Cut-and-Solve: An Iterative 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 to combinatorial optimization problems. In this paper, we introduce an iterative search strategy which we refer to as *cut-and-solve* and prove optimality and termination 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 requirement is negligible. For these reasons, it has potential 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 a generic version of it for the Asymmetric Traveling Salesman Problem (ATSP). Our unoptimized implementation outperformed state-of-the-art solvers for five out of seven real-world problem classes of the ATSP. For four of these classes, cut-and-solve was able to solve larger (sometimes substantially larger) problems. 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: `sharlee@climer.us` (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* [28], 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 search tree is pruned to an adequately small size, the problem 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* [28] 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 [49]. 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. The Artificial Intelligence community has invested a great deal of effort in addressing this issue. Branching techniques [4], heuristics investigations [42], and search techniques such as limited discrepancy [24] and randomization and restarts [19] have been developed in an effort to combat this persistent problem.

In this paper, we introduce an iterative search strategy which overcomes the problem of making wrong choices in depth-first branch-and-bound, while keeping memory requirements nominal. We refer to this search strategy as *cut-and-solve* and demonstrate it on integer linear programs. Being an iterative strategy, 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. 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 updated as needed at subsequent nodes. When the search terminates, the current incumbent solution is guaranteed to be an optimal solution. In this paper, we prove optimality and termination 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 a generic procedure for using cut-and-solve is presented. This generic procedure is demonstrated 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 tested a preliminary, unoptimized implementation of this algorithm and compared it with branch-and-bound and branch-and-cut solvers. Our tests show that cut-and-solve is not always as fast as these state-of-the-art solvers for relatively simple problem instances. However, it is faster than these solvers on the largest instances for five out of seven real-world problem classes, and solves larger (sometimes substantially larger) instances for four of these classes. This paper is concluded with a discussion of this technique and related work. A preliminary version of this paper appeared in [11].

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 integer linear programs (IPs). An IP is an optimization problem that is subject to a set of linear constraints. IPs have been used to model a wide variety of problems, including Traveling Salesman Problems (TSP) [22,36], Constraint Satisfaction Problems (CSP) [15], and network optimization problems [45]. Moreover, a wealth of problems can be cast as one of these more general problems. TSP applications include 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 [30]. 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. CSPs are used to model configuration, design, diagnosis, spatio-temporal reasoning, resource allocation, graphical interfaces, and scheduling problems [15]. Finally, examples of network optimization problems include delay-tolerant network routing, cellular radio network base station locations, and the minimum-energy multicast problem in wireless ad hoc networks. There exist interesting research problems that can be cast as IPs in virtually every field of computer science. Furthermore, there are a staggering number of commercial applications that can be cast as IPs.

A general IP can be written in the following form:

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

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

$$x_i \in I \tag{3}$$

where the c_i values are instance-specific constants and the set of x_i represents the *decision variables*. Constraints (2) are linear equalities or inequalities composed of constants, decision variables, and possibly some auxiliary variables. Constraints (3) enforce integrality of the decision variables. A *feasible* solution is one that satisfies all of the constraints (2) and (3). The set of all feasible solutions is the *solution space*, *SS*, for the problem. The solution space is defined by the given problem. In contrast, the search space is defined by the algorithm used to solve the problem. An *optimal* solution is a feasible solution with the least (or greatest) value, as defined by the *objective function* (1).

Linear programs (LPs) are similar to IPs except the former allow real numbers for the decision variables. A general LP has the same form as the general IP given above with the omission of constraints (3). LPs can be solved in polynomial time using the ellipsoid method [33]. However, in practice the simplex method [13] is commonly used, despite its exponential worst-case running time [31]. The simplex algorithm moves along the edges of the polyhedron defined by the constraints, always in a direction that improves the solution. Usually this approach is very effective.

In contrast, most IPs cannot be solved in polynomial time in the worst case. For example, the ATSP is a NP-hard problem and can be defined by the following IP:

$$ATSP(G) = min \left(\sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij} \right) \tag{4}$$

subject to:

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

$$\sum_{j \in V} x_{ij} = 1, \; \forall i \in V \tag{6}$$

$$\sum_{i \in W} \sum_{j \in W} x_{ij} \leq |W| - 1, \; \forall \, W \subset V, \; W \neq \emptyset \tag{7}$$

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

for directed graph $G = (V, A)$ with vertex set $V = \{1, \ldots, n\}$ (where n is the number of cities), 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 (8) 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). Constraints (5) and (6) require that each city is entered exactly once and departed from exactly once. Constraints (7) are called *subtour elimination constraints* (SECs) as they require that no more than one cycle can exist in the solution. Note that there are an exponential number of SECs. It is common practice to initially omit these constraints and add only those that are necessary as the problem is solved. Finally, the objective function (4) 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 $(n - 1)!$ distinct solutions.

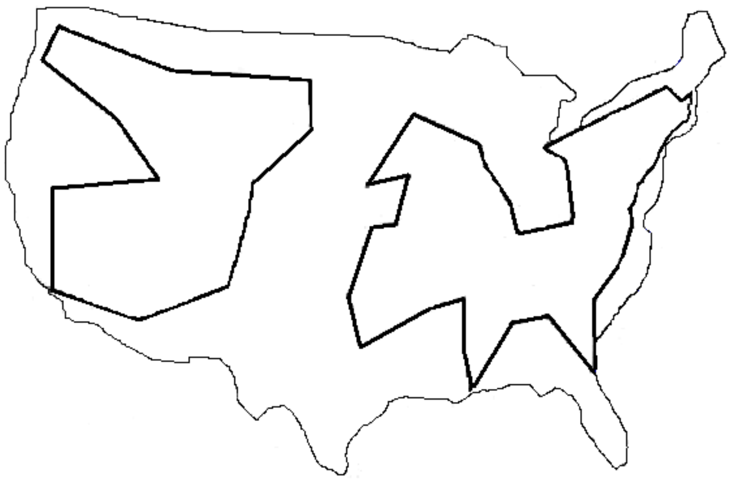

Fig. 1. AP relaxation of a 49-city TSP. Two subtours occurred in this solution.

2.1 Using bounds

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

An IP 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 entire solution space of the original problem, SS_o, however, the converse is not necessarily true.

An IP 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 completely omitting constraints (7). This relaxation allows any number of subtours to exist in the solution. This relaxed problem is simply the Assignment Problem (AP) [39]. 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. Figure 1 depicts a solution for the AP relaxation of a 49-city symmetric TSP (STSP). STSPs are a special class in which the cost from city i to city j is equal to the cost from city j to city i for all cities i and j.

Another relaxation can be realized by relaxing the integrality requirement of

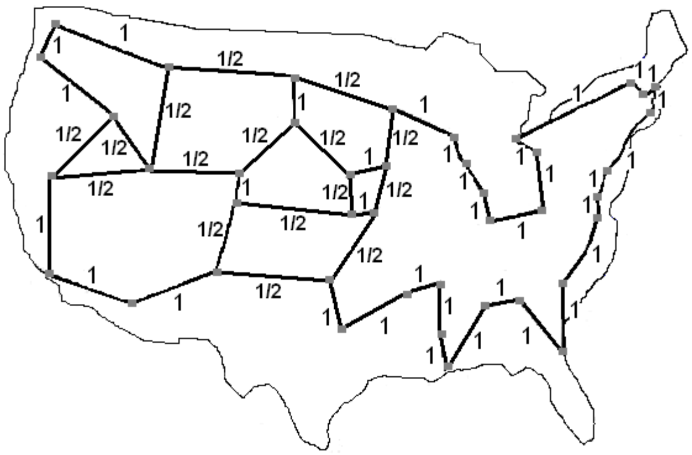

Fig. 2. Held-Karp relaxation of a 49-city TSP. The values shown on the arcs are their corresponding x_{ij} values. This example appeared in [14].

constraints (8). This can be accomplished by replacing constraints (8) with the following constraints:

$$0 \leq x_{ij} \leq 1, \ \forall i, j \in V \tag{9}$$

This relaxation transforms the IP into an LP and is referred to as the *Held-Karp* relaxation [25,26]. Figure 2 depicts a solution for the Held-Karp relaxation of the 49-city STSP. At first glance it appears that subtour elimination constraints are also violated. However, constraints (7) are not violated due to the fact that some of the x_{ij} variables have values less than one.

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

In 1958, several papers appeared that used branch-and-bound search [5,12,16,43]. This method organizes the search space into a tree structure. At each level of the tree, branching rules are used to generate child nodes. The children are tightened versions of their parents as constraints are added that set values of decision variables. Each node inherits all of the tightening constraints that were added to its ancestors. These tightened problems represent subproblems of the parent problem and the tightening may reduce the size of their individual solution spaces.

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

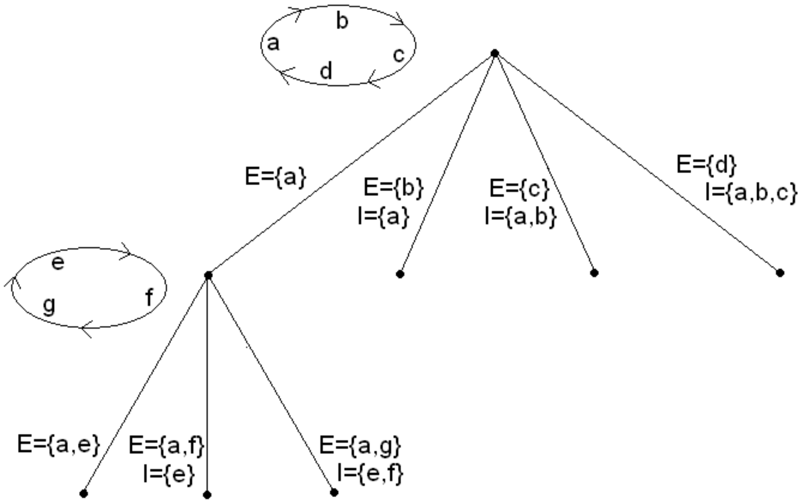

Fig. 3. An example of the first few nodes of a CDT search tree for an ATSP. The cycles shown are the subtours with the fewest free arcs for the AP solutions of the corresponding nodes. Arcs that are forced to be excluded are in sets E and arcs that are forced to be included are in sets I.

relaxation of the problem is solved. At every other node, a doubly-modified problem is solved; one that is simultaneously tightened and relaxed. The solution space of one of these doubly-modified problems may contain extra solutions that are not in the solution space of the original problem and may be 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 [7] as depicted in Figure 3. For this algorithm, the AP is used for the relaxation, allowing any number of subtours in the solution. 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 subtour 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 subtour. 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 an iterative search strategy in which *cutting planes* were systematically derived and applied to a relaxed problem [20]. 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 original (unrelaxed) problem, as none of the solutions to the original 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 equal to the current incumbent. At this point, the search is terminated and the incumbent solution is declared optimal.

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 the incumbent and increase future pruning potential.

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

`Concorde` [1,2] is an award-winning branch-and-cut algorithm designed for solving the symmetric TSP (STSP). This code has been used to solve many very large STSP instances, including a 24,978-city instance corresponding to cities in Sweden [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 to be used and an algorithm for solving this relaxation, branching rules, and a search method, 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 pruning capability.

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

Finally, a search method must be selected. *Best-first* search selects the node with the best heuristic value to be explored first. A* search is a best-first strategy that uses the sum of the cost of the path up to a given node and an estimate of the cost from this node to a goal node as the heuristic value [23]. 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 [49], 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 computational demands of solving the relaxed problem or finding cutting planes. For instance, the simplex method is commonly used for solving the relaxation for IPs, despite the fact that it has exponential worst-case performance.

3 Cut-and-Solve Search Strategy

3.1 Basic algorithm

Unlike the cutting planes in branch-and-cut search, cut-and-solve uses 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 Figure 4. [1]

```
algorithm cut_and_solve (IP)
    select cut
    find optimal feasible solution in space removed by cut
        update best if necessary
    add cut to problem
    find lower bound
    if (lower bound >= best) return best
    otherwise, repeat
```

Fig. 4. Cut-and-solve algorithm

In this algorithm, each iteration corresponds to one node in the search path. First, a piercing cut is selected. Let SS_{sparse} be the set of feasible solutions in the solution space removed by the cut. Then the best solution in SS_{sparse} 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 referred to as the incumbent, or *best*, solution. If later iterations find a solution that is better than *best*, *best* is replaced by this new solution.

In the next step, the piercing cut is added to the IP. This piercing cut excludes all of the solutions in SS_{sparse} from the IP. Thus, the piercing cut tightens the IP and reduces the size of its solution space. The lower bound for this tightened IP is then found. If this lower bound is greater than or equal to *best*, then the search is terminated and *best* is an optimal solution.

At subsequent nodes, the process is repeated. The incumbent solution is updated as needed. 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.

[1] We thank an anonymous reviewer for suggesting this pseudo code.

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 in the solution spaces that were cut away by the piercing cuts. The solution space of the final doubly-modified problem contains all of the solutions for the original problem except those that were cut away by the piercing cuts. If the relaxation of this reduced solution space has a value that is greater than or equal to the incumbent value, then this reduced solution space 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 both the relaxation algorithm and the algorithm for selecting and solving the sparse problem are guaranteed to terminate, then the cut-and-solve algorithm is guaranteed to terminate.*

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 problems solved, each of which are guaranteed to terminate. □

Cut-and-solve can be cast as a binary search tree with highly disproportionate children as shown in Figure 5. At each node, a relaxation is solved and a cut is chosen. This cut is used for the branching rule. The left child's solution space is small as it only contains the solutions that are cut away by the piercing cut. The right child has the huge solution space that is remaining after removing the left child's solutions. The left child's solution space is easily explored and produces potential incumbent solutions. It is immediately solved and the right child is subdivided into another set of disproportionate children. When the relaxed solution of the right child is greater than or equal to the current incumbent, it is pruned and the search is finished. Thus, the final configuration of the search tree is a single path at the far right side with a single branch to the left at each level. This search strategy is essentially linear: consistently solving the easy problems immediately and then redividing.

3.2 Using cut-and-solve as a complete anytime algorithm

The cut-and-solve algorithm is easily adapted to be an *anytime* algorithm [6]. Anytime algorithms allow the termination of an execution at any time and return the best approximate solution that has been found thus far. Many anytime algorithms are based on local search solutions. This type of anytime algorithm does not guarantee that the optimal solution will eventually be found

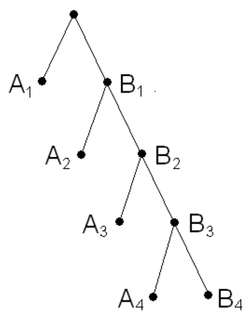

Fig. 5. Cut-and-solve depicted as a binary search tree. At each of the A_i nodes, the optimal solution is found for the small chunk of the solution space that is cut away by the corresponding piercing cut. At each of the B_i nodes, a relaxation is solved over the remaining solution space.

and cannot determine whether the current solution is optimal. *Complete* anytime algorithms [35,46,47] overcome these two restrictions. These algorithms will find an optimal solution if given adequate time and optimality is guaranteed. Cut-and-solve finds an incumbent solution at the root node so there is an approximate solution available any time after the root node is solved. This solution improves until the optimum is found or the execution is terminated, making it a complete anytime algorithm.

Cut-and-solve offers a favorable benefit when used as an anytime solver. Many approximation algorithms are unable to provide a bound on the quality of the solution. This can be especially problematic when the result is to be used in further analysis and an estimated error needs to be propagated. Upon termination of cut-and-solve, the value of the most recently solved doubly-modified problem is a lower bound on the optimal solution, yielding the desired information. Furthermore, the rate of decrease of the gap between the incumbent and the doubly-modified problem solutions can be monitored and used to determine when to terminate the anytime solver.

3.3 Parallel processing

Cut-and-solve can be computed using parallel processing. Each of the sparse problems can be solved independently on different processors. It is not necessary that every sparse problem runs to completion. Whenever a sparse problem is solved, it is determined which iteration had the *smallest* lower bound that is greater than or equal to the solution of the sparse problem. It is only necessary that the sparse problems prior to that iteration run to completion to ensure optimality.

Consider the search depicted in Figure 5. The solutions for the B_i nodes are non-decreasing as i increases. Suppose that A_4 completes first with a solution of x. Suppose further that B_1 has a solution that is less than x and B_2 has a solution that is equal to x. (In this case, B_2's solution can't be greater than x as it is a lower bound on the solution space that contains the solution found at B_4.) Only A_1 and A_2 need to run to completion and the search at A_3 can be terminated as it can't contain a solution that is better than x.

More generally, let y equal the cost of a solution found at a node A_p. Let B_q be the node with the smallest cost that is also greater than or equal to y. All of the searches at nodes A_i where $i > q$ can be 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 IP:

$$min \;\; Z = y - \frac{4}{5}x \tag{10}$$

subject to:

$$x \geq 0 \tag{11}$$

$$y \leq 3 \tag{12}$$

$$y + \frac{3}{5}x \geq \frac{6}{5} \tag{13}$$

$$y + \frac{13}{6}x \leq 9 \tag{14}$$

$$y - \frac{5}{13}x \geq \frac{1}{14} \tag{15}$$

$$x \in I \tag{16}$$

$$y \in I \tag{17}$$

This IP has only two decision variables, allowing representation as a 2D graph as shown in Figure 6. Every x, y pair that is feasible must obey the constraints. Each of the first five linear constraints (11) to (15) corresponds to an edge of the polygon in Figure 6. All of the feasible solutions must lie inside or on the edge of the polygon. Constraints (16) and (17) require that the decision variables assume integral values. Therefore, the feasible solutions for this IP are shown by the dots located inside and on the edge of the polygon.

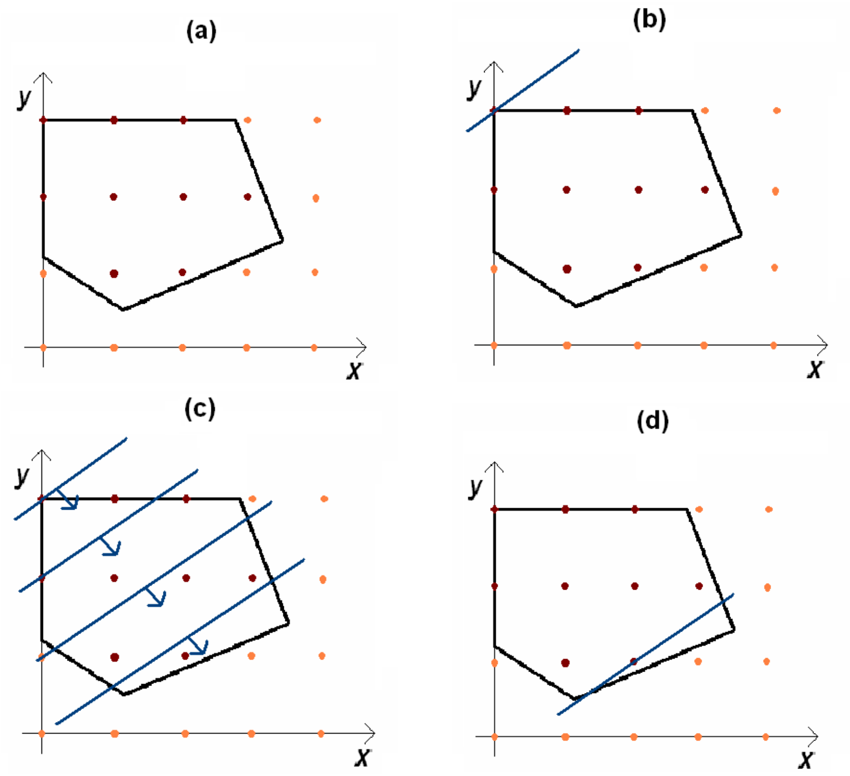

Fig. 6. Graphical solution of the example IP. (a) The feasible solution space. Dots within and on the polygon represent all of the feasible solutions. (b) One of the lines representing the objective function where $Z = 3$. It intersects the feasible solution $x = 0$, $y = 3$. (c) Lines representing the objective function for several values of Z. Arrows show the direction in which Z decreases. (d) The optimal solution of $x = 2$, $y = 1$, and $Z = -0.6$.

The terms of the objective function (10) can be rearranged into the slope-intercept form as follows: $y = \frac{4}{5}x + Z$. Therefore, the objective function of this IP 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 IPs, the y-intercept is equal to a constant times Z.) Figure 6(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 6(d). The optimal solution is $x = 2$ and $y = 1$, yielding $Z = -0.6$.

For this IP, the solution space of the original problem, SS_o, contains the nine points that lie within and on the polygon. Each of these nine points are feasible solutions as they satisfy all of the constraints. To apply cut-and-solve, a relaxation must be chosen. In this example, we relax constraints (16) and (17). 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 7 shows the steps taken when solving this IP using cut-and-solve. First, the relaxed solution is found. The solution to this relaxation is shown in Figure 7(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.

Next, a piercing cut is selected as shown in Figure 7(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 is -0.4. *Best* is set to -0.4.

The line that cuts away the shaded region from the polygon in Figure 7(b) can be represented by a linear constraint: $y - \frac{17}{3}x \geq -14$. This constraint is added to the IP. Now the feasible region of the IP is reduced and its current solution space contains eight points as shown in Figure 7(c). The relaxation of the current IP as shown in Figure 7(d). The value of this relaxed solution is -1.1. Notice that this lower bound is less than *best*, so the search continues to the second iteration.

At the second node of the search, we select a piercing cut as shown in Figure 7(e). The sparse problem represented by the shaded area in Figure 7(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 *best* is set to -0.6.

The linear constraint corresponding to the current piercing cut is added to the IP, resulting in a reduced feasible solution space as shown in Figure 7(f). The relaxed problem is solved on the current IP as shown in Figure 7(g). The new lower bound is -0.2. This value is greater than the *best* 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. -0.2 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 example problem presented in this section is easily solved by inspection. However, problems of interest may contain many thousands of decision variables and constraints, yielding solution spaces that are defined by convex

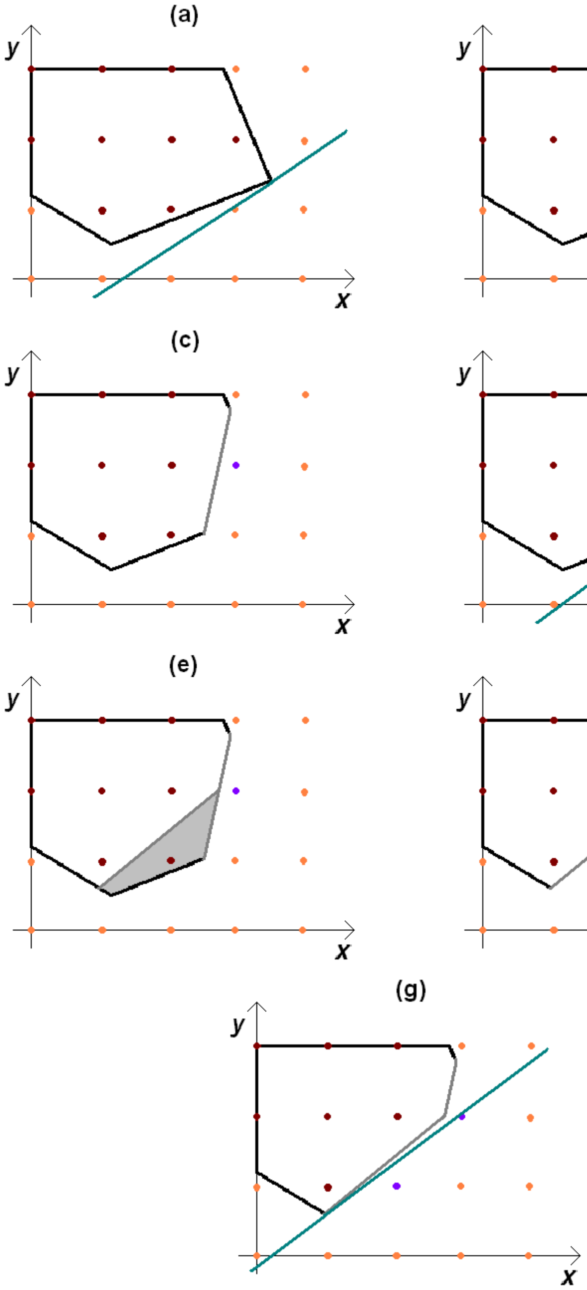

Fig. 7. Solving the example IP using cut-and-solve. (a) The relaxed solution of the original problem. (b) The first piercing cut. (c) The solution space of the IP with the piercing cut added. (d) The relaxed solution of the doubly-modified problem. (e) The second piercing cut. (f) The solution space of the IP with both piercing cuts added. (g) The relaxed solution of the doubly-modified problem, resulting in a value that is worse than the current incumbent.

polyhedrons in a correspondingly high-dimensional space. For example, a 100-city ATSP has 9,900 decision variables and 10,100 constraints, not counting subtour elimination constraints.

The piercing cuts chosen for the example problem in this section were customized for this particular instance. In the next section, a generic procedure for deriving piercing cuts is presented.

5 A generic cut-and-solve procedure

The cut-and-solve algorithm requires three selections. A relaxation must be chosen, a method for deriving piercing cuts must be devised, and a technique for solving the sparse problem must be selected. For best results, each of these choices should be determined by careful evaluation of the particular problem that is being addressed. For many problems of interest, relaxation techniques and algorithms for solving sparse instances may have been previously researched. However, the derivation of piercing cuts has not been previously explored. In this section, we summarize favorable characteristics of piercing cuts and present a generic procedure for deriving them. Then we present a generic cut-and-solve algorithm with all three selections specified. This general procedure can be used for any IP. While a more customized approach should yield better results, this generic approach is used in our implementation for the ATSP with very good results.

Every IP can be cast as a binary IP (BIP) [27], where the decision variables are restricted to the values of 0 and 1. For a given BIP, the decision variables can be partitioned into two sets: a small set S and a large set L. The sparse problem is solved over the small set S. This is essentially the same as setting the values of all of the variables in L to zero. After the sparse problem is solved, the constraint that set all of the variables in L to zero is removed. Then the following piercing cut is added to the BIP:

$$\sum_{x_i \in L} x_i \geq 1 \tag{18}$$

Using this cut partitions the solution space. Either the sum of the variables in L is equal to zero or it is greater than or equal to one. The solutions corresponding to the first case are precisely the solutions for the sparse problem. All of the other solutions in the solution space are contained in the original problem with the addition of piercing cut (18). The question that remains is how to partition the variables into sets S and L, producing effective piercing cuts. The answer to this question is believed to be problem dependent. We offer some guidelines here.

Following are desirable properties of piercing cuts:

(1) Each piercing cut should remove the solution to the current relaxed prob-

lem so as to prevent this solution from being found in subsequent iterations.

(2) The space that is removed by the piercing cut should be adequately sparse, so that the optimal solution can be found relatively easily.

(3) The piercing cuts should attempt to capture an optimal solution for the original problem. The algorithm will not terminate until an optimal solution has been cut away and consequently made the incumbent.

(4) In order to guarantee termination, each piercing cut should contain at least one feasible solution for the original, unrelaxed, problem.

Sensitivity analysis is extensively studied and used in Operations Research [44]. We borrow a tool from that domain to devise a totally generic approach to partitioning the decision variables into sets S and L as follows. First, the integrality constraints are relaxed and the BIP is solved as an LP. An LP solution defines a set of values referred to as *reduced costs*. Each decision variable x_i has a reduced cost value, which is a lower bound on the increase of the LP solution cost if the value of the variable is increased by one unit. Each of the decision variables with non-zero values in the LP solution has a reduced cost of zero. If another decision variable, x_i, has a reduced cost of zero, it may be possible to increase the value of x_i without increasing the cost of the LP solution. On the other hand, if x_i has a large reduced cost, then an equally large, or larger, increase of the LP solution cost would occur if there is a unit increase of the value of x_i.

We use this concept in our generic algorithm. Set S can be composed of all the decision variables with reduced costs less than a parameter α, where $\alpha > 0$. α should be set small enough that the resulting sparse problem is relatively easy to solve – satisfying property (2). On the other hand, it should be set as large as possible to increase the likelihood that set S contains all of the decision variables necessary for an optimal solution of the original problem. Property (1) is satisfied as the LP solution must be included in the sparse solution space because the decision variables with non-zero values in the LP solution have reduced costs of zero. Intuitively, decision variables with small reduced costs seem more likely to appear in an optimal solution. Thus, property (3) is addressed by using reduced costs as a guide for selecting variables for set S and by making α as large as practical. If guaranteed termination is required, property (4) can be satisfied by setting α to an adequately large value or adding variables to S that will guarantee at least one feasible solution for the original problem.

Having determined a method for partitioning the decision variables, we now present a completely generic cut-and-solve program that can be used for any BIP (and consequently any IP), as outlined in Figure 8. The relaxation used is the omission of the integrality constraints. The sparse problem can be solved using any BIP or IP solver. The piercing cut that is added to the BIP is

$\sum_{x_i \notin S} x_i \geq 1$.

```
algorithm generic_cut_and_solve (BIP)
    relax integrality and solve LP
    if (LP solution >= best) return best
    let S = {variables with reduced costs <= alpha}
    find optimal feasible solution in S
        update best if necessary
        if (LP solution >= best) return best
    add (sum of variables not in S >= 1) to BIP
    repeat
```

Fig. 8. A generic cut-and-solve algorithm

Notice that the performance of this generic approach could be improved by customizing it for the problem at hand. Tighter lower bounds, more effective piercing cuts, and/or use of a solver that is designed to solve sparse instances of the problem of interest may yield dramatic increases in performance. Yet, in the next two sections we show how a simple implementation of this generic approach can produce impressive results.

6 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.

Seven real-world problem classes of the ATSP have been studied in [8]. Instance generators for these seven problem classes are available on David Johnson's webpage at: http://www.research.att.com/~dsj/chtsp/atsp.html. A brief description of the seven problems follows. The first class is the approximate shortest common superstring (`super`). This problem pertains to genome reconstruction, in which a given set of strings are combined to form the shortest possible superstring. The second class is tilted drilling machine, with an additive norm (`rtilt`). This class corresponds to the problem of scheduling the drilling of holes into a tilted surface. The third class (`stilt`) corresponds to the same problem, but with a different norm. The fourth class is random Euclidean stacker crane (`crane`), which pertains to planning the operation of a crane as it moves a number of items to different locations. The next class is disk drive (`disk`). This class represents the problem of scheduling a computer disk read head as it reads a number of files. The sixth class is pay phone collection (`coin`), corresponding to the problem of collecting coins from pay phones. The phones are located on a grid with two-way streets, with the perimeter of the grid consisting of one-way streets. Finally, the no-wait flow shop (`shop`) class represents the scheduling of jobs that each require the use of multiple machines. In this problem, there cannot be any delay for a job between machines.

In the generic algorithm, lower bounds are found by relaxing integrality. For the ATSP, this is the Held-Karp lower bound. Then arcs with reduced costs less than α are selected for set S and a sparse graph composed of these arcs is solved. 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 arcs in S is less than or equal to $n-1$, where n is equal to the number of cities. This has the same effect as the piercing cut presented in the generic algorithm ($\sum_{x_i \notin S} x_i \geq 1$) as there are n arcs in an optimal solution for the ATSP. 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. Furthermore, after the first iteration, the incumbent solution can be used as an upper bound for the sparse problem solver, potentially improving its performance.

Selection of an appropriate value for α may be 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 class and the number of cities. We believe that determining α directly from the distribution of reduced costs would enhance this implementation as *a priori* knowledge of the problem class 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 using very few iterations, so guaranteeing termination was not of practical importance.

We used an LP and IP solver, `Cplex` [29], to solve both the relaxation and the sparse problem. All of the parameters for this solver were set to their default modes and no "warm starts" were used. For some of the larger instances, this generic solver became 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) 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, or (3) implementing a hybrid algorithm that integrates constraint programming with linear programming. However, despite the crudeness of our implementation, it suffices to demonstrate the potential of the cut-and-solve method. We compare our solver with branch-and-bound and branch-and-cut implementations in the next section.

7 Computational Results for the ATSP

In this section, we compare cut-and-solve with branch-and-bound and branch-and-cut solvers. We used the seven real-world problem generators [8] that were discussed in Section 6 to conduct large scale experiments. For each problem class and size, we ran 500 trials, revealing great insight as to the expected behavior of each solver for each problem class. We started with $n = 25$ and incremented by 25 cities at a time until the solver was unable to solve the instances in an average time of ten minutes or less; thus allowing at most 5,000 minutes per trial.

With the exception of the `super` problem generator, the generators use a parameter which has an effect on the number of significant digits used for arc costs. We set this parameter to 100,000. The `crane` class uses a second parameter which was increased for larger numbers of cities in [8]. We set this parameter to 10 for $n \leq 100$, 13 for $n = 125$, and 15 for $n = 150$.

We compare cut-and-solve (`CZ`) with the branch-and-bound algorithm introduced by Carpaneto, Dell'Amico, and Toth (`CDT`) [7], which was discussed in Section 2. We also make comparisons with two branch-and-cut solvers: `Concorde` [1,2] and `Cplex` [29].

`Concorde` was introduced in Section 2. It is used for solving symmetric TSPs

(STSPs). ATSP instances can be transformed into STSP instances using a *2-node* transformation [32]. David Johnson has publicly-available code for this 2-node transformation at http://www.research.att.com/~dsj/chtsp/atsp.html. When using the 2-node transformation, the number of arcs is increased to $4n^2 - 2n$. However, the number of arcs that have neither zero nor (essentially) infinite cost is $n^2 - n$, as in the original problem.

`Concorde` allows adjustment of its "chunk" size. This parameter controls the degree to which *project-and-lift cuts* [3] are utilized. Project-and-lift cuts are dynamically produced to suit the particular instance being solved. If the chunk size is set to zero, no project-and-lift cuts are generated and `Concorde` behaves as a pure STSP solver. It was shown in [17] that the default chunk size of 16 produced the best results for real-world ATSP instances. We used this default value in all of our trials.

The only publically available branch-and-cut code for specifically solving TSPs that we could find was `Concorde`. However, `Cplex` is a robust branch-and-cut solver for general IPs, so we also included it in comparisons. This solver dynamically determines search strategies and cuts for each instance. For instance, the variables chosen for branching and the direction of the search are customized to the given instance. Furthermore, `Cplex` uses nine different classes of cutting planes (Gomory cuts comprise one of the classes) and automatically determines the frequency of each class of cuts to use. Finally, `Cplex` uses various heuristics to find integer feasible solutions while traversing the search tree.

Generally speaking, `CZ` is as generic as our `Cplex` implementation. Both algorithms identify subtours and add appropriate subtour elimination constraints. Other than this, both algorithms are generic IP solvers.

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 (although we did not use parallel processing). In order to identify subtours in the relaxed problem, we used Matthew Levine's implementation of the Nagamochi and Ibaraki minimum cut code [37], which is available at [38]. Our code is available at [10].

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.

The average normalized computation times and the 95% confidence intervals for the real-world problem classes are shown in Figures 9 through 15. One may be interested in the "typical" time that is required for each of these solvers, so the median times for all of the trials are given in Figures 16 through 22. Table 1 lists the largest problem instance size that is solved by each algorithm

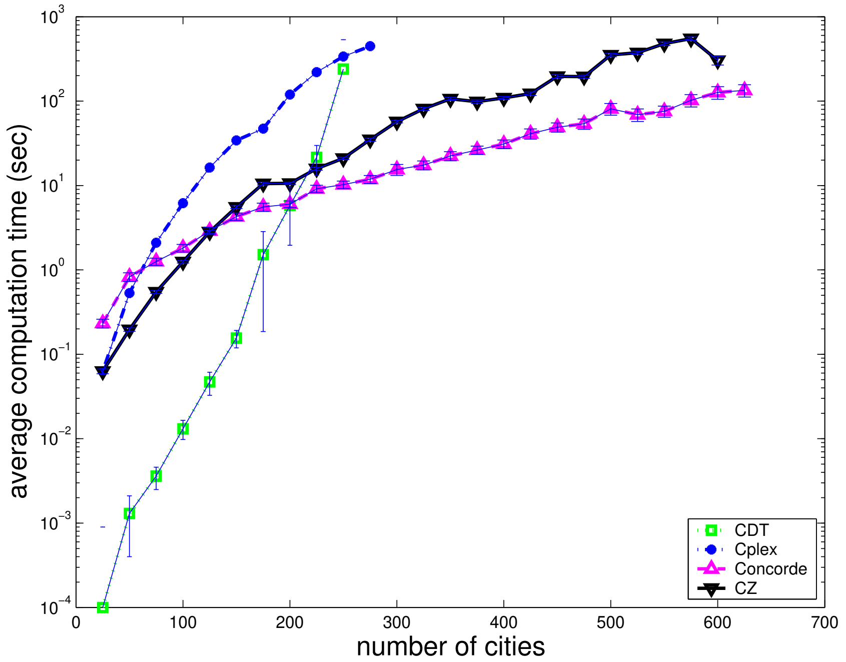

Fig. 9. Average normalized computation times with 95% confidence interval shown for the `super` problem class.

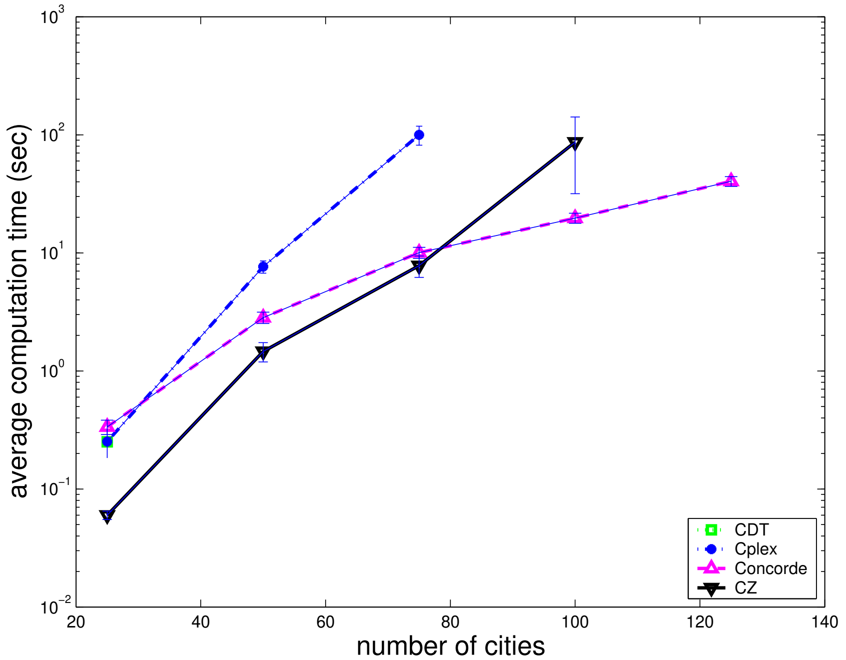

Fig. 10. Average normalized computation times with 95% confidence interval shown for the `rtilt` problem class.

within the execution time limit.

`Concorde` outperformed all of the other solvers for the `super` class as shown in Figures 9 and 16. It was able to solve instances with $n = 625$, `CZ` solved

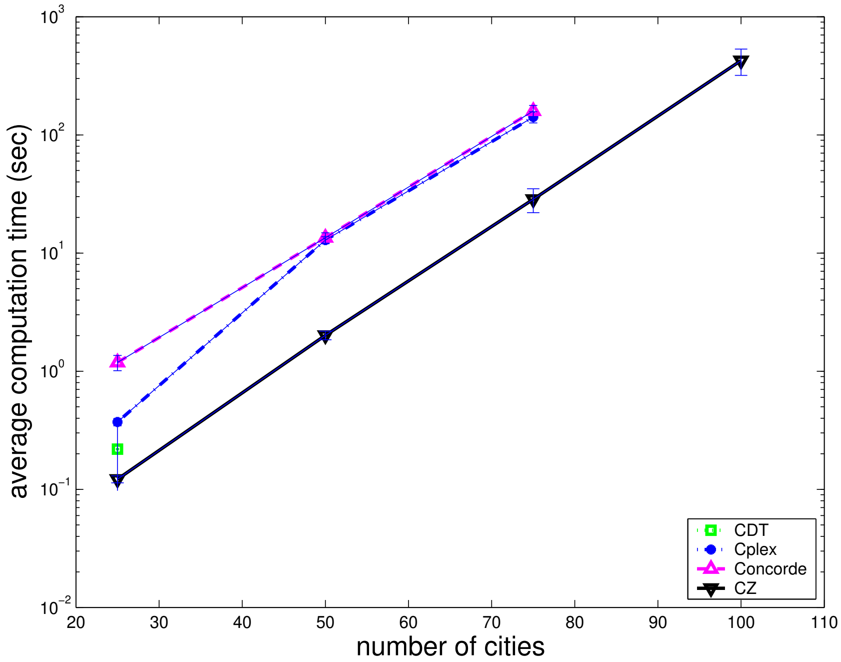

Fig. 11. Average normalized computation times with 95% confidence interval shown for the `stilt` problem class.

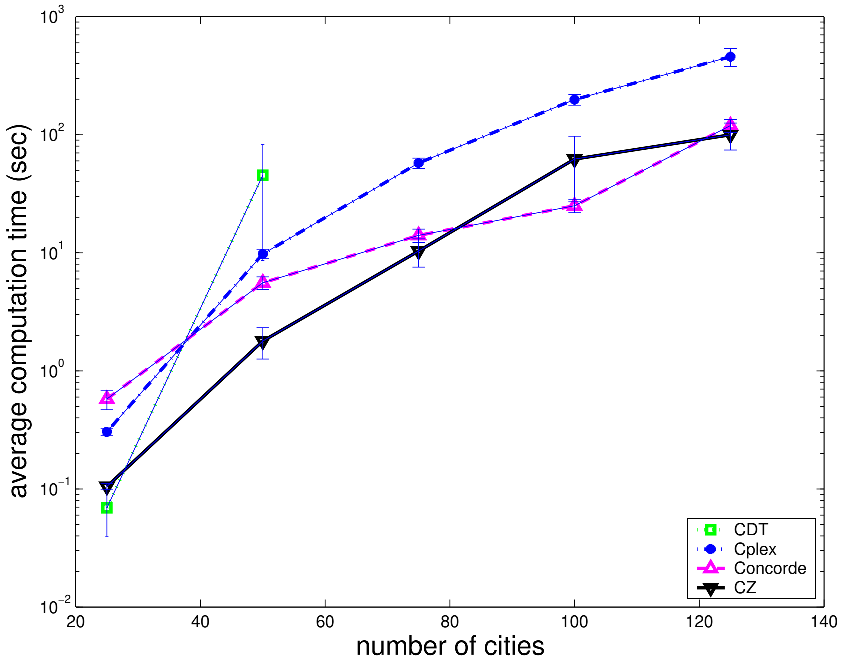

Fig. 12. Average normalized computation times with 95% confidence interval shown for the `crane` problem class.

600 cities, `Cplex` solved 275 cities, and `CDT` solved 250 cities.

`Concorde` also outperformed the other solvers for the `rtilt` problem class as shown in Figures 10 and 17. It solved 125-city instances, `CZ` solved 100 cities,

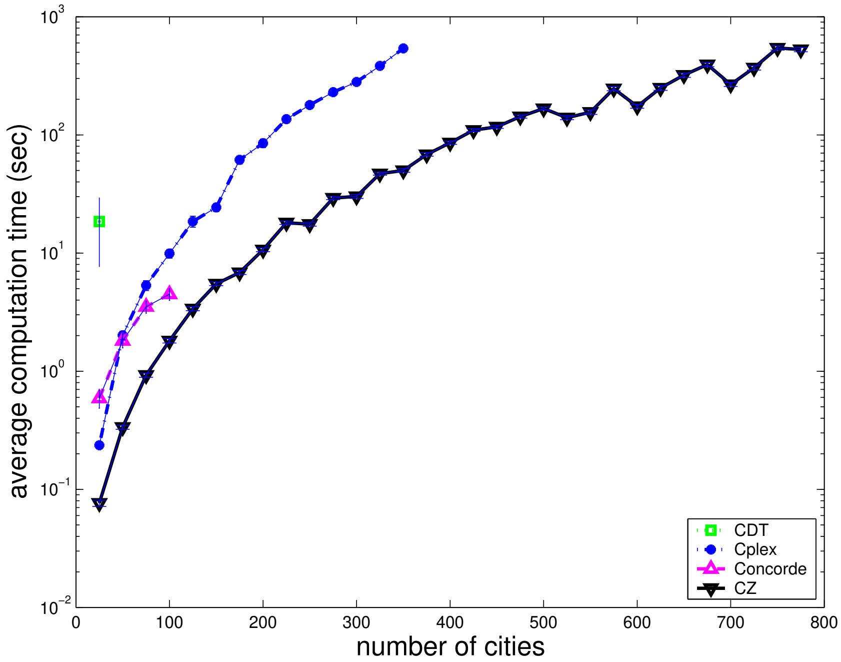

Fig. 13. Average normalized computation times with 95% confidence interval shown for the `disk` problem class.

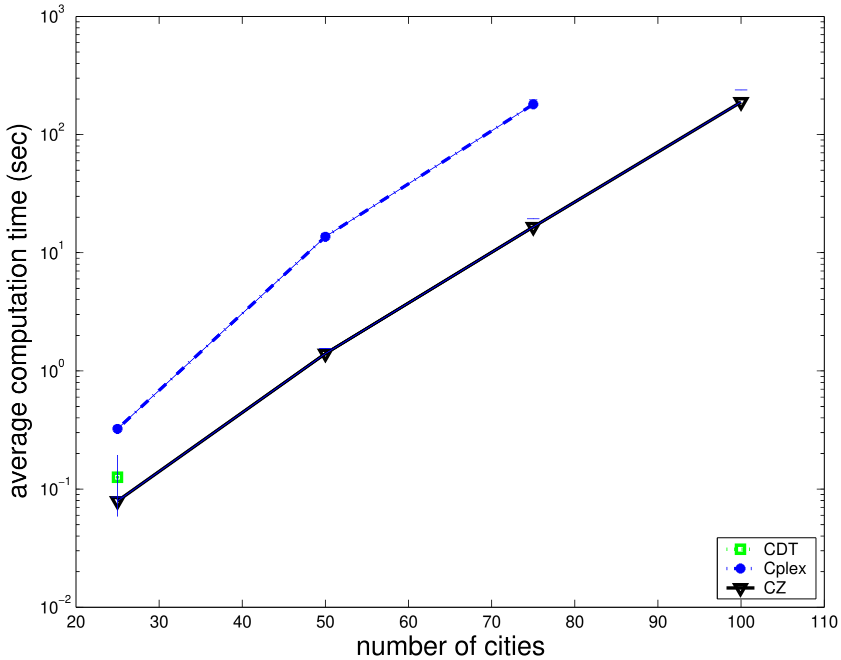

Fig. 14. Average normalized computation times with 95% confidence interval shown for the `coin` problem class.

`Cplex` solved 75 cities, and `CDT` solved 25 cities.

We solved some of the `super` and `rtilt` instances for all solutions and found they both had many optimal solutions.

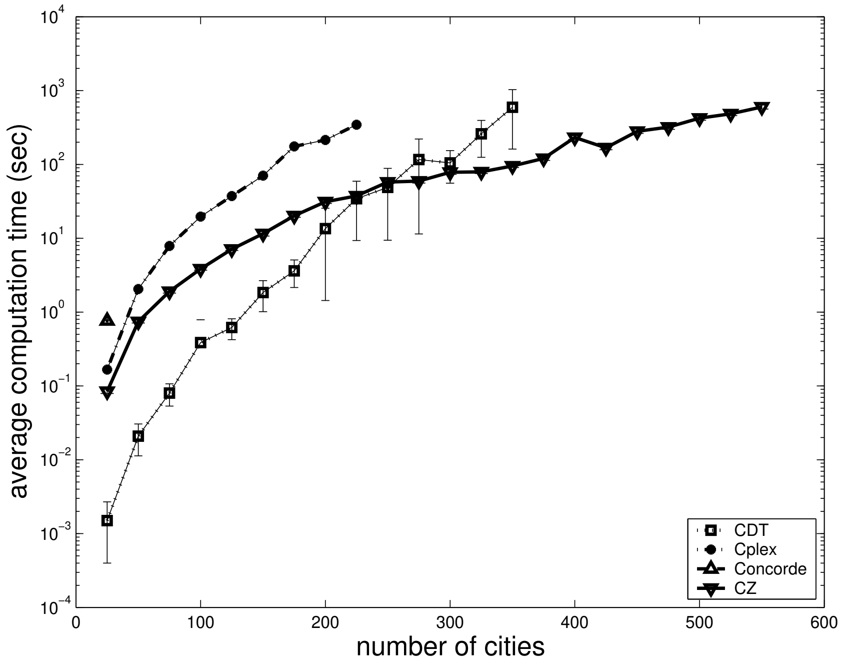

Fig. 15. Average normalized computation times with 95% confidence interval shown for the `shop` problem class.

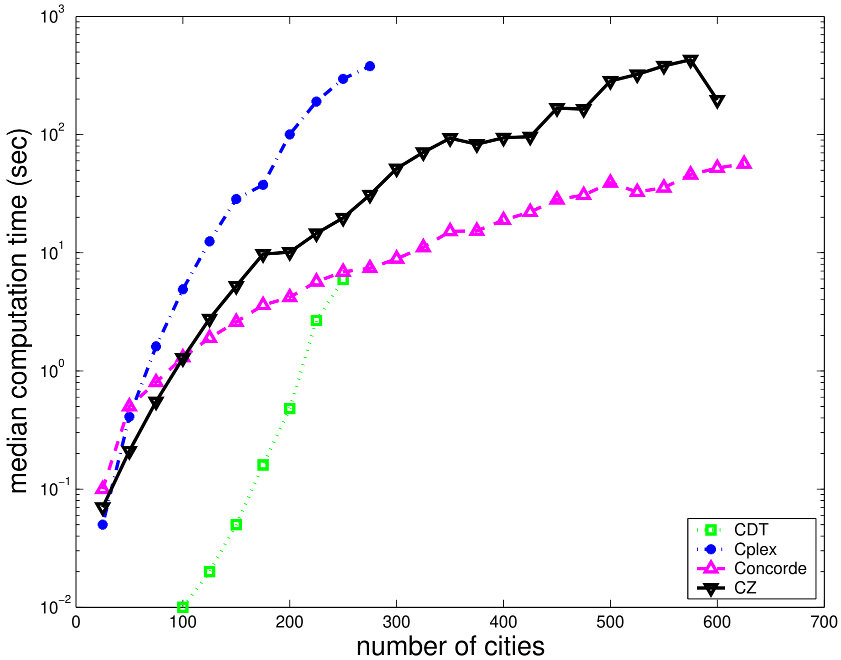

Fig. 16. Median computation times for the `super` problem class.

For the other five problem classes, `CZ` outperformed the other solvers. For the `crane` class, `CZ`, `Concorde`, and `Cplex` were able to solve up to 125 cities as shown in Figures 12 and 19. `CZ` was the fastest for this number of cities, followed by `Concorde`.

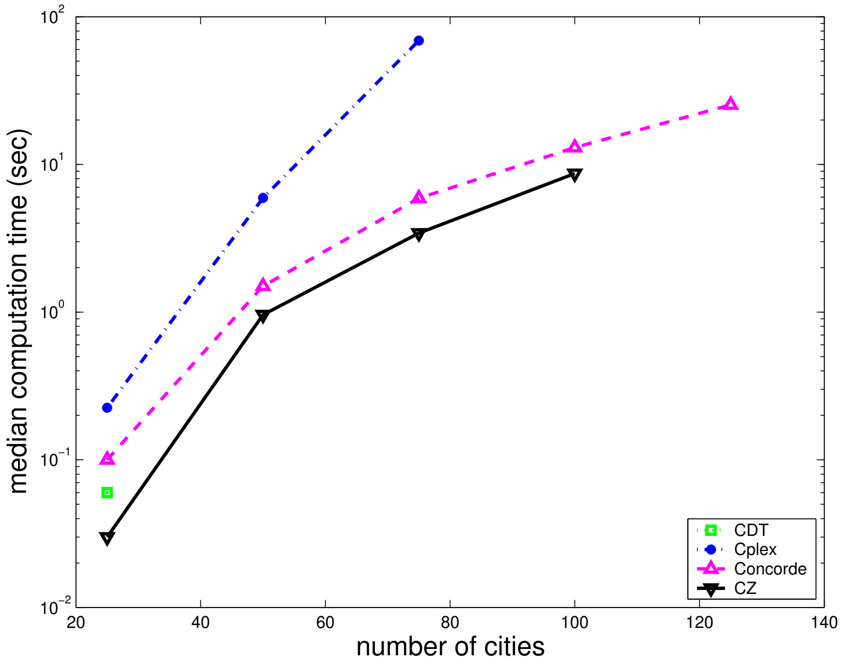

Fig. 17. Median computation times for the `rtilt` problem class.

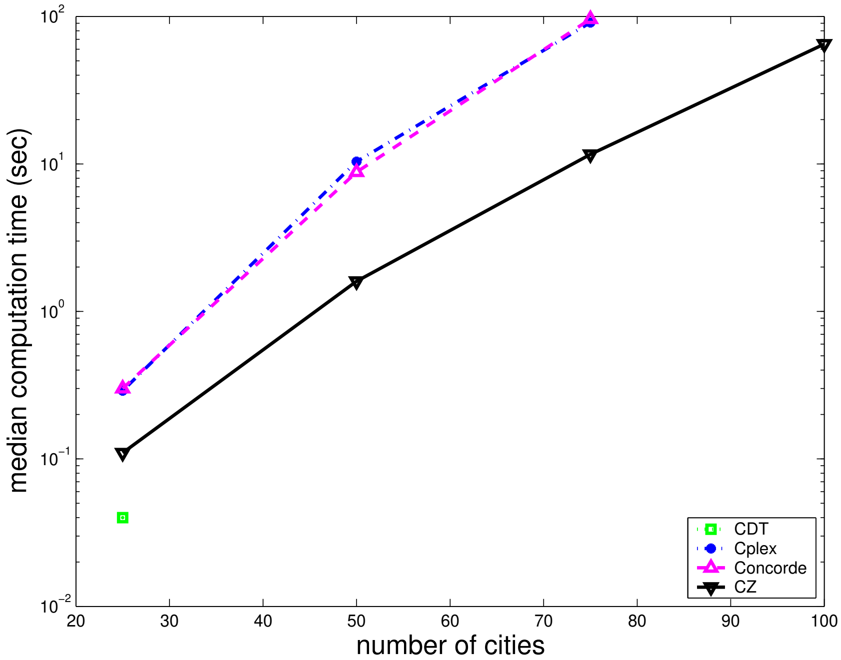

Fig. 18. Median computation times for the `stilt` problem class.

For the four other problem classes, `CZ` was able to solve larger instances than any of the other solvers. For `stilt`, `CZ` solved 100 cities within the allotted time, while `Concorde` and `Cplex` solved 75 cities as shown in Figures 11 and 18. For `disk`, `CZ` solved 775 cities, while `Cplex` followed far behind with 325 cities as shown in Figures 13 and 20. `CZ` solved 100-city `coin` instances and

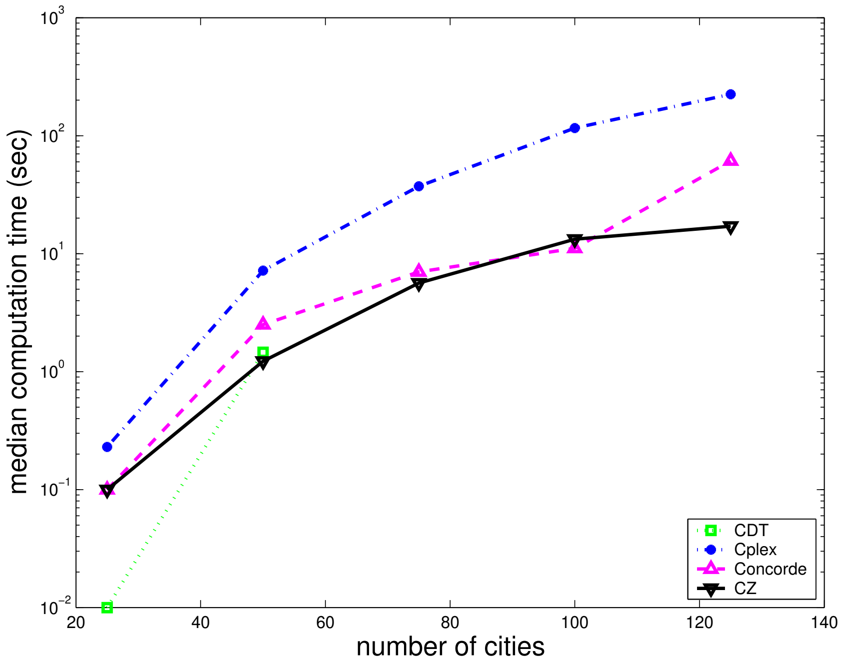

Fig. 19. Median computation times for the `crane` problem class.

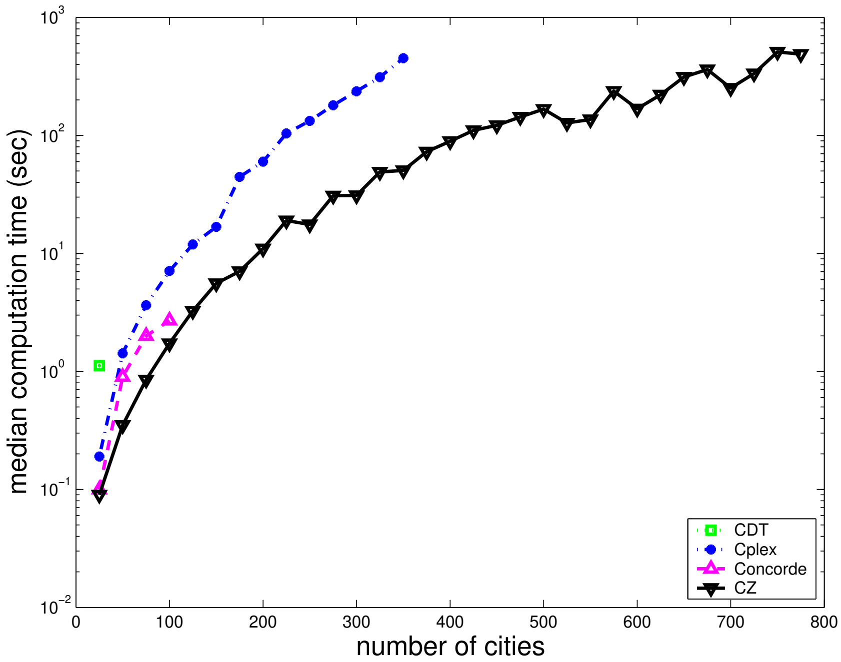

Fig. 20. Median computation times for the `disk` problem class.

`Cplex` solved 75 cities as shown in Figures 14 and 21. Finally, `CZ` was able to solve 550-city `shop` instances, with `CDT` following with 350 cities as shown in Figures 15 and 22.

`CDT` performed moderately well for `super` and `shop`. However, it was only able to solve 50 cities for `crane` and 25 cities for the other four problem classes.

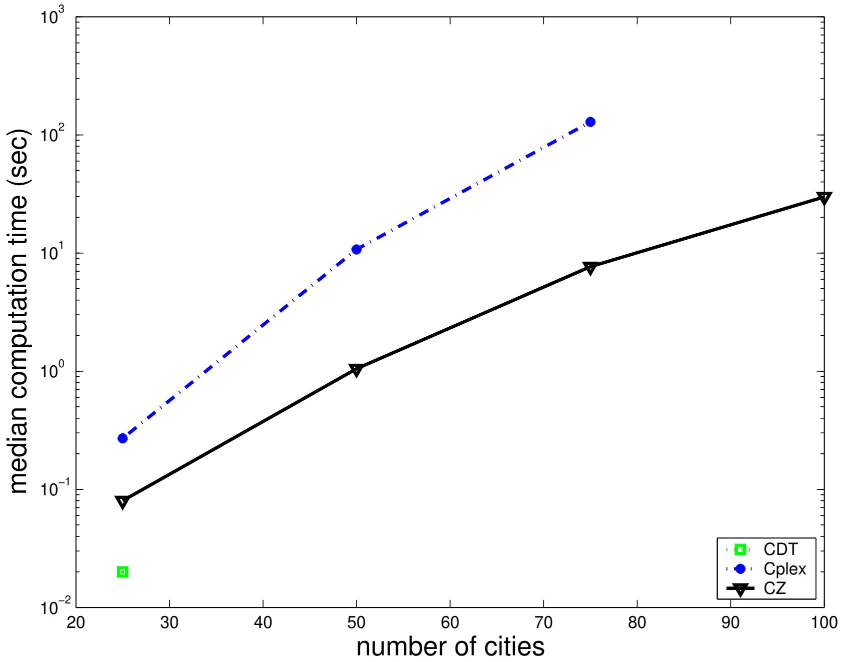

Fig. 21. Median computation times for the `coin` problem class.

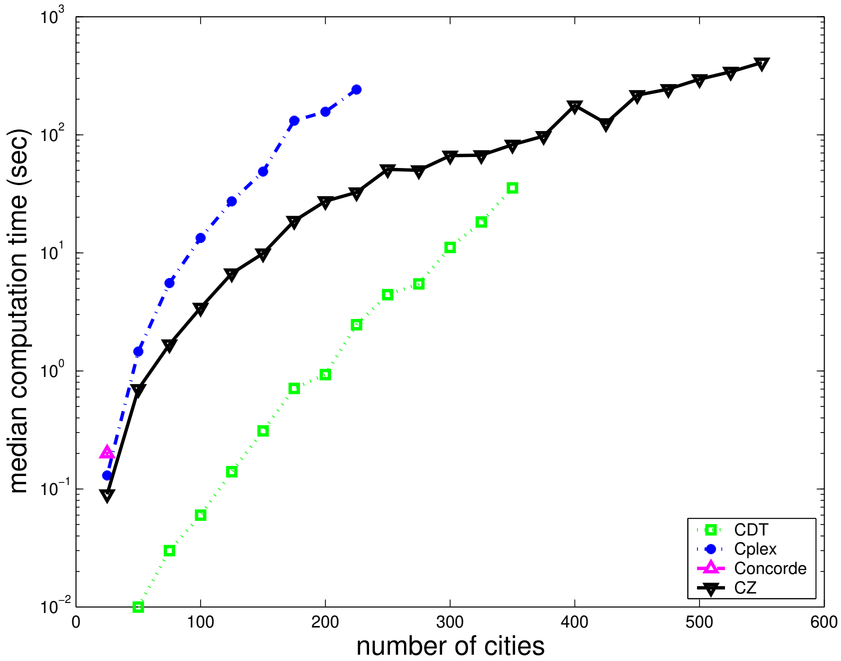

Fig. 22. Median computation times for the `shop` problem class.

We let the computations run past the time limit and found that many of these trials required substantial amounts of time. The worst case was for the 50-city `disk` class, which we finally killed at 25,900 minutes.

In summary, for the problem instances we tested, CDT was not very robust. `Cplex` was robust, but had moderate performance in general. `Concorde` was

	CDT	Concorde	Cplex	CZ
super	250	625	275	600
rtilt	25	125	75	100
stilt	25	75	75	100
crane	50	125	125	125
disk	25	100	325	775
coin	25	-	75	100
shop	350	25	250	550

Table 1
Largest problem size that was solved within the allotted time for each algorithm and each problem class. (`Concorde` terminated with errors for the `coin` class.)

the fastest for the two problem classes that had a large number of optimal solutions. `CZ` was robust and had the best performance for the other five classes.

8 Discussion and Related Work

Many optimization problems of interest are difficult to solve to optimality, so approximations are employed. Sacrificing optimality can be costly in several situations. Approximations can be problematic when the solution is important for the progress of research, such as biological applications. Approximations can be costly when the solution is an expensive procedure, such as routing a spacecraft, or when the solution is reused a number of times, as in manufacturing applications. Moreover, it can be deleterious to use an approximation in any application if the approximation is poor. Cut-and-solve offers an alternative to branch-and-bound and branch-and-cut when optimality is desired.

In our experiments, we observed that the performance of cut-and-solve diminished for problems with large numbers of optimal solutions. We monitored a few of these trials and noticed that cut-and-solve found an optimal solution early in the search, but required a substantial amount of computation time to prove optimality. We suspect that many optimal solutions remained in the doubly-modified problem, making it difficult for the relaxation to be equal to the optimal solution value. It may be that most of the optimal solutions had to be cut away before the computation could complete.

One concern about the viability of 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 [40,48]), then setting its value to zero is calamitous. Conversely, forcing the inclusion of a variable that doesn't appear in any feasible solution (a *fat* variable [9]) 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.

Search tree methods such as branch-and-bound and branch-and-cut must choose between memory problems or the risk of fruitlessly searching subtrees containing no optimal solutions. Cut-and-solve is free from these difficulties. Its memory requirement is 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 an iterative search, there are no subtrees in which to get lost. A great number of techniques have been devised in an effort to overcome this problem for depth-first search. Among these are iterative deepening [34], limited discrepancy [24], and randomization and restarts [19].

Another interesting comparison is that branch-and-bound and branch-and-cut must compute at least one relaxed solution that is not only feasible for the original problem, but it must also be optimal for the original problem. In contrast, cut-and-solve is never required to find a relaxed solution that is optimal or even feasible for the original problem. Consider, for example, generic cut-and-solve. Each of the right-hand children in Figure 5 may have non-integral solutions. Yet, the search is terminated as soon as the value of the relaxation is greater than or equal to the incumbent value.

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 an iterative progression of the search. 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.

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. Piercing 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 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 more expeditiously.

Cut-and-solve is also similar to an algorithm for solving the Orienteering Problem (OP) as presented in [18]. In this work, *conditional cuts* remove feasible solutions to the original problem. These cuts are used in conjunction with 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 an iterative fashion, however, 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 [18] stores sparse problems and combines and solves them as a single, larger problem after the initial branch-and-cut tree is explored. Finally, this OP algorithm is not truly iterative as branching is allowed.

In a more broad sense, cut-and-solve is similar to divide-and-conquer [41] 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. When comparing these two techniques, cut-and-solve might be thought of as divide-and-conquer with powerful pruning rules.

We conclude this section with a note about the generality of piercing cuts. Piercing cuts are used in an iterative fashion for cut-and-solve, but they could also be applied in other search strategies such as branch-and-cut. As long as the sparse problem defined by the cut is solved, piercing cuts could be added to an IP. For example, Fischetti and Toth's branch-and-cut algorithm [17] solves a number of sparse problems at each node of the search tree in order to improve the upper bound. Piercing cuts corresponding to these sparse problems could be added to the IP without loss of optimality. These piercing cuts may tighten the problem to a greater degree than the tightening due to traditional cuts and may reduce the time required to solve the instance.

9 Conclusions

In this paper, we presented a search strategy which we referred to as cut-and-solve. We showed that optimality and termination are guaranteed despite the fact that no branching is used. Being an iterative 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 negligible, 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 demonstrated cut-and-solve for integer programs and have implemented this strategy for solving the ATSP. In general, our generic implementation is robust and outperforms state-of-the-art solvers for difficult real-world instances that don't have many optimal solutions. For five of the seven problem classes, cut-and-solve outperformed the other solvers. The two remaining classes had many optimal solutions.

In the future, we plan to improve our implementation by designing a custom solver for sparse ATSPs. As for the general cut-and-solve search strategy, we are looking into dynamically determining α so that it is customized to each particular instance. Finally, we are investigating the use of cut-and-solve for other IPs. We are currently implementing it for a biological problem, haplotype inferencing [21].

Life is full of optimization problems. Many such problems arise in the field of Artificial Intelligence 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, ITR/EIA-0113618, and IIS-0535257, and in part by DARPA Cooperative Agreement F30602-00-2-0531. We are grateful to the anonymous reviewers of this paper, as well as a preliminary conference paper, who provided a number of valuable insights. We thank Matthew Levine, for the use of his minimum cut code, and David Applegate, Robert Bixby, Vašek Chvátal, and William Cook for the use of Concorde. We also extend thanks to Matteo Fischetti, who generously shared his expertise in several 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.tsp.gatech.edu//concorde.html*, web.

[3] D. Applegate, Robert Bixby, V. Chvátal, and W. Cook. On the solution of Traveling Salesman Problems. In *Documenta Mathematica*, pages 645–656. 1998. Extra Volume ICM III.

[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] F. Bock. An algorithm for solving 'traveling-salesman' and related network optimization problems. Technical report, Armour Research Foundation, 1958. Presented at the Operations Research Society of America 14th National Meeting, St. Louis, Oct. 1958.

[6] M. Boddy and T. Dean. Solving time-dependent planning problems. In *IJCAI-89*, pages 979–984, Detroit, MI, 1989.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6:791–812, 1958.

[13] G. Dantzig. Maximization of linear function of variables subject to linear inequalities. In T.C. Koopmans, editor, *Activity Analysis of Production and Allocation.* Wiley, NY, 1951.

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

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

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

[17] 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.

[18] 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.

[19] 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.

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

[21] D. Gusfield and S.H. Orzack. Haplotype inference. In S. Aluru, editor, *Handbook on Bioinformatics*. CRC, 2005. to appear.

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

[23] T. P. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science and Cybernetics*, 4:100–107, 1968.

[24] 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.

[25] M. Held and R. M. Karp. The traveling salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

[26] M. Held and R. M. Karp. The traveling salesman problem and minimum spanning trees: Part ii. *Mathematical Programming*, 1:6–25, 1971.

[27] F. Hillier and G. Lieberman. *Introduction to Operations Research*. McGraw-Hill, Boston, 6th edition, 2001.

[28] 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.

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

[30] 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.

[31] 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.

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

[33] L. G. Khaciyan. A polynomial algorithm for linear programming. *Doklady Akademii Nauk SSSR*, 244:1093–1096, 1979.

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

[35] R. E. Korf. A complete anytime algorithm for number partitioning. *Artificial Intelligence*, 105:133–155, 1998.

[36] 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.

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

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

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

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

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

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

[43] M. J. Rossman and R. J. Twery. A solution to the travelling salesman problem. *Operations Research*, 6:687, 1958. Abstract E3.1.3.

[44] A. Saltelli, S. Tarantola, F. Campolongo, and M. Ratto. *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*. John Wiley & Sons, Ltd., West Sussex, England, 2004.

[45] H. P. Williams. *Model Building in Mathematical Programming*. John Wiley and Sons, Chichester, 2nd edition, 1985.

[46] W. Zhang. Complete anytime beam search. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 425–430, Madison, WI, July 1998.

[47] W. Zhang. Depth-first branch-and-bound vs. local search: A case study. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, pages 930–935, Austin, Texas, July 2000.

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

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