Journal of Heuristics, 2: 55–85 (1996)

Case-Based Reasoning for Repetitive Combinatorial Optimization Problems, Part I: Framework

DAVID R. KRAAY
Krannert School of Management, Purdue University, West Lafayette, IN 47907
e-mail: dkraay@mgmt.purdue.edu

PATRICK T. HARKER
Department of Systems Engineering, School of Engineering and Applied Science, University of Pennsylvania, Philadelphia, PA 19104-6315 *e-mail: harker@eniac.seas.upenn.edu*

Abstract

This article presents a case-based reasoning approach for the development of learning heuristics for solving repetitive operations research problems. We first define the subset of problems we will consider in this work: repetitive combinatorial optimization problems. We then present several general forms that can be used to select previously solved problems that might aid in the solution of the current problem, and several different techniques for actually using this information to derive a solution for the current problem. We describe both fixed forms and forms that have the ability to change as we solve more problems. A simple example for the 0-1 knapsack problem is presented.

Key Words: artificial intelligence, combinatorial optimization

Mathematical modeling consists of three major steps: model creation, solution, and application. Many early models were relatively easy to create, so much of the effort was concentrated on the solution process. As people became more familiar with the area, models became more realistic but also more complicated. The model creation process became very difficult, but many recent works in the area of modeling languages and model management systems (MMS) will help to make this process easier. One type of model that has many current applications is integer programming or combinatorial optimization problems (COP). The power to model logical decisions has led to many applications, especially in manufacturing and transportation. Many of these problems are operational decisions, which need to be made much more often than strategic or one-shot decisions. Very similar problems may need to be solved every day, every hour, or even more often. We shall refer to this combination of model type and frequency as repetitive combinatorial optimization problems (RCOP).

If a problem needs to be solved very frequently, we are also going to be very limited in the amount of time available for the entire modeling process. If the problems are related, a modeling language or MMS should allow for relatively easy modification of previous models to create a model for the current problem. In this article, we will concentrate on the second step, solving the model. The choice of our technique also has bearing on the

application of the solution. Most COP models are very difficult to solve, and if the problems are large and need to be solved frequently and quickly, standard mathematical programming solution techniques will be too slow. One idea is to use previous problems' information to help solve the current problem, especially if we are solving a large number of related problems. This idea was first presented in the artificial intelligence (AI) literature as case-based reasoning, but most of the previous applications have been in areas that have much less structure than mathematical programming problems.

Since we are opening a new domain for case-based reasoning and developing a new, general heuristic technique for mathematical models, we start at a very basic level. We start with a detailed discussion of the different ways we could use the information from previous problems to help solve the current problem when we are dealing with RCOPs. We are not attempting to come up with a better heuristic for standard COP problems, since this is a first try at implementing these techniques and well-established domain-specific techniques already exist. Rather, we are trying to define a technique that might work well across a variety of problems, including actual applications that are much more likely to be repetitive in nature. This technique may also provide significant advantages in explanatory power and human interaction over standard mathematical programming techniques. Since people often make decisions based on their past experience, it should be much easier for them to understand the proposed solution algorithm than some complicated numerical solution process. It might be possible to have a person interact with or help guide the case-based reasoning process, since it consists of two relatively simple steps. Also, if users can understand the solution process, they may be better able to verify whether the solution can really be applied, which is always a consideration since a mathematical model cannot consider every detail of the actual problem. This technique also has an advantage over other AI techniques that have been applied to mathematical programming problems. It *requires* significantly less domain-specific knowledge since knowledge is stored explicitly in the previous problems, though if such knowledge is available it can be used to improve the process. This may be especially important for newer, more difficult applications where there is no expert for solving the problem, which is often why a mathematical model is being used for the problem. Finally, this technique has the possibility of changing and learning over time, which most standard mathematical heuristics cannot do.

The following is a detailed outline of the remaining sections of this article. Section 1 presents a formal mathematical description of a repetitive combinatorial optimization problem, which is one possible method for defining a series of related mathematical programming models.

The main discussion of methods for using the additional information that is present when we are solving a series of related problems is presented in Section 2. We start with a brief review of case-based reasoning and a few of the previous applications areas. Following the philosophy behind case-based reasoning, the method for RCOPs is also divided into two steps. The first step is deciding which of the previous problem's information might be most useful for solving the current problem. This may take into account similarities between the two problems and possibly other information such as the solution to the old problem. We describe both fixed methods and methods that can change over time, using genetic algorithms as one possible means for varying the method. We then discuss the possible methods of using the information from the previous problem(s) to solve the current

problem. We define several possible ways of adapting the old problem solution to the current problem, once again describing both fixed forms, and methods that might vary over time. Finally, we discuss several of the problems associated with implementing this type of methodology in Section 4, including how we might form a set of previously solved problems if such a set did not exist at the time of implementation.

Section 4 will present a brief description of some of the other AI and standard solution techniques that have been applied to COPs. Most of these are traditional solution methods; there is very little work in the area of applying information from past problems to the solution of the current problem. Some basic comparisons are given between these methods and the techniques described in this article.

In Secton 5, we present a small example of how these ideas might be applied to one combinatorial optimization problem, the 0-1 knapsack problem. An approximate performance measure is derived for one simple adaptation and selection process. We conclude by listing some of the other COPs to which this technique may be applied and future research areas.

1. Repetitive combinatorial optimization problems

In this section we present the most general form of the repetitive combinatorial optimization problem (RCOP) with which we have chosen to work. We have chosen RCOPs for several reasons. First, we chose not to look at linear programming and convex nonlinear programming (the two other main types of mathematical programming models besides combinatorial optimization) because the state of the art in solving these problems is relatively advanced compared to COPs. Thus, even though the method we will present would probably work well for continuous optimization, this might not present a great contribution to the solution of these types of problems. Also, COPs (which include both integer and mixed-integer programs) allow us to model a much larger range of problems, since many actual applications involve logical or discrete decisions that usually cannot be modeled accurately without using integer variables. The major reason for considering repetitive problems is that in actual practice, a given user of mathematical programming models will often be solving a series of related problems, rather than one single problem. One reason for assuming this is that the use of such models is often very costly (considering not just the solution time but also the costs of model definition and validation) and hence will more likely be used for problems that need to be solved repeatedly rather than one-shot decisions. Another reason that we believe repetitive problems are common is the large interest in the areas of modeling languages and model management systems. These types of systems are useful for defining a series of related models or problems, and several of these have been designed specifically for mathematical programming models. Given these systems, we will not describe the process by which the user defines the related problems but instead concentrate on the solution process.

We start by giving a description of the variables involved and the basic structure of the problem. We then describe some various forms that these problems might actually take.

1.1. Problem definition

First, the set of variable definitions are:

- I the set of indexes of the continuous decision variables (this set may be empty);
- x_i a continuous decision variable, $i \in I$;
- c_i the objective function's coefficient value of x_i, which takes on random values from some probability density f_{c_i};
- u_i upper bound on x_i, which takes on random values from some probability density f_{μ_i} (there may be lower bounds on the variables, but by basic techniques the problem can always be translated into this standard form);
- P_i an indicator whether variable x_i exists in the current problem, it takes on the value of one with some probability p_i and the value zero with probability $1 - p_i$;
- J the set of indexes of the discrete decision variables, we assume that this set is nonempty;
- Y_i a discrete decision variable, $j \in J$;
- d_j the objective function's coefficient value of Y_j, which takes on random values from some probability density f_{d_j};
- Q_j an indicator whether variable Y_j exists in the current problem, it takes on the value of one with some probability p_j and the value zero with probability $1 - p_j$;
- K the maximal set of constraints in the problem;
- a_{kl} the coefficient of the variable $l \in I \cup J$ in the row k of the constraint matrix, it takes on random values from some probability density $f_{a_{kl}}$;
- R_k an indicator whether constraint k exists in the current problem, it takes on the value of one with probability p_k and zero with probability $1 - p_k$;
- b_k the right-hand side value for constraint k, which takes on random values from some probability density f_{b_k}.

None of the probability densities listed above is assumed to be known (unlike the probabilistic combinatorial optimization problem in Bertsimas, 1988) and might be functions of many different things, including the number of problems solved or even previous realizations of the variables.

The general formulation is then

$$\min \sum_{i \in I} P_i c_i x_i + \sum_{j \in J} Q_j d_j Y_j, \tag{1}$$

subject to

$$R_k \left(\sum_{i \in I} P_i a_{ki} x_i + \sum_{j \in J} Q_j a_{kj} Y_j \right) \leq R_k \cdot b_k \qquad \forall k \in K \tag{2}$$

$$0 \leq x_i \leq u_i \qquad \forall i \in I \tag{3}$$

$$Y_j \in \{0, 1\} \qquad \forall j \in J. \tag{4}$$

The overall goal is to solve a sequence of these problems where each problem is a realization of all the different probability distributions listed above. We will denote these problem instances $\mathcal{P}_m$, where m is the index of the problem (the mth problem to be solved). In actual practice, we may have in addition to the current problems, a set of m_0 previously solved problems generated from the same sets of probability distributions.

To summarize, we are interested in solving a series of *deterministic* combinatorial optimization problems, each of which is a *realization* of a general problem class. As long as the general structure of the objective and constraints is known, we don't actually need the property that the parameters are realizations of probability distributions, the probability distributions are just a convenient method for describing related problems. The key property is that we are able to identify the corresponding variables, parameters, and constraints between problem instances.

Note that these problems are very similar to the *probabilistic* combinatorial optimization problems (PCOP) described by Bertsimas (1988), with two major differences. The first is that the PCOP must be solved a priori, and the solution must be kept (or adapted in some fixed manner) for all occurrences of the problem; whereas, we are solving each problem as it occurs. The other major difference is that in the PCOP it is assumed that all of the probability distributions are known exactly, and this information is used to solve the problem, whereas we do not assume we know these distributions and must therefore base our algorithm on just the problem occurrences.

1.2. Specific forms

The model given above is a very general description of a RCOP. In many actual problem applications, some of the above distributions may be constant. Often, costs or profits that are part of the objective function will remain constant over considerable periods of time. Likewise, technical constraint values (represented by a_{kl}) will often remain the same for many problems. The probabilistic combinatorial optimization problems listed in Bertsimas (1988) have both sets of these values constant, the only random variables are the indicators whether a particular variable is present in any problem instance. The most important case, which we will consider in some detail, is the values for R_k.

Before deciding the likely values for R_k, we must consider what constraints in mathematical programs typically represent. In most cases, these constraints represent physical limitations present in the problem being modeled. Some examples of these might include the capacity of a truck in a vehicle routing problem, a department's capacity in a production problem, or the requirement that the solution form a tour in a traveling salesman problem. If we consider these types of constraints, it would be fairly unlikely that R_k would ever take on a value other than one. It is much more likely that a physical constraint might *change* (through the values of a and b) rather than suddenly cease to exist. The only case where a constraint would probably no longer hold is if none of the variables in that constraint were present, which would be represented by having all of the appropriate P and Q indicators taking on values of zero. In this case, the entire left-hand side is zero, so we would set the value of R_k to one so the constraint can be ignored.

In summary, most problems whose constraints map to actual physical limitations will probably have all of the R variables always constant at a value of one. Also, in many problems, some or all of the continuous parameters may have constant values.

The following small mathematical program presents a problem that we will use throughout the rest of this article to explain how the concepts we are discussing can be related to a RCOP:

$$\max_{Y_{1m}Y_{2m}Y_{3m}} Q_{1m}3Y_{1m} + Q_{2m}d_{2m}Y_2 + d_{3m}Y_{3m}, \tag{5}$$

subject to

$$Q_{1m}Y_{1m} + Q_{2m}Y_{2m} + Y_{3m} \leq 2 \tag{6}$$

$$Y_{jm} \in \{0, 1\} \quad \forall j \in J, \tag{7}$$

where

m is the number of the problem being solved;

$p(Q_{1m} = 1) = 0.5$, 0 otherwise;

$p(Q_{2m} = 1) = 0.5$, 0 otherwise;

$f_{d_{2m}}, f_{d_{3m}} \sim N(3, 0.5)$.

2. A case-based approach for mathematical programming

In this section, we present the framework that will be used when developing learning or case-based reasoning heuristics for repetitive combinatorial optimization problems (RCOPs). The two basic steps of case-based reasoning are finding relevant cases and then adapting their solutions, as defined by Riesbeck and Schank (1989). The case-based approach provides several advantages over more standard mathematical programming techniques. The solution process is much closer to how humans solve problems than standard mathematical programming techniques. Therefore, it should be easier to understand how the solution was derived and to modify either the solution or the solution process if necessary. Models are usually imperfect descriptions of the real-life problem they are trying to represent, so an optimal solution to a model may not be the best solution for the problem. Obviously, if an optimal or very specialized heuristic technique already exists for a COP, then the objective function values for the case-based heuristic will probably not be as good. The advantages of the case-based heuristic are the better explanatory power and interaction capabilities, and the ability to develop heuristics for models for which it is very difficult to come up with a standard mathematical search technique, which will probably be true for a large number of real-life applications.

As described in Riesbeck and Schank (1989), most case-based reasoning applications have been in problem domains where there is often significant difficulty in determinng whether two problems are analogous, such as legal reasoning in Ashley (1987), Bain (1986)

Rissland and Ashley (1986), or generating plans of various types (recipes in Hammond, 1989, or football plays in Collins, 1987). The advantage of working in a domain where the problems are well defined, such as combinatorial optimization, is that it is usually not difficult to determine analogies between the problems, especially given the increase in interest in the areas of mathematical modeling languages and model management systems such as Bhargava and Kimbrough (1990), Bhargava and Krishnan (1990), and Fourer (1983). The disadvantage is that the problems are usually much larger and more complicated than the standard domains to which case-based reasoning (and many other artificial intelligence techniques) have been applied. There has been some work in the application of case-based techniques to more structured problems such as Chaturvedi (1993) and Koton (1989), but no formal framework has been proposed for applying case-based reasoning for these types of problems. Because of the differences between these domain types, we present new techniques for the selection and adaptation processes in case-based reasoning. For each of these two steps, we first present the simpler possible forms these steps could take, and then we discuss how some additional techniques can be used to help determine better possible methods for performing these steps.

A solution procedure for a RCOP can be defined as a set of steps (actions) performed on a problem instance to arrive at an answer. One assumption is that the actions taken are a function of at least the problem instance. If the answer is only a function of the current problem instance, then we have what we consider a standard solution procedure. The first possible learning prospect occurs when the steps taken can be a function of more than the current problem instance. The next important step is to look at what other information can be used in the solution procedure.

The most general case would be when each solution step and the answer to problem m are a function of

- The current problem,
- Previous solution steps to the current problem,
- Previous problems,
- Previous problems' solution steps,
- Previous problems' answers,
- m (procedure may vary with time),
- Other sources (such as human input), and
- Random variables

The ultimate goal would be to find, for any given RCOP, which of the eight items listed above should be used and what function of them should be used for each solution step. Unfortunately, determining the best possible function will probably be more difficult than solving the problem in the first place.

We will describe some various ideas and techniques that might make the process described above feasible in a situation where there is a significant limit on computational time or power. It is important to remember as we are discussing these ideas that when implementing them careful thought should be given as to whether the ideas are appropriate for a particular RCOP being studied. The problem of deciding which types of techniques are best for which types of problems is a metaquestion that probably cannot be answered until

some of the techniques have been explored and tested. The other thing to keep in mind is that it may not be possible to consider some of these techniques in isolation on many problems because of interactions in their implementation. We will give some idea of the possible interactions when they are not obvious.

2.1. Choosing the old instances

At some point, if we have solved many problem instances belonging to a particular RCOP, we may not wish to have the solution to the current problem be a function of *all* past instances. There are several possible reasons to limit these functions. First, the computational time involved in computing the function of the past instances might take longer than we have to solve the problem. A second reason is that many of the past instances may not be useful in solving the current problem instance. This brings up one of the major problems associated with this type of system, defining which previous problem instances are most likely to be helpful in the solution of the current problem. If we solved exactly the same problem before, then we could always use that information. The important question is *how do we determine if this problem is similar, in some useful sense, to any previous problem instances and then how to use that information.* We start by addressing which old instances to use. We will give a more detailed discussion on methods for using the information from previous problems in Section 2.2.

2.1.1. Fixed functions for the selection process. In this section we describe some possible methods for choosing which previous problem instance(s) should be used to help determine the solution to the current problem. There are a tremendous number of possible ways to choose these instances, so in this section we describe the basic forms these choices may take. For the purposes of this section, we assume that the method that we use for choosing the instances is determined and fixed before we observe any problem instances. In the next section, we will describe extensions of these methods that would allow the selection method to change as a function of the problem instances. For the remainder of this section we will use the singular, choosing a previous *instance*, but there is no reason that these techniques could not be used repeatedly if more than one instance could be used for the solution process (Section 2.2).

The first possible mapping is to take the current problem instance and look for any previous problem instance that consisted of the same sets of variables—that is, the indicator variables $P_i(\forall i)$ and $Q_j(\forall j)$ being the same for the two problems. In the example problem, this would imply (with h being the previous problem and m being the current problem) $Q_{1h} = Q_{1m}$ and $Q_{2h} = Q_{2m}$. In this way we have a solution value for each variable in the current problem. If the sets I and J are large and we have not solved many problems previously, then such an instance may not exist. In this case, we could take the previous problem that had the largest intersection for the P_i and/or Q_j sets with the current problem. In the example problem, this would be

$$Q_{1h} \times Q_{1m} + Q_{2h} \times Q_{2m}. \tag{8}$$

We might also choose problems for which the intersection of R_k for the old and current problem is the largest, but for most problems these are identical (with value one) for all of the problem instances (the reason for this assumption was described in Section 1.2).

We next consider the other random parameters, the objective function coefficients, the constraint matrix and right-hand-side coefficients and the upper bounds on the variables. Given the nature of these values, they will usually take on continuous rather than logical or discrete values. The most common exception would be a constraint

$$\sum_{j \in J} Q_j a_{kj} Y_j \leq b_k, \tag{9}$$

where the Y_j are the discrete variables, and b_k might take on integral values (such as maximum number of plants to be opened, trucks to be run, and so on). For this case, we operate as if b_k is taking on continuous values. If there are a significant number of coefficients that took on logical (0, 1) values, then we may need to define a discrete measure similar to the ones in the previous paragraph. Returning to the continuous valued coefficients, we can define a continuous function of the differences between a previous problem's coefficients and the current problem's coefficients for the parameters that the two problems have in common. There are many possible choices for these functions, some of the most obvious being the standard mathematical L-norms, such as L_1 = the sum of the absolute differences, L_2 = the sum-squared of the differences, or even L_∞ = the largest absolute difference. In the example problem, we might choose the previous problem m for which the value of

$$|d_{2h} - d_{2m}| + |d_{3h} - d_{3m}| \tag{10}$$

is minimized for the current problem m. Even more specialized functions could be used, such as having a weighted summation of the difference of the constraint matrix variables, in which the weights are a function of the objective function coefficients. Obviously, there are many possibilities, so the final choice will have to be determined considering the RCOP being solved.

Up to this point, we have only considered previous problem selection based on the problem instance. But we may also want to base our choice on the answer to the previous problem or even on the solution steps we chose to solve that problem. The latter case is likely to be specific to a particular RCOP, so we will just discuss basing our choice on the previous problem's answers. The most obvious idea is that we might want to choose a previous problem that had a very good objective function solution if all of the previous problems were not solved to optimality. This might be either an absolute measure or a relative measure of the problem data, depending on the problem being studied. The other possibility is to have the choice be a function of the values of the variables in the solution. We will discuss this specific choice in Section 2.2.

Another very important possibility is that the person designing or using the RCOP might have some knowledge as to the relative importance of certain variables or parameters of the problem. Perhaps certain variables or coefficients are known to be very important for

deriving a good solution to the problem. We might want to place more weight on the "closeness" of these particular values in the selection function. Deciding the relative weights could either be done manually or by using some idea similar to the ideas described in Section 2.1.2.

One additional problem we may face is deciding whether to compare the current problem instance to all previous problem instances or only a subset of them. Obviously, if the number of past instances grows very large, as it might in some operational decisions that are made daily or even more often, at some point comparing to all past instances is going to become expensive relative to the possible returns. There also might be some storage limitation if the problem instances are large. Two possible solutions that could be used separately or in conjunction with one another are (1) to store only a representative subset of the past problem instances and (2) to subdivide the *possible* problem instances into subsets and compare only within a given set. The latter would probably be used to take advantage of some structure in a given domain. For an example using daily cyclicity in the train industry, see Kraay (1993). The former can be done in several possible ways. One simple heuristic would be to keep track of how often a particular instance is used and keep only those instances that are used frequently. This might be the most appropriate method if there were long-term changes or trends in the underlying problem being modeled. It might also be possible to use cluster analysis or other methodologies for grouping very similar instances for which the same or similar solution procedures were used.

We have described just a few of the many possible methods for choosing an instance or set of previous instances from which to use the information when solving the current problem. In many RCOPs, several of these functions may need to be combined to form a good selection procedure. So far, all of the methods we have discussed assume that we have fixed the functions a priori, before the realization of any of the problem instances. In general, it might be possible to have the selection function and adaptation process also be a function of the realizations of the problems. In Figure 1 we present a continuum representing the possibilities for having the selection and adaptation processes be functions of the problem instances. The dependence on instances listed toward the right of the continuum might also take on many forms. This could range from a direct functional relationship with values of the problem instances to an indirect relationship such as how well the function has performed on the chosen instances. For most problems, functions further toward the left would be easier to implement, whereas those further to the right have the potential for producing higher-quality solutions. In the following section, we will describe one possible method for implementing a selection function that is not fixed a priori.

2.1.2. Changing functions for the selection process. There are three major reasons why it may be desirable for the methods of comparing instances to change over time. The first reason is that for many RCOPs we may not be able to determine the best possible method of comparison a priori, especially if the selection function must combine several of the functions listed in the previous section or if we do not have enough knowledge about the probability distributions of the RCOP. The second reason is that as the number of previously solved instances increases, we may want to consider a different selection function. For example, if we had a large number of similar instances, we might want to choose the one with the best objective function value if the previously solved instances were not solved

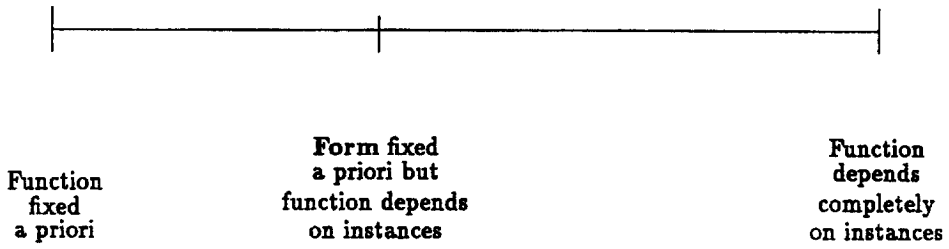

Figure 1. Continuum of possible selection and adaptation functions.

to optimality. The third possible reason is that the RCOP itself may be changing over time (the underlying probability distributions being time-dependent), which would almost certainly require the comparison method to change over time. We will concentate our discussion on the former cases, though many of the techniques we describe will also be applicable to the latter.

For large-scale problems, it will probably not be possible to choose a comparison methodology that is a function of all of the different parameters of the problem. Therefore, we are looking for a function that will perform well over a number of different problems that are realizations of the probability distributions in the case of the RCOP. We are looking for a selection method that will perform well in an expected value sense, not just for a particular instance of the problem. Each time the method is applied we are creating a realization of the method, which is going to depend on the previous problems and the current problem. Another point we have not previously discussed is that we will be applying this procedure multiple times for a number of new, related problems. After we have solved a given problem, we will usually choose to add that problem to the set of previously solved problems. An exception to this might be the case where we were unable to come up with a satisfactory solution using the adaptation process. Therefore, the future applications of the selection process will be a function of the current application of the selection (and adaptation) process. These two properties may have major implications on how we choose to design a changing selection methodology.

Since our problem domain consists of well-structured problems (mathematical programming models), we choose to limit our discussion of solution methodologies to mathematical comparison functions of the problem and variable values. For domains where the problems are not as well structured, other methods such as rule generalization in Riesbeck and Schank (1989) or explanatory coherence in Thagard (1989) might be more appropriate, but an attempt to apply these methods to the RCOP domain is beyond the scope of this article since these would require significant modification to be applicable to the domain we have chosen. The remainder of this section will be devoted to describing some possible methods for finding good selection functions.

The problem of finding the best or better functions to use in the selection process is a more general problem than choosing a previous problem instance, which is an *application* of the selection function. Finding a good selection function can be viewed as an optimization problem (with the special features noted above), and hence there are many possible methods for performing this task. The number of available choices decreases significantly

when we consider a few features of this problem. The first, which we mentioned above, is that unless the form is a function of all the variables and coefficients, then an application of the function is only a single sample point from the *distribution of the applications* of the function. This means that if we were to attempt to optimize (in the mathematical programming sense) this function, we would have to solve a stochastic program over all of the possible distributions of previous instances and current problems and require a closed form solution for the process of using the information for the current problem. Both of these restrictions make this impractical for any but the most trivial RCOPs. So we are limited to some heuristic methods (in the sense that they do not guarantee an optimal solution) that might give us a very good function for selecting instances.

The simplest heuristic is simple simulation or other random search methods. Even if we were to limit our search to certain functional forms with unknown numerical (weighting) values, it might take too long to find the best function using simulation, since each step would involve generating instances, performing the comparison process, and using the information to form a solution. Even more limiting, random search would require that we know all of the probability distributions a priori (at least approximately), an assumption we don't otherwise need to make. Neural networks are another possible method for defining this function. One of the major requirements is that we have a set of training problems for which we know the correct solution. We would have to use the set of previous problems as the training problems, which might lead to some difficulty if these were not enough to initialize the network, plus we may not be able to calculate the best selection function for a given problem instance. One possible exception is if the set of previously solved problems were permanently fixed, and the selection function simply had to choose which problem from this set to use. This model would be extremely limiting, not being able to learn as we gain more problems and hence will not be considered in this work. If there is more research in the area of adaptive neural networks—that is, having the weights in the net *and* the number of alternatives constantly changing as the network is used, then this might become a reasonable alternative for choosing a selection function. The last possible method we will consider is genetic algorithms, which we discuss in some detail as this seems to have the most promise for finding good selection functions.

Genetic algorithms (GA) are a mathematical search technique based on the biological model of the "survival of the fittest." Strings of numbers represent genes, and multiple generations are tested and "bred" to find which are the best suited for a particular objective function. We now look at a few of the properties of GAs and why these properties make them especially well suited for the problem of finding a good selection function (for further references and descriptions of genetic algorithms, see Section 4.1). The most important property of GAs that makes them appropriate for choosing a good selection function is that they work only with objective function information and not derivatives or other forms of knowledge. This is important since it would be very difficult to derive a mathematical model of the selection and adaptation process (as described above). Also, GAs usually lead to a global rather than local optimum, which may be important since we may not be able to determine beforehand whether any local optima, other than the global optimum, may even exist. Second, GAs may often provide much better interim performance than more deterministic methods (we will discuss this further below). Finally, GAs have been found to be fairly robust, working well across a large range of problems. This

last observation will be severely tested given the use to which we will be putting the GA methodology. We now give one example of how GAs could be used to help determine a good selection function for choosing old instances.

As discussed under the area of fixed functions, there are several possible numerical measures that might be appropriate for choosing old instances. Assume that we are using the example problem from Section 1.2 and considering using the two functions given before, equations (8) and (10). Since one of these is a maximization and one a minimization, let us redefine equation (10) as

$$10 - (|d_{2h} - d_{2m}| + |d_{3h} - d_{3m}|) \tag{11}$$

(ten was chosen as an arbitrary value for which the function would probably not take on negative values). Then we could consider as a possible combined selection function

$$\alpha \cdot (Q_{1h} \cdot Q_{1m} + Q_{2h} \cdot Q_{2m}) + (1 - \alpha) \cdot [10 - (|d_{2h} - d_{2m}| + |d_{3h} - d_{3m}|)], \tag{12}$$

where α takes on a value zero and one. In general, there could be more than one α—for example, we could weigh each variable or parameter separately if we had some reason to believe that they might have different degrees of importance in the problem. But for the simple illustration of this section, we will include only one weighting parameter. So the problem for the genetic algorithm (or any method we might choose) is to determine the value of α for which the selection function and some adaptation process have the best chance of giving a good solution for the new problem instance. In this simple case, we could encode the parameter α as a binary string, preferably with a gray code where adjacent values change only by one bit. A string evaluation consists of the following operations. First, we decode the binary string to an α value between zero and one. Next, we calculate the value of the selection function, in this case equation (12), for the current instance m with each previous instance h. We then select the old instance h, which maximizes the selection function, and apply whatever adaptation process we have chosen to come up with a solution to problem m. Finally, we give the string a fitness measure that is some function of the heuristic solution to problem m. We now discuss several important problems and properties of this procedure.

The first point is that an evaluation of a particular string involves choosing a previous instance, based on the current problem and the selection function. Since the set of previous problems and the current problem are both just possible occurrences of a larger set of problems, this evaluation is equivalent to a single point estimate. So to obtain an accurate evaluation of a given string, we would actually need to evaluate it many times (either for many new problems or for sets of different old problems if these were available). There are studies on how GAs perform when the functions are noisy (e.g., Dejong, 1975; Greffenstette and Fitzpatrick, 1985), but to the best of our knowledge no one has tested a GA where the evaluation is simply a single point chosen out of a large sample space. Specific research into the design of GAs for this type of problem will need to be done before a well-designed system can be built.

The second point we must consider is whether we are performing the string evaluations as we are actually solving the RCOP, or separately. The advantage of the latter is that when

we are ready to use the selection function, we will probably have a better function. Unfortunately, it may be very expensive to perform these additional evaluations. De Jong (1975) found that through careful selection of problem parameters, one could produce reasonably good results for the case where you actually use the function evaluations. This is referred to as on-line performance. In addition, evaluating the strings as we are using the selection function implies that the GA will be able to adjust if the underlying RCOP distributions are time dependent. We probably will not see the standard convergence that would be achieved in GAs where the evaluation functions are not time dependent. There is one other very important thing to consider when implementing the GA evaluation. If we are evaluating the GA strings as we are actually solving the problem, then after we have chosen an old instance and determined a solution to the new problem, we will probably want to add the new problem to the list of previously solved problems. This would imply that future function evaluations will be dependent on previous function evaluations through the possible selection of a previous instance that was itself the selection function chosen by another string, a case that has never been considered before in the GA literature.

In this section we discussed how the selection function could change over time, using genetic algorithms as one particular method that seems to be well suited to guiding this process. The first advantage of this would be to help us determine what is a good selection function, since this will probably be very difficult to determine given just the description of the RCOP, especially if we do not have much information about the probability distributions. The second advantage is to allow the function to change if the RCOP distribution functions are time dependent. We now turn to the question of what to do with the information associated with the old problem instance or instances we have selected.

2.2. *Using the information*

In this section, we describe possible methods for using the information from a previous problem instance or instances that we have chosen using the techniques described in Section 2.1. We follow the same format as before, first describing solution procedures that are chosen to be of some predetermined form and then describing solution procedures that may vary with the problem instances.

2.2.1. Fixed forms for the adaptation process. Many of the ways of using information discussed here will be directly related to the functions used to select the previous instances whose information we are going to use. Descriptions of the appropriate selection functions are located in Section 2.1. We will attempt to start with the simplest choices and progress to those that are more sophisticated (and will probably be more successful).

The first case is if we have selected a previous instance whose answer is of the right form for the current problem—that is, it has all of the same variables. In this case, we could simply propose this as the answer to the current problem. Unfortunately, unless the solution space were very small or the old problem was very similar, this would probably not be a very good answer for the current problem (it might even be infeasible if the constraint sets were not the same!). Also, as discussed previously, it is unlikely for many RCOPs that there would be a problem with exactly the same set of variables, so some form of

adaptation is probably going to be necessary to get a solution (or a good solution) for the current problem. We will divide the discussion of the remaining adaptation techniques into two basic classes. The first class consists of adaptation techniques that leave the variables unchanged that the problems have in common, and just set the values for variables that are in the new problem and not in the chosen old instance. The second class of adaptation techniques is those that can change the values of all of the variables, depending on the problem instances.

If we do not have values for all of the variables, one possibility is to actually solve the remaining mathematical program for these variables. Even though we are working with COPs, if there were only a handful of integer variables for which we needed values, in the worst case we could probably do complete enumeration. If we had the values for all of the integer variables, then it would usually be easy to solve for the continuous variables using a standard mathematical programming technique. Many mixed-integer programming problems will have the property that the problem is convex in the continuous variables after we have fixed the values of the integer variables. If we could not actually solve for the remaining variables, we may be able to use some type of heuristic, either general or domain specific, to set the values of the remaining variables. Another possibility would be to use the information from more than one previous problem instance. For example, we could take a second instance, perhaps the one for which the selection function had the second highest value, and use this solution to set any variable values that did not exist in the first instance chosen. In this case we might also want to have a different selection function for choosing the second closest instance, perhaps concentrating on the undecided variables but also considering the fact that we might want the decided variables to be relatively close to their corresponding values in the solution to the second chosen problem instance. The relative importance of these might be found with a changing form as described in Section 2.2.2. Also, when using more than one previous instance, we might need to change some of the values from the first previous instance.

For many reasons, we may not be able to or want to keep the same variable values for the current problem as we had in the previous problem instance(s). If the old problem instance and the current problem have significantly different values for some of the constraints or right-hand-side coefficients, or if there are additional constraints or variables in the new problem instance, then the old solution might be infeasible for the current problem. Even if it were feasible, if the objective or constraint matrix coefficients are sufficiently different, then the solution may no longer be a good solution (the objective function value being significantly below the best value for the current problem). There is also the case that we described above, where we choose to use multiple old instances to find values for all of the variables, but there are conflicts for certain variable values that lead to problem infeasibility. We now describe some techniques that might allow us to adapt these old solution values to get feasible or better solutions to the current problem.

The easiest type of change to consider is the situation in which there is a significant difference between one or more of the old and new problem's objective function coefficients. If these changes affected continuous variables, we could fix the integer variables' values, and either calculate a new solution using a dual simplex algorithm or at least gain information by considering the dual prices and attempting to adjust the values of the continuous variables in the correct direction. For instance, we could use a greedy heuristic

where we start with the largest objective function coefficient difference and attempt to change the value of the corresponding continuous variable as far as possible in the direction of a better solution. We might also try a similar procedure for a change in a binary decision variable's objective value, except we would just change it to the opposite value (0 to 1 or vice versa) to try to obtain a new feasible solution. Many other types of local search heuristics, as appropriate to the RCOP, could also be used with the additional information of the probable direction of improvement. The more difficult case might be if the constraint or right-hand-side coefficients have changed. If the solution is feasible, then we could try to improve the solution as just discussed. If the solution is now infeasible, we first might try some form of local search to regain feasibility (such as dual simplex). This may be very difficult if logical decision variables are present in the violated constraint(s). For this case, we may consider checking to see if there is another previous problem instances that is also similar to the current problem and then either use that solution directly (if it is feasible and gives a good solution) or as an indicator as to what the integer (and perhaps continuous) variable values should be to give a feasible solution. In fact, we might even want to choose this previous problem instance as the one that has the property of having the most similarity between its constraint set and the violated constraints in the current problem. One could continue to invent better adaptation procedures both for general classes of RCOPs and for specific ones, so we will defer this discussion until several of these techniques are tried on some specific problems.

2.2.2. Changing forms for the adaptation process. Given the variety of possibilities discussed in the previous section on fixed-form adapation, it would be very difficult to list all of the possible combinations of techniques that might be combined to form adaptation techniques that change over time. So we will just discuss the general form of these types of techniques.

The problem of determining the *best* method to adapt the solution of old problem instance(s) has the same difficulty as we discussed under variable form selection functions—namely, it is a very complicated stochastic problem so we can usually only get point estimates. In addition, the point estimates are expensive to calculate and can be found a priori only if we know all of the distributions. If we store solutions as we find them, then subsequent adaptations will be functions of the current adaptation and the selection functions. Unlike the selection function, it would be hard to imagine a case where we could take the weighted sum of two or more adaptation functions, but it might be possible if they could be applied sequentially. In this case we might implement a genetic algorithm similar to the one discussed in Section 2.1. If we were just selecting from a set of adaptation functions, then it might be more natural to choose the best adaptation procedure using a neural network. An example of choosing a heuristic for a problem using neural networks is presented by Nygard, Juell, and Radaba (1990). We could use this same type of scheme except the heuristic would be the adaptation process. The disadvantage of this scheme is that we would have to evaluate each of the adaptation schemes on each problem instance to develop a set of training problems. With the genetic algorithm approach we would evaluate only one function but might require more problem instances to achieve convergence. The best scheme would probably be a function of the number of instances available and how much the instances change over time.

Up to this point, we have mostly considered the selection function and adaptation process as separate steps, but in actuality they may be tightly linked. This means we might get totally different answers if we find the best selection function for a fixed adaptation process and the best adaptation process for a fixed selection function, or if we *simultaneously* find the best selection function and adaptation process. Fortunately, through careful definition of the string structure in GAs, it should be possible to have both of these searches occur simultaneously. The only remaining question is whether it will take too many generations (that is, too many strings and problems) before we are able to find the best heuristic. It might also be possible to combine neural networks to form a complete search process if that was the methodology chosen.

3. Implementation considerations

In this section we first describe some of the features that may make a problem amenable to the types of learning heuristics described in previous sections. We will also discuss what steps we might take if we do not have a set of m_0 previously solved problems when we want to start the comparison and adaptation processes or if the answers to those problem instances are not necessarily good solutions.

One property that would lead to an improvement in the learning process would be having the optimal solution space (or good solution space) be a relatively smooth function of the values of the inputs of the instances. For example, in a linear program the solution will be a linear function of the right-hand-side coefficients and the objective function coefficients over a given range and a smooth function of the matrix coefficients until a new variable enters the basis. Unfortunately, having to make logical decisions in the problem will probably imply that there are more discontinuities. Even a problem with a large number of logical decisions could still be relatively smooth if these jumps are close to regular (in the sense that in a regression, the sum-squared of the errors would be relatively small). Next we will look at another property of the solutions that may improve the chance of the learning heuristic working well.

We start by assuming that there is some probability distribution on the set of possible instances. We will consider only the set of instances that have solutions. If necessary, we can normalize the probability distribution so it only covers the instances with solutions. The other simplifying assumption we make is that each instance has only one optimal solution (or that there is some known probability distribution on the multiple optimal solutions). Given these assumptions, we have a probability distribution on the optimal solutions for the possible instances. If our past instances are not solved to optimality, we can use the same argument substituting good solutions. Given this distribution, there are two possible cases to consider.

The first possibility is that the distribution of the solution is very nonuniform. In this case it would probably take fewer previous instances to increase the probability that the optimal (or good) solution to the current instance has already been found in a previous instance. The only problem in this case would be to determine which of the solutions is the best one for this instance. If the problem/solution mappings were relatively smooth (as described above), this should not be too difficult to determine.

Unfortunately, for many types of problems, almost every optimal solution will be optimal for only one instance. This is definitely the case if the descriptions of the instances have different dimensions (or number of dimensions). In this case, what we would like to find is some feature of the answers that is nonuniform. For instance, the optimal route might be different for every instance of a set of traveling salesman problems, but there might be an arc or set of arcs that are in a large subset of the answers (and probably another set of arcs that may appear in very few or only one). This implies we may be able to design a learning scheme based on the arcs of the problem, realizing we may have to modify the solutions to get a feasible answer for the given instance. This nonuniform feature may enable us to identify the key to the mapping from old instances and answers to the answer for the current instance.

When we are first implementing this case-based reasoning approach, and we have not solved this type of problem before (not as an RCOP but as a set of separate problems), then we may not have a set of previous instances to work with. This may be the case for many real-world COPs, especially if the model for the problem is relatively new. One possible method is to use some other type of solution procedure (either exact or approximate) to solve the first few problems and then to use this other method and the learning method described in this work simultaneously, until a significant number of problems have been solved. One possible measure for the number of previous problems needed might be when the learning heuristic is performing almost as well or better than the other technique, depending on how expensive it is to perform the more standard solution process. Unfortunately, there is no other easy way around this problem, and so exploration into good standard solution procedures may still be necessary. A good standard solution technique might also be important if there is a significant amount of off-line computing time available that could be used to attempt to improve the solution found by the learning heuristic for use as a previous instance for future problems. For example, a given model may need to be solved in less than an hour, but we could look for improved solutions using an additional computer before it is used as a previous instance, which might not occur for a relatively long period of time, especially if there is any cyclicity in the underlying problems.

The above discussion leads to the question of how the solution is affected if the previously solved problems were not solved optimally. One problem may be that as we use the learning heuristic over a period of time, the solutions may degenerate or we may find the answers converging to a local optimum. If there is no off-line computing time available to allow us to try to improve previous solutions, then it will probably be even more important to allow for a changing solution form (as with GAs). If the changing form allowed random factors, this might allow us to change strategies occasionally, thus possibly avoiding the local convergence problems. This is a very important question that we must consider more for any actual RCOP implementation.

Both of the previous two problems—not having enough old instances and nonoptimal previous solutions—could possibly be reduced through the incorporation of expert knowledge or including a human in the overall solution process. Even if there is a totally new model for a problem, usually the problem was solved by a human before the model was developed. Though the previous solutions may not have been particularly good, they could provide an initial set of instances. It may also be possible for the person who solved the problem in real time to significantly improve on his solution given the benefit of hindsight. This

same hindsight might also help to improve problem solutions frm the learning heuristic. If a problem solution was implemented and did not work very well, a human expert can often diagnose what was wrong with that particular solution. In this case, the solution could be modified before being added to the set of previous problem instances. If there seemed to be a general deterioration or problem with the learning process, the user might be able to suggest new factors to be considered in the selection and adaptation processes.

In Section 5, we illustrate some of the principles described in this chapter for a particular RCOP, the 0-1 knapsack problem. Extensive numerical results will be presented in Kraay and Harker for three of the more standard combinatorial optimization problems, the knapsack, traveling salesman, and simple plant location problems. These problems allow us to test several of our ideas and allow us to compare the efficiency of our algorithm since optimal solution algorithms exist for these problems. We will implement a very basic version of the heuristic that uses as little domain-specific information as possible. This will help us to determine how the algorithm may perform on a variety of problems. If we were to specialize the heuristic for a particular problem at some point in the future, then the heuristic performance should only improve.

4. Comparison with other techniques

In this section we give a description of some of the other artificial intelligence and related techniques that might be used for the RCOPs described in Section 1. There are many more possible techniques than we can describe here. For a general description as to how artificial intelligence and integer programming can be linked, see Glover (1986).

We now give a brief description of three different techniques: genetic algorithms, neural networks, and expert or rule-based systems. We will start by giving a very basic description of each technique and then describe how they might be applied to the RCOPs. We also discuss some of the advantages and disadvantages of the techniques, as compared to the learning (case-based reasoning) approach described previously in this article. It is important to note that we are comparing these methodologies with the selection and adaptation idea, *not* which technique is best for coming up with the best selection and adaptation functions (this was discussed in the appropriate sections above).

Finally, one other combination approach—target analysis and TABU search—also uses information from previous problems to guide the search for an answer to the current problem as described in Glover and Laguna (1991) and Glover (1989, 1990). Target analysis invests more time initially finding very good solution and then uses these solutions as a basis for analyzing and improving the strategies applied by the current solution method. Another example of target analysis combined with a branch-and-bound solution technique is presented in Glover, Klingman, and Phillips (1989). Target analysis could be viewed in the context of this article as a more advanced method of using previous solutions to modify the solution or adaptation procedure than those suggested in Section 2.2.2. Future research into possible ties using the selection techniques presented in this article and target analysis for solution adaptation might provide even better solution procedures.

4.1. Genetic algorithms

The first of the three approaches that we describe is the work in genetic algorithms (GAs). Genetic algorithms are search procedures that are based on the ideas of natural selection and genetics. The first GA application was presented by Bagley (1967). The primary description of GAs was presented in Holland's *Adaptation in Natural and Artificial Systems* (1975). We will present a very simple idea of how GAs operate and present some of the references for applications to the area of mathematical programming.

The first step in a GA is to define the problem in terms of string structures. Starting with an initial generation (set of strings), a series of new generations are created. In each generation a new set of strings is created using some random weighted (by some fitness function) selection of strings that are then combined using various functions and that may undergo some random mutations. The major differences between GAs and more standard optimization procedures are that GAs work from a set of strings (points), have probabilistic transition rules, and use only objective (payoff) information and not derivatives or other knowledge.

A complete survey of the work in GAs is presented by Goldberg (1989). We will just point out a few of the applications GAs related to mathematical programming to which GAs have been applied. Work on applying GAs to the traveling salesman problem is presented in Goldberg and Lingle (1985), Greffenstette et al. (1985), and Wetzel (n.d.). Ideas for solving bin-packing and graph-coloring problems and job-shop scheduling problems are presented by Davis (1985a) and David (1985b). The blind knapsack problem was considered in Goldberg and Smith (1986). These are just a small sample of the problems to which genetic algorithms have been applied.

In Section 2.1.2, we described how GAs could be used to help aid in determining the best selection and adaptation functions, so in the remainder of this section we just consider the case of using GAs directly to solve the problem. We will not discuss the difficulty of modeling continuous variables with GAs, which could be a serious problem if we had a large, mixed-integer problem, but rather discuss the more general ideas related to the combinatorial optimization.

As described above, GAs use only objective function information in determining the transition between generations. This works very well for unconstrained problems, but there is some difficulty when dealing with constrained optimization. There are many possible methods for dealing with the constraints; we describe three of the most obvious here. The simplest (but perhaps least effective) method for dealing with a constrained problem is to just give an evaluation value of negative infinity (or zero, if scaled fitness values are used) to any string that does not evaluate to a feasible solution. This would probably require a large generation size and mutation rate to ensure that we are left with a large, viable number of strings. We might have a good feasible solution and an infeasible solution very close, thus violating the assumption of having short-ordder strings define important features of the problem. Another possible method would be to form a relaxation of the problem, similar to the Lagrangian relaxation of mathematical programming, where there is some penalty weight for violating a constraint. This might be appropriate for problems where the constraints are relatively soft or there is some possible measurement error in their specification. However, if some or all of the constraints represent physical or logical

requirements of the solution (such as having a "tour" in a traveling salesman problem), then this method may not yield feasible solutions. A third technique would be to define the strings and the crossover and mutation operators such that they would always take feasible strings and convert them to other feasible strings. Since this approach seems the most reasonable for the types of structured problems discussed in this article, this is the technique we will compare with a case-based selection and adaptation approach.

The difference in difficulty involved in designing a genetic algorithm approach instead of a case-based reasoning heuristic will probably depend greatly on the problem class. For most problems, the design of an appropriate selection function should not require a large amount of domain-specific knowledge. We are dealing with related problems where the correspondence in variables is known, so we do not have to consider the variable mapping. The domain-specific difficulty lies in the choice of adaptation processes or, in the case of genetic algorithms, string operators that maintain feasibility. If the problem structure (constraints) are fairly simple, then designing an operator to maintain feasibility may be relatively easy. But if the constraint structure is large or complicated, then this may be a more difficult task than coming up with an adaptation process to gain feasibility. The reason for this is that the adaptation process is not limited to the set of operators dealing with the string representation of the problem, which is necessary for genetic algorithms, but can use other techniques such as mathematical programming or other local search techniques. Unfortunately, deciding which of these techniques would be most effective is very difficult without devising implementation methodologies for a variety of domains.

Finally, the case-based reasoning process might provide more explanation to the user than a GA algorithm. Many people use previous experiences to determine how to solve a new problem; very few people solve problems using random search. For case-based reasoning, we can tell the user which old instance(s) we have chosen and what adaptation process we are using to solve the problem. Since the GA proces uses probabilistic transition rules, it is much more difficult to explain to a person how the technique arrived at the chosen answer.

4.2. Neural networks

One type of learning system, connectionist learning systems (also called *neural networks* or *parallel distributed systems*), has recently received much attention. Neural networks typically consist of many simple neuron-like processing elements called *units* that interact using weighted connections. Each unit has a *state* or *activity level* that is determined by the input received from other units in the network. The more recent networks have layers of hidden units between the input and the output units. These multilayer networks can compute much more complicated functions than the networks that lack hidden units, but the learning is much slower as described in Quinlan (1986). Connectionist learning procedures can be supervised, which require a teacher or test problems to specify the desired output vector, or unsupervised, where an internal model is constructed that captures the regularities in the input vectors. Learning in these procedures consists of readjusting weights on a network via different learning algorithms, such as Boltzmann or backpropagation.

A more complete introduction is provided in McClelland and Rumelhart (1987) and Rumelhart and McClelland (1986), and several related papers are available in Shavlik and Dietterich (1990). One article that is more closely related to our work is the paper by Nygard, Juell, and Kadaba (1990), where one of several various vehicle routing heuristics was chosen as best for a specific problem by a trained neural network. How this type of technique might be used in our current work was discussed in Section 2.2. The major drawback in using neural networks is that it typically takes thousands or even millions of training problems for an actual problem of realistic size. As more work is done in both neural networks and case-based reasoning, there may be more possibilities in combining these two powerful but different techniques.

4.3. Rule-based and expert systems

We include a brief discussion of rule-based and expert systems for the sake of completeness, as there have been some attempts to use these methods for well-structured domains, and these types of approaches might seem attractive given that we are solving a series of related problems. An introduction and overview of expert systems is presented by Waterman (1986) and many similar texts.

Despite the relative frequency with which expert systems are now appearing, there are still several problems with trying to use them for the types of mathematical programming problems described in this article. One problem is that for many of these difficult problems, there is no human expert capable of solving the problem. In fact, this is often one of the reasons the problem is being modeled as a mathematical program. In this regard, expert systems require much more domain-specific knowledge than the learning approach. Another possible problem is that expert and particularly rule-based system have difficulty handling constraints containing continuous variables. These are sometimes approximated by discretizing the continuous variables, but for large problems this will require a tremendous number of values, and the relatively inefficient (compared to mathematical programming algorithms) search algorithms of expert systems may require too much time to find a reasonable answer. One additional problem relates to the aspect of learning. Earlier in this article, we describe how we may be able to add to our knowledge base each time a problem is solved. This might correspond to adding rules to a rule-based system, but to perform this activity often might lead to large costs to maintain consistency and to deal with the other problems of maintaining a large expert system.

Expert systems' approaches do offer advantages for general problem solving. One of the major advantages is often referred to as explanation facility—that is, an expert system can explain to the user how it came to the answer it did and why it implemented certain steps along the way. This is more information than is often available from more standard optimization methods. The method described in this article falls somewhere between the two approaches, since it can provide information such as which previous instance is being used for adaptation. The adaptation process may be more difficult to explain than a set of rules, depending on the problem domain. Also, it is much easier to design a full decision support system using an expert system than an optimization code, though this may be partially offset through the advent of model management systems. As with the explanation facility, the

learning technique would probably be toward the middle, allowing the user to interact with the system in various ways. One possibility would be to have the selection function present the user with a subset of the old instances to choose from, along with the selection function measures, and allow the user to choose on which old problem to base the solution. It may also be possible to allow the user to help guide the adaptation process, depending on the domain and the degree of difficulty involved in this process. In conclusion, it seems that a good implementation of the case-based approach described in this article could have *some* of the important user capabilities that are present in expert systems.

5. The repetitive knapsack problem

One of the simplest integer programming problems is the zero-one knapsack problem (we will omit the "zero-one" when referring to the problem for the remainder of the section). Nemhauser and Wolsey (1988) contain some discussion on the problem, and Dudzinski and Walukiewicz (1987) provide a review of exact methods for the knapsack problem and its generalizations. The mathematical programming formulation is (using one of the standard mathematical notations for this problem, which does not exactly correspond to the RCOP notation in Section 1, most notably we use x instead of Y as a binary decision variable

$$\text{maximize} \quad \Sigma_{i=1}^{n} c_i x_i$$

subject to

$$\Sigma_{i=1}^{n} w_i x_i \leq b$$
$$x_i \in \{0, 1\} \quad \forall i, \tag{13}$$

where

$n \equiv$ number of items to choose from,
$x_i \equiv$ whether item i is chosen,
$c_i \equiv$ the value of item i,
$w_i \equiv$ the weight of item i,
$b \equiv$ the capacity of the knapsack.

In this section we describe how we are generating related random knapsack problems and two possible ways to apply the learning technique discussed in the previous section. This analysis is an attempt to determine some of the basic properties of the heuristic when applied to this simple problem. The learning heuristic we are using is very general and will probably require both more of the types of structures found in real-life problems than randomly generated theoretical problems and specialized domain knowledge before it can come close to the solution values given by a standard mathematical programming technique.

It is slightly difficult to define what types of random knapsack problems would belong to the same group (as defined in Section 1). We start with a relatively straightforward case, which has the advantage of being generalizable to many other types of integer programming problems. We will start by generating a large set of items, N, containing items x_1 to x_n having objective function values c_1 to c_n and weights w_1 to w_n. A given instance m will consist of a subset of items $I_m \subseteq N$, with each item $x_i \in N$ having a probability $p \in (0, 1)$ of being in I_m. So all of the coefficients, including the size of the knapsack, b, are unchanged for different problem instances; the only random variables are which items are available for a particular problem.

5.1. Fixed-form selection and adaptation

The next step is to define how we can apply the learning heuristic to these randomly generated knapsack instances. We start by assuming that we hve a set I_{m_0} of instances for which we have obtained and stored the optimal solution. For the new instance I_m with $m > m_0$, find the previous instance $I_j \in I_{m_0}$, which maximizes the function

$$|I_m \cap I_j| - \delta|I_j|, \tag{14}$$

where $\delta \in (0, 1/n)$, is a small constant; if these values are the same for two different instances, choose one arbitrarily (for example, choose the most recent instance). This is the selection function described in Section 2.1. Let Opt_j be the set of items that were chosen as the optimal solution to instance j. We next take the intersection of Opt_j and I_m forming X_{mj}. if X_{mj} is not empty, then we include all of the items in X_{mj} as part of the heuristic solution to instance m. Since we assumed that all of the knapsacks were of the same size, this must form a feasible solution to instance m but could be very poor since the intersection might contain few or perhaps no items. Though there may be several possibilities involving heuristic solution methods to improve the solution, we will describe a multistage learning process that should improve the value of the solution.

The amount of available space in the knapsack after the first step is

$$b - \sum_{n \in X_{mj}} w_n. \tag{15}$$

If this amount is greater than some tolerance level, then we could proceed in the following manner. Find the instance $I_k \in I_{m_0}$ with $k \neq j$ for which the value of equation (14) is the largest. Define X_{mk} as above, and add the items in X_{mk} to those in X_{mj} (eliminating duplications). Continue this process until one of three possibilities occur. The first possibility is that we run out of old instances or reach some predefined limit on the number of iterations. The second possible stopping condition is that we have enough items so that the value in equation (15) is less than the tolerance level, in which case we give those items as the heuristic solution. The third possibility is that we have taken too many items and exceeded the capacity of the knapsack. In this case, we randomly remove items from the

knapsack. In this manner we always determine a feasible solution to the current instance, using almost no domain-specific knowledge about knapsacks. Obviously, we could improve the performance using domain-specific knowledge, but for more difficult RCOPs, this information might be difficult or impossible to obtain. Thus, we feel that this is the first method that should be studied. The performance of this simple heuristic will be analyzed in the following section.

5.2. *Theoretical analysis*

In this section, a slightly simplified version of the algorithm is used to derive several analytical results. We start with the same basic idea—that we have N as the set of total items, having values c_1 to c_n and weights w_1 to w_n all independently generated from some distributions (say, $U(0, 1)$). We let Q_{im} be an (0, 1) indicator random variable, whether item i is in instance m, with $P(Q_{im} = 1) = p > 0 \ \forall i, m$. Let $b = 1$ for all instances, and define x_{im} as the decision variable for item i in knapsack problem m. The problem instance m can be defined as

$$V_m = \max \ \Sigma_{i=1}^{n} c_i Q_{im} x_{im},$$

subject to

$$\Sigma_{i=1}^{n} w_i Q_{im} x_{im} \leq b$$
$$x_{im} \in \{0, 1\} \quad \forall i. \tag{16}$$

We assume that we have previously solved m_0 instances are are currently solving instance $k = m_0 + 1$. We start by having only one iteration of the learning process—that is, we just take the instance with the most items in common. The expected performance of the heuristic can then be measured by

$$\frac{E[\Sigma_{i=1}^{n} c_i x_{iJ} Q_{iJ} Q_{ik}]}{E[V_k]}, \tag{17}$$

where

$$J = \arg_{j=1,\ldots,m_0} \max \left\{ \sum_{i=1}^{n} (Q_{ik} Q_{ij} - \delta Q_{ij}) \right\} \tag{18}$$

and $\delta \in (0, 1/n)$ is a small constant.

The following is a description of one method for determinng an approximate formula for (17). The distribution for the number of items in instance k is a simple binomial. An asymptotic value for V_k as a function of the number of items in instance k was derived by Frieze and Clarke (1984)—specifically, that the value approaches $\sqrt{2s/3}$ where s is the

number of items available for the knapsack. This same procedure can be used as an approximation for $\Sigma_{i=1}^{n} c_i x_{iJ}$ since the items are independently and identically distributed random variables. The approximate value would be

$$\sqrt{\frac{2(\Sigma_{i=1}^{n} Q_{iJ})}{3}}. \tag{19}$$

Given these assumptions, we can form an approximate formula for (17). Since we cannot find an exact formula for the expected optimal objective function value of the knapsack problem (or other COPs), and the learning heuristics are a function of these previous values, it is unlikely that we will ever be able to determine an exact formula.

Let us define four intermediate variables to help derive the formula. Let

$r \equiv$ number of items in the current instance,
$s \equiv$ number of items in the maximal instance J,
$z \equiv$ number of items in the intersection $Q_{ik}Q_{ij}$,
$d \equiv$ number of old instances with cardinality equal to z,

and let $q = 1 - p$. Then (17) is approximately equal to

$$\frac{\Sigma_{r=1}^{n} A \cdot \binom{n}{r} \cdot p^r \cdot q^{n-r}}{\Sigma_{r=1}^{n} \sqrt{r} \cdot \binom{n}{r} \cdot p^r \cdot q^{n-r}}, \tag{20}$$

where

$$A = \sum_{z=1}^{r} \sum_{s=z}^{n-r+z} \left(A(z, s) \equiv \sum_{d=1}^{k} \left[B(d) \cdot \left(\sqrt{s} \cdot \frac{z}{s} \cdot C(s) \right) \right] \right) \tag{21}$$

$$B(d) = \binom{k}{d} \left[\binom{r}{z} p^z q^{r-z} \right]^d \left[\sum_{j=0}^{z-1} \binom{r}{j} p^j q^{r-j} \right]^{k-d} \tag{22}$$

and

$$C(s) = \left[\sum_{i=s}^{n-r+z} \binom{n-r}{i-z} p^{i-z} q^{n-r-(i-z)} \right]^d - \left[\sum_{i=s+1}^{n-r+z} \binom{n-r}{i-z} p^{i-z} q^{n-r-(i-z)} \right]^d. \tag{23}$$

Equation (20) gives the probability of r items in the current instance times the value function for having r items, $\sqrt{r}$ for the optimal solution and A for the heuristic solution (the $\sqrt{2/3}$ will cancel out of all the equations). In equation (21), we sum over the possible values for the maximal intersection, the number of instances with maximal intersection,

and the value gained from choosing that old instance. The value function is the square root of the number of items in the old instance times the fraction of the old instances in which that item would have been chosen (on average, remembering that the items are i.i.d. random variables). B (in equation (22)) is the conditional probability of having d old instances with the same overlap z, given we have r items in the current instance. Finally, (23) gives the conditional probability that the number of items in instance J is equal to s, given r, z, and d. If we consider J from equation (18) and amount of overlap $z = \Sigma_{i=1}^{n} Q_{ik}Q_{ij}$ is fixed, then among those d instances with the same number of overlapping items z, the function will be maximal for the instance with the fewest items (because we subtract δQ_{ij} and $\delta \in (0, 1/n)$). Thus, equation (23) is the probabilistic equivalent of a minimization function over the number of items in the instance (s).

It is almost impossible to perform most types of sensitivity analysis on equation (20) due to the presence of the same variables both in summations and factorials/exponents. One result that we can obtain is the asymptotic value of equation (20) as we let $k \to \infty$ (assuming that all of the other parameters are fixed). Intuitively, as we let the number of old instances approach infinity, then the chance of finding exactly the same instance (among the old ones) as the one we are currently trying to solve should approach one. This is stated in the following theorem.

Theorem 5.1. *The limit as k goes to infinity of equation (20) is equal to one.*

The proof of the above theorem is given in Kraay (1993).

Unfortunately, the analysis becomes increasingly complicated when we consider the multistage learning described in Section 5.1. After some work, it was discovered that to obtain a good bound for the heuristic value in a reasonable period of time, a very slight modification to the heuristic is necessary. This modification relates to how we remove items from the heuristic solution when we have chosen more items than can fit in the knapsack. The modification is that we will only remove items that were put in during the last stage of the learning procedure. This means that in the two-stage case, only items from the second closest intance can be dropped from the knapsack, since all of the items from the first solution must fit. Given this assumption, we now proceed to describe an approximate lower bound on the two-stage heuristic. The analysis is almost identical for more stages, but the calculation of the value will become increasingly difficult, and the bound will get increasingly farther from the true value, unless we use a more complicated procedure.

We will define a set of temporary variables similar to those used for the single-stage case, with a few minor differences. let

$r \equiv$ number of items in the current instance,
$s_1 \equiv$ number of items in maximal instance J,
$s_2 \equiv$ number of items in second closest instance,
$z_1 \equiv$ number of items in the intersection with the closest instance,
$z_2 \equiv$ number of items in the intersection with the second closest instance,
$d_1 \equiv$ number of old instances with the same cardinality, z_1,
$d_2 \equiv$ number of old instances with the same cardinality, z_2.

We will also define two additional functions for notational convenience:

$$f(a, b, c) = \binom{n-a}{b-c} p^{b-c} q^{n-a-(b-c)}. \tag{24}$$

and

$$g(a, b, c) = \sum_{i=b}^{n-a+c} \binom{n-a}{i-c} p^{i-c} q^{n-a-(i-c)}. \tag{25}$$

Equation (20) is identical to the single-stage case, but in the two-stage case we now have

$$A = \sum_{(z_1=z_2)=1}^{r} \sum_{d_1=2}^{k} B_1 \cdot (C_1 + C_2) + \sum_{z_2=0}^{r} \sum_{z_2=0}^{z_1-1} \sum_{d_2=1}^{k-1} B_2 \cdot C_3 \tag{26}$$

$$B_1 = \binom{k}{d_1} \left[\binom{r}{z_1} p^{z_1} q^{r-z_1} \right]^{d_1} \left[\sum_{j=0}^{z_1-1} \binom{r}{j} p^j q^{r-j} \right]^{k-d_1} \tag{27}$$

$$C_1 = \sum_{(s_1=s_2)=(z_1=z_2)}^{n-r+z_1} E \cdot [(g(r, s_1, z_1)^{d_1} - g(r, s_1 + 1, z_1)^{d_1}) - D] \tag{28}$$

$$D = \sum_{s_2=s_1+1}^{n-r+z_1} d_1 \cdot f(r, s_1, z_1) \cdot [g(r, s_2, z_1)^{d_1-1} - g(r, s_2 + 1, z_1)^{d_1-1}] \tag{29}$$

$$C_2 = \sum_{s_1=z_1}^{n-r+z_1} d_1 \cdot f(r, s_1, z_1) \cdot \left[\sum_{s_2=s_1+1}^{n-r+z_1} E \cdot (g(r, s_2, z_1)^{d_1-1} - g(r, s_2 + 1, z_1)^{d_1-1}) \right] \tag{30}$$

$$B_2 = k \cdot \left[\binom{r}{z_1} p^{z_1} q^{r-z_1} \right] \cdot \binom{k-1}{d_2} \left[\binom{r}{z_2} p^{z_2} q^{r-z_2} \right]^{d_2} \left[\sum_{j=0}^{z_2-1} \binom{r}{j} p^j q^{r-j} \right]^{k-1-d_2} \tag{31}$$

$$C_3 = \sum_{s_1=z_1}^{n-r+z_1} f(r, s_1, z_1) \cdot \left[\sum_{s_2=z_2}^{n-r+z_2} E \cdot (g(r, s_2, z_2)^{d_2} - g(r, s_2 + 1, z_2)^{d_2}) \right] \tag{32}$$

$$E = \sqrt{s_1} \cdot \frac{z_1}{s_1} + \sqrt{s_2} \cdot H \cdot q \cdot \left(1 - \max\left\{\frac{z_1}{s_1}, H\right\}\right) \tag{33}$$

$$H = \min\left\{1 - \frac{z_1}{s_1}, \frac{z_2}{s_2}\right\}. \tag{34}$$

Equation (26) is composed of two major parts. The first part covers the case where the maximal intersection is the same for the closest and second closest old instances ($z_1 = z_2$), the second part is for when they are different (the second must be less than the first, by definition). For the first case, equation (27) gives the probability of having d_1, which must be at least two, instances with the same maximal overlap. There are then two further subcases, whether there are two instances with the same minimal number of items or whether there is just one. In the case with two or more with the same minimal number, the probability times the payoff is given by (28); equation (29) is just the sum of the probabilities for one of the instances having more items. The payoff and probability for when the number of items in the old instances is different is given by equation (30). For the second major case ($z_1 \geq z_2$), equation (31) gives the probability for one instance with intersection z_1 and d_2 old instances with intersection z_2. Equation (32) gives the probability function for the number of items in the two instances, multiplied by the payoff E. For all of the equations (26) through (32), the probabilities are exactly as shown. Unfortunately, it is much harder to find an exact answer for the value of the heuristic solution for the two-stage algorithm.

Equation (33) is an underestimation of the value of the heuristic solution for all values of z_1, z_2, s_1, s_2. The first major reason is that we modify the amount we are sampling from the second instance. If we were to take z_1/s_1 (the actual conditional distribution is hypergeometric) and z_2/s_2 of the items from the second instance, and the sum of these two fractions were greater than one, then the probability that all of these items would fit in the knapsack would be fairly small, especially if s_1 and s_2 are fairly large. Since this value would be small and fairly hard to find, we reduce the number of items we can take in this case. One simple method is to reduce the amount taken from the second instance to $1 - (z_1/s_1)$ (the value given in (34)). The next problem is to find the probability that all of the items will fit, given that we are taking z_1/s_1 and H from the two instances, respectively. The worst case would occur if one of the optimal solutions had only one item (the one with the larger ratio), and the other optimal solution had a very large number of items. In this case, the probability of the items fitting is just $1 - \max\{z_1/s_1, H\}$, for other combinations the probability is larger than this value. This is actually a very rough lower bound, since there should be an additional term that is the conditional probability of a smaller fraction fitting if the H would not fit. Unfortunately, finding even a lower bound on this probability is extremely complicated and hence will not be considered here. The third underestimation in equation (33) comes from multiplying by the value q, which is to account for the case of an item being in both problems. The true value would be less than q, since multiplying by q implies that if an item is in the optimal solution of one problem, it must be in the optimal solution of the other, which is not always the case. Since E is an underestimation of the value gained by taking the two closest instances, and all the probabilities are exact, then the entire term is an approximate lower bound on the ratio of the heuristic to optimal value (approximate because of the use of the square root).

As is the case in the single-stage learning procedure, the only relatively simple analysis is to prove that the asymptotic value when $k \to \infty = 1$. The details of the proof are found in Kraay (1993). As mentioned earlier in the section, we would also like to determine how these values behave as functions of n, k and p, which is very difficult to state analytically due to the presence of discrete summations and those same variables being used as functions raised to powers (except for trivial cases such as $p = 1$). Numerical results for the actual performance under a variety of parameter values will be presented in Kraay and Harker.

6. Conclusion

In this article, we have presented a general framework for applying case-based reasoning to repetitive combinatorial optimization problems (RCOP). This framework allows for a combination of different selection functions and adaption procedures depending on the type of the optimization problem. Many of the more complicated functions and procedures may allow for integration with other artificial intelligence and mathematical programming techniques to form heuristics that may be usable on large, difficult problems.

The knapsack problem presented in this article gives just one simple example of how case-based reasoning can be applied directly to a RCOP. In Part 2 of this article (Kraay and Harker, forthcoming), we given numerical results for the application of these procedures to a number of different combinatorial optimization problems. There is still a large amount of research needed to determine the most efficient method for implementing the ideas discussed in this article, especially for real-world problems where traditional methods may not be capable of solving the problem.

References

Ashley, K.D. (1987). "Distinguishing: A Reasoner's Wedge." In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society* (pp. 737–747). Hillsdale, NJ: Lawrence Erlbaum.

Bagley, J.D. (1967). "The Behavior of Adaptive Systems Which Employ Genetic and Correlation Algorithms," Doctoral dissertation, University of Michigan.

Bain, W.M. (1986). "Case-Based Reasoning: A Computer Model of Subjective Assessment." Doctoral dissertation, Yale University.

Bertsimas, D.J. (1988). "Probabilistic Combinatorial Optimization Problems." Technical Report No. 194, Operations Research Center, Massachusetts Institute of Technology, Boston.

Bhargava, H.K., and S.O. Kimbrough. (1990). "On Embedded Language for Model Management." In *Proceedings of the Twenty-Third Hawaii International Conferences on System Sciences* (pp. 443–452).

Bhargava, H.K., and R. Krishnan. (1990). "A Formal Approach for Model Formulation in a Model Management System." In *Proceedings of the Twenty-Third Hawaii International Conferences on System Sciences* (pp. 453–462).

Chaturvedi, A. (1993). "Acquiring Implicit Knowledge in a Complex Domain." *Expert Systems with Applications* 6, 23–35.

Collins, G.C. (1987). "Plan Creation: Using Strategies as Blueprints." Doctoral disseration, Yale University.

Davis, L. (1985a). "Applying Adaptive Algorithms to Epistatic Domains." In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 162–164).

Davis, L. (1985b). "Job Shop Scheduling with Genetic Algorithms." In *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (pp. 136–140).

De Jong, A.K. (1975). "An Analysis of the Behaviour of a Class of Genetic Adaptive Systems." Doctoral dissertation, University of Michigan.

Dudzinski, K., and S. Walukiewicz. (1987). "Exact Methods for the Knapsack Problem and Its Generalizations." *European Journal of Operational Research* 28, 3–21.

Fourer, R. (1983). "Modeling Languages Versus Matrix Generators for Linear Programming." *ACM Transactions Mathematical Software* 9, 143–183.

Frieze, A.M., and M.R.B. Clarke. (1984). "Approximation Algorithms for the m-Dimensional 0-1 Knapsack Problem: Worst-Case and Probabilistic Analyses." *European Journal of Operational Research* 15, 100–109.

Glover, F. (1986). "Future Paths for Integer Programming and Links to Artificial Intelligence." *Computers and Operations Research* 13, 533–549.

Glover, F. (1989). "Tabu Search: Part I." *ORSA Journal on Computing* 1, 190–206.

Glover, F. (1990). "Tabu Search: Part II." *ORSA Journal on Computing* 2, 4–32.

Glover, F., D. Klingman, and N. Phillips. (1989). "A Network-Related Nuclear Power Plant Model with an Intelligent Branch-and-Bound Solution Approach." *Annals of Operations Research* 21, 317–331.

Glover, F., and M. Laguna. (1991). "Target Analysis to Improve a Tabu Search Method for Machine Scheduling." *Arabian Journal for Science and Engineering* 16, 239–253.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

Goldberg, D.E., and R. Lingle. (1985). "Alleles, Loci, and the Traveling Salesman Problem." In *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (pp. 154–159).

Goldberg, D.E., and R.E. Smith. (1986). "AI Meets OR: Blind Inferential Search with Genetic Algorithms." Paper presented at the ORSA/TIMS Joint National Meeting, Miami.

Grefenstette, J.J., et al. (1985). "Genetic Algorithms for the Traveling Salesman Problem." In *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (pp. 112–120).

Grefenstette, J.J., and J.M. Fitzpatrick. (1985). "Genetic Search with Approximate Function Evaluations." In *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (pp. 112–120).

Hammond, K.J. (1989). *Case-Based Planning: Viewing Planning as a Memory Task. Perspectives in Artificial Intelligence*. Boston: Academic Press.

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

Koton, P. (1989). "SMARTPLAN: A Case-Based Resource Allocation and Scheduling System." *Proceedings of a Workshop on Case-Based Reasoning* (pp. 285–289).

Kraay, D. (1993). "Learning Heuristics for Repetitive Combinatorial Optimization Problems: With an Application in Train Scheduling." Doctoral dissertation, University of Pennsylvania, Philadelphia.

Kraay, D., and P.T. Harker. (Forthcoming). "Case-Based Reasoning for Repetitive Combinatorial Optimization Problems: Part II: Numerical." *Heuristics*.

McClelland, J.L., and D.E. Rumelhart. (1987). *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises*. Cambridge, MA: MIT Press.

Nemhauser, G.L., and L.A. Wolsey. (1988). *Integer and Combinatorial Optimization*. New York: Wiley.

Nygard, K.E., P. Juell, and N. Kadaba. (1990). "Neural Networks for Selecting Vehicle Routing Heuristics." *ORSA Journal on Computing* 2, 353–364.

Quinlan, J.R. (1986). "Induction of Decision Trees." In R.S. Michalski, J.G. Carbonell, and T. Mitchell (Eds.), *Machine Learning* (1, pp. 81–106). Palo Alto: Tioga.

Riesbeck, C., and R. Schank. (1989). *Inside Case-Based Reasoning*. Hillsdale, NJ: Erlbaum.

Rissland, E.L., and K.D. Ashley. (1986). "Hypotheticals as Heuristic Device." In *Proceedings of AAAI-86 (American Association for Artificial Intelligence)*. Los Altos, CA: Morgan Kaufmann.

Rumelhart, D.E., and J.L. McClelland (Eds.). (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge, MA: MIT Press.

Shavlik, J.W., and T.G. Dietterich (Eds.). (1990). *Readings in Machine Learning*. San Mateo, CA: Morgan Kaufmann.

Thagard, P. (1989). "Explanatory Coherence." *Behavioral and Brain Sciences* 12, 435–502.

Waterman, D.A. (1986). *A Guide to Expert Systems*. Reading, MA: Addison-Wesley.

Wetzel, A. (n.d.). "Evaluation of the Effectiveness of Genetic Algorithms in Combinatorial Optimization." Unpublished manuscript, University of Pittsburgh, Pittsburgh, PA.