Exploiting Bounds in Operations Research and Artificial Intelligence[*]

Sharlee Climer and *Weixiong Zhang*

Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO 63130, USA

Abstract. Combinatorial optimization problems are ubiquitous in scientific research, engineering, and even our daily lives. A major research focus in developing combinatorial search algorithms has been on the attainment of efficient methods for deriving tight lower and upper bounds. These bounds restrict the search space of combinatorial optimization problems and facilitate the computation of what might otherwise be intractable problems. In this paper, we survey the history of the use of bounds in both AI and OR. While research has been extensive in both domains, until very recently it has been too narrowly focused and has overlooked great opportunities to exploit bounds. In the past, the focus has been on the relaxations of constraints. We present methods for deriving bounds by tightening constraints, adding or deleting decision variables, and modifying the objective function. Then a formalization of the use of bounds as a two-step procedure is introduced. Finally, we discuss recent developments demonstrating how the use of this framework is conducive for eliciting methods that go beyond search-tree pruning.

1 Introduction

Combinatorial optimization problems are ubiquitous in scientific research, engineering, and even our daily lives. For example, we try to devise the best alignment of a set of DNA or protein sequences, an optimal routing scheme when designing a circuit board, or a shortest plan or schedule when sequencing a set of tasks. It seems that we are constantly trying to maximize profit, performance, production, or some other desirable goal; or to minimize cost, time, energy, or some other valuable resource.

Combinatorial optimization problems are difficult, albeit interesting and satisficing, as there are frequently a very large number of *feasible* solutions that satisfy all the constraints; the challenge lies in searching through this vast solution space and identifying an optimal solution within the given criteria. When the number of solutions is large, intelligent enumeration paradigms exist to look at the solutions that have to be examined in order to guarantee an optimal solution. One such enumeration paradigm is *branch-and-bound* (BnB), which was perhaps first used by Dantzig, Fulkerson, and Johnson [11, 12] for the Traveling Salesman Problem. BnB is in fact a problem solving

[*] This research was supported in part by NDSEG and Olin Fellowships and by NSF grants IIS-0196057 and ITR/EIA-0113618.

paradigm that supports many specific search strategies, such as depth-first search [2, 25, 24, 33], best-first search [2, 25, 24, 33], and iterative deepening [28], as special cases. BnB has also been enhanced, leading to variants such as branch-and-cut. BnB has been found to be exceptionally powerful for nearly all combinatorial optimization problems in various disciplines and areas [2, 25, 24]. Even though most of the difficult optimization problems belong to the category of NP-hard [18], which is a worst-case complexity measure, in practice, a BnB algorithm can indeed very often alleviate computational difficulties and make large and complex problem instances solvable with reasonable amounts of computational resources.

BnB organizes a systematic enumeration of solutions in a search tree, in which a leaf node represents a feasible solution. It typically traverses a tree in a depth-first fashion in order to make it memory efficient and applicable to large problems [38, 35]. At each node of the tree, BnB solves a modified variation of the original problem. When the modified variation gives rise to a feasible solution better than the current *incumbent* solution, it is made the new incumbent. As other solutions of this sort are encountered, the incumbent is updated as needed so as to always retain the best feasible solution found thus far. When the search tree is exhausted, the final incumbent must be an optimal solution.

Most optimization problems have huge search trees which cannot be searched in entirety. The power of BnB comes from two bound functions and the pruning rule that it employs. The first bound function computes an upper bound to the cost of an optimal solution. The incumbent solution provides such a bound. In order to have an immediate upper bound, it is common to find a simple upper bound prior to starting the search using an approximation algorithm. The second bound function estimates, at each node of a search tree, a lower bound on the solution to the modified problem at the node. When the lower bound exceeds the current upper bound, the node, along with the entire subtree beneath it, can be pruned with no compromise of optimality. Therefore, the efficacy of BnB depends on the effectiveness of the lower and upper bound functions that it uses. It is thus of fundamental importance to develop efficient and tight bound functions for BnB. This necessity has been reflected by a long history of research in operations research (OR) and artificial intelligence (AI).

In this paper, we first survey the history of the use of bounds in both AI and OR. While research has been extensive in both domains, until very recently it has been too narrowly focused and has overlooked great opportunities to exploit bounds. In the past, the focus has been on the relaxations of constraints. We discuss methods for deriving bounds by tightening constraints, adding or deleting decision variables, and modifying the objective function. Then a formalization of the use of bounds as a two-step procedure is covered. Finally, we discuss recent developments demonstrating how the use of this framework is conducive for eliciting methods that go beyond search-tree pruning.

2 Previous work

In this section, we survey the historical use of bounds in OR and AI. We use linear programs (LPs) for the problem representation for OR and the STRIPS representation for AI. It is important to note that the concepts presented in this paper are applicable to an even broader range of problems than those represented by LPs or STRIPS. However, we use these models in order to make the discussion concrete.

2.1 Bound functions in operations research

In OR, optimization problems are commonly cast as linear programs (LPs). Without loss of generality, we only consider minimization problems in the ensuing discussion (minimization LPs can be converted to maximization LPs and vice-versa [22]). A general minimization LP can be written in the following form:

$$Z = \min \sum_i c_i x_i \tag{1}$$

$$subject\ to: \quad a\ set\ of\ linear\ constraints \tag{2}$$

where the c_i values are instance-specific coefficients, the set of x_i represents the *decision variables*, and the constraints are linear equalities or inequalities composed of coefficients, decision variables, and possibly some auxiliary variables. A *feasible* solution is one that satisfies all the constraints. The set of all feasible solutions is the *solution space* for the problem. An *optimal* solution is a feasible solution with the least value, as defined by the *objective function* (1).

LPs that have one or more decision variables that must have integral values compose a subset of LPs that are referred to as mixed-integer linear programs (MIPs). MIPs can be used to model many important combinatorial optimization problems. Unfortunately, while LPs without any integrality requirements are relatively easy to solve, many MIPs are NP-hard and are very difficult to solve.

When solving LPs, bounds have been used in different ways. Following is a brief summary of deriving bounds via constraint modification, using bounds in branch-and-cut search, and bounds that evolve from duality theory.

Direct constraint modification Lower bound functions can be obtained by modifying and/or eliminating constraints. We use the Asymmetric Traveling Salesman Problem (ATSP) to illustrate these modifications. The ATSP is the NP-hard problem of finding a minimum-cost Hamiltonian cycle for a set of cities in which the cost from city i to city j is not necessarily equal to the cost from j to i. The ATSP has numerous significant applications and has been extensively studied in the fields of computer science, mathematics, and OR. The ATSP can be defined by the following LP:

$$ATSP(G) = \min \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij} \tag{3}$$

subject to:

$$\sum_{i \in V} x_{ij} = 1, \quad \sum_{j \in V} x_{ij} = 1, \ \forall i, j \in V; \tag{4}$$

$$\sum_{i \in W} \sum_{j \in W} x_{ij} \leq |W| - 1, \ \forall W \subset V, \ W \neq \emptyset; \tag{5}$$

$$x_{ij} \in \{0, 1\}, \ \forall i, j \in V, \tag{6}$$

for directed graph $G = (V, A)$ with vertex set $V = \{1, \cdots, n\}$, arc set $A = \{(i, j) \mid i, j = 1, \cdots, n\}$, and cost matrix $c_{n \times n}$ such that $c_{ij} \geq 0$ and $c_{ii} = \infty$ for all i and j in V. Each decision variable, x_{ij}, corresponds to an arc (i, j) in the graph. The solution space of this problem is the set of all permutations of the cities and contains $(n - 1)!$ distinct complete tours. The objective function (3) demands a tour of minimum cost.

The three types of constraints in (4) to (6) can be directly modified or eliminated to obtain lower bound functions. Constraints (4) require that each city is entered exactly once and departed from exactly once. If these constraints are removed from the consideration of the ATSP, the problem becomes the minimum spanning tree (MST) in a directed graph, called shortest arborescence, which can be solved in $O(n^2)$, where n is the number of cities, using an efficient implementation [3, 34] of the Chu-Liu/Edmonds algorithm [6, 13]. In fact, MST on undirected graphs is one of the most effective lower bound functions for the symmetric TSP (STSP) [20]. (The STSP is a special case of the ATSP, where the cost from city i to city j is equal to the cost from city j to city i.)

Constraints (5) are called *subtour elimination constraints* as they require that no more than one cycle can exist in the solution. Ignoring these constraints leads to the *Assignment Problem* (AP) [31], which is among the most effective lower bound functions for random ATSPs. The AP is the problem of finding a minimum-cost matching on a bipartite graph constructed by including all of the arcs and two nodes for each city, where one node is used for the tail of all its outgoing arcs and one is used for the head of all its incoming arcs. The AP can be solved in $O(n^3)$ time, where n is the number of cities [31].

Constraints (6) require that either an arc (i, j) is traversed (x_{ij} is equal to 1) or is not traversed (x_{ij} is equal to 0). The ATSP can be relaxed by relaxing the integrality requirement of constraints (6). This can be accomplished by allowing x_{ij} to be a real value between 0 and 1 inclusive. This relaxation is the well-known *Held-Karp* relaxation [20, 21], and is solvable in polynomial time. In general, relaxing integrality is commonly used in branch-and-cut search while solving all types of MIPs.

On the other hand, we can produce an upper-bound modification of the ATSP by adding additional constraints. For instance, we could force the inclusion of a set of arcs by adding constraints that their corresponding decision variables are set to one (*i.e.* adding $x_{ij} = 1$ forces the arc (i, j) to be included in all solutions). Setting the values of decision variables reduces the problem size. This method is commonly employed by BnB branching rules, in which the problem is continually subdivided into smaller subproblems as the search tree is traversed [2].

Branch-and-cut *Branch-and-cut* (BnC) is essentially the same as BnB search except that the modified problem that is solved at each node of the search is sometimes tightened by the addition of *cutting planes* [19]. A cutting plane is an added constraint that tightens the modified problem without eliminating any feasible solutions to the original problem. An example of a cutting plane follows. Assume we are given three binary decision variables, x_1, x_2, x_3, a constraint $10x_1 + 16x_2 + 12x_3 \leq 20$, and the integrality relaxation is solved at each node ($0 \leq x_i \leq 1$ is substituted for the binary constraints). It is observed that the following cut could be added to the problem: $x_1 + x_2 + x_3 \leq 1$ without removing any of the solutions to the original problem. However, solutions would be removed from the modified problem, such as $x_1 = 0.5$, $x_2 = 0.25$, and $x_3 = 0.5$. Note that while a cutting plane tightens the relaxation, the problem is still strictly a relaxation. The cutting planes only make it a *tighter* relaxation.

Concorde [1] is a branch-and-cut algorithm designed for solving the STSP. This code has been used to solve STSP instances with as many as 15,112 cities [1]. This success was made possible by the design of a number of clever cutting planes custom tailored for this problem. In general, branch-and-cut is a powerful tool that is commonly used in the field of OR, and this power springs from the use of bounds.

Duality theory Associated with any minimization LP is its *dual* problem, which is a maximization LP [22]. After solving an LP, the dual supplies bounds on changes that can be made to coefficients in the original problem without affecting optimality. These bounds are used for *sensitivity analysis*, an important process in practical applications, where sensitive coefficients are identified and insensitive coefficients are sometimes adjusted, without affecting optimality.

These bounds have also been used in more creative ways. For example, Carpaneto, Dell'Amico, and Toth [5] developed a pre-processing tool for graph reduction, or *sparsification*, for the ATSP using these bounds. First, a simple upper bound is determined by finding a feasible, though not necessarily optimal, solution. Then, each arc (i, j) is eliminated if the lower bound when forcing (i, j) into the solution exceeds the simple upper bound. This lower bound is derived from the dual problem.

In addition to bounds used in sensitivity analysis, the dual provides another useful bound. It is known that any feasible, though not necessarily optimal, solution to the dual provides a lower bound for the optimal value of the original (primal) problem [22]. This bound can be used to determine the quality of an approximate solution for the primal problem. If the lower bound provided by the dual is close to the value of the approximate solution, then the approximate solution is close to optimal.

2.2 Heuristic evaluation functions in artificial intelligence

Most AI systems are symbolic (or logic) and are very often concerned with finding shortest or cheapest sequences of actions to transform the systems from initial states (e.g., before starting to solve a combinatorial puzzle) to goal states (e.g., when the puzzle is solved). Thus, solving such a problem involves searching for a sequence of actions. Formally, let S_0 and S_G be the initial and goal states of a problem, and $\mathcal{O}$ the set of all possible sequences of operations that can turn S_0 into S_G. Finding a cheapest sequence of operations to change S_0 into S_G is equivalent to solving the minimization problem of

$$O = \arg\min_{O'=\langle o_1,o_2,\cdots,\rangle\in\mathcal{O}} \sum_i \{c_i | c_i \text{ is the cost of operator } o_i\} \tag{7}$$

subject to the constraints that operator o_i is applicable to the corresponding state. The success in solving such a problem depends mainly on the performance of the search method used; the performance of the search is in turn determined by the bound functions adopted.

Developing lower bound functions has been and still remains a core problem in AI. Lower bound functions are termed as admissible heuristic evaluation functions, or evaluation function for short. Here, admissibility means a function will never overestimate the cost of the best solution. The research on developing evaluation functions can be traced back to the early days of AI. When Herb Simon, a Nobel laureate and a founding father of AI, and his colleagues were building their General Problem Solving (GPS) system and chess program, and Samuel was developing his checkers program, these pioneers were fully aware of the importance of evaluation functions and made great strides in developing effective evaluation functions. In early computer board game programs, they introduced evaluation functions that sum up the positions and strengths of individual pieces on the board. These concepts form the basis of evaluation functions of modern computer game programs such as Deep Blue [4], which beat Kasporov, the human champion, in a tournament setting.

Abstraction and pattern databases Automatic, problem-independent approaches for deriving evaluation functions have been pursued over the past decades. An early related work was the development of macro operators by Korf [29, 30], which essentially worked on an abstract search space. Problem abstraction is a general method for developing evaluation functions, and has been studied and applied to planning and combinatorial games and puzzles as well as many other AI problems [23, 39]. The goal here is to construct an abstraction, or a simplified version, of the problem by ignoring some of its constraints. The optimal solution to the abstraction then constitutes an evaluation function.

Another line of work is pattern databases [10]. The main idea is to consider only a partial problem, e.g., an end game in chess, where many entries and constraints do not exist or are ignored. A partition to the original problem is then considered; a combination of the costs of optimal solutions to these partial problems is then an evaluation function [14]. The pattern database method has been studied on many combinatorial puzzles, such as sliding-tile puzzles, 4-peg Towers of Hanoi and Rubik's cube, as well as the well-known NP-hard vertex cover problem [14].

In short, the central scheme of deriving evaluation functions by abstraction and pattern databases is constraint relaxation and elimination.

Pearl's insight Based on the work in OR, e.g., those described in Section 2.1, Pearl had the insight that evaluation functions can be obtained by eliminating some constraints to a problem [33]. As most AI systems are logic, he suggested following the STRIPS representation [15] to formulate an operator, in the form of a set of preconditions (constraints) that must be satisfied before the operator can be applied, a set of positive effects (add list), and a set of negative effects (delete list) that the operator will result in, all represented in predicates. For example, in a planning problem in which shipping cargo from one place to another using airplanes can be represented as

$$\text{Operator}: Fly(X_{plane}, X_{origin}, X_{destination}, X_{cargo}) - \tag{8}$$

$$\text{Precondition}: At(X_{plane}, X_{origin}); Inside(X_{cargo}, X_{plane}); Fuel(X_{plane}); \tag{9}$$

$$\text{Add-list}: At(X_{plane}, X_{destination}); \tag{10}$$

$$\text{Delete-list}: At(X_{plane}, X_{origin}); Fuel(X_{plane}). \tag{11}$$

where variables $X_{plane}, X_{origin}, X_{destination}$ and X_{cargo} need be substituted by airplane, a place where the airplane is currently located, a destination for the flight, and a piece of cargo to be shipped. In this example, to ship the cargo, it must be inside of the airplane, which must be fueled and at the origin of the trip. After the flight, the airplane will be at the destination (with the cargo still inside) and the fuel consumed. These new properties are properly captured by the Add-list and Delete-list.

Pearl's idea is to ignore some of the constraints (preconditions) of the operators, giving rise to a simplified problem. The cost of an optimal solution to the modified problem is an underestimate of the optimal cost to the original problem. Pearl's ideas can evidently lead to a systematic approach to deriving lower bound functions within the STRIPS paradigm. Unfortunately, these ideas have not been pursued.

2.3 Differences and commonalities

To summarize the existing work using both LP and STRIPS representations, we point out differences and commonalities. The first difference lies in the objectives that they attempt to capture. For most combinatorial optimization problems in OR, the objectives demand optimal solutions that satisfy global constraints such as those in (4-6). On the other hand, for most problems in AI, the objectives are shortest sequences of actions to transform initial states to goal states. The second difference is the locality of the constraints involved. The constraints in an LP are global, while the constraints for a STRIPS operator are local to the operator and the states on which the operator can apply. Another difference is that OR problems are usually numerical, while most AI problems are logic. Finally, the LP formulation is richer in content than the STRIPS formulation, as the former also has a dual representation.

Even though LP and STRIPS representations were developed in different fields of study and for different types of problems, they both have constraints as the main components. In other words, both representations share a similar syntax of optimizing an objective function under a set of constraints, even if the constraints have different semantics. Note that it is through modifications to some constraints in both formulations that lower bound functions were derived, as we discussed above. These observations motivated us to explore systematic ways to manipulate constraints and other elements in these representations to develop effective bound functions, which is the central theme of this paper.

3 Bounding framework

In this section we develop systematic bounding techniques for combinatorial optimization problems in OR and AI. First, we explore various ways that bounds can be derived. We then present a two-step procedure for using these bounds.

3.1 Systematic bounding techniques

We explore in this section the multitude of ways that problems in LP and STRIPS formulations can be modified and manipulated to obtain upper and lower bounds. Note that

LP and STRIPS representations in spirit have similar syntax and the LP formulation is richer than the STRIPS formulation as the former has a dual representation. To simplify our discussion and without loss of generality, we will refer to the LP formulation in the discussion.

Note that, unlike modifications made in sensitivity analysis, these modifications may result in problems that have completely different optimal solutions than the original problem. The purpose of these modifications is to identify a problem whose solution is an upper or lower bound on the original optimal solution, so preserving optimality is of no consequence.

Producing upper bounds An LP can be modified to produce a new problem that has a solution value Z' that is an upper bound on the original optimal solution Z using the following seven different types of modifications.

Tightening or adding constraints: An upper-bound modification can be made by adding new constraints to (2) and/or tightening the original constraints. For instance, if a constraint is an inequality with a constant for its right hand side, this constant can be appropriately increased or decreased to achieve a tightening of the original constraint. In general, tightening constraints usually causes the deletion of a number of feasible solutions and cannot create new feasible solutions.

Modifying objective function coefficients: This modification manipulates the instance-specific coefficients, c_i's, in the objective function (1). An upper-bound modification can be made by increasing any of these coefficients. Furthermore, if we decrease and/or increase any number of these coefficients, solve the problem, and then substitute the original c_i values when calculating Z', this Z' would be an upper bound on the optimal solution for the original problem as it would be the actual cost of a feasible, though not necessarily optimal, solution.

Deleting decision variables: Assuming that zero is in the allowable range of a decision variable, its deletion is an upper-bounding modification. This is equivalent to adding a constraint that sets the variable's value to zero, which reduces the number of feasible solutions without creating new solutions.

Relaxing optimality: This type of modification is simple and commonly used. An upper-bound modification can be made by relaxing the minimization requirement in the objective function (1). This modification is very useful as it yields solutions that are feasible, though not necessarily optimal. Many polynomial approximation algorithms for NP-hard problems fall into this category.

Relaxing constraints in the dual problem: Relaxing constraints in the dual, which is a maximization problem, is an upper-bounding modification. This relaxation may increase the value of the dual problem and this new value is an upper bound for the primal problem. This modification may increase the number of feasible solutions for the dual.

Adding decision variables to the dual problem: Assuming that a new decision variable has zero in its allowable range, adding such a variable leads to an upper-bounding modification for the primal problem. For this revised problem, there is a subset of the solution space for the dual in which the new variable has a value of zero. This subset is identical to the entire solution space of the dual before the addition of the new variable. Among the new solutions created by this variable addition there may exist a solution whose cost is greater than the cost of the original optimal solution for the dual. Thus, this modification is upper bounding for the dual, and consequently, for the primal problem.

Increasing objective function coefficients in the dual: Increasing the objective function coefficients for the dual is an upper-bounding modification for both the dual and the primal problem.

Producing lower bounds Similarly, lower-bounding modifications can be obtained by using the following seven modifications.

Relaxing constraints: The relaxation, or the complete omission, of one or more of the constraints in (2) is a lower-bound modification for a minimization LP. In general, relaxing constraints maintains the original feasible solutions and usually yields additional feasible solutions.

Decreasing objective function coefficients: This modification manipulates the instance-specific coefficients, c_i's, in the objective function (1). A lower-bound modification can be created by decreasing any number of these coefficients.

Adding decision variables: Assuming that zero is in the allowable range for a new decision variable, its addition is a lower-bounding modification. As with adding variables to the dual, adding them to the primal problem creates additional feasible solutions without excluding any of the feasible solutions to the original problem.

Relaxing optimality for the dual: As discussed in Section 2.1, a lower bound is produced by relaxing the maximization requirement of the dual to derive a feasible, though not necessarily optimal, solution to the dual.

Tightening constraints for the dual: Tightening and/or adding constraints for the dual is a lower-bounding modification for the dual as well as the primal problem.

Deleting decision variables in the dual: Assuming that zero is in the allowable range for a variable, its deletion has the same effect as adding a constraint setting its value to zero. When this modification is made to the dual, it yields a lower bound for the dual as well as the primal problem.

Modifying objective function coefficients for the dual: Decreasing the objective function coefficients for the dual is a lower-bounding modification for the dual as well as the primal problem. Decreasing and/or increasing these coefficients, solving the modified dual problem, and substituting the original coefficients when computing the value of the dual's objective function yield a feasible, though not necessarily optimal, solution for the dual. Since the dual is a maximization problem, this new solution is a lower bound for the dual as well as the primal problem.

Summary of modifications and discussion Relaxing the minimization requirement of the objective function yields an upper bound, while relaxing the maximization requirement of the dual problem produces a lower bound. Constraints can be modified in any number of ways. One or more constraints can be relaxed, additional constraints can be introduced, and/or coefficients in the constraints can be altered. For the primal problem, modifications that result in the elimination of feasible solutions generate an upper bound, while modifications that increase the number of feasible solutions give rise to a lower bound. For the dual, the reverse is true. Deleting decision variables in the primal or adding variables in the dual produces an upper bound. Conversely, adding more decision variables to the primal or deleting variables in the dual provides a lower bound. Finally, the instance-specific coefficients in the objective function can be increased and/or decreased. For both the primal and dual problems, increasing these coefficients is upper bounding, while decreasing them is lower bounding. However, if the original c_i values are substituted back in after the problem has been solved, then the resulting solution value is strictly an upper bound when the primal problem is modified and a lower bound when the dual problem is modified.

Some of these modifications have been extensively used for deriving bounds while others have been overlooked. Relaxing the minimization of the objective function and/or relaxing the maximization requirement for the dual is commonly used for deriving bounds. Furthermore, relaxing constraints to derive bounds is extensively used in BnB search. On the other hand, tightening constraints for bounds has a very limited history. The only common tightening is the addition of constraints that set values of particular decision variables. These types of constraints are essentially the branching rules used in BnB search. Finally, the modification of objective function coefficients, to our knowledge, has not been previously used to derive bounds. In this paper, we demonstrate ways that these overlooked modifications can produce effective bounds.

3.2 Limit crossing

The previous discussion describes a variety of ways that upper-bounding and lower-bounding modifications can be made to an LP. In this section, a two-step procedure for using these bounds is synopsized. For convenience, we refer to this procedure as *limit crossing*.

The first step is to find a simple upper bound for a problem using one of the modifications described. The second step is to simultaneously combine upper- and lower-bounding modifications and solve the resultant doubly-modified problem. If the solution cost is greater than the simple upper bound, it can be concluded that the upper-bounding modification of the second step cannot hold for *any* optimal solution. Similarly, if the solution cost is less than the simple lower bound, then the lower-bounding modification of the second step cannot hold for *any* optimal solution.

These conclusions are easy to justify. Consider the first case, when the simple upper bound is exceeded. Since the combined modifications result in the simple upper bound being exceeded, at least one of those modifications must be invalid for every optimal solution. The lower-bounding modification cannot contribute to any increase in the solution cost, therefore, for each optimal solution, the upper-bounding modification must contain at least one part that is invalid.

Now, consider the case when the doubly-modified problem solution is *equal* to the simple upper-bound value. The Z' value derived from the doubly-modified problem is the least possible value that can be derived given its upper-bounding modification. If this value is equal to a simple upper bound that has already been found, then the double modifications cannot lead to a better solution value than the simple upper-bound value. Here again, the upper-bounding portion of the double modification must be the cause of this situation. Conversely, if the doubly-modified problem solution is equal to a simple lower bound, then the lower-bounding portion of the double modifications cannot lead to a solution with a better value than the simple lower bound.

3.3 Selections of modifications

There are numerous possibilities for selecting modifications. However, there are a couple of considerations to contemplate when making these choices for effective modifications. When selecting a modification for the simple upper or lower bound, the tightest bound that can be achieved in the allotted time is desirable as this will increase the possibility for the bounds to cross. For the doubly-modified problem, the modifications should be selected so as to maximize the usefulness of the information gained. It is important to observe that while either of these modifications may yield difficult problems, the *combination* of the two modifications must render a relatively easy problem for the method to be efficient.

It is easy to see how modifying constraints and deleting decision variables can lead to a new problem that is relatively easy to solve. Following are examples demonstrating how modifying objective function coefficients and adding decision variables can also lead to relatively easy problems.

Modifications by exploiting phase transitions An exciting new area of development in the study of combinatorial problems concerns the existence of *phase transitions* in which the difficulty in solving a problem changes dramatically when a parameter is increased beyond a distinct value [27, 32, 37]. When optimally solving random ATSPs, it has been shown that the cost range affects average-case performance of BnB searches. Indeed, evidence of dramatic jumps from difficult instances to easy instances when the range of values in the c_{ij} matrix falls below a certain value are demonstrated in [37].

Phase transitions may be the root of Frieze's ability to introduce a polynomial-time algorithm [17] that solves the STSP *exactly* with a probability that tends to 1 as the number of cities, n, tends to infinity, for instances with random cost values drawn from a range of zero to $B(n) - 1$, where $B(n) = o(\frac{n}{\log \log n})$.

In our recent work [36], we exploit phase transitions using an unusual upper-bounding technique. We modify the objective function coefficients of "hard" problems with large cost ranges, in order to reduce the range of values and derive an "easy" problem on the other side of the phase transition. After quickly solving this modified problem, we substitute the original coefficients into the objective function when calculating Z', yielding an upper bound as well as a feasible approximate solution. In addition to being a practical tool for finding approximate solutions and for deriving upper bounds, this technique demonstrates how modifications of objective function coefficients can yield relatively easy problems to solve.

Converting ATSPs to STSPs Deleting decision variables reduces the size of the problem, so it can be expected that the removal of a portion of the decision variables would yield an easier problem to solve. It is not so apparent how the *addition* of decision variables can yield an easier problem. Yet, the ATSP again offers an example of such a case. ATSP instances can be transformed into STSP instances using a *2-node* transformation [26]. While the number of arcs after this transformation is increased to $4n^2 - 2n$, the number of arcs that have neither a zero nor infinite cost is $n^2 - n$, as in the original problem. Once the problem is converted to an STSP, it can be solved by STSP solvers, such as Concorde [1]. It is shown in [16] that the computation time to solve five 100-city ATSPs that correspond to tilted drilling machine problems (with additive norms) is on average faster using the STSP conversion than using state-of-the-art ATSP solvers on the original instances. Furthermore, similar results are shown in [9, 7], where a total of 800 100-city instances were solved. In these cases, the number of decision variables were more than quadrupled, yet the resulting modified problems were generally easier to solve than the original instances.

3.4 Previous use of limit crossing

This limit crossing method has not been previously formalized, but a less general, preliminary technique was made explicit and analyzed in our recent work on finding backbone and fat variables for the ATSP[8]. Furthermore, the limit crossing method has been implicitly used in previous work by others. A prevalent example is the pruning rules of BnB search. The simple upper bound used in this strategy is the incumbent solution. The doubly modified problem is composed of the relaxation that is solved at each node (the lower-bounding modification), while the upper-bounding modification is due to the branching rules (e.g., those in [2]), where decision variables are fixed. When the value of this doubly-modified problem exceeds the simple upper bound, the entire subtree can be pruned as every node in this subtree is subject to the upper-bounding modifications of the double modifications. Another example is Carpaneto, Dell'Amico, and Toth's sparsification technique that is summarized in Section 2.1. After finding a simple upper bound, the doubly-modified problem consists of determining a lower bound while forcing the inclusion of an arc (i, j). If the limits cross, the inclusion of (i, j) cannot lead to any optimal solution.

While many spontaneous "discoveries" that use the limit crossing concept have been previously made, formalizing this strategy is conducive for eliciting novel methods that go beyond search tree pruning.

4 Applications

In this section, we present two recent developments that are direct products of the limit crossing strategy to demonstrate how the use of our new framework is conducive for eliciting methods that go beyond search-tree pruning.

4.1 Cut-and-solve

In this section, we discuss a linear search strategy which we refer to as *cut-and-solve* [9, 7]. This method is based on the limit crossing concept and exploits an unusual upper-bounding modification in the doubly-modified problem. This modification is the addition of constraints that require that the sum of a set of decision variables be less than or equal to a constant.

Cut-and-solve is different from traditional tree search as there is no branching. At each node in the search path, a relaxed problem and a sparse problem are solved and a constraint is added to the relaxed problem. The sparse problems provide incumbent solutions. When the constraining of the relaxed problem becomes tight enough, its solution cost becomes no better than the incumbent solution cost. At this point, the incumbent solution is declared to be optimal.

To avoid branching, cut-and-solve uses cutting planes, as its name implies. However, unlike the cutting planes in branch-and-cut, cut-and-solve uses what we call *piercing cuts* that intentionally cut out at least one feasible solution from the solution space of the original (unrelaxed) problem. The addition of piercing cuts to a relaxation doesn't just tighten the relaxation, as is the case for conventional cutting planes.

Cut-and-solve enjoys a few favorable properties. First, its memory requirements are insignificant as only the current incumbent solution and the current doubly-modified problem need to be saved as the search path is traversed. Second, since there is no branching, there is no "wrong" subtree in which the search may get lost. A tree search method such as BnB, on the other hand, is commonly solved using depth-first search or best-first search. Depth-first search may very often search fruitlessly in a subtree with no optimal solution. Best-first search can overcome this aimless search problem, but requires a large amount of memory. Cut-and-solve is free from both of these difficulties. Third, it can be easily adapted to an *anytime* algorithm as an incumbent solution is found at the root node and continuously updated during the search. The algorithm can be terminated at any time during its execution with the current incumbent as an approximate solution.

We have had excellent results using cut-and-solve for solving difficult, real-world instances of the ATSP [9, 7]. Many real-world ATSP instances have intrinsic structures that make them relatively difficult to solve using conventional algorithms. While state-of-the-art implementations outperform cut-and-solve for instances that are relatively easy to solve, cut-and-solve was faster, on average, for the difficult real-world problems.

4.2 Finding backbones and fat

Another project that explicitly uses the limit-crossing strategy is a tool for identifying *backbones* and *fat* [8]. Here, a backbone variable is a binary decision variable that has a value of one for all optimal solutions, and a fat variable is one that has a value of zero for all optimal solutions. Finding backbone variables is useful as they are critical components of a system and it may be desirable to reduce their number and derive a more redundant system. It is useful to identify fat variables as they can be completely eliminated from the system without any loss. In addition to being beneficial for improving physical systems, identification of backbones and fat can reduce the problem size and this reduction is especially valuable when enumerating all optimal solutions. These problems are #P problems and harder than the original ATSP. In the limit crossing method we used in [8], we used the relaxation of the subtour elimination constraints coupled with the elimination of arcs eminating from a particular city to identify backbone and fat variables. This procedure uses polynomial time and we were able to identify roughly half of the backbones and 99% of the fat variables for random ATSPs.

5 Conclusions

In this paper, we introduce a framework for exploiting bounds in OR and AI. New techniques for deriving bounds are explored and a two-step procedure, called limit crossing, is formalized. A goal of this paper is to shed light on potential limit-crossing opportunities that may have been previously overlooked by pure discovery. Within the realm of linear programs, we have identified numerous ways in which limit crossing might be invoked.

Limit crossing is a powerful technique, as demonstrated by previous instantiations that resulted from spontaneous discoveries as well as recent projects that are the direct product of utilizing this formalized approach. Nevertheless, these implementations have only scraped the surface of the vast possibilities that may be exploited. Future work is needed to reveal the full potential of this intriguing methodology.

References

1. D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Concorde - A code for solving Traveling Salesman Problems. 15/12/99 Release, *http:www.keck.caam.rice.edu/concorde.html*, web.
2. E. Balas and P. Toth. Branch and bound methods. In *The Traveling Salesman Problem*, pages 361–401. John Wiley & Sons, Essex, England, 1985.
3. P.M. Camerini, L. Fratta, and F. Maffioli. A note on finding optimum branchings. *Networks*, 9:309–12, 1979.
4. M. Campbell, A.J. Hoane, and F h. Hsu. Deep blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.
5. G. Carpaneto, M. Dell'Amico, and P. Toth. Exact solution of large-scale, asymmetric Traveling Salesman Problems. *ACM Trans. on Mathematical Software*, 21:394–409, 1995.
6. Y.J. Chu and T.H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–00, 1965.
7. S. Climer and W. Zhang. Cut-and-solve: A linear search strategy for combinatorial optimization problems. *Artificial Intelligence*, accepted.
8. S. Climer and W. Zhang. Searching for backbones and fat: A limit-crossing approach with applications. In *Proc. of the 18th Nat. Conf. on Artificial Intelligence (AAAI-02)*, pages 707–712, Edmonton, Alberta, July 2002.
9. S. Climer and W. Zhang. A linear search strategy using bounds. In *14th Inter. Conf. on Automated Planning and Scheduling*, pages 132–141, Whistler, Canada, June 2004.
10. J. Culberson and J. Schaeffer. Searching with pattern databases. *Advances in AI, 11th Biennial Conf. of the Canadian Soc. for Computational Studies of Intelligence, AI'96. Springer Lecture Notes in Artificial Intelligence*, 1081:402–416, 1996.
11. G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410, 1954.
12. G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. On a linear programming, combinatorial approach to the traveling salesman problem. *Operations Research*, 7:58–66, 1959.
13. J. Edmonds. Optimum branchings. *J. Research of the National Bureau of Standards*, 71B:233–40, 1967.
14. A. Felner, R.E. Korf, and S. Hanan. Additive pattern database heuristics. *J. Artificial Intelligence Research*, 22:279–318, 2004.
15. R.E. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 5(2):189–208, 1971.

16. M. Fischetti, A. Lodi, and P. Toth. Exact methods for the Asymmetric Traveling Salesman Problem. In G. Gutin and A. Punnen, editors, *The Traveling Salesman Problem and its Variations*. Kluwer Academic, Norwell, MA, 2002.
17. A. Frieze. On the exact solution of random traveling salesman problems with medium size integer coefficients. *SIAM Computing*, 16:1052–1072, 1987.
18. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, NY, 1979.
19. R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958.
20. M. Held and R. M. Karp. The traveling salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.
21. M. Held and R. M. Karp. The traveling salesman problem and minimum spanning trees: Part ii. *Mathematical Programming*, 1:6–25, 1971.
22. F. Hillier and G. Lieberman. *Introduction to Operations Research*. McGraw-Hill, Boston, 6th edition, 2001.
23. R.C. Holte and B.Y. Choueiry. Abstraction and reformulation in artificial intelligence. *Philosophical Transactions of the Royal Society of London, SeriesB: Biological Sciences*, 358(1435):1197–204, 2003.
24. T. Ibaraki. *Enumerative Approaches to Combinatorial Optimization — Part II*, volume 11 of *Annals of Operations Research*. Scientific, Basel, Switzerland, 1987.
25. T. Ibaraki. *Enumerative Approaches to Combinatorial Optimization - Part I*, volume 10 of *Annals of Operations Research*. Scientific, Basel, Switzerland, 1987.
26. R. Jonker and T. Volgenant. Transforming asymmetric into symmetric traveling salesman problems. *Operations Research Letters*, 2:161–163, 1983.
27. S. Kirkpatrick and G.Toulouse. Configuration space analysis of traveling salesman problems. *J. de Physique*, 46:1277–1292, 1985.
28. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.
29. R.E. Korf. *Learning to Solve Problems by Searching for Macro-Operators*. Pitman Publishing Ltd, London, 1985.
30. R.E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.
31. S. Martello and P. Toth. Linear assignment problems. *Annals of Discrete Math.*, 31:259–282, 1987.
32. R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400:133–137, 1999.
33. J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, MA, 1984.
34. R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, Philadelphia, 1983.
35. W. Zhang. *State-Space Search: Algorithms, Complexity, Extensions, and Applications*. Springer, New York, NY, 1999.
36. W. Zhang. Phase transitions, backbones, measurement accuracy, and phase-aware approximation: The ATSP as a case study. *Proc. 4th Intern. Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 345–357, 2002.
37. W. Zhang. Phase transitions and backbones of the asymmetric Traveling Salesman. *Journal of Artificial Intelligence Research*, 20:471–497, 2004.
38. W. Zhang and R. E. Korf. Performance of linear-space search algorithms. *Artificial Intelligence*, 79:241–292, 1995.
39. J.-D. Zucker. A grounded theory of abstraction in artificial intelligence. *Philosophical Trans. of the Royal Society of London, SeriesB: Biological Sciences*, 358(1435):1293–309, 2003.