Report Number: WUCSE-2005-1

2005-08-25

Cut-and-Solve: A Linear Search Strategy for Combinatorial Optimization Problems

Sharlee Climer and Weixiong Zhang

Branch-and-bound and branch-and-cut use search trees to identify optimal solutions. In this paper, we introduce a linear search strategy which we refer to as cut-and-solve and prove optimality and completeness for this method. This search is different from traditional tree searching 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 value becomes no better than the incumbent solution value. At... Read complete abstract on page 2.

Cut-and-Solve: A Linear Search Strategy for Combinatorial Optimization Problems

Sharlee Climer and Weixiong Zhang

Complete Abstract:

Branch-and-bound and branch-and-cut use search trees to identify optimal solutions. In this paper, we introduce a linear search strategy which we refer to as cut-and-solve and prove optimality and completeness for this method. This search is different from traditional tree searching 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 value becomes no better than the incumbent solution value. At this point, the incumbent solution is declared to be optimal. This strategy is easily adapted to be an anytime algorithm as an incumbent solution is found at the root node and continuously updated during the search. Cut-and-solve enjoys two favorable properties. Since there is no branching, there are no "wrong" subtrees in whihc the search may get lost. Furthermore, its memory requirements are nominal. For these reasons, it may be potentially useful as an alternative approach for problems that are difficult to solve using depth-first or best-first search tree methods. In this paper, we demonstrate the cut-and-solve strategy by implementing it for the Asymmetric Traveling Salesman Problem (ATSP). We compare this implementation with state-of-the-art ATSP solvers to validate the potential of this novel search strategy. Our code is available at our websites.

2005-1

Cut-and-Solve: A Linear Search Strategy for Combinatorial Optimization Problems

Authors: Climer, Sharlee; Zhang, Weixiong

August 25, 2004

Abstract: Branch-and-bound and branch-and-cut use search trees to identify optimal solutions. In this paper, we introduce a linear search strategy which we refer to as cut-and-solve and prove optimality and completeness for this method. This search 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 value becomes no better than the incumbent solution value. At this point, the incumbent solution is declared to be optimal. This strategy is easily adapted to be an anytime algorithm as an incumbent solution is found at the root node and continuously updated during the search.

Cut-and-solve enjoys two favorable properties. Its memory requirements are nominal and, since there is no branching, there are no "wrong" subtrees in which the search

Cut-and-Solve: A Linear Search Strategy for Combinatorial Optimization Problems

Sharlee Climer, Weixiong Zhang

Department of Computer Science and Engineering, Washington University, One Brookings Drive; St. Louis, MO 63130-4899, USA

Abstract

Branch-and-bound and *branch-and-cut* use search trees to identify optimal solutions. In this paper, we introduce a linear search strategy which we refer to as *cut-and-solve* and prove optimality and completeness for this method. This search 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 value becomes no better than the incumbent solution value. At this point, the incumbent solution is declared to be optimal. This strategy is easily adapted to be an *anytime* algorithm as an incumbent solution is found at the root node and continuously updated during the search.

Cut-and-solve enjoys two favorable properties. Since there is no branching, there are no "wrong" subtrees in which the search may get lost. Furthermore, its memory requirements are nominal. For these reasons, it may be potentially useful as an alternative approach for problems that are difficult to solve using depth-first or best-first search tree methods.

In this paper, we demonstrate the cut-and-solve strategy by implementing it for the Asymmetric Traveling Salesman Problem (ATSP). We compare this implementation with state-of-the-art ATSP solvers to validate the potential of this novel search strategy. Our code is available at our websites.

Key words: Search strategies, branch-and-bound, branch-and-cut, anytime algorithms, linear programming, Traveling Salesman Problem

Email addresses: sclimer@cse.wustl.edu (Sharlee Climer), zhang@cse.wustl.edu (Weixiong Zhang).

1 Introduction

Life is full of optimization problems. We are constantly searching for ways to minimize cost, time, energy, or some other valuable resource, or maximize performance, profit, production, or some other desirable goal, while satisfying the constraints that are imposed on us. Optimization problems are interesting as there are frequently a very large number of *feasible* solutions that satisfy the constraints; the challenge lies in searching through this vast solution space and identifying an optimal solution. When the number of solutions is too large to explicitly look at each one, two search strategies, *branch-and-bound* [4] and *branch-and-cut* [23], have been found to be exceptionally useful.

Branch-and-bound uses a search tree to pinpoint an optimal solution. (Note there may be more than one optimal solution.) If the entire tree were generated, every feasible solution would be represented by at least one leaf node. The search tree is traversed and a relaxed variation of the original problem is solved at each node. When a solution to the relaxed subproblem is also a feasible solution to the original problem, it is made the *incumbent* solution. As other solutions of this type are found, 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 current incumbent is returned as an optimal solution.

If the number of solutions is too large to allow explicitly looking at each one, then the search tree is also too large to be completely explored. The power of branch-and-bound comes from its *pruning* rules, which allow pruning of entire subtrees while guaranteeing optimality. If the tree is pruned to an adequately small size, the problem becomes soluble and can be solved to optimality.

Branch-and-cut improves on branch-and-bound by increasing the probability of pruning. At some or all of the nodes, *cutting planes* [23] are added to tighten the relaxed subproblem. These cutting planes remove a set of solutions for the relaxed subproblem. However, in order to ensure optimality, these cutting planes are designed to never exclude any feasible solutions to the current unrelaxed subproblem.

While adding cutting planes can substantially increase the amount of time spent at each node, these cuts can dramatically reduce the size of the search tree and have been used to solve a great number of problems that were previously insoluble.

Branch-and-bound and branch-and-cut are typically implemented in depth-first fashion due to its linear space requirement and other favorable features [40]. However, depth-first search can suffer from the problem of exploring subtrees with no optimal solution, resulting in a large search cost. A wrong choice of a subtree to explore in an early stage of a depth-first search is usually difficult to rectify without exploring the entire search space of the chosen subtree. Much effort has been devoted to addressing this issue in depth-first search in general. Branching techniques [4], heuristics investigations [37], and search techniques such as limited discrepancy [19] and randomization and restarts [16] have been developed in an effort to combat this persistent problem.

In this paper, we introduce a linear search strategy to overcome the problem of making wrong choices in depth-first branch-and-bound for optimization problems while keeping memory requirements nominal. We refer to this linear search strategy as *cut-and-solve* and demonstrate it on linear programs. Being linear, there is no search tree, only a search path that is directly traversed. In other words, there is only one child for each node, so there is no need to choose which child to traverse next. We search for a solution along a predetermined path. At each node in the search path, two relatively easy subproblems are solved. First, a relaxed solution is found. Then a *sparse* problem is solved. Instead of searching for an optimal solution in the vast solution space containing every feasible solution, a very sparse solution space is searched. An incumbent solution is found at the first node and modified as needed at subsequent nodes. When the search terminates, the current incumbent solution is declared to be an optimal solution. In this paper, we prove the optimality and completeness of the cut-and-solve strategy.

The paper is organized as follows. In the next section, branch-and-bound and branch-and-cut are discussed in greater detail. In the following section, the cut-and-solve strategy is described and compared with these prevalent techniques. Next, we illustrate this strategy by applying it to a simple linear programming problem. Then we demonstrate how cut-and-solve can be utilized by implementing an algorithm for the Asymmetric Traveling Salesman Problem (ATSP). (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 may not necessarily be equal to the cost from city j to city i.) We have quickly produced an implementation of this algorithm and compare it with branch-and-bound and branch-and-cut ATSP solvers. Our tests show that cut-and-solve is competitive with these state-of-the-art solvers. This paper is concluded with a discussion of this technique and related work. An early version of this paper appeared in [9].

2 Background

In this section, we define several terms and describe branch-and-bound and branch-and-cut in greater detail, using the Asymmetric Traveling Salesman Problem (ATSP) as an example.

Branch-and-bound and branch-and-cut have been used to solve a variety of optimization problems. The method we present in this paper can be applied to any such problem. However, to make our discussion concrete, we will narrow our focus to Linear Programs (LPs). An LP is an optimization problem that is subject to a set of linear constraints. LPs have been used to model a wide variety of problems, including the Traveling Salesman Problem (TSP) [18,31], Constraint Satisfaction Problem (CSP) [12], and minimum cost flow problem [22]. Moreover, a wealth of problems can be cast as one of these more general problems. The TSP has applications for a vast number of scheduling, routing, and planning problems such as the no-wait flowshop, stacker crane, tilted drilling machine, computer disk read head, robotic motion, and pay phone coin collection problems [26]. Furthermore, the TSP can be used to model surprisingly diverse problems, such as the shortest common superstring problem, which is of interest in genetics research. The CSP is used to model configuration, design, diagnosis, spatio-temporal reasoning, resource allocation, graphical interfaces, network optimization, and scheduling problems [12]. Finally, the minimum cost flow problem is a general problem that has the shortest path, maximum flow, transportation, transshipment, and assignment problems as special cases [22].

A general LP can be written in the following form:

$$Z = min\ (or\ max)\ \sum_i c_i x_i \tag{1}$$

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

where the c_i values are instance-specific constants, the set of x_i represents the *decision variables*, and the constraints are linear equalities or inequalities composed of constants, decision variables, and possibly some auxiliary variables. A *feasible* solution is one that satisfies all of the constraints. The set of all feasible solutions is the *solution space*, SS, for the problem. Here, the solution space is defined by the given problem. (In contrast, the search space is defined by the algorithm used to solve the problem.) For minimization problems, an *optimal* solution is a feasible solution with the least value, as defined by the *objective function* (1).

For example, the ATSP can be defined as:

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

subject to:

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

$$\sum_{j \in V} x_{ij} = 1, \; \forall i \in V \tag{5}$$

$$\sum_{i \in W} \sum_{j \in W} x_{ij} \leq |W| - 1, \; \forall \, W \subset V, \; W \neq \emptyset \tag{6}$$

$$x_{ij} \in \{0,1\}, \; \forall i, j \in V \tag{7}$$

for directed graph $G = (V, A)$ with vertex set $V = \{1, \ldots, n\}$, arc set $A = \{(i,j) \mid i, j = 1, \ldots, n\}$, 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. Constraints (7) requires that either an arc (i,j) is traversed (x_{ij} is equal to 1) or is not traversed (x_{ij} is equal to 0). Constraints (4) and (5) require that each city is entered exactly once and departed from exactly once. Constraints (6) are called *subtour elimination constraints* as they require that no more than one cycle can exist in the solution. Finally, the objective function (3) requires that the sum of the costs of the traversed arcs is minimized. In this problem, the solution space is the set of all permutations of the cities and contains $(n-1)!$ discrete solutions.

2.1 Using bounds

Without loss of generality, we only discuss minimization problems in the remainder of this paper.

An LP can be *relaxed* by relaxing one or more of the constraints. This relaxation is a *lower-bounding* modification as an optimal solution for the relaxation cannot exceed the optimal solution of the original problem. Furthermore, the solution space of the relaxed problem, SS_r, contains the solution space of the original problem, SS_o, however, the converse is not necessarily true.

An LP can be *tightened* by tightening one or more of the constraints or adding additional constraints. This tightening is an *upper-bounding* modification as an optimal solution for the tightened problem cannot have a smaller value than the optimal solution of the original problem. Furthermore, the solution space of the original problem, SS_o, contains the solution space of the tightened problem, SS_t, however, the converse is not necessarily true. In summary, $SS_t \subseteq SS_o \subseteq SS_r$.

For example, the ATSP can be relaxed by relaxing the integrality requirement of constraints (7). This can be accomplished by replacing constraints (7) with the following constraints:

$$0 \leq x_{ij} \leq 1, \ \forall i, j \in V \tag{8}$$

This relaxation is referred to as the *Held-Karp* relaxation [20,21].

Another relaxation can be realized by completely omitting constraints (6). This relaxation enforces integrality but allows any number of subtours to exist in the solution. This relaxed problem is simply the Assignment Problem (AP) [34]. 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.

One way the ATSP can be tightened is by adding constraints that set the values of selected decision variables. For example, adding $x_{ij} = 1$ forces the arc (i, j) to be included in all solutions.

2.2 Branch-and-bound search

The branch-and-bound concept was perhaps first used by Dantzig, Fulkerson, and Johnson [10,11], and first approached in a systematic manner by Eastman [13]. This method organizes the search space into a tree structure. At each level of the tree, branching rules are used to generate and tighten each child node. Every node inherits all of the tightening modifications of its ancestors. These tightened problems represent subproblems of the parent problem and the tightening may reduce the size of their individual solution spaces.

Since the original problem is too difficult to solve directly, at each node a relaxation of the original problem is solved. This relaxation may enlarge the size of the node's solution space. Thus, at the root node, a relaxation of the problem is solved. At every other (non-leaf) node, a doubly-modified problem is solved; one that is simultaneously tightened and relaxed. The solution space of these doubly-modified problems contains extra solutions that are not in the solution space of the original problem and is missing solutions from the original problem as illustrated by the following example.

Consider the Carpaneto, Dell'Amico, and Toth (CDT) implementation of branch-and-bound search for the ATSP [5]. For this algorithm, the AP is used for the relaxation. The branching rule dictates forced inclusions and exclusions of arcs. Arcs that are not forced in this way are referred to as *free* arcs. The branching rule selects the cycle in the AP solution that has the fewest free arcs and each child node forces the exclusion of one of these free arcs. Furthermore, each child node after the first forces the inclusion of the arcs excluded by their elder siblings. More formally, given a parent node, let E denote its set of excluded arcs, I denote its set of included arcs, and $\{a_1, \ldots, a_t\}$ be the free arcs in the selected cycle. In this case, t children would be generated with the kth child having $E_k = E \cup \{a_k\}$ and $I_k = I \cup \{a_1 \ldots a_{k-1}\}$. Thus, child k is tightened by adding the constraints that the decision variables for the arcs in E are equal to zero and those for the arcs in I are equal to one. When child k is processed, the AP is solved with these additional constraints. The solution space of this doubly-modified problem is missing all of the tours containing an arc in E_k and all of the tours in which any arc in I_k is absent. However, it is enlarged by the addition of all the AP solutions that are not a single cycle, do not contain an arc in E_k, and contain all of the arcs in I_k.

The CDT algorithm is experimentally compared with cut-and-solve in the Results section of this paper.

2.3 Gomory cuts

In the late fifties, Gomory proposed a linear search strategy in which *cutting planes* were systematically derived and applied to a relaxed problem [17]. 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 ($0 \leq x_i \leq 1$ 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 relaxed problem (such as $x_1 = 0.5$, $x_2 = 0.25$, and $x_3 = 0.5$).

Gomory cuts tighten the relaxed problem by removing part of its solution space. These cuts do not tighten the unrelaxed problem, as none of the solutions to the unrelaxed problem are removed. However, the removal of relaxed solutions tends to increase the likelihood that the next relaxed solution found is also a solution to the unrelaxed problem. Such solutions may be used to establish or update the incumbent solution. Cuts are added and relaxations are solved iteratively until the tightening on the relaxed problem becomes so constrictive that its solution is greater than or equal to the current incumbent. At this point, the search is terminated and the incumbent solution is declared optimal.

Although Gomory's algorithm only requires solving a series of relaxed problems, it was found to be inefficient in practice and fell into disuse [3].

2.4 Branch-and-cut search

Branch-and-cut search is essentially branch-and-bound search with the addition of the application of cutting planes at some or all of the nodes. These cutting planes tighten the relaxed problem and increase the pruning potential in two ways. First, the value of the solution to this subproblem may be increased (and cannot be decreased) by this tightening. If such increase causes the value to be greater than or equal to the incumbent solution value, then the entire subtree can be pruned. Second, forcing out a set of the relaxed solutions may increase the possibility that a feasible solution to the unrelaxed problem is found. If this feasible solution has a value that is less than the current incumbent solution, it will replace this incumbent and increase the pruning potential.

The number of nodes at which cutting planes are applied is algorithm-specific. Some algorithms only apply the cuts at the root node, while others apply cuts at many or all of the nodes.

`Concorde` [1,2] is a branch-and-cut algorithm designed for solving the symmetric TSP (STSP). (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.) This code has been used to solve STSP instances with as many as 15,112 cities [2]. This success was made possible by the design of a number of clever cutting planes custom tailored for this problem.

2.5 Branch-and-bound and branch-and-cut design considerations

When designing an algorithm using branch-and-bound or branch-and-cut, a number of policies must be determined. These include determining a relaxation of the original problem and an algorithm for solving this relaxation, branching rules, and a search strategy, which determines the order in which the nodes are explored.

Since a relaxed problem is solved at every node, it must be substantially easier to solve than the original problem. However, it is desirable to use the tightest relaxation possible in order to increase the potential for pruning.

Branching rules determine the structure of the search tree. They determine the depth and breadth of the tree. Moreover, branching rules tighten the subproblem. Thus, strong branching rules can increase pruning potential.

Finally, a search strategy must be selected. *Best-first* search selects the node with the best heuristic value to be explored first. This strategy ensures that the least number of nodes are explored for a given search tree and heuristic. Unfortunately, identifying the best current node requires storing all active nodes and even today's vast memory capabilities can be quickly exhausted. For this reason, *depth-first* search is commonly employed. While this strategy solves the memory problem and is asymptotically optimal [40], it introduces a substantial new problem. Heuristics used to guide the search can lead in the wrong direction, resulting in large subtrees being fruitlessly explored.

Unfortunately, even when a combination of policies is fine-tuned to get the best results, many problem instances remain insoluble. This is usually due to inadequate pruning. On occasion, the difficulty is due to the complexity of solving the relaxed problem or finding cutting planes. For instance, the simplex method is commonly used for solving the relaxation for mixed-integer linear programs, despite the fact that it has an exponential worst-case performance.

3 Cut-and-Solve Search Strategy

Unlike the cutting planes in branch-and-cut search, cut-and-solve uses piercing cuts that intentionally cut out solutions from the original solution space. We use the term *piercing cut* to refer to a cut that removes at least one feasible solution from the original (unrelaxed) problem solution space. The cut-and-solve algorithm is presented in Algorithm 1.

Algorithm 1 cut_and_solve (LP)

```
1: lowerbound ← $-\infty$
2: upperbound ← $\infty$
3: while (lowerbound < upperbound) do
4:    lowerbound ← solve_relaxed(LP)
5:    if (lowerbound $\geq$ upperbound) then
6:       break
7:    cut ← select_piercing_cut(LP)
8:    new_solution ← find_optimal(cut)
9:    if (new_solution < upperbound) then
10:      upperbound ← new_solution
11:   add_cut(LP, cut)
12: return upperbound
```

Each iteration of the while loop corresponds to one node in the search path. First a relaxed problem is solved (`solve_relaxed(LP)`). Then a set of solutions are selected (`select_piercing_cut(LP)`). Let SS_{sparse} be this set of solutions. SS_{sparse} is selected in a way that it will contain the optimal solution of the relaxed problem and at least one feasible solution from the original solution space, SS_o.

Next, a sparse problem (`find_optimal(cut)`) is solved, finding the best solution from SS_{sparse} that is also a feasible solution for the original problem. In other words, the best solution in $SS_{sparse} \cap SS_o$ is found. This problem tends to be relatively easy to solve as a sparse solution space, as opposed to the vast solution space of the original problem, is searched for the best solution. At the root node, this solution is made the incumbent solution. If later iterations find a solution that is better than the incumbent, this new solution becomes the incumbent.

Finally, a piercing cut is added to the LP. This piercing cut excludes all of the solutions in SS_{sparse} from the LP. Thus, the piercing cut tightens the LP and reduces the size of its solution space. Furthermore, since the solution of the relaxed problem is in SS_{sparse}, that solution cannot be returned by `solve_relaxed(LP)` on any other iterations.

At subsequent nodes, the process is repeated. The call to `solve_relaxed(LP)` is actually a doubly-modified problem. The LP has been tightened by the piercing cuts and a relaxation of this tightened problem is solved. The incumbent solution is updated as needed after the call to `find_optimal(cut)`. The piercing cuts accumulate with each iteration. When the tightening due to these piercing cuts becomes constrictive enough, the solution to this doubly-modified problem will become greater than or equal to the incumbent solution value. When this occurs, the incumbent solution is returned as optimal.

Theorem 1 *When the cut-and-solve algorithm terminates, the current incumbent solution must be an optimal solution.*

Proof The current incumbent is the optimal solution for all of the solutions contained in the piercing cuts. The solution space of the final doubly-modified problem contains all of the solutions for the original problem except those in the piercing cuts solution space. If the relaxation of this problem has a value that is greater than or equal to the incumbent value, then the solution space of this doubly-modified problem cannot contain a solution that is better than the incumbent. □

Termination of the algorithm is summarized in the following theorem:

Theorem 2 *If the solution space for the original problem, SS_o, is finite and the relaxation algorithm and the algorithm for selecting and solving the sparse problem are complete, then the cut-and-solve algorithm is complete.*

Proof The number of nodes in the search path must be finite as a non-zero number of solutions are removed from SS_o at each node. Therefore there are a finite number of complete problems solved. □

The cut-and-solve algorithm is easily adapted to be an *anytime* algorithm.

Anytime algorithms allow the termination of an execution at any time and return the best approximate solution that has been found thus far. If time constraints allow the execution to run to completion, then the optimal solution is returned. Since an incumbent solution is found at the first node, there is an approximate solution available any time after the root node is solved, and this solution improves until the optimum is found or the execution is terminated.

4 A Simple Example

In this section, we present a simple example problem and step through the search process using cut-and-solve. Consider the following LP:

$$Z = min\left(y - \frac{4}{5}x\right) \tag{9}$$

subject to:

$$x \geq 0 \tag{10}$$

$$y \leq 3 \tag{11}$$

$$y + \frac{3}{5}x \geq \frac{6}{5} \tag{12}$$

$$y + \frac{13}{6}x \leq 9 \tag{13}$$

$$y - \frac{5}{13}x \geq \frac{1}{14} \tag{14}$$

$$x \in \{\ldots, -1, 0, 1, \ldots\} \tag{15}$$

$$y \in \{\ldots, -1, 0, 1, \ldots\} \tag{16}$$

This LP has only two decision variables, allowing representation as a 2D graph as shown in Figure 1. Every x, y pair that is feasible must obey the constraints. Each of the first five linear constraints (10) to (14) corresponds to an edge of the polygon. All of the feasible solutions must lie inside or on the edge of the polygon in Figure 1. The constraints (15) and (16) require that the decision variables assume integral values. Therefore, the feasible solutions for this LP are shown by the dots located inside and on the edge of the polygon.

The terms of the objective function (9) can be rearranged into the slope-intercept form as follows: $y = \frac{4}{5}x + Z$. Therefore, the objective function of

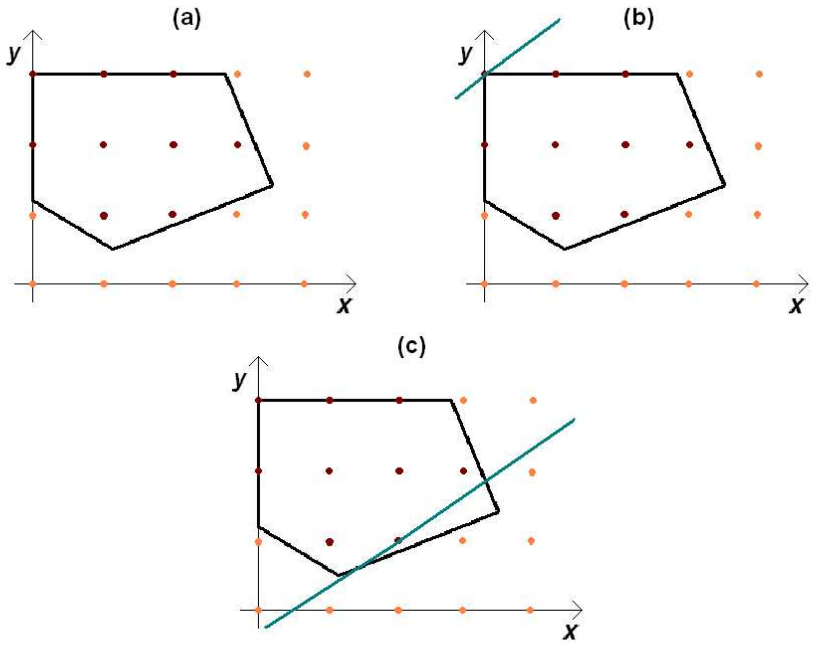

Fig. 1. Graphical solution of the example LP. (a) The feasible solution space. Dots within and on the polygon represent all of the feasible solutions. (b) One of the lines representing the objective function. It intersects the feasible solution $x = 0$, $y = 3$. (c) The optimal solution of $x = 2$ and $y = 1$.

this LP represents an infinite number of parallel lines with the slope of $\frac{4}{5}$. In this example, each value of Z is equal to its corresponding y-intercept. (For general 2D LPs, the y-intercept is equal to a constant times Z.) Figure 1(b) shows one of the lines in this family. The feasible solution at this point has an x value of zero and a y value of 3, yielding a Z value of 3. Clearly, this is not the optimal solution. By considering all of the lines with a slope of $\frac{4}{5}$, it is apparent that the line with the smallest y-intercept value that also intersects a feasible solution will identify the optimal Z value. This line can be found by inspection, and is shown in Figure 1(c). The optimal solution is $x = 2$ and $y = 1$, yielding $Z = -0.6$.

For this LP, the solution space of the original problem, SS_o, contains the nine points that lie within and on the polygon. To apply cut-and-solve, a relaxation must be chosen. In this example, the relaxation ignores constraints (15) and (16). The solution space for this relaxed problem, SS_r, contains every point of real values of x and y inside and on the polygon - an infinite number of solutions.

Figure 2 shows the steps taken when solving this LP using cut-and-solve. First, the relaxed solution is found. The solution to this relaxation is shown in Figure 2(a). The value of x is 3.5, y is 1.4, and the solution value of this relaxed subproblem is equal to -1.4. This is a lower bound on the optimal solution value and `lowerbound` in Algorithm 3 is set to -1.4.

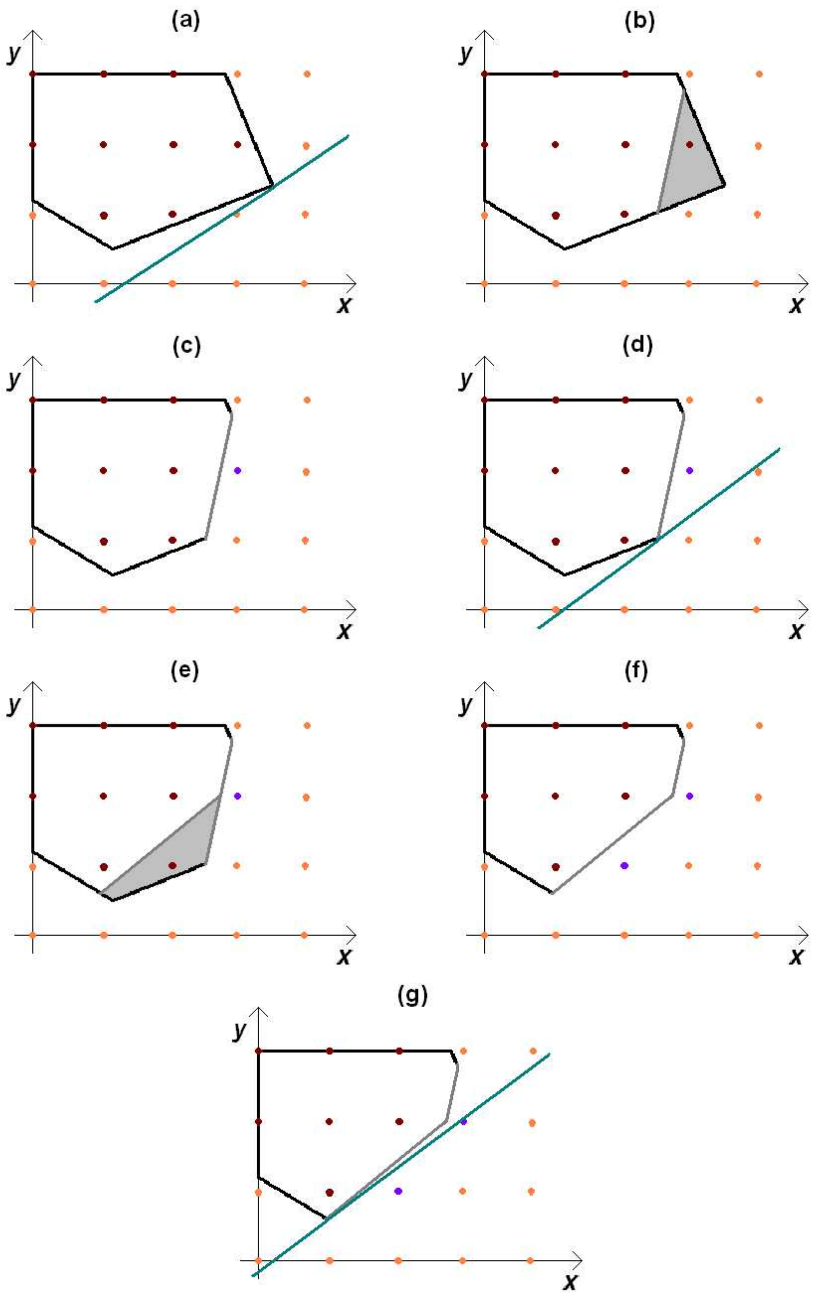

Fig. 2. Solving the example LP using cut-and-solve. (a) The relaxed solution of the first iteration. (b) The first piercing cut. (c) The solution space of the LP with the piercing cut added. (d) The relaxed solution of the second iteration. (e) The second piercing cut. (f) The solution space of the LP with both piercing cuts added. (g) The relaxed solution of the third iteration, resulting in a value that is worse than the current incumbent.

Next, a piercing cut is selected as shown in Figure 2(b). The shaded region contains the solution to the relaxed problem as well as a feasible solution to the original problem. It also contains an infinite number of feasible solutions to the relaxed problem. This sparse problem is solved to find the best *integral* solution. There is only one integral solution in this sparse problem, so it is the best. Thus the incumbent solution is set to $x = 3$, $y = 2$, and the objective function value of the incumbent is -0.4. `upperbound` in Algorithm 3 is set to -0.4.

The line that cuts away the shaded region from the polygon in Figure 2(b) can be represented by a linear constraint: $y - \frac{17}{3}x \geq -14$. This constraint is added to the LP. Now the feasible region of the LP is reduced and its current solution space contains eight points as shown in Figure 2(c). This completes the first iteration of the while loop in Algorithm 3. Thus, the root node has been solved and the next node in the search path is now explored.

This iteration begins by solving the relaxation of the current LP as shown in Figure 2(d). `lowerbound` is set to the value of this relaxed solution, -1.1. Notice that `lowerbound` is less than `upperbound`, so the search continues.

Next, we select a piercing cut as shown in Figure 2(e). The sparse problem represented by the shaded area in Figure 2(e) is solved yielding a value of -0.6. This value is less than the current incumbent, so this solution becomes the new incumbent and `upperbound` is set to -0.6. Notice that `upperbound` is still greater than `lowerbound`, so the search continues.

The linear constraint corresponding to the current piercing cut is added to the LP, resulting in a reduced feasible solution space as shown in Figure 2(f). This completes the second iteration.

The third node in the search path is now explored. The relaxed problem is solved on the current LP as shown in Figure 2(g). `lowerbound` is set to the value of this relaxed solution, which is -0.2. This value is greater than the `upperbound` of -0.6, so the search is finished.

The current incumbent must be an optimal solution. The incumbent is the best solution in the union of the solution spaces of all of the sparse problems that have been solved. `lowerbound` is a lower bound on the best possible solution that is in the remaining solution space. Since it is greater than the incumbent, there cannot be a solution in this space that is better than the incumbent. Therefore, optimality is ensured.

The method used to generate piercing cuts is problem specific. In general, these cuts should attempt to cut away optimal solutions. Furthermore, care should be taken to avoid cutting away solution spaces that are too large to be solved relatively easily.

The example problem presented in this section is easily solved by inspection. However, problems of interest may contain many thousands of decision variables and have solution spaces that are defined by convex polyhedrons in a correspondingly high-dimensional space. The next section demonstrates the use of cut-and-solve on such a problem.

5 Cutting Traveling Salesmen Down to Size

We have implemented the cut-and-solve algorithm for solving real-world instances of the ATSP. The ATSP can be used to model a host of planning, routing, and scheduling problems in addition to a number of diverse applications as noted in Section 1. Many of these real-world applications are very difficult to solve using conventional methods and as such are good candidates for this alternative search strategy. Our code is available at [8].

We use the Held-Karp lower bound for our relaxation as it is quite tight for these types of instances [26]. A parameter, α, is set to a preselected value. Then arcs with *reduced costs* less than α are selected and a sparse graph composed of these arcs is solved. (A reduced cost value is generated for each arc when the Held-Karp relaxation is calculated. This value represents a lower bound on the increase of the Held-Karp value if the arc were forced to be included in the optimal Held-Karp solution.) The best tour in this sparse graph becomes our first incumbent solution. The original problem is then tightened by adding the constraint that the sum of the decision variables for the selected set of arcs is less than or equal to $n-1$, where n is equal to the number of cities. This is our piercing cut. If all of the arcs needed for an optimal solution are present in the selected set of arcs, this solution will be made the incumbent. Otherwise, at least one arc that is not in this set is required for an optimal tour. This constraint is represented by the piercing cut.

The process of solving the Held-Karp lower bound, solving a sparse problem, and adding a piercing cut to the problem repeats until the Held-Karp value of the deeply-cut problem is greater than or equal to the incumbent solution. At this point, the incumbent must be an optimal tour.

The worst-case complexities of solving the Held-Karp lower bound (using the simplex method) and solving the sparse problem are both exponential. However, in practice these problems are usually relatively easy to solve.

Selection of an appropriate value for α is dependent on the distribution of reduced cost values. In our current implementation, we simply select a number of arcs, m_{cut}, to be in the initial cut. At the root node, the arcs are sorted by their reduced costs and the m_{cut} lowest arcs are selected. α is set equal to the maximum reduced cost in this set. At subsequent nodes, α is used to determine the selected arcs. The choice of the value for m_{cut} is dependent on the problem type and the number of cities. We believe that determining α

directly from the reduced costs would enhance this implementation as *a priori* knowledge of the problem type would not be necessary and α could be custom tailored to suit variations of instances within a class of problems.

If a cut does not contain a single feasible solution, it can be enlarged to do so. However, in our experiments this check was of no apparent benefit. The problems were solved after traversing no more than three nodes in all cases, so guaranteeing completeness was not of practical importance.

We use `cplex` [24] to solve both the relaxation and the sparse problem. All of the parameters for this solver were set to their default modes. For some of the larger instances, this generic solver becomes bogged down while solving the sparse problem. Performance could be improved by the substitution of an algorithm designed specifically for solving sparse ATSPs. We were unable to find such code available. We are investigating three possible implementations for this task: (1) adapting a Hamiltonian circuit enumerative algorithm to exploit ATSP properties, (2) using a dynamic programming approach to the problem, or (3) enhancing the `cplex` implementation by adding effective improvements such as the Padberg and Rinaldi shrinking procedures, external pricing, cutting planes customized for sparse ATSPs, heuristics for node selection, and heuristics for determining advanced bases. However, despite the crudeness of our implementation, it suffices to demonstrate the potential of the cut-and-solve method. We compare our solver with two branch-and-bound and two branch-and-cut implementations in the next section.

6 Computational Results for the ATSP

In this section, we compare our cut-and-solve implementation (`CZ-c&s`) with the four ATSP solvers that are compared in *The Traveling Salesman Problem and its Variations* [18,14]. Our testbed consists of all of the 27 ATSP instances in `TSPLIB` [38] and six instances of each of seven real-world problem classes as introduced in [6] and used for comparisons in [14].

Our code was run using `cplex` version 8.1. We used Athlon 1.9 MHz dual processors with two gigabytes shared memory for our tests. In order to identify subtours in the relaxed problem, we use Matthew Levine's implementation of the Nagamochi and Ibaraki minimum cut code [32], which is available at [33].

In the experiments presented here, we found that the search path was quite short. Typically only one or two sparse problems and two or three relaxed problems were solved. This result indicates that the set of arcs with small reduced costs is likely to contain an optimal solution and that the Held-Karp relaxation is tight.

We make comparisons with two branch-and-bound implementations - the Carpaneto, Dell'Amico, and Toth (`CDT`) and the Fischetti and Toth additive (`FT-add`) algorithms; and two branch-and-cut implementations - the Fischetti and Toth branch-and-cut (`FT-b&c`) and `concorde` algorithms.

`Concorde` [1,2] is an award-winning code used for solving symmetric TSPs (STSPs). ATSP instances can be transformed into STSP instances using a *2-node* transformation [29]. 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. For consistency, we use the same transformation parameters and set `concorde`'s random seed parameter and chunk size as done in [14].

We did not run the `CDT`, `FT-add`, and `FT-b&c` codes on our machine as the code was not available. The comparisons are made by *normalizing* both the results in [14] and our computation times according to David Johnson's method [27] (see also [25]). The times are normalized to approximate the results that might be expected if the code were run on a Compaq ES40 with 500-MHz Alpha processors and 2 Gigabytes of main memory. As described in [27], these normalized time comparisons are subject to multiple sources of potential inaccuracy. Furthermore, low-level machine-specific code tuning and other speedup techniques can compound this error. For these reasons, it is suggested in [27] that conclusions about relative performance with differences less than an order of magnitude may be questionable.

A substantial normalization error appears in our comparisons. Tables 1 and 2 show the comparisons of normalized computation times for the four implementations compared in [14] and run on their machine along with the normalized times for `concorde` and `CZ-c&s` run on our machine. Comparing the normalized times for `concorde` for the two machines, we see that the normalization error consistently works against us - in several cases there is an order of magnitude difference. `Concorde` requires the use of `cplex` [24], for its LP solver. We used `cplex` version 8.1 while [14] used version 6.5.3. Assuming that `cplex` has not gotten substantially slower with this newer version, we can speculate the normalization error is strongly biased against us. We suspect this error may be due to the significant differences in machine running times. For instance, for 100-city instances the normalization factor for our machine is 5.0, while it is 0.25 for the Fischetti, Lodi, and Toth machine.

The CDT implementation performs well for many of the TSPLIB instances and for most of the computer disk read head (`disk`), no-wait flowshop (`shop`), and shortest common super string (`super`) problems. Unfortunately, the code is not robust and fails to solve 45% of the instances within the allotted time (1,000 seconds on the Fischetti, Lodi, and Toth machine).

The `FT-add` implementation behaves somewhat like the previous branch-and-bound search. It performs fairly well for most of the same instances that `CDT` performs well on and fails to solve almost all of the same instances that are missed by `CDT`. (`FT-add` fails to solve 36% of the instances.)

Both `FT-b&c` and `CZ-c&s` behave more robustly than the branch-and-bound algorithms. They solve all of the TSPLIB instances but fail to solve the 316-city instances of the coin collection (`coin`), stacker crane (`crane`), and one of the tilted drilling machine (`stilt`) instances. `CZ-c&s` also fails to solve the other tilted drilling machine instance (`rtilt`).

Although `FT-b&c` consistently has better normalized running times than `CZ-c&s`, the normalization error bias should be considered. One comparison that is not machine dependent is the ratio of each algorithm's time to their corresponding time for `concorde`. We calculate these ratios by summing the run times for all instances except the 316-city instances of `coin`, `crane`, `rtilt`, and `stilt` as these instances are not run to completion for all of the algorithms. `crane` runs to completion only for `concorde` and `rtilt` does not run to completion for `CZ-c&s`. `rtilt` does run to completion for `FT-b&c`, using about 89% of the allotted time. `FT-b&c` solves all of the other instances in 41.1% of the time required by `concorde` on the same machine and `CZ-c&s` solves the instances in 29.7% of the time required by `concorde` on the same machine. (Note that `cplex` version 6.5.3 is used on the former machine, while version 8.1 is used on the latter.) This comparison is not scientific. However, it gives a vague sense of how these algorithms might compare without the normalization error bias.

Finally, we compare `concorde`, `CZ-c&s`, and our own implementation of `CDT` for 100-city instances of the seven problem classes and average over 100 trials. These comparisons were all run on our machine, so there is no normalization error. We varied the degree of accuracy of the arc costs by varying the number of digits used for the generator parameter. The relationship between the degree of accuracy of arc costs and computation time to solve random ATSPs is discussed in [39]. In general, a smaller generator parameter corresponds to a greater number of optimal solutions.

There is no graph for the `super` class as it is not dependent upon the generator parameter. The average normalized time to solve these instances using `CDT` is 0.073 seconds, while `concorde` required 8.15 seconds and `CZ-c&s` took 2.07 seconds.

The average normalized computation times and the 95% confidence intervals for the other classes are shown in Figures 3 and 4. The confidence intervals are large as might be expected due to the heavy-tailed characteristics of the ATSP.

The `CDT` algorithm performed extremely well for the `shop` and `super` instances.

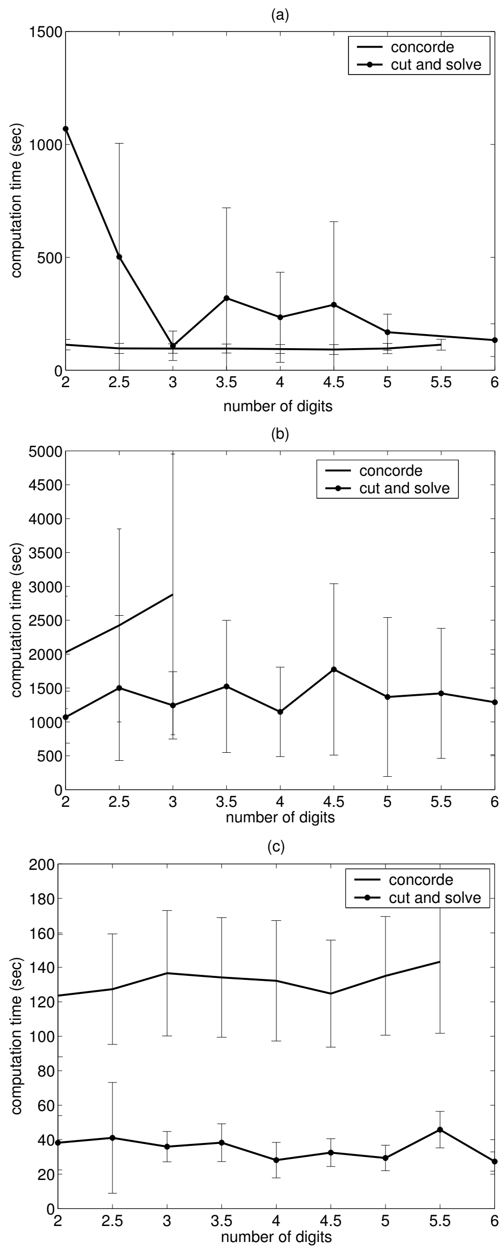

Fig. 3. Average normalized computation times with 95% confidence interval shown. (a) `rtilt` problem class. (b) `stilt` problem class. (c) `crane` problem class.

However, it failed to complete any of the other tests. Although `CDT` performed well for five of the six `disk` instances in the testbed (including the 316-city instance), it failed to solve 100 of the `disk` instances for any of the parameter settings. We allowed 20 days of normalized computation time for each parameter setting; indicating that the average time would be in excess of 17,000 seconds.

The missing data points for the `concorde` code are due to the implementation terminating with an error. Although `CZ-c&s` performed better than `concorde`

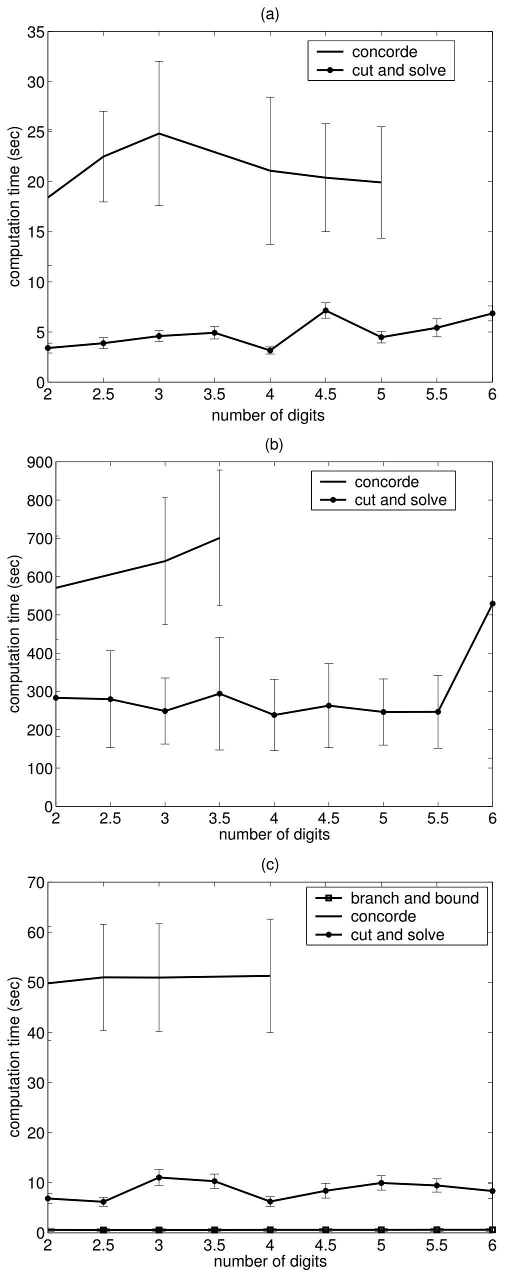

Fig. 4. Average normalized computation times with 95% confidence interval shown. (a) `disk` problem class. (b) `coin` problem class. (c) `shop` problem class.

for the five 100-city `rtilt` instances in the testbed, on average, it does not perform as well as `concorde` for the `rtilt` class in this set of tests. However, it outperforms `concorde` for all of the other problem classes.

In conclusion, our implementation of the cut-and-solve strategy for the ATSP appears to be more viable than the two branch-and-bound solvers. It is difficult to make decisive comparisons with `FT-b&c` due to normalization errors, however, our implementation appears to generally perform better than `concorde` for these asymmetric instances.

7 Discussion and Related Work

Many optimization problems of interest are difficult to solve to optimality, so approximations are employed. Sacrificing optimality can be costly when the solution is an expensive procedure (such as routing a spacecraft), the solution is reused a number of times (as in manufacturing applications), or the approximation is poor. Cut-and-solve offers an alternative to branch-and-bound and branch-and-cut when attempting to optimally solve these important problems.

Search tree methods such as branch-and-bound and branch-and-cut must choose between memory problems or the problem of fruitlessly searching subtrees containing no optimal solutions. Cut-and-solve is free from these difficulties. 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. Furthermore, being a linear search, there are no subtrees in which to get lost.

When designing a cut-and-solve algorithm, the selections of a relaxation algorithm and a method for selecting piercing cuts are subject to trade-offs and should be chosen carefully. The relaxation algorithm should be tight and the cuts should try to capture optimal solutions. Yet, to be efficient, both problems need to be relatively easy to solve.

One concern about the viability of cut-and-solve might be that adding extra constraints might tend to make the problem more difficult to solve. On the contrary, we may expect the opposite to be true in general. Consider branch-and-bound search. At the root node, the problem is subdivided into subproblems, one for each child node. Presumably each of these subproblems are easier to solve than the original problem. The difference between the children and the root is that the child nodes each has one or more additional constraints. For instance, the value of a variable may be set to a particular number. As the search proceeds down the tree, the children accumulate more and more constraints. If no pruning occurs along a path, the leaf node is so highly constrained that there exists only a single feasible solution. Since adding constraints reduces the size of the solution space, they tend to make the problem easier to solve.

Another concern about cut-and-solve might be that choosing a poor cut may be similar to choosing the wrong path in a depth-first search tree. These two actions are very different. When choosing the wrong child in depth-first search, the decision is permanent for all of the descendants in the subtree. For example, if a variable appears in every optimal solution (a *backbone* variable [35]), then setting its value to not be included is calamitous. Conversely, forcing the inclusion of a variable that doesn't appear in any feasible solution (a *fat* variable [7]) results in a similar dire situation. Every node in the subtree is doomed. On the other hand, when a poor cut is chosen in cut-and-solve, the only consequence is that the time spent exploring that single node was relatively ineffective. It wasn't completely wasted as even a poor cut helps to tighten the problem to some degree and once it is made, this cut cannot be repeated in future iterations. Moreover, subsequent nodes are not "locked in" by the choice made for the current cut.

7.1 Related work

When compared to depth-first branch-and-bound search, the main advantage of cut-and-solve is that it cannot get lost in subtrees void of any optimal solutions. A great number of techniques have been devised in an effort to overcome this problem for depth-first branch-and-bound. Among these are iterative deepening [30], limited discrepancy [19], and randomization and restarts [16].

Iterative deepening search performs a number of depth-first searches, each with a limit on the depth of the search. After each iteration, the depth is increased until an optimal solution is reached. Limited discrepancy search allows a limited number of discrepancies away from the heuristic choice in a traversal of the tree. At each iteration, the number of discrepancies is increased until an optimal solution is found. The randomization and restarts technique has two prominent components. The next node to explore is randomly chosen from the nodes with high heuristic values and the search is restarted from the root after a specified number of backtracks have been taken. Completeness is ensured by the use of linear-space bookkeeping and the allowable number of backtracks is gradually increased as the search is conducted.

We now discuss several algorithms that are similar to cut-and-solve.

Cut-and-solve is similar to Gomory's algorithm in that cuts are used to constrain the problem and a linear path is searched. Gomory's algorithm is sometimes referred to as "solving the problem at the root node" as this is essentially its behavior when comparing it to branch-and-cut. However, cutting planes are applied and relaxed problems are solved in an iterative manner until the search is terminated, suggesting a linear progression of the search.

The major difference between the two methods is that Gomory's cuts are not *piercing* cuts as they do not cut away any feasible solutions to the original problem.

Cut-and-solve can be thought of as an extension of Gomory's method. Like Gomory's technique, cuts are applied, one at a time, until an optimal solution is found. For each cut, a relaxation of the current tightened problem is solved. This value is used to determine when the search can be terminated. Cut-and-solve cuts are deeper than Gomory cuts as they are not required to trim around feasible solutions for the unrelaxed problem. Cutting deeply yields two benefits. First, it provides a set of solutions from which the best one is chosen for a potential incumbent. Second, these piercing cuts tighten the relaxed problem in an aggressive manner, and consequently tend to increase the solution of the doubly-modified problem. Once this solution meets or exceeds the incumbent value, the search is finished.

Cut-and-solve is also similar to an algorithm for solving the Orienteering Problem (OP) as presented in [15]. In this work, *conditional cuts* remove feasible solutions to the original problem. These cuts are used in conjunction with more traditional cuts and are used to tighten the problem. When a conditional cut is applied, an enumeration of all of the feasible solutions within the cut is attempted. If the enumeration is not solved within a short time limit, the cut is referred to as a *branch cover cut* and the sparse graph associated with it is stored. This algorithm attempts to solve the OP in a linear fashion, however, due to tailing-off phenomena, branching occurs after every five branch cover cuts have been applied. After this branch-and-cut tree is solved, a second branch-and-cut tree is solved over the union of all of the graphs stored for the branch cover cuts.

Cut-and-solve differs from this OP algorithm in several ways. First, incumbent solutions are forced to be found early in the cut-and-solve search. These incumbents provide useful upper bounds as well as improve the *anytime* performance. Second, the approach used in [15] stores sparse problems and combines and solves them as a single, larger problem after the initial cut-and-solve tree is explored. Finally, this OP algorithm is not truly linear as branching is allowed.

In a more broad sense, cut-and-solve shares similarities with divide-and-conquer [36] and a binary tree search with highly disproportionate children. We discuss each of these in turn.

Cut-and-solve is similar to divide-and-conquer in that both techniques identify small subproblems of the original problem and solve them. While divide-and-conquer solves all of these subproblems, cut-and-solve solves a very small percentage of them. If solving all of the subproblems is feasible, then divide-and-conquer would probably be the method of preference. However, when divide-and-conquer proves to be insoluble, cut-and-solve may be a potential alternative. When comparing these two techniques, cut-and-solve might be thought of as divide-and-conquer with powerful pruning rules.

Finally, cut-and-solve is similar to binary tree search with highly disproportionate children. At each node, a relaxation is solved and a cut is chosen. This cut is used for the branching rule. The left child has a huge subtree and the right has a very small subtree. The right child's subtree is easily explored and produces potential incumbent solutions. It is immediately solved and the left child is subdivided into another set of disproportionate children. When the relaxed solution is greater than or equal to the current incumbent, the huge subtree is pruned and the search is finished. Thus, the final configuration of the search tree is a single path at the far left side with a single, small subtree branching to the right at each level. This search strategy is essentially linear: consistently solving the easy problems immediately and then redividing.

7.2 *Is cut-and-solve truly "linear"?*

This is a debatable question. We use the term "linear search" as this is the structure of the search space when viewed at a high level. In general, the algorithms used for solving subproblems within a search algorithm are not relevant in identifying the search strategy. For example, branch-and-bound search is not defined by the algorithms used for solving the relaxed problem, identifying potential incumbent solutions, or guiding the search, as these algorithms are of a number of varieties and could themselves be solved by branch-and-bound or some other technique such as divide-and-conquer.

However, as pointed out by Matteo Fischetti, one could write a paper with a "Null search" strategy, in which a single subproblem is solved by calling a black-box ATSP solver. It appears that the pertinent question here is whether the "subproblems" that are solved in cut-and-solve are truly subproblems.

Let us consider the subproblems that are solved in the `FT-b&c` algorithm. At each node, a number of cutting planes are derived and applied, the relaxation is iteratively solved, and a sparse problem is solved every time the subtour elimination constraints are not in violation. The sparse problem is solved by enumerating the Hamiltonian circuits. This procedure is terminated if the number of backtracking steps exceeds $100 + 10n$. The number of iterations performed at each node is also limited by terminating when the lower bound does not increase for five consecutive iterations. The relaxations are solved using the simplex method, despite the fact that it has an exponential worst-case running time. The ellipsoid method could be used to solve the relaxation with a polynomial worst-case time, however, this method tends to run quite slow [28]. In fact, the simplex method is commonly used for solving the Held-Karp relaxation as, in practice, it is expected to run efficiently.

Two subproblems, a relaxation and a sparse problem, are solved at each node in the `CZ-c&s` algorithm. Like the `FT-b&c` algorithm, the relaxation is solved using simplex. The sparse problem is solved using `cplex` and we cannot assert that it is a substantially easier problem to solve than the original. However, for "difficult" problems in which a considerable amount of the solution space is explored, in practice we might expect that finding the best solution in a small chunk of the solution space is substantially easier than finding the optimal solution in the entire space. (Furthermore, after the first sparse problem is solved, subsequent sparse problems have the advantage of having an upper bound provided by the incumbent solution.) In our experiments, solving the first sparse problem tended to be the bottleneck. However, as indicated by the performance of other algorithms, the time spent solving the sparse problem tends to be substantially less than the time required to solve the entire problem outright. For these reasons, we (cautiously) refer to cut-and-solve as a "linear" search.

7.3 Is this method applicable to other problems?

It appears that cut-and-solve might be applicable to other optimization problems, including Integer Linear Programs. Hypothetically speaking, the method may be applied to virtually any optimization problem. However, there are four requirements that are apparently necessary for any hope of this method being useful. First, there must be an efficient algorithm available for finding a tight relaxation of the problem. Second, an efficient algorithm is also needed for solving the sparse problem. Third, a strategy must be devised for easily identifying succinct cuts that tend to capture optimal solutions. Finally, it appears that this method works best for problems that are otherwise difficult to solve. In these cases, solving sparse problems can be considerably easier than tackling the entire problem at once.

8 Conclusions

In this paper, we present a search strategy which we refer to as cut-and-solve. We show that optimality and completeness are guaranteed despite the fact that no branching is used. Being a linear strategy, this technique is immune to some of the pitfalls that plague search tree methods such as branch-and-bound and branch-and-cut. Memory requirements are nominal, as only the incumbent solution and the current tightened problem need be saved as the search path is traversed. Furthermore, there is no need to use techniques to reduce the risks of fruitlessly searching subtrees void of any optimal solution.

We demonstrate cut-and-solve for linear programs and have implemented this strategy for solving the ATSP. Comparisons with ATSP solvers in [14] have been thwarted by a substantial normalization error. However, by using `concorde` as a baseline, we are able to sense that our simple implementation is competitive with state-of-the-art solvers.

In the future, we plan to improve our implementation by designing a custom solver for sparse ATSPs.

Life is full of optimization problems and a number of search strategies have emerged to tackle them. Yet, many of these problems stubbornly defy resolution by current methods. It is our hope that the unique characteristics of cut-and-solve may prove to be useful when addressing some of these interesting problems.

Acknowledgments

This research was supported in part by NDSEG and Olin Fellowships, by NSF grants IIS-0196057 and ITR/EIA-0113618, and in part by DARPA Cooperative Agreement F30602-00-2-0531. We thank three anonymous reviewers who identified weak areas of an earlier version of this paper [9] and provided valuable insights. We extend special thanks to Matteo Fischetti, who generously shared his experience and expertise in a number of discussions.

References

[1] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. TSP cuts which do not conform to the template paradigm. In M. Junger and D. Naddef, editors, *Computational Combinatorial Optimization*, pages 261–304. Springer, New York, N.Y., 2001.

[2] 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.

[3] E. Balas, S. Ceria, G. Cornuéjols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19:1–9, 1996.

[4] E. Balas and P. Toth. Branch and bound methods. In *The Traveling Salesman Problem*, pages 361–401. John Wiley & Sons, Essex, England, 1985.

[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] J. Cirasella, D.S. Johnson, L. McGeoch, and W. Zhang. The asymmetric traveling salesman problem: Algorithms, instance generators, and tests. In *Proc. of the 3rd Workshop on Algorithm Engineering and Experiments*, 2001.

[7] S. Climer and W. Zhang. Searching for backbones and fat: A limit-crossing approach with applications. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02)*, pages 707–712, Edmonton, Alberta, July 2002.

[8] S. Climer and W. Zhang. Cut-and-solve code for the ATSP. *http://www.climer.us* or *http://www.cse.wustl.edu/~zhang/projects/tsp/cutsolve.*, 2004.

[9] S. Climer and W. Zhang. A linear search strategy using bounds. In *14th International Conference on Automated Planning and Scheduling (ICAPS'04)*, pages 132–141, Whistler, Canada, June 2004.

[10] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 2:393–410, 1954.

[11] 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.

[12] R. Dechter and F. Rossi. Constraint satisfaction. In *Encyclopedia of Cognitive Science*. March 2000.

[13] W. L. Eastman. *Linear programming with pattern constraints.* PhD thesis, Harvard University, Cambridge, MA, 1958.

[14] 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.

[15] M. Fischetti, J. J. Salazar, and P. Toth. The generalized traveling salesman and orienteering problems. In G. Gutin and A. Punnen, editors, *The Traveling Salesman Problem and its Variations*, pages 609–662. Kluwer Academic, Norwell, MA, 2002.

[16] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 431–438, New Providence, RI, 1998.

[17] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, 64:275–278, 1958.

[18] G. Gutin and A. P. Punnen. *The Traveling Salesman Problem and its variations.* Kluwer Academic Publishers, Norwell, MA, 2002.

[19] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 1, Montreal, Canada, August 1995.

[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] K. L. Hoffman and M. Padberg. Improving LP-representations of zero-one linear programs for branch-and-cut. *ORSA Journal on Computing*, 3:121–134, 1991.

[24] Ilog. *http://www.cplex.com.*, web.

[25] D. S. Johnson. 8th DIMACS implementation challenge. *http://www.research.att.com/˜dsj/chtsp/.*, web.

[26] D. S. Johnson, G. Gutin, L. A. McGeoch, A. Yeo, W. Zhang, and A. Zverovich. Experimental analysis of heuristics for the ATSP. In G. Gutin and A. Punnen, editors, *The Traveling Salesman Problem and its Variations.* Kluwer Academic, Norwell, MA, 2002.

[27] D. S. Johnson and L. A. McGeoch. Experimental analysis of heuristics for the STSP. In G. Gutin and A. Punnen, editors, *The Traveling Salesman Problem and its Variations*, pages 369–443. Kluwer Academic, Norwell, MA, 2002.

[28] D. S. Johnson, L. A. McGeoch, and E. E. Roghberg. Asymptotic experimental analysis for the Held-Karp Traveling Salesman bound. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 341–350, 1996.

[29] R. Jonker and T. Volgenant. Transforming asymmetric into symmetric traveling salesman problems. *Operations Research Letters*, 2:161–163, 1983.

[30] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[31] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. *The Traveling Salesman Problem.* John Wiley & Sons, Essex, England, 1985.

[32] M. S. Levine. *Experimental study of minimum cut algorithms.* PhD thesis, Computer Science Dept., Massachusetts Institute of Technology, Massachusetts, May 1997.

[33] M. S. Levine. Minimum cut code. *http://theory.lcs.mit.edu/˜mslevine/mincut/index.html.*, web.

[34] S. Martello and P. Toth. Linear assignment problems. *Annals of Discrete Math.*, 31:259–282, 1987.

[35] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic ‘phase transitions’. *Nature*, 400:133–137, 1999.

[36] Z. G. Mou and P. Hudak. An algebraic model of divide-and-conquer and its parallelism. *Journal of Supercomputing*, 2:257–278, November 1988.

[37] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, MA, 1984.

[38] G. Reinelt. TSPLIB - A Traveling Salesman Problem library. *ORSA Journal on Computing*, 3:376–384, web. *http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95.*

[39] W. Zhang. Phase transitions and backbones of the asymmetric Traveling Salesman. *Journal of Artificial Intelligence Research*, 20:471–497, 2004.

[40] W. Zhang and R. E. Korf. Performance of linear-space search algorithms. *Artificial Intelligence*, 79:241–292, 1995.

	Fischetti, Lodi, & Toth machine				Climer & Zhang machine	
Name	CDT	FT-add	concorde	FT-b&c	concorde	CZ-c&s
br17	0.6	0.0	0.0	0.0	0.7	0.0
ft53	-	0.0	0.1	0.0	1.1	0.4
ft70	0.1	0.1	0.7	0.0	4.7	0.5
ftv33	0.0	0.0	0.1	0.0	0.5	0.0
ftv35	0.0	0.0	1.8	0.1	5.3	0.5
ftv38	0.0	0.1	2.9	0.1	14.2	0.5
ftv44	0.0	0.0	1.9	0.1	10.0	0.6
ftv47	0.0	0.1	4.9	0.1	35.7	1.0
ftv55	0.2	0.3	2.0	0.3	12.3	0.9
ftv64	0.2	0.3	4.6	0.6	49.8	1.5
ftv70	0.8	0.9	4.1	0.3	15.8	1.9
ftv90	0.2	0.6	3.7	0.1	18.9	1.4
ftv100	4.2	7.4	3.2	0.6	30.1	2.8
ftv110	1.3	10.0	6.7	1.9	18.2	4.9
ftv120	13.5	26.8	14.7	3.5	61.5	6.8
ftv130	1.7	4.6	4.6	0.4	36.4	4.6
ftv140	4.0	11.3	7.4	0.6	23.0	5.6
ftv150	0.9	5.1	8.1	0.8	21.3	5.9
ftv160	29.1	98.0	17.3	1.2	38.6	11.9
ftv170	-	-	13.0	1.3	31.7	20.0
kro124p	-	33.9	2.5	0.3	12.4	5.0
p43	-	-	4.8	2.0	23.2	6.1
rbg323	0.0	0.1	10.3	0.2	55.4	48.5
rbg358	0.0	0.2	13.2	0.2	23.8	72.8
rbg403	0.0	0.5	22.7	0.6	31.0	194.9
rbg443	0.0	0.6	16.6	0.7	33.9	155.3
ry48p	-	4.3	4.8	0.2	44.0	2.1

Table 1
Normalized CPU times (in seconds) for TSPLIB instances. Instances not solved in the allotted time are labeled by "-". The normalization error is strongly biased against the Climer & Zhang machine, so a machine-independent comparison is made using `concorde` as a baseline.

	Fischetti, Lodi, & Toth machine				Climer & Zhang machine	
`Name`	`CDT`	`FT-add`	`concorde`	`FT-b&c`	`concorde`	`CZ-c&s`
coin100.0	-	-	26.5	2.2	118.0	14.2
coin100.1	-	-	12.3	1.5	75.4	17.4
coin100.2	-	-	10.1	2.3	65.7	28.0
coin100.3	-	-	5.3	0.7	50.6	21.2
coin100.4	-	-	16.0	1.2	138.5	76.4
crane100.0	-	-	1.6	0.4	8.4	8.1
crane100.1	-	-	4.0	0.4	23.8	6.6
crane100.2	-	-	88.9	51.2	411.9	51.1
crane100.3	-	10.2	0.9	0.1	6.0	5.3
crane100.4	-	-	69.4	29.4	267.9	46.4
disk100.0	0.2	0.2	1.8	0.3	16.9	1.7
disk100.1	-	7.4	10.1	0.7	41.3	4.0
disk100.2	0.0	0.2	1.4	0.1	6.9	2.3
disk100.3	0.2	0.4	0.6	0.0	3.4	1.9
disk100.4	0.0	0.1	2.3	0.1	15.2	1.9
disk316.10	0.9	18.7	13.4	2.4	36.2	51.0
rtilt100.0	-	-	32.3	56.9	208.5	202.9
rtilt100.1	-	-	6.7	1.7	57.9	27.9
rtilt100.2	-	-	2.0	0.1	14.6	4.3
rtilt100.3	-	-	4.8	1.1	23.4	8.5
rtilt100.4	-	-	6.7	1.2	49.2	20.6
shop100.0	0.0	0.1	7.2	0.2	39.8	2.5
shop100.1	0.3	0.7	9.9	0.4	40.1	4.0
shop100.2	0.1	1.3	4.4	0.3	24.8	4.9
shop100.3	0.1	0.7	6.1	0.4	33.0	4.0
shop100.4	0.0	0.3	4.2	0.1	18.2	5.1
shop316.10	1.3	39.3	52.7	6.2	164.5	69.8
stilt100.0	-	-	131.1	12.3	741.6	84.2
stilt100.1	-	-	55.4	14.3	387.2	61.2
stilt100.2	-	-	14.2	1.5	59.0	39.8
stilt100.3	-	-	165.6	35.3	1147.4	535.0
stilt100.4	-	-	423.8	324.9	2454.3	200.2
super100.0	0.0	0.0	1.5	0.1	2.7	0.5
super100.1	0.0	0.1	2.1	0.1	6.4	1.0
super100.2	0.0	0.1	0.4	0.0	14.4	1.0
super100.3	0.0	0.1	0.9	0.1	10.7	1.5
super100.4	0.0	0.0	1.5	0.0	14.6	2.2
super316.10	-	102.0	6.4	1.7	42.2	53.1
coin316.10	-	-	-	-	-	-
crane316.10	-	-	1847.8	-	-	-
rtilt316.10	-	-	255.8	3830.3	80.7	-
stilt316.10	-	-	-	-	-	-

Table 2
Normalized CPU times (in seconds) for real-world problem class instances. Instances not solved in the allotted time are labeled by "-". The normalization error is strongly biased against the Climer & Zhang machine, so a machine-independent comparison is made using `concorde` as a baseline.