opt and nested 2-opt neighbourhood see (section 9.5.8).

The figure below contains an example of a disjoint 2-opt inside rotational nested 2-opt move consisting of the five 2-opt moves: $M(5, 8)$, $M(8, 5)$, $M(9, 13)$, $M(13, 4)$ and $M(14, 2)$.



The strictly disjoint inside rotational nested block can be formed either by performing a strictly disjoint rotational block operation on a strictly disjoint inside rotational nested block and a strictly rotational disjoint block, or by performing a strictly nested rotational block operation on a strictly rotational disjoint inside nested block.

$$\begin{aligned} B &= B_{SR}^0 \\ B &\xrightarrow{N_{SR}} (B) \\ B &\xrightarrow{D_{SR}} (B, B_{SR}^D) \end{aligned}$$

where the block B_{SR}^N and the block operations $\xrightarrow{N_{SR}}$ and $\xrightarrow{D_{SR}}$ are defined in section 9.4.2, and the block B_{SR}^D is defined in 9.5.2.

Dynamic programming algorithm

The rotational disjoint 2-opt inside nested 2-opt neighbourhood can be searched in $O(n^3)$ time. Firstly we find $\Delta^D(i, j)$ for $1 \leq i, j \leq n$ by searching the rotational

disjoint 2-opt neighbourhood as in section 9.5.2.

Let $\Delta(i, j)$ be the maximum reduction in length achievable by performing disjoint 2-opt inside rotational nested 2-opt moves exclusively involving the set of cities $S_{i,j}$, where cities $\sigma(i)$ and $\sigma(j)$ remain in positions i and j respectively..

Initialisation

$$\Delta(i, i) = \Delta(i, i + 1) = 0 \quad \text{for } i = 1, \dots, n.$$

Recursion

Calculate $\Delta(i, j)$ for increasing $|S_{i,j}|$ until $\Delta(i + 1, i)$ is known for $1 \leq i \leq n$.

$$\Delta(i, j) = \max \left\{ \begin{array}{l} \max_{k \in S_{i,j}^-} \left\{ \begin{array}{l} \Delta(i, k) + \Delta^D(k, j) \\ \Delta^D(i, k) + \Delta(k, j) \end{array} \right. \\ \Delta(i + 1, j - 1) + \delta(i, j). \end{array} \right.$$

Optimal reduction in solution value

$$\max_{1 \leq i \leq n} \Delta(i + 1, i).$$

Neighbourhood size

The size of the disjoint 2-opt inside rotational nested 2-opt neighbourhood is $\Omega(2.35^n)$ and $O(n2.35^n)$. We now give an outline of how the size U_n of the non-rotational version neighbourhood can be calculated.

Let D_n be the size of the disjoint 2-opt neighbourhood for a problem instance of size n (see section 9.5.1). So $D_n = D_{n-1} + \sum_{i=0}^{n-3} D_i$ and the generating function of

$$D_n \text{ is } D(x) = \sum_{i=0}^{\infty} D_i x^i = (1 - x)/(1 - 2x + x^2 - x^3).$$

As before with the other neighbourhoods built up of 2-opt moves, when conditioning on a reversal it is useful to consider one city after the reversal to be connected to it.

Starting with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$, we condition on reversals connected to $\sigma(n)$:

- Given $\sigma(n)$ is not connected to a reversal, $U_{n-1} - 1$ different neighbourhood moves are possible.

- Given the reversal of the section of tour $(\sigma(i+1), \dots, \sigma(n-1))$ and that there are no reversals nested within it, then nesting is allowed outside the section and the number of possible neighbourhood moves is U_i , where $0 \leq i \leq n-3$.
- Given the reversal of the section of tour $(\sigma(i+1), \dots, \sigma(n-1))$ and that there are reversals nested within it, then no nesting is allowed outside the section and the number of possible neighbourhood moves is $D_i(U_{n-i-2} - 1)$, where $0 \leq i \leq n-3$.

Summing up the different neighbourhood moves and adding the null move of performing no reversals, we obtain

$$U_n = U_{n-1} + \sum_{i=0}^{n-3} (U_i + D_i(U_{n-i-2} - 1)) \quad \text{for } n \geq 1,$$

where $U_0 = 1$. The corresponding generating function is

$$U(x) = \sum_{i=0}^{\infty} U_i x^i = \frac{1 - 3x + 2x^2 - x^3 + x^4}{1 - 4x + 5x^2 - 4x^3 + 4x^4 - 2x^5 + x^6}$$

Since the smallest root of $1 - 4x + 5x^2 - 4x^3 + 4x^4 - 2x^5 + x^6 = 0$ is approximately $1/2.35$, the non-rotational neighbourhood size is $\Theta(2.35^n)$. The size of the rotational neighbourhood is at least as large, and at most a factor of n larger than the non-rotational neighbourhood. Hence, the size of the disjoint 2-opt inside rotational nested 2-opt neighbourhood is $\Omega(2.35^n)$ and $O(n2.35^n)$.

9.5.8 Rotational combined disjoint 2-opt and nested 2-opt

This neighbourhood simply consists of any set of disjoint and nested 2-opt moves in any combination. There are no restrictions; any moves are allowed to be nested within any others, or disjoint to (outside of) any others.

The figure below contains an example of a rotational combined disjoint 2-opt and nested 2-opt move consisting of the five 2-opt moves: $M(6, 5)$, $M(8, 13)$, $M(9, 12)$, $M(13, 4)$ and $M(15, 3)$.



Any block of moves is allowed to be nested within any other block of moves and any block of moves is allowed to be disjoint from any other block of moves. So a rotational strictly disjoint reverse and nested reverse block may be formed either by combining two rotational strictly disjoint reverse and nested reverse blocks using the rotational strictly disjoint block operator, or by performing a rotational strictly nested block operation on a rotational strictly disjoint reverse and nested reverse block.

$$B = B_{SR}^0$$

$$B \xrightarrow{N_{SR}} (B)$$

$$B \xrightarrow{D_{SR}} (B, B)$$

where the block B_{SR}^N and the block operations $\xrightarrow{N_{SR}}$ and $\xrightarrow{D_{SR}}$ are defined in section 9.4.2.

Dynamic programming algorithm

The rotational combined disjoint 2-opt and nested 2-opt neighbourhood can be searched in $O(n^3)$ time. Let $\Delta(i, j)$ be the maximum reduction in length by performing rotational combined disjoint 2-opt and nested 2-opt neighbourhood moves exclusively involving the set of cities $S_{i,j}$, where cities $\sigma(i)$ and $\sigma(j)$ remain in positions i and j respectively.

Initialisation

$$\Delta(i, i) = \Delta(i, i + 1) = 0 \text{ for } i = 1, \dots, n.$$

Recursion

Calculate $\Delta(i, j)$ for increasing $|S_{i,j}|$ until $\Delta(i + 1, i)$ is known for $1 \leq i \leq n$.

$$\Delta(i, j) = \max \left\{ \begin{array}{l} \delta(i, j) + \Delta(i + 1, j - 1) \\ \max_{k \in S_{i,j}^-} \{\Delta(i, k) + \Delta(k, j)\} \end{array} \right\}.$$

Optimal reduction in solution value

$$\max_{1 \leq i \leq n} \Delta(i + 1, i).$$

Neighbourhood size

The size of the neighbourhood is conjectured to be $O(2.61^n)$. We now give an outline of how the size U_n of the non-rotational neighbourhood can be calculated.

To produce a combined disjoint and nested move any pair of 2-opt moves in the neighbourhood must not involve the breaking the same edge. In light of this when conditioning on a reversal it is useful to consider one city after the reversal to be connected to it.

Starting with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$, we condition on reversals connected to $\sigma(n)$:

- Given $\sigma(n)$ is not connected to a reversal, $U_{n-1} - 1$ different neighbourhood moves are possible.
 - Given the reversal of the section of tour $(\sigma(i+1), \dots, \sigma(n-1))$, the number of possible neighbourhood moves is $U_i U_{n-i-2}$, where $0 \leq i \leq n-3$.
- Note if $(\sigma(j), \dots, \sigma(k-1))$ is reversed then city $\sigma(k)$ is said to be connected to that reversal.

Summing up the different neighbourhood moves and adding the null move of performing no reversals, we obtain

$$U_n = U_{n-1} + \sum_{i=0}^{n-3} U_i U_{n-2-i} \quad \text{for } n \geq 1,$$

where $U_0 = 1$. The corresponding generating function is

$$U(x) = \sum_{i=0}^{\infty} U_i x^i = \frac{x^2 - x + 1 \pm \sqrt{x^4 - 2x^3 - x^2 - 2x + 1}}{2x^2}$$

As explained in section 8.2, given the generating function above, we conjecture that the size of the non-rotational neighbourhood is $\Theta(2.61^n)$ since the smallest root of $1+x^4-2x^3-x^2-2x=0$ is approximately equal to $1/2.61$. The size of the rotational neighbourhood is at least as large, and at most a factor of n larger than the non-rotational neighbourhood. Hence, the size of the rotational combined disjoint 2-opt and nested 2-opt neighbourhood is conjectured to be $\Omega(2.61^n)$ and $O(n2.61^n)$.

9.6 PSEN based on disjoint and nested reversals.

In this section unlike the last, there does not need to be a gap of at least two edges between any pair of disjoint reversals and a nested reversal can start or finish at the same position as the reversal in which it is nested. Trivially for any neighbourhood in this section the corresponding strict neighbourhood, if one exists, forms a sub-neighbourhood (i.e. is contained within it).

In this section, where the neighbourhoods are based on reversal moves which may interact, we found it more convenient to follow the more conventional approach of deriving the dynamic programming algorithms in terms of F .

9.6.1 Disjoint reverse

The non-rotational disjoint 2-opt neighbourhood consists of a combination of disjoint reversals on the given permutation representing the tour.

The figure below contains an example of a disjoint reverse move.



The disjoint reverse (non-rotational disjoint) neighbourhood can be formed by performing a disjoint block operation on a pair of disjoint blocks.

$$B^D = B^0$$

$$B^D \xrightarrow{D} (B^D, B^D)$$

where the block B^0 and the block operation \xrightarrow{D} are defined in section 9.4.3.

Dynamic programming algorithm

The dynamic programming algorithm below for searching the the disjoint reverse neighbourhood requires $O(n^3)$ time.

Let $\sigma(1) = 1$, i.e. let $\sigma(1)$ be fixed in the first position. Let $F(j, \sigma(i))$ be the shortest path up to position j which ends with city $\sigma(i)$, where $1 < i \leq j$, allowing any disjoint reversals to be performed that exclusively involve the first j cities except for city $\sigma(1)$.

Initialisation

$$F(1, \sigma(1)) = 0 \quad F(2, \sigma(2)) = d_{\sigma(1), \sigma(2)}.$$

Recursion

Calculate F values for $j = 3, \dots, n$ where $2 \leq i \leq j$

$$F(j, \sigma(i)) = \min_{\min\{i-1, 2\} \leq k < i} \{F(i-1, \sigma(k)) + d_{\sigma(k), \sigma(j)} + D_{i,j}\}.$$

Optimal solution value

$$\min_{2 \leq j \leq n} \{F(n, \sigma(j)) + d_{\sigma(1), \sigma(j)}\}.$$

Neighbourhood size

The neighbourhood size is $\Theta(2^n)$. We now give an outline of how the size U_n of the neighbourhood can be calculated. Start with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$. Condition on the section of tour reversed if any involving city $\sigma(n)$ in any possible neighbourhood move:

- Given that $\sigma(n)$ is not involved in any reversals then $U_{n-1} - 1$ neighbourhood moves are possible.
- Given that $\sigma(n)$ is involved in the reversal of the section of tour between $\sigma(i+1)$ and $\sigma(n)$ inclusively, namely $(\sigma(i+1), \dots, \sigma(n))$, then U_i different neighbourhood moves are possible, where $0 \leq i \leq n-2$.

So summing up the different possible moves and adding the null move of performing no reversals, we obtain

$$U_n = \sum_{i=0}^{n-1} U_i \quad \text{for } n \geq 2,$$

where $U_0 = U_1 = 1$.

The corresponding generating function is $U(x) = \sum_{i=0}^{\infty} U_i x^i = (1-x)/(1-2x)$

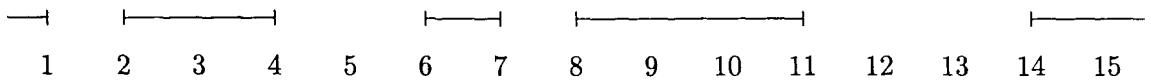
Since the only root of $1-2x=0$ is $1/2$, the neighbourhood size is $\Theta(2^n)$.

Strictly U_n is always larger by 2 than it should be because reversing the interval $(\sigma(1), \dots, \sigma(n))$ or $(\sigma(2), \dots, \sigma(n))$ does not affect the tour. However, the size of the neighbourhood in terms of its order is unaffected and remains $\Theta(2^n)$.

9.6.2 Rotational disjoint reverse

The rotational disjoint reverse neighbourhood consists of a combination of disjoint reversals, and can be formed by exclusively using disjoint rotational block operations.

The figure below contains an example of a rotational disjoint reverse move.



A disjoint rotational block may be formed by performing a disjoint rotational block operation on a pair of disjoint rotational blocks.

$$B_R^D = B_R^0$$

$$B_R^D \xrightarrow{D_R} (B_R^D, B_R^D)$$

where the block B_R^0 and the block operation $\xrightarrow{D_R}$ are defined in section 9.4.4.

Dynamic programming algorithm

The rotational disjoint reverse neighbourhood can be searched in $O(n^4)$ time. This can be done either by implementing the algorithm for the disjoint reverse neighbourhood n times where each of the cities is fixed in turn; or by another dynamic programming algorithm where there are four state variables, of which two signify the interval over which the shortest partial tour in the neighbourhood has been found and the other two represent the cities at the end points of the interval, $F(i, j; \sigma(k)\sigma(l))$.

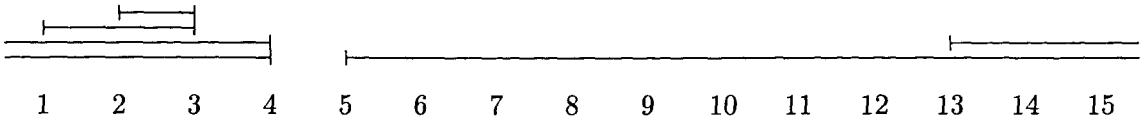
Neighbourhood size

The rotational disjoint reverse neighbourhood at least as large, and at most a factor of n larger than the disjoint reverse neighbourhood. Thus, its size is $\Omega(2^n)$ and $O(n2^n)$.

9.6.3 Rotational nested reverse

The rotational nested reverse neighbourhood consists of a combination of reversals nested within each other, and can be formed by exclusively using nested rotational block operations.

The figure below contains an example of a rotational nested reverse move.



A nested rotational block may be formed by performing a nested rotational block operation on a nested rotational block.

$$B_R^N = B_R^0$$

$$B_R^N \xrightarrow{N_R} (B_R^N)$$

where the block B_R^0 and the block operation $\xrightarrow{N_R}$ are defined in section 9.4.4.

The computational complexity of searching the nested reverse neighbourhood is the same as computational complexity of searching the rotational nested reverse neighbourhood. Consequently, only the the rotational nested reverse neighbourhood is considered.

Dynamic programming algorithm

The dynamic programming algorithm below for searching the the rotational nested reverse neighbourhood requires $O(n^3)$ time.

Let $F(i, j; \sigma(i), \sigma(m))$ for $m \in S_{i,j} - \{i\}$ and $F(i, j; \sigma(m), \sigma(j))$ for $m \in S_{i,j} - \{j\}$ be the length of a shortest path exclusively involving the cities $S_{i,j}$, beginning and ending with cities $\sigma(i)$ and $\sigma(m)$, and with $\sigma(m)$ and $\sigma(j)$, respectively.

Initialisation

$$F(i, i+1; \sigma(i), \sigma(i+1)) = d_{\sigma(i), \sigma(i+1)} \text{ for } i = 1, \dots, n.$$

Recursion

Calculate F values for increasing $|S_{i,j}|$ until $F(i+1, i; \sigma(i+1), \sigma(i))$ is known for $1 \leq i \leq n$

$$F(i, j; \sigma(i), \sigma(j)) = \min \left\{ \begin{array}{l} \min_{a \in S_{i,j}^-} \{ F(i, j-1; \sigma(i), \sigma(a)) + d_{\sigma(a), \sigma(j)} \} \\ \min_{a \in S_{i,j}^+} \{ F(i+1, j; \sigma(a), \sigma(j)) + d_{\sigma(i), \sigma(a)} \} \end{array} \right.$$

For $a \in S_{i,j}^-$,

$$F(i, j; \sigma(i), \sigma(a)) = \min \left\{ \begin{array}{l} F(i+1, j; \sigma(i+1), \sigma(a)) + d_{\sigma(i), \sigma(i+1)} \\ F(i+1, j; \sigma(a), \sigma(j)) + d_{\sigma(i), \sigma(j)} \end{array} \right.$$

$$F(i, j; \sigma(a), \sigma(j)) = \min \left\{ \begin{array}{l} F(i, j-1; \sigma(a), \sigma(j-1)) + d_{\sigma(j-1), \sigma(j)} \\ F(i, j-1; \sigma(i), \sigma(a)) + d_{\sigma(i), \sigma(j)}. \end{array} \right.$$

Optimal solution value

$$\min \left\{ \begin{array}{l} \min_{\substack{1 \leq a \leq n \\ a \neq i+1}} \{ F(i+1, i; \sigma(i+1), \sigma(a)) + d_{\sigma(i+1), \sigma(a)} \} \\ \min_{\substack{1 \leq a \leq n \\ a \neq i}} \{ F(i+1, i; \sigma(a), \sigma(i)) + d_{\sigma(a), \sigma(i)} \}. \end{array} \right.$$

To search the nested reverse neighbourhood the final step reduces to

$$\min \left\{ \begin{array}{l} \min_{1 \leq a \leq n} \{ F(1, n; \sigma(1), \sigma(a)) + d_{\sigma(1), \sigma(a)} \} \\ \min_{1 \leq a < n} \{ F(1, n; \sigma(a), \sigma(n)) + d_{\sigma(a), \sigma(n)} \}, \end{array} \right.$$

but overall the computational complexity is unchanged.

Neighbourhood size

We now give an outline of how the size U_n of the nested reverse neighbourhood can be calculated. Start with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$. Conditioning on the largest reversal that city $\sigma(n)$ is involved in, within any possible neighbourhood move:

- Given $\sigma(n)$ is not involved in a reversal, then $U_{n-1} - 1$ neighbourhood moves are possible.
- Given that the largest reversal involving city $\sigma(n)$ is the reversal of the tour between $\sigma(n-i+1)$ and $\sigma(n)$, i.e. $(\sigma(n-i+1), \dots, \sigma(n))$, then $(1/2)U_i$ neighbourhood moves are possible. Note the half is because the decision to reverse $(\sigma(n-i+1), \dots, \sigma(n))$ has already been made. This argument holds for $2 \leq i \leq n$.

So summing up the different possible moves and adding the null move of performing no reversals, we obtain

$$\begin{aligned} U_n &= U_{n-1} + (1/2) \sum_{i=2}^n U_i \\ &= 3U_{n-1} + \sum_{i=2}^{n-2} U_i \quad \text{for } n \geq 3 \end{aligned}$$

where $U_2 = 2$.

The corresponding generating function is $U(x) = \sum_{i=2}^{\infty} U_i x^i = \frac{2x^2(1-x)}{1-4x+2x^2}$

Since the smallest root of $1 - 4x + 2x^2 = 0$ is $1/(2 + \sqrt{2})$, the size of the nested reverse neighbourhood is $\Theta((2 + \sqrt{2})^n)$.

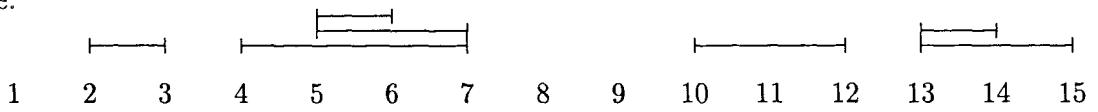
The rotational nested reverse neighbourhood is at least as large as the nested reverse neighbourhood and at most a factor of n times larger. So the size of the rotational nested reverse neighbourhood is $\Omega((2 + \sqrt{2})^n)$ and $O(n(2 + \sqrt{2})^n)$.

The Carlier-Villon neighbourhood which also takes $O(n^3)$ to search is a sub-neighbourhood of rotational nested reverse neighbourhood and is significantly smaller only $\Theta(n2^n)$ as compared to $\Omega((2 + \sqrt{2})^n)$.

9.6.4 Nested reverse inside disjoint reverse

The nested reverse inside disjoint reverse neighbourhood consists of a combination of nested reversals within disjoint reversals. Only nested moves are allowed within the disjoint moves, thus enabling the best nested neighbourhood move to be found for each partial sequence as a preprocessing step. Consequently, there is a reduction in computational complexity as compared with the more general combined disjoint reverse and nested reverse neighbourhood (twisted sequence neighbourhood see section 9.6.7).

The figure below contains an example of a nested reverse inside disjoint reverse move.



Firstly, we need to form a non-rotational nested block.

$$B^N = B^0$$

$$B^N \xrightarrow{N} (B^N)$$

where the block B^0 and the block operation \xrightarrow{N} are defined in section 9.4.3. Now using the non-rotational nested block, we can form the non-rotational nested inside disjoint block. The nested inside disjoint block can be formed using the disjoint block operation on a nested inside disjoint block and a nested block.

$$B = B^N$$

$$B \xrightarrow{D} (B, B^N)$$

where the block operation \xrightarrow{D} is defined in section 9.4.3.

Dynamic programming algorithm

Firstly, the dynamic programming algorithm for the nested reverse neighbourhood is used to calculate F values for all possible intervals in $O(n^3)$ time. Then the algorithm below can be used to search the nested reverse inside disjoint reverse neighbourhood in $O(n^4)$ time. So overall the complexity of searching the nested reverse inside disjoint reverse neighbourhood is $O(n^4)$.

Let $\sigma(1) = 1$, i.e. let $\sigma(1)$ be fixed in the first position. Let $F'(j, \sigma(i))$ be the shortest path involving the cities $(\sigma(1), \dots, \sigma(j))$ which ends with city $\sigma(i)$, allowing any reversals in the neighbourhood to be performed that exclusively involve the first j cities except for city $\sigma(1)$.

Initialisation

$$F'(1; \sigma(1)) = 0.$$

Recursion

Calculate $F'(j; \sigma(j))$ and $F'(j; \sigma(i))$ iteratively for $j = 2, \dots, n$ and for $1 < i < j$

$$F'(j; \sigma(j)) = \min_{\min\{j-1, 2\} \leq l < j} \left\{ F'(j-1; \sigma(l)) + d_{\sigma(l), \sigma(j)} \right\}$$

$$F'(j; \sigma(i)) = \min \begin{cases} \min_{\substack{\min\{i-1, 2\} \leq l < i \\ i < k < j}} \left\{ F'(i-1; \sigma(l)) + d_{\sigma(l), \sigma(k)} + F(i, j; \sigma(i), \sigma(k)) \right\} \\ \min_{\substack{1 < k \leq i \\ \min\{k-1, 2\} \leq l < k}} \left\{ F'(k-1; \sigma(l)) + d_{\sigma(l), \sigma(j)} + F(k, j; \sigma(i), \sigma(j)) \right\}. \end{cases}$$

Optimal solution value

$$\min_{2 \leq j \leq n} \left\{ F'(n, \sigma(j)) + d_{\sigma(j), \sigma(1)} \right\}.$$

Neighbourhood size

We now give an outline of how the size U_n of the nested reverse inside disjoint reverse neighbourhood can be calculated. Let N_n be the size of the nested reverse neighbourhood (see section 9.6.3) so $N_n = 3N_{n-1} + \sum_{i=2}^{n-2} N_i$ and $N(x) = \sum_{i=2}^{\infty} N_i x^i$. $N(x) = 2x^2(1-x)/(1-4x+2x^2)$. Start with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$. Condition on the section of tour reversed if any involving city $\sigma(n)$ in any possible neighbourhood move:

- Given that $\sigma(n)$ is not involved in any reversals then $U_{n-1} - 1$ neighbourhood moves are possible.
- Given that the largest reversal involving $\sigma(n)$ is the reversal of the section of tour $(\sigma(i+1), \dots, \sigma(n))$ then $(1/2)N_{n-i}U_i$ different neighbourhood moves are possible, where $0 \leq i \leq n-2$. Note only nested moves are allowed inside the reversal, any of $(1/2)N_{n-i}$ possibilities exist, whereas outside the reversal any of U_i different moves can be made.

So summing up the different possible moves and adding the null move of performing no reversals, we obtain $U_n = \sum_{i=0}^{n-2} (1/2)U_i N_{n-i} + U_{n-1}$ for $n \geq 1$, where $U_0 = 1$.

The corresponding generating function is $U(x) = \sum_{i=0}^{\infty} U_i x^i = \frac{1-4x+2x^2}{1-5x+5x^2-x^3}$ and the neighbourhood size is $\Theta((2+\sqrt{3})^n)$

9.6.5 Nested reverse inside rotational disjoint reverse

The nested reverse inside rotational disjoint reverse neighbourhood consists of a combination of nested reversals within rotational disjoint reversals. Only nested moves are allowed within the rotational disjoint moves, thus enabling the best nested neighbourhood move to be found for each partial sequence as a preprocessing step. Consequently, there is a reduction in computational complexity as compared with the more general combined disjoint reverse and nested reverse neighbourhood (twisted sequence neighbourhood see section 9.6.7).

The figure below contains an example of a nested reverse inside rotational disjoint reverse move.



The rotational nested inside disjoint block can be formed using the disjoint block operation on a rotational nested inside disjoint block and a rotational nested block. The rotational nested block was defined in section 9.6.3.

$$\begin{aligned} B &= B_R^N \\ B &\xrightarrow{D_R} (B, B_R^N) \end{aligned}$$

where the block B_R^N and the block operation $\xrightarrow{D_R}$ are defined in sections 9.6.3 and 9.4.4, respectively.

Dynamic programming algorithm

As with the rotational disjoint reverse neighbourhood, the nested reverse inside rotational disjoint reverse neighbourhood can be searched in $O(n^5)$ time, either by implementing the algorithm for the nested reverse inside disjoint reverse neighbourhood n times where each of the cities is fixed in turn, or by a specific dynamic programming algorithm.

Neighbourhood size

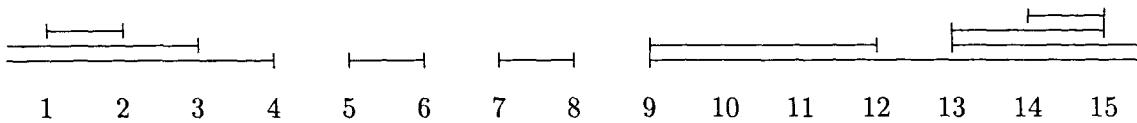
The size of the rotational neighbourhood is at least as large, and at most factor of n larger than the non-rotational neighbourhood. Hence, the size of the nested reverse inside rotational disjoint reverse neighbourhood is $\Omega(2 + \sqrt{3})^n$ and $O(n(2 + \sqrt{3})^n)$.

9.6.6 Disjoint reverse inside rotational nested reverse

The disjoint reverse inside rotational nested reverse neighbourhood is an extension of the nested reverse neighbourhood in exactly the same way as the nested reverse inside disjoint reverse neighbourhood is an extension of the disjoint reverse neighbourhood.

Only disjoint reversals are allowed on the parts of the tour which are unaffected by the rotational nested reversals, although these disjoint reversals may be nested within future blocks, thus enabling the best disjoint reverse neighbourhood move to be found for each section of tour as a preprocessing step. Consequently, there is a reduction in computational complexity as compared with the more general combined disjoint reverse and nested reverse neighbourhood (twisted sequence neighbourhood see section 9.6.7).

The figure below contains an example of a disjoint reverse inside rotational nested reverse move.



The disjoint inside rotational nested block can be formed either by performing a disjoint rotational block operation on a disjoint inside rotational nested block and a rotational disjoint block, or by performing a nested rotational block operation on a rotational disjoint inside nested block.

$$\begin{aligned} B &= B_R^0 \\ B &\xrightarrow{N_R} (B) \\ B &\xrightarrow{D_R} (B, B_R^D) \end{aligned}$$

where the block B_R^0 and the block operations $\xrightarrow{N_R}$ and $\xrightarrow{D_R}$ are defined in section 9.4.4, and the block B_R^D is defined in section 9.6.2.

The moves in the diagram which are effectively disjoint reverse moves are the reversals (5,6), (7,8), (9,10,11,12) and (1,2) in which there is no further nesting.

Dynamic programming algorithm

The dynamic programming algorithm below calculates the neighbourhood in $O(n^5)$ time. Let $F(i, j; \sigma(k), \sigma(l))$ for $k \in S_{i,j}$, $l \in S_{i,j} - \{k\}$ and $k < l$ be the length of a shortest path exclusively involving the cities $S_{i,j}$, beginning and ending with cities $\sigma(k)$ and $\sigma(l)$. We only need to consider $k < l$ because $F(i, j; \sigma(k), \sigma(l)) = F(i, j; \sigma(l), \sigma(k))$.

Initialisation

$$F(i, i; \sigma(i), \sigma(i)) = 0 \text{ for } i = 1, \dots, n.$$

Recursion

Calculate F values for increasing $|S_{i,j}|$ until $F(j+1, j; \sigma(k), \sigma(l))$ is known for all $1 \leq j, k, l \leq n$ and $k < l$.

$$F(i, j; \sigma(k), \sigma(l)) = \min \left\{ \begin{array}{l} \min_{m \in S_{i,l} - \{l,k\}} \{ F(i, l-1; \sigma(k), \sigma(m)) + d_{\sigma(m), \sigma(j)} + D_{l,j} \} \\ \min_{m \in S_{k,j} - \{k,l\}} \{ F(k+1, j; \sigma(m), \sigma(l)) + d_{\sigma(m), \sigma(i)} + D_{i,k} \} \end{array} \right\}.$$

Optimal solution value

$$\min_{\substack{1 \leq j \leq n \\ 1 \leq k < l \leq n}} \{ F(j+1, j; \sigma(k), \sigma(l)) + d_{\sigma(k), \sigma(l)} \}.$$

We only need to consider $k < l$ because $F(j+1, j; \sigma(k), \sigma(l)) = F(j+1, j; \sigma(l), \sigma(k))$.

Neighbourhood size

We now give an outline of how the size U_n of the disjoint reverse inside nested reverse neighbourhood can be calculated.

Let D_n be the size of the disjoint reverse neighbourhood (see section 9.6.1) so $D_n = \sum_{i=0}^{n-1} D_i$ and $D(x) = \sum_{i=0}^{\infty} D_i x^i = (1-x)/(1-2x)$. Start with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$. Condition on the section of tour reversed if any involving city $\sigma(n)$ in any possible neighbourhood move:

- Given that $\sigma(n)$ is not involved in any reversals then $U_{n-1} - 1$ neighbourhood moves are possible.
- Given that the largest reversal $\sigma(n)$ is involved in is the reversal of the section of tour $(\sigma(i+1), \dots, \sigma(n))$, and there is no nested reversal within this section,

then U_i different neighbourhood moves are possible, where $0 \leq i \leq n - 2$.

- Given that the largest reversal $\sigma(n)$ is involved in is the reversal of the section of tour $(\sigma(i+1), \dots, \sigma(n))$, and there is at least one nested reversal within this section, then $D_i((1/2)U_{n-i} - 1)$ different neighbourhood moves are possible. Note, only disjoint moves are allowed outside the reversal so D_i moves are possible outside the reversal. However, inside the reversal any moves within the disjoint reverse inside nested reverse neighbourhood are allowed so $(1/2)U_{n-i} - 1$ different moves are possible: $(1/2)U_{n-i}$ because the section has already been reversed and -1 because at least one reversal must occur inside the interval. A reversal can be performed inside the interval $(\sigma(i+1), \dots, \sigma(n))$ for $0 \leq i \leq n - 3$.

So summing up the different possible moves and adding the null move of performing no reversals, we obtain

$$\begin{aligned} U_n &= \sum_{i=0}^{n-3} D_i((1/2)U_{n-i} - 1) + \sum_{i=0}^{n-1} U_i \\ &= \sum_{i=1}^{n-3} D_i(U_{n-i} - 2) + 2 \sum_{i=0}^{n-1} U_i - 2 \quad \text{for } n \geq 1, \end{aligned}$$

where $U_0 = 1$.

The corresponding generating function is $U(x) = \sum_{i=0}^{\infty} U_i x^i = \frac{1 - 5x + 3x^2 + 2x^3 - x^4}{1 - 6x + 7x^2}$

Since the smallest root of $1 - 6x + 7x^2 = 0$ is $1/(3 + \sqrt{2})$, the size of disjoint reverse inside nested reverse neighbourhood is $\Theta((3 + \sqrt{2})^n)$. The size of the rotational neighbourhood is at least as large, and at most factor of n larger than the non-rotational neighbourhood. Hence, the size of the disjoint reverse inside rotational nested reverse neighbourhood is $\Omega((3 + \sqrt{2})^n)$ and $O(n(3 + \sqrt{2})^n)$.

9.6.7 The rotational twisted sequence neighbourhood

Twisted sequences were introduced by Aurenhammer (1988). However, it was not until 1997 that Deňeko & Woeginger (1997) proposed using the sequences to form a neighbourhood (see section 4.2.3). The rotational twisted sequence (rotational combined disjoint reverse and nested reverse) neighbourhood contains all of the neighbourhoods covered thus far in this chapter. Unfortunately, the dynamic program to search the neighbourhood has a higher computational complexity than any of the preceding neighbourhoods.

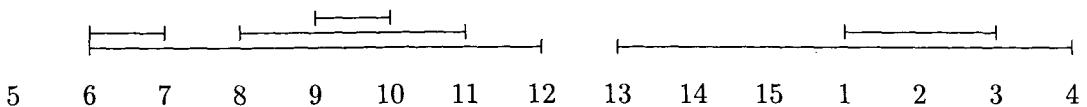
Any solution in the twisted sequence neighbourhood of a given permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$ can be obtained by reversing (twisting) a set of intervals over $(\sigma(1), \sigma(2), \dots, \sigma(n))$ in which a pair of intervals in the set are either disjoint or one is nested within the other.

Any block of moves is allowed to be nested within any other block of moves and any block of moves is allowed to be disjoint from any other block of moves. So a twisted sequence block (combined disjoint and nested block) may be formed by combining two twisted sequence blocks using the disjoint block operator, or by performing a nested block operation on a twisted sequence block.

$$\begin{aligned} B &= B^0 \\ B &\xrightarrow{N} (B) \\ B &\xrightarrow{D} (B, B) \end{aligned}$$

where the block B^0 and the block operations \xrightarrow{N} and \xrightarrow{D} are defined in section 9.4.3.

For example: the permutation $(5, 12, 8, 10, 9, 11, 6, 7, 4, 1, 2, 3, 15, 14, 13)$ lies in the twisted sequence neighbourhood of the permutation $(5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 1, 2, 3, 4)$ as illustrated in the figure below.



Deňeko & Woeginger 1997 provide a dynamic programming algorithm for searching the neighbourhood in $O(n^7)$.

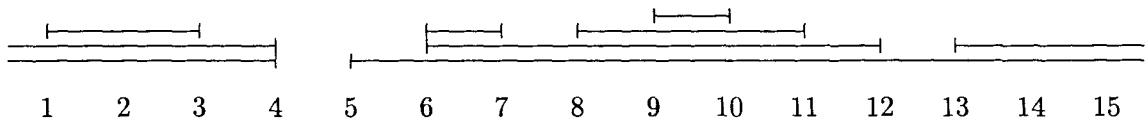
The dynamic programming algorithm is capable of searching all the twisted sequence neighbours, including the rotations of the current permutation. Therefore,

the rotated twisted sequence neighbourhood can also be searched in $O(n^7)$ time. The rotated twisted sequence neighbourhood is formally defined below.

$$\begin{aligned} B &= B_R^0 \\ B &\xrightarrow{N_R} (B) \\ B &\xrightarrow{D_R} (B, B) \end{aligned}$$

where the block B_R^0 and the block operations $\xrightarrow{N_R}$ and $\xrightarrow{D_R}$ are defined in section 9.4.4.

An example of a neighbour in the rotational twisted sequence neighbourhood of the permutation $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)$ is the permutation $(5, 12, 8, 10, 9, 11, 6, 7, 4, 1, 2, 3, 15, 14, 13)$ as illustrated in the figure below. Note that this would not be a neighbour in the standard twisted sequence.



Neighbourhood size

Deineko & Woeginger (1997) state that the twisted sequence neighbourhood size is $\Omega(2^n)$ and $O(6^n)$. An outline of how we calculated the size U_n of the twisted sequence neighbourhood is below.

Starting with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$. Conditioning on the section of tour reversed if any involving city $\sigma(n)$ in any possible neighbourhood move:

- Given that $\sigma(n)$ is not involved in any reversals then $U_{n-1} - 1$ neighbourhood moves are possible.
- Given that the largest reversal $\sigma(n)$ is involved in the reversal of the section of tour $(\sigma(i+1), \dots, \sigma(n))$ then $(1/2)U_{n-i}U_i$ different neighbourhood moves are possible, where $0 \leq i \leq n-2$.

So summing up the different possible moves and adding the null move of performing no reversals, we obtain $U_n = \sum_{i=0}^{n-2} (1/2)U_iU_{n-i} + U_{n-1}$ for $n \geq 2$, where $U_0 = U_1 = 1$. Therefore,

$$\begin{aligned}
U_n &= U_{n-1} + (1/2)U_n + (1/2) \sum_{i=2}^{n-1} U_i U_{n-i} \\
&= 2U_{n-1} + \sum_{i=2}^{n-1} U_i U_{n-i} \\
&= U_{n-1} + \sum_{i=1}^{n-1} U_i U_{n-i}
\end{aligned}$$

So $U_n = U_{n-1} + \sum_{i=1}^{n-1} U_i U_{n-i}$ for $n \geq 2$,
where $U_1 = 1$.

The sequence R_n where $R_n = U_{n+1}$ for $n \geq 0$, is known as “Royal paths in a lattice”. The sequence “Royal paths in a lattice” is connected to the better known sequence, $C_n = (1/2)R_n$ for $n \geq 1$, where C_n is the sequence known as Schröder’s second problem (1870) or the super Catalan numbers. Note that $C_0 = R_0 = 1$. Schröder (1870) gives the limit of the ratio of two consecutive terms in the second problem sequence as $3 + 2\sqrt{2}$, but gives no simple recurrence relationship for the sequence. Motzkin (1948) gives the following recurrence relationship for the sequence:

$$C_{n+1} = C_0 C_n + 2(C_1 C_{n-1} + \dots + C_n C_0) \quad \text{for } n \geq 0 \quad \text{where } C_0 = 1$$

and states that the limit for C_{n+1}/C_n , as n tends to infinity, is $3 + 2^{3/2}$. The recurrence relationship given enables us to show how the sequences are connected.

$$\begin{aligned}
U_n &= U_{n-1} + \sum_{i=1}^{n-1} U_i U_{n-i} \\
&= U_{n-1} + 2U_1 U_{n-1} + \sum_{i=2}^{n-2} U_i U_{n-i}
\end{aligned}$$

Note that $U_1 = 1$. Also, let $C_n = (1/2)U_{n+1}$ for $n \geq 1$, and $C_0 = 1$. Then,

$$\begin{aligned}
2C_{n-1} &= 2C_{n-2} + 4C_0 C_{n-2} + 4 \sum_{i=2}^{n-2} C_{i-1} C_{n-i-1} \\
C_{n-1} &= C_{n-2} + 2C_0 C_{n-2} + 2 \sum_{i=1}^{n-3} C_i C_{n-i-2} \\
C_{n+1} &= C_n + 2C_0 C_n + 2 \sum_{i=1}^{n-1} C_i C_{n-i} \\
C_{n+1} &= C_0 C_n + 2(C_1 C_{n-1} + \dots + C_n C_0)
\end{aligned}$$

The final equation is in the form that Motzkin (1948) wrote the recursion, so

$$\lim_{n \rightarrow \infty} \frac{U_n}{U_{n-1}} = \lim_{n \rightarrow \infty} \frac{(1/2)U_{n+2}}{(1/2)U_{n+1}} = \lim_{n \rightarrow \infty} \frac{C_{n+1}}{C_n} = 3 + 2^{3/2}$$

Therefore the size of the twisted sequence neighbourhood is $\Theta((3 + 2\sqrt{2})^n)$

Note that the generating function is $U(x) = \sum_{i=0}^{\infty} U_i x^i = \frac{1 - x \pm \sqrt{(x-1)^2 - 4x}}{2}$ and by our conjecture (explained in section 8.2) we would predict that the neighbourhood size was $\Theta((3 + 2\sqrt{2})^n)$, since the smallest root of $(x-1)^2 - 4x = 0$ is $3 - 2\sqrt{2} = 1/(3 + 2\sqrt{2})$.

The size of the rotational twisted sequence neighbourhood is at least as large, and at most factor of n larger than the non-rotational neighbourhood. Hence, the rotational twisted sequence neighbourhood is $\Omega((3 + 2\sqrt{2})^n)$ and $O(n(3 + 2\sqrt{2})^n)$.

9.6.8 The rotational twisted sequence neighbourhood and k -opt neighbourhoods

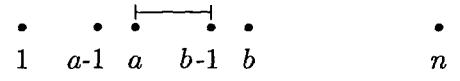
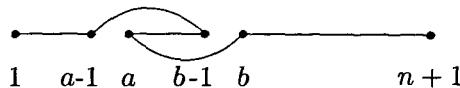
We will now demonstrate how all k -opt neighbourhoods for $k \leq 4$ are contained in the rotational twisted sequence neighbourhood.

If a generalised neighbourhood move can be defined by a set of reversals, where any pair of the reversals are either disjoint or nested, then the neighbourhood is sub-neighbourhood of the rotational twisted sequence neighbourhood. We show how k -opt neighbourhood moves for $k \leq 4$ can be performed by a series of reversals with just this characteristic. In addition we will show how 5-opt is not a sub-neighbourhood of the the rotational twisted sequence neighbourhood.

This is done by describing them in diagrams consisting of a list of cities in their current order above which there are a set of lines, each line signifying that the order of the cities below it are reversed (see section 9.3 for a fuller description of the diagrams).

Pure 2-opt moves

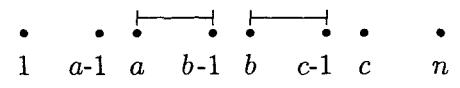
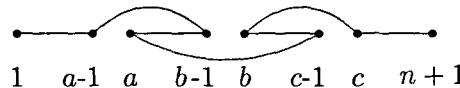
$$M(a-1, b)$$



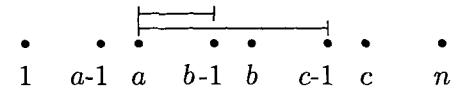
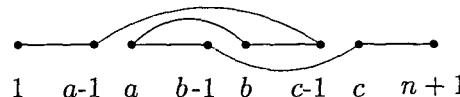
Pure 3-opt moves

All the possible pure 3-opt moves also form a sub-neighbourhood of the twisted sequence as shown in the figures below.

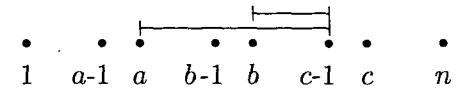
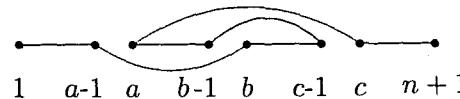
$$M(1; a-1, b, c)$$



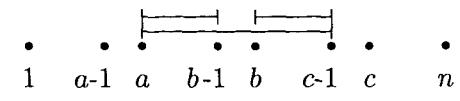
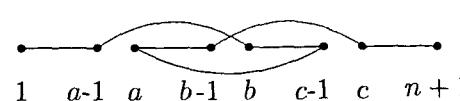
$$M(2; a-1, b, c)$$



$$M(3; a-1, b, c)$$

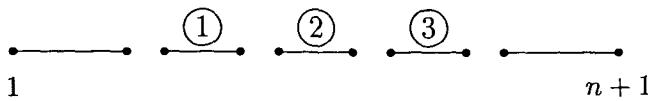


$$M(4; a-1, b, c)$$

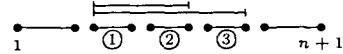
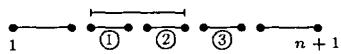
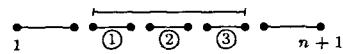
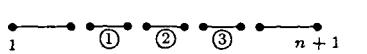


Pure 4-opt moves

In a pure 4-opt, move there are 3 sections of tour to be reconnected.

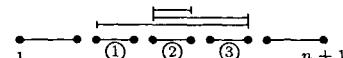


As illustrated in the 6 small diagrams below, the 6 possible orders in which the 3 sections of tour can be connected can all be reached by simple twisted sequence moves.

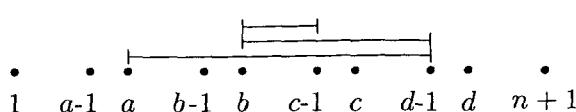
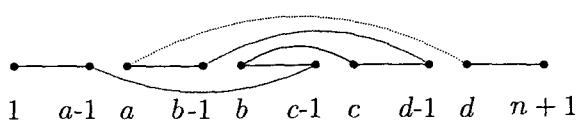


In the 4-opt neighbourhood each of the three sections of tour can be traversed either from left to right or from right to left (not all of them will form pure 4-opt moves). Any given one of these possibilities can be illustrated by adding between 0 and 3 extra lines, to one of the six diagrams above, where each line indicates the reversal of one of the three sections of tour.

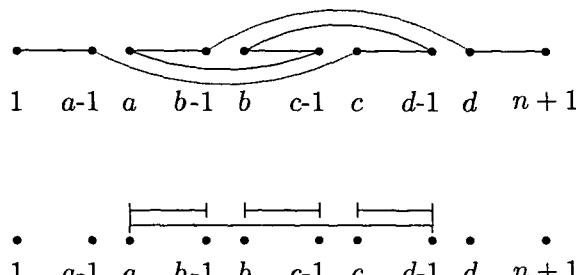
For example adding an extra line, above the 2nd section of tour, to the diagram below left forms the move below right.



The move formed is illustrated once more in the diagram below.

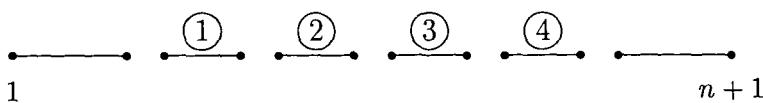


By describing how any of the 4-opt moves can be illustrated by a diagram consisting of a set of reversals we have shown that all pure 4-opt moves form a sub-neighbourhood of the twisted sequence. Unfortunately, there are too many pure 4-opt moves for all of them to be illustrated so only one more example is given. The well-known double-bridged 4-opt move referred to by Bentley and generally used as the kick in iterated Lin-Kernighan.



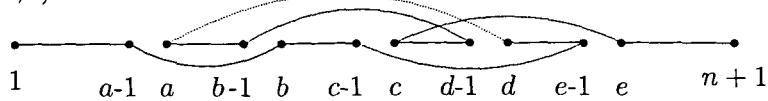
Pure 5-opt moves

In a 5-opt, move there are 4 sections of tour to be reconnected.

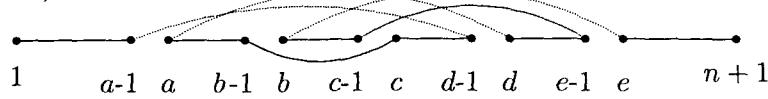


The twisted sequence contains 5-opt moves corresponding to 22 of the possible orders in which the sections can be connected but the 5-opt moves corresponding to the orders (2,4,1,3) and (3,1,4,2) are not present. Note that (2,4,1,3) reversed is the same as (3,1,4,2). An example of each of the two types of 5-opt move which are not members of the twisted sequence are illustrated below.

(2,4,1,3)



(3,1,4,2)

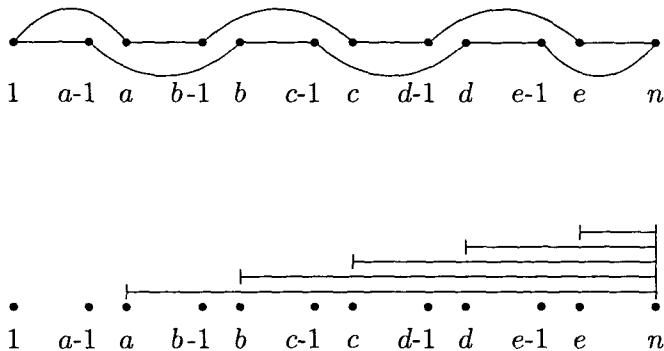


9.6.9 Pyramidal (Sarvanov and Doroshko 1981a)

The pyramidal neighbourhood, as far as we know, is the first PSEN that was discovered. In the USSR, it was shown that the TSP can be minimised over the set of pyramidal permutations in $O(n^2)$ by Klyaus (1976) and the pyramidal neighbourhood was subsequently introduced by Sarvanov and Doroshko (1981a). (See section 4.2.2 for more details)

A permutation σ' is pyramidal in relation to the permutation $\sigma = (\sigma(1), \sigma(2), \dots, \sigma(n))$ if σ' is of the form $(\sigma(i_1), \sigma(i_2), \dots, \sigma(i_k), \sigma(n), \sigma(j_1), \sigma(j_2), \dots, \sigma(j_{n-k-1}))$, where $0 \leq k \leq n-1$, $i_1 < i_2 < \dots < i_k$ and $j_1 > j_2 > \dots > j_{n-k-1}$

The pyramidal neighbourhood consists of all the pyramidal tours for a given permutation representing the tour. Without loss of generality we relabel the cities so that the permutation defining the current tour is $(1, \dots, n)$. The diagrams below both represent the same pyramidal move.



If $2 < a < b < c < d < e < n$, then 6 edges are broken, producing a generic 6-opt move lying in the pyramidal neighbourhood. A 6-opt move breaks the tour into six pieces, and for the 6-opt move to lie in the pyramidal neighbourhood these sections must be traversed in a given order. Let the sections be labelled one to six, the first section starting at city 1 and the sixth ending at city n . Then the sections must be traversed in the following order 1,3,5,6,4,2, where the cities in sections 1,3,5 are traversed in their original order and the cities in sections 6,4,2 are traversed in reverse order.

The second representation gives the move in terms of reversals, showing that the pyramidal neighbourhood is a subneighbourhood of the twisted sequence neighbourhood.

All moves involve breaking between 2 and $n - 2$ edges, effectively producing a k -opt move, where $2 \leq k \leq n - 2$, according to the number of edges broken. All moves in the pyramidal neighbourhood break edge $\{\sigma(1), \sigma(n)\}$. No moves in the pyramidal neighbourhood break edges $\{\sigma(1), \sigma(2)\}$ or $\{\sigma(n - 1), \sigma(n)\}$.

Dynamic programming algorithm

The pyramidal neighbourhood can be searched in $O(n^2)$ time. Although a dynamic programming algorithm is given in the Carlier & Villon 1990 paper, we aim to provide a simpler presentation. The way in which we view the construction of a particular pyramidal tour is (“V-shaped”) as follows. Starting with the path $P_2 = (\sigma(1), \sigma(2))$. Then for $k = 3, \dots, n$, we must decide to which end of the path P_{k-1} to add city $\sigma(k)$, thereby forming P_k . Finally the two ends of the path P_n are joined up to form a tour.

Let $F(\sigma(i), \sigma(j))$ where $i < j$ equal the length of the shortest path in the pyramidal neighbourhood, exclusively involving all the cities in the first j positions, that starts at the city currently in the i th position and ends at the city currently in the j th position.

Initialisation

$$F(\sigma(1), \sigma(2)) = d_{\sigma(1), \sigma(2)}.$$

Recursion

Calculate F values for $k = 3, \dots, n$ where $1 \leq i \leq k - 2$

$$\begin{aligned} F(\sigma(i), \sigma(k)) &= F(\sigma(i), \sigma(k-1)) + d_{\sigma(k-1), \sigma(k)} \\ F(\sigma(k-1), \sigma(k)) &= \min_{1 \leq j \leq k-2} \{F(\sigma(j), \sigma(k-1)) + d_{\sigma(j), \sigma(k)}\}. \end{aligned}$$

Optimal solution value

$$\min_{1 \leq j \leq n-1} \{F(\sigma(j), \sigma(n)) + d_{\sigma(j), \sigma(n)}\}.$$

Neighbourhood size

It is well known that the size of the neighbourhood is $\Theta(2^n)$. However, we will demonstrate how easy it is calculate the size of the neighbourhood U_n using our procedure of conditioning on the largest reversal in which a city is involved.

Starting with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$. Conditioning on the largest reversal involving city $\sigma(n)$

- Given that $\sigma(n)$ is not involved in any reversal then no moves are possible.
- Given that $\sigma(n)$ is involved in the reversal of the section of tour between $\sigma(n - i)$ and $\sigma(n)$ inclusively, namely $(\sigma(n - i), \dots, \sigma(n))$, then U_i different neighbourhood moves are possible, where $0 \leq i \leq n - 1$.

So summing up the different possible moves and adding the null move of performing no reversals, we obtain

$$U_n = \sum_{i=0}^{n-1} U_i \quad \text{for } n \geq 1,$$

where $U_0 = 1$.

The corresponding generating function is $U(x) = \sum_{i=0}^{\infty} U_i x^i = \frac{1-x}{1-2x}$

Since the only root of $1 - 2x = 0$ is $1/2$, the neighbourhood size is $\Theta(2^n)$.

9.6.10 Rotational pyramidal (Carlier and Villon 1990)

The rotational pyramidal neighbourhood was introduced by Carlier & Villon 1990 (see section 4.2.2). The rotational pyramidal neighbourhood consists of all the pyramidal tours that correspond to any of the n rotations representing the current tour. The rotational pyramidal neighbourhood is a sub-neighbourhood of the rotational twisted sequence neighbourhood.

Dynamic programming algorithm

The neighbourhood is easily searched in $O(n^3)$ time by repeatedly applying the $O(n^2)$ dynamic programming algorithm for the pyramidal neighbourhood to each of the n rotations representing the tour.

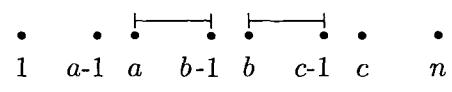
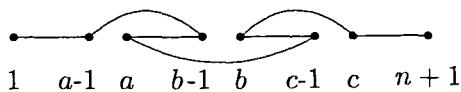
Neighbourhood size

This neighbourhood, although taking $O(n^3)$ to search, only contains a factor of n more neighbours $\Theta(n2^n)$ than the pyramidal neighbourhood. However, it appears to have slightly more practical relevance (see section 4.2.2) containing over 75% of the 3-opt neighbourhood (Dejneko & Woeginger 1997).

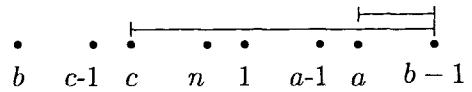
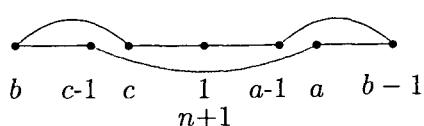
A demonstration of how rotational pyramidal contains over 75% of the 3-opt neighbourhood

We will show how the three of the four types of 3-opt move defined in section 9.2 can be of the form of a pyramidal move if the permutation is rotated.

$$M(1; a-1, b, c)$$



If the permutation representing the tour in the figure above is rotated so that it starts with city b , the move becomes pyramidal as demonstrated in the figure below.



$M(2; a - 1, b, c)$



If the permutation representing the tour in the figure above is rotated so that it starts with city a , the move becomes pyramidal as demonstrated in the figure below.



$M(3; a - 1, b, c)$



If the permutation representing the tour in the figure above is rotated so that it starts with city c , the move becomes pyramidal as demonstrated in the figure below.



$M(4; a - 1, b, c)$



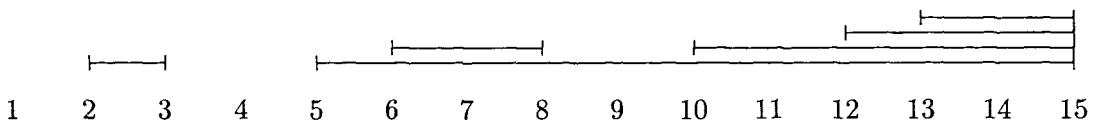
We observe that 3-opt moves of type 4 are not in the rotational pyramidal neighbourhood. So the rotational pyramidal neighbourhood contains over 75% of the 3-opt neighbourhood, since it contains all 2-opt moves and 3 out of the four types of pure 3-opt move. A formal proof of this fact is given by Deïneko & Woeginger (1997).

However, as commented in section 9.6.3, the neighbourhood seems inefficient when compared with the rotational nested reverse neighbourhood which, whilst also taking $O(n^3)$ to search, contains the rotational pyramidal neighbourhood and is significantly larger, $\Omega((2 + \sqrt{2})^n)$ as compared with $\Theta(n2^n)$.

9.6.11 Nested 2-opt inside pyramidal neighbourhood

In the nested 2-opt inside pyramidal neighbourhood, any section of tour which has no edges broken by the underlying pyramidal move can, if it is large enough, contain a nested 2-opt move.

The diagram below illustrates a nested 2-opt inside pyramidal neighbourhood move. The particular move contains the 2-opt moves $M(1, 4)$ and $M(5, 9)$ respectively, producing the reversals $(\sigma(2), \sigma(3))$ and $(\sigma(6), \sigma(7), \sigma(8))$. These moves are possible because neither the edges $\{\sigma(1), \sigma(2)\}$ and $\{\sigma(3), \sigma(4)\}$ nor the edges $\{\sigma(5), \sigma(6)\}$ and $\{\sigma(8), \sigma(9)\}$ are broken by the pyramidal move.



Dynamic programming algorithm

The nested 2-opt inside pyramidal neighbourhood can be searched in $O(n^2)$ time. Firstly, $\Delta(i, j)$, the maximum reduction in path length between positions i and j in the nested 2-opt neighbourhood, is calculated for all i and j in $O(n^2)$, (see section 9.5.3). Then $N(i, j)$ is calculated, the length of the shortest path between i and j that is a move in the nested 2-opt neighbourhood beginning with $\sigma(i)$ and ending with $\sigma(j)$. $N(i, j)$ can be calculated for $1 \leq i, j \leq n$ in $O(n^2)$ time using the equation $N(i, j) = D_{i,j} - \Delta(i, j)$.

Finally, the nested 2-opt inside pyramidal neighbourhood can be built up in $O(n^2)$ time. Let $F(\sigma(i), \sigma(j))$ where $i < j$ equal the length of the shortest path in the nested 2-opt inside pyramidal neighbourhood, exclusively involving all the cities in the first j positions, that starts at the city currently in the i th position and ends at the city currently in the j th position.

Initialisation

$$F(\sigma(1), \sigma(2)) = d_{\sigma(1), \sigma(2)}.$$

Recursion

Calculate F values for $k = 3, \dots, n$ where $1 \leq i \leq k - 2$

$$\begin{aligned} F(\sigma(i), \sigma(k)) &= F(\sigma(i), \sigma(i+1)) + N(i+1, k) \\ F(\sigma(k-1), \sigma(k)) &= \min_{1 \leq j \leq k-2} \{F(\sigma(j), \sigma(k-1)) + d_{\sigma(j), \sigma(k)}\}. \end{aligned}$$

Optimal solution value

$$\min_{1 \leq j \leq n-1} \{F(\sigma(j), \sigma(n)) + d_{\sigma(j), \sigma(n)}\}.$$

Neighbourhood size

We now give an outline of how the size U_n of the neighbourhoods can be calculated. Let N_n be the size of the nested 2-opt neighbourhood (see section 9.5.3) so $N_n = \sum_{i=1}^{n-1} N_i$ and the generating function of N_n , $N(x) = \sum_{i=1}^{\infty} N_i = x(1-x)/(1-2x)$. Starting with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$. Conditioning on the largest reversal involving city $\sigma(n)$:

- Given that $\sigma(n)$ is not involved in any reversal then $N_{n-1} - 1$ nested moves are possible, none of which involve $\sigma(n)$.
- Given that $\sigma(n)$ is involved in the reversal of the section of tour between $\sigma(n-i)$ and $\sigma(n)$ inclusively, namely $(\sigma(n-i), \dots, \sigma(n))$, then for $1 \leq i \leq n-3$ $U_i N_{n-2-i}$ different neighbourhood moves are possible. However, if $i = n-2$ or $n-1$, then U_{n-1} and U_{n-2} moves are possible respectively, because no nested moves are possible outside the reversal.

So summing up the different possible moves and adding the null move of performing no reversals, we obtain

$$U_n = U_{n-1} + U_{n-2} + N_{n-1} + \sum_{i=1}^{n-3} U_i N_{n-2-i} \quad \text{for } n \geq 2,$$

where $U_1 = 1$.

The corresponding generating function is $U(x) = \sum_{i=1}^{\infty} U_i x^i = \frac{x(1-x-x^2)}{1-3x+x^2+x^3+x^4}$

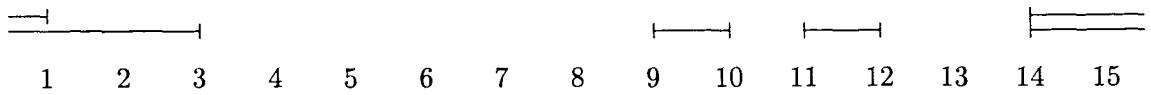
Since the smallest root of $1-3x+x^2+x^3+x^4=0$ is approximately 1/2.29, the nested 2-opt inside pyramidal neighbourhood size is $\Theta(2.29^n)$.

9.7 Some 3-opt dynasearch neighbourhoods.

A 3-opt move $M(i; j, k, l)$ is said to *affect* the cities in $S_{j,l}$. We define a pair of 3-opt moves affecting the cities $S_{i,j}$ and $S_{k,l}$ to be *disjoint* if $S_{i,j} \cap S_{k,l}^- = \emptyset$. A move affecting the cities $S_{i,j}$ be said to be *nested* in $M(k; l, m, n)$ if either $S_{i,j} \subseteq S_{l,m}^-$ or $S_{i,j} \subseteq S_{m,n} - \{n\}$.

9.7.1 Rotational disjoint pure 3-opt

The rotational disjoint 3-opt neighbourhood consists of a set of disjoint pure 3-opt moves.



The diagram above illustrates a move in the neighbourhood consisting of two 3-opt moves namely $M(1; 8, 11, 13)$ and $M(2; 13, 2, 4)$.

Dynamic programming algorithm

The rotational disjoint pure 3-opt neighbourhood can be searched in $O(n^3)$ time. Compute $\Delta'(i, j)$ for all values of i and j is calculated as a preprocessing step in $O(n^3)$ time, using the following equation:

$$\Delta'(i, j) = \max_{k \in S_{i,j}^- - \{i+1\}} \delta(i, k, j)$$

where $\delta(i, k, j)$ is defined in section 9.2. The rotational disjoint pure 3-opt can now be built up in $O(n^3)$ time, by an almost identical dynamic program to that used to build the rotational disjoint 2-opt neighbourhood (see section 9.5.2), the only superficial difference is that $\delta(i, j)$ is replaced by $\Delta'(i, j)$. Let $\Delta(i, j)$ be the maximum reduction in length by performing a disjoint pure 3-opt move exclusively involving the set of cities $S_{i,j}$, where cities $\sigma(i)$ and $\sigma(j)$ remain in positions i and j respectively..

Initialisation

$$\Delta(i, i) = 0 \text{ for } i = 1, \dots, n.$$

Recursion

Calculate $\Delta(i, j)$ for increasing $|S_{i,j}|$ until $\Delta(i + 1, i)$ is known for $1 \leq i \leq n$.

$$\Delta(i, j) = \max_{k \in S_{i,j}^- - \{j\}} \{\Delta(i, k) + \Delta'(k, j)\}.$$

Optimal reduction in solution value

$$\max_{1 \leq i \leq n} \Delta(i + 1, i).$$

Note: The disjoint pure 3-opt neighbourhood still takes $O(n^3)$ time to search, even though the recursion for the disjoint pure 3-opt only takes $O(n^2)$ time. The reason is that the preprocessing step above is still required to calculate $\Delta'(i, j)$.

Neighbourhood size

We now give an outline of how the size U_n of the non-rotational version of the neighbourhood can be calculated, from which we can gain insight into the rotational neighbourhood size.

Starting with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$ and conditioning on moves involving $\sigma(n)$:

- Given $\sigma(n)$ is not involved in a move, $U_{n-1} - 1$ different neighbourhood moves are possible.
- Given a 2.5-opt move involving $(\sigma(i+1), \dots, \sigma(n))$ then there are $2U_i$ different possible moves, where $0 \leq i \leq n-4$.
- Given a 3-opt move (which is not in the 2.5-opt neighbourhood) involving $(\sigma(i+1), \dots, \sigma(n))$ then there are $4(n-i-4)U_i$ different possible moves, where $0 \leq i \leq n-5$.

An explanation of why U_n is multiplied by $4(n-i-4)$ follows. Given that a 3-opt move (which is not in the 2.5-opt neighbourhood) involving $(\sigma(i+1), \dots, \sigma(n))$ the 3-opt move is of the form $M(h; i, k, n)$, then $1 \leq h \leq 4$ and $i+3 \leq k \leq n-2$. So the move can be one of four types of 3-opt move and k , which determines the middle edge that is broken, can take any one of $n-i-4$ different values.

Summing up the different neighbourhood moves and adding the null move of performing no reversals, we obtain

$$U_n = U_{n-1} + \sum_{i=0}^{n-5} 4(n-i-4)U_i + \sum_{i=0}^{n-4} 2U_i \quad \text{for } n \geq 1,$$

where $U_0 = 1$. The corresponding generating function is

$$U(x) = \sum_{i=0}^{\infty} U_i x^i = \frac{(1-x)^2}{(1-3x+3x^2-x^3-2x^4-2x^5)}$$

Since the smallest root of $1-3x+3x^2-x^3-2x^4-2x^5=0$ is $1/2.12$, the non-rotational neighbourhood size is $\Theta(2.12^n)$. The size of the rotational neighbourhood is at least as large, and at most factor of n larger than the non-rotational neighbourhood. Hence, the size of the rotational disjoint pure 3-opt neighbourhood is $\Omega(2.12^n)$ and $O(n2.12^n)$.

9.7.2 Rotational disjoint 3-opt (with 2-opt sub-neighbourhood)

The rotational disjoint 3-opt neighbourhood consists of a set of disjoint 3-opt moves. Unlike in the rotational disjoint pure 3-opt neighbourhood (see section 9.7.1), disjoint 2-opt moves are allowed in this neighbourhood.



The diagram above illustrates a move in the neighbourhood consisting of two 3-opt moves and a single 2-opt move, namely $M(1; 8, 11, 13)$, $M(2; 13, 2, 4)$ and $M(5, 8)$.

Dynamic programming algorithm

The rotational disjoint 3-opt neighbourhood can be searched in $O(n^3)$ time. Compute $\Delta'(i, j)$ for all values of i and j is calculated as a preprocessing step in $O(n^3)$ time, using the following equation:

$$\Delta'(i, j) = \max \begin{cases} \max_{k \in S_{i,j}^- - \{i+1\}} \delta(i, k, j) \\ \delta(i, j) \end{cases}$$

where $\delta(i, k, j)$ is defined in section 9.2.

The rotational disjoint 3-opt can now be built up in $O(n^3)$ time, by an identical dynamic program to that used to build the rotational disjoint pure 3-opt neighbourhood (see section 9.7.1). Let $\Delta(i, j)$ be the maximum reduction in length by performing a disjoint 3-opt move exclusively involving the set of cities $S_{i,j}$, where cities $\sigma(i)$ and $\sigma(j)$ remain in positions i and j respectively..

Initialisation

$$\Delta(i, i) = 0 \text{ for } i = 1, \dots, n.$$

Recursion

Calculate $\Delta(i, j)$ for increasing $|S_{i,j}|$ until $\Delta(i + 1, i)$ is known for $1 \leq i \leq n$.

$$\Delta(i, j) = \max_{k \in S_{i,j} - \{j\}} \{\Delta(i, k) + \Delta'(k, j)\}.$$

Optimal reduction in solution value

$$\max_{1 \leq i \leq n} \Delta(i+1, i).$$

Neighbourhood size

We now give an outline of how the size U_n of the non-rotational neighbourhood can be calculated, from which we can gain insight into the rotational neighbourhood size.

Starting with the identity permutation $(\sigma(1), \sigma(2), \dots, \sigma(n))$ and conditioning on moves involving $\sigma(n)$:

- Given $\sigma(n)$ is not involved in a move $U_{n-1} - 1$ different neighbourhood moves are possible.
- Given a 2.5-opt move involving $(\sigma(i+1), \dots, \sigma(n))$ then there are $2U_i$ different possible moves, where $0 \leq i \leq n-4$.
- Given a 3-opt move (which is not in the 2.5-opt neighbourhood) involving $(\sigma(i+1), \dots, \sigma(n))$ then there are $4(n-i-4)U_i$ different possible moves, where $0 \leq i \leq n-5$ (see section 9.7.1 for an explanation).
- Given a 2-opt move involving $(\sigma(i+1), \dots, \sigma(n))$ (the section of tour $(\sigma(i+1), \dots, \sigma(n-1))$ is reversed), there are U_i different possible moves, where $0 \leq i \leq n-3$.

Summing up the different neighbourhood moves and adding the null move of performing no reversals, we obtain

$$U_n = U_{n-1} + U_{n-3} + \sum_{i=0}^{n-5} 4(n-i-4)U_i + \sum_{i=0}^{n-4} 3U_i \quad \text{for } n \geq 1,$$

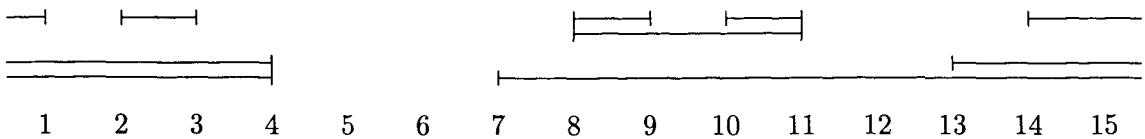
where $U_0 = 1$. The corresponding generating function is

$$U(x) = \sum_{i=0}^{\infty} U_i x^i = \frac{(1-x)^2}{1-3x+3x^2-2x^3-x^4-2x^5}$$

Since the smallest root of $1-3x+3x^2-2x^3-x^4-2x^5 = 0$ is approximately $1/2.22$, the non-rotational neighbourhood size is $\Theta(2.22^n)$. The size of the rotational neighbourhood is at least as large, and at most factor of n larger than the non-rotational neighbourhood. Hence, the size of the rotational disjoint 3-opt neighbourhood is $\Omega(2.22^n)$ and $O(n2.22^n)$.

9.7.3 Rotational nested pure 3-opt

The rotational nested pure 3-opt neighbourhood consists of a set of nested pure 3-opt moves.



The diagram above illustrates a nested move consisting of three 3-opt moves: $M(3; 6, 13, 5)$, $M(4; 7, 10, 12)$ and $M(1; 13, 2, 4)$

Dynamic programming algorithm

The nested pure 3-opt neighbourhood can be searched in $O(n^3)$ time. Note the similarities with the dynamic program for the 2-opt version given in (section 9.5.3). Let $\Delta(i, j)$ be the maximum reduction in length by performing a nested pure 3-opt move exclusively involving the set of cities $S_{i,j}$, where cities $\sigma(i)$ and $\sigma(j)$ remain in positions i and j respectively.

Initialisation

$$\Delta(i, i) = 0 \text{ for } i = 1, \dots, n.$$

Recursion

Calculate $\Delta(i, j)$ for increasing $|S_{i,j}|$ until $\Delta(i+1, i)$ is known for $1 \leq i \leq n$.

$$\Delta(i, j) = \max \begin{cases} \Delta(i, j - 1) \\ \Delta(i + 1, j) \\ \max_{k \in S_{i,j} - \{i+1\}} \{\delta(i, k, j) + \Delta(i + 1, k - 1) + \Delta(k, j - 1)\}. \end{cases}$$

Optimal reduction in solution value

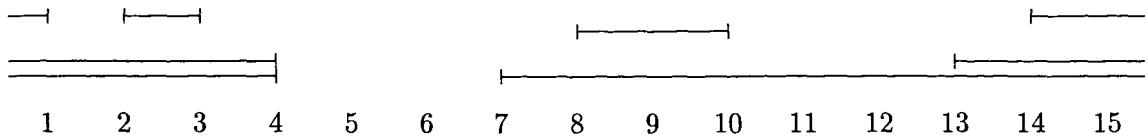
$$\max_{1 \leq i \leq n} \Delta(i + 1, i).$$

Neighbourhood size

The size of this neighbourhood remains open.

9.7.4 Rotational nested 3-opt (with 2-opt sub-neighbourhood)

The rotational nested 3-opt neighbourhood consists of a set of nested 3-opt moves. Unlike in the rotational nested pure 3-opt neighbourhood (see section 9.7.3), nested 2-opt moves are allowed in this neighbourhood.



The diagram above illustrates a nested move consisting of two 3-opt moves and a single 2-opt move: $M(3; 6, 13, 5)$, $M(1; 13, 2, 4)$ and $M(7, 11)$

Dynamic programming algorithm

The nested 3-opt neighbourhood can be searched in $O(n^3)$ time. Let $\Delta(i, j)$ be the maximum reduction in length by performing a nested 3-opt move exclusively involving the set of cities $S_{i,j}$, where cities $\sigma(i)$ and $\sigma(j)$ remain in positions i and j respectively.

Initialisation

$\Delta(i, i) = 0$ for $i = 1, \dots, n$.

Recursion

Calculate $\Delta(i, j)$ for increasing $|S_{i,j}|$ until $\Delta(i+1, i)$ is known for $1 \leq i \leq n$.

$$\Delta(i, j) = \max \begin{cases} \Delta(i, j - 1) \\ \Delta(i + 1, j) \\ \delta(i, j) + \Delta(i + 1, j - 1) \\ \max_{k \in S_{i,j}^- - \{i+1\}} \{\delta(i, k, j) + \Delta(i + 1, k - 1) + \Delta(k, j - 1)\}. \end{cases}$$

Optimal reduction in solution value

$$\max_{1 \leq i \leq n} \Delta(i + 1, i).$$

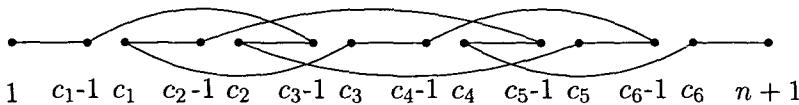
Neighbourhood size

The neighbourhood size is unknown, but obviously it must be greater than the rotational nested 2-opt which is $\Omega(2^n)$

9.8 PSEN in which each neighbour is reachable by a set of k -opt moves but not by a set of reversals.

9.8.1 Overlapping 2-opt

A generic example of an overlapping 2-opt move which consists of the three 2-opt moves $M(c_1 - 1, c_3)$, $M(c_2 - 1, c_5)$ and $M(c_4 - 1, c_6)$, where $c_1 < c_2 < c_3 < c_4 < c_5 < c_6$.



To form an overlapping move, only certain numbers of 2-opt move can be combined. In fact $3k$ and $3k + 1$ 2-opt moves can be combined, for $k \geq 1$ and k integer.

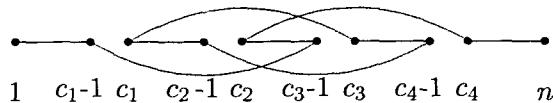
Given that an overlapping 2-opt move comprising of $3k$ 2-opt moves breaks edges $\{c_i - 1, c_i\}$ for $1 \leq i \leq 6k$, then the overlapping 2-opt move must be formed from the 2-opt moves $M(c_1 - 1, c_3)$, $M(c_{6k-2} - 1, c_{6k})$, and $M(c_{2i} - 1, c_{2i+3})$ for $1 \leq i \leq 3k - 2$.

Alternatively, given an overlapping 2-opt move comprising of $3k + 1$ 2-opt moves breaks edges $\{c_i - 1, c_i\}$ for $1 \leq i \leq 6k + 2$, then the overlapping 2-opt move must be formed from the 2-opt moves $M(c_1 - 1, c_3)$, $M(c_{6k} - 1, c_{6k+2})$, and $M(c_{2i} - 1, c_{2i+3})$ for $1 \leq i \leq 3k - 1$.

The diagram below gives an example of an illegal move produced by combining two 2-opt moves, which demonstrates how a loop of tour, joining

$$(c_1, c_3, c_3 + 1, \dots, c_4 - 1, c_2 - 1, c_2 - 2, \dots, c_1)$$

forms a sub-tour. The same effect, of a disconnected loop of tour, is seen in all illegal moves, i.e. in overlapping combinations of $3k + 2$ 2-opt moves, for $k \geq 0$.



Dynamic programming algorithm

The overlapping 2-opt neighbourhood can be searched in $O(n^3)$ time. The dynamic program is slightly more complicated than those used to search neighbourhoods introduced earlier in the chapter. The added complication comes from the

need to keep track to some extent of the number of 2-opt moves currently combined, because only $3k$ and $3k + 1$ 2-opt moves can be combined, for $k \geq 1$. Even then we need to keep track of moves made currently made up of $3k + 1$ moves because if further 2-opt moves are added, a legal overlapping 2-opt move may be formed. Whilst building the overlapping 2-opt move we keep track of whether there are $3k$, $3k + 1$ or $3k + 2$ moves in the current partial move by coding the move 0, 1 or 2 respectively (if i is the number of 2-opt moves, then we evaluate $i \bmod 3$).

Let $\Delta(c, d, i \bmod 3)$ denote the largest reduction in tour length obtainable from overlapping/disjoint 2-opt moves only involving cities in the first d positions in the current tour, where no edges are broken between positions c and $d - 1$ ($c < d$), where i is equal to the number of moves in the group of overlapping moves of which the move breaking edge $\{d - 1, d\}$ is a member and is 0 if the edge $\{d - 1, d\}$ is not broken by any move as yet.

Initialise

$$\Delta(c, d, i) = 0 \quad c < d \quad d = 1, 2, 3 \quad i = 0, 1, 2$$

$$\Delta(0, d, i) = -\infty \quad d = 1, \dots, n \quad i = 0, 1, 2$$

Recursion

Calculate Δ values for $d = 4, \dots, n$ where $1 \leq c < d$

$$\begin{aligned} \Delta(c, d, 0) &= \max \left\{ \begin{array}{l} \Delta(c - 1, d, 0) \\ \Delta(c - 1, c, 0) \\ \Delta(c - 1, c, 1) \\ \max_{2 \leq b \leq c-2} \{\delta(b, d) + \Delta(b, c, 2)\} \end{array} \right. \\ \Delta(c, d, 1) &= \max \left\{ \begin{array}{l} \Delta(c - 1, d, 1) \\ \delta(c - 1, d) + \Delta(c - 2, c - 1, 0) \\ \max_{2 \leq b \leq c-2} \{\delta(b, d) + \Delta(b, c, 0)\} \end{array} \right. \\ \Delta(c, d, 2) &= \max \left\{ \begin{array}{l} \Delta(c - 1, d, 2) \\ \max_{2 \leq b \leq c-2} \{\delta(b, d) + \Delta(b, c, 1)\}. \end{array} \right. \end{aligned}$$

We need to keep track of currently illegal moves of the type $\Delta(c, d, 2)$ because they may eventually become part of a legal move. In the final step, however, we are only interested in legal moves $\Delta(n, n + 1, 0)$ and $\Delta(n, n + 1, 1)$.

Optimal reduction in solution value

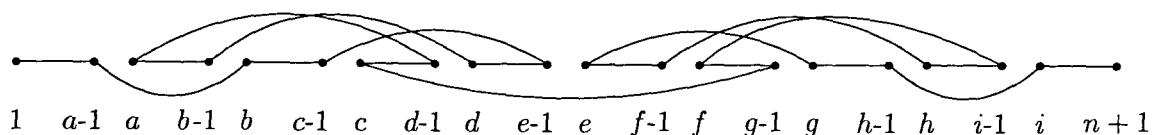
$$\max \left\{ \begin{array}{l} \Delta(n, n + 1, 0) \\ \Delta(n, n + 1, 1). \end{array} \right.$$

Neighbourhood size

The size of the overlapping 2-opt remains unknown, but it must be greater than the size of disjoint 2-opt which is a sub-neighbourhood.

9.8.2 Overlapping 3-opt

The diagram below gives an example of an overlapping 3-opt move formed by combining the three 3-opt moves: $M(4; a - 1, b, d)$, $M(1; c - 1, e, g)$ and $M(4; f - 1, h, i)$.



The types of move which can follow a type 4 move to form a legal combined move are dependent on the move, if any, which occurred before the type 4 move. Therefore, to make the description of the constraints required to produce a legal move in this neighbourhood easier we will view moves of type 4 as being of two distinct types, which we refer to as type 5 and 6.

A type 5 move in this section is viewed as collecting a section of tour, reversing it and inserting it earlier in the tour as in $M(4; f - 1, h, i)$. $M(4; f - 1, h, i)$ is seen as collecting the section of tour $(h, \dots, i - 1)$ and inserting it between $f - 1$ and f .

A type 6 move in this section is viewed as collecting a section of tour, reversing it and inserting it later in the tour as in $M(4; a - 1, b, d)$. $M(4; a - 1, b, d)$ is seen as collecting the section of tour $(a, \dots, b - 1)$ and inserting it between $d - 1$ and d .

There are no restrictions on the number of 3-opt moves that need to be combined to form a legal overlapping 3-opt move. However, only certain types of 3-opt move can follow each other.

A move of type...	can only follow a move of type...
1	3,6
2	1,2,3,5,6
3	3,6
5	1,2,3,5,6
6	3,6

In the diagram above a move of type 6 is followed by a move of type 1, which in turn is followed by a move of type 5, forming a legal move.

Dynamic programming algorithm

The overlapping 3-opt neighbourhood can be searched in $O(n^3)$ time. Firstly we will define the following terms, referring to the reduction in tour length due to the best move in the overlapping 3-opt neighbourhood with the following restrictions.

$\Delta(c)$: The combined move only involves the cities in the first c positions.

$\Delta'(b, c, i)$: The combined move only involves the cities in the first c positions, in which edges $\{b - 1, b\}$ and $\{c - 1, c\}$ are involved in the combined move but edges connecting cities between positions b and $c - 1$ are not involved in the combined move. (This means that edges $\{b - 1, b\}$ and $\{c - 1, c\}$ are the last two edges in the previous 3-opt move made.) The last 3-opt move made is of type i .

$\Delta''(b, c, i)$: The combined move only involves the cities in the first c positions where edge $\{c - 1, c\}$ is involved but edges connecting cities between positions $b - 1$ and $c - 1$ are not involved. The last 3-opt move made is of type i .

$\Delta'''(b, c, i)$: The combined move only involves the cities in the first c positions where edge $\{b - 1, b\}$ is not involved in the combined move and only one edge in the interval (b, \dots, c) has been broken to form the combined move. The last 3-opt move made is of type i .

Dynamic programming algorithm cont.

Initialisation

$$\begin{aligned}
 \Delta(c) &= 0 & c \leq 5, \forall i \\
 \Delta'(b, c, i) &= -\infty & c \leq 5, \forall i \\
 \Delta'(b, c, i) &= -\infty & b \geq c - 1, \forall i \\
 \Delta''(b, c, i) &= -\infty & b \geq c, \forall i \\
 \Delta'''(b, c, i) &= 0 & 1 < b < c \leq 5, \forall i.
 \end{aligned}$$

Recursion

Using the equations below for $c = 6, \dots, n + 1$ and $b = 1, \dots, c - 1$.

For $j \in \{2, 5\}$

$$\Delta'(b, c, j) = \max \left\{ \begin{array}{l} \max_{\substack{1 \leq a < b, \\ i \in \{1, 2, 3, 5, 6\}}} \{\Delta'''(a, b - 1, i) + \delta(j; a - 1, b, c)\} \\ \max_{\substack{1 < a < b}} \{\Delta(a - 1) + \delta(j; a - 1, b, c)\} \end{array} \right.$$

For $j \in \{1, 3, 6\}$

$$\Delta'(b, c, j) = \max \left\{ \begin{array}{l} \max_{\substack{1 \leq a < b, \\ i \in \{3, 6\}}} \{\Delta'''(a, b - 1, i) + \delta(j; a - 1, b, c)\} \\ \max_{\substack{1 < a < b}} \{\Delta(a - 1) + \delta(j; a - 1, b, c)\} \end{array} \right.$$

For $j \in \{1, 2, 3, 5, 6\}$

$$\Delta''(b, c, j) = \max_{1 < a < b} \Delta'(a, c, j)$$

For $j \in \{1, 2, 3, 5, 6\}$

$$\Delta'''(b, c, j) = \max_{b < a \leq c} \Delta''(b, a, j)$$

$$\Delta(c) = \max \left\{ \begin{array}{l} \Delta(c - 1) \\ \max_{\substack{1 \leq a < c, \\ i \in \{1, 2, 3, 5, 6\}}} \{\Delta'(a, c, i)\}. \end{array} \right.$$

Optimal solution value

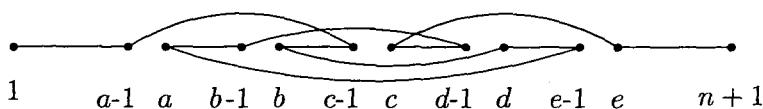
Finally, $\Delta(n + 1)$, the improvement due to the best move in the neighbourhood is known and the best move can be found by backtracking.

Neighbourhood size

The size of the overlapping 3-opt remains unknown but must be greater than the size of disjoint 3-opt which is a sub-neighbourhood.

9.8.3 Other Overlapping neighbourhoods

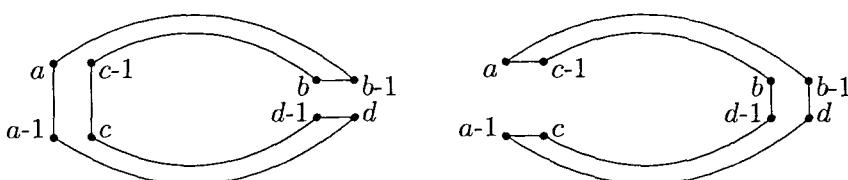
Other neighbourhoods that may be of interest are those produced by combining 2-opt and 3-opt moves to form overlapping neighbourhoods searchable in $O(n^3)$. A simple example of an overlapping move is shown below. Note that although the 2-opt move is totally contained within the 3-opt move, we have reserved the term nested for the situation where the 2-opt move is wholly contained in either the interval $(a, \dots, c - 1)$ or $(c, \dots, e - 1)$. All nested neighbourhoods made by combining k -opt moves, where $k \leq 4$, by this definition are sub-neighbourhoods of the twisted sequence (reachable by a set of reversals).



The figure above contains the 3-opt move $M(1; a - 1, c, e)$ and the 2-opt move $M(b - 1, d)$.

9.9 Other PSEN neighbourhoods

Other neighbourhoods are possible involving k -opt moves for $k \geq 4$. However, as indicated above, they only appear to be of academic interest. Johnson (1990) noted many speedups are based on the locality of cities involved in a move and these cannot be generalised for 4-opt and above. One pair of edges involved in an improving 4-opt move may be arbitrarily far from another pair of edges. This is illustrated in the diagram below (from Bentley 1992), where the pair of edges $\{a - 1, a\}$ and $\{c - 1, c\}$ can be arbitrarily far from the pair of edges $\{b - 1, b\}$ and $\{d - 1, d\}$ because the edges added are $\{a - 1, c\}$, $\{c - 1, a\}$, $\{b - 1, d\}$ and $\{d - 1, b\}$.



9.10 Conclusion

We have shown how the neighbourhoods for the TSP given in tables 9.1, 9.2 and 9.3 can be formed giving a dynamic programming algorithm enabling each neighbourhood to be searched in polynomial time, and for many of the neighbourhoods calculating the order of its size.

Type of strict reverse neighbourhood	Complexity of DP	Size of neighbourhood	Section number
Disjoint	$O(n^2)$	$\Theta(1.76^n)$	9.5.1
Rotational nested	$O(n^2)$	$\Omega(2^n)$ and $O(n2^n)$	9.5.3
Restricted 2-opt	$O(n)$	$\Theta(\sqrt{2}^n)$	9.5.4
Nested inside disjoint	$O(n^2)$	$\Theta(2.21^n)$	9.5.5
Disjoint inside rotational nested	$O(n^3)$	$\Omega(2.35^n)$ and $O(n2.35^n)$	9.5.7
Rotational combined disjoint and nested	$O(n^3)$	$\Omega(2.61^n)$ and $O(n2.61^n)$	9.5.8

Table 9.1: PSEN for the TSP based on strict nested/disjoint reversals

Type of reverse neighbourhood	Complexity of DP	Size of neighbourhood	Section number
Disjoint	$O(n^3)$	$\Theta(2^n)$	9.6.1
Rotational nested	$O(n^3)$	$\Omega((2 + \sqrt{2})^n)$ and $O(n(2 + \sqrt{2})^n)$	9.6.3
Nested inside disjoint	$O(n^4)$	$\Theta((2 + \sqrt{3})^n)$	9.6.4
Disjoint inside rotational nested	$O(n^5)$	$\Omega((3 + \sqrt{2})^n)$ and $O(n(3 + \sqrt{2})^n)$	9.6.6
Rotational twisted	$O(n^7)$	$\Omega((3 + 2\sqrt{2})^n)$ and $O(n(3 + 2\sqrt{2})^n)$	9.6.7
Pyramidal	$O(n^2)$	$\Theta(2^n)$	9.6.9
Rotational pyramidal	$O(n^3)$	$\Theta(n2^n)$	9.6.10
Nested 2-opt inside pyramidal	$O(n^2)$	$\Theta(2.29^n)$	9.6.11

Table 9.2: PSEN for the TSP based on nested/disjoint reverse

Generally we feel that the strict reverse (dynasearch) neighbourhoods in table 9.1 are likely to have greater practical relevance than the reverse neighbourhoods in table 9.2, due to their lower computational complexities. In particular the nested inside disjoint 2-opt and the rotational combined disjoint and nested 2-opt neighbourhoods appear to be most likely to be of practical interest.

It would be interesting to see how our rotational nested reverse neighbourhood compares with the Carlier-Villon rotational pyramidal neighbourhood for the TSP, since the rotational nested reverse neighbourhood is significantly larger $\Omega((2+\sqrt{2})^n)$ as compared to $\Theta(n2^n)$. Given that the rotational nested reverse neighbourhood actually contains the Carlier-Villon neighbourhood and both take $O(n^3)$, to search it would seem on this evidence that the new neighbourhood is clearly superior. However, computational complexity is only a rough guide to effective running times on practical problems.

Type of dynasearch <i>k</i> -opt neighbourhood	Complexity of DP	Size of neighbourhood	Section number
Rotational disjoint pure 3-opt	$O(n^3)$	$\Omega(2^n)$ and $O(n2^n)$	9.7.1
Rotational disjoint 3-opt	$O(n^3)$	$\Omega(2.08^n)$ and $O(n2.08^n)$	9.7.2
Nested pure 3-opt	$O(n^3)$		9.7.3
Nested 3-opt	$O(n^3)$		9.7.4
Overlapping 2-opt	$O(n^3)$		9.8.1
Overlapping 3-opt	$O(n^3)$		9.8.2

Table 9.3: Other dynasearch PSEN

We believe that the nested inside disjoint 3-opt is of particular interest as it has the same complexity as the disjoint 3-opt neighbourhood ($O(n^3)$), which it contains. Although we have not given the dynamic program to search this neighbourhood in this chapter, it can be searched by a corresponding method to the one used for the nested inside disjoint 2-opt (section 9.5.6).

Chapter 10

Conclusions and Extensions

In this thesis we have derived many new polynomially searchable exponential neighbourhoods for three sequencing problems and successfully applied one of our PSEN to each of the problems.

The new dynasearch neighbourhoods have been produced by combining simple well-known neighbourhood moves (swap, insert and k-opt), so that the moves can be performed together as a single move. The neighbourhood moves have been combined in such a way that the effect of the combined move on the objective function is equal to the sum of the effects of the individual moves from the underlying neighbourhood. We have formed dynasearch neighbourhoods containing underlying moves:

- nested within each other;
- disjoint from each other;
- and in the case of the TSP overlapping each other.

Since each dynasearch move corresponds to a series of moves in a traditional local search algorithm, dynasearch has a lookahead capability that is not present in these previous methods.

We have successfully implemented disjoint dynasearch neighbourhoods for the total weighted tardiness, linear ordering and travelling salesman problem. Our implementation of iterated disjoint dynasearch is state-of-the-art both for the total weighted tardiness problem and the linear ordering problem. Our implementation of GLS (Guided local search) dynasearch appears to be competitive for the more widely researched travelling salesman problem. We have found that the quality of implementation may be a crucial factor in determining the relative competitiveness of local search algorithms. Allowing for this factor, we feel that GLS (Voudouris & Tsang 1999) deserves further research.

Through our computational work we have come to believe that not only should all new local search heuristics be compared as a matter of course with multi-start first improve descent, but they should also be compared with iterated first improve descent. The work load in implementing such an algorithm is small, particularly if a kick comprises of multiple random moves from the underlying neighbourhood as we have proposed. It may be harder to defend the need for some of the more elaborate parts of a given algorithm if it cannot outperform an algorithm which simply iteratively descends to a local minimum and then performs a collection of random moves.

We have also introduced some neighbourhoods for the TSP where the effect on the objective function of the combined move is not equal to the sum of the effects of the individual underlying moves. The underlying neighbourhood moves are sections of reversed code that interact with each other. We have shown how both the twisted sequence and pyramidal neighbourhoods can be formed in this manner, and in so doing have shown that that the rotational pyramidal neighbourhood introduced by Carlier & Villon (1990) is a sub-neighbourhood of the rotational twisted sequence. In addition, the insight obtained through visualising the twisted sequences in this manner has enabled us to calculate the number of sequences contained in the twisted sequence data structure, the value of which has remained open since their introduction by Aurenhammer (1988).

Our work has left some open ends, both theoretical and computational. Further computational work is required to implement some of the more promising new neighbourhoods which we have derived. Although in Chapter 8 we derived five insert dynasearch neighbourhoods for the linear ordering problem, we have only implemented the disjoint insert neighbourhood. The nested inside disjoint neighbourhood looks particularly promising, since it is considerably bigger than the disjoint neighbourhood whilst still being searchable in $O(n^2)$ time. Perhaps iterated local search based on the nested inside disjoint insert neighbourhood would out-perform our current state-of-the-art local search method, iterated local search using the disjoint insert neighbourhood.

Additional computational work is needed to evaluate the practical significance of the numerous neighbourhoods for the TSP which we have derived. In particular, it would be interesting to see how our rotational nested reverse neighbourhood com-

pares with the Carlier-Villon rotational pyramidal neighbourhood for the TSP, since the rotational nested reverse neighbourhood is significantly larger $\Omega((2 + \sqrt{2})^n)$ as compared to $\Theta(n2^n)$. Given that the rotational nested reverse neighbourhood actually contains the Carlier-Villon neighbourhood and both take $O(n^3)$, to search it would seem on this evidence that the new neighbourhood is clearly superior. However, computational complexity is only a rough guide to effective running times on practical problems.

We believe that the nested inside disjoint 3-opt is of particular interest as it has the same complexity as the disjoint 3-opt neighbourhood ($O(n^3)$), which it contains. We have not given the dynamic program to search this neighbourhood in this thesis but it can be searched by a corresponding method to the one used for the nested inside disjoint 2-opt (section 9.5.6). An implementation of GLS (Voudouris & Tsang 1999) using the nested inside disjoint 3-opt neighbourhood may be more effective than our implementation of GLS using the disjoint 3-opt neighbourhood.

On the theoretical side the dynasearch neighbourhoods for the TSP formed from overlapping k -opt moves and combinations of disjoint and nested 3-opt moves deserve further investigation. More neighbourhoods can be derived, for example for forming legal combinations of 2-opt and 3-opt moves. In addition, the sizes of the overlapping neighbourhoods and nested 3-opt neighbourhoods introduced remain unknown.

A wider question remains open, as to the exact characteristics of a problem that enable dynasearch neighbourhoods to be formed.

In this thesis, we believe that we have demonstrated the power of PSENs built up from underlying traditional neighbourhoods both as a concept for developing new effective neighbourhoods and as a way of gaining greater insight into the underlying structure of some known neighbourhoods. This insight has enabled both the discovery of sub-neighbourhoods and the size of the neighbourhoods to be calculated.

Bibliography

- [1] E.H.L. AARTS, J.H.M KORST AND P.J.M VAN LAARHOVEN (1997). Simulated Annealing. In *Local Search in Combinatorial Optimization*, eds. E.H.L. AARTS AND J.K. LENSTRA, Wiley, Chichester, UK.
- [2] E.H.L. AARTS AND J.K. LENSTRA (1997). Introduction. In *Local Search in Combinatorial Optimization*, eds. E.H.L. AARTS AND J.K. LENSTRA, Wiley, Chichester, UK.
- [3] E.J. ANDERSON, C. GLASS AND C.N. POTTS (1997). *Machine scheduling*. in *Local Search in Combinatorial Optimization*, eds. E.H.L. AARTS AND J.K. LENSTRA, Wiley, Chichester, UK.
- [4] D. APPLEGATE, R. BIXBY, V. CHVATAL, AND W. COOK (1995). *Finding cuts in the TSP: a preliminary report*, Report95-05, DIMACS, Rutgers University, New Brunswick, New Jersey.
- [5] D. APPLEGATE, R. BIXBY, V. CHVATAL, AND W. COOK (1997). A new paradigm for finding cutting planes in the TSP. Presented at the international symposium on mathematical programming, Lausanne.
- [6] D. APPLEGATE, R. BIXBY, V. CHVATAL, AND W. COOK (1998). On the solution of traveling salesman problems. *Documenta Mathematica* Extra Volume: Proceedings ICM III 645-656.
- [7] D. APPLEGATE, R. BIXBY, V. CHVATAL, AND W. COOK (1999). *Finding Tours in the TSP*. Preprint, DIMACS, Rutgers University, New Brunswick, New Jersey.
- [8] F. AURENHAMMER (1988). On-line sorting of twisted sequences in linear time, *BIT* 28, 194–204.

- [9] E. BALAS (1999). New classes of efficiently solvable generalized traveling salesman problems. *Annals of Operations Research* 86, 529–558.
- [10] E. BALAS AND N. SIMONETTI (1998). *Linear time dynamic programming algorithms for new classes of restricted TSP's a computational study*. Preprint, Carnegie Mellon University, Pittsburgh PA 15213, USA
- [11] E. BALAS, A. VAZACOPOULOS (1994). *Guided local search with shifting bottleneck for job shop scheduling*. Management Science Research Report MSRR-609, Carnegie Mellon University, Pittsburgh, PA.
- [12] E. BALAS, A. VAZACOPOULOS (1998). Guided local search with shifting bottleneck for job shop scheduling. *Management Science* 44, No 2, 262–275.
- [13] E.B. BAUM (1986A). *Iterated descent: a better algorithm for local search in combinatorial optimization problems*, Unpublished manuscript.
- [14] E.B. BAUM (1986B). Towards practical ‘neural’ computation for combinatorial optimization problems. J.S. DENKER (ED.). *Neural Networks for Computing*, Proceedings AIP conference 151, American Institute of Physics, New York, 53–58.
- [15] J. BAXTER (1981). Local optima avoidance in depot location. *Journal of the Operational Research Society* 32, 815-819.
- [16] J.E. BEASLEY (1990). OR Library: Distributing Test Problems by Electronic Mail. *Journal of the Operational Research Society* 41, 1069-1072.
- [17] O. BECKER (1967). Das Helmstdtersche Reihenfolgeproblem — die Effizienz verschiedener Nherungsverfahren. Computer uses in the Social Sciences, Berichtsteiner Working Conference, Wien, January 1967.
- [18] J.L. BENTLEY (1990). Experiments on traveling salesman heuristics. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, Philadelphia, 91–99.
- [19] J.L. BENTLEY (1992). Fast algorithms for geometric traveling salesman problems. *ORSA Journal on Computing* 4, 387–411.

- [20] R.G. BLAND AND D.F. SHALLCROSS (1989). Large traveling salesman problems arising from experiments in X-ray crystallography: a preliminary report on computation. *Operations Research Letters* 8, 125-128.
- [21] J. BLAZEWICZ AND K.H. ECKER (1994). *Scheduling in computer and manufacturing systems*, Springer-Verlag Berlin.
- [22] K. BOENCHENDORF (1982). Reihenfolgenprobleme / Mean-Flow-Time Sequencing. *Mathematical Systems in Economics 74*, Veragsgruppe Athenäum Hain Scriptor Hanstein.
- [23] J.F. BOYCE, C.H.D. DIMITROPOULOS, G. VOM SCHEIDT AND J.G TAYLOR (1995). GENET and tabu search for combinatorial optimisation problems, in *proceedings of World Congress on Neural Networks*, Washington D.C., INNS press.
- [24] P. BRUCKER, J. HURINK AND F. WERNER (1996). Improving local search heuristics for some scheduling problems. *Discrete Applied Mathematics* 65, 97-122.
- [25] P. BRUCKER, J. HURINK AND F. WERNER (1997). Improving local search heuristics for some scheduling problems. Part 2, *Discrete Applied Mathematics* 72, 47-69.
- [26] R.E. BURKARD, V.G. DEINEKO (1995). Polynomially solvable cases of the traveling salesman problem and a new exponential neighbourhood. *Computing*, 54, 191-211.
- [27] R.E. BURKARD, V.G. DEINEKO AND G.J. WOEGINGER (1996). The traveling salesman and the PQ-tree, in *proceedings of IPCO V, LNCS 1084*, Springer Verlag, 490-504.
- [28] J. CARLIER AND P. VILLON (1990). A new heuristic for the traveling salesman problem. *RAIRO Recherche Opérationnelle* 24, 245-253.
- [29] L. CAVIQUE, C. REGO AND I THEMIDO (1999). Subpath ejection chains and tabu search for the crew scheduling problem. *Journal of the Operational Research Society* 50, 608-616.

- [30] V. ČERNÝ (1985). A thermodynamical approach to the travelling salesmans problem: an efficient simulation algorithm. *Journal of Optimisation Theory and Application* 45, 41-55.
- [31] S. CHANAS AND P. KOBYLANSKI (1996). A New Heuristic Algorithm Solving the Linear Ordering Problem. *Computational Optimization and Applications* 6, 191205.
- [32] I. CHARON AND O. HUDRY (1993). The noising method: a new method for combinatorial optimization. *Operations Research Letters* 14, 133-137.
- [33] R.K. CONGRAM, C.N. POTTS AND S.L. VAN DE VELDE (1998). An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. preprint No. OR95, Faculty of Mathematical studies, University of Southampton, Southampton.
- [34] G. CORNUÉJOLS, D. NADDEF AND W.R. PULLEYBLANK (1983). Halin graphs and the travelling salesman problem. *Mathematical Programming* 26, 287-294.
- [35] H.A.J. CRAUWELS, C.N. POTTS AND L.N. VAN WASSENHOVE (1998). Local search heuristics for the single machine total weighted tardiness scheduling problem, *Informs Journal on Computing* 10, 341–350.
- [36] G.A. CROES (1958). A method for solving traveling-salesman problems. *Operations Research* 6, 791–812.
- [37] H. CROWDER AND M. PADBERG (1980). Solving large-scale symmetric travelling salesman problems to optimality. *Management Science* 26, 495–509.
- [38] V. DEINEKO AND G.J. WOEGINGER (1997). *A study of exponential neighborhoods for the traveling salesman problem and for the quadratic assignment problem*, Report Woe-05, Institut für Mathematik, TU Graz, Graz, Austria.
- [39] M. DORIGO (1992). *Optimization, Learning, and Natural Algorithms*. PhD thesis, Politecnico di Milano.
- [40] M. DORIGO, G. DI CARO AND L.M. GAMBARDELLA (1999). Ant algorithms for discrete optimization. *Artifical Life* 5, 137–172.

- [41] M. DORIGO AND L.M GAMBARDELLA (1997). Ant colonies for the traveling salesman problem. *Bio Systems* 43, 73–81.
- [42] M. DORIGO, V. MANIEZZO AND A. COLORNI (1991). *Positive feedback as a search strategy*. Technical report 91-016, Politecnico di Milano.
- [43] M. DORIGO, V. MANIEZZO AND A. COLORNI (1996). The ant system : Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics - Part B* 26, 29-41.
- [44] U. DORNDORF AND E. PESCH (1994). Fast clustering algorithms. *ORSA Journal on Computing* 6, 141–153.
- [45] K.A. DOWSLAND (1995). Simulated Annealing. In *Modern heuristic techniques for combinatorial problems*, ed. C.R. REEVES, McGraw-Hill book company, London.
- [46] G. DUECK AND T. SCHEUER (1990). Threshold accepting: a general purpose optimisation algorithm appearing superior to simulated annealing. *Journal of Computational Physics* 90, 161-175.
- [47] R. DURBIN AND D. WILLSHAW (1987). An analogue approach to the traveling salesman problem using an elastic net method. *Nature* 326, 689-691.
- [48] T.A. FEO AND M.G.C RESENDE (1989). A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters* 8, 67-71.
- [49] T.A. FEO AND M.G.C RESENDE (1995). Greedy randomized adaptive search procedures. *Journal of Global Optimization* 6, 109-133.
- [50] C. FLEURENT AND F. GLOVER (1998). Improved constructive multistart strategies for the quadratic assignment problem using adaptive memory. *INFORMS Journal on Computing* To appear.
- [51] M.M. FLOOD (1990). Exact and heuristic algorithms for the weighted feedback arc set problem: A special case of the skew-symmetric quadratic assignment problem. *Networks* 20, 1-23.
- [52] S. FRENCH (1982). *Sequencing and Scheduling, An introduction to the mathematics of the job-shop*, Ellis Horwood Limited a division of John Wiley & sons, Chichester, England.

- [53] M.R. GAREY AND D.S. JOHNSON (1979). *Computers and intractability A guide to the theory of NP-completeness*, W.H Freeman and company, San Francisco, USA.
- [54] P.C. GILMORE, E.L. LAWLER, AND D.B. SHMOYS, (1985). Well-solved special cases. In *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, eds. E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN, AND D.B. SHMOYS, Wiley, New York.
- [55] F. GLOVER (1986). Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research* 13, 533–549.
- [56] F. GLOVER (1990). Tabu search: A Tutorial *Interfaces* 20, 74–94.
- [57] F. GLOVER (1992). New ejection chain and alternating path methods of traveling salesman problems. *Computer Science and Operations Research* 449–509.
- [58] F. GLOVER (1996). Ejection chains, reference structures and alternating path methods for traveling salesman problems. *Discrete Applied Mathematics* 65, 223-253.
- [59] F. GLOVER, G. GUTIN, A. YEO, A. ZVEROVICH (1999). *Construction heuristics and domination analysis for the asymmetric TSP*. Proc. 3rd Workshop on Algorithm Engineering, London July 1999, J.S. Vitter and C.D. Zaroliagis (eds.), LNCS, 1668.
- [60] F.S. GLOVER AND T. IBARAKI AND M. YAGIURA (1999). An ejection chain approach for the generalized assignment problem. *Proceedings of the Third Metaheuristics International Conference* 223-228. (although not presented)
- [61] F. GLOVER AND T. KLASTORIN AND D. KLINGMAN (1974). Optimal Weighted Ancestry Relationships. *Management Science* 20, 1190-1193.
- [62] F. GLOVER AND E. PESCH (1995). *Ejection chains and graph-related applications*. Preprint Graduate School of Business, University of Colorado, Boulder.
- [63] F. GLOVER AND A.P. PUNNEN (1997). The travelling salesman problem: New solvable cases and linkages with the development of approximation algorithms. *Journal of the Operational Research Society* 48, 502-510.

- [64] M. GRÖTSCHEL, M. JÜNGER AND G. REINELT (1984A). A cutting plane algorithm for the linear ordering problem. *Operations Research* 32, No. 6, 1195-1220.
- [65] M. GRÖTSCHEL, M. JÜNGER AND G. REINELT (1984B). Optimal triangulation of large real world input-output matrices. *Statistische Hefte* 25, 261-295.
- [66] M. GRÖTSCHEL, M. JÜNGER AND G. REINELT (1985A). On the acyclic subgraph polytope. *Mathematical Programming* 33, 28-42.
- [67] M. GRÖTSCHEL, M. JÜNGER AND G. REINELT (1985B). Facets of the linear ordering polytope. *Mathematical Programming* 33, 43-60.
- [68] M. GRÖTSCHEL, L. LOVASZ AND A. SCHRIJVER (1988). *Geometric Algorithms and Combinatorial Optimisation*, Springer, Berlin.
- [69] G. GUTIN (1997). *Exponential neighbourhood local search for the travelling salesman problem*. Research Report TR/2/97, Brunel University , Uxbridge, Middlesex.
- [70] G. GUTIN (1999). Exponential neighbourhood local search for the traveling salesman problem. *Computers & Operations Research* 26, 313-320.
- [71] G. GUTIN, A. YEO (1998A). *TSP heuristics with large domination number*, Technical report No. 12/98, Department of Mathematics and Statistics, Brunel University , Uxbridge, Middlesex.
- [72] G. GUTIN, A. YEO (1998B). *Polynomial approximation algorithms for the TSP and the QAP with a factorial domination number*, To appear in *Discrete Applied Mathematics*.
- [73] G. GUTIN, A. YEO (1999A). Small diameter neighbourhood graphs for the traveling salesman problem: at most four moves from tour to tour. *Computers & Operations Research* 26, 321-327.
- [74] G. GUTIN, A. YEO (1999B). *TSP tour domination and hamilton cycle decompositions of regular digraphs*, preprint Department of Mathematics and Statistics, Brunel University , Uxbridge, Middlesex.

- [75] B. HAJEK (1988). Cooling Schedules for optimal annealing, *Maths of Operational Research* 13, 311-329.
- [76] P. HANSEN (1986). The steepest ascent mildest descent heuristic for combinatorial programming, Talk presented at the congress on Numerical Methods in Combinatorial Optimisation, Capri.
- [77] P. HANSEN AND N. MLADENOVIĆ (1998). Variable neighbourhood search: Principles and applications. Preprint, GERAD and École des Hautes Études Commerciales, Montréal.
- [78] R. HASSIN AND S. RUBINSTEIN (1994). Approximations for the maximum acyclic subgraph problem. *Information Processing Letters* 51, 133-140.
- [79] J.P. HART AND A.W. SHOGAN (1987). Semi-greedy heuristics: An empirical study. *Operations Research Letters* 6, 107-114.
- [80] M. HELD AND R.M. KARP (1970). The traveling-salesman problem and minimum spanning trees. *Operations Research* 18, 1138-1162.
- [81] M. HELD AND R.M. KARP (1971). The traveling-salesman problem and minimum spanning trees: part II. *Mathematical Programming* 1, 6-25.
- [82] J.J. HOPFIELD AND D.W. TANK (1985). ‘Neural’ computation of decisions in optimization problems. *Biological Cybernetics* 52, 141-152.
- [83] J. HURINK (1999). An exponential neighborhood for a one-machine batching problem. *Operations Research Spektrum* 21, 461-476.
- [84] D.S. JOHNSON (1990). Local optimization and the traveling saleman problem. M.S. PATERSON (ED.). *Automata, Languages and Programming*, Lecture Notes in Computer Science 443, Springer, Berlin, 446-461.
- [85] D.S. JOHNSON AND L.A. McGEOCH (1997). The traveling saleman problem: a case study. In *Local Search in Combinatorial Optimization*, eds. E.H.L. AARTS AND J.K. LENSTRA, Wiley, Chichester, UK.
- [86] D.S. JOHNSON, J.L. BENTLEY, L.A. McGEOCH AND E.E. ROTHBERGH (1998). *Near-optimal solutions to very large traveling salesman problems*. In preparation.

- [87] N. KARMARKAR (1984). A new polynomial time algorithm for linear programming. *Combinatorica* 4, 373–395.
- [88] R.M. KARP (1972). Reducibility among combinatorial problems. in R.E. Miller and J.W. Thatcher (eds.), *Complexity of Computer Computations* Plenum Press, New York, 85–103.
- [89] R.M. KARP (1979). A patching algorithm for the nonsymmetric travelling salesman problem. *SIAM Journal of Computing*, 8, 561-572.
- [90] R. KAAS (1981). A branch and bound algorithm for the acyclic subgraph problem. *European Journal of Operational Research* 8, 355-362.
- [91] B.W KERNIGHAN AND S. LIN (1970). An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49 291-307.
- [92] L.G. KHACHIYAN (1997). A polynomial algorithm for linear programming. *Soviet Mathematics Doklady* 20, 191–194.
- [93] S. KIRKPATRICK, C.D. GELATT, AND M.P. VECCHI (1983). Optimisation by simulated annealing. *Science* 220, 671-680.
- [94] P.S. KLYAUS (1976). The structure of the optimal solution of certain classes of travelling salesman problems, (in Russian) *Vestsi Akad. Nauk BSSR, Physics and Math. Sci., Minsk*, 95-98.
- [95] D.E. KNUTH (1993). *The Stanford GraphBase: A Platform for Combinatorial Computing*, Addison Wesley, New York.
- [96] B.H. KORTE (1989). Applications of combinatorial optimization. in M. Iri and K. Tanabe (eds.), *Mathematical Programming: Recent Developments and Applications*, Kluwer, Dordrecht, 1-55.
- [97] M. LAGUNA, R. MARTI, AND V. CAMPOS (1998). Intensification and Diversification with Elite Tabu Search Solutions for the Linear Ordering Problem. Accepted for publication in *Computers and Operations Research*.
- [98] B.J. LAGEWEG, E.L. LAWLER, J.K. LENSTRA AND A.H.G. RINNOOY KAN (1978). *Computer aided complexity classification of deterministic scheduling problems*. unpublication manuscript, Mathematisch Centrum, Amsterdam.

- [99] E.L. LAWLER (1977). A Pseudopolynomial Algorithm for sequencing Jobs to Minimise Total Tardiness, *Annals of Discrete Mathematics* 1, 331–342.
- [100] E.L. LAWLER, J.K. LENSTRA, A.H.G. RINNOOY KAN, AND D.B. SHMOYS (1985). *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, New York.
- [101] J. VAN LEEUWEN AND A.A. SCHOONE (1980). *Untangling a traveling salesman tour in the plane*. Report RUU-CS-80-11, Department of Computer Science, Utrecht University, Utrecht.
- [102] J.K. LENSTRA (1973). *The acyclic subgraph problem*. Report BW26, Mathematisch Centrum, Amsterdam.
- [103] J.K. LENSTRA (1977). *Sequencing by enumerative methods*. Mathematical Centre Tracts 69, Mathematisch Centrum, Amsterdam.
- [104] J.K. LENSTRA, A.H.G. RINNOOY KAN, AND P. BRUCKER (1977). Complexity of Machine Scheduling Problems, *Annals of Discrete Mathematics* 1, 343-362.
- [105] Y. LI, P.M. PARDALOS, AND M.G.C. RESENDE (1994). A greedy randomized adaptive search procedure for the quadratic assignment problem. In *Quadratic assignment and related problems*, eds, P.M PARDALOS AND H. WOLKOWICZ volume 16 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science* American Mathematical Society.
- [106] S. LIN (1965). Computer solutions of the traveling salesman problem. *Bell System Technical Journal* 44, 2245–2269.
- [107] S. LIN AND B.W. KERNIGHAN (1973). An effective heuristic algorithm for the traveling salesman problem. *Operations Research* 21, 498–516.
- [108] H.R.D. LOURENÇO (1995). Job-shop scheduling: computational study of local search and large-step optimization methods. *European Journal of Operational Research* 83, 347-364.
- [109] H.R.D. LOURENÇO AND M. ZWIJNENBURG (1996). Combining the large-step optimization with tabu-search: application to the job-shop scheduling

- problem. In *Meta-Heuristics: Theory and Applications*, EDS. I.H. OSMAN AND J.P. KELLY, Kluwer, Norwell, MA.
- [110] C.L. LUCCHESI (1976). *A minimax equality for directed graphs*. PhD thesis, University of Waterloo, Waterloo, Ontario.
- [111] G. LUEKER (1976). Manuscript, Department of Computer Science, Princeton University, Princeton, NJ.
- [112] V. MANIEZZO (1998). *Exact and approximate nondeterministic tree-search procedures for the quadratic assignment problem*. Technical Report CSR 98-1, Scienze dell'Informazione, Universitá di Bologna, Sede di Cesena.
- [113] V. MANIEZZO (1999). Ant colony Optimization. *Proceedings of the Third Metaheuristics International Conference* 299-303.
- [114] V. MANIEZZO, A. CARBONARO AND R. MONTEMANNI (1999). An approach to frequency assignment problem based on ANTS heuristic. *Proceedings of the Third Metaheuristics International Conference* 311-315.
- [115] O. MARTIN, S.W. OTTO (1996). Combining Simulated annealing with local search heuristics, *Annals of Operations Research* 63, 57-75
- [116] O. MARTIN, S.W. OTTO AND E.W. FELTEN (1991). Large-step Markov chains for the travelling salesman problem. *Complex Systems* 5, 299-326.
- [117] O. MARTIN, S.W. OTTO, AND E.W. FELTEN (1992). Large-step Markov chains for the TSP incorporating local search heuristics. *Operations research letters* 11, 219-224.
- [118] H. MATSUO, C.J. SUH AND R.S. SULLIVAN (1987). *A controlled search simulated annealing method for the single machine weighted tardiness problem*, Working paper 87-12-2, Department of Management, University of Texas at Austin, TX.
- [119] W.S. McCULLOCH AND W.H. PITTS (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 115-133.
- [120] N. METROPOLIS, A. ROSENBLUTH, M. ROSENBLUTH, A. TELLER AND E. TELLER (1953). Equation of state calculations by fast computing machines. *Journal of Chemical Physics* 21, 1089-1092.

- [121] P. MERZ AND B. FREISLEBEN (1997). Genetic local search for the TSP: new results *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation* 159-164.
- [122] T.E. MORTON AND D.W. PENTICO (1993). *Heuristic Scheduling Systems with Applications to Production Systems and Projects Management*, Wiley, Chichester, UK.
- [123] T.E. MORTON, R.M. RACHAMADUGU AND A. VEPSALAINEN (1984). *Accurate Myopic Heuristics for Tardiness Scheduling*, GSIA Working Paper No. 36-83-84, Carnegie-Mellon University, PA.
- [124] T.H. MOTZKIN (1948). Relations between hypersurface cross ratios, and a combinatorial formula for partitions of a polygon, for permanent preponderance, and for non-associative products. *Bulletin of the American Mathematical Society*, 54, 352-360.
- [125] H. MÜHLENBEIN, M. GEORGES-SCHLEUTER, AND O. KRÄMER (1988). Evolution algorithms in combinatorial optimization. *Parallel Computing* 7, 65-85.
- [126] Y. NAGATA, AND S. KOBAYASHI (1997). Edge assembly crossover: A high-power GA for the travelling salesman problem. In *Proc. 7th Int. Conf. on Genetic Algorithms*, 450-457. Morgan Kauffman, San Francisco, CA.
- [127] I. OR (1976). *Traveling salesman-type combinatorial problems and their relation to logistics of regional blood banking*, Ph.D. Thesis, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, Illinois.
- [128] C.H. PAPADIMITRIOU AND K. STEIGLITZ (1982). *Combinatorial optimization: Algorithms and complexity*. Prentice Hall, Englewood Cliffs, New Jersey.
- [129] M. PENN AND Z. NUTOV (1993). Minimum feedback arc set and maximum integral dicycle packing in $K_{3,3}$ -free digraphs.
- [130] E. PESCH AND F. GLOVER (1997). TSP ejection chains. *Discrete Applied Mathematics* 76, 165-181.

- [131] C. PETERSON (1990). Parallel distributed approaches to combinatorial optimization: benchmark studies on traveling salesman problem. *Neural Computation* 2, 261-269.
- [132] C. PETERSON AND B. SÖDERBERG (1989). A new method for mapping optimization problems onto neural networks. *International Journal of Neural Systems* 1, 3-22.
- [133] C. PETERSON AND B. SÖDERBERG (1997). Artifical neural networks. In *Local Search in Combinatorial Optimization*, eds. E.H.L. AARTS AND J.K. LENSTRA, Wiley, Chichester, UK.
- [134] C.N. POTTS AND S.L. VAN DE VELDE (1995). *Dynasearch—iterative local improvement by dynamic programming: part I, The traveling salesman problem*, preprint, Faculty of Mathematical Studies, University of Southampton.
- [135] C.N. POTTS AND L.N. VAN WASSENHOVE (1985). A branch and bound algorithm for the total weighted tardiness problem, *Operations Research* 33, 363-377.
- [136] C.N. POTTS AND L.N. VAN WASSENHOVE (1991). Single Machine Tardiness Sequencing Heuristics. *IEE Transactions* 23, 346-354.
- [137] M. PRAIS AND C.C. RIBEIRO (1998). *Reactive GRASP: An application to a matrix decomposition problem in TDMA traffic assignment*, Technical report, Department of Computer Science, Catholic University of Rio de Janeiro, Rio de Janeiro, RJ 22453-900 Brazil.
- [138] A.P. PUNNEN (1996). *The traveling salesman problem: New polynomial approximation algorithms and domination analysis*. To appear in *Journal of Information and Optimization*
- [139] A.P. PUNNEN AND F. GLOVER (1997). *Implementing ejection chains with combinatorial leverage for the TSP*. preprint Department of Mathematical Statistics and Computer Science, University of New Brunswick, Saint John, New Brunswick, Canada.
- [140] A. PUNNEN AND S.N. KABADI (1999). *Domination analysis of some heuristics for the asymmetric traveling salesman problem*. preprint Department of

Mathematical Statistics and Computer Science, University of New Brunswick,
Saint John, New Brunswick, Canada.

- [141] V. RAMACHANDRAN (1988). Finding a minimum feedback arc set in reducible flow graphs. *Journal of Algorithms* 9, 299-313.
- [142] C.R. REEVES AND J.E. BEASLEY (1995). Introduction, Chapter 1 in *Modern heuristic techniques for combinatorial problems*, ed. C.R. REEVES, McGraw-Hill book company, London.
- [143] C. REGO (1998A). Relaxed tours and path ejections for the traveling salesman problem. *European Journal of Operational Research* 106, 522-538.
- [144] C. REGO (1998B). A subpath ejection chain method for the vehicle routing problem. *Management Science* 44, 1447-1459.
- [145] C. REGO AND C. ROUCAIROL (1996). Parallel tabu search algorithm based on ejection chains for the vehicle routing problem. eds. H.I. Osman, J.P. Kelly *Metaheuristics: Theory and Applications*. Kluwer Academic Publishers, Boston, MA.
- [146] G. REINELT (1985). The linear ordering problem: Algorithms and applications *research and exposition in mathematics* 8 H. H. Hofmann and R. Wille, Heldermann Verlag, Berlin.
- [147] G. REINELT (1991). A Travelling Salesman Problem Library. *ORSA Journal on Computing*, 3 376-384.
- [148] M.G.C. RESENDE (1998). *Greedy adaptive search procedures (GRASP)* Technical report 98.41.1, AT&T labs Research, Florham Park, New Jersey, USA.
- [149] M.G.C. RESENDE (1999). *A bibliography of GRASP* AT&T labs Research, Florham Park, New Jersey, USA.
- [150] A.H.G RINNOOY KAN, B.J LAGEWEG, J.K.LENSTRA (1975). Minimizing total costs in one-machine scheduling. *Operations Research* 23, 908- 927.
- [151] D.J ROSENKRANTZ, R.E. STEARNS AND P.M. LEWIS II (1977). An analysis of several heuristics for the traveling salesman problem. *SIAM Journal of Computing*, 6, 563-581.

- [152] V.I. SARVANOV AND N.N. DOROSHKO (1981A). The approximate solution of the travelling salesman problem by a local algorithm that searches neighborhoods of exponential cardinality in quadratic time(in Russian). *Software: Algorithms and Programs* 31, Mathematical Institute of the Belorussian Academy of Sciences, Minsk, 8-11.
- [153] V.I. SARVANOV AND N.N. DOROSHKO (1981B). The approximate solution of the travelling salesman problem by a local algorithm with scanning neighborhoods of factorial cardinality in cubic time(in Russian). *Software: Algorithms and Programs* 31, Mathematical Institute of the Belorussian Academy of Sciences, Minsk, 11-13.
- [154] E. SCHRÖDER (1870). Vier combinatorische problems. *Z. f. Math. Phys* 15, 361-376.
- [155] N. SIMONETTI (1998). *Applications of a dynamic programming approach to the travelling salesman problem*. Ph.D. Thesis, Carnegie Mellon University, Pittsburgh PA 15213, USA
- [156] N. SIMONETTI AND E. BALAS (1996). Implementation of a linear time algorithm for certain generalized traveling salesman problems. *Proc. IPCO V, Lecture Notes in Computer Science*, Vol. 1084 (Springer, 1996) 316-329.
- [157] STANDARD PERFORMANCE EVALUATION CORPORATION. 1992. Performance results. 10754 Ambassador Drive, Suite 201, Manassas, VA 20109. <http://performance.netlib.org/performance/html/new.spec.cint92.col0.html>
- [158] STANDARD PERFORMANCE EVALUATION CORPORATION. 1995. Performance results. 10754 Ambassador Drive, Suite 201, Manassas, VA 20109. <http://performance.netlib.org/performance/html/new.spec.cint95.col0.html>
- [159] K. STEIGLITZ AND P. WEINER (1968). Some improved algorithms for computer solution of the travelling salesman problem. In *6th Annual Alleton Conference on Circuit and Systems Theory*, (October) 814-821.
- [160] R.H. STORER, S.D. WU AND R. VACCARI (1992). New search spaces for sequencing problems with application to job shop scheduling. *Management Science* 38, 1495-1509.

- [161] T. STÜTZLE (1998A). Applying iterated local search to the permutation flow shop problem. Preprint FB Informatik, Darmstadt, University of Technolgy.
- [162] T. STÜTZLE (1998B) DARMSTADT, UNIVERSITY OF TECHNOLGY. Iterated 3opt code.
- [163] T. STÜTZLE (1998C). An ant approach to the flow shop problem. *Proceedings of EUFIT'98, Aachen*, 1560-1564.
- [164] T. STÜTZLE (1998D). *Local Search Algorithms for Combinatorial Problems – Analysis. Improvements. and New Applications.* PhD thesis, FB Informatik. Darmstadt, University of Technolgy.
- [165] T. STÜTZLE AND M. DORIGO (1999). ACO algorithms for the traveling salesman problem. In *Evolutionary Algorithms in Engineering and Computer Science* eds. K. MIETTINEN, M. MAKELA, P. NEITTAANMAKI AND J. PERIAUX, Wiley.
- [166] T. STÜTZLE AND H. HOOS (1998A). The Max-Min ant system and local search for combinatorial optimisation problems: Towards adaptive tools for combinatorial global optimization. In *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization* eds. S. VOSS, S. MARTELLO, I.H. OSMAN, AND C. ROUCAIROL, Kluwer Academic Publishers.
- [167] T. STÜTZLE AND H. HOOS (1998B). Improvements on the ant system: Introducing Max-Min ant system. In *Artificial Neural Networks and Genetic Algorithms*, eds. G.D. SMITH AND N.C. STEELE.
- [168] T. STÜTZLE AND H. HOOS (1999). Analyzing the run-time behaviour of iterated local search. *Proceedings of the Third Metaheuristics International Conference* 449-453.
- [169] E. TAILLARD (1991). Robust taboo search for the quadratic assignment problem. *Parallel Computing* 17, 443-455.
- [170] S. TIOURINE, C. HURKENS AND J.K. LENSTRA (1995). *An overview of algorithmic approaches to frequency assignment problems.*, EUCLID CALMA Project Overview Report, Delft University of Technology, The Netherlands.

- [171] C. VOUDOURIS (1997). *Guided Local Search for Combinatorial Optimization Problems*. PhD Thesis, Department of Computer Science, University of Essex, Colchester, UK.
- [172] C. VOUDOURIS AND E. TSANG (1998). Solving the radio link frequency assignment problems using guided local search *In Proceedings of NATO symposium on Frequency Assignment, Sharing and Conservation in Systems (AEROSPACE) AGARD*, paper No. 14, Aalborg, Denmark.
- [173] C. VOUDOURIS AND E. TSANG (1999). Guided local search and its application to the traveling salesman problem, *European Journal of Operational Research* 113, 469-499.
- [174] C.J. WANG AND E. TSANG (1991). Solving constraint satisfaction problems using neural-networks, *Proceedings of IEE Second International Conference on Artificial Neural Networks*, 295-299.
- [175] J. WATSON, C. ROSS, V. EISELE, J. DENTON, J. BINS, C. GUERRA, D. WHITLEY AND A. HOWE (1998). The Travelling Salesrep problem, edge assembly crossover and 2-opt. *Parallel Problem-Solving from Nature-PPSN V*, 823-832. Springer-Verlag, Berlin
- [176] M. YANNAKAKIS (1997). Computational Complexity Chapter 2 in *Local Search in Combinatorial Optimization*, eds. E.H.L. AARTS AND J.K. LENSTRA, Wiley, Chichester, UK.
- [177] A. YEO (1997). *Large exponential neighbourhoods for the traveling salesman problem*, preprint no. 47, Department of Mathematics and Computer Science, Odense University, Odense, Denmark.