Chapter 8

LOCAL SEARCH AND METAHEURISTICS

César Rego
Hearin Center for Enterprise Science
School of Business Administration, University of Mississippi, MS 38677
crego@bus.olemiss.edu

Fred Glover
Hearin Center for Enterprise Science
School of Business Administration, University of Mississippi, MS 38677
fglover@bus.olemiss.edu

1. Background on Heuristic Methods

The Traveling Salesman Problem (TSP) is one of the most illustrious and extensively studied problems in the field of Combinatorial Optimization. Covering just the period from 1993 to mid-2001 alone, the web databases of INFORMS and Decision Sciences report more than 150 papers devoted to the TSP. The problem can be stated in graph theory terms as follows. Let $G = (V, A)$ be a weighted complete graph, where $V = \{v_1, \ldots, v_n\}$ is a vertex (node) set and $A = \{(v_i, v_j) | v_i, v_j \in V, i \neq j\}$ is an edge set. $C = [c(v_i, v_j)]$ is a $n * n$ matrix associated with A, where $c(v_i, v_j)$ is a non-negative weight (distance or cost) on edge (v_i, v_j) if there is an edge between v_i and v_j. Otherwise $c(v_i, v_j)$ is infinity.

The problem is said to be symmetric (STSP) if $c(v_i, v_j) = c(v_j, v_i)$ for all $(v_i, v_j) \in A$, and asymmetric (ATSP) otherwise. Elements of A are often called arcs (rather than edges) in the asymmetric case. The STSP (ATSP) consists of finding the shortest Hamiltonian cycle (circuit) in G, which is often simply called a *tour*. In the symmetric case, G is an *undirected* graph, and it is common to refer to the edge set $E = \{(v_i, v_j) | v_i, v_j \in V, i < j\}$ in place of A. The version of STSP in which distances satisfy the triangle inequality $(c(v_i, v_j) + c(v_j, v_k) \geq c(v_i, v_k)$ for all distinct $v_i, v_j, v_k \in V)$ is perhaps the most-studied special case of

G. Gutin et al. (eds.), *The Traveling Salesman Problem and Its Variations*

the problem, notably including the particular instance where V is a set of points in a 2-dimensional plane and $c(v_i, v_j)$ is the Euclidean distance between v_i and v_j.

Important variants and extensions of the TSP arise in the setting of vehicle routing (see Laporte and Osman [539]). A variety of other interesting problems not directly related to routing can also be modeled as TSPs, as is shown in the survey of Laporte [533]. Distances or costs that are symmetric and satisfy the triangle inequality are predominantly encountered in such applications. This chapter mainly deals with the STSP and for the sake of simplicity we will generally refer to the problem as the TSP.

The TSP can be formulated as an integer linear programming (ILP) model, and a number of exact algorithms are based on such a formulation. However, there are also some advantages to representing the TSP directly as a permutation problem without transforming it into an ILP, and we will focus on such a direct representation. Let π denote a cyclic permutation mapping so that the sequence $i, \pi_{(i)}, \pi^2_{(i)}, \ldots, \pi^{n-1}_{(i)}$ for $i \in N = \{1, \ldots, n\}$ identifies a permutation of the elements of N, where $\pi^n_{(i)} = i$. Let Π be the set of all such mappings. Thus, solving a particular instance of the TSP consists of finding a cycle permutation (tour) $\pi^* \in \Pi$ such that

$$\sum_{i=1}^{n-1} c_{i\pi^*_{(i)}} = \min_{\pi \in \Pi} \sum_{i=1}^{n-1} c_{i\pi_{(i)}}$$

The TSP is one of the classical instances of an NP-complete problem, and therefore there is no polynomial-time algorithm able to determine π^* for all possible instances of the problem (unless P=NP). Perhaps because of the simplicity of its statement the TSP has been a key source of important developments in NP-completeness theory (see e.g., Johnson and Papadimitriou, 1985). It has also been the subject of several polyhedral studies (see Chapters 2 and 3). As a result, although P=NP continues to be an improbable hypothesis, and hence a polynomial-time algorithm for the TSP is not likely to be discovered, current specialized TSP optimization codes have been solving general TSP instances involving about three-thousand vertices. Specifically, the Concorde package of Applegate, Bixby, Chvatal and Cook [29] solved all instances up to 3200 cities in the 8^{th} DIMACS TSP Challenge testbed (Johnson, McGeoch, Glover, and Rego [462]) using its default settings, except one 3162-city random uniform Euclidian instance for which non-default twiddling was necessary to find the optimal tour.

State-of-the-art exact solution methods (which guarantee an optimal solution if run long enough) can typically solve problems involving about 1000 vertices in reasonable computation time, but encounter significant difficulties in solving larger problems, where they generally require computational effort that exceeds the realm of practicality. Even for modest-size problems, exact methods require substantially greater computation time than the leading heuristic methods, which in addition are capable of finding optimal or very-close-to-optimal solutions for instances far larger than those reasonably attempted by exact methods. An extensive empirical analysis of computational results and algorithmic performance for several classes of TSP heuristics is described in Chapter 9.

The aim of this chapter is to present an overview of classical and modern local search procedures for the TSP and discuss issues involved in creating more efficient and effective algorithms. Heuristic algorithms for the TSP can be broadly divided into two classes: *tour construction procedures*, which build a tour by successively adding a new node at each step; and *tour improvement procedures*, which start from an initial tour and seek a better one by iteratively moving from one solution to another, according to adjacency relationships defined by a given neighborhood structure. (Specialized tour construction heuristics are treated in Chapter 9.) Combined, such approaches yield *composite procedures* that attempt to obtain better solutions by applying an improvement procedure to a solution given by a construction procedure. Often, the success of these algorithms depends heavily on the quality of the initial solution. Iterated variants of construction and improvement procedures provide a natural means of elaborating their basic ideas, as subsequently discussed.

Recent Developments in Overview. Recent progress in local search methods has come from designing more powerful neighborhood structures for generating moves from one solution to another. These advances have focused on *compound neighborhood structures*, which encompass successions of interdependent moves, rather than on simple moves or sequences of independent moves. On the other hand, the more sophisticated neighborhood structures entail greater numbers of operations, and therefore an increased effort to perform each step of the algorithm. Thus, several studies have investigated strategies to combine neighborhoods efficiently, and thereby reduce the computational effort of generating trajectories within them. These methods are generally *variable depth methods*, where the number of moves carried out at each iteration is dynamically determined, and usually varies from one iteration to the next. A common characteristic of these methods is a *look ahead* process where a relatively large sequence of moves is generated, each step leading

to a different trial solution, and the compound move that yields the best trial solution (from the subsequences beginning with the initial move) is the one chosen. Two types of variable depth neighborhood structures have become prominent:

(1) *connected neighborhood structures* as represented by:

- **a.** Variable Neighborhood Search (VNS) (Mladenović and Hansen [602], Hansen and Mladenović [433, 432]),
- **b.** Sequential Fan (SF) methods (Glover and Laguna [379]),
- **c.** Filter and Fan (FF) methods (Glover [376]);

(2) *disconnected neighborhood structures* as represented by:

- **a.** Lin-Kernighan (LK) methods (Lin and Kernighan [563]),
- **b.** Chained and Iterated LK methods (Martin, Otto and Felten [588], Johnson and McGeogh [463], Applegate, Cook and Rohe [32]),
- **c.** Ejection Chain (EC) methods (Glover [372], [374]).

In the TSP setting, connected neighborhood procedures are exemplified at a simple level by classical k-opt and Or-opt methods which keep the Hamiltonian (feasible tour) property at each step. Variable depth methods of these types consist of component moves that directly link one tour to the next, thus generating streams of moves and associated trial solutions. Conversely, the LK and EC methods consider sequences of moves that do not necessarily preserve the connectivity of the tour, although they enable a feasible tour to be obtained as a trial solution by performing an additional move. Apart from this commonality, Lin-Kernighan and Ejection Chains differ significantly in the form of the intermediate (disconnected) structures that link one move to the next in the sequence. LK methods rely on a Hamiltonian path as an intermediate structure, while EC methods embrace a variety of intermediate structures, each accompanied by appropriate complementary moves to create feasible trial solutions. The Lin-Kernighan procedure is described in Section 2.2. Ejection chains structures and the moves that join and complement them are elaborated in Section 2.3, followed by advanced variable depth methods in Section 2.4.

Local Search and Meta-Heuristic Approaches. Local search techniques (which terminate at a local optimum) and associated meta-heuristic strategies (which modify and guide local techniques to explore the solution space more thoroughly) have been the focus of widespread scientific investigation during the last decade. For more than twenty years

two main "meta models" for heuristic techniques have been ascendant: those based on "single stream" trajectories, and those based on "multiple stream" trajectories, where the latter seek to generate new solutions from a collection (or population) of solutions. The distinction is essentially the same as that between serial and parallel algorithms, with the allowance that population-based methods can also be applied in a serial manner, as in a serial simulation of a parallel approach. Consequently, as may be expected, there are some overlaps among the best procedures of these two types. Traditionally, however, population-based methods have often been conceived from a narrower perspective that excludes strategies commonly employed with single stream methods. Thus, more modern approaches that embody features of both methods are often called hybrid procedures.

Some of the methods discussed in this chapter have fairly recently come into existence as general purpose methods for a broad range of combinatorial optimization problems, and have undergone adaptation to provide interesting specializations for the TSP. This manifests one of the reasons for the enduring popularity of the TSP: it often serves as a "problem of choice" for testing new methods and algorithmic strategies.

The remainder of this chapter is organized as follows. Section 2 presents classical and more recent improvement methods that have proven effective for the TSP. It also discusses several special cases of neighborhood structures that can be useful for the design of more efficient heuristics. Section 3 gives an overview of the tabu search metaheuristic, disclosing the fundamental concepts and strategies that are relevant in the TSP context. Section 4 extends the exploration of metaheuristics to the description and application of recent unconventional evolutionary methods for the TSP. Section 5 presents some concluding observations and discusses possible research opportunities.

2. Improvement Methods

Broadly speaking, improvement methods are procedures that start from a given solution, and attempt to improve this solution by iterative change, usually by manipulating relatively basic solution components. In graph theory settings, depending on the problem and the type of algorithm used, these components can be nodes, edges, (sub)paths or other graph-related constructions. We consider three classes of improvement methods according to the type of neighborhood structures used:

(1) *constructive neighborhood methods*, which successively add new components to create a new solution, while keeping some components of the current solution fixed. (These include methods that assem-

ble components from different solutions, and methods that simply choose different parameters of a construction procedure using information gathered from previous iterations.)

(2) *transition neighborhood methods*, usually called local search procedures, which iteratively move from one solution to another based on the definition of a neighborhood structure.

(3) *population-based neighborhood methods*, which generalize (1) and (2) by considering neighborhoods of more than one solution as a foundation for generating one or more new solutions.

In this section we focus our discussion on the second class of improvement methods and specifically on those that are either considered classical or the core of the most efficient TSP algorithms known to date.

For the following development we assume a starting TSP tour is given and is recorded by identifying the immediate predecessor and successor of each node v_i, which we denote respectively v_{i-} and v_{i+}.

2.1. Basic Improvement Procedures

Fundamental neighborhood structures for the TSP (and for several other classes of graph-based permutation problems) are based on edge-exchanges and node-insertion procedures. Classical procedures of these types are the k-exchange (Lin [562]) and the Or-insertion (Or [635]) which also form the core of several more advanced procedures covered in the next sections. Before describing the various neighborhood structures underlying these two classes of procedures, it is appropriate to note that the concept of local optimality has a role in the nomenclature of k-Opt and Or-Opt – terms sometimes used inappropriately in the TSP literature. In a local search method a neighborhood structure is introduced to generate moves from one solution to another and, by definition, a local optimum is a solution that can not be improved by using the neighborhood structure under consideration. Accordingly, a local optimum produced by an improvement method using k-exchanges or Or-insertion yields what is called a k-optimal (k-Opt) or a Or-optimal (Or-opt) solution, respectively.

2.1.1 k-exchange Neighborhoods. The terminology of k-exchange neighborhoods derives from methods initially proposed by Lin [562] to find so-called "k-opt" TSP tours. The 2-exchange (2-opt) procedure is the simplest method in this category and is frequently used in combinatorial problems that involve the determination of optimal cir-

cuits (or cycles) in graphs. This includes the TSP and its extensions to the wider classes of assignment, routing and scheduling problems.

The 2-opt procedure is a local search improvement method, and a starting feasible solution is required to initiate the approach. The method proceeds by replacing two non-adjacent edges (v_i, v_{i+}) and (v_j, v_{j+}) by two others (v_i, v_j) and (v_{i+}, v_{j+}), which are the only other two edges that can create a tour when the first two are dropped. In order to maintain a consistent orientation of the tour by the predecessor-successor relationship, one of the two subpaths remaining after dropping the first two edges must be reversed. For example, upon reversing the subpath $(v_{i+}, \ldots, v_j)$ the subpath $(v_i, v_{i+}, \ldots, v_j, v_{j+})$ is replaced by $(v_i, v_j, \ldots, v_{i+}, v_{j+})$. Finally, the solution cost change produced by a 2-exchange move can be expressed as $\Delta_{ij} = c(v_i, v_j) + c(v_{i+}, v_{j+}) - c(v_i, v_{i+}) - c(v_j, v_{j+})$. A 2-optimal (or 2-opt) solution is obtained by iteratively applying 2-exchange moves until no possible move yields a negative Δ value.

The 2-opt neighborhood process can be generalized to perform k-opt moves that drop some k edges and add k new edges. There are $\binom{n}{k}$ possible ways to drop k edges in a tour and $(k-1)!2^{k-1}$ ways to relink the disconnected subpaths (including the initial tour) to recover the tour structure. For small values of k, relative to n, this implies a time complexity of $O(n^k)$ for the verification of k-optimality, and therefore the use of k-opt moves for $k > 3$ is considered impractical unless special techniques for restricting the neighborhood size are used. (To date, $k = 5$ is the largest value of k that has been used in algorithms for large scale TSPs.) We now summarize some of the main advances in the design of more efficient k-opt procedures.

2.1.2 Special Cases of k-opt Neighborhoods. A useful observation for implementing restricted k-opt moves is that any k-opt move for $k > 2$ is equivalent to a finite sequence of 2-opt moves, assuming the graph is fully dense. (This is a result of the easily demonstrated fact that in such a graph any tour can be transformed into any other by a succession of 2-opt moves.) Consequently, if no sequence of k consecutive 2-opt moves can improve the current tour, then it is also a k-optimal tour. However, the reverse is not necessary true – i.e. a tour can be k-optimal, but obviously there may exist a sequence of k successive 2-opt moves that reduces the length of the tour (since every tour can be reached in this way in a fully dense graph). Thus, a comparative analysis of neighborhoods with successive k values provides a foundation for designing more efficient k-opt procedures by restricting the attention to moves that are not included within $(k-1)$-opt neighborhoods.

By direct analogy to the 2-opt move, a 3-opt move consists of deleting three edges of the current tour (instead of two) and relinking the endpoints of the resulting subpaths in the best possible way. For example, letting (v_i, v_{i+}), (v_j, v_{j+}) and (v_k, v_{k+}) denote the triplet of edges deleted from the current tour, two of the seven possible ways to relink the three subpaths consist of (1) creating edges (v_i, v_{j+}), (v_k, v_{i+}), and (v_j, v_{k+}); and (2) creating edges (v_i, v_j), (v_{j+}, v_{k+}), and (v_{i+}, v_k).

An important difference between these two possibilities to create 3-opt moves is that the orientation of the tour is preserved in (1), while in (2) the subpaths $(v_{i+}, \ldots, v_j)$ and $(v_{j+}, \ldots, v_k)$ have to be reversed to maintain a feasible tour orientation. The cost of a 3-opt move can be computed as Δ_{ijk}, the sum of the costs of the added edges minus the sum of the costs of the deleted edges, where a negative Δ represents an improving move. Generically, similar computations and conditions for reversing subpaths result for any k-opt move.

Another way to reduce the time complexity of 3-opt moves comes from the observation that a 2-opt move is a special case of a 3-opt move in which a deleted edge is added back to relink a subpath. Consequently, three of the seven possible 3-opt moves correspond to 2-opt moves. Thus, if the tour is already 2-optimal, then these three types of 2-exchange moves need not be checked in the 3-opt process. An additional class of 3-opt moves can be obtained as a sequence of two (non-independent) 2-opt moves. This special case may occur when an edge inserted by the first 2-opt move is deleted by the application of the second 2-opt move. Three other 3-opt moves fall into this special case. Figure 8.1 illustrates one of these possibilities, applied to the TSP tour given in Figure 8.1A. The 3-opt move is represented in Figure 8.1B where the symbol e_0 is used to label the edges deleted by the move. Similarly, Figure 8.1C illustrates the application of two successive 2-opt moves where e_0 and e_1 are the edges deleted by the first and the second application of the move. Note that edge e_1 is one of the edges added by the first application of the 2-opt move. Figure 8.1D represents the TSP tour that results from the application of either the 3-opt move or the indicated sequence of two 2-opt moves.

The foregoing observations indicate that out of the seven possible 3-opt moves only one requires a sequence of three 2-opt moves, so that only a very small fraction of the $\begin{pmatrix} 3 \\ k \end{pmatrix}$ possible combinations need to be considered. Also, as described in Christofides and Eilon [190] a 4-opt neighborhood (which involves 47 possible 4-opt moves) includes six 2-opt

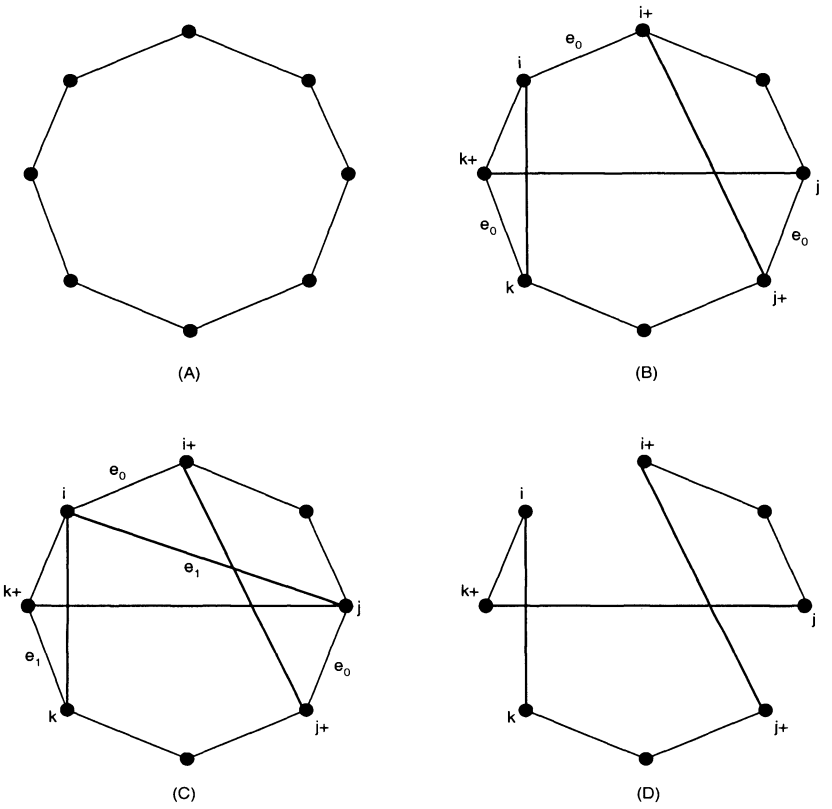

Figure 8.1. 3-opt obtained by two successive 2-opt moves.

moves, sixteen sequences of two 2-opt moves, and twenty-five sequences of three 2-opt moves. Consequently, the consideration of sequences of three 2-opt moves derived from interconnecting restricted 2-opt moves in successive levels is sufficient to ensure the tour is 4-optimal. In contrast, the determination of a 5-optimal tour requires examination of a sequence of at least five 2-opt moves, yielding significantly more combinatorial possibilities than the sequences of three 2-opt moves required by 3-opt and 4-opt neighborhoods. This provides an indication of the relatively greater advantage of 5-opt neighborhoods over 4-opt neighborhoods when compared to the advantages of 4-opt over 3-opt, as demonstrated by Christofides and Eilon [190] and more recently by Helsgaun [446]. A multi-stage 2-opt approach appears particularly useful to implement variable depth methods, as discussed in Section 2.4.

2.1.3 Special Cases of Insertion and k-Opt Neighborhoods.

Another useful relationship emerges from the comparative analysis of k-opt moves relative to several classes of insertion moves. We define two basic types of node-based moves within the TSP setting:

(1) *node insertion moves*: a selected node v_i is inserted between two adjacent nodes v_p and v_q in the tour by adding edges (v_p, v_i), (v_i, v_q), (v_{i-}, v_{i+}) and dropping edges (v_p, v_q), (v_{i-}, v_i), (v_i, v_{i+}).

(2) *node exchange moves*: two nodes v_i and v_j exchange positions by adding edges (v_{i-}, v_j), (v_j, v_{i+}), (v_{j-}, v_i), (v_i, v_{j+}) and dropping edges (v_{i-}, v_i), (v_i, v_{i+}), (v_{j-}, v_j), (v_j, v_{j+}). An exception occurs if (v_i, v_j) is an edge of the tour, in which case the move is equivalent to inserting v_i between v_j and v_{j+} (or inserting v_j between v_{i-} and v_i).

Or-Opt Neighborhoods and Extensions. A generalization of the foregoing node-based neighborhoods consists of extending these processes to insert and exchange sequences (or subpaths) of consecutive edges in the tour. By treating subpaths as if they were nodes this generalized process can be implemented using operations similar to the ones defined for the node insertion/exchange moves.

Two classical methods that seek to reduce the complexity of the 3-opt procedure are Bentley's 2.5-opt and Or-opt (Or [635]). 2.5-opt is an extension of the 2-opt procedure that considers a single-node insertion move when 2-opt fails to improve (Bentley [103]). The Or-opt heuristic proceeds as a multi-stage generalized insertion process, which starts by considering the insertion of three-node subpaths (between two adjacent nodes) and then successively reduces the process to insert two-node subpaths (hence edges) and finally to insert single nodes, changing the type

of move employed whenever a local optimum is found for the current neighborhood. Note that node-insertion and edge-insertion moves are special cases of 3-opt moves when a subpath between two dropped edges of the 3-opt move consists of just one node or edge, respectively. Also, as mentioned before, most 3-opt moves do not preserve the orientation of the tour. Now, it is easy to see that the Or-opt procedure restricts the 3-opt neighborhood to a subclass of moves that preserves the current tour orientation. The time complexity of this procedure is $O(n^2)$. However, while the Or-opt neighborhood has proved relatively efficient when applied to some constrained traveling salesman and vehicle routing problems, the procedure does not appear to present a competitive advantage when compared to efficient implementations of the 3-opt procedure. (Chapter 9 provides details on efficient implementations of 3-opt.) A possible enhancement of the classical Or-opt procedure arises from a generalization based on an ejection chain framework, as we discuss in Section 2.3.2. Such a generalization gives the procedure the ability to create a variable depth neighborhood search similar to the one the Lin-Kernighan procedure performs with classical k-opt moves.

Constructive/Destructive Neighborhoods for Restricting k-opt Moves. Gendreau, Hertz, and Laporte [351] propose a generalized insertion procedure (GENI) which may be viewed as a combination of single-node insertion moves with 4-opt and 5-opt moves. GENI is used in the constructive phase of their GENIUS algorithm to create a starting tour, beginning from an arbitrary cycle of 3 vertices. The alternating use of GENI with its reverse procedure (destructively removing nodes from the tour) forms the basis of the String/Unstring (US) neighborhood structure used in the local search phase of the GENIUS algorithm. Thus, the successive application of Unstring and String creates a destructive/constructive type of neighborhood structure that typically generates restricted forms of 8-opt, 9-opt, and 10-opt moves. The process can be viewed as a one-step (unit depth) strategic oscillation (see Section 3.3).

The destructive *Unstring* neighborhood structure removes a vertex from the current tour by replacing k edges by $k-1$ other edges for $k = 4$ or 5. Figure 8.2 depicts an example of the *Unstring* process where node v_i is removed from the tour. In the figure, diagrams A and C represent the initial tours to apply an Unstring process with $k = 4$ and $k = 5$, respectively. Diagrams B and D represent the resulting subgraphs after applying the *Unstring* move, which removes the edges labeled "e" and relinks the respective subpaths as illustrated. In this procedure, edges defined by nodes identified with different letters generically represent subpaths having these nodes as their endpoints.

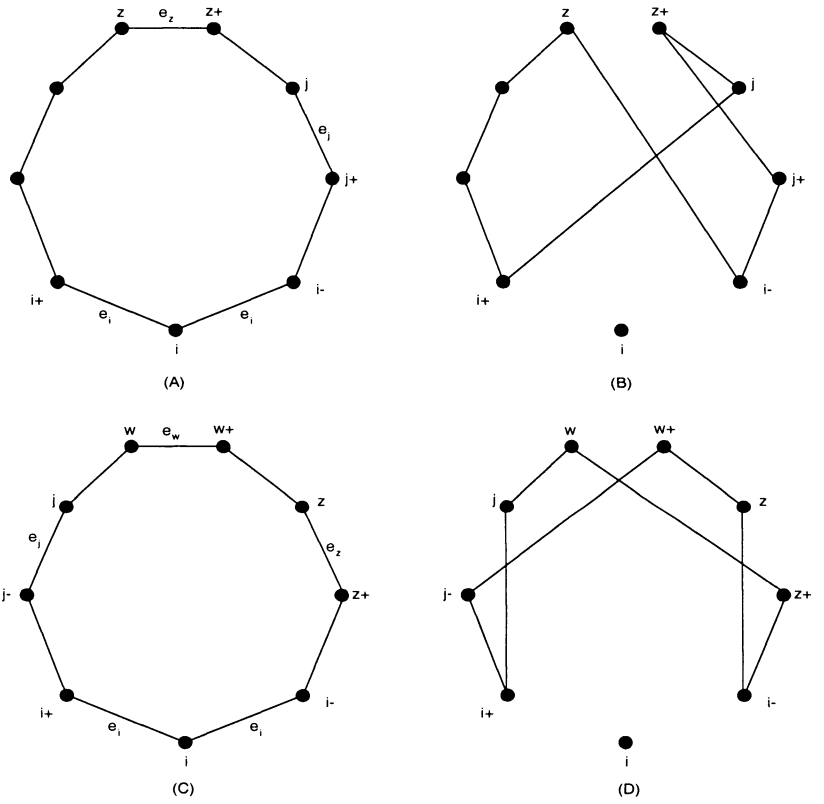

Figure 8.2. The String/Unstring Neighborhood Structure

String is a constructive neighborhood structure that reverses the operations of the *Unstring* procedure to insert a disconnected vertex v_i between two other vertices (not necessarily consecutive in the tour) by replacing k edges by $k+1$ other edges for $k = 3$ or 4. Figure 8.2 illustrates *String* moves for $k = 3$ and $k = 4$ by following the diagrams in the order from B to A and from D to C, respectively.

Recent Results on Restricting k-Opt Neighborhoods. Useful results for reducing computation required by k-opt procedures are provided by Glover [375] and Helsgaun [446]. Glover's paper shows that the best move from a sub-collection of 4-opt moves (which embraces all 2-opt moves, an additional class of 3-opt moves, and two principal classes of 4-opt moves) can be found in the same order of time required to find a best 2-opt move. The method is based on an acyclic shortest path model underlying the creation of *dynamic alternating paths and cycles* generated by an ejection chain framework as discussed in Section 2.3. The use of ejection chains to generate special forms of alternating paths and cycles also proves useful in the implementation of the stem-and-cycle ejection chain method described in Rego [704] and discussed in Section 2.3.3. Helsgaun considers the use of 5-opt moves to replace 2-opt moves in the basic step of the classic implementation of the Lin-Kernighan (LK) procedure as discussed in Section 2.2. In Helsgaun's particular variant of the LK procedure 5-opt moves are made computationally practicable by restricting the possible alternatives using special candidate lists, in this case augmenting a "close-by neighbor" list to include additional nodes identified by solving Lagrangean relaxations over minimum spanning tree (1-tree) relaxations as introduced by Held and Karp ([444, 445]).

2.2. The Classical Lin-Kernighan Procedure

The Lin-Kernighan (LK) procedure (Lin and Kernighan [563]) is a strategy for generating k-opt moves where the value of k is dynamically determined by performing a sequence of 2-opt moves. Although, as noted, any k-opt move can be represented by a sequence of 2-opt moves, the LK procedure limits attention to a particular subset of these sequences. The goal is to restrict the neighborhood search and at the same time to generate high quality k-opt moves. The 2-opt moves are generated in successive levels where a 2-opt move of level i ($i = 2, \ldots, L, k = i+1$) drops one of the edges that has been added by the 2-opt move of the previous level ($i-1$). An exception is made for the first 2-opt move of the sequence, which can start either from the current tour or after performing a special class of 3-opt or 4-opt moves. However, this excep-

tion is only allowed when a sequence of 2-opt moves in the regular LK search fails to improve the initial tour. These special cases will become clear later in the detailed explanation of each step of the method.

The method starts by generating a 2-opt, 3-opt or 4-opt move and then deletes an edge adjacent to the last one added to create a Hamiltonian path. Thereafter, each new move consists of adding an edge to the degree 1 node that was not met by the last edge added, and dropping the unique resulting edge that again will recover a Hamiltonian path (thus completing the last add-drop portion of a 2-opt move).

It is interesting to note that these moves (and in fact the first add-drop half of any 2-opt move) "pass through" a *stem-and-cycle structure*, which is one of the reference structures introduced by Glover [374] as a foundation for creating more flexible and general types TSP moves. However, this structure is not identified as relevant in the LK method, which relies solely on the *Hamiltonian path structure* as a basis for composing its moves, and consequently is left unexploited within the LK framework. (The expanded set of possibilities provided by the stem-and-cycle structure is described in Section 2.3.3.)

Figure 8.3 illustrates the three types of starting moves for initiating the LK approach (2, 3 and 4-opt moves).

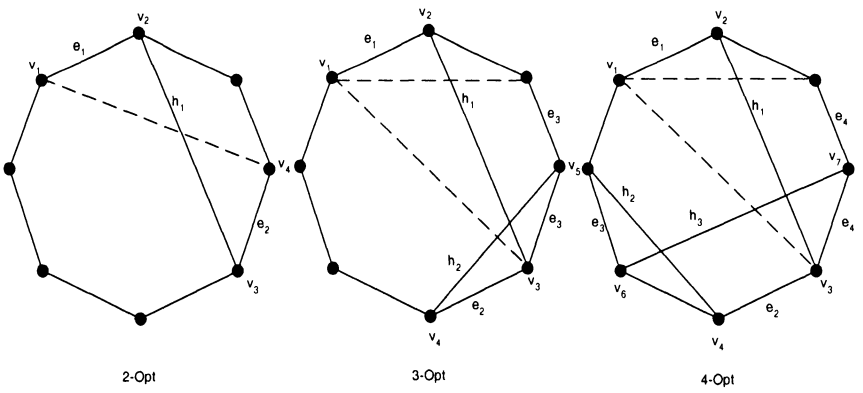

Figure 8.3. Possible moves at the first level of the Lin-Kernighan process

The initial k-opt moves ($k = 2, 3, 4$) that are used to launch the LK procedure can be created as follows, as illustrated by the diagrams in Figure 8.3, preceding:

$k = 2$ – The first step of the LK procedure selects the first edges $e_1 = (v_1, v_2)$ and $h_1 = (v_2, v_3)$ to drop and add (respectively), so that

they produce a cost improvement, and hence yield a negative value of $E_1 = c(v_2, v_3) - c(v_1, v_2)$. (Such a choice must be possible if a better tour exists.) A first level 2-opt move is obtained by dropping edge $e_2 = (v_3, v_4)$ where v_4 is in the tour neighbor of v_3 that is in the cycle $(v_2, v_3, v_4, \ldots, v_2)$ created when edge h_1 was added. This last edge-drop operation implicit when h_1 is chosen creates a Hamiltonian path $H_1 = (v_1, \ldots v_3, v_2, \ldots, v_4)$ that is used to close up the tour by adding edge (v_4, v_1). Compute $T_1 = c(v_4, v_1) - c(v_3, v_4)$ and examine the solution cost change by computing $\Delta_1 = E_1 + T_1$.

$k = 3$ – Accompanying the initial drop-add choice that removes e_1 and adds h_1, drop edge $e_2 = (v_3, v_4)$ where v_4 is adjacent to v_3. There are two possibilities to create a 3-opt move: (1) if v_4 is in the cycle, the move is direct extension of the LK process from the level 1 to level 2; (2) if v_4 is the endpoint of the path from v_1 to v_4, create a Hamiltonian path H_2 by linking v_4 to one vertex v_5 (corresponding to edge $h_2 = (v_4, v_5)$) and drop one of its adjacent edges $e_3 = (v_5, v_6)$ where v_5 is a vertex in the cycle $(v_2, v_3, \cdots, v_2)$. Link v_6 to v_1 to create a tour. There are some subtleties in the way edge e_3 is chosen. When v_4 is in the path, the method selects v_5 by computing $v_5 = \text{argmin}\{c(v_4, v_5) - \max\{c(v_5, v_{5-}), c(v_5, v_{5+})\}\}$. Once v_5 is chosen two trial values are examined to select v_6 by computing $v_6 = \text{argmin}\{c(v_6, v_1) - c(v_5, v_6) : v_6 = v_{5-}, v_{5+}\}$. (The computation of v_6 in the way just defined is equivalent to finding the better of the two trial tours produced by adding the links (v_{5-}, v_1) and (v_{5+}, v_1), as suggested in the original Lin and Kernighan paper.) The corresponding cost changes are given by $E_2 = E_1 + c(v_4, v_5) - c(v_3, v_4)$, $T_2 = c(v_6, v_1) - c(v_5, v_6)$, resulting in a tour cost change of $\Delta_2 = E_2 + T_2$.

$k = 4$ – As in the $k = 2$ and $k = 3$ cases, begin by dropping e_1 and adding h_1, and then drop edge $e_2 = (v_3, v_4)$ where v_4 is adjacent to v_3. There are two possibilities to create a 4-opt move: (1) if v_4 is in the cycle, the move is direct extension of the LK process from the level 2 to level 3; (2) if v_4 is the endpoint of the path from v_1 to v_4, create a Hamiltonian path H_3 in two steps: (i) link v_4 to a vertex v_5 in the path (creating edge $h_2 = (v_4, v_5)$) and drop its adjacent edges $e_3 = (v_5, v_6)$ where v_6 is in the path $(v_5, v_6, \ldots, v_4)$; (ii) link v_6 to one vertex v_7 in the cycle (creating $h_3 = (v_6, v_7)$) and drop one of its adjacent edges $e_4 = (v_7, v_8)$. Link the endpoints of the resulting Hamiltonian path H_3 to create a tour. Edge e_3 is chosen by setting $e_3 = \text{argmin}\{c(v_4, v_5) - \max\{c(v_5, v_{5-}), c(v_5, v_{5+})\} : v_6 = v_{5-}, v_{5+}\}$. Once e_3 is chosen, edge e_4 is selected by computing $e_4 = \text{argmin}\{c(v_6, v_7) - \max\{c(v_7, v_{8-}), c(v_7, v_{8+})\} : v_8 = v_{7-}, v_{7+}\}$. (In this case vertices v_7 and v_8 are selected at the same time simply by choosing the largest cost edge incident at v_7, which implies that only one trial tour will be examined as suggested in the original paper.) The corresponding

cost changes are given by $E_3 = E_1 + c(v_4, v_5) + c(v_6, v_7) - c(v_3, v_4)$, $T_3 = c(v_8, v_1) - c(v_7, v_8)$, and consequently the tour cost change is given by $\Delta_3 = E_3 + T_3$.

Extending LK moves to further levels. As mentioned, the regular LK search (which does not include the special 3-opt and 4-opt moves) consists of generating k-opt moves ($k \geq 2$) (as a sequence of 2-opt moves) in successive levels where at each level a new vertex is selected to extend the process to the next level by recovering a Hamiltonian path H_i (dropping the last link (v_{2i}, v_1)) and using this path as a reference structure to generate a new path H_{i+1} of the same type (by linking its endpoint to another vertex in the path and dropping one of its adjacent vertices). Starting with $E_0 = 0$, for a fixed vertex v_{2i} at a level i of the LK search, a new vertex v_{2i+1} is chosen in such a way that $E_i = E_{i-1} + c(v_{2i}, v_{2i+1}) - c(v_{2i-1}, v_{2i})$ yields a negative value. (An exception is made for some special alternative 3-opt and 4-opt moves as explained above.) Consequently, a trial move (that yields a trial tour) for the level $i (i > 0)$ can be obtained by computing $T_i = c(v_{2i+2}, v_1) - c(v_{2i+1}, v_{2i+2})$. Similarly, the total tour cost change is given by $\Delta_i = E_i + T_i$. At each level i of the LK process, the method keeps track of the minimum Δ value (corresponding to the best trial tour) and records the sequence of the vertices associated with the e_i and h_i edges, which will be used to perform a move at the end of the LK search.

The LK procedure also considers a backtracking process which allows the method to repeat, each time choosing a different starting vertex associated with an untried alternative for inserting or deleting an edge $h_i = (v_i, v_{i+1})$ or $e_i = (v_{2i-1}, v_{2i})$, respectively. Such alternatives (which include the special 3-opt and 4-opt moves) are successively examined starting from level i and proceeding back to level 1, where the examination of edges h_i and e_i is alternated so that a candidate for edge e_i is examined after exhausting all possible candidates for h_i without finding any improvement of the starting tour. Candidates for level $i - 1$ are examined after exploring all alternatives for level i. As soon as an improvement is found at some level, the backtracking process stops and the LK process continues from that level (and its accompanying e_i or h_i choice), progressing forward again to higher levels as previously described. If no improvement occurs in the backtracking process, including those cases where alternative choices for the vertex v_1 are examined, the procedure stops and returns the best tour found so far. (In the original paper backtracking is used only for $i = 1$ and 2.)

A refinement of the basic LK method considers a look-ahead process where the choice of a current h_i edge takes into consideration the cost of the associated e_{i+1} edge. The evaluation criterion (via this "look-ahead" rule) is the one that minimizes $E_i = E_{i-1} + c(v_{2i}, v_{2i+1}) - c(v_{2i+1}, v_{2i+2})$ for a fixed vertex $v_{2i} (i > 1)$, which gives the shortest Hamiltonian path for the next reference structure. Lin and Kernighan mention evaluating the tour length only for this choice, but of course there is little extra effort in evaluating the tour length for all choices and record the best for future reference (even if it is not the one that generates the next reference structure). Proceeding from this level a trial tour can be obtained at each additional level of the LK process by adding an edge (v_{2i+2}, v_1) which closes up the current Hamiltonian path adding the cost $T_i = c(v_{2i+2}, v_1)$. Again, the total tour cost change for the current level is given by $\Delta_i = E_i + T_i$.

Before providing the outline of the LK procedure we recall that a standard local search process involves solving a subproblem at each iteration. The solution space for this subproblem is implicitly defined by the neighborhood structure and the particular subset of available moves that is singled out to identify "reasonable choices" (and hence to restrict the solution space for this subproblem). Thus, it is relevant to keep in mind that the various minimization functions used in neighborhood search for TSPs assume the consideration of a neighbor list that restricts the number of choices for the vertices/edges involved in the moves. The original paper of Lin and Kernighan suggests a neighbor list made up of the 5 nearest neighbors of the node. However, other authors such as Johnson and McGeoch [463] claim better results for lists of the 20 nearest neighbors, and Applegate, Bixby, Chvatal, and Cook [32] use lists of different lengths at different levels of the search. In addition, Lin and Kernighan required that no added edges be subsequently deleted in the same k-opt move and no deleted edges be subsequently added. Johnson and McGeoch apply only the first of these two restrictions, which by itself is enough to insure that the search will have no more than n levels. A general outline of the Lin-Kernighan procedure is presented in Figure 8.4.

The Lin-Kernighan procedure introduces an important framework to generate compound moves. The wider class of variable depth methods known as Ejection Chains methods discussed in the next section shares several characteristics with this procedure. First, the creation of a reference structure (which in the LK procedure is implicitly given by the path H_1) makes it possible to create moves whose depth goes significantly beyond the one that can be handled in a practical manner

Step 1. Initialization

(a) Generate a starting tour T.

(b) Set $i = 1$. Choose for v_1 some vertex that has not taken this role since the last best tour was found.

Step 2. Choose e_1 and h_1 to initiate the LK search.

(a) Set $e_1 = (v_1, v_2)$.

(b) Select $h_1 = (v_2, v_3)$ such that $E_1 < 0$. If this is not possible, stop.

Step 3. Perform LK Search

(a) Set $i = i + 1$. Choose $e_i = (v_{2i-1}, v_{2i})$ such that v_{2i} is in the cycle created when h_{i-1} was added.

(b) Compute Δ_i and keep track of the e and h edges associated with the current Hamiltonian path H_i. Record the level i^* associated with the minimum Δ value found so far.

(c) Choose $h_i = (v_{2i}, v_{2i+1})$ such that $E_i < 0$ and e_{i+1} exists. If such a h_i exists, go to Step 3.

Step 4. Backtraching for Levels 1 and 2

Perform 3-opt moves:

(a) If there is at least one h_2 not examined, set $i = 2$ and go to Step 3(c).

(b) If there is at least one e_2 not examined, choose $e_2 = (v_3, v_4)$ such that v_4 is in the path $(v_1, \ldots, v_4, v_3)$.

(c) Choose $h_2 = (v_4, v_5)$ and the associated adjacent edge $e_3 = (v_5, v_6)$ such that v_5 is in the cycle $(v_2, v_3, \ldots, v_2)$.

(d) Compute E_2 and Δ_2. If Δ_2 is smaller than the best (least) Δ value found so far, update i^* and the new e and h edges considered in this Step. If $E_2 < 0$, set $i = 3$ and go to Step 3(c).

Perform 4-opt moves:

(e) Choose $h_2 = (v_4, v_5)$ and the associated adjacent edge $e_3 = (v_5, v_6)$ such that v_5 is in the path $(v_1, \ldots, v_4)$ and e_3 is in the cycle $(v_5, \ldots, v_4, v_5)$ created when h_2 was added.

(f) Choose $h_3 = (v_6, v_7)$ and the associated $e_4 = (v_7, v_8)$ where v_7 is in the cycle $(v_2, v_3, \ldots, v_2)$ and v_8 corresponds to the largest cost edge incident at v_7.

(g) Compute E_3 and Δ_3. If Δ_3 is smaller than the best Δ value found so far, update i^* and the new e and h edges considered in this Step. If $E_3 < 0$, set $i = 4$ and go to Step 3(c).

Perform 2-opt moves:

(h) If there is at least one h_1 not examined, set $i = 1$ and go to Step 2(b).

(i) If there is at least one e_1 not examined, set $i = 1$ and go to Step 2(a).

(j) If there is at least one v_1 not examined, go to Step 1(b). Otherwise Stop.

Figure 8.4. The General Lin-Kernighan Procedure

with classic k-opt neighborhoods. Second, the iterative characteristic of building the neighborhood to successive levels provides a form of "advance look-ahead" which leads to better choices. Finally, the evaluation of trial solutions throughout the neighborhood construction provides a way for the method to adaptively select the most appropriate level (or depth) of the move.

A fundamental drawback of the k-opt neighborhoods traditionally used, including the ones constructed in the basic Lin-Kernighan approach, is that the edges added and dropped are successively adjacent. These moves can be identified by numbering the vertices in the sequence determined by the added and dropped edges, noting that the last vertex is always connected by a tour edge to the first. We call these *sequential neighborhoods*, as opposed to *non-sequential neighborhoods* where such a successive adjacency requirement is not imposed. Sequential neighborhoods can be shown to generate a restricted instance of a classical alternating path, as introduced in graph theory by Berge [104].

Sequential neighborhoods can fail to find certain types of improved tours even if they are close to the current tour. This is illustrated in Figure 8.5, which depicts the so-called *double-bridge* as an example of a move in a non-sequential neighborhood. The tour reached by this move cannot be reached by means of any move within a bounded sequential neighborhood. Lin and Kernighan first identified this class of moves and suggested looking for improving double-bridge moves as a supplement to their variable-depth moves . Subsequently, Martin, Otto, and Felten [588] proposed using random double-bridge moves as a method for restarting the algorithm once a local optimum had been found, and variants on the resulting "Chained" (or "Iterated") Lin-Kernighan algorithm have proved quite successful (Applegate et al. [27], Applegate, Cook, and Rohe [32], Helsgaun [446], Johnson and McGeoch [463]). A discussion of the performance of these approaches in practice is provided in Chapter 9.

2.3. Ejection Chain Methods

We have noted that the LK procedure relies on a Hamiltonian path as the basic reference structure to generate moves at each level of neighborhood construction. The fact that this structure has a configuration very close to a valid tour is convenient for visualization, but also constitutes a limitation of the procedure. More general Ejection Chain methods avoid this limitation by providing a wide variety of reference structures, which have the ability to generate moves not available to neighborhood search

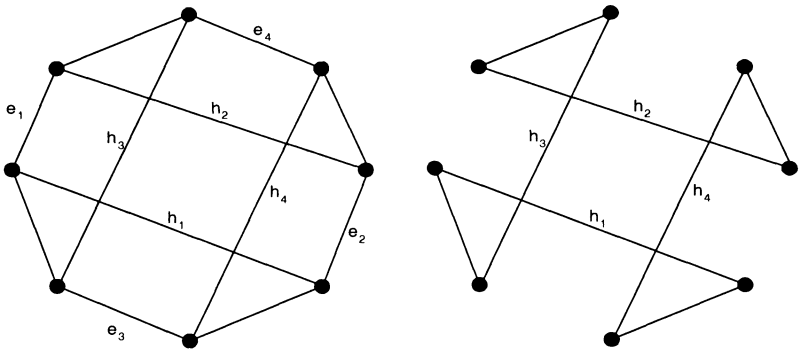

Figure 8.5. The *double-bridge* neighborhood.

approaches traditionally applied to TSPs. This section explores some of these ejection chain approaches and provides a framework for efficient implementations.

Ejection Chains are variable depth methods that generate a sequence of interrelated simple moves to create a more complex compound move. There are several types of ejection chains, some structured to induce successive changes in problem variables and others structured to induce changes in particular types of model components (such as nodes and edges of a graph). For a comprehensive description of ejection chain methods on graphs we refer the reader to Glover [371] and Glover ([372, 374]. Implementations and some extensions of these types of ejection chains for the TSP can be found in Pesch and Glover [667], Rego [704] and Glover and Punnen [382]. Applications of ejection chains to a cardinality-constrained TSP are discussed in Cao and Glover [158].

Ejection chains have also been successfully applied to combinatorial problems other than the TSP. For example, Dorndorf and Pesch [259] and Hubscher and Glover [372] use node-based ejection chains for clustering problems, while Laguna et al. [528] and Yagiura, Ibaraki and Glover [830] use ejection chains for the generalized assignment problem. Rego [706, 705], examines neighborhood structures based on node and subpath ejections to produce highly effective results for the vehicle routing problem. Finally, Cavique, Rego, and Themido [172] apply an ejection chain model to combine different types of moves for a real-world crew scheduling problem.

In this section we provide basic definitions and concepts of ejection chains, and then discuss some specialized ejection chain methods for the traveling salesman problem.

2.3.1 Ejection Chains Basics. Broadly speaking, an ejection chain consists of a succession of operations performed on a given set of elements, where the kth operation changes the state of one or more elements which are said to be *ejected* in the $k+1$th operation. This ejection thereby changes the state of other elements, which then lead to further ejections, until no more operations can be made according to pre-defined conditions. State-change steps and ejection steps typically alternate, and the options for each depend on the cumulative effect of previous steps (usually, but not necessarily, being influenced by the step immediately preceding).

In the ejection chain terminology, the order in which an element appears in the chain determines its *level*. The conditions coordinating the ejection chain process are called *legitimacy conditions*, which are guaranteed by associated *legitimacy restrictions*.

We focus on ejection chain methods for carrying out operations on graphs. The objective is to create mechanisms, namely neighborhood structures, allowing one solution subgraph to be successfully transformed into another.

In this context, relative to a specific graph G, an ejection chain of L levels consists of a succession of operations $m_1, \ldots, m_k, \ldots, m_L$ called *ejection moves*, where m_k transforms a subgraph G_k of G into another subgraph G_{k+1} by disconnecting (or ejecting) specified components (nodes, edges, subpaths) and relinking them to other components. The number of levels L is the *depth* of the ejection chain. The particular level chosen (from among the L levels generated to provide a move executed by a local search method) usually varies from one iteration to the next. The total number of levels L can likewise vary, and hence ejection chains fall within the class of so-called *variable depth methods*. In an ejection chain framework, the subgraph obtained at each level of the chain may not represent a feasible solution but may be transformed into a feasible solution by using a complementary operation called a *trial move*.

More formally, let S_i be the current solution at iteration i of the local search method, and let m_k, t_k be the ejection move and the trial move, respectively, at a level k of the chain. A neighborhood search ejection chain process consists of generating a sequence of moves $m_1, t_1, \ldots, m_k, t_k, \ldots, m_L, t_L$ on S_i such that the transition from solution S_i to S_{i+1} is given by performing a *compound move* $m_1, m_2, \ldots, m_{k*}, t_{k*}$, where k^* represents the level associated with the highest quality trial solution visited during the ejection chain construction. In the ejection chain context we use the terms *compound move* and *transition move* interchangeably, to specify the move leading from one solution to another in an iteration of the local search procedure.

The effectiveness of such a procedure depends on the criterion for selecting component moves. More specifically, neighboring solutions obtained by an ejection chain process are created by a succession of embedded neighborhoods that lead to intermediate trial solutions at each level of the chain. However, the evaluation of ejection moves can be made independently from the evaluation of the trial moves, in which case trial moves are only evaluated after performing the ejection move at the same level of the chain. In this variant of the approach, the evaluation of an ejection move m_k only depends on the cumulative effect of the previous ejection moves, $m_1, \ldots, m_{k-1}$, and is kept separate from the evaluations of trial solutions encountered along the way. The trial moves are therefore restricted to the function of finding the best trial solution that can be obtained after performing the associated ejection move.

We stress that our preceding description of ejection chain processes simply constitutes a taxonomic device for grouping methods that share certain useful features. The value of the taxonomy, however, is evidenced by the role it has played in uncovering new methods of considerable power for discrete optimization problems across a broad range of applications. Within the TSP setting, as will be seen, the ejection chain framework provides a foundation for methods that embrace a variety of compound neighborhood structures with special properties for combining moves, while entailing a relatively modest computational effort.

2.3.2 Node-based Ejection Chain Methods. Node-based ejection chain methods derive from extensions of customary single node insertion and node exchange neighborhoods found useful in several classes of graph problems including: machine scheduling, clustering, graph-coloring, vertex covering, maximum clique or independent problems, vehicle routing problems, generalized and quadratic assignment problem, and the traveling salesman problem, to cite only a few.

Since the worst case complexity of evaluating a single node-insertion and node-exchange neighborhood is $O(n^2)$, creating compound neighborhoods by combinations of these moves requires an effort that grows exponentially with the number of moves considered in the combination. More precisely, the best compound neighborhood of k moves can be generated and evaluated with $O(n^k)$ effort. This effort can be notably reduced by using appropriate candidate lists that we discuss in Section 3.1. Such lists also apply to several other types of neighborhood structures, including the ones discussed in this section.

We present here ejection chain methods to implement a *multi-node insertion* move and a *multi-node exchange* move that yield an important form of *combinatorial leverage*. Specifically, the number of moves

represented by a level k neighborhood is multiplicatively greater than the number of moves in a level $k-1$ neighborhood, but the best move from the neighborhoods at each successive level can be determined by repeating only the effort required to determine a best first level move. In our application, for example, the moves of the first, second and third levels are respectively $O(n^2)$, $O(n^3)$, and $O(n^4)$ in number, but the best member of each can be found by repeating the $O(n^2)$ effort required to determine the best move of the first level, so the total effort is still $O(n^2)$. For a worst case analysis and proofs of the complexity of these ejection chain processes see Glover [371]. Here we focus on special properties for comparative analysis of different neighborhood structures and examine some implementation issues for improving algorithm performance.

Figure 8.6 illustrates a multi-node insertion produced by an ejection chain method. In the figure, a starting TSP tour is represented by the convex hull of the nodes, e_k denotes edges which are deleted at level k of the chain (and which identify the associated ejected nodes). Edges shown "inside" the starting tour are the ones that are added by the ejection chain process. To simplify the diagrams node labels are not used, but a node v_k is implicitly identified by the two adjacent e_k edges.

The ejection chain starts by identifying a node pair v_0, v_1 that yields the best (highest evaluation) ejection move that disconnects node v_0 from its current position and inserts it into the position currently occupied by node v_1. Thus, a first level ejection move consists of adding edges (v_0, v_{1-}), (v_0, v_{1+}) and deleting edges e_0 and e_1. This creates an intermediate structure where node v_1 is temporarily disconnected from the tour. However, a trial move can be performed by creating edge (v_{0-}, v_{0+}), and inserting node v_1 between nodes p_1 and q_1, creating edges (v_1, v_{p_1}), and (v_1, v_{q_1}), and deleting edge t_1. For the subsequent levels, ejection moves consist of selecting a new candidate node to be ejected by the previously ejected node, and so forth, until no other legitimate node exists for ejection.

This move is illustrated in the second level of the ejection chain shown in the middle diagram of Figure 8.6, where node v_1 ejects node v_2 by adding edges (v_1, v_{2-}), (v_1, v_{2+}) and deleting edge e_2. The trial move used in the first level is not considered for the construction of further levels of the chain. Instead, the ejection move generates a new move (of the same type) for the next level. A trial move is then executed as previously indicated, now by linking node v_2 to nodes p_2 and q_2, and deleting edge t_2. The corresponding level 2 trial solution is given in diagram on the right in Figure 8.6.

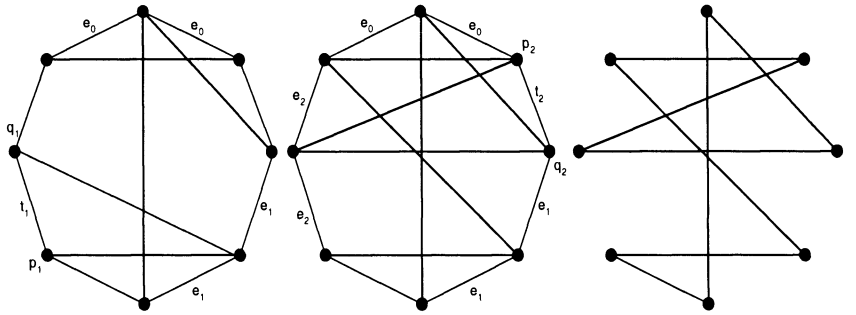

Figure 8.6. Two levels of a multi-node insertion ejection chain

A multi-node exchange move can be obtained at each level of the chain by considering a trial move that simply relocates the current ejected node to occupy the vacant position left by the node v_0 that initiates the chain. This is carried out by creating two edges (v_{0-}, v_k), (v_k, v_{0+}), where v_k denotes the node ejected at a level k of the chain.

In the multi-node insertion ejection chain method a trial move can be evaluated in time $O(n)$, but in the multi-node exchange method the move is evaluated in constant time, $O(1)$. Experiments with this ejection chain method for the vehicle routing problem (VRP) have shown that multi-node insertion is usually more efficient than multi-node exchange (Rego [706]). However, both types of moves can be efficiently combined in the same ejection chain process to select the best compound move at each level of the chain. Such an ejection chain produces a more complex neighborhood which dynamically combines insertion with exchange moves.

Figure 8.7 depicts this second type of ejection chain using the same ejection moves illustrated in Figure 8.6. Note that the first level of the chain is a standard single-node exchange move where nodes v_0 and v_1 exchange their positions. However, this exchange move produced by the ejection chain does not necessarily represent the highest evaluation two-node exchange move, unless we decide (for this first level) to evaluate the ejection move and the trial move conjointly. This decision is just a matter of preference since in this particular type of ejection chain either criterion can be evaluated in $O(n^2)$.

In the figure, level 1 and level 2 trial moves consist of adding edges (v_{0-}, v_1), (v_1, v_{0+}) and edges (v_{0-}, v_2), (v_2, v_{0+}), respectively. Note that although edge (v_{0-}, v_2) has been deleted by the second ejection move

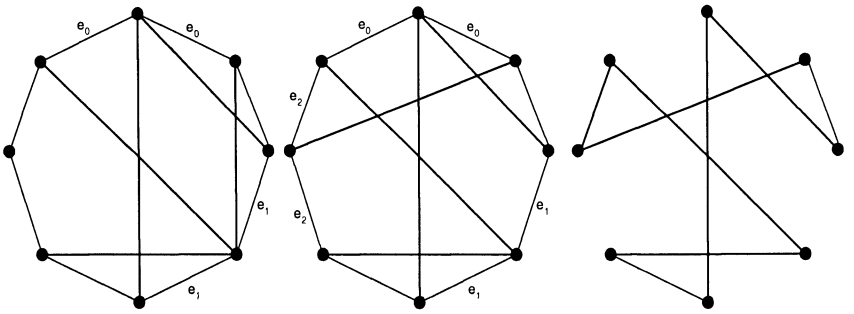

Figure 8.7. Two levels of a multi-node exchange ejection chain

it is added back by the associated trial move to create a tour from the intermediate structure.

In each of these methods, a *legitimate structure* for the ejection chain is defined by the requirement that each ejected node occurs only once in the chain. The preceding characterization of legitimacy implies that no edge will be added or dropped more than once by the transition move associated with the ejection chain, and consequently the tour cost change created by the transition move equals the sum of the added edges minus the sum of the dropped edges. These relationships are useful for the goal of efficiently generating and evaluating chains of progressively greater depth with a suitable combinatorial leverage.

Both types of node-based ejection chains can be completely determined by the succession of ejected nodes, $v_0, \ldots, v_k, \ldots, v_L$, which we designate by the set Z_L. Accordingly, we let Z_{L-} and Z_{L+} respectively denote the set of predecessors and the set of successors of vertices in Z_L, and let W_L denote the set of all the vertices involved in the ejection chain process, $W_L = Z_{L-} \cup Z_L \cup Z_{L+}$. Thus, the legitimacy restrictions consist of stipulating that each vertex in Z_L occurs only once in W_L. However, any vertex in Z_{L-} may reappear in Z_{L+} and vice versa, without violating this restriction.

An ejection chain of L levels can be recursively evaluated by computing the ejection values for these levels and summing them to give the trial value for each level. We denote a legitimate neighborhood for a node v_k in Z_k by $LN(v_k)$, thereby identifying a subset of nodes of G that do not violate the legitimacy restrictions. For convenience we denote the cost of two adjacent edges (v_i, v_j) and (v_j, v_k) as $c(v_i, v_j, v_k) = c(v_i, v_j) + c(v_j, v_k)$. Figure 8.8 provides a general neighborhood search procedure.

As shown in Glover [371] it is possible to create other variants of these ejection chain methods by growing the chain in the opposite direction.

Step 0. Initialization

(a) Initialize a legitimate neighborhood for all vertices.

(b) Denote the starting solution by S.

(c) Set $k = 0$.

Step 1. Create the first level of the ejection chain

(a) Determine a set of two initial vertices v_k,v_{k+1} by computing:

(b) $E_k = min\{c(v_{k-}, v_{k-1}, v_{k+}) - c(v_{k-1-}, v_{k-1}, v_{k-1+}) - c(v_{k-}, v_k, v_{k+}) + \lambda c(v_{k-1-}, v_{k-1+}) : v_i, v_j \in V\}$, where $\lambda = 1$ if multi-node insertion is used and $\lambda = 0$ otherwise.

(c) Set $Z_k = \{v_k\}$.

Step 2. Grow the chain to further levels

(a) Set $k = k + 1$ and set $Z_k = Z_{k-1} \cup \{v_k\}$.

(b) Evaluate the trial tour cost for the current level by computing the value: $\Delta_k = E_k + min\{c(v_p, v_k, v_{p+}) - c(v_p, v_{p+}) : v_p \in V \backslash Z_k\}$ if multi-node insertion is used. Otherwise compute $\Delta_k = E_k + c(v_{0-}, v_k, v_{0+})$.

(c) Keep track of the best level k^* that produces the best trial tour.

(d) Determine the new vertex $v \in LN(v_k)$ by computing: $E_k = E_{k-1} + min\{c(v_{k-}, v_{k-1}, v_{k+}) - c(v_{k-}, v_k, v_{k+}) : v_i, v_j \in W_{k-1}\}$.

(e) Set $v_{k+1} = v$.

(f) Update the legitimate neighborhoods for each vertex $v_i \in W_{k*}$.

(g) If $k < L$ and $LN(v_k)$ is not empty, return Step 2. Otherwise go to Step 3.

Step 3. Perform the compound move

(a) Apply to S the sequence of ejection moves up to the level k^*.

(b) Complete the update of S by executing the trial move for the level k^* associated with multi-node insertion or multi-node exchange.

Figure 8.8. Neighborhood search iteration for node-based ejection chains.

Thus, for a multi-node ejection chain, the method starts by an insertion move which disconnects one node from its current position, followed by inserting it between two others. Then, the chain grows by selecting a node to fill the vacant position, which in turn opens a new "hole" for the next level of the chain. This technique is particularly relevant for using ejection chains to provide a *construction method* with attractive properties. A constructive multi-node insertion ejection chain method starts by choosing an initial single-node insertion move and making the

corresponding edge additions to generate a partial subgraph of the tour. Then, the subgraph is extended by adding one node (external to the current subgraph) to become the new v_0 node in the chain. The process is repeated until the partial subgraph becomes a spanning subgraph of G, thus corresponding to a TSP tour in G. The use of the ejection chain as a construction method always assures a legitimate TSP structure is produced. Since each new node v_0 is external to the current subgraph, it can not correspond to any of the spanning nodes of the ejection chain.

2.3.3 Generalizing Insertion and Exchange Ejection Chain Methods. The foregoing ejection chain process can be easily extended to eject subpaths in addition to nodes. In its simplest form the procedure can be viewed as a generalization of the Or-opt neighborhood implemented in an ejection chain framework. A straightforward way to implement this generalized insertion ejection chain method is to collapse subpaths so they are essentially treated as nodes. (These collapsed subpaths are sometimes called *supernodes*.) Conversely, the method can implicitly create "holes" in subpaths and these can be possibilities for ejecting nodes inside of subpaths.

2.3.4 Subpath Ejection Chain Methods. In a subpath ejection chain (SEC) the ejection moves involve the re-arranging of paths rather than individual nodes. One example is the variable depth search of the Lin-Kernighan procedure. In this section we discuss a potentially more powerful SEC method that forms the core of one the most efficient local search algorithms for the TSP, and whose performance is discussed in Chapter 9. The method is based on the *stem-and-cycle* (S&C) reference structure, which is a spanning subgraph of G that consists of a path $ST = \{v_t, \ldots, v_r\}$called a stem, attached to a cycle $CY = (v_r, v_{s_1}, \ldots, v_{s_2}, v_r)$.

The first diagram of Figure 8.9 illustrates the creation of an S&C reference structure for the first level of the ejection chain process. The structure is created by dropping one edge in the tour (denoted by e_0) and linking one of the endpoints of the resulting Hamiltonian path to a different vertex in this path (denoted by v_r). Vertex v_r, which is the intersection of the stem and the cycle, is called the *root*. The two vertices in the cycle adjacent to v_r, denoted by v_{s_1} and v_{s_2}, are called *subroots*. The vertex v_t is called the *tip* of the stem.

The method starts by creating a stem-and-cycle reference structure from a TSP tour. Then, the method proceeds by performing an ejection move which links the tip node v_t to one of the remaining nodes v_p of

the graph, excluding the one that is adjacent to the tip. The root is considered as belonging to the cycle.

We differentiate two types of ejection moves depending on whether the operation is applied to the stem or to the cycle:

(1) *Stem-ejection move*: add an edge (v_t, v_p) where v_p belongs to the stem. Identify the edge (v_p, v_q) so that v_q is a vertex on the subpath $(v_t, \ldots, v_p)$. Vertex v_q becomes the new tip node.

(2) *Cycle-ejection move*: add an edge (v_t, v_p) where v_p belongs to the cycle. Select an edge (v_p, v_q) of the cycle to be deleted where $v_q = v_{p-}$ or $v_q = v_{p+}$. Vertex v_q becomes the new tip node.

As with other types of ejection chains, an ejection move transforms an intermediate structure into another of the same type, which usually does not represent a feasible structure for the problem. The only exception is the degenerate case where the tip v_t is also the root v_r and hence the stem is of length 0 and the cycle is a tour. This can arise for instance as the result of a cycle-ejection move where v_p is a cycle-neighbor of v_r. Even though the root is fixed during one ejection chain search it is possible to change it whenever the degenerate case occurs.

In the general case of a non-degenerate S&C structure a feasible tour can always be obtained at each level of the chain by linking the tip node to one of the subroots and deleting the edge that links the subroot to the root node.

Figure 8.9 illustrates one level of the stem-and-cycle ejection chain process where edges that lie on the convex-hull of the vertex set are members of the initial tour and edges "inside" the tour are those added by the component ejection chain moves. We denote by e_k and d_k the edges deleted at level k by ejection and trial moves, respectively, and denote by t_k the tip node at level k. The S&C reference structure is created in the left-hand diagram by adding a link between two nodes in the tour and deleting one of the edges adjacent to either one of these nodes. Hence v_r becomes the root node with subroots s_1 and s_2, and t_0 identifies the initial tip node. The middle diagram illustrates an example of a *stem-ejection move* which links t_0 to s_2 and deletes e_1, thus making t_1 the new tip node. In the example, the associated trial move consists of adding the edge (t_1, s_1) and deleting edge (s_1, v_r). Another possible trial move can be obtained by relinking t_1 to s_2 and deleting d_1.

The right-hand side diagram illustrates a *cycle-ejection move* which links t_0 to v_p (in the cycle) and deletes e_1. Again, two possible trial moves can be obtained by linking t_1 to one of the subroots and deleting the associated d_1. A trial move can also be generated just after creating the

S&C structure. However, at this initial level only one trial move leads to a different tour, which in the example consists of adding edge (t_0, s_1) and deleting edge (s_1, v_r). This restricted possibility yields the initial 2-opt trial move considered in the LK procedure. At each subsequent level, the two trial moves available by the S&C reference structure, and the enriched set of continuations for generating successive instances of this structure, provide a significantly enlarged set of tours accessible to the S&C approach by comparison to those accessible to the LK approach.

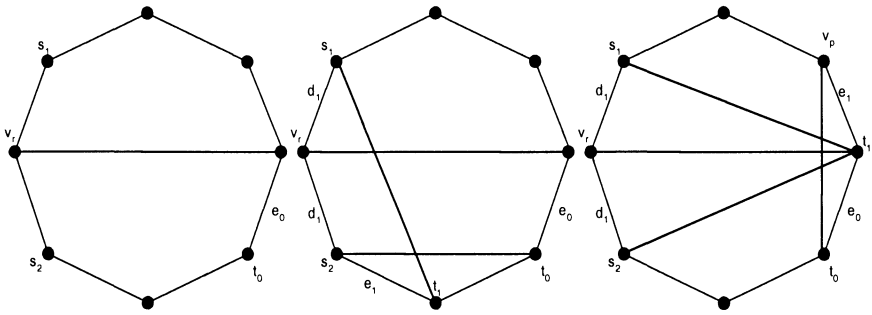

Figure 8.9. The stem-and-cycle ejection chain

In the design of the stem-and-cycle neighborhood search procedure legitimacy conditions can be added with two main purposes: (1) to prevent the method from visiting solutions already inspected during the ejection chain process; (2) to generate special forms of *alternating paths* which have proved useful in several classical graph theory problems. For the first purpose it is sufficient to stipulate that no deleted edge is added back during the construction of the chain. The second purpose deserves some additional consideration.

In classical alternating path methods in graph theory, and in neighborhood search processes related to them, the customary approach is to restrict the edges deleted to be edges of the starting solution. Methods that use this approach, which include the classical Lin-Kernighan procedure, may be viewed as *static alternating path methods.* However, certain neighboring solutions can not be obtained except by generating alternating paths in which previously added path edges are also candidates to be deleted. Thus, in contrast to classical approaches, this produces a *dynamic alternating path.* In fact, the paths provided by the S&C structure give the ability to reach any TSP tour from any other tour, in contrast to the situation illustrated earlier where the paths provided by the LK approach are unable to reach some tours that differ only

by 4 edges from the current tour. Moreover, as demonstrated in Glover [374], this ability can be assured by a simple "non-reversal" condition, which prevents an edge from being deleted if is inserted immediately after deleting another edge that was previously inserted. These restrictions define the legitimacy conditions for the S&C algorithm described in Rego [704], and are also incorporated into an enhanced version of this algorithm reported in the 8^{th} DIMACS TSP Implementation Challenge (Johnson, McGeoch, Glover, and Rego [462]).

A general design of the stem-and-cycle neighborhood search procedure can be described as in Figure 8.10, where we define a legitimate neighborhood for a node v_i, denoted by $LN(v_i)$, as the subset of nodes of G that do not violate the legitimacy restrictions identified above. Also, as shown in Rego [704] the maximum number of levels for a S&C ejection chain is bounded by $2n$, but since the best trial solution is usually found in a relatively lower level, L is considered a user-supplied parameter.

2.4. New Methods for Variable Depth Search

We have illustrated how ejection chain methods can be useful to generate compound neighborhood structures of several types, encompassing a variety of special properties for the traveling salesman problem. As previously mentioned, this framework for generating neighborhoods has proved highly effective for exploring the solution space in several other hard combinatorial problems. However, we recall that ejection chains characteristically generate moves that can not be obtained by neighborhoods that preserve feasibility at each step. We now discuss methods to efficiently combine other, more customary, neighborhoods based on the creation of appropriate candidate lists, which likewise can easily be exploited by parallel processing.

In the context of the TSP the terms *candidate lists* and *neighbor lists* are often used interchangeably because the restricted sets of elements considered for evaluation are usually defined by the relative distance between nodes in the problem data space. We will examine several types of neighbor lists in Section 3.1. However, for the exposition of the methods discussed in the present section, it is important to clearly differentiate neighbor lists from candidate lists. While both lists are associated with strategies to reduce the neighborhood size, neighbor lists constitute simply one possibility for creating candidate lists, and in many settings are used to initialize a more general candidate list approach. Neighbor lists are static, and keep neighbor elements from changing during the search process. (Methods that temporarily change the problem data

Step 0. Initialization

(a) Denote the starting solution by S.

(b) Select the initial tip node $v_{t_0} = v_r$.

(c) Set $k = 0$.

Step 1. Generate the ejection chain

(a) Ejection Move:
Compute the value of the ejection move for each vertex $v_p \in LN(v_{t_k})$ as follows: $E_k = c(v_{t_k}, v_p) - c(v_p, v_q)$ if $v_p \in ST$; $E_k = c(v_{t_k}, v_p) - min\{c(v_p, v_{p+}), c(v_p, v_{p-})\}$ if $v_p \in CY$;

(b) Select the vertex v_{p*} that yields the minimum E_k value and keep track of its adjacent vertex v_q considered for the move.

(c) Trial Move:
Compute the value of the trial moves associated with each subroot $s_i (i = 1, 2)$ and chose the one that minimizes $T_k = c(v_p, v_{si}) - c(v_{si}, v_r)$. The trial tour cost is given by $\Delta_k = E_k + T_k$.

(d) Keep track of the level k^* that produces the best trial tour so far and record the subroot node involved in the trial move.

(e) Update LN.

(f) Set $k = k + 1$ and set $v_{t_k} = v_q$.

(g) If $k < L$ and LN is not empty return to Step 1. Otherwise go to Step 2.

Step 2. Perform the compound move

(a) Apply to S each ejection move considered in the ejection chain up to the level k^*.

(b) Complete the update of S by executing the trial move for the level k^*.

Figure 8.10. An Iteration of the Stem-and-Cycle Procedure

such as *space smoothing* (Gu and Huang [409], Steven et al. [774]) and *noising methods* (Charon and Hudry [179, 180]), can change the static relationship, however.) Conversely, candidate lists do not necessarily rely on the problem data but rather are made up of solution attributes. These attributes, which can include the types of elements used in the neighbor lists and many others, change dynamically, based on information gathered from the search process. Candidate lists have been chiefly proposed in association with tabu search, which exploits strategic information embodied in memory structures. In such settings the changes in the candidate lists are represented within an *adaptive memory program-*

ming implementation either by explicitly replacing some attributes with others or by changing values of these attributes.

We next discuss candidate lists that offer a useful basis for creating compound moves within exponential neighborhoods, while using an economical degree of effort. For illustration purposes we consider standard single-node insertion moves as a basic element to create more complex moves.

2.4.1 The Sequential Fan Method. This Sequential Fan method may be viewed as a natural generalization of *beam search*, an approach that is extensively used in scheduling. (See Morton and Pentico [609], for a survey of beam search and its applications.) Beam search is applied within sequential construction methods as a restricted breadth-first tree search, which progresses level by level, without backtracking, but only exploring the most promising nodes at each level. As a schedule is constructed, beam search progressively truncates the tree by choosing a parameter (the "beam width") that determines a constant number β of nodes (partial solutions) at each depth from the root that are permitted to generate nodes at the next depth. A variant called *filtered beam search* (Ow and Morton [638]) refines this approach by using a two-step evaluation to chose the β best moves at each level. The method first picks some number δ (the "filter width") of locally best moves, and then submits these to more rigorous examination by extending each one in a single path look-ahead to the end of the tree (choosing the locally best move at each step of the path). The total path evaluations are used to select the β best moves from the initial δ candidates.

By extension, the sequential fan method operates in the local search setting as well as in the sequential construction setting, and works with complete solutions in neighborhood spaces as well as with the types of partial solutions used by beam search. It also more generally varies the width of the search strategically as a function of both the depth and the quality of the solutions generated. In a simple application to the TSP, for example, moves that interchange their current positions in the tour can be checked to identify a few good options, and for each of these, follow-on options are checked by pruning the total number of alternatives that are tracked at each level according to quality and depth.

The basic construction of the Sequential Fan tree underlies the use of a candidate list strategy based on weeding out promising moves by applying evaluations in successive levels of the tree search, where the moves that pass the evaluation criteria at one level are subjected to additional evaluation criteria at the next. The basis for the creation of the *sequential fan candidate list strategy* can be described as follows. A

list of moves $M(k)$ is associated with each level k, where list $M(k)$ is derived by applying criterion k to evaluate the moves on list $M(k-1)$. To start, list $M(1)$ is created from the set of all available moves or from a subset determined by another type of candidate list (e.g. a *neighbor list* as commonly used in the TSP setting) and contains the α_1 best of these moves by criterion 1. List $M(2)$ then contains the α_2 best of the moves from $M(1)$ according to criterion 2, and so on.

More generally, subsequent lists may not merely contain moves that are members of earlier lists, but may contain moves derived from these earlier moves. A useful way to make successive refined evaluations is to employ a deeper look-ahead in subsequent layers. For example, list $M(1)$ may apply criterion 1 to evaluate immediate moves, while $M(2)$ may apply criterion 2 to evaluate moves from the solutions produced by $M(1)$ to create *compound moves*. More advanced constructions of this look-ahead process may be conceived by the use of ejection chain processes (performed from nodes at the current level) as a foundation to determine promising component moves to dynamically update the candidate list. Also, high evaluation trial solutions found throughout the ejection chain can be recorded for further consideration, as we discuss in Section 4.3.

2.4.2 The Filter and Fan Method. The Filter and Fan method (F&F) is a combination of the filtration and sequential fan candidate list strategies used in tabu search.

By our earlier conventions, a compound move is one that can be decomposed into a sequence of more elementary component moves (or submoves), and the best compound move is the best (highest evaluation) combination of these submoves. As we have seen, a complete evaluation of simple node-insertion and node-exchange moves in dense TSPs requires $O(n^2)$ effort, and the effort of evaluating a combination of L of these moves is $O(n^L)$, and hence grows exponentially with L. However, this effort can be notably reduced based on the assumption that the best L-compound move is a combination of L submoves such that each is one of the $M(k)$ highest evaluation moves for the corresponding level k of the tree ($k = 1, \ldots, L$). Thus, instead of evaluating all possible combinations of k moves the F&F method proceeds by progressively creating new solutions for a level $k (k > 0)$, which derive from the solutions generated in the level $k-1$ by applying a restricted subset $A(k)$ of the highest evaluation moves, selected from a larger set $M(0)$ of potentially good moves, $|M(0)| = \eta_0$.

The Filter and Fan Model. The F&F model can be viewed as a *neighborhood tree* where branches represent submoves and nodes identify solutions produced by these moves. An exception is made for the root node, which represents the starting solution to which the compound move is to be applied. The maximum number of levels considered in one sequence defines the depth of the tree. The neighborhood tree is explored level by level in a breadth search strategy. For each level k, the method generates $\eta_1 * \eta_2$ moves by the *fan candidate list strategy*, then a subset $M(k)$ of η_2 moves is selected by the *filter candidate list strategy* to generate the solutions for the next level.

An illustration of the Filter and Fan model is provided in Figure 8.11, where black nodes identify a local optimum with respect to the L-neighborhood. The method starts as a standard descent method by performing 1-moves as long as they improve the best current solution. Once a local optimum is found (in the descent phase) the best $M(0)$ moves (among the M moves evaluated to establish local optimality) are used to create the first level of the F&F neighborhood tree. The next levels are created as follows. Letting η_1 be the number of $M(k)$ moves for level k, the method proceeds by selecting a subset $A_i(k)$ of η_2 moves from $M(0)$ associated with each solution $X_i(k)(i = 1, \ldots, \eta_1)$ to generate $\eta = \eta_1 * \eta_2$ trial solutions for the level $k + 1$ (as a result of applying η_2 moves to each solution at level k). For convenience we consider $\eta_1 = 4$ and $\eta_2 = 2$ for the example illustrated in Figure 8.11. (The process of selecting η_2 moves has to obey to a set a legitimacy conditions that will be specified later.)

We define $A(k) = \{A_1(k), A_2(k), \ldots, A_{\eta_1}(k)\}(|A_i(k)| = \eta_2)$ as the set of η moves evaluated at the level k from which the set $M(k) = \{m_{1k}, m_{2k}, \ldots, m_{\eta_1 k}\}$ is selected, $M(k) \subset A(k), k > 0$. The process is repeated by creating a set $X(k + 1)$ of solutions obtained by applying $M(k)$ moves to the associated solutions in $X(k)$ and keeping these solutions as starting points for the next level of the F&F tree.

For the purpose of illustration we consider the *fan candidate list strategy* to be the one that identifies the best η_2 component moves for each solution at a level k, and the *filter candidate list strategy* to be the one that identifies a subset of η_1 of the η moves generated. Also, our example constitutes a variant in which the method stops branching as soon as an improved solution is found, then switches back to the descent phase starting with this new solution. However, other strategies to create both types of candidate lists are possible.

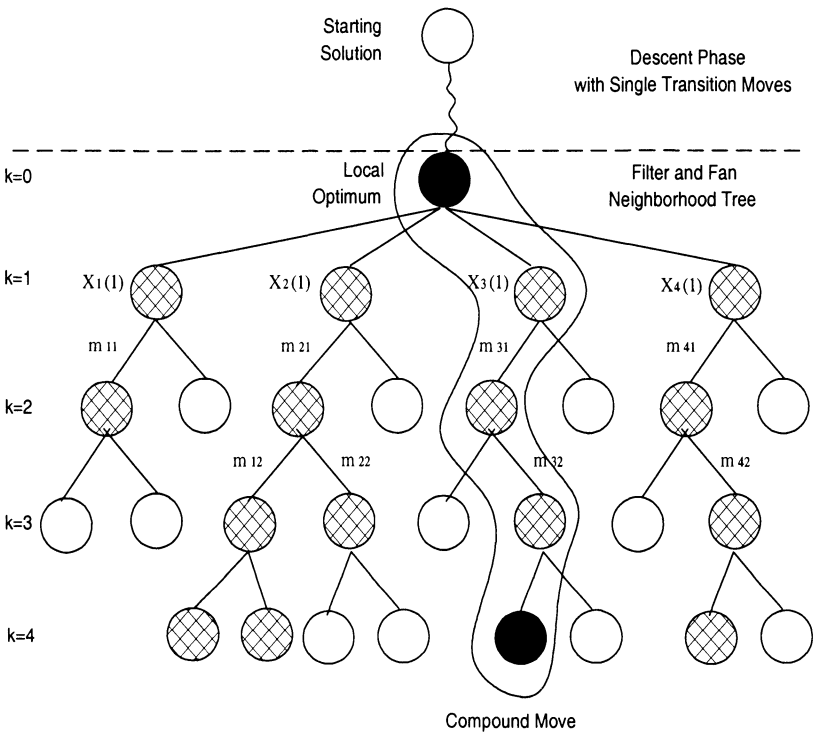

Figure 8.11. The Filter and Fan Model

More elaborate designs of the F&F method allow different types of moves for combination at each level of the tree, so that compound moves can be obtained by different neighborhoods applied under appropriate legitimacy conditions. By continuing the tree search after a local optimum is found, local optimality is overcome in "intermediate" levels of the F&F tree. Then the best trial solution encountered throughout the tree is chosen to re-initiate the descent phase.

More advanced versions result by replacing the descent phase with a tabu search phase, which, for example, can activate the F&F strategy based on the use of critical event memory. Thus, the F&F strategy can be used either to intensify the search in regions of high quality (elite) solutions or to diversify the search by propelling the method to a different region of the solution space.

Generating legitimate multi-node insertion moves by an F&F strategy. In order to create legitimate trial solutions when applying the F&F method legitimacy conditions have to be defined according to the type of component move used for the problem. We characterize legitimacy conditions for an F&F method using single-node insertion component moves for the TSP.

A component move will be called *legitimate* at a level k if this move can be performed by the usual neighborhood search rules (e.g. as a customary node insertion) after performing the associated $(k-1)$-move. Otherwise, the move is illegitimate. By this definition, a move that is illegitimate relative to a solution $X_i(k)(1 \leq i \leq \eta_1)$ will remain illegitimate throughout further levels of the subtree rooted by $X_i(k)$.

We further stipulate that the legitimacy conditions ensure the component move evaluations do not change during the F&F neighborhood search. Thus, the solution cost-changes associated with each move in M are carried forward through the tree to provide information for evaluating the $A(k)$ moves. By doing so, the neighborhood of a solution $X_i(k)$ can be restricted to consist of η_2 potentially good moves. The $M(k)$ moves $(k > 0)$ are chosen according to the quality of the trial solutions produced by the $A(k)$ moves.

Consider an F&F process based on single node-insertion moves, which insert a node v_i between two consecutive nodes v_p and v_q in the tour. To maintain the legitimacy of an L-move it is sufficient to forbid the insertion of a node v_i between nodes for which the corresponding edge (v_p, v_q) has been deleted in one of the $L-1$ levels of the corresponding L-move.

Additional considerations for implementation. An efficient implementation of the F&F procedure requires the identification of appro-

priate data structures for handling different parts of the method and speeding up the search.

The first issue in implementing the F&F method concerns the creation of $M(0)$ in the descent phase. Assume the simplest form of the F&F strategy is employed, where the initial phase is a pure memoryless descent procedure. Hence $M(0)$ is a subset of the best M moves evaluated on the last step of the descent (to verify that local optimality has been reached). It may be computationally more efficient to create $M(0)$ by performing an additional iteration of the local search procedure after reaching the local optimum S^*, rather than to keep track of the η_0 best moves at each iteration. A priority queue based on a binary heap data structure can be used to identify the best η_0 moves during the neighborhood search process in $O(log(\eta_0))$ time. (See, e.g., Cormen, Leiserson, and Rivest, [218], pages 140-152.) Since the additional iteration consists of repeating the evaluation of moves in the previous iteration, several strategies can be used to restrict the size of the neighborhood, thus reducing the time complexity to create $M(0)$.

Another issue concerns the creation of $A_i(k)$ for each solution $X_i(k)$. Instead of searching $M(0)$ for the best η_2 legitimate moves it can be advantageous to consider the m_{jk} moves ($j = 1, \ldots, \eta_{1k}, j \neq i$) as the candidates for $A_i(k)$. The creation of this candidate list assumes that good moves in one level of the tree are potentially good in deeper levels of the tree. However, such a strategy increases the chance for re-generating solutions previously visited. One way to counter this tendency is to use a tabu list of move attributes associated with each solution $X_i(k)$, thus introducing a further level of legitimacy. Additional moves to complete $A_i(k)$ can be examined in $M(0)$ whenever the number of legitimate moves for $X_i(k)$ is smaller than η_2. An outline of the general F&F procedure is provided in Figure 8.12.

3. Tabu Search

The Tabu Search (TS) metaheuristic has proved highly successful in solving a wide range of challenging problems. A key feature of TS is its use of adaptive memory, accompanied by a collection of strategies for taking advantage of this memory in a variety of contexts. Characteristically, TS can be implemented at multiple levels to exploit tradeoffs between ease of implementation and sophistication of the search. Simpler forms of TS incorporate a restricted portion of its adaptive memory design and are sometimes applied in preliminary analyses. These approaches have proved useful for testing the performance of a limited subset of TS components, and for identifying cases where more fully in-

Step 0. Generate a candidate list of component moves

- **(a)** Consider a starting solution S and perform 1-moves using a local search method until a local optimum S^* is found.
- **(b)** Create a candidate list $M(0)$ with the η_0 highest evaluation moves in the neighborhood where S^* was found.
- **(c)** Apply the best η_1 moves in $M(0)$ to S^* to create the first level of the F&F tree with solutions $X_i(1)(i = 1, \ldots, \eta_1)$. Set $k = 1$.

Step 1. Generate the Filter and Fan tree

- **(a)** Identify the best η_2 legitimate moves derived from $M(0)$ for each solution $X_i(k)(i = 1, \ldots, \eta_1)$ to create sets $A_i(k)(j = 1, \ldots, \eta_1)$.
- **(b)** Evaluate each move in $A_i(k)$, applied to the associated solution $X_i(k)$, and compute the value of the corresponding trial solution.
- **(c)** If the best trial solution found is better than S^*, perform the associated move from $X_i(k)$ on S and go to Step 0.
- **(d)** Otherwise, select the $A(k)$ moves that led to the best η_1 trial solutions to become the members of $M(k)$.
- **(e)** Apply the $M(k)$ moves to the corresponding solutions $X_i(k)$ to create $X(k + 1)$.
- **(f)** If $k = L$ stop. Otherwise set $k = k + 1$ and repeat Step 1.

Figure 8.12. A General Filter and Fan Procedure

tegrated strategies are not required. However, versions of tabu search that include a more comprehensive and advanced set of its elements generally prove superior to more limited versions of the approach.

A strategic component of TS that is sometimes omitted involves maintaining and analyzing a collection of high quality solutions to infer properties of other high quality solutions. Such processes provide a connection between Tabu Search and certain evolutionary approaches, as represented by the Scatter Search method discussed in the next section.

So far algorithmic studies of large TSP instances have chiefly focused on isolating efficient neighborhood structures (such as those based on Lin-Kernighan and Ejection Chain procedures) and on using appropriate candidate lists. As reported in the 8^{th} DIMACS TSP Implementation Challenge, recent implementations of LK and EC procedures can now find near-optimal solutions for very-large scale TSP instances in a relatively short time.

Motivated by the experiences reported in other problem settings we speculate that still better TSP solutions may be found by including advanced features of tabu search. In this section we discuss some key

strategies in TS that deserve special consideration to achieve improved outcomes.

3.1. Candidate List Strategies

As we have already emphasized, efficient procedures for isolating good candidate moves are critically important for the efficiency of local search algorithms. In the TSP setting, for example, the use of candidate lists is mandatory when large instances have to be solved.

There are some subtleties in the ways candidate list strategies may be used. A number of studies have observed that optimal or near optimal solutions often can be constructed for the TSP by limiting consideration to a small number of shortest (least cost) arcs out of each node. A natural procedure is to create a candidate list defined by the nearest neighbor graph, giving the neighbor list previously discussed, where some limited number of nodes closest to each given node determines the edges permitted to be included in the tours generated. However, TSP instances exist where the best solutions significantly violate this restriction, as exemplified by the situation where vertices on a Euclidian plane occur in separated clusters. A drawback of the nearest neighbor list is the fact that its size is fixed and it does not exploit the geometric structure of the problem. Consequently, more efficient approaches create moves by requiring some but not all edges to belong to a shortest-edge collection. Reinelt [710] suggests a candidate list approach based on the computation of a Delaunay graph, which provides a set of edges to initialize the candidate list. Then the list is expanded by adding an edge (v_i, v_k) for each pair (v_i, v_j) and (v_j, v_k) in the initial set. It has been observed that even if this approach provides useful edges for clustered TSP instances it misses several other important edges and thus restricts the performance of the local search procedure.

A more efficient candidate list construction is the so-called k-quadrant neighbor graph, initially proposed by Miller and Pekny [598] for a 2-matching problem (which is a relaxation of the TSP) and first used by Johnson and McGeogh [463] in the TSP context. In this graph, each vertex v_j is the origin of a quadrant in a Euclidian plane and the $k/4$ vertices closest to the origin in each quadrant define the neighbors for vertex v_j. Let q_{ij} denote the number of vertices in the quadrant i for vertex v_j. If $\sum_{i=1}^{4} q_{ij} < k$, then we fill out the candidate list for v_j with the $k - \sum_{i=1}^{4} q_{ij}$ nearest cities to v_j not already included. This candidate list is used in several of the most efficient implementations of local search algorithms submitted to the DIMACS TSP Challenge, including imple-

mentations of the Lin-Kernighan and Ejection Chain algorithms. A more sophisticated approach is used in Helsgaun's variant of Lin-Kernighan [446], where the candidate list for v_i consists of its k nearest neighbors v_j under a new metric produced in a two-step derivation process from the original distances (see Chapter 9).

We conjecture that the design of efficient candidate lists for these hard TSP instances (where vertices are not uniformly diffused but may clump and cluster) depend in part on their *amplification factor* [371], which is the ratio of the total number of arcs added by the move to the number that belong to the k-shortest category. For a simple example, consider single node insertion and exchange moves. Requiring that a "first added edge" in each of these moves must be on a nearest neighbor list) instead of requiring that all added edges belong to such lists) will achieve amplification factors of 3 and 4 respectively. The logical conditions defining such a candidate list in the present context can be specified more precisely as follows. For an insertion move, where a selected node v_i is repositioned to lie between nodes v_p and v_q, we require one of these two added edges (v_p, v_i) or (v_i, v_q) to be among the k shortest arcs linked to node v_i. Since three edges are added by the move (including the arc joining v_{i-} to v_{i+}), this single-arc requirement gives an amplification factor of 3. (More than one of the three added edges may belong to the k-shortest category, but only one is compelled to do so.) Given node v_i, the form of the insertion move is completely determined once either edge (v_p, v_i) or (v_i, v_q) is specified. Similarly, for exchange moves, where nodes v_i and v_j interchange positions, we require only one of the four added edges (v_{i-}, v_j), (v_j-, v_i), (v_i, v_{j+}), (v_j, v_{i+}) to belong to the k-shortest group, yielding an amplification factor of 4. Here, a given added edge can be used to define two different moves. By extension, the subpath insertions and exchanges of the type described in the ejection chain method provide a means for achieving significantly higher amplification factors.

The features attending these cases are characteristic of those exhibited by a special type of candidate list proposed in tabu search called a *preferred attribute candidate list*. In this instance, the k shortest edges of a node v_i identify the "preferred attributes" used to control the construction of moves where each attribute (or attribute set) on the list exhibits a property expected to be found in good solutions. For the present setting, these candidate lists can be constructed as follows.

Consider first the case of insert moves where each preferred arc (v_i, v_j) generates two candidate moves: the first inserting v_i between v_j and v_{j+}, and the second inserting v_j between v_i and v_{i+}, excluding the case where (v_i, v_j) is an edge of the tour. Since we are dealing with the symmetric

case, the preferred edge (v_i, v_j) generates two insert moves in addition to those indicated. The first inserts v_i between v_j and v_{j+}, and the second inserts v_j between v_{i-} and v_i. The preferred attribute candidate list for exchange moves is similarly constructed. Each preferred edge (v_i, v_j) generates four candidate exchange moves, the first exchanging v_j with v_{j+}, the second exchanging v_i with v_{i-}, and two others that result by treating a preferred edge in its two equivalent forms of (v_i, v_j) and (v_j, v_i). Note that the generalization of these constructions for multi-node insertion and exchange moves of the type considered by Or-opt neighborhoods is straightforward.

We suggest that fuller advantage can be gained from the preferred candidate list by replacing the costs $c(v_i, v_j)$ by non-negative reduced costs derived by solving 1-tree relaxation of the TSP. This will not change the move evaluations, but typically will change the identities of the k-shortest edges of each node. (Ties can be broken by reference to the original costs.) Additional shortest edges may be included as determined by "modified" reduced costs, where constraints violating the node degree are plugged in a Lagrangian function to amend the 1-tree structure.

In addition to the design of candidate list strategies, a careful organization that saves the results of evaluations from previous iterations rather than re-computing them from scratch, can also be valuable for reducing time. Time saved in this way allows a chance to devote more time to the search. In the TSP setting this objective has been chiefly achieved by the use of the so-called *don't-look bits* strategy introduced by Bentley [103]. This strategy is based on the observation that if the base vertex, e.g. v_1 in the LK procedure, and if the "tour neighbors" of this vertex have not changed since that time, it is unlikely that the selection of this vertex will produce an improving move. Thus, by associating a binary variable (or flag) with each vertex, the neighborhood is restricted to moves for which the base vertex v_1's associated bit is turned off. A bit for a vertex v_i is turned on the first time the selection of this vertex does not produce an improving move. Conversely, it is turned off when one of its adjacent vertices is used for a move.

3.2. Intensification and Diversification Strategies

Intensification and diversification in tabu search underlie the use of memory structures which operate by reference to four main principal dimensions: *recency*, *frequency*, *quality*, and *influence*. The strategic

integration of different types of memory along these dimensions is generally known as *adaptive memory programming*.

Elements of memory can refer to both *attributive memory* and *explicit memory*. Attributive (or "Attribute-based") memory refers to either basic or created attributes – instead of recording complete solutions – as a way to generate strategies to guide the search. Attributive memory records information about solution attributes that change in moving from one solution to another. For example, in the TSP setting, attributes can consist of nodes or arcs that are added, dropped or repositioned by the moves executed. (In more abstract problem formulations, attributes may correspond to values of variables or functions.) Sometimes attributes are also strategically combined to create other attributes, as by hash functions or by chunking or "vocabulary building" methods ([379]). Tabu search also uses explicit memory (complete solutions), usually by recording a limited set of elite solutions which are analyzed to determine relationships between attributes in these solutions.

Broadly speaking, recency/frequency and quality/influence can be viewed as complementary dimensions. Recency-based and frequency-based memory record timing information about the use of specific memory elements while quality and influence classify solutions in terms of their significance for representing promising solution characteristics (or regions in the solution space) and the impact of certain choices on the quality of the solutions produced. The time span considered in recency-based and frequency-based memory gives rise to an important distinction between *short-term memory* and *longer-term memory*.

The short term memory component of tabu search, which is the starting point for many tabu search implementations, characteristically embodies a recency-based memory that modifies the solution trajectory by *tabu restrictions* (or *conditions*) and *aspiration criteria*. A tabu restriction prevents a particular solution, or set of solutions, from being chosen as the outcome of the next move. Most commonly used short term memory keeps track of solution attributes that have changed during the recent past. Recency-based memory is exploited by assigning a tabu-active designation to selected attributes that contribute to creating a tabu restriction. This prevents certain solutions from the recent past from belonging to the admissible neighborhood of the current solution and hence from being revisited. The process implicitly subdivides solutions into changing "similarity classes" whereby all solutions that share tabu-active attributes with solutions recently visited may likewise be prevented from being visited. Aspiration levels provide a supplementary basis for controlling the search, by allowing a move to be selected if the resulting solution is sufficiently good or novel, in spite of being classified

tabu-active. A simple aspiration criterion is to allow a tabu move to be selected if it leads to a solution better than the best one seen so far, or the best one seen within a particular region or duration of search. Advanced forms of short-term memory may consider various types of tabu restrictions associated with several aspiration criteria, which may be used in conjunction to make a decision about the declination of the tabu status of a particular move.

In the TSP context, tabu restrictions may be created, for example, by (1) preventing a dropped edge from being subsequently added back; (2) preventing an added edge from being subsequently dropped; (3) preventing a move that simultaneously adds a previously dropped edge and drops a previously added edge. Since there are generally fewer edges that can be dropped than can be added, a tabu restriction of type (1) allows a greater degree of flexibility than a restriction of type (2) or (3). (Still greater flexibility is provided by a restriction that prevents a dropped edge from being added back only if the move simultaneously drops a previously added edge.)

Tabu restrictions remain in operation for a certain number of iterations (the tabu tenure) which can vary according to the solution attributes involved and the current search state. In some implementations where all attributes receive the same tenure, the tabu restrictions are activated by placing the attributes on a *tabu list*, and the size of this list identifies the tenure. (An attribute whose tenure expires is removed from the list at the same time that a new attribute is added.) A first level of intensification and diversification can be achieved by changing the tabu list size. Small sizes encourage the exploration of solutions near a local optimum, and larger ones push the search out of the vicinity of the local optimum. Varying the tabu list size during the search provides one way to explore such an effect, which has proved useful in a number of tabu search applications.

A common means of implementing this type of short term memory is to create an array which records the iteration that an attribute becomes tabu-active. Then, the attribute remains tabu-active as long as the current iteration does not exceed the initial iteration value by more than the tabu tenure. A special type of tabu list results by creating "coded attributes" using hash functions. Such a representation may be viewed as a *semi-explicit memory* that can be used as an alternative to attributive memory. One variant of such an approach is a special form of tabu search known as *reactive tabu search* (Battiti and Tecchiolli [91]). The goal of this technique is to differentiate more precisely among individual solutions, by making use of a fine guage attribute memory. (Only individual solutions can pass through the mesh, if the hashing

is highly effective.) Other TS approaches usually incorporate broader gauge attribute definitions, which implicitly differentiate among subsets of solutions rather than individual solutions. In reactive TS, when the search appears to revisit a particular solution (by encountering its coded attributes) too often, the method introduces a diversification step to drive the solution into a new region.

Frequency-based memory provides a type of information that complements the information provided by recency-based memory. Frequency-based memory has two forms: *transition frequency memory* and *residence frequency memory*. Transition frequency relates to the number of times an attribute enters or leaves the solutions generated (as, for example, the number of times an edge is added or dropped). Residence frequency relates to the number of iterations during which an attribute belongs the solutions generated (as, for example, the number of iterations an edge belongs to a TSP tour, considering all tours generated). Frequency based memory can also be different according to the interval (or intervals) of time chosen for the memory. Frequency based memory that is applied only to elite solutions gives different information and is used in different ways than frequency based memory that is applied to all solutions (or "average" solutions). These memories are sometimes accompanied by extended forms of recency-based memory.

Intensification is sometimes based on keeping track of the frequency that attributes (assignments of elements to positions, edges of tours, fairly narrow ranges of value assignments, etc.) occur in elite solutions, and then favoring the inclusion of the highest frequency elements so the search can concentrate on finding the best supporting uses (or values) of other elements.

As part of a longer term intensification strategy, elements of a solution may be selected judiciously to be provisionally locked into the solution on the basis of having occurred with a high frequency in the best solutions found. In that case, choosing different mutually reinforcing sets of elements to be treated in such a fashion can be quite beneficial. In the TSP setting where typically good solutions have many elements in common, edges that belong to the intersection of elite tours may be locked into the solution, in order to focus the search on manipulating other parts of the tour. This creates a *combinatorial implosion* effect (opposite to a *combinatorial explosion* effect) that shrinks the solution space to a point where best solutions over the reduced space tend to be found

more readily. Such an intensification approach, where restrictions are imposed on parts of the problem or structure is a form of *intensification by decomposition* proposed in tabu search.

The backtracking mechanism used in the Lin-Kernighan procedure may be viewed as a simple type of intensification process that attempts to find a new improving solution by jumping back to successive trial solutions examined in the first steps of the current Lin-Kernighan iteration. This is a limited form of intensification in the sense that elite solutions chosen to restart the method are restricted to those encountered at the immediately preceding level and therefore are very similar to one another. In fact, since the backtracking process is only applied when no improving solution is found during the LK move generation, backtracking may be viewed as a perturbation mechanism locally applied to the last local optimum found (and therefore limited to the exploration of one elite solution at a time). The reverse extreme of this technique is the process of restarting the method from a new initial solution generated either randomly or by varying parameters of a constructive procedure. This represents a primitive form of diversification, without reference to memory to preserve promising characteristics of the elite solutions visited so far or to compel the generation of solutions that differ in specific ways from those previously seen.

An important strategy used in the most efficient implementations of the Lin-Kernighan procedure is the so-called "don't look bits" (DLB) approach described in Section 3.1. The strategy may be viewed as an instance of applying a critical event tabu list structure, where the tabu-active status of an attribute terminates as soon as a specified event occurs. In the case of the DLB approach, the attribute under consideration is a node, which is forbidden to be involved in a move, and hence is not examined to introduce a new edge, after it fails to be considered for an improving move.

More precisely, the usual DLB implementation can be succinctly formulated as a restricted application of tabu conditions, making use of TS terminology to describe its operation, as follows. An attribute (in this case a node) is assigned a tabu-active status as soon as a search for an improving move with that node as the base node v_1 fails. The tabu-active status of the node renders moves that involve this node tabu, and the status is removed in the DLB approach as soon as an improving move is found that drops an edge adjacent to the tabu-active node, thus identifying the "critical event" in this particular context. More general TS designs identify unattractive attributes by frequency memory over specified regions of the search, and then penalize such attributes during an intensification or diversification phase. The "region" for the Don't

Look Bits approach is simply the domain of moves examined during an iteration when the addition of the edge fails to produce an improving move, and the penalty is pre-emptive, as in the more common forms of short-term TS memory.

We conjecture that the memory structure introduced by the "don't look bits" strategy, in conjunction with efficient candidate list constructions, provides a key contribution to the performance of modern implementations of the Lin-Kernighan procedure. If so, there may be advantages to using more general types of critical event memory structures, governed by correspondingly more general uses of frequency memory, as a basis for alternative implementations. In this connection, it is interesting to note that the present implementations of the Stem-and-Cycle ejection chain method do not incorporate any type of memory structures (including the "don't look bits" structure) to restrict the solution space and guide the search process. The attractive outcomes of this ejection chain approach compared to the LK implementations are therefore somewhat surprising, and invite further examination of the Stem-and-Cycle and other ejection chain structures, where consideration is given to including the types of supplemental implementation strategies that have supported the LK procedures.

3.3. Strategic Oscillation

Strategic oscillation represents a special diversification approach in tabu search that deserves its own discussion. An important challenge in the design of local search algorithms is to create strategies that effectively avoid the trap of getting stuck in local optima. It is not unusual in combinatorial optimization for high quality local optima to lie in deep (or "large") valleys of the solution space, sometimes attended by numerous smaller variations in elevation along the general valley floor. In such cases, a significant number of iterations may be required to leave these basins of attraction in order to find new local optima of higher quality. One way to overcome this difficulty is to change the neighborhood when the telltale features of such a basin of attraction are observed. The identification of critical levels of change required to escape from "insidious valleys" provides the basis to implement a strategic oscillation that alternates between different (and somewhat complementary) neighborhood structures.

A key issue often encountered in strategic oscillation is to allow the method to cross boundaries of feasibility instead of strictly remaining within the feasible region. In general combinatorial problems, a common technique for doing this consists of relaxing some of the "hard"

constraints and introducing penalties associated with those that are violated as a result. Penalty values are appropriately adjusted at each iteration in order to bring the search back into the feasible region. Approaches of this type have been adopted to the heuristic context in the tabu search algorithm of Gendreau, Hertz, and Laporte [352] for the classic vehicle routing problem (VRP), which includes embedded TSPs. In this application, the vehicle capacity and the route length constraints are temporarily relaxed and handled by a penalty function as described.

In the TSP setting where constraints consist of enforcing a particular graph structure – in this case, a Hamiltonian circuit (or cycle) – oscillation strategies must rely upon the ability of the neighborhood structures to deal with infeasible graph structures. Typical examples of such neighborhoods are provided by the reference structures used in the Lin-Kernighan and Stem-and-Cycle Ejection Chain procedures. In these approaches, as previously noted, a feasible TSP tour chosen at one level of the move generation process is obtained by performing a sequence of (infeasible) moves that transform one reference structure into another, and recovering feasibility by performing a complementary trial move.

Another way to implement a strategic oscillation is to utilize constructive/destructive neighborhoods, which follow each construction phase by destroying the feasibility of the problem graph structure, and then build up a new solution by reconnecting the solution subgraph in a different way.

The destructive process can be done either one component at a time or based on selecting a subset of graph components as in the vocabulary building strategy of tabu search. In either case, the destructive process yields a partial subgraph made up of a subset of disconnected components. The aim of the constructive process is then to efficiently re-insert the missing components into the partial graph to create a new complete tour. The GENIUS algorithm of Gendreau, Hertz, and Laporte [351] uses a simple one-step (unit depth) oscillation, as noted earlier, but more advanced forms of oscillation are clearly possible.

4. Recent Unconventional Evolutionary Methods

It is useful to base the design of the constructive/destructive process on the observation of commonalties between good TSP tours, making use of associated tabu search memory components. Additional ways to create memory structures to explore intensification and diversification arise in connection with Scatter Search and Path Relinking methods which embody a population-based approach.

4.1. Scatter Search Overview

Scatter search [368] is an evolutionary method proposed as a primal counterpart to the dual approaches called surrogate constraint methods, which were introduced as mathematical relaxation techniques for discrete optimization problems. As opposed to eliminating constraints by plugging them into the objective function as in Lagrangean relaxations, surrogate relaxations have the goal of generating new constraints that may stand in for the original constraints. The approach is based on the principle of capturing relevant information contained in individual constraints and integrating it into new surrogate constraints as a way to generate composite decision rules and new trial solutions (Glover [367]).

Scatter search combines vectors of solutions in place of the surrogate constraint approach of combining vectors of constraints, and likewise is organized to capture information not contained separately in the original vectors. Also, in common with surrogate constraint methods, SS is organized to take advantage of auxiliary heuristic solution methods to evaluate the combinations produced and generate new vectors. As any evolutionary procedure, the method maintains a population of solutions that evolves in successive generations.

A number of algorithms based on the scatter search approach have recently been proposed for various combinatorial problems (Kelly, Rangaswamy and Xu [502], Fleurent et al. [313], Cung et al. [231], Laguna and Martí [530], Campos et al. [157], Glover, Løkketangen and Woodruff [381], Atan and Secomandi [48], Laguna, Lourenço and Martí [529], Xu, Chiu and Glover [829], Cavique, Rego and Themido [173]). For tutorial descriptions of Scatter Search with examples of different applications we refer the reader to Glover, Laguna, and Martí [380], and Rego and Leão [708].

Scatter search operates on a set of reference solutions to generate new solutions by weighted linear combinations of structured subsets of solutions. The reference set is required to be made up of high-quality and diverse solutions and the goal is to produce weighted centers of selected subregions that project these centers into regions of the solution space that are to be explored by auxiliary heuristic procedures. Depending on whether convex or nonconvex combinations are used, the projected regions can be respectively internal or external to the selected subregions.

For problems where vector components are required to be integer, a rounding process is used to yield integer values for such components. Rounding can be achieved either by a generalized rounding method or iteratively, using updating to account for conditional dependencies that can modify the rounding options.

Regardless of the type of combinations employed, the projected regions are not required to be feasible and hence the auxiliary heuristic procedures are usually designed to incorporate a double function of mapping an infeasible point to a feasible trial solution and then to transform this solution into one or more trial solutions. (The auxiliary heuristic commonly includes the function of restoring feasibility, but this is not a strict requirement since scatter search can be allowed to operate in the infeasible solution space.) From the implementation standpoint the scatter search method can be structured to consist of the following subroutine.

Diversification Generation Method - Produces diverse trial solutions from one or more arbitrary seed solutions used to initiate the method.

Improvement Method - Transforms a trial solution into one or more enhanced trial solutions. (If no improvement occurs for a given trial solution, the enhanced solution is considered to be the same as the one submitted for improvement.)

Reference Set Update Method - Creates and maintains a set of reference solutions that are the "best" according to the criteria under consideration. The goal is to ensure diversity while keeping high-quality solutions as measured by the objective function.

Subset Generation Method - Generates subsets of the reference set as a basis for creating combined solutions.

Solution Combination Method - Uses structured and weighted combinations to transform each subset of solutions produced by the subset generation method into one or more combined solutions.

A general template for a scatter search algorithm can be organized in two phases outlined as follows (Figure 8.13).

4.2. Scatter Search for the TSP

An important aspect in any evolutionary approach is the way solutions are encoded as members of the population. In genetic algorithms solutions were originally encoded as bit strings, though there have been some departures to this practice in recent years. The disposition to use bit strings in GA methods derives from the fact that the first GA crossover mechanism for combining solutions were based on simple exchanges of

Step 0. Initial Phase

- **(a)** Diversification Generator
- **(b)** Improvement Method
- **(c)** Reference Set Update Method
- **(d)** Repeat this initial phase until producing a desirable level of high-quality and diverse solutions.

Step 1. Scatter Search Phase

- **(a)** Subset Generation Method
- **(b)** Solution Combination Method
- **(c)** Improvement Method
- **(d)** Reference Set Update Method
- **(e)** Repeat this scatter search phase until the reference set converges or until a specified cutoff limit on the total number of iterations is reached.

Figure 8.13. A Scatter Search Template

bits. In the classical GA bit string representations, continuous decision variables are usually encoded as substrings of the solution strings and their length depends on the precision required for these variables. Discrete decision variables are commonly encoded in these representations as a collection of zero-one variables, each corresponding to a single binary character in the solution string. For combinatorial problems defined on graphs, decision variables are typically associated with nodes or edges of the problem graph. In the TSP setting, where the number of edges is typically much larger than the number of nodes, solutions are usually encoded as a sequence of nodes representing a possible permutation for the problem. However, such a permutation-based representation used by GA approaches for the TSP entails several drawbacks subsequently noted.

By contrast, the original form of Scatter Search was not restricted to a specific type of encoding such as using bit strings, because the mechanism for combining solutions was not constrained to the limited crossover operations that governed the original GA formulations. In fact, SS readily incorporates different types of solution encodings in different parts of the method. In this section we discuss a Scatter Search approach for the TSP that utilizes such a "differential encoding" scheme. A node-based variable representation is used where information about

the relative value of the variables is not a primary issue, and an edge-based encoding is used otherwise.

To provide a general framework that discloses some critical features for applying scatter search to the TSP, a general design of the scatter search template for the TSP may be stated as follows.

Initial Phase

Diversification Generator. Scatter search starts by generating an initial set of diverse trial solutions, characteristically using a systematic procedure, which may include a stochastic component but which is highly "strategic" as opposed to relying chiefly on randomization.

Treating the TSP as a permutation problem, an illustrative approach for generating diverse combinatorial objects may be described as follows. A trial permutation P is used as a seed to generate subsequent permutations. Define the subsequence $P(h : s)$ to be the vector $P(h : s) = (s, s + h, s + 2h, \ldots, s + rh)$, where r is the largest nonnegative integer such that $s + rh \leq n$. Relative to this, define the permutation $P(h)$ for $h < n$, to be $P(h) = (P(h : h), P(h : h - 1), \ldots, P(h : 1))$. In the TSP context we consider permutations as n-vectors whose components are the vertices $v_i \in V$. Consider for illustration a TSP with $n = 14$ vertices, $h = 4$, and a seed permutation $P(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14)$ given by the sequence of vertices ordered by their indices. The recursive application of $P(4 : s)$ for $s = 4, \ldots, 1$ results in the subsequences, $P = \{4, 8, 12\}$, $P = \{3, 7, 11\}$, $P = \{2, 6, 10, 14\}$, and $P = \{1, 5, 9, 13\}$, hence $P(4) = \{4, 8, 12, 3, 7, 11, 2, 6, 10, 14, 1, 5, 9, 13\}$. By varying h it is possible to generate up to n different permutations to initialize the reference set. The generated permutations can themselves represent tours by successively linking vertices in the order they appear in the permutation and attaching the initial and ending vertices to close up the tour.

Improvement Method. The improvement method used in the initial phase may or may not be the same method used in the scatter search phase. This decision usually depends on the context and on the search strategy one may want to implement. Here we consider the context of the Euclidian TSPs, where distances between vertices are ordinary Euclidian distances in the plane. For instance, since a diversification generator such as the one we are using characteristically generates edges that cross in the initial tours, and such crossings are non-optimal for Euclidian problems, a simple form of improvement method in the initial phase can be one of eliminating possible edge crossings rather than doing extensive local optimization. The objective is to avoid premature convergence and to keep a reference set with diverse and high quality solutions at each

iteration. In this context, the use of a classical k-opt procedure ($k = 2$ or 3) under a *first-improvement startegy*, which performs a move whenever it improves the current tour, may be appropriate for the initial phase of the SS procedure while a more powerful technique such as Lin-Kernighan or Stem-and-Cycle variable depth methods would be appropriate for the scatter search phase.

Reference Set Update Method. This method is used to create and maintain a set of reference solutions. As in any evolutionary method, a set of solutions (population of individuals) containing high evaluation combinations of attributes replaces less promising solutions at each iteration (generation) of the method. In genetic algorithms, for example, the updating process relies on randomized selection rules which select individuals according to their relative fitness value. In scatter search the updating process relies on the use of memory and is organized to maintain a good balance between intensification and diversification of the solution process. In advanced forms of scatter search reference solutions are selected based on the use of memory which operates by reference to different dimensions as defined in tabu search. Depending on the context and the search strategy, different types of memory are called for. As we have seen the term *adaptive memory programming* refers to the general realm of strategies for integrating and managing various types of memory to achieve both intensification and diversification. (See Glover and Laguna [379], and Rego and Alidaee [707], for a detailed explanation of various forms and uses of memory within search processes.) For the purpose of this discussion we consider a simple rule to update the set of reference solutions, where intensification is achieved by selecting high-quality solutions in terms of the objective function value and diversification is induced by including diverse solutions from the current candidate set CS. Thus the reference set RS can be defined by two distinct subsets B and D, representing respectively the subsets of high-quality and diverse solutions, hence $RS = B \cup D$.

Denote the cardinalities of B and D by $|B| = r_1$ and $|D| = r_2$, which do not need to be identical and can vary during the search. For instance, relatively larger sizes of B (D) can be appropriate during a phase that is more strongly characterized by an intensification (diversification) emphasis. Different schemes can be chosen to implement these variations. A dynamic variation of these sizes can be implemented by a perturbation scheme, for example, and a strategic oscillation approach with critical event memory can be used as an indicator of the order of magnitude of the relative variations.

It is important to distinguish the concepts of *difference* and *distance*. In the context of the TSP, the difference between two TSP tours is defined as the number of edges by which the two solutions differ. The distance between two solutions X and Y is defined as the minimum number of steps (or iterations) necessary for the local search algorithm to move from solution X to solution Y. Thus, the distance between two TSP tours depends on the type of neighborhood used in the local search algorithm and may not be directly related to the difference between the two TSP tours.

For a visual representation consider a solution space graph $\hat{G}$ where nodes represent solutions and arcs define direct moves from one solution to another associated with a given neighborhood structure. The distance between two solutions X and Y is given by the shortest path (in terms of the number of arcs) from node X to node Y in the graph $\hat{G}$. It is easy to see that the distance between solutions is a more accurate measure of diversity than the difference between them. However, for the sake of simplicity it is common to use the difference between solutions as an indicator of their diversity, and for the same reason this measure can be used for the selection of diverse solutions to update D in the reference set.

Let CS denote the set of solutions generated and improved during the method's application. If some of these solutions produced by the diversification generator are not sufficiently distant from each other, it is possible that the improvement method may generate the same solution from several different members of CS. Therefore, it can be valuable to have a fast procedure to identify and eliminate solutions from CS that duplicate or "lie very close" to others before creating or updating the reference set. Such an approach can be facilitated by using an appropriate hash function.

A straightforward way to create a reference set RS consists of selecting the r_1 best solutions from CS to create B, and then to generate the set D of r_2 diverse solutions by successively selecting the solution that differs by the greatest amount from the current members of RS. As a diversity measure we define $d_{ij} = |(S_i \cup S_j) \setminus (S_i \cap S_j)|$ as the difference between solutions S_i and S_j, which identifies the number of edges by which the two solutions differ from each other. The d_{ij} values are computed for each pair of solutions $S_i \in RS$ and $S_j \in CS$.

Candidate solutions are included in RS according to the *Maxmin* criterion which maximizes the minimum distance of each candidate solution to all the solutions currently in the reference set. The method starts with $RS = B$ and at each step extends RS by selecting a solution $S_j \in CS$ and setting $RS = RS \cup \{S_j\}$ and $CS = CS \setminus \{S_j\}$.

More formally, the selection of a candidate solution is given by $S_j = \text{argmax } \min_{i=1,\ldots,|RS|}\{d_{ij} : j = 1,\ldots,|CS|\}$. The process is repeated until RS is filled to the desired level.

Scatter Search Phase

Subset Generation Method. This method consists of generating subsets of reference solutions to create structured combinations, where subsets of solutions are organized to cover different promising regions of the solution space. In a spatial representation, the convex-hull of each subset delimits the solution space in subregions containing all possible convex combinations of solutions in the subset. In order to achieve a suitable intensification and diversification of the solution space, three types of subsets are organized to consist of:

1) subsets containing only solutions in B,

2) subsets containing only solutions in D, and

3) subsets that mix solutions in B and D in different proportions.

Subsets defined by solutions of type 1 are conceived to intensify the search in regions of high-quality solutions while subsets of type 2 are created to diversify the search to unexplored regions. Finally, subsets of type 3 integrate both high-quality and diverse solutions with the aim of exploiting solutions across these two types of subregions.

Adaptive memory once again is useful to define combined rules for clustering elements in the various types of subsets. This has the advantage of incorporating additional information about the search space and problem context.

The use of sophisticated memory features is beyond the scope of this discussion. However, for illustrative purposes, we may consider a simple procedure that generates the following types of subsets:

1) All 2-element subsets.

2) 3-element subsets derived from two element subsets by augmenting each 2-element subset to include the best solution (as measured by the objective function value) not in this subset.

3) 4-element subsets derived from the 3-element subsets by augmenting each 3-element subset to include the best solution (as measured by the objective function value) not in this subset.

4) The subsets consisting of the best b elements (as measured by the objective function value), for $b = 5, \ldots, |B|$.

Solution Combination Method. The Solution Combination method is designed to explore subregions within the convex-hull of the reference set. We consider solutions encoded as vectors of variables x_{ij} representing edges (v_i, v_j). New solutions are generated by weighted linear combinations which are structured by the subsets defined in the preceding step. In order to restrict the number of solutions only one solution is generated in each subset by a convex linear combination defined as follows. Let E be a subset of RS, $|E| = r$, and let $H(E)$ denote the convex hull of E. We generate solutions $S \in H(E)$ represented as

$$\begin{aligned} S &= \sum_{t=1}^{r} \lambda_t S_t \\ \sum_{t=1}^{r} \lambda_t &= 1 \\ \lambda_t &\geq 0, \qquad t = 1, \ldots, r \end{aligned}$$

where the multiplier λ_t represents the weight assigned to solution S_t. We compute these multipliers by

$$\lambda_t = \frac{\dfrac{1}{C(S_t)}}{\displaystyle\sum_{t=1}^{r} \frac{1}{C(S_t)}}$$

so that the better (lower cost) solutions receive higher weight than less attractive (higher cost) solutions. Then, we calculate the score of each variable x_{ij} relative to the solutions in E by computing

$$score(x_{ij}) = \sum_{t=1}^{r} (\lambda_t x_{ij}^t)$$

where x_{ij}^t denotes that x_{ij} is an edge in the solution S_t. Finally as variables are required to be binary, the final x_{ij} value is obtained by rounding its score to give $x_{ij} = \lfloor score(x_{ij}) + .5 \rfloor$. The computation of the value for each variable in E results in creating a linear combination of the solutions in E and a new solution can be produced using edges associated with variables $x_{ij} = 1$. Nevertheless, the set of these edges. Nevertheless,

the set of these edges does not necessarily (and usually doesn't) represent a feasible graph structure for a TSP solution, since it typically produces a subgraph containing vertices whose degrees differ from two. Such subgraphs can be viewed as fragments of solutions (or partial tours). When the subgraph resulting from a linear combination contains vertices of degree greater than two, a very straightforward technique consists of successively dropping edges with the smallest scores in the *star* (incident edge set) of these vertices until their degree becomes equal to two. By doing so, the subgraph obtained will be either feasible or fall into the case where some of the vertices have degree 1. At this juncture there are several possibilities to create a feasible solution subgraph and an appropriate tradeoff has to be chosen. For example, a simple possibility is to group vertices of degree 1 and use a heuristic that simply links them two by two according to some distance measure or savings criterion. Another possibility is to solve the linear assignment problem to match each pair of nodes according to their relative distances.

4.3. Path Relinking

Scatter Search (SS) provides a natural evolutionary framework for adaptive memory programming, as we have seen, by its incorporation of strategic principles that are shared with certain components of Tabu Search. Another strategy for integrating SS and TS principles consists of replacing vector spaces with neighborhood spaces as a basis for combining solutions, which gives raise to a TS strategy called Path-Relinking (PR).

More particularly, while SS considers linear combinations of solution vectors, PR combines solutions by generating paths between them using local search neighborhoods, and selecting new solutions encountered along these paths.

This generalization of SS can be described by the same general template outlined in Figure 8.13. Figure 8.14 provides a visual interpretation of the PR process. The lines leaving S in the figure shows an alternative paths traced by the path-relinking strategy having the solutions denoted by S_1, S_2 and S_3 operate as *guiding solutions*, which collectively determine the path trajectory taken from the *initial solution* S during the local search process. In the simplest case, a single guiding solution can be used.

The process of generating paths between solutions is accomplished by selecting moves that introduce attributes contained in the solutions

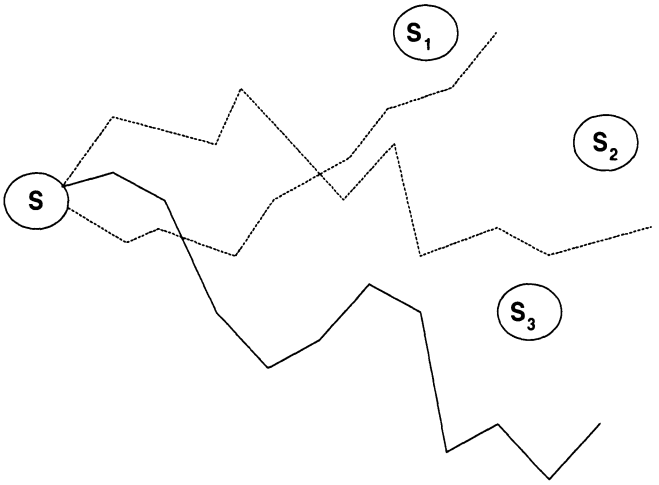

Figure 8.14. Path Relinking

that operate as guiding solutions. In the move generation process, these attributes are weighted to determine which moves are given higher priority. Again, by analogy with the SS design, each intermediate point lying in a path between solution S and a given guiding solution S' can be viewed as the result of a combination of these solutions.

By extension, a number of strategies are possible for a fuller exploration of the solution space in a path-relinking framework. Alternative paths from S under the influence of the guiding solutions can be generated by using memory structures of the type embodied in TS. Also, in a given collection of elite solutions, the roles of initiating solution and guiding solutions can be alternated. That is, a set of current solutions may be generated simultaneously, extending different paths, and allowing an initiating solution to be replaced (as a guiding solution for others) whenever its associated current solution satisfies a sufficiently strong aspiration criterion. Because their roles are interchangeable, the initiating and guiding solutions are collectively called *reference solutions*.

The possibility of exploring different trajectories in the neighborhood space suggests the use of alternative neighborhood structures with the objective of reaching solutions that might otherwise be bypassed. This strategy, denoted in TS terminology by *tunneling*, offers additional possibilities to explore boundaries between regions of feasible and infeasible solutions as a form of *strategic oscillation*.

Path-relinking provides a useful means for integrating intensification and diversification, by reference to groups (or clusters) of elite solutions that are organized according to some measure of relative difference or

distance that gives an indicator of their diversity. Solutions declared "close" to each other according to a given criterion typically are assigned to the same cluster and the objective is to maintain a set of clusters along the search that differ from each other by a significant degree. Here the concept of proximity is broad rather than restrictive in the sense that solutions may be considered close to one another if they share some particular characteristics relevant in the context of the problem under consideration. In the TSP context, for example, proximate solutions may be the ones containing edges that are common to many good solutions. In a path relinking strategy, choosing solutions S and S' from the same cluster stimulates intensification (by preserving common characteristics of these solutions), while choosing them from two different clusters stimulates diversification (by including attributes of the guiding solutions notcontained in the initial ones). This approach can go beyond the target solutions by extrapolation, creating an effect analogous to the creation of non-convex linear combinations allowed in the Euclidian space. But if an attractive departure from a guided trajectory is found along the way (using aspiration criteria), then this alternative route can also be explored, providing a dynamic integration of intensification and diversification.

Given that Ejection Chain methods, including the important special case represented by the Lin-Kernighan approach, have provided some of the most efficient algorithms for the TSP, a natural possibility is to join such methods with path relinking to provide a broader range of strategies. Such an approach, which is currently unexplored, can also take advantage of other heuristic processes previously described. For example, a combination of ejection chains and path relinking, can draw upon a sequential fan method to generate paths within the path-relinking procedure. The move components of a sequential fan candidate list can be organized in this setting to include the attributes of the designated guiding solutions. By applying ejection chain constructions to provide a look-ahead process in the sequential fan method, high evaluation trial solutions can be chosen to update the reference set (RS) of guiding solutions for a multi-parent path-relinking strategy. In such a strategy, it is important to consider appropriate measures of distance between the initial solution and the guiding solutions so that solutions in RS differ by approximately the same degree from the initial solution. By extension, if a sufficient number of ejection chain levels is generated to reach solutions that lie at distances beyond those of the current guiding solutions, then high quality solutions found in this extended neighborhood space can be used as guiding points for an extrapolated path-relinking process. Approaches of this form can be relevant not only for TSPs but also for

generalizations that include additional constraints and "embedded TSP" structures.

Finally, we observe that additional metaheuristic approaches exist that offer the potential to create useful hybrid methods for the TSP and its variants. It is beyond the scope of this chapter to provide a detailed description of such methods, but we refer the reader to Glover and Kochenberger [378] for an extensive coverage of these alternative procedures.

5. Conclusions and Research Opportunities

The current state-of-the-art discloses that the key to designing efficient algorithms for large scale traveling salesman problems is to combine powerful neighborhood structures with specialized candidate list strategies, while giving careful attention to appropriate data structures for implementation. As reported in Chapter 9, the Lin-Kernighan (LK) procedure and the Stem-and-Cycle procedure, which represent alternative instances of Ejection Chain (EC) methods, currently provide the most effective algorithms for solving large TSPs. The merit of the EC approaches derives from the use of reference structures to generate compound moves from simpler components, where the evaluation of a move at each level of construction is subdivided into independent operations to gain efficiency. The definition of the reference structure is highly important in these methods, and more advanced reference structures (such as the doubly-rooted loop constructions of [372], for example) invite examination in the future. Such structures provide an opportunity to generate moves with special properties not incorporated in k-opt moves generated by present TSP procedures.

Another potential strategic enhancement comes from the fact that the LK and the Stem-and-Cycle procedures characteristically create paths in neighborhood space by elaborating only a single thread of alternatives throughout successive levels of construction. A more aggressive way to employ such processes is to embed them in special neighborhood search trees, as described by Sequential Fan (SF) and Filter and Fan (F&F) methods. This affords the possibility to go beyond "greedy one-step choices" in the creation of neighborhood paths, while utilizing mechanisms that are highly susceptible to parallel implementation. SF and F&F approaches can also be used to merge neighborhoods of varying characteristics within different stages and threads of the search. Coupling such elements with more carefully designed memory-based strategies, such as those derived from adaptive memory programming considerations, provide additional avenues for future investigation.

Acknowledgements

We are indebted to David Johnson whose perceptive and detailed observations have improved this chapter in a number of places.

This research was supported in part by the Office of Naval Research (ONR) grant N000140110917.