Chapter 2
Classical Techniques

Kathryn A. Dowsland

2.1 Introduction

The purpose of this chapter is to provide an introduction to three classical search techniques—branch and bound, dynamic programming and network flow programming—all of which have a well established record in the solution of both classical and practical problems. All three have their origins in, or prior to, the 1950s and were the result of a surge in interest in the use of mathematical techniques for the solution of practical problems. The timing was in part due to developments in Operations Research in World War II, but was also spurred by increasing competition in the industrial sector and the promise of readily accessible computing power in the foreseeable future. A fourth technique belonging to this class, that of Integer Programming, is covered in Chap. 3. Given their age, it is not surprising that they no longer generate the same level of excitement as the more modern approaches covered elsewhere in this volume, and as a result they are frequently overlooked. This effect is reinforced as many texts such as this omit them—presumably because they have already been covered by a range of sources aimed at a wide variety of different abilities and backgrounds. In this volume we provide an introduction to these well-established classics alongside their more modern counterparts. Although they have shortcomings, many of which the more recent approaches were designed to address, they still have a role to play both as stand-alone techniques and as important ingredients in hybridized solution methods.

The chapter is meant for beginners and it is possible to understand and use the techniques covered without any prerequisite knowledge. However, some of the examples in the chapter are based on problems in graph theory. In all cases the problems and specialist terms are defined in full, but a basic knowledge of graph theory terminology such as that provided in Balakrishnan (1997) would be useful. Some of

K.A. Dowsland (✉)
Gower Optimal Algorithms Ltd, Swansea, UK
e-mail: k.a.dowsland@btconnect.com

E.K. Burke and G. Kendall (eds.), *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, DOI 10.1007/978-1-4614-6940-7_2,

the examples also belong to a class of problems known as linear programs (LP) and some of the discussion in the section on network flows makes use of the relationship between network flow problems and linear programming problems. Although knowledge of LPs is not necessary to understand the algorithms or examples as these are all couched in purely combinatorial terms we start the chapter with a brief introduction to linear programming. Further details can be found in Anderson et al. (1997).

The chapter is organized as follows. The overview of linear programming is followed by three sections introducing branch and bound, dynamic programming and network flow programming. In each case an introductory description is followed by two or more examples of their use in solving different problems, including a worked numerical example in each case. Each section ends with a brief discussion of more advanced issues. Section 2.6 looks at some problems that frequently occur as subproblems in the solution of more complex problems and suggests algorithms based on the techniques covered in Sects. 2.3–2.5 for their solution. Section 2.7 takes a brief look at potential future applications and Sect. 2.8 provides some hints and tips on how to get started with each of the techniques. Additional sources of information not covered in the references are given at the end of the chapter.

2.2 Linear Programming

2.2.1 Introduction

This section provides a brief overview of those aspects of linear programming (LP) that are relevant to the remainder of this chapter. We start by outlining the basic features of an LP model and then go on to look at an important concept of such models—that of duality. We do not go into any detail with regard to solution algorithms for two reasons. Firstly, they are not necessary in order to understand the material presented in the remainder of the chapter. Secondly, LP packages and solution code are available from a wide variety of sources so that it is no longer necessary for a potential user to develop their own solution code.

2.2.2 The Linear Programming Form

A linear programming problem is an optimization problem in which both the objective (i.e. the expression that is to be optimized) and the constraints on the solution can be expressed as a series of linear expressions in the decision variables. If the problem has n variables then the constraints define a set of hyper-planes in n-dimensional space. These are the boundaries of an n-dimensional region that defines the set of feasible solutions to the problem and is known as the feasible region. The following example illustrates the form of a simple linear programming problem.

2.2.2.1 A Simple Example

A clothing manufacturer makes three different styles of T-shirt. Style 1 requires 7.5 min cutting time, 12 min sewing time, 3 min packaging time and sells at a profit of £3. Style 2 requires 8 min cutting time, 9 min sewing time, 4 min packaging time and makes £5 profit. Style 3 requires 4 min cutting time, 8 min sewing time and 2 min packaging time and makes £4 profit. The company wants to determine production quantities of each style for the coming month. They have an order for 1,000 T-shirts of style 1 that must be met, and have a total of 10,000 man hours available for cutting, 18,000 man hours for sewing and 9,000 man hours available for packaging. Assuming that they will be able sell as many T-shirts as they produce in any of the styles how many of each should they make in order to maximize their profit?

We can formulate the problem mathematically as follows. First we define three decision variables x_1, x_2 and x_3 representing the number of T-shirts manufactured in styles 1, 2 and 3 respectively. Then the whole problem can be written as

$$\begin{aligned}
\text{Maximize} \quad & 3x_1 + 5x_2 + 4x_3 && (2.1)\\
\text{subject to} \quad & 7.5x_1 + 8x_2 + 4x_3 \leq 10{,}000 && (2.2)\\
& 12x_1 + 9x_2 + 8x_3 \leq 18{,}000 && (2.3)\\
& 3x_1 + 4x_2 + 2x_3 \leq 9{,}000 && (2.4)\\
& x_1 \geq 1{,}000 && (2.5)\\
& x_1, x_2, x_3 \geq 0. && (2.6)
\end{aligned}$$

Expression (2.1) defines the profit. This is what we need to maximize and is known as the *objective function*. The remaining expressions are the *constraints*. Constraints (2.2)–(2.4) ensure that the hours required for cutting, sewing and packaging do not exceed those available. Constraint (2.5) ensures that at least 1,000 T-shirts of style 1 are produces and constraint (2.6) stipulates that all the decision variables must be non-negative. Note that all the expressions are linear in the decision variables, and we therefore have a linear programming formulation of the problem.

2.2.2.2 The General LP Format

In general, a linear program is any problem of the form

$$\begin{aligned}
\max \text{or} \min \quad & \sum_{i=1}^{n} c_i x_i \\
\text{such that} \quad & \sum_{i=1}^{n} a_{1i} x_i \sim b_1 \qquad (2.7)\\
& \vdots \\
& \sum_{i=1}^{n} a_{mi} x_i \sim b_m
\end{aligned}$$

where $\sim$ is one of $\geq$, = or $\leq$.

The important point about a linear programming model is that the feasible region is a convex space and the objective function is a convex function. Optimization theory therefore tells us that as long as the variables can take on any real non-negative values (possibly bounded above) then the optimal solution can be found at an extreme point of the feasible region. It is also possible to derive conditions that tell us whether or not a given extreme point is optimal. Standard LP solution approaches based on these observations can solve problems involving many thousands of variables in reasonable time. As we will see later in this chapter there are special cases where these techniques work even when the variables are constrained to take on integer or binary values. The general case where the variables are constrained to be integer, known as integer programming, is more difficult and is covered in Chap. 3.

Although it makes sense when formulating linear programmes to use the flexibility of the formulation above in allowing either a maximization or minimization objective and any combination of inequalities and equalities for the constraints, much linear programming theory (and therefore the solution approaches based on the theory) assume that the problem has been converted to a standard form in which the objective is expressed in terms of a maximization problem, all the constraints apart from the non-negativity conditions are expressed as equalities, all right-hand side values $b_1 \ldots b_m$ are non-negative and all decision variables are non-negative. A general formulation can be converted into this form by the following steps.

1. If the problem is a minimization problem the signs of the objective coefficients $c_1 \ldots c_n$ are changed.
2. Any variable not constrained to be non-negative is written as the difference between two non-negative variables.
3. Any constraint with a negative right-hand side is multiplied by -1.
4. Any constraint which is an inequality is converted to an equality by the introduction of a new variable, (known as a slack variable) to the left-hand side. The variable is added in the case of a $\leq$ constraint and subtracted in the case of a $\geq$ constraint. The formulation is often written in matrix form:

$$\begin{aligned} \max \quad & CX \\ \text{s.t. } & AX = b \\ & X \geq 0 \end{aligned} \tag{2.8}$$

where $C = (c_1 \ldots c_n)$, $b = (b_1 \ldots b_m)^T$, $X = (x_1 \ldots x_n)^T$ and $A = (a_{ij})_{mXx}$.

2.2.3 Duality

An important concept in linear programming is that of duality. For a maximization problem in which all the constraints are $\leq$ constraints and all variables are non-negative, i.e. a problem of the form

$$\begin{aligned} \max \quad & CX \\ \text{s.t. } & AX \leq b \\ & X \geq 0 \end{aligned} \tag{2.9}$$

the dual is defined as

$$\begin{aligned} \min \quad & b^T Y \\ \text{s.t. } & A^T Y \geq C^T \\ & Y \geq 0. \end{aligned} \tag{2.10}$$

The original problem is known as the *primal*. Note that there is a dual variable y_i associated with each constraint in the primal problem and a dual constraint associated with each variable x_i in the primal. (The dual of a primal with a more general formulation can be derived by using rules similar to those used to convert a problem into standard form to convert the primal into the above form, with equality constraints being replaced by the equivalent two inequalities.)

The dual has the important property that the value of the objective in the optimal solution to the primal problem is the same as the optimal solution for the dual problem. Moreover, the *theorem of complimentary slackness* states that if both dual and primal have been converted to standard form, with $s_1 \ldots s_m$ the slack variables in the primal problem and $e_1 \ldots e_n$ the slack variables in the dual, then if X = $(x_1, \ldots, x_n)$ is feasible for the dual and $Y = (y_1, \ldots, y_m)$ is feasible for the primal, X and Y are optimal solutions to the primal and dual respectively if and only if $s_i y_i = 0 \; \forall i = 1, m$ and $e_j x_j = 0 \; \forall j = 1, n$. This relationship is an important feature in the specialist solution algorithm for the minimum cost network flow algorithm presented later in this chapter and underpins other LP based solution approaches.

2.2.4 Solution Techniques

Solution approaches for linear programming fall into two categories. Simplex type methods search the extreme points of the feasible region of the primal or the dual problem until the optimality conditions are satisfied. The technique dates from the seminal work of Dantzig (1951). Since then the basic technique has been refined in a number of ways to improve the overall efficiency of the search and to improve its operation on problems with specific characteristics. Variants of the simplex approach are available in a number of specialist software packages, as a feature of some spreadsheet packages, and as freeware from various web-based sources.

Although they perform well in practice simplex based procedures suffer from the disadvantage that they have poor worst case time-performance guarantees. This deficiency lead to the investigation of interior point methods, so called because they search a path of solutions through the interior of the feasible region in such a way as to arrive at an optimal point when they hit the boundary. Practical interior point

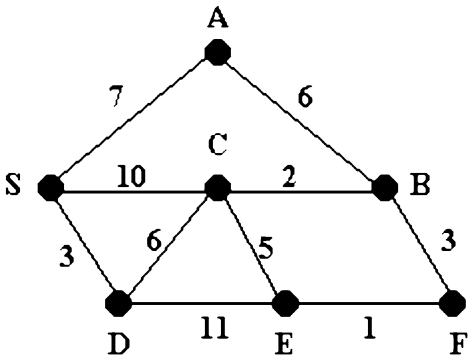

Fig. 2.1 Shortest path network

methods can be traced back to the work of Kamarkar (1984), although Khachiyan (1979) ellipsoid method was the first LP algorithm with a polynomial time guarantee. Recent interior point methods have proved efficient, in particular for large LPs, and there has also been some success with hybridizations of interior point and simplex approaches. While the choice of solution approach may be important for very large or difficult problems, for most purposes standard available code based on simplex type approaches should-suffice.

2.3 Branch and Bound

2.3.1 Introduction

When faced with the problem of finding an optimum over a finite set of alternatives an obvious approach is to enumerate all the alternatives and then select the best. However, for anything other than the smallest problems such an approach is computationally infeasible. The rationale behind the branch and bound algorithm is to reduce the number of alternatives that need to be considered by repeatedly partitioning the problem into a set of smaller subproblems and using local information in the form of bounds to eliminate those that can be shown to be sub-optimal. The simplest branch-and-bound implementations are those based on a constructive approach in which partial solutions are built up one element at a time. We therefore start by introducing the technique in this context before going on to show how it can be extended to its more general form.

Assume that we have a problem whose solutions consist of finite vectors of the form $(x_1, x_2, \ldots, x_k)$ where k may vary from solution to solution. Those combinations of values that form feasible vectors are determined by the problem constraints. The set of all possible solutions can be determined by taking each feasible value for x_1, then for each x_1 considering all compatible values for x_2, then for each partial solution $(x_1, x_2, \ldots)$ considering all compatible x_3, etc. This process can be represented

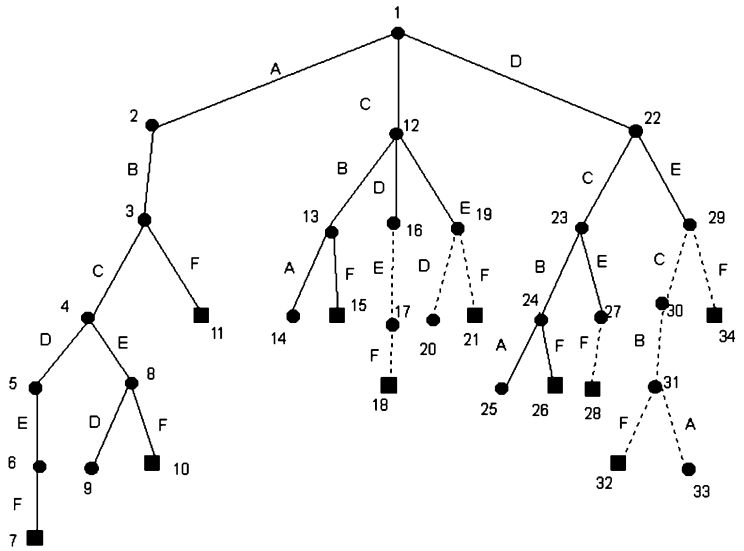

Fig. 2.2 Tree enumerating all simple routes

as a tree in which the branches at level i correspond to the choices for x_i given the choices already made for $x_1, \ldots, x_{i-1}$, and the nodes at level i correspond to the partial solutions of the first i elements. This is illustrated with reference to Figs. 2.1 and 2.2. Figure 2.2 is the tree enumerating all simple routes (i.e. routes without loops) from S to F in the network shown in Fig. 2.1. The branches at level 1 represent all the possibilities for leaving S and the branches at lower levels represent all the possibilities for extending the partial solution represented by the previous branches by one further link. The node at the top of the tree is sometimes referred to as the *root*, and the relationship between a given node and one immediately below is sometimes referred to as *parent/child* or *father/son*. All nodes that can be reached from the current node by traveling down the tree are referred to as *descendents* and nodes without any descendents are *terminal nodes*.

If we wish to search the tree in an ordered fashion we need to define a set of rules determining the order in which the branches are to be explored. This is known as the *branching strategy*. The two simplest strategies are known as *depth-first search* and *breadth-first search*. Depth-first search (also known as branch and backtrack) moves straight down a sequence of branches until a terminal node is reached before backtracking up to the nearest junction. If there are any unexplored branches below this junction it will select one of these, again continuing downwards until a terminal node is reached. If all children of the current node have been explored the search backtracks to the previous node and continues from there. In contrast, breadth-first search enumerates all the branches at one level before moving on to the next level. Depth-first search is likely to find a feasible solution early and it does not have the vast storage requirements of breadth-first search. However, breadth-first search

allows comparisons to be made across the tree, facilitating removal of dominated sub-solutions. In Fig. 2.2 the nodes are numbered in depth-first search order using a strategy of ordering the branches at each level from left to right. For a breadth-first search the search order would be 1, 2, 12, 22, 3, 13, 16, 19, 23, 29, 4, 11, etc.

As mentioned above, although it is possible to enumerate the whole tree in a small example like this, the size of the tree grows explosively with problem size, and for most real-life problems complete enumeration is not possible. For example, complete enumeration of all feasible allocations of n tasks to n machines would result in a tree with $n!$ terminal nodes, i.e. $\sim 2.4 \times 10^{18}$ terminal nodes for 20 machines. The branch-and-bound process allows us to prove that some partial solutions cannot lead to optimal solutions and to cut them and their descendants from the search tree—a process known as *pruning*. This is achieved through the use of upper and lower bounds satisfying lower_bound $\leq z \leq$ upper_bound, where z is the optimal solution obtained over all descendents of the current node. For a minimization problem the upper bound is a quantity UB such that we know we can do at least as well as UB. This is usually the best feasible solution found so far. The lower bound at node i, LB_i, is an optimistic estimate of the best solution that can be obtained by completing the partial solution at node i, i.e. we know that in exploring below node i we cannot do better than LB_i. Nodes where $LB_i \geq UB$ need not be explored further and we say that they are *fathomed*. When all nodes are fathomed the upper bound is the optimal solution. For a maximization problem the roles of the upper and lower bounds are reversed.

2.3.2 Branch and Bound Based on Partial Solutions

The success of a branch-and-bound implementation for a large problem depends on the number of branches that can be pruned successfully. This is largely dependent on the quality of the bounds, but the branching strategy can also have a significant effect on the number of nodes that need to be explored. The basic branch-and-bound approach and the influence of bound quality and branching strategy will be illustrated with reference to our shortest-path example.

2.3.2.1 Example 1: Finding the Shortest Path

We start by illustrating the use of simple bounds to prune the tree using the branching strategy defined by the node numbering. Our upper bound will be the cost of the shortest path found to date. Thus at the start we have $UB = \infty$. For the local lower bound at each node in the search tree we simply use the sum of the costs on the links traveled so far. Figure 2.3 gives details of the order in which the nodes are visited and the upper and lower bounds at each node.

The first seven branches correspond to the construction of the first path SABCDEF and the lower bound column gives the cost incurred by the partial

	Node	LB	UB		Node	LB	UB		Node	LB	UB
1	1	0	∞	16	4	15	21	31	19	15	15
2	2	7	∞	17	3	13	21	32	12	10	15
3	3	13	∞	18	11	16	16	33	1	0	15
4	4	15	∞	19	3	13	16	34	22	3	15
5	5	21	∞	20	2	7	16	35	23	9	15
6	6	32	∞	21	1	0	16	36	24	11	15
7	7	33	33	22	12	10	16	37	25	17	15
8	6	32	33	23	13	12	16	38	24	11	15
9	5	21	33	24	14	18	16	39	26	14	14
10	4	15	33	25	13	12	16	40	24	11	14
11	8	20	33	26	15	15	15	41	23	9	14
12	9	31	33	27	13	12	15	42	27	14	14
13	8	20	33	28	12	10	15	43	23	9	14
14	10	21	21	29	16	16	15	44	22	3	14
15	8	20	21	30	12	10	15	45	29	14	14

Fig. 2.3 Using simple bonds to prune the tree

solution at each stage. Note that when we reach node 7 we have our first complete path and so the upper bound is reduced to the cost of the path, i.e. 33. We now know that the optimal solution cannot be worse than 33. From node 7 we backtrack via nodes 6 and 5 to the junction at node 4 before continuing down the tree. Each time a cheaper path is completed the upper bound is reduced but the local lower bounds remain below the upper bound until we reach node 16. Here the cost of the partial solution SCD = 16 and the upper bound = 15. Thus this node is fathomed, and we backtrack immediately to node 12. Similarly, nodes 19, 27 and 29 are fathomed by the bounds and the branches below them can be pruned. When the search is complete the cost of the optimal solution = $UB = 14$ and is given by the path to node 26, SDCBF. All the branches represented by the dotted lines in Fig. 2.2 have been pruned and the search has been reduced from 33 to 23 branches.

This is already a significant reduction but we can do better by strengthening the bounds. The bound is based on the cost to date and does not make use of any estimate of the possible future cost. This can easily be incorporated in two ways. First, we know that we must leave the current vertex along a link to an unvisited vertex. Thus, we will incur an additional cost of at least as much as the cost of the cheapest such link. Similarly, we must eventually enter F. Thus, we must incur an additional cost at least as great as the cheapest link into F from the current vertex or a previously unvisited vertex. However, we cannot simply add both these quantities to the original bound as at vertices adjacent to F this will incur double counting. This highlights the need for caution when combining different bounds into a more powerful bound. We can now define our new lower bound LB_i as follows.

Let $(x_1,\ldots,x_j)$ be the current partial solution. Define L_1 = cost of path $(x_1,\ldots,x_j)$, L_2 = cheapest link from x_j to an unvisited vertex and L_3 = cheapest link into F from any vertex other than those in path $(x_1,\ldots,x_{j-1})$ Then $LB_i = L_1 + L_2 + L_3$ if x_j is not adjacent to F, and $LB_i = L_1 + \max(L_2, L_3)$ otherwise. Figure 2.4 gives details of the search using this new bound.

Note how the lower bound is now higher at each node and a good estimate of the cost of each path is obtained before the path is completed. Lower bounds of ∞ are

	Node	LB (L1,L2,L3)	UB		Node	LB (L1,L2,L3)	UB		Node	LB (L1,L2,L3)	UB
1	1	4(0,3,1)	∞	13	8	21(20,1,1)	33	26	15	15	15
2	2	14(7,6,1)	∞	14	10	21	21	27	13	∞	15
3	3	15(13,2,1)	∞	15	8	∞	21	28	12	16(10,5,1)	15
4	4	21(15,5,1)	∞	16	4	∞	21	29	1	4(0,3,1)	15
5	5	33(21,11,1)	∞	17	3	16(13,3,1)	21	30	22	10(3,6,1)	15
6	6	33(32,1,1)	∞	18	11	16	16	31	23	12(9,2,1)	15
7	7	33	33	19	3	∞	16	32	24	14(11,3,1)	15
8	6	∞	33	20	2	∞	16	33	25	∞	15
9	5	∞	33	21	1	4(0,3,1)	16	34	24	14(11,3,1)	15
10	4	21(15,5,1)	33	22	12	13(10,2,1)	16	35	26	14	14
11	8	21(20,1,1)	33	23	13	15(12,3,1)	16	36	24	17(11,6,1)	14
12	9	∞	33	24	14	∞	16	37	23	15(9,5,1)	14
				25	13	15(12,3,1)	16	38	22	15(3,11,1)	14

Fig. 2.4 Search using improved bounds

	Node	LB (L1,L2,L3)	UB		Node	LB (L1,L2,L3)	UB		Node	LB (L1,L2,L3)	UB
1	1	4(0,3,1)	∞	6	24	17(11,6,1)	14	11	1	11(0,10,1)	14
2	22	10(3,6,1)	∞	7	23	15(9,5,1)	14	12	12	13(10,2,1)	14
3	23	12(9,2,1)	∞	8	22	15(3,11,1)	14	13	13	15(12,3,1)	14
4	24	14(11,3,1)	∞	9	1	8(0,7,1)	14	14	12	17(10,6,1)	14
5	26	14	14	10	2	14(7,6,1)	14	15			

Fig. 2.5 Search using improved branching strategy

recorded whenever the branches from a particular node are exhausted. There is more work incurred in calculating this bound, not only because the actual calculation is more complex, but also because the bound at a given node may change when returning to the node in a backtracking step. This strengthens the bound and reduces the total number of branches searched to 19.

Finally, we consider the branching strategy. So far we have used simple depth-first search taking the branches at each node in the given order. In general, the efficiency of the search will be increased if the upper bound can be reduced sooner, or if the levels of the tree can be organized so as to incur high lower bounds closer to the root of the tree. Here we apply a strategy that will favor the former, and bias the search towards finding shorter routes first. This is achieved by exploring the branches at each node in increasing cost order instead of from left to right. Figure 2.5 shows the result of the search using this strategy.

Now the search starts by selecting the D branch from the root node and quickly finds the optimal solution of 14. This results in early pruning of the nodes from the other two branches and the whole search is completed in seven branches.

This example has illustrated how the size of the search tree is dependent on both the quality of the bounds and the branching strategy. However, it should be noted that this is not the most efficient way of solving the shortest-path problem and better approaches are suggested in Sect. 2.5. Nevertheless, since early papers on the technique in the early 1960s, it has proved successful in solving a wide range of both classical and practical problems. Examples include algorithms for a range of graph-theoretic problems, such as node coloring (Brown 1972; Zykov 1949),

clique and independent set problems (Bron and Kerbosch 1973), location problems (Erlenkotter 1978; Jarvinen et al. 1972) and the TSP (Little et al. 1963; Held and Karp 1970; Balas and Christofides 1981), and for several classical combinatorial optimization problems such as knapsack problems (Martello and Toth 1981), set covering and set partitioning (Garfinkel and Nemhauser 1969) and generalized assignment problems (Ross and Soland 1975). We use one of these, Brown's graph-coloring algorithm, to consolidate the ideas of the last section.

2.3.2.2 Example 2: Brown's Algorithm for Graph Coloring

An example of a problem that has been tackled using a variety of branch-and-bound approaches is that of graph coloring. The graph-coloring problem is that of minimizing the number of colors needed to color the vertices of a graph so that no two adjacent vertices are given the same color. The graph-coloring problem is an interesting example as it is the underlying model for many timetabling and scheduling problems. Brown's (1972) algorithm is an example of a branch-and-bound approach based on partial solutions. The algorithm is a straightforward application of the branch-and-bound process using simple bounds. As with the shortest-path problem, its efficiency can be improved by applying some intelligence to the branching strategy. However, the strength of the algorithm lies in its backtracking strategy that essentially prunes away branches by backtracking up several levels at a time where appropriate. The algorithm can be summarized as follows.

The branches in the tree correspond to the decision to color a given vertex in a given color. In the simplest version, the vertices are pre-ordered and the branches at level i correspond to choosing a color for the ith vertex. The colors are also ordered and the branches at each node are searched in color order. The upper bound is given by the number of colors in the best solution to date and the lower bound on each partial solution is given by the number of colors used up to that point. If the upper bound is equal to Q, then when the search backtracks to a vertex v_i for which there are no unexplored branches corresponding to colors below Q in the ordering, that node is obviously bounded and a further backtracking step must be executed. Rather than simply backtracking to the previous node the search recognizes the fact that in order to progress it is necessary for an alternative color to become free for v_i. This will only be achieved if a neighbor of v_i is uncolored. Therefore, the search backtracks up the tree until a neighbor of v_i is encountered before attempting a forward branch. If further backtracking, say at vertex v_j, is required before v_i has been successfully re-colored then re-coloring a neighbor v_j may also achieve the desired result so v_j's neighbors are added to those of v_i in defining a potential backtracking node. In order to manage this backtracking strategy in a complex search, those vertices that are neighbors of backtracking vertices are stored in a queue in reverse order and branching takes place from the first vertex in the queue. A formal definition of the algorithm is given below. The list of identifiers, J, is the set of nodes

which trigger the bounding condition and *Queue* is an ordered list of the neighbors of elements in J:

Step 0. *Define orderings*
Order the vertices $1,2,\ldots,n$ and colors $c_1,c_2,\ldots$
$\Gamma^-(i)$ denotes those neighbors of vertex i which precede i in the ordering.
Step 1. *Find initial coloring*
Color the vertices in order using the lowest indexed feasible color.
Step 2. *Store new solution*
Let q be the number of colors used. Set the upper bound equal to q and store the current q-coloring.
Set list of identifiers, $J=\emptyset$.
Step 3. *Backtrack*
3.1 *Find first vertex corresponding to local LB* $=q$
Let i^* be the first vertex colored q.
3.2 *Update list of backtracking vertices and corresponding queue of neighbors*
Remove all $j<i^*$ from J.
Set $J=J\cup\{i^*\}$.
Set *Queue* $=\bigcup_{j\in J}\Gamma^-(j)$ in reverse order.
3.3 *Backtrack to level defined by first vertex on the queue*
Let i' be the first vertex on the queue. Let q' be its color.
If $i'=k$ and vertices $1,2,\ldots,k$ are colored $c_1,c_2,\ldots,c_k$ then STOP. Stored solution is optimal.
Otherwise uncolor all $i\geq i'$.
Step 4. *Branch*
4.1 *Recolor* i'
Color i' in the first feasible color $\{q'+1,q'+2,\ldots,q-1\}$.
If no feasible color set $i^*=i'$ and goto 3.2.
4.2 *Recolor remaining vertices*
Attempt to color vertices $i=i'+1,n$ in colors 1 to $q-1$ using first feasible color.
If vertex i requires color q then set $i^*=i$ and goto 3.2.
Otherwise go to step 2.

Note that Steps 1 and 4.2 amalgamate several forward branches into one step, building on the partial coloring until the bounding condition is reached. Similarly, in Step 3.3 several backtracking steps are amalgamated. Note also that the lower bounds are not stored explicitly as the first node with a local lower bound of q will always correspond to the point where color q was used for the first time.

The algorithm is illustrated with reference to the graph in Fig. 2.6 using the given ordering of the vertices and colors in alphabetical order:

Step 1. Initial coloring $=1A,2B,3A,4C,5A,6D,7B,8D$. $q=4$.
Step 2. $J=\emptyset$.
Step 3. $i^*=6$. $J=\{6\}$. *Queue* $=\{4,2,1\}$.

Backtrack to node 4. $q' = C$. Partial coloring $= 1A, 2B, 3A$.
Step 4. Vertex 4 already colored in $q-1$. $i^* = 4$. goto 3.2.
Step 3. $J = \{6,4\}$. *Queue* $= \{3,2,1\}$.
Backtrack to node 3. $q' = A$. Partial coloring $= 1A, 2B$.
Step 4. Color 3 in color C and continue coloring $1A, 2B, 3C, 4A, 5B, 6C, 7A, 8C$.
$q = 3$.
Step 2. $J = \emptyset$.
Step 3. $i^* = 3$. $J = \{3\}$. *Queue* $= \{2\}$. Stopping condition reached and solution with $q = 3$ is an optimal solution in three colors.

As with the shortest-path implementation, the efficiency of the search can be improved by an intelligent ordering of the vertices to encourage good solutions to be found more quickly. The simplest improvement is to pre-order the vertices in decreasing degree order. However, there is no reason why the ordering of the vertices should remain the same throughout the tree, and greater efficiency gains can be obtained using some form of dynamic ordering such as selecting the vertex with the largest number of colors already used on its neighbors to color next—a strategy known as DSATUR. It is also worth considering the stopping condition in terms of the vertex ordering. The condition is valid because backtracking beyond the point where vertices 1 to k are colored in colors 1 to k will simply result in equivalent colorings with a different permutation of colors. This condition will be achieved more quickly if the ordering starts with a large clique. Thus a good strategy is to find a large clique and place these vertices in a fixed order at the start and then to use a dynamic ordering for the remaining vertices.

2.3.3 A Generalization

So far the discussion has focused on search trees based on building up partial solutions. Such approaches have proved popular for a variety of problems. However, they are just a special case of a more general strategy in which the branches correspond to adding constraints to the problem. In the case of a partial solution the constraints take the form $x_i = a_i$. The more general format underpins the branch-and-bound strategy for integer programming and will be treated in detail in Chap. 3. Therefore we will not go into detail here. Instead we will briefly illustrate the approach with an alternative tree search approach to the graph-coloring problem.

2.3.3.1 Zykov's Algorithm for Graph Coloring

Consider any two non-adjacent vertices v_i and v_j. In any solution to the graph-coloring problem there are two possibilities. Either they will be allocated the same color or they will be allocated different colors. The optimal solution subject to them being in the same color is the optimal coloring in a graph with v_i and v_j merged

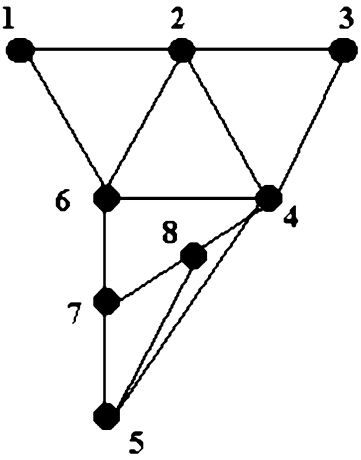

Fig. 2.6 Coloring example

into a single vertex, while the optimal solution in the latter case is the optimal coloring in the original graph with edge (v_i, v_j) added. Obviously the better of these two solutions is optimal with respect to the original problem. We also observe that if we continue a sequence of adding edges and/or merging vertices in an arbitrary graph then we will eventually be left with a complete graph (i.e. a graph in which every vertex is adjacent to every other). A complete graph with n vertices obviously requires n colors. These observations form the basis of Zykov's algorithm (1949) in which there are two branches at each level of the tree corresponding to the decision as to whether two non-adjacent vertices will be allocated the same or different colors. The two child nodes represent the coloring problems in the two suitably modified graphs and the terminal nodes will all be complete graphs. The smallest complete graph defines the optimal coloring. This is illustrated in Fig. 2.7, which shows the search tree that results from the problem of finding the optimal coloring of the sub-graph defined by vertices 1, 2, 3, 4 and 6 of the graph in Fig. 2.6.

The left-hand branch at each level constrains two non-adjacent vertices to be the same color and the child node is obtained by taking the graph at the parent node and merging the two vertices. The right-hand branch constrains the same two vertices to be different colors and the child is formed by adding an edge between the two relevant vertices in the parent graph. Branching continues until the resulting child is a complete graph. Here the terminal nodes reading from left to right are complete graphs of size 4, 3, 4, 4 and 5 respectively. The optimal solution is given by the complete graph on three vertices in which vertices 1 and 4 are allocated to one color, 3 and 6 to a second color and vertex 2 to the third.

A suitable upper bound is again the best solution found so far. A lower bound on the optimal coloring in each sub-graph can be defined by the largest clique it contains (a clique is a set of vertices such that each vertex in the set is adjacent to every other). Finding the largest clique is itself a difficult problem but a heuristic can be used to get a good estimate. In Fig. 2.7 using a depth-first search and exploring

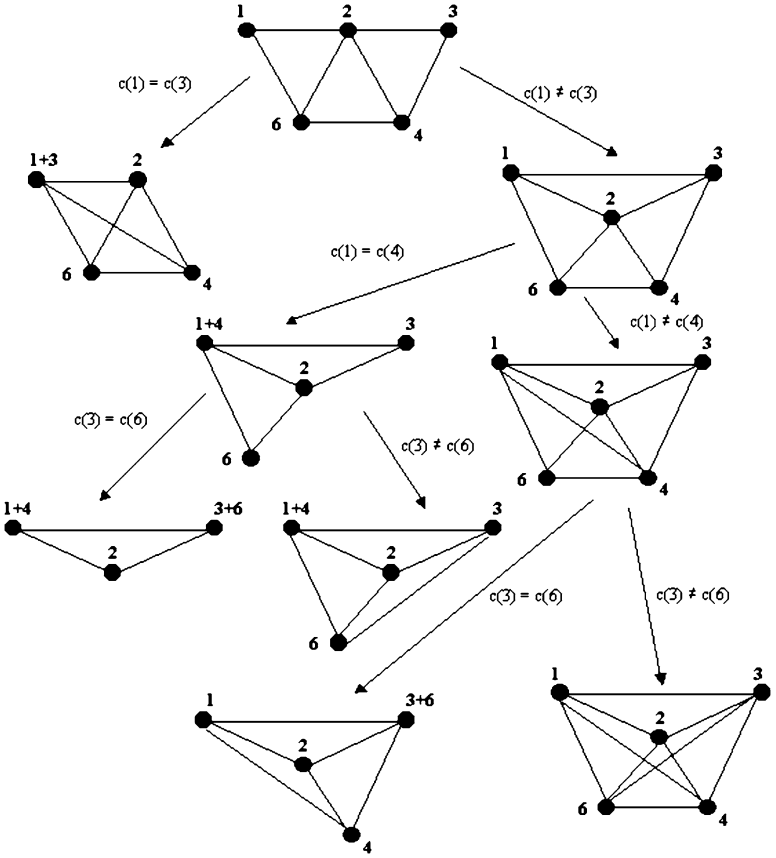

Fig. 2.7 Zykov's search tree for optimal coloring

the $c(i) = c(j)$ branch first at each node, we might recognize that the parent of the node representing the optimal solution contains two cliques of size 3. Similarly, its parent contains cliques of size 3. Thus we can backtrack straight up to the route node saving a total of four branches. Branching strategies can be designed to produce dense sub-graphs quickly thereby increasing the chances of finding large cliques early in the tree. More recent versions of the algorithm make use of theoretical results that state that certain classes of graph, known as perfect graphs, are easy to color. Branching strategies are designed to produce graphs belonging to one of the classes of perfect graph as early in the tree as possible. These can then be colored optimally, thus saving all the branches required in order to reduce them to complete graphs. See Golumbic (1980) for a wide-ranging treatment of perfect graphs and associated algorithms.

2.3.4 Other Issues

2.3.4.1 Bounds

The most important feature of a branch-and-bound algorithm is probably the quality of the bounds and it is usually worth putting considerable effort into ensuring that these are as tight as possible. In the case of lower bounds this is often achieved by exploiting as much information about the problem as possible. For example, Dowsland (1987) used a clique model to solve a class of packing problems known as the pallet loading problem. She combined the bounds in a published maximum clique algorithm with bounds derived from geometric aspects of the physical problem and showed that the percentage of problems solved within a given time frame rose from 75 to 95 %. In some cases a problem may become easy to solve if some of the constraints are removed. This is a process known as relaxation and the solution to the relaxed problem will always provide a valid bound on the solution to the original problem. This approach was used by Christofides and Whitlock (1977) in their solution to a guillotine cutting problem in which a large stock-sheet of material must be cut into a set of smaller rectangular pieces, using a sequence of guillotine cuts, so as to maximize the value of the cut pieces. In their version of the problem the demand for pieces of each dimension was constrained, whereas the unconstrained version of the problem is relatively easy to solve. Their solution uses a tree search in which each branch represents a cut, and the nodes define the set of rectangles in a partial solution. Bounds are obtained by solving the unconstrained problems for each of the sub-rectangles in the partial solution. Although such relaxation bounds can be quite effective, there is often a gap between them and the solution to the constrained problem. This gap can sometimes be reduced by incorporating a suitable penalty for the violated constraints into the objective function. This is the basis of Lagrangian relaxation, which has proved a popular bound for a variety of branch-and-bound algorithms. For example, Beasley (1985) uses the approach for a non-guillotine version of the cutting stock problem. In Lagrangian relaxation an iterative approach is used to set parameters that will increase the tightness of the bound. Such an approach is obviously time-consuming but for moderately-sized problems in a wide variety of application areas the computational effort is well worth the number of branches it saves. Details of Lagrangian relaxation can be found in Fisher (1985).

The search efficiency is also affected by the upper bound. As we have seen, pruning is more effective when good solutions are found early. Earlier pruning may result if a heuristic is used before the start of the search to find a good solution that can be used as an upper bound at the outset. Similarly, using a heuristic to complete a partial solution and provide a local upper bound should also help in fathoming nodes without branching all the way down to terminal nodes.

It is also worth noting that bounding conditions based on information other than numeric upper and lower bounds may be useful in avoiding infeasible solutions or solutions that are simply permutations or sub-sets of solutions already found. An example of this approach is the maximal clique algorithm of Bron and Kerbosch

(1973) which includes bounding conditions based on relationships between the branches already explored and those yet to be visited from a given node.

2.3.4.2 Branching

There are also issues concerning branching strategies that have not been discussed. We have assumed that the search is always carried out in a depth-first manner. An alternative that can be successful if the bounds are able to give a good estimate of the quality of solutions below each branch is to use a best-first strategy, in which nodes at different levels across the breadth of tree may be selected to be evaluated next according to an appropriate definition of *best*. In our examples, the ordering of branches was geared towards finding good solutions as early as possible. An alternative strategy is to select branches that will encourage bounding conditions to be satisfied sooner rather than later. This approach is taken by Bron and Kerbosch in their branch-and-bound algorithm for finding cliques in a graph. At each node the next branch is chosen so as to encourage the bounding condition to occur as early as possible. Comparisons between this version and a version in which the branches are selected in the natural order show that the advantage, in terms of computation time, of using the more complex strategy increases rapidly with problem size.

2.3.4.3 Miscellaneous

Although good bounds and branching strategies have resulted in many successful branch-and-bound algorithms, it should be noted that the size of the search tree will tend to grow exponentially with problem size. It is therefore important to make sure that the underlying tree is as small as possible. For example, thought should be given to avoiding branches that lead to solutions that are symmetric to each other if at all possible. It may also be possible to apply some form of problem reduction in a pre-processing phase. For example, Garfinkel and Nemhauser (1969) outline a set of reductions for both set-covering and set-partitioning problems. It should also be noted that such a strategy may provide further reductions when applied to the subproblems produced within the search itself.

Finally, it is worth noting that in many implementations the optimal solution is found quickly and most of the search time is spent in proving that this is in fact the optimal solution. Thus if there is insufficient time to complete the search, the best solution to date can be taken as a heuristic solution to the problem. An alternative that is useful if a heuristic solution with a performance guarantee is required is to replace the upper bound with $UB(1-\alpha)$. The tighter bound should enable the search to be completed more quickly and will guarantee a solution within $\alpha \times 100\,\%$ of the optimum.

2.4 Dynamic Programming

2.4.1 Introduction

Like branch and bound, dynamic programming (DP) is a procedure that solves optimization problems by breaking them down into simpler problems. It solves the problem in stages, dealing with all options at a particular stage before moving on to the next. In this sense it can often be represented as a breadth-first search. However, unlike the levels of the tree in branch and bound which partition the problem by adding constraints, the stages in DP are linked by a recursive relationship. The name dynamic programming derives from its popularity in solving problems that require decisions to be made over a sequence of time periods. Even when this is not the case, the name dynamic programming is still widely used, but the term *multi-stage programming* is sometimes used as an alternative.

The basis of DP is Bellman's *principle of optimality* (Bellman 1957) states that "the sub-policy of an optimal policy is itself optimal with regard to start and end states". As an illustration, consider the shortest-route problem. If we are told that in Fig. 2.1 the optimal route from S to F goes via E then we can be sure that part of the route from S to E is the optimal route between S and E, and that part from E to F is the optimal route from E to F. In other words, each sub-path of the optimal path is itself the shortest path between its start and end points. Any DP implementation has four main ingredients. These are stages, states, decisions and policies. At each stage, for each feasible state we make a decision as to how to achieve the next stage. The decisions are then combined into sub-policies that are themselves combined into an overall optimal policy. DP is a very general technique that has been applied at varying levels of complexity. These have been classified into four levels: deterministic, stochastic, adaptive and residual. Our treatment here will be limited to deterministic problems.

The design of a DP algorithm for a particular problem involves three tasks; the definition of the stages and states, the derivation of a simple formula for the cost/value of the starting stage/state(s) and the derivation of a recursive relationship for all states at stage k in terms of previous stages and states.

The definition of the stages and states will obviously depend on the problem being tackled, but there are some definitions that are common. Stages are frequently defined in terms of time periods from the start or end of the planning horizon, or in terms of an expanding subset of variables that may be included at each stage. Common definitions of states are the amount of product in stock or yet to be produced, the size or capacity of an entity such as stock sheet, container, factory or budget, or the destination already reached in a routing problem.

2.4.2 Developing a DP Model

2.4.2.1 Forward Recursion and the Unbounded Knapsack Problem

We will first illustrate the concepts of DP by reference to a classical optimization problem—the unbounded knapsack problem. The problem can be stated as follows. Given a container of capacity b and a set of n items of size w_i and value v_i for $i = 1, n$ such that the number of each item available is unbounded, maximize the value of items that can be packed into the container without exceeding the capacity.

The problem can be formulated as follows:

$$\max \sum_{i=1}^{n} v_i x_i \tag{2.11}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq b \tag{2.12}$$

for $x_i \geq 0$ and integer, where x_i equals the number of copies of item i in the solution.

We can formulate this problem as DP as follows. Define $F_k(S)$ to be the maximum value for a container of capacity S using items of sizes *1* to k. Here the items available represent the stages and the capacity available the states. $F_1(S)$ is the value that can be obtained if the only pieces available are those of type 1.

This is given by

$$F_1(S) = \int \left(\frac{S}{w_1} \right) v_1. \tag{2.13}$$

All that remains is to define a recursive relationship for $F_k(S)$ in terms of previous stages and states. This is achieved as follows. The optimal solution either makes use of at least one item of type k, or it does not contain any items of type k. In the latter case $F_k(S) = F_{k-1}(S)$. In the former case one copy of item k takes up w_k units of capacity and adds v_k units of value. Bellman's principle tells us that the remaining $S - w_k$ units of capacity must be packed optimally. This packing may contain further items of type k and is given by $F_k(S - w_k)$. Thus we have for $k > 1$:

$$\begin{aligned} F_k(S) &= \max\{F_{k-1}(S), F_k(S - w_k) + v_k\} \quad \text{for} \quad S \geq w_k \\ F_k(S) &= F_{k-1}(S) \quad \text{otherwise.} \end{aligned} \tag{2.14}$$

The solution to the overall problem is given by $F_n(b)$. We will illustrate the procedure with the following example.

Let $n = 3$, $b = 19$, w_1, w_2, $w_3 = 3, 5$ and 7 respectively and v_1, v_2, $v_3 = 4, 7$ and 10 respectively.

The values of $F_k(S)$ for $k = 1, 3$ and $S = 0, 19$ are given in Table 2.1. The values in column $k = 1$ are first calculated using Eq. (2.3). Then the subsequent columns are calculated in order starting from the top and working down using Eq. (2.4). For $S \geq w_k$ the appropriate value is obtained by comparing the value in row S in the previous column with the sum of v_k and the value in the current column w_k rows up.

Table 2.1 Unbounded knapsack calculations

S/k	1	2	3	S/k	1	2	3
0	0	0	0	10	12	14	14
1	0	0	0	11	12	15	15
2	0	0	0	12	16	16	17
3	4	4	4	13	16	18	18
4	4	4	4	14	16	19	20
5	4	7	7	15	20	21	21
6	8	8	8	16	20	22	22
7	8	8	10	17	20	23	24
8	8	11	11	18	24	25	25
9	12	12	12	19	24	26	27

The value of 27 in row 19, column 3 tells us that the optimal value is 27. In order to determine how this solution is achieved we need to work backwards. We need to find out whether this value came from $F_2(19)$ or $F_3(19-7)+10$. The latter option is the correct one. We therefore record one item of type 3 and check the source of the value 17 in $F_3(12)$. This is either $F_2(12)$ or $F_3(5)+10$. Once again the latter option is correct. We record a second item of type 3 and move to $F_3(5)$. $F_3(5) = F_2(5)$ so we check the source of the value of $F_2(5) = F_2(0)+7$. Thus we record an item of type 2. As we have reached the top row of the table corresponding to capacity $= 0$ the solution is completed and is given by $x_1 = 0, x_2 = 1, x_3 = 2$.

2.4.2.2 Backward Recursion and a Production Planning Problem

In the above example the recursion worked in a forward direction with the stages corresponding to an increasing subset of variables. Our second example is taken from the field of production planning and is typical of many multi-period problems in that it is solved using backwards recursion, i.e. by working backwards from the end of the planning period. The problem can be summarized as follows.

Production of a single product is to be planned over a fixed horizon of n time periods. At the start there are S_0 units in stock and no stock is required at the end. Each time period t_i has a known demand d_i which *must* be met. Up to Q units can be produced in any one time period. The cost of making q units is given by $c(q)$ and economies of scale mean that $c(q)$ is not linear in q. Surplus units can be stored from one time period to the next at a warehousing cost of w per unit. There are natural definitions of the stages and states for this problem in terms of the time periods and stock levels. However, there is no simple formula for deciding what should be done in time period 1. Instead we re-order the time periods in reverse order and relate stage k to period $(n-k)$. If we start the last period with S units in stock, then we must meet the demand d_n exactly as we are to finish without any surplus. Thus we must produce $d_n - S$ units. The formula for the optimal policy at the starting stage is therefore

Table 2.2 Production costs and demands

Production costs

Units	0	1	2	3	4	5
Cost (£1000s)	0	7	13	16	20	24

Demands

Period	1	2	3	4		
Demand	3	6	1	2		

$$F_0(S) = c(d_n - S). \tag{2.15}$$

We now need to define a recursive formula for $F_k(S)$ in terms of previous stages. If we start period $n-k$ with S units in stock and make q units we end with $S+q-d_{n-k}$ in stock. This will incur a warehousing cost and will define the starting stock for the next time period. The optimal policy from this point on has already been calculated as $F_{k-1}(S+q-d_{n-k})$. Thus the recursive formula is given by

$$F_k(S) = \min_{d_{n-k}-S \le q \le Q} \{c(q) + w(S+q-d_{n-k}) + F_{k-1}(S+q-d_{n-p})\}. \tag{2.16}$$

The lower limit on q ensures that production is sufficient to meet demand.

We also need to define the set of states that need to examined at each stage k. This can be limited in three ways. First, there is no point in having more stock than can be sold in the remaining time periods. Second, it is not possible to have more stock than could be produced up to that time period less that already sold. Third, it is not feasible to have a stock level that will not allow demand in future periods to be met. Thus for period $n-k$ we have $MIN_k \le S \le \min\{MAX1_k, MAX2_k\}$, where

$$MAX1_k = \sum_{i=n-k}^{n} d_i, \quad MAX2_k = S_0 + \sum_{i=1}^{n-k-1} (Q - d_i)$$

and

$$MIN_k = \max\left\{0, \max_{n-k \le j \le n}\left\{\sum_{i=n-k}^{j} (d_i - Q)\right\}\right\}.$$

Once again we illustrate the formulation with a concrete example. Let $n=4$, $Q=5$, $S_0=1$, $w=2{,}000$ and production costs and demands as given in Table 2.2. Working in units of £1,000 the calculations for the stages are then

Stage 0:
$MAX1_0 = 2$, $MAX2_0 = 6$, $MIN_0 = 0$
$F_0(0) = c(2) = 13$, $F_0(1) = c(1) = 7$, $F_0(2) = c(0) = 0$.
Stage 1:
$MAX1_1 = 3$, $MAX2_1 = 2$, $MIN_1 = 0$
$F_1(0) = \min\{c(1) + 0 \cdot w \quad + F_0(0), c(2) + 1 \cdot w + F_0(1), c(3) + 2 \cdot w + F_0(2)\}$
$= \min\{\underline{7+0+13}, 13+2+7, \underline{16+4+0}\} = 20$

$F_1(1) = \min\{\underline{0+0+13}, 7+2+7, 13+4+0\} = 13$
$F_1(2) = \min\{\underline{0+2+7}, 7+4+0\} = 9$
Stage 2:
$MAX1_2 = 9, MAX2_1 = 3, MIN_1 = 1$
$F_2(1) = \min\{\underline{24+0+20}\} = 44$
$F_2(2) = \min\{20+0+20, \underline{24+2+13}\} = 39$
$F_2(3) = \min\{16+0+20, \underline{20+2+13}, 24+4+9\} = 35$
Stage 3: We do not need to calculate limits on S as we know that starting stock equals 1:

$$F_3(1) = \min\{\underline{16+2+44}, 20+4+39, 24+6+35\} = 62.$$

Note that in many cases the full range of values for q from $S - d_{n-k}$ to Q have not been included in the minimization as they would lead to overstocking or understocking. For example, in calculating $F_1(0)$ we do not consider any value of q above 3 as this would give more than two units at the start of the last time period. Similarly, we do consider q less than 3 in $F_3(1)$ as we need at least one unit in stock at the start of time period 2.

As with the knapsack problem, the calculations give the cost of the optimal solution but we need to work backwards in order to derive the optimal solution. Starting with $F_3(1)$ we note that the minimum value resulted from manufacturing three units, which leaves one unit in stock once the demand for three units has been met. Thus the policy from time period 2 onwards is given by $F_2(1)$. This is optimized by producing five units, leaving zero in stock. We therefore move to $F_1(0)$ which is optimized in two ways—producing 1 and leaving 0 in stock or producing 3 and leaving 2 in stock. This implies that there are two optimal solutions. The former is completed using $F_0(0)$ and the latter using $F_0(2)$. The two solutions are summarized in Table 2.3.

2.4.3 Other Issues

One of the main criticisms of a DP approach is that the number of subproblems that need to be solved is dependent not only on the stages but also on the states. While the number of stages is usually related to the size of the problem in the traditional sense (i.e. as a function of the number of variables) the number of states are frequently related to the size of the constants in the problem. For example, in the knapsack problem the number of states depends on the capacity of the container, while the number of states for the production planning problem is essentially bounded by a function of the maximum production capacity Q. For real-life problems such quantities may be extremely large. This is often exacerbated by the fact that the states may be multi-dimensional. For example in the standard DP formulation for two-dimensional cutting problems the states are defined by rectangles of dimension $X \times Y$. Our two examples were also relatively simple in that the recursive

Table 2.3 The two optimal solutions

Production plan 1					
Period	Starting stock	Make	Sell	Closing stock	Cost (£) production + warehousing
1	1	3	3	3	18,000
2	1	5	6	0	24,000
3	0	3	1	2	20,000
4	2	0	2	0	0
Total					62,000
Production plan 2					
Period	Starting stock	Make	Sell	Closing stock	Cost (£) production + warehousing
1	1	3	3	3	18,000
2	1	5	6	0	24,000
3	0	1	1	0	7,000
4	0	2	2	0	13,000
Total					62,000

relationship relied only on the solutions at the previous stage. Many DP formulations require recursive relationships that use all previous stages, thus necessitating the results of all previous calculations to be stored, resulting in pressure on available memory in addition to long computation times. It is therefore important that some thought is given to reducing the number of states. Findlay et al. (1989) use a model similar to our production planning example to plan daily production in an oil field so as to meet a quarterly production target. The states at each stage are given by the amount still to be produced before the end of the quarter. Thus in their basic model the number of states is given by the number of days in the quarter multiplied by the total quarterly target. However, there are upper and lower bounds on daily production and by using these to produce three bounds on the feasible states at each stage, the total size of the search space can be reduced to less than half its original size.

Although the need to calculate and store all sub-solutions is often seen as a drawback of DP, it can also be viewed as an advantage, as there is no need to carry out a whole new set of calculations if circumstances change. For example, in the production planning example, if for some reason we only managed to make two units instead of three in the third period, we could adopt the optimal policy from that point on simply by selecting the policy given by $F_1(1)$ instead of $F_1(2)$. This flexibility is cited as one of the reasons for the choice of DP as a solution technique by Findlay et al. (1989), as oil production is regularly affected by problems that may cause a shortfall in production on a particular day. Another example of the usefulness of being able to access the solutions to all subproblems without additional computational effort arises in the solution of two-dimensional cutting problems. The bounds used by Christofides and Whitlock (1977) in their branch-and-bound algorithm cited in

the previous section are calculated using a DP approach. The bound at the root node requires the solution to the unconstrained guillotine cutting problem in a rectangle of dimensions $X \times Y$ and the bounds at the other nodes require solutions to the same problem in smaller rectangles. These are precisely the problems solved in the various stages. Therefore, once the bound at the root node has been calculated, bounds for all the other nodes are available without further computation.

It is also worth emphasizing that DP is a very general approach. While this can be regarded as one of its strengths, it can also be a drawback in that there are few rules to guide a beginner in its use for a completely new problem. In many cases it is relatively easy to define the stages of an implementation but it is more difficult to find a suitable definition for the states. Although there are examples of DP being used to solve a variety of problems, the vast majority still lie in the areas of multi-period planning, routing and knapsack type problems where it is relatively easy to adapt existing approaches. We have already mentioned the production planning problem tackled by Findlay et al. (1989). Other examples are a multi-period model for cricketing strategy (Clarke and Norman 1999), a model for optimizing the route taken in orienteering (Hayes and Norman 1984), and a multiple-choice knapsack model for pollution control (Bouzaher et al. 1990).

2.5 Network Flow Programming

2.5.1 Introduction

Network flow programming deals with the solution of problems that can be modeled in terms of the flow of a commodity through a network. At first glance it appears that such models might be very limited in their application, perhaps encompassing areas such as the flow of current in electrical networks, the flow of fluids in pipeline networks, information flow in communications networks and traffic flow in road or rail networks. However, their scope is far wider. They not only encompass a wide range of graph and network problems that appear to have little to do with flows, such as shortest path, spanning tree, matching and location problems, but also model a wide range of other problems ranging from scheduling and allocation problems to the analysis of medical x-rays. Network flow problems can be categorized as integer programming problems with a special structure. For the basic network flow models that deal with homogeneous flows this structure impacts on the solution process in two ways. First, the constraint matrix of the LP formulation has the property that it is *totally unimodular*. This implies that any solution at the extreme points of the feasible region will be integer valued. From a practical point of view this means that as long as all the constants in a problem are integer valued then solution via the simplex method will also be integer valued. Thus integer programs that have the special structure of a network flow problem can be solved without recourse to any of the specialist integer programming techniques described in Chap. 3. However the

structure of the problem also means that it can be solved directly by combinatorial algorithms that are simpler to implement than the full simplex algorithm. The inspiration for and the verification of the optimality of these procedures is rooted in the underlying LP theory. We will start by looking at the maximum flow problem, the simplest but least flexible of the network flow formulations, in order to introduce the basic concepts and building blocks that will be used in the solution of a more flexible model, the minimum cost flow problem.

2.5.2 The Maximum Flow Problem

2.5.2.1 Introduction

The maximum flow problem is that of maximizing the amount of flow that can travel from source S to sink T in a network with capacities or upper bounds on the flow through each of its arcs. The problem can be stated as follows:

Let S = source, T = sink
x_{ij} = flow in arc (i, j)
V = total flow from S to T
u_{ij} = upper bound on arc (i, j)

$$\max V \text{ s.t. } \sum_{\substack{j \\ (i,j)\in A}} x_{ij} - \sum_{\substack{k \\ (k,i)\in A}} x_{ki} \quad \begin{cases} = \ \ v \text{ if } i = \text{S} \\ = -v \text{ if } i = \text{T} \\ = \ \ 0 \text{ for all other } i. \end{cases} \tag{2.17}$$

$$x_{ij} \le u_{ij} \quad \text{for all } (i, j) \in A; \qquad x_{ij} \ge 0. \tag{2.18}$$

Constraints (2.17) ensure that the total flow arriving at the sink and the flow leaving the source are both equal to the objective V, while at all other nodes the amount of flow entering the node is equal to that leaving.

As stated above, the problem could be solved by applying the simplex algorithm to the above formulation. However, a simpler and more intuitive algorithm is the Ford–Fulkerson labeling algorithm (Ford and Fulkerson 1956) which is based on the idea of flow-augmenting chains. Given a feasible flow in a network (i.e. a flow satisfying constraints (2.17) and (2.18)), a flow-augmenting chain from S to T is a chain of arcs (in any orientation) such that the flow can be increased in forward arcs and decreased in backward arcs. This concept will be illustrated with reference to Fig. 2.8.

The two-part labels on the arcs represent the capacity and the flow respectively. A total of seven units of flow travels from S to T. We can increase this flow along a chain of arcs in two ways. We could take chain {(S,B), (B,C), (C,T)} made up entirely of forward arcs and increase the flow by three units. The limit of the increase

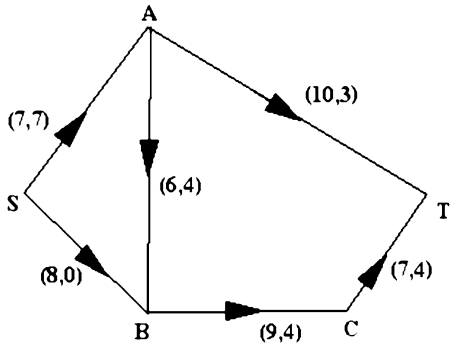

Fig. 2.8 Maximum flow principle

is three as any larger increase would violate the capacity of arc (C,T). Alternatively, we could take the chain {(S,B), (A,B), (A,T)}, in which arc (A,B) is a backward arc as it is oriented in the opposite direction to the chain. Using this chain we can increase the flow by four units by increasing the flow in arcs (S,B) and (C,T) and decreasing the flow in arc (A,B). This will have the effect of diverting four units of flow from (A,B) to (A,T), thus allowing an additional four units to arrive at B from S. The limit of four units derives from the fact that this will reduce the flow in (A,B) to zero. Ford and Fulkerson proved the result that a flow is optimal if and only if it has no flow-augmenting chain. This suggests that the maximum flow problem can be solved by repeatedly finding a flow-augmenting chain and augmenting the flows in the chain, until no flow-augmenting chain exists. The Ford–Fulkerson labeling algorithm provides a mechanism for doing this while guaranteeing to identify a flow-augmenting chain if it exists. It builds up one or more partial chains by successively labeling nodes with a two-part label (p_i, b_i) where p_i defines the predecessor of node i in the chain and b_i is an upper bound on the capacity of the chain up to node i. The objective is either to reach node T in which case a flow-augmenting chain has been found, or to terminate without reaching T, in which case no flow-augmenting chain exists.

2.5.2.2 The Ford–Fulkerson Labeling Algorithm

For maximum flow from source S to sink T.

Notation:

- x_{ij} is the current flow in arc (i, j)
- u_{ij} is the capacity of arc (i, j).

Step 1. Find an initial feasible flow. (All flows $= 0$ will do.)
Find a flow-augmenting chain as follows.
Step 2. Label S $(-, \infty)$ and set all other nodes as unlabeled.
Step 3. Select a forward arc (i, j) from a labeled to an unlabeled node such that $u_{ij} - x_{ij} > 0$, or select a backward arc (j, i) from an unlabeled node to a labeled such that $x_{ij} > 0$.

If no such arc exists STOP *current flow is maximal.*
Step 4. If forward arc label j $(i, \min\{b_i, u_{ij} - x_{ij}\})$.
If backward arc label i $(-j, \min\{b_j, x_{ij}\})$.
Step 5. If T not labeled go to step 3.
Otherwise adjust flow as follows.
Step 6. Trace path back from T using labels p_i to determine preceding node.
Increase flows in forward arcs on the path by b_T and decrease flows in backward arcs on the path by b_T.
Step 7. Go to step 2.

We illustrate the algorithm with reference to Fig. 2.8. We start with the given flow:

Labeling: S$(-,\infty)$, B(S,min$(\infty, 8-0) = 8)$, C(B,min$(8, 9-4) = 5)$, T(C,min $(5, 7-4) = 3)$, $b_T = 3$.

Thus we can augment the flow by three units. Using the labels p_i and working back from p_T we get chain S, B, C, T. All arcs are forward so the flow is increased by three units in each to give (S, A) seven units, (S, B) three units, (A, B) four units, (A, T) three units, (B, C) seven units, (C, T) seven units.

Attempt to find a flow-augmenting chain given updated flows:

Labeling: S $(-,\infty)$, B(S, 5), C(B, 2), A(−B, 4), T(A, 4).

Note that in this small example we can see that labeling C will lead to a dead end. However, we have labeled it here to show that in the general case all labeled nodes need not appear in the final augmenting chain. The chain is given by SBAT where the link from B to A is backward so that flow is decreased by four units on this link and increased an all others. This gives: (S, A) seven units, (S, B) seven units, (A, B) zero units, (A, T) seven units, (B, C) seven units, (C, T) seven units.

Attempt to find a flow-augmenting chain given updated flows:

Labeling: S $(-,\infty)$, B(S, 1), C(B, 1).

No further nodes are available for labeling. Thus the current flow of 14 units is optimal.

2.5.3 Minimum Cost Flow Problem

2.5.3.1 Introduction

Having introduced the basic concepts via the maximum flow problem we now move on to the more flexible model of the minimum cost flow in a closed network. This

problem consists of cyclic network, i.e. a network without a source and sink and has upper and lower bounds on the capacities of the arcs as well as a cost associated with each arc. The objective is to find a feasible flow in the network such that the total cost is minimized. The LP formulation to the problem is as follows.

Let x_{ij}, be the flow in arc (i,j), u_{ij} the upper bound on arc (i,j), l_{ij} the lower bound on arc (i,j) and c_{ij} the cost:

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij} \text{ such that } \sum_{(i,j)\in A} x_{ij} - \sum_{(k,i)\in A} x_{ki} = 0 \forall i \tag{2.19}$$

$$x_{ij} \leq u_{ij}, \forall (i,j) \in A \tag{2.20}$$

$$x_{ij} \geq l_{ij} \forall (i,j) \in A \tag{2.21}$$

$$x_{ij} \geq 0 \forall (i,j) \in A$$

Constraint (2.19) ensures that the flow into each node equals that flowing out, while constraints (2.10) and (2.11) are the upper and lower bound constraints respectively. As with the maximum flow problem this problem can be solved using the simplex method, but there are also a number of specialist solution techniques. Here we introduce one of these, the out-of-kilter algorithm.

2.5.3.2 The Out-of-Kilter Algorithm

The derivation of the out-of-kilter algorithm (Minty 1960; Fulkerson 1961) is based on LP theory, but the algorithm can be implemented without any prior LP knowledge. We associate a real number π(i) with each node i. These numbers are sometimes called *node potentials* and are simply the dual variables associated with the flow balancing constraints (2.9). LP theory states that the solution is optimal if and only if the following conditions are satisfied for each arc (i,j):

Complementary slackness or kilter conditions:

- If $x_{ij} = l_{ij}$ then $c_{ij} + \pi(i) - \pi(j) > 0$
- If $l_{ij} < x_{ij} < u_{ij}$ then $c_{ij} + \pi(i) - \pi(j) = 0$
- If $x_{ij} = u_{ij}$ then $c_{ij} + \pi(i) - \pi(j) < 0$.

For a given arc we can represent these conditions diagrammatically by a two-dimensional plot in which x_{ij} is plotted on the x-axis and $c_{ij} + \pi(i) - \pi(j)$ is plotted on the y-axis. The set of all points satisfying the kilter conditions form two vertical lines defined by $x = l_{ij}$ for $y \geq 0$ and $x = u_{ij}$ for $y \leq 0$, connected by a horizontal line segment from $(l_{ij}, 0)$ to $(u_{ij}, 0)$. This is called the *kilter line* and the diagram is a *kilter diagram*. Figure 2.10 shows the six kilter diagrams relating to the arcs of the network in Fig. 2.9.

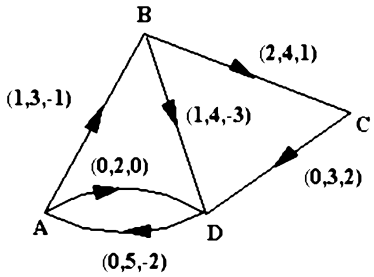

Fig. 2.9 Minimum cost flow problem (the three-part labels are (l_{ij}, u_{ij}, c_{ij}))

The bold lines are the kilter lines. The markers on the diagrams are plots of $(x_{ij}, c_{ij} + \pi(i) - \pi(j))$ for different flows and potentials. When the marker lies on the kilter line the corresponding arc is said to be *in kilter*, if not it is said to be *out of kilter*. We will refer to this figure again when we illustrate the out-of-kilter algorithm.

The out-of-kilter algorithm works with solutions that satisfy the flow balance constraints (2.19), but may violate the upper and lower bounds. By changing the flows or the potentials it gradually moves each arc into kilter without moving any other arc away from the kilter line in the process. It can be stated as follows.

Out-of-kilter algorithm for min. cost flow in a closed network:

Find an initial flow satisfying the flow balance equations and a set of node potentials $\pi(i) \forall i$. Let $y(i,j) = c_{ij} + \pi(i) - \pi(j)$ (note that all flows and potentials equal to 0 will do).
If $y(i,j) > 0$, $x_{\min}(i,j) = \min(x_{ij}, l_{ij})$, $x_{\max} = \max(x_{ij}, l_{ij})$.
If $y(i,j) = 0$, $x_{\min}(i,j) = \min(x_{ij}, l_{ij}), x_{\max}(i,j) = \max(x_{ij}, u_{ij})$.
If $y(i,j) < 0$, $x_{\min}(i,j) = \min(x_{ij}, u_{ij}), x_{\max}(i,j) = \max(x_{ij}, u_{ij})$.
While any arcs out of kilter and procedure is successful do.
Attempt to update flows.
Select and out-of-kilter arc (p,q).
If $x(p,q) > x_{\min}(p,q)$ then set $s = p$, $t = q$, $v = x_{pq} - x_{\min}(p,q)$.
If $x(p,q) < x_{\max}(p,q)$ then set $s = q$, $t = p$, $v = x_{\max}(p,q) - x_{pq}$.
Attempt to find a flow-augmenting chain from s to t to carry up to v additional units of flow without using (p,q) and without exceeding $x_{\max}(i,j)$ in forward arcs or falling below $x_{\min}(i,j)$ in backward arcs.
Note: this can be achieved by starting the labeling algorithm with $s(-,v)$ and respecting $x_{\max}$ and $x_{\min}$ when adjusting the flows.
If successful increase flow in the chain and increase/decrease flow in (p,q) by b_t
Otherwise *attempt to update node potentials as follows*.
Let L be the set of all arcs (i,j) labeled at one end and not the other such that $l_{ij} \le x_{ij} \le u_{ij}$.
For those arcs in L labeled at i: if $y(i,j) > 0$ set $\delta_{ij} = y(i,j)$, otherwise set $\delta_{ij} = \infty$.
For those arcs in L labeled at j: if $y(i,j) < 0$ set $\delta_{ij} = -y(i,j)$, otherwise set $\delta_{ij} = \infty$.

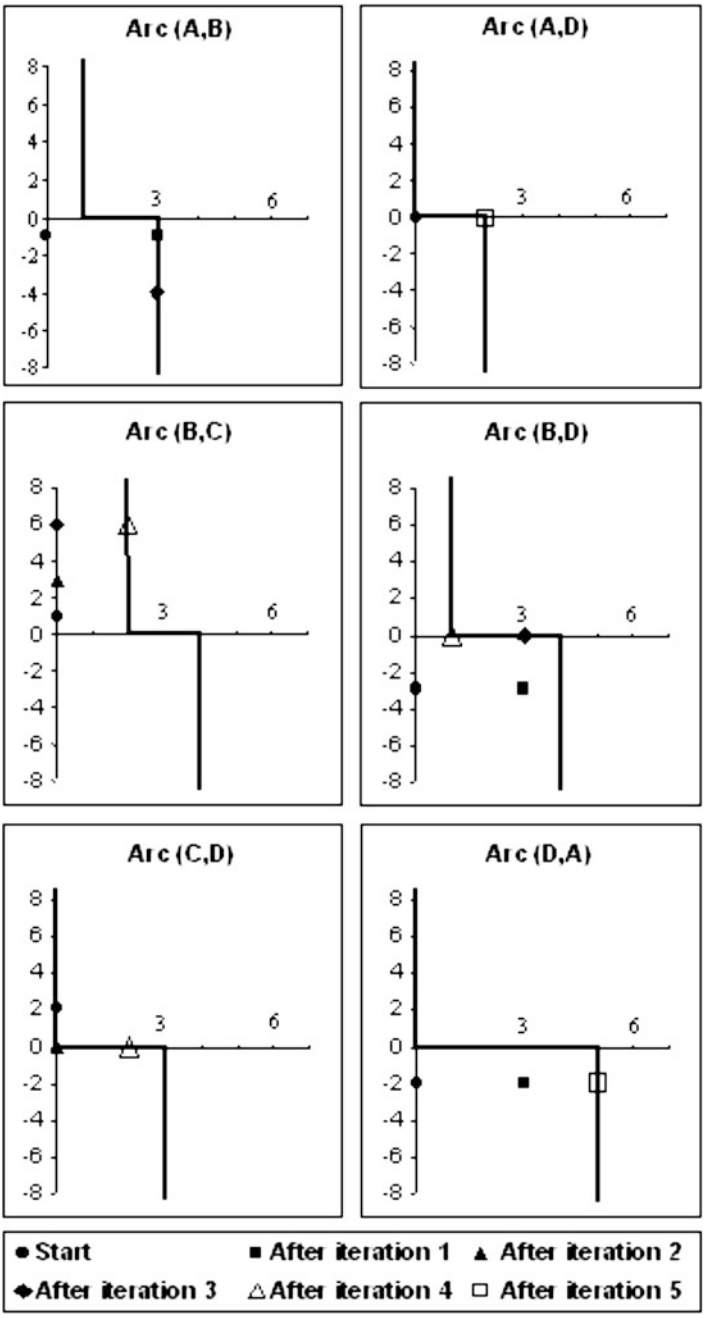

Fig. 2.10 Kilter diagrams for Fig. 2.9 problem

Set $\delta = \min\{\delta_{ij} : (i,j) \in L\}$.
If $\delta = 0$ then stop—no feasible flow.
Otherwise set $\pi(k) = \pi(k) + \delta$ for all unlabeled nodes and update $y(i,j)$ for all arcs labeled at one end and not the other.
Repeat.

When the algorithm terminates, either all the arcs are in kilter and the current flows are optimal or no feasible solution exists.

We illustrate the algorithm with reference to the network in Fig. 2.9 and the kilter diagrams in Fig. 2.10. Note that $x_{\min}$ and $x_{\max}$ are simply the minimum and maximum flow values that ensure that an arc never crosses or moves further away from the kilter line in a horizontal direction and δ serves the same purpose for moves in a vertical direction.

Initialization:

We start with all flows and potentials equal to zero. Thus $c_{ij} + \pi(i) - \pi(j)$ is simply the arc cost for all arcs. This situation is given by the solid circles in Fig. 2.10. Note that arcs (A, D) and (C, D) are already in kilter, but arc (D, A) is not, even though its flow lies within its lower and upper bounds.

Iteration 1:

We select out-of-kilter arc (A, B).

We would like to increase the flow in this arc by three units. Thus $v = 3$, $s = \text{B}$, $t = \text{A}$.

Labeling: B(−, 3), D(B, 3), A(D, 3).

Labeling has been successful. Therefore we increase flow in chain {(B, D), (D, A)}and in arc (A, B) by three units and update $x_{\min}$ and $x_{\max}\forall$ updated arcs. This is shown by the solid squares in Fig. 2.10.

Note that arc (A,B) is now in kilter.

Iteration 2:

Select out-of-kilter arc (B, C), $s = \text{C}$, $t = \text{B}$, $v = 2$.

Labeling: $\text{C}(-,2)$ no further labeling possible as we cannot increase flow in (C, D) without moving away from the kilter line.

$L = \{(\text{C},\text{D})\}$ and as (C, D) is labeled at C $\delta_{\text{CD}} = 2$. Note that this is the maximum distance this arc can move down without leaving the kilter line.

Unlabeled nodes are A, B, D. Increase potentials on these nodes by two to give $\pi(\text{A}) = 2$, $\pi(\text{B}) = 2$, $\pi(\text{C}) = 0$, $\pi(\text{D}) = 2$. This will change the $y(i,j)$ values for arcs (B, C) and (C, D) as shown by the solid triangles. Note that for (C, D) this also changes $x_{\max}$.

Iteration 3:

We try again with arc (B, C), $s = \text{C}$, $t = \text{B}$, $v = 2$.

Labeling: $\text{C}(-,2)$, D(C, 2), A(D, 2) no further labeling possible as flow in (A, B) is at full capacity and decreasing flow in (B, D) will move away from the kilter line.

$L = \{(\text{A},\text{B}),(\text{B},\text{D})\}$. $\delta_{\text{AB}} = \infty$, $\delta_{\text{BD}} = 3$, $\delta = 3$.

Unlabeled node B. Increase $\pi(B)$ by 3 giving $\pi(A) = 2$, $\pi(B) = 5$, $\pi(C) = 0$, $\pi(D) = 2$ and changing positions of arcs (A, B), (B, C) and (B, D) as given by the solid diamonds. Arc (B, D) is now in kilter and has a new value for $x_{\min}$.

Iteration 4:

We try again with arc (B, C) $s = C$, $t = B$, $v = 2$.

Labeling: C(−, 2), D(C, 2), B(−D, 2). t is labeled.

Adjust flows in chain {(C, D), (B, D)}and increase flow in (B, C) by two units as shown by the unfilled triangles. Arc (B, C) is now in kilter.

Iteration 5:

Select only remaining out-of-kilter arc (D, A), $s = A$, $t = D$, $v = 2$.

Labeling; A(−, 2), D(A, 2). t is labeled. Increase flow in (D, A) and (A, D) by two units as shown by the unfilled squares.

All arcs are in kilter, therefore solution is optimal. Flows are as given by the last update for each arc, i.e. $x_{AB} = 3$, $x_{AD} = 2$, $x_{BC} = 2$, $x_{BD} = 1$, $x_{CD} = 2$ and $x_{DA} = 5$, with a total cost of -13.

2.5.4 Other Issues

The previous sections introduced two relatively simple algorithms for the maximum flow and minimum cost flow problems. The out-of-kilter algorithm also has the advantage that it is easy to find an initial solution as the upper and lower bounds do not need to be satisfied. However, these are not necessarily the most efficient algorithms in each case. For example, it possible to design pathological cases of the max flow problem for which the number of iterations made by the Ford–Fulkerson algorithm is only bounded by the capacity on the arcs. There are also inherent inefficiencies in the algorithms in that subsequent iterations may need to recalculate labels already calculated in previous iterations. A more efficient algorithm is the network simplex algorithm which can be found in Ahuja et al. (1993). The algorithm is so called because it can be shown that the iterations used to improve an initial solution correspond exactly to those computed by the simplex method, but the special structure of the problem means that the algorithm can be expressed purely in network terms without recourse to the simplex tableau. For large problems the additional efficiencies of the network simplex algorithm may pay off, but for small to moderate problems the out-of-kilter algorithm should be fast enough, and has the advantage that it is easy to implement from scratch and that code is available from a number of sources.

As already mentioned, the efficiency of network flow solution algorithms means that it is worthwhile attempting to model any new problem as a network flow problem. A wide range of problems can be modeled using the two formulations already given. However, the scope of network flow approaches is even wider when we consider problems that can be modeled using the dual formulations of network flow problems. Examples of this type of model include Mamer and Smith's (1982) approach to an infield repair kit problem and Johnson's (1968) open-cast mine plan-

ning problem. Even when the network model is not flexible enough to incorporate all the constraints in a problem it may still provide an effective optimization tool. For example Glover et al. (1982) model the problem of determining the allocation of space on aircraft to different fare structures as a minimum cost flow problem. Their model does not incorporate all the constraints needed to model the problem and thus infeasible solutions may be returned. If this occurs, the solution is excluded from the model and the solution process reiterated until a feasible solution results. The authors comment that this was far more efficient than attempting to include all such constraints directly into an IP model.

We have limited our focus to problems in which flows are homogeneous and there is no gain or leakage of flow along an arc. Problems in which multiple commodities share the arc capacities and problems where flow is not preserved along an arc have also been widely studied. Unlike the simple problems covered here both these problems are NP-complete. Algorithms for solving such problems are beyond the scope of this chapter but can be found in many network flow texts. Ahuja et al. (1993) give a comprehensive treatment of network flows including models, algorithms and practical applications.

2.6 Some Useful Models

The previous sections have outlined three general approaches to optimization problems. In this section we focus on two classes of problem that frequently occur as subproblems in the solution of larger or more complex problems over a wide range of application areas. In each case solution approaches based on the methods covered in the previous sections are described.

2.6.1 Shortest-Path Problems: DP Approaches

The first class of common subproblems are those involving shortest paths. As observed in Sect. 2.3.2, the branch-and-bound shortest-path algorithm presented there is not the most efficient way of tackling such problems. In this section more efficient approaches based on DP are presented. We start by considering the problem of finding the shortest path from a source vertex s to all other vertices in a graph.

2.6.1.1 Bellman's Shortest-Path Algorithm

This can be solved by Bellman's shortest-path algorithm in which the stages are defined by the number of links allowed in the path and the states are defined by the vertices. The formulae for the starting state and the recursive relationship can be defined as follows:

$$F_0(v) = 0 \text{ if } v = s, \quad F_0(v) = \infty \text{ otherwise,}$$

$$F_k(v) = \min\left\{F_{k-1}(v) \min_{w \in Q_{k-1},(w,v)\in E} F_{k-1}(w) + c_{vw}\right\},$$

where Q_k is the set of vertices whose values were updated at stage k and E is the set of links in the network.

Bellman's shortest-path algorithm can then be stated as follows:

Set $F_0(v) \forall v \in V$
While $Q_k \neq \emptyset$ and $k \leq n$ do
Calculate $F_k(v) \forall v \in V$ and determine Q_k
End while

If $Q_k = \emptyset$ then $F_k(v)$ defines the length of the shortest path from s to $v \forall v$.

If $k = n$ and $Q_k \neq \emptyset$ then the network contains negative cost circuits and shortest paths cannot be defined.

If paths from every vertex to every other are required then rather than execute Bellman's algorithm n times it is more efficient to use Floyd's shortest-path algorithm (Floyd 1962).

2.6.1.2 Floyd's Shortest-Path Algorithm

This is also a DP type approach but in this case $F_k(i,j)$ represents the shortest path between i and j allowing only vertices 1 to k as intermediate points. The initial states are given by $F_0(i,j) = C_{ij}$, where C_{ij} is the cost of link (i,j), and the recursive relationship by $F_k(i,j) = \min\{F_{k-1}(i,j),\ F_{k-1}(i,k) + F_{k-1}(k,j)\}$. As $F_k(i,i) = 0 \forall k$ unless the network contains a negative cost circuit neither $F_k(i,k)$ or $F_k(k,j)$ will be updated during iteration k. Therefore the subscript k is usually dropped in practice and matrix $F(i,j)$ is overwritten at each iteration. The algorithm can then be stated as follows.

Floyd's algorithm for the shortest path between all pairs of vertices in an arbitrary graph:

Step 1
$k = 0, F(i,j) = c(i,j) \forall (i,j)$
Step 2
$\forall k = 1,n$
$\forall i = 1,n$ s.t. $i \neq k$ and $F(i,k) \neq \infty$
$\forall j = 1,n$ s.t. $j \neq k$ and $F(k,j) \neq \infty$
$F(i,j) = \min\{F(i,j), F(i,k) + F(k,j)\}$
if $F(i,i) < 0$ for any i STOP. *(negative cost circuit detected.)*
end loops

It should be noted that if all costs are non-negative then an algorithm due to Dijkstra (1959) is more efficient than Bellman's algorithm. Dijkstra's algorithm can be found in most basic texts covering graph and network algorithms (e.g. Ahuja et al. 1993).

2.6.2 Transportation Assignment and Transhipment Problems: Network Flow Approaches

In this section we consider a second commonly occurring class of subproblems: the family of transportation type problems, including the assignment and transhipment problems. All three problems can be modeled as minimum cost flow problems. Once the appropriate model has been derived, solutions can be obtained using the out-of-kilter algorithm or any other minimum cost flow algorithm. The focus of this section is therefore on defining appropriate models.

2.6.2.1 The Transportation Problem

The transportation problem is that of determining the amount of product to be supplied from each of a given set of supply points to each of a given set of demand points, given upper bounds on availability at each supplier, known demands at each demand point and transportation costs per unit supplied by supplier i to demand point j. The problem can be formulated as follows:

$$\begin{aligned} \min & \sum_{i=1}^{n}\sum_{j=1}^{m} c_{ij}x_{ij} \\ \text{s.t.} & \sum_{i=1}^{n} x_{ij} \geq d_j \\ & \sum_{j=1}^{m} x_{ij} \leq s_i. \end{aligned} \tag{2.22}$$

We can model this problem as a minimum cost flow network by defining nodes, arcs, lower and upper bounds and costs as follows:

Nodes:
A dummy source node S.
A dummy sink node T.
One node for each supplier, $i = 1, n$.
One node for each demand point j, $j = 1, m$.
Arcs:
An arc from each supplier node i to each demand node j with lower bound $= 0$, upper bound $= s_i$ and cost $= c_{ij}$. Note if all $c_{ij} \geq 0$ then the upper bound can be replaced by $\min\{s_i, d_j\}$.
An arc from S to each supplier node with lower bound $= 0$, upper bound $= s_i$ and cost $= 0$.

An arc from each demand node, j, to T with lower bound $= d_j$ upper bound $= M$, where M is some suitably large number and cost $= 0$. Note if all $c_{ij} \geq 0$ then the upper bound can be replaced by d_j.

An arc from T to S with lower bound $= 0$, upper bound $= \sum_{i=1}^{n} s_i$, cost $= 0$.

The minimum cost flow in the network defined above will give the optimal solution to the transportation problem, and the flows in the arcs (i, j) define the value of the variables x_{ij}.

Many management science and operational research texts cover the stepping-stone algorithm or MODI for the transportation problem. It is interesting to note that the rules used by the MODI method for determining those x_{ij} that should be considered for increase are precisely the kilter conditions that define those arcs that lie to the left of the kilter line in the network flow model, and that the stepping-stone process for finding a suitable path to update the solution corresponds to finding a flow-augmenting chain. Many texts also suggest a heuristic method known as Vogel's approximation method to obtain an initial solution. This approach can be taken to find an initial flow in the above model, thus reducing the number of iterations required when compared with a starting solution of 0.

2.6.2.2 The Assignment and Transhipment Problems

Several relatives of the transportation problem are also encountered as subproblems. Here we consider the assignment and transhipment problems.

The assignment problem is that of assigning tasks to resources. It is assumed that there are n tasks and n resources and a cost c_{ij} associated with the assignment of task i to resource j. The objective is to assign each task to exactly one resource and each resource to exactly one task such that the total cost of the assignments is minimized. This problem can be regarded as a transportation problem in which $n = m$ and $s_i = d_j = 1 \forall i$ and $\forall j$. Thus any assignment problem can be modeled using the transportation model above with the appropriate values for s_i and d_j.

The transhipment problem is a transportation problem in which goods may travel from supplier to demand point via any or none of a set of intermediate points—known as transhipment points. These points may also be supply points or demand points in their own right. The network flow model for the transportation problem can easily be adapted to the transhipment model by adding new nodes for the transhipment points and adding arcs from each supply point and to each demand point with appropriate bounds and transhipment costs. If a transhipment point is also a supply point or demand point then it is also connected to S or T respectively.

2.6.2.3 Other Useful Models

The above classes of problem are two of the most frequently occurring subproblems that can be solved using the methods covered in this chapter. Two other classes of

problem that deserve special mention are the binary and bounded knapsack problems and matching problems in bipartite graphs.

We have already presented a DP approach for the unbounded knapsack problem in Sect. 2.3. The binary version of the problem occurs when at most one piece of each type is available. An effective solution approach to this problem is a tree search in which there are two branches from each node, one corresponding to fixing the value of a variable x_i at 0 and the other to fixing the same variable at 1. Upper and lower bounds are easily obtained, simply by sorting the variables in v_i/w_i order and fitting and adding each to the knapsack in turn, until the capacity exceeded. In its basic form the algorithm is an implementation of the standard Integer Programming branch-and-bound algorithm as discussed in Chap. 3. However, the bounds can be improved by adding problem specific information. Martello and Toth (1990) give details of several variants of the method and suggest a series of increasingly powerful bounds. They also show how the method can be extended to the more general case in which the variables are not binary, but are restricted by upper bounds.

Matching problems also occur widely. A matching in a graph is a set of edges no two of which have a vertex in common. Maximum matching problems in arbitrary graphs can be solved using the blossom algorithm due to Edmonds, and described in Ahuja et al. (1993). However, if the graph is bipartite, i.e. the vertices can be partitioned into two subsets such that there are no edges between any two vertices in the same subset, then the problem can be solved more simply using network flow type algorithms. Such models are useful in practice as bipartite graphs often appear in allocation or scheduling problems.

2.7 Promising Areas for Future Application

In this section we outline some potential areas for the future application of the methods described in this chapter. All three approaches have been used extensively in the solution of a broad range of problems for several decades. The increase in computer power available to individuals and organisations since their introduction has lead to a continuously expanding range of problems that can be solved to optimality within a feasible amount of time. At the same time theoretical advances in potential application areas have lead to improved bounds, once again increasing the scope of branch-and-bound approaches. There are currently many researchers active in each of the areas. Thus it is likely that we will continue to see new theoretical developments as well as new branch-and-bound or DP implementations for old problems and the application of classical approaches to new practical problems. However, there is also considerable potential for combining these techniques with some of the more modern approaches covered elsewhere in this book. One option is to enhance the performance of a tree search algorithm by using a solution obtained by a powerful heuristic to provide a good upper bound. Alternatively the techniques described in both this chapter and Chap. 3 can be used to enhance the performance of a heuristic approach. As evidenced by the survey article of Fernandes and Lourenço (2007), this has developed into a popular and promising area of research. Such inte-

gration can be at one of three levels: pre- or post-processing, true hybridization and cross-fertilization of ideas. We take a brief look at each of these in turn.

2.7.1 Pre- and Post-processing

The simplest form of integration is to use a classical approach as part of a staged solution to a problem. Tree search approaches are often used to enumerate all the variables required for the optimization phase of the problem. For example in crew scheduling problems the set of feasible tours of duty satisfying all the necessary constraints are enumerated first and then these are used as input to an optimization algorithm, or in the case of timetabling and scheduling problems the allocation of events to rooms may be carried out in a post-processing phase once the schedule has been determined. In other cases pre-processing may be used to reduce the size of the solution space. For example, Dowsland and Thompson (2000) solve a nurse scheduling problem using tabu search. Before calling the tabu search routine they use a tree search approach to solve a modified knapsack problem that enables them to determine the precise number of additional nurses required to cover the weekly demand on a ward. This allows them to minimize the size of the solution space and to simplify the evaluation function in the tabu search part of the solution process.

2.7.2 True Hybrids

A greater degree of integration is provided by true hybrid approaches in which a classical approach is embedded into a modern search tool or vice versa. Many researchers have tried this. There are several instances of the integration of branch and bound with genetic algorithms. For example Cotta et al. (1995) use a tree search to find the best child from a set of possibilities given by a loosely defined crossover, while Nagar et al. (1995) use a genetic algorithm to search out a promising set of ranges on the values of a series of variables and then use a tree search to find the optimal solution within the range. Classical techniques have also been embedded into neighborhood search approaches such as simulated annealing, tabu search and variable-depth search. For example, there are many problems in which the variables can be partitioned into two sets, A and B, such that if the values of A are fixed the problem of optimizing B reduces to a network flow problem. The size of the solution space can be reduced to cover only the variables in A, with each solution being completed by solving for the variables in B. Network flow type problems are especially amenable to this sort of role, as neighborhood moves will typically involve changing a cost or bound, or adding or deleting an arc. All of these changes can be accommodated in the out-of-kilter algorithm using the previous solution to initialize the solution process for its neighbor(s), thus minimizing the computational effort in resolving each new subproblem. Examples of this type of strategy include

Hindi et al. (2003) who solve transhipment subproblems to complete solutions in their variable depth search approach to the lot sizing problem, and Dowsland and Thompson (2000) who use a network flow problem to allocate nurses on days to the morning or afternoon shifts. This reduces the number of variables in the search space for each nurse by up to a factor of 32.

As discussed in other chapters, many neighborhood searches can be improved by extending the size of the neighborhood. One way of doing this is to introduce chains of moves. Rather than select a chain at random or enumerate all chains in the neighborhood, it makes sense to find optimal or improving chains directly using some form of shortest-path algorithm. For example, Dowsland (1998) uses a mixture of tree search and Floyd's algorithm to find improving chains of moves for a nurse scheduling problem. More recently, Abdullah et al. (2007) have used a similar strategy in a tabu search solution to a timetabling problem. Other forms of compound moves may not involve chains. In such cases other techniques can be used to find optimal moves. For example, Potts and van de Velde (1995) use DP to search for good moves in neighborhoods made up of multiple swaps for the TSP. Gutin (1999) also considers the TSP but utilizes a bipartite matching problem in order to determine the best option from a neighborhood defined by removing and reinserting k vertices in the tour. Other researchers have worked with this neighborhood for different definitions of k. Further examples can be found in Ahuja et al. (2002).

2.7.3 Cross-fertilization

As well as true hybrids such as those outlined above there are also examples of cross-fertilization of ideas in which an ingredient from a classical approach has been embedded into a more modern method. One example is the incorporation of bounds into a heuristic search in order to avoid or leave non-promising areas of the solution space. Hindi et al. (2003) use this strategy in their variable depth approach to avoid wasting time in solving transhipment problems to complete solutions that can be shown to be bounded, while (Dowsland 1998) uses bounds in combination with a tabu list to avoid wasting time in areas of the search space that cannot lead to a better solution than the best found so far. Similar approaches have been taken with genetic algorithms by Tamura et al. (1994) and Dowsland et al. (2006) who use mutation and crossover respectively to destroy partial solutions that exceed a tree search type bound. A second example of cross-fertilization is in the use of ejection chains, a concept suggested by Glover and Laguna (1997) which is a generalization of alternating chains—a term used for the way in which flow is updated in certain classes of network flow problems.

2.8 Tricks of the Trade

2.8.1 Introduction

For newcomers to the field the prospect of applying any of the above techniques to a given problem can appear daunting. This section suggests a few tips on overcoming this feeling and getting started on a basic implementation, and then going on to identify possible areas for algorithm improvement. We start with a few general observations that apply to all three techniques and follow this with more specialist advice for each of the techniques in turn.

1. *Get a basic understanding of the technique and how it might be applied to a given problem*. This involves reading suitable books and articles. Due to the relatively long history of the techniques covered in this chapter there is a wealth of introductory material available. Although some of the seminal material and/or early texts in each field provide valuable insights and technical detail they often use specialist terminology and can be difficult to understand. Therefore they are probably best left until some practical experience has been gained, and the best place to start would be one of the more up-to-date texts given in the References and Sources of Additional Information at the end of this chapter. These will provide a thorough background but may not include examples that are closely related to the reader's own problem. It is therefore desirable to supplement these sources with journal articles related to the relevant application area.
2. *Don't reinvent the wheel*. There are many published articles describing implementations of these techniques to classical combinatorial optimization problems. If your problem can be formulated as, or is closely related to, one of these problems then it is likely that an effective implementation has already been published, and in many cases suitable code may also be readily available. Sometimes the relationship to a classical problem is obvious from the problem definition, but this is not always the case. It is therefore well worth considering different ways of modeling a problem e.g. using graph-theoretic models or trying different LP type formulations and comparing these with well-known classical problems.
3. *Don't be too pessimistic*. Although the complexity of most branch and bound and dynamic programming algorithms is likely to be exponential (or at best pseudo-polynomial) they can still be effective tools for solving moderately sized problems to optimality, and may well compete with the more modern methods described later in this book when used as heuristics. While there may be many good practical reasons for selecting a heuristic rather than an exact approach to a problem, many real-life NP-hard problems have proved amenable to solution by both branch and bound and dynamic programming. However, this may require a little ingenuity on the part of the algorithm designer. If the initial implementation seems too slow or memory intensive it is well worth spending some time and effort trying to make improvements. The situation with regard to network flow programming is somewhat different in that the simple models covered in detail in

this chapter can all be solved to guaranteed optimality in polynomial time and are the obvious first choice approaches for problems that have the required structure.

2.8.2 Tips for Branch and Bound

The issues arising in developing and implementing a branch and bound approach can be broadly divided into three categories:

1. *Representations*. For most problems there are several different potential tree search representations and search strategies each of which will impact differently on solution time and quality. In order to appreciate this it is worth reading articles that use different representations for the same problem. Before starting to code any implementation think about what you have read in relation to your own problem. You should also think about whether you want to apply a depth-first search or if there may be advantages in a more memory-intensive breadth or best-first search.
2. *Coding*. Although it is relatively easy for a beginner to construct a tree search on paper it can be difficult to translate this into computer code. We therefore recommend finding code or detailed pseudo-code for a similar tree structure to the one you are planning on and using it as a template for your own program structure. It is also often helpful to code the three operations of branching, backtracking and checking the bounding conditions separately. It is also a good idea to start with very simple bounds and a branching strategy based on a natural ordering of the vertices and test on a small problem, building up more sophisticated bounds and branching strategies as necessary.
3. *Performance*. Once the basic code is working the following pointers will help in getting the best out of an implementation.

   Analyze the time taken to calculate the bounds and their effectiveness in cutting branches and consider how the trade-off influences overall computation time. While computationally expensive bounds or branching strategies are likely to increase solution times for small problems they may well come into their own as problem size grows.

   In the same way as it is important to test heuristics for solution quality on problem instances that closely match those on which they are to be used in terms of both problem size and characteristics, so it is important to ensure that a branch and bound approach will converge within a realistic time-scale on typical problem instances.

   Remember that branch and bound can be used as a heuristic approach, either by stopping after a given time or by strengthening the bound to search for solutions that improve on the best so far by at least $\alpha\%$. In this case it is important to consider whether you want to bias the search towards finding good solutions early or precipitating the bounding conditions earlier.

2.8.3 Tips for Dynamic Programming

Due to its generality, getting started with dynamic programming can be difficult but the following pointers may be of assistance:

1. *Representation*. When faced with a new problem simply deciding on the definition of stages and states that form the basis of a correct dynamic programming formulation can be difficult, and access to a successful formulation to a similar problem can be invaluable. As a starting point it is worth considering if the problem can be classified as a multi-period optimization problem, routing problem, or knapsack type problem. If so then there are a wealth of examples in standard texts or journal articles that should help.

   Remember that the objective is to produce something that is considerably more efficient than complete enumeration. Therefore it is important to ensure that the calculations relating to the first stage are trivial, and that there is a relatively simple recursive relationship that can be used to move from one stage to the next. If there is no apparent solution when defining the stages in a forwards direction then consider the possibility of backward recursion.
2. *Performance*. DP can be expensive both in terms of computational time and storage requirements and the main cause of this is the number of states. Therefore once a basic structure has been determined it is worthwhile considering ways of cutting down on the number of states that need to be evaluated. It is also worthwhile ensuring that only those stages/states that may be required for future reference be stored. If the environment requires the solution of several similar problems—e.g. if the DP is being used to calculate bounds or optimize large neighborhoods—consider whether or not the different problems could all be regarded as subproblems of one large problem, thereby necessitating just one DP to be solved as a pre-processing stage. As with branch and bound it is important to ensure that the time and memory available are sufficient for typical problem instances.

2.8.4 Tips for Network Flow Programming

The problems facing a beginner with network flow programming are different in that there are standard solution algorithms available off-the-peg, so that the only skill involved is that of modeling. Because these algorithms are readily available and operate in polynomial time then it is certainly worth considering whether any new problem might be amenable to modeling in this way. Some tips for recognizing such problems are given below.

If the problem involves physical flows through physical networks then the model is usually obvious. However, remember that network flow models simply define an abstract structure that can applied to a variety of other problems. Typical pointers to possible network flow models are allocation problems (where flow from i to j

represents the fact that i is allocated to j), sequencing problems (where flow from i to j represents the fact that i is a predecessor of j), and problems involving the selection of cells in matrices where flow from (i to j represents the selection of cell (i, j)). However, this list is by no means exhaustive.

Other tips for modeling problems as network flows are to remember that network links usually represent important problem variables and that even if the given problem does not have an obvious network flow structure the dual problem might. In addition, although typical practical problems will be too large and complex to draw the whole network it is worthwhile making a simplified sketch of potential nodes and links. If the problem has complexities such as hierarchies of resources, multiple pricing structures etc. this may be facilitated by simplifying the problem first and then trying to expand it without compromising the network flow structure.

2.9 Conclusions

Although the classical techniques described in this chapter were developed to meet the challenges of optimization within the context of the computer technology of the 1950s they are still applicable today. The rapid rate of increase of computing power per unit cost in the intervening years has obviously meant a vast increase in the size of problems that can be tackled. However, this is not the sole reason for the increase. Research into ways of improving efficiency has been continuous both on a problem-specific and generic basis—for example all the techniques lend themselves well to parallelization. Nevertheless, they do have drawbacks. The scope of network flow programming is limited to problems with a given structure, while the more general methods of branch and bound and DP may require vast amounts of computer resource. DP solutions to new problems are often difficult to develop, and it may not be easy to find good bounds for a branch-and-bound approach to messy practical problems. Hence the need for the more recent techniques described elsewhere in this volume.

However, it should be noted that these techniques have not eclipsed the classical approaches, and there are many problems for which one of the techniques described here is still the best approach. Where this is not the case and a more modern approach is appropriate it is still possible that some form of hybrid may enhance performance by using the strengths of one approach to minimize the weaknesses of another.

Sources of Additional Information

Basic material on dynamic programming and network flows can be found in most undergraduate texts in management science and OR. Their treatment of branch and bound tends to be limited to integer programming. The generic version of branch

and bound covered in this chapter can be found in most texts on combinatorial algorithms/optimization, or in subject-oriented texts (e.g. algorithmic graph theory, knapsack problems).

Below are a small sample of relevant sources of information:

- http://www2.informs.org/Resources/ (INFORMS OR/MS Resource Collection—links to B&B, DP and Network Flow sources)
- http://people.brunel.ac.uk/mastjjb/jeb/or/contents.html (J. Beasley, Imperial College)
- http://jorlin.scripts.mit.edu/ (J. Orlin, MIT)
- http://math.illinoisstate.edu/sennott/ (Sennott 1998)
- http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/ (Lectures and Videos on DP and other algorithms)
- http://en.wikipedia.org/wiki/Dynamic_programming
- http://en.wikipedia.org/wiki/Branch_and_bound
- Bather, J. (2000). Decision Theory: An Introduction to Dynamic Programming and Sequential Decisions. Wiley.
- Bellman, R. (2003). Dynamic Programming. Dover Publications.
- Christofides, N. (1975). Graph Theory—An Algorithmic Approach. Academic Press.
- Hu, T. C. (1981). Combinatorial Algorithms. Addison-Wesley.
- Lawler, E. L. et al. (1990). The Travelling Salesman Problem. Wiley.
- Nemhauser, G. L., Wolsey, L. A. (1988). Integer and Combinatorial Optimisation. Wiley.
- Papadimitriou, C. H. (1982). Combinatorial Optimisation: Algorithms and Complexity. Prentice-Hall.
- Reingold, E. M. et al. (1977). Combinatorial Algorithms. Prentice-Hall.
- Sennott, L. I. (1998). Stochastic Dynamic Programming, Wiley.
- Taha, H. A. (2002). Operations Research. Prentice-Hall.

References

Abdullah S, Ahmadi S, Burke EK, Dror M and Mc Collum B (2007) A tabu-based large neighbourhood search methodology for the capacitated examination timetabling problem. J Oper Res Soc 58:1494–1502

Ahuja RK, Magnanti TL, Orlin JB (1993) Network flows: theory, algorithms and applications. Prentice-Hall, Englewood Cliffs

Ahuja RK, Ergun O, Orlin JB, Punnen AO (2002) A survey of very large-scale neighbourhood search techniques. Discret Appl Math 123:75–102

Anderson DR, Sweeney DJ, Williams TA (1997) Introduction to management science: quantitative approaches to decision making. West Publishing, Eagan

Balakrishnan VK (1997) Schaum's outline of graph theory (Schaum's Outline Series). Schaum Publishers, Mequon

Balas E, Christofides N (1981) A restricted Lagrangian approach to the travelling salesman problem. Math Prog 21:19–46

Beasley JE (1985) An exact two-dimensional non-guillotine cutting tree-search procedure. Oper Res 33:49–64

Bellman R (1957) Dynamic programming. Princeton University Press, Princeton

Bouzaher A, Braden JB, Johnson GV (1990) A dynamic programming approach to a class of non-point source pollution control problems. Manage Sci 36:1–15

Bron C, Kerbosch J (1973) Finding all cliques of an un-directed graph—alg 457. Commun ACM 16:575–577

Brown JR (1972) Chromatic scheduling and the chromatic number problem. Manage Sci 19:456–463

Christofides N, Whitlock C (1977) An algorithm for two-dimensional cutting problems. Oper Res 25:30–44

Clarke SR, Norman JM (1999) To run or not?: some dynamic programming models in cricket. J Oper Res Soc 50:536–545

Cotta C, Aldana JF, Nebro AJ, Troya JM (1995) Hybridising genetic algorithms with branch and bound techniques for the resolution of the TSP. In: Poras CC et al (eds) Proceedings of the international conference on artificial neural networks and genetic algorithms, Ales, pp 277–280

Dantzig GB (1951) Maximization of a linear function of variables subject to linear inequalities. In: Koopmans TC (ed) Activity analysis of production and allocation. Wiley, New York

Dijkstra EW (1959) A note on two problems in connection with graphs. Numer Math 1:269

Dowsland KA (1987) An exact algorithm for the pallet loading problems. EJOR 31:78–84

Dowsland KA (1998) Nurse scheduling with tabu search and strategic oscillation. EJOR 106:393–407

Dowsland KA, Thompson JM (2000) Solving a nurse scheduling problem with knapsacks, networks and tabu search. J Oper Res Soc 51:825–833

Dowsland KA, Herbert EA, Kendall G (2006) Using tree search bounds to enhance a genetic algorithm approach to two rectangle packing problems. EJOR 168:390–402

Erlenkotter D (1978) A dual-based procedure for uncapacitated facility location. Oper Res 26:992–1009

Fernandes S, Lourenço HR (2007) Hybrids combining local search heuristics with exact algorithms. In: Rodriguez F, Mélian B, Moreno JA, Moreno JM (eds) Proc V Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados, MAEB'2007, Tenerife, 14–16 Feb 2007, pp 269–274

Findlay PL, Kobbacy KAH, Goodman DJ (1989) Optimisation of the daily production rates for an offshore oil field. J Oper Res Soc 40:1079–1088

Fisher ML (1985) An applications oriented guide to Lagrangian relaxation. Interfaces 15:10–21

Floyd RW (1962) Algorithm 97—shortest path. Commun ACM 5:345

Ford LR, Fulkerson DR (1956) Maximal flow through a network. Can J Math 18:399–404

Fulkerson DR (1961) An out-of-kilter method for minimal cost flow problems. SIAM J Appl Math 9:18–27

Garfinkel RS, Nemhauser GL (1969) The set partitioning problem: set covering with equality constraints. Oper Res 17:848–856

Glover F, Laguna M (1997) Tabu search. Kluwer, Dordrecht

Glover F, Glover R, Lorenzo J, Mcmillan C (1982) The passenger mix problem in the scheduled airlines. Interfaces 12:73–79

Golumbic MC (1980) Algorithmic graph theory and perfect graphs. Academic, New York

Gutin GM (1999) Exponential neighbourhood local search for the travelling salesman problem. Comput OR 26, 313–320

Hayes M, Norman JM (1984) Dynamic programming in orienteering—route choice and the siting of controls. J Oper Res Soc 35:791–796

Held M, Karp RM (1970) The travelling salesman problem and minimum spanning trees. Oper Res 18:1138–1162

Hindi KS, Fleszar K, Charalambous C (2003) An effective heuristic for the CLSP with setup times. J Oper Res Soc 54:490–498

Jarvinen P, Rajala J, Sinervo H (1972) A branch and bound algorithm for seeking the p-median. Oper Res 20:173

Johnson TB (1968) Optimum pit mine production scheduling. Technical report, University of California, Berkeley

Kamarkar NK (1984) A new polynomial-time algorithm for linear programming. Combinatorica 4:373–395

Khachiyan LG (1979) A polynomial algorithm in linear programming. Dokl Akad Nauk SSSR 244:1093–1096 (in Russian) (English transl.: Sov Math Dokl 20:191–194(1979))

Little JDC, Murty KG, Sweeney DW, Karel C (1963) An algorithm for the travelling salesman problem. Oper Res 11:972–989

Martello S, Toth P (1981) A branch and bound algorithm for the zero-one multiple knapsack problem. Discret Appl Math 3:275–288

Martello S, Toth P (1990) Knapsack problems: algorithms and computer implementations. Wiley, New York

Mamer JW, Smith SA (1982) Optimising field repair kits based on job completion rate. Manage Sci 28:1328–1334

Minty GJ (1960) Monotone networks. Proc R Soc 257A:194–212

Nagar A, Heragu SS, Haddock J (1995) A meta-heuristic algorithm for a bi-criteria scheduling problem. Ann OR 63:397–414

Potts CN, van de Velde SL (1995) Dynasearch—iterative local improvement by dynamic programming. Part 1: the TSP. Technical report, University of Twente

Ross GT, Soland RM (1975) A branch and bound algorithm for the generalised assignment problem. Math Prog 8:91–103

Tamura H, Hirahara A, Hatono I, Umano M (1994) An approximate solution method for combinatorial optimisation—hybrid approach of genetic algorithm and Lagrangian relaxation method. Trans Soc Instrum Control Eng 130:329–336
Zykov AA (1949) On some properties of linear complexes. Math Sb 24:163–188

http://www.springer.com/978-1-4614-6939-1

Search Methodologies
Introductory Tutorials in Optimization and Decision Support Techniques
Burke, E.K.; Kendall, G. (Eds.)
2014, XIV, 716 p. 135 illus., 15 illus. in color., Hardcover
ISBN: 978-1-4614-6939-1