ALENEX 2001 Proceedings, Springer Lecture Notes in Computer Science 2153, pp. 32-59

The Asymmetric Traveling Salesman Problem: Algorithms, Instance Generators, and Tests

Jill Cirasella[1], David S. Johnson[2], Lyle A. McGeoch[3], and Weixiong Zhang[4]*

[1] Boston Architectural Center Library
320 Newbury Street, Boston, MA 02115
jill.cirasella@the-bac.edu
[2] AT&T Labs – Research, Room C239,
Florham Park, NJ 07932-0971, USA
dsj@research.att.com
[3] Department of Mathematics and Computer Science
Amherst College, Amherst, MA 01002
lam@cs.amherst.edu
[4] Department of Computer Science, Washington University
Box 1045, One Brookings Drive, St. Louis, MO 63130
zhang@cs.wustl.edu

Abstract. The purpose of this paper is to provide a preliminary report on the first broad-based experimental comparison of modern heuristics for the asymmetric traveling salesmen problem (ATSP). There are currently three general classes of such heuristics: classical tour construction heuristics such as Nearest Neighbor and the Greedy algorithm, local search algorithms based on re-arranging segments of the tour, as exemplified by the Kanellakis-Papadimitriou algorithm [KP80], and algorithms based on patching together the cycles in a minimum cycle cover, the best of which are variants on an algorithm proposed by Zhang [Zha93]. We test implementations of the main contenders from each class on a variety of instance types, introducing a variety of new random instance generators modeled on real-world applications of the ATSP. Among the many tentative conclusions we reach is that no single algorithm is dominant over all instance classes, although for each class the best tours are found either by Zhang's algorithm or an iterated variant on Kanellakis-Papadimitriou.

1 Introduction

In the traveling salesman problem, one is given a set of N cities and for each pair of cities c_i, c_j a distance $d(c_i, c_j)$. The goal is to find a permutation π of the cities that minimizes

$$\sum_{i=1}^{N-1} d\left(c_{\pi(i)}, c_{\pi(i+1)}\right) + d\left(c_{\pi(N)}, c_{\pi(1)}\right)$$

* Supported in part by NSF grants IRI-9619554 and IIS-0196057 and DARPA Cooperative Agreement F30602-00-2-0531.

Most of the research on heuristics on this problem has concentrated on the symmetric case (the STSP), where $d(c,c') = d(c',c)$ for all pairs of cities c, c'. Surveys such as [Rei94,JM97] experimentally examine the performance of a wide variety of heuristics on reasonably wide sets of instances, and many papers study individual heuristics in more detail. For the general not-necessarily-symmetric case, typically referred to as the "asymmetric TSP" (ATSP), there are far fewer publications, and none that comprehensively cover the current best approaches. This is unfortunate, as a wide variety of ATSP applications arise in practice. In this paper we attempt to begin a more comprehensive study of the ATSP.

The few previous studies of the ATSP that have made an attempt at covering multiple algorithms [Rep94,Zha93,Zha00,GGYZ] have had several drawbacks. First, the classes of test instances studied have not been well-motivated in comparison to those studied in the case of the symmetric TSP. For the latter problem the standard testbeds are instances from `TSPLIB` [Rei91] and randomly-generated two-dimensional point sets, and algorithmic performance on these instances seems to correlate well with behavior in practice. For the ATSP, `TSPLIB` offers fewer and smaller instances with less variety, and the most commonly studied instance classes are random distance matrices (asymmetric and symmetric) and other classes with no plausible connection to real applications of the ATSP.

For this study, we have constructed seven new instance generators based on a diverse set of potential ATSP applications, and we have tested a comprehensive set of heuristics on classes produced using these generators as well as the traditional random distance matrix generators. We also have tested the algorithms on the ATSP instances of `TSPLIB` and our own growing collection of instances from actual real-world applications.

We have attempted to get some insight into how algorithmic performance scales with instance size, at least within instance classes. To do this, we have generated test suites of ten 100-city instances, ten 316-city instances, three 1000-city instances, and one 3162-city instance for each class, the numbers of cities going up by a factor of approximately $\sqrt{10}$ at each step. At present, comprehensive study of larger instances is difficult, given that the common interface between our generators and the algorithms is a file of all N^2 inter-city distances, which would be over 450 megabytes for a 10,000-city instance. Fortunately, trends are already apparent by the time we reach 3162 cities.

We also improve on previous studies in the breadth of the algorithms that we cover. Current ATSP heuristics can be divided into three classes: (1) classical tour construction heuristics such as Nearest Neighbor and the Greedy algorithm, (2) local search algorithms based on re-arranging segments of the tour, as exemplified by the Kanellakis-Papadimitriou algorithm [KP80], and (3) algorithms based on patching together the cycles in a minimum cycle cover (which can be computed as the solution to an Assignment Problem, i.e., by constructing a minimum weight perfect bipartite matching). Examples of this last class include the algorithms proposed in [Kar79,KS85] and the $O(\log N)$ worse-case ratio "Repeated Assignment" algorithm of [FGM82]. We cover all three classes, including recent improvements on the best algorithms in the last two.

In the case of local search algorithms, previous studies have not considered implementations that incorporate such recent innovations from the symmetric TSP world as "don't-look bits" and chained (iterated) local search. Nor have they made use of the recent observation of [Glo96] that the best "2-bridge" 4-opt move can be found in time $O(N^2)$ rather than the naive $O(N^4)$. Together, these two ideas can significantly shift the competitive balance.

As to cycle-patching algorithms, these now seem to be dominated by the previously little-noted "truncated depth-first branch-and-bound" heuristic of Zhang [Zha93]. We present the first independent confirmation of the surprising results claimed in that paper, by testing both Zhang's own implementation and a new one independently produced by our first author.

A final improvement over previous ATSP studies is our use of the Held-Karp (HK) lower bound on optimal tour length [HK70,HK71,JMR96] as our standard of comparison. Currently all we know theoretically is that this bound is within a factor of $\log N$ of the optimal tour length when the triangle inequality holds [Wil92]. In practice, however, this bound tends to lie within 1 or 2% of the optimal tour length whether or not the triangle inequality holds, as was observed for the symmetric case in [JMR96] and as this paper's results suggest is true for the ATSP as well. By contrast, the Assignment Problem lower bound (AP) is often 15% or more below the Held-Karp bound (and hence at least that far below the optimal tour length). The Held-Karp bound is also a significantly more reproducible and meaningful standard of comparison than "the best tour length so far seen," the standard used for example in the (otherwise well-done) study of [Rep94]. We computed Held-Karp bounds for our instances by performing the NP-completeness transformation from the ATSP to the STSP and then applying the publicly available `Concorde` code of [ABCC98], which has options for computing the Held-Karp bound as well as the optimal solution. Where feasible, we also applied `Concorde` in this way to determine the optimal tour length.

The remainder of this paper is organized as follows. In Section 2 we provide high-level descriptions of the algorithms whose implementations we study, with citations to the literature for more detail where appropriate. In Section 3 we describe and motivate our instance generators and the classes we generate using them. In Section 4, we summarize the results of our experiments and list some our tentative conclusions. This is a preliminary report on an ongoing study. A final section (Section 5) describes some of the additional questions we intend to address in the process of preparing the final journal version of the paper.

2 Algorithms

In this section we briefly describe the algorithms covered in this study and our implementations of them. Unless otherwise stated, the implementations are in `C`, but several were done in `C++`, which may have added to their running time overhead. In general, running time differences of a factor of two or less should not be considered significant unless the codes come from the same implementation family.

2.1 Tour Construction: Nearest Neighbor and Greedy

We study randomized ATSP variants of the classical Nearest Neighbor (`NN`) and Greedy (`GR`) algorithms. In the former one starts from a random city, and then successively goes to the nearest as-yet-unvisited city. In our implementation of the latter, we view the instance as a complete directed graph with edge lengths equal to the corresponding inter-city distances. Sort the edges in order of increasing length. Call an edge *eligible* if it can be added to the current set of chosen edges without creating a (non-Hamiltonian) cycle or causing an in- or out-degree to exceed one. Our algorithm works by repeatedly choosing (randomly) one of the two shortest eligible edges until a tour is constructed. Randomization is important if these are to be used for generating starting tours for local optimization algorithms, since a common way of using the latter is to perform a set of runs and taking the best.

Running times for these algorithms may be inflated from what they would be for stand-alone codes, as we are timing the implementations used in our local search code. These start by constructing ordered lists of the 20 nearest neighbors to and from each city for subsequent use by the local search phase of the algorithms. The tour construction code exploits these lists where possible, but the total time spend constructing the lists may exceed the benefit obtained. For `NN` in particular, where we have an independent implementation, we appear to be paying roughly a 50% penalty in increased running time.

2.2 Patched Cycle Cover

We implement the patching procedure described in [KS85]. We first compute a minimum cycle cover using a weighted bipartite matching (assignment problem) code. We then repeatedly select the two biggest cycles and combine them into the shortest overall cycle that can be constructed by breaking one edge in each cycle and patching together the two resulting directed paths. Despite the potentially superquadratic time for this patching procedure, in practice the running times for this algorithm are dominated by that for constructing the initial cycle cover (a potentially cubic computation). This procedure provides better results than repeatedly patching together the two shortest cycles.

A variety of alternative patching procedures are studied in [Rep94,GGYZ,GZ], several of which, such as the "Contract-or-Patch" heuristic of [GGYZ], obtain better results at the cost of greater running time. The increase in running time is typically less than that required by Zhang's algorithm (to be described below). On the other hand, Zhang's algorithm appears to provide tour quality that is always at least as good and often significantly better than that reported for any of these alternative patching procedures. In this preliminary report we wish to concentrate on the best practical heuristics rather than the fastest, and so have not yet added any of the alternative patching procedures to our test suite. The simple `PATCH` heuristic provides an interesting data point by itself, as its tour is the starting point for Zhang's algorithm.

2.3 Repeated Assignment

This algorithm was originally studied in [FGM82] and currently has the best proven worst-case performance guarantee of any polynomial-time ATSP heuristic (assuming the triangle inequality holds): Its tour lengths are at most $\log N$ times the optimal tour length. This is not impressive in comparison to the $3/2$ guarantee for the Christofides heuristic for the symmetric case, but nothing better has been found in two decades.

The algorithm works by constructing a minimum cycle cover and then repeating the following until a connected graph is obtained. For each connected component of the current graph, select a representative vertex. Then compute a minimum cycle cover for the subgraph induced by these chosen vertices and add that to the current graph. A connected graph will be obtained before one has performed more than $\log N$ matchings, and each matching can be no longer than the optimal ATSP tour which is itself a cycle cover. Thus the total edge length for the connected graph is at most $\log N$ times the length of the optimal tour. Note also that it must be strongly connected and all vertices must have in-degree equal to out-degree. Thus it is Eulerian, and if one constructs an Euler tour and follows it, shortcutting past any vertex previously encountered, one obtains an ATSP tour that by the triangle inequality can be no longer than the total edge length of the graph.

We have implemented two variants on this, both in `C++`. In the first (`RA`) one simply picks the component representatives randomly and converts the Euler tour into a ATSP tour as described. In the second (`RA+`) we use heuristics to find good choices of representatives and rather than following the Euler tour, we use a greedy heuristic to pick, for each vertex in turn that has in- and out-degree exceeding 1, the best ways to short-cut the Euler tour so as to reduce these degrees to 1. This latter approach improves the tours found by Christofides in the STSP by as much as 5% [JBMR]. The combination of the two heuristics here can provide even bigger improvements, depending on the instance class, even though `RA+` takes no more time than `RA`. Unfortunately, `RA` is often so bad that even the substantially improved results of `RA+` are not competitive with those for `PATCH`. To save space we shall report results for `RA+` only.

2.4 Zhang's Algorithm and its Variants

As described in [Zha93], Zhang's algorithm is built by truncating the computations of an AP-based branch-and-bound code that used depth first search as its exploration strategy. One starts by computing a minimum length cycle cover M_0 and determining an initial *champion* tour by patching as in `PATCH`. If this tour is no longer that M_0 (for instance if M_0 was itself a tour), we halt and return it. Otherwise, call M_0 the initial *incumbent* cycle cover, and let I_0, the set of *included* edges and X_0, the set of excluded edges, be initially empty. The variant we study in this paper proceeds as follows.

Inductively, the incumbent cycle cover M_i is the minimum length cycle cover that contains all edges from I_i and none of the edges from X_i, and we assume that M_i is shorter than the current champion tour and is not itself a tour. Let $C = \{e_1, e_2, \ldots, e_k\}$ be a cycle (viewed as a set of edges) of minimum size in M_i. As pointed out in [CT80], there are k distinct ways of breaking this cycle: One can force the deletion of e_1, retain e_1 and force the deletion of e_2, retain e_1 and e_2 and force the deletion of e_3, etc. We solve a new matching problem for each of these possibilities that is not forbidden by the requirement that all edges in I_i be included and all edges in X_i be excluded. In particular, for all h, $1 \leq h \leq k$ such that e_h is not in I_i and $X_i \cap \{e_j : 1 \leq j < h\} = \phi$ we construct a minimum cycle cover that includes all the edges in $I_i \cup \{e_j : 1 \leq j < h\}$ and includes none of the edges in $X_i \cup \{e_h\}$. (The exclusion of the edges in this latter set is forced by adjusting their lengths to a value exceeding the initial champion tour length, thus preventing their use in any subsequent viable child.) If one retains the data structures used in the construction of M_i each new minimum cycle cover can be computed using only one augmenting path computation.

Let us call the resulting cycle covers the *children* of M_i. Call a child *viable* if its length is less than the current champion tour. If any of the viable children is a tour and is better than the current champion, we replace the champion by the best of these (which in turn will cause the set of viable children to shrink, since now none of the children that are tours will be viable). If at this point there is no viable child, we halt and return the best tour seen so far. Otherwise, let the new incumbent M_{i+1} be a viable child of minimum length. Patch M_{i+1} and if the result is better than the current champion tour, update the latter. Then update I_i and X_i to reflect the sets of included and excluded edges specified in the construction of M_{i+1} and continue. This process must terminate after at most N^2 phases, since each phase adds at least one new edge to $I_i \cup X_i$, and so we must eventually either construct a tour or obtain a graph in which no cycle cover is shorter than the initial champion tour.

We shall refer to Zhang's `C` implementation of this algorithm as `ZHANG1`. We have also studied several variants. The two most significant are `ZHANG2` in which in each phase *all* viable children are patched to tours to see if a new champion can be produced, and `ZHANG0`, in which patching is only performed on M_0. `ZHANG2` produces marginally better tours than does `ZHANG1`, but at a typical cost of roughly doubling the running time. `ZHANG0` is only slightly faster than `ZHANG1` and produces significantly worse tours. Because of space restrictions we postpone details on these and other variants of `ZHANG1` to the final report.

That final report will also contain results for an independent implementation of a variant of `ZHANG1` by Cirasella, which we shall call `ZHANG1-C`. This code differs from `ZHANG1` in that it is implemented in `C++`, uses different tie-breaking rules, and departs from the description of `ZHANG1` in one detail: only the shortest viable child is checked for tour-hood. All things being equal, this last change cannot improve the end result even though it may lead to deeper searches. However, because of differences in tie-breaking rules in the rest of the code, `ZHANG1-C` often does find better tours than `ZHANG1` – roughly about as often as it finds worse ones. Thus future implementers should be able to obtain similar quality tours so long as they follow the algorithm description given above and break ties as they see fit. If running time is an issue, however, care should be exercised in the implementation of the algorithm to solve the AP. Because it uses a differently implemented algorithm for solving the assignment problem, `ZHANG1-C` is unfortunately a factor of 2 to 3 times slower than `ZHANG1`.

Note: All our implementations differ significantly from the algorithm called "truncated branch-and-bound" and studied in [Zha00]. The latter algorithm is allowed to backtrack if the current matching has no viable children, and will keep running until it encounters a viable matching that is a tour or runs out of branch-and-bound tree nodes to explore. For some of our test instances, this latter process can degenerate into almost a full search of the branch and bound tree, which makes this approach unsuitable for use as a "fast" heuristic.

2.5 Local Search: 3-Opt

Our local search algorithms work by first constructing a Nearest Neighbor tour, and then trying to improve it by various forms of tour rearrangement. In 3-Opt, the rearrangement we consider breaks the tour into three segments S_1, S_2, S_3 by deleting three edges, and then reorders the segments as S_2, S_1, S_3 and relinks them to form a new tour. If the new tour is shorter, it becomes our new current tour. This is continued until no such improvement can be found. Our implementation follows the schema described for the STSP in [JBMR] in sequentially choosing the endpoints of the edges that will be broken. We also construct near-neighbor lists to speed (and possibly limit) the search, and exploit "don't-look" bits to avoid repeating searches that are unlikely to be successful. Because of space limitations, this preliminary report will not report on results for 3-Opt itself, although it does cover the much more effective "iterated" algorithm based on 3-Opt, to be described below.

2.6 Local Search: Kanellakis-Papadimitriou Variants

Note that local search algorithms that hope to do well for arbitrary ATSP instances cannot reverse tour segments (as is done in many STSP heuristics), only reorder them. Currently the ultimate "segment-reordering" algorithm is the Kanellakis-Papadimitriou algorithm [KP80], which attempts to the extent possible to mimic the Lin-Kernighan algorithm for the STSP [LK73,JBMR,JM97]. It consists of two alternating search processes.

The first process is a variable-depth search that tries to find an improving k-opt move for some odd $k \geq 3$ by a constrained sequential search procedure modeled on that in Lin-Kernighan. As opposed to that algorithm, however, [KP80] requires that each of the odd h-opt moves encountered along the way must have a better "gain" than its predecessor. (In the Lin-Kernighan algorithm, the partial moves must all have positive gain, but gain is allowed to temporarily decrease in hopes that eventually something better will be found.)

The second process is a search for an improving 4-Opt move, for which K&P used a potentially $\Omega(N^4)$ algorithm which seemed to work well in practice. The original paper on Lin-Kernighan for the STSP [LK73] also suggested finding 4-Opt moves of this sort as an augmentation to the sequential search process (which was structurally unable to find them), but concluded that they were not worth the added computation time incurred. In our implementation of Kanellakis-Papadimitriou, we actually find the *best* 4-Opt move in time $O(N^2)$, using a dynamic programming approach suggested by [Glo96]. This makes the use of 4-Opt moves much more cost-effective; indeed they are necessary if one is to get the best tours for a given investment of running time using a Kanellakis-Papadimitriou variant.

Our implementation also strengthens the sequential search portion of the original K&P: We use neighbor lists and don't-look bits to speed the sequential search. We also take the Lin-Kernighan approach and allow temporary decreases in the net gain, and have what we believe are two improved versions of the search that goes from an h-opt move to an $(h+2)$-opt move, one designed to speed the search and one to make it more extensive. The basic structure of the algorithm is to perform sequential searches until no improving move of this type can be found, followed by a computation of the best 4-opt move. If this does not improve the tour we halt. Otherwise we perform it and go back to the sequential search phase. Full details are postponed to the final paper.

We have studied four basic variants on Kanellakis-Papadimitriou, with names constructed as follows. We begin with `KP`. This is followed by a "4" if the 4-opt search is included, and an `F` if the more extensive sequential search procedure is used. For this preliminary report we concentrate on `KP4` applied to Nearest Neighbor starting tours, which provides perhaps the best tradeoff between speed, tour quality, and robustness.

2.7 Iterated Local Search

Each of the algorithms in the previous two sections can be used as the engine for a "chained" or "iterated" local search procedure as proposed by [MOF92] to obtain significantly better tours. (Other ways on improving on basic local search algorithms, such as the dynamic programming approach of [SB96] do not seem to be as effective, although hybrids of this approach with iteration might be worth further study.) In an "iterated" procedure, one starts by running the basic algorithm once to obtain an initial champion tour. Then one repeats the following process some predetermined number of times:

> Apply a *random* 4-opt move to the current champion, and use the resulting tour as the starting tour during another run of the algorithm. If the resulting tour is better than the current champion, declare it to be the new champion.

Typically don't-look bits persist from one iteration to the next, which means that only 8 vertices are initially available as starting points for searches, which offers significant speedups.

In our implementations we choose uniformly from all 4-opt moves. Better performance may be possible if one biases the choice toward "better" 4-opt moves, as is done in the implementation of chained Lin-Kernighan for the STSP by Applegate et al. [ACR]. We leave the study of such potential improvements to future researchers, who can use our results as a guide.

We denote the iterated version of an algorithm `A` by `iA`, and in this report will concentrate on N-iteration `iKP4F` and $10N$-iteration `i3opt`. The former is the variant that tends to find the best tours while still running (usually) in feasible amounts of time and the latter is typically the fastest of the variants and was used in the code optimization application described in [YJKS97]. Searches for the best 4-opt move in `iKP4F` occur only on the first and last iterations, so as to avoid spending $\Omega(N^2)$ on each of the intermediate iterations.

2.8 STSP Algorithms

For the three instance classes we consider that are actually symmetric, we also include results for the Johnson-McGeoch implementations [JBMR,JM97] of Lin-Kernighan (`LK`) and N-iteration iterated Lin-Kernighan (`iLK`). These were applied to the symmetric representation of the instance, either as a upper-triangular distance matrix or, where the instances were geometric, as a list of city coordinates.

2.9 Lower Bound Algorithms

We have already described in the introduction how Held-Karp lower bounds `HK` and optimal solutions `OPT` were calculated using `Concorde`. Here we only wish to point out that while the times reported for `HK` are accurate, including both the time to compute the NP-completeness transformation and to run `Concorde` on the result, the times reported for `OPT` are a bit flakier, as they only include the time to run `Concorde` on the already-constructed symmetric instance with an initial upper bound taken from the better of `ZHANG1` and `iKP4`. Thus a conservative measure of the running time for `OPT` would require that we increase the time reported in the table by both the time for the better of these two heuristics and the time for `HK`. In most cases this is not a substantial increase.

All running times reported for `OPT` were measured in this way. For some of the larger size symmetric instances, we present tour length results without corresponding running times, as in these cases we cheated and found optimal tour lengths by applying `Concorde` directly to the symmetric representation with an upper bound supplied by `iLK`.

3 Instance Generators

In this section we describe and introduce shorthand names for our 12 instance generators, all of which were written in `C`, as well as our set of “real-world” instances.

3.1 Random Asymmetric Matrices (amat)

Our random asymmetric distance matrix generator chooses each distance $d(c_i, c_j)$ as an independent random integer x, $0 \le x \le 10^6$. Here and in what follows, "random" actually means pseudorandom, using an implementation of the shift register random number generator of [Knu81].

For these instances it is known that both the optimal tour length and the AP lower bound approach a constant (the same constant) as $N \to \infty$. The rate of approach appears to be faster if the upper bound U on the distance range is smaller, or if the upper bound is set to the number of cities N, a common assumption in papers about optimization algorithms for the ATSP (e.g., see [MP96,CDT95]). Surprisingly large instances of this type can be solved to optimality, with [MP96] reporting the solution of a 500,000-city instance. (Interestingly, the same code was unable to solve a 35-city instance from a real-world manufacturing application.)

Needless to say, there are no known practical applications of asymmetric random distances matrices or of variants such as ones in which $d(c_i, c_j)$ is chosen uniformly from the interval $[0, i + j]$, another popular class for optimizers. We include this class to provide a measure of comparability with past results, and also because it provides one of the stronger challenges to local search heuristics.

3.2 Random Asymmetric Matrices Closed under Shortest Paths (tmat)

One of the reasons the previous class is uninteresting is the total lack of correlation between distances. Note that instances of this type are unlikely to obey the triangle inequality and so algorithms designed to exploit the triangle inequality, such as the Repeated Assignment algorithm of [FGM82] will perform horribly on them. A first step toward obtaining a more reasonable instance class is thus to take a distance matrix generated by the previous generator and close it under shortest path computation. That is, if $d(c_i, c_j) > d(c_i, c_k) + d(c_k, c_j)$ then set $d(c_i, c_j) = d(c_i, c_k) + d(c_k, c_j)$ and repeat until no more changes can be made. This is also a commonly-studied class.

3.3 Random Symmetric Matrices (smat)

For this class, $d(c_i, c_j)$ is an independent random integer x, $0 \le x \le 10^6$ for each pair $1 \le i < j \le N$, and $d(c_i, c_j)$ is set to $d(c_j, c_i)$ when $i > j$. Again, there is no plausible application, but these are also commonly studied and at least provide a ground for comparison to STSP algorithms.

3.4 Random Symmetric Matrices Closed under Shortest Paths (tsmat)

This class consists of the instances of the previous class closed under shortest paths so that the triangle inequality holds, another commonly studied class for ATSP codes.

3.5 Random Two-Dimensional Rectilinear Instances (rect)

This is our final class of symmetric instances that have traditionally been used to test ATSP codes. It is a well-studied case of the STSP, providing useful insights in that domain. The cities correspond to random points uniformly distributed in a 10^6 by 10^6 square, and the distance is computed according to the rectilinear metric. We use the rectilinear rather than the more commonly-used Euclidean metric as this brings the instances closer to plausible ATSP applications, as we shall see below. In the STSP, experimental results for the Euclidean and rectilinear metrics are roughly the same [JBMR].

3.6 Tilted Drilling Machine Instances with Additive Norm (rtilt)

These instances correspond to the following potential ATSP application. One wishes to drill a collection of holes on a tilted surface, and the drill is moved using two motors. The first moves the drill to its new x-coordinate, after which the second moves it to its new y-coordinate. Because the surface is tilted, the second motor can move faster when the y-coordinate is decreasing than when it is increasing. Our generator places the holes uniformly in the 10^6 by 10^6 square, and has three parameters: u_x is the multiplier on $|\Delta x|$ that tells how much time the first motor takes, u_y^+ is the multiplier on $|\Delta y|$ when the direction is up, and u_y^- is the multiplier on $|\Delta y|$ when the direction is down.

Note that the previous class can be generated in this way using $u_x = u_y^+ = u_y^- = 1$. For the current class, we take $u_x = 1$, $u_y^+ = 2$, and $u_y^- = 0$. Assuming instantaneous movement in the downward direction may not be realistic, but it does provide a challenge to some of our heuristics, and has the interesting side effect that for such instances the AP- and HK-bounds as well as the optimal tour length are all exactly the same as if we had taken the symmetric instance with $u_x = u_y^+ = u_y^- = 1$. This is because in a cycle the sum of the upward movements is precisely balanced by the sum of the downward ones.

3.7 Tilted Drilling Machine Instances with Sup Norm (stilt)

For many drilling machines, the proper metric is the maximum of the times to move in the x and y directions rather than the sum. For this generator, holes are placed as before and we have the same three parameters, although now the distance is the maximum of $u_x|\Delta x|$ and $u_y^-|\Delta y|$ (downward motion) or $u_y^+|\Delta y|$ (upward motion). We choose the parameters so that downward motion is twice as fast as horizontal motion and upward motion is half as fast. That is we set $u_x = 2$, $u_y^+ = 4$, and $u_y^- = 1$.

3.8 Random Euclidean Stacker Crane Instances (crane)

In the *Stacker Crane Problem* one is given a collection of source-destination pairs s_i, d_i in a metric space where for each pair the crane must pick up an object at location s_i and deliver it to location d_i. The goal is to order these tasks so as to minimize the time spent by the crane going between tasks, i.e., moving from the destination of one pair to the source of the next one. This can be viewed as an ATSP in which city c_i corresponds to the pair s_i, d_i and the distance from c_i to c_j is the metric distance between d_i and s_j. (Technically, the goal may be to find a minimum cost Hamiltonian *path* rather than a cycle, but that is a minor issue.) Our generator has a single parameter $u \geq 1$, and constructs its source-destination pairs as follows.

First, we pick the source s uniformly from the 10^6 by 10^6 square. Then we pick two integers x and y uniformly and independently from the interval $[-10^6/u, 10^6/u]$. The destination is then the vector sum $s + (x, y)$. To preserve a sense of geometric locality, we want the typical destination to be closer to its source than are all but a bounded number of other sources. Thus, noting that for an N-city instance of this sort the expected number of sources in a $10^6/\sqrt{N}$ by $10^6/\sqrt{N}$ is 1, we generated our instances using u as approximately $\sqrt{n}$. In particular we took $u = 10, 18, 32, 56$ for $N = 100, 316, 1000, 3162$. Preliminary experiments suggested that if we had simply fixed u at 10, the instances would have behaved more and more like random asymmetric ones as N got larger.

Note that these instances do not necessarily obey the triangle inequality (since the time for traveling from source to destination is not counted), although there are probably few large violations.

3.9 Disk Drive Instances (`disk`)

These instances attempt to capture some of the structure of the problem of scheduling the read head on a computer disk, although we ignore some technicalities, such as the fact that tracks get shorter as one gets closer to the center of the disk. This problem is similar to the stacker crane problem in that the files to be read have a start position and an end position in their tracks. Sources are generated as before, but now the destination has the same y-coordinate as the source. To determine the destination's x-coordinate, we generate a random integer $x \in [0, 10^6/u]$ and add it to the x-coordinate of the source, but do so modulo 10^6, thus capturing the fact that tracks can wrap around the disk. The distance from a destination to the next source is computed based on the assumption that the disk is spinning in the x-direction at a given rate and that the time for moving in y direction is proportional to the distance traveled (a slightly unrealistic assumption given the need for acceleration and deceleration) at a significantly slower rate. To get to the next source we first move to the required y-coordinate and then wait for the spinning disk to deliver the x-coordinate to us. For our instances in this class, we set $u = 10$ and assumed that y-direction motion was 10 times slower than x-direction motion. Full details are postponed to the final paper. Note that this is another class where the triangle inequality need not be strictly obeyed.

3.10 Pay Phone Coin Collection Instances (coins)

These instances model the problem of collecting money from pay phones in a grid-like city. We assume that the city is a k by k grid of city blocks. The pay phones are uniformly distributed over the boundaries of the blocks. Although all the streets (except the loop surrounding the city) are two-way, the money collector can only collect money from pay phones on the side of the street she is currently driving on, and is not allowed to make "U-turns" either between or at corners. This problem becomes trivial if there are so many pay phones that most blocks have one on all four of their sides. Our class is thus generated by letting k grow with n, in particular as the nearest integer to $10\sqrt{N}$.

3.11 No-Wait Flowshop Instances (shop50)

The no-wait flowship was the application that inspired the local search algorithm of Kanellakis and Papadimitriou. In a k-processor no-wait flowshop, a job $\bar{u}$ consists of a sequence of tasks $(u_1, u_2, \ldots, u_k)$ that must be worked on by a fixed sequence of machines, with processing of u_{i+1} starting on machine $i+1$ as soon as processing of u_i is complete on machine i. This models situations where for example we are processing heated materials that must not be allowed to cool down, or where there is no storage space to hold waiting jobs.

In our generator, the task lengths are independently chosen random integers between 0 and 1000, and the distance from job $\bar{v}$ to job $\bar{u}$ is the minimum possible amount be which the finish time for u_k can exceed that for v_k if $\bar{u}$ is the next job to be started after $\bar{v}$. A version of this class with $k = 5$ processors was studied in [Zha93,Zha00], but for randomly generated instances with this few processors the AP bound essentially equals the optimal tour length, and even `PATCH` averaged less than 0.1% above optimal. To create a bit more of an algorithmic challenge in this study, we increased the number of processors to 50.

3.12 Approximate Shortest Common Superstring Instances (super)

A very different application of the ATSP is to the shortest common superstring problem, where the distance from string A to string B is the length of B minus the length of the longest prefix of B that is also a suffix of A. Unfortunately, although this special case of the ATSP is NP-hard, real-world instances tend to be easy [Gia] and we were unable to devise a generator that produced non-trivial instances. We thus have modeled what appears to be a harder problem, *approximate* shortest common superstring. By this we mean that we allow the corresponding prefixes and suffixes to only approximately match, but penalize the mismatches. In particular, the distance from string A to string B is the length of B minus $\max\{j + 2k$: there is a prefix of B of length j that matches a suffix of A in all but k positions$\}$. Our generator uses this metric applied to random binary strings of length 20.

3.13 Specific Instances: TSPLIB and Other Sources (realworld)

In addition to our randomly-generated instance classes, and as a sanity check for our results on those classes, we have also tested a variety of specific "real-world" instances from TSPLIB and other sources. Our collection includes the 27 ATSP instances currently in TSPLIB plus 20 new instances from additional applications. The full list is given in Table 14, but here is a summary of the sources and applications involved.

The TSPLIB instances are as follows: The four rbg instances come from a stacker crane application. The two ft instances arose in a problem of optimally sequencing tasks in the coloring plant of a resin production department, as describe in [FT92]. The 17 ftv instances, described in [FTV94,FT97], come from a pharmaceutical delivery problem in downtown Bologna, with instances ftv90 through ftv160 derived from ftv170 by deleting vertices as described in [FT97]. Instances ry48p and kro124p are symmetric Euclidean instances rendered asymmetric by slight random perturbations of their distance matrices as described in [FT92]. Instance p43 comes from a scheduling problem arising in chemical engineering. Instance br17 is from an unknown source.

Our additional instances come from five sources. big702 models an actual (albeit outdated) coin collection problem and was provided to us by Bill Cook. The three td instances came from a detailed generator constructed by Bruce Hillyer of Lucent for the problem of scheduling reads on a specific tape drive, based on actual timing measurements. The nine dc instances come from a table compression application and were supplied by Adam Buchsbaum of AT&T Labs. The two code instances came from a code optimization problem described in [YJKS97]. The five atex instances come from a robotic motion planning problem and were provided to us by Victor Manuel of the Carlos III University of Madrid.

4 Results and Conclusions

Our experiments were performed on what is today a relatively slow machine: a Silicon Graphics Power Challenge machine with 196 Mhz MIPS R10000 processors and 1 Megabyte 2nd level caches. This machine has 7.6 Gigabytes of main memory, shared by 31 of the above processors. Our algorithms are sequential, so the parallelism of the machine was exploited only for performing many individual experiments at the same time. For the randomized algorithms NN, RA+, KP4, i3opt, iKP4F, the results we report are averages over 5 or more runs for each instance (full details in the final report).

Tables 2 through 13 present average excesses over the HK bound and running times in user seconds for the algorithms we highlighted in Section 2 and the testbeds we generated using each of our 12 random instance generators. In each table, algorithms are ordered by their average tour quality for instances of size 1000. Our first conclusion is that for each of the classes, at least one of our algorithms can find a fairly good solution quickly. See Table 1, which for each of the classes lists the algorithm that gets the best results on 1000-city instances in less than 80 user seconds. Note that in all cases, at least one algorithm is able to find tours within 9% of the HK bound within this time bound, and in all but three cases we can get within 2.7%. There is, however, a significant variety in the identity of the winning algorithm, with `ZHANG1` the winner on five of the classes, `iKP4F` in four, `i3opt` in two, and `KP4` in one. If time is no object, `iKP4F` wins out over the other local search approaches in all 12 classes. However, its running time grows to over 19,000 seconds in the case of the 1000-city `rtilt` and `shop50` instances, and in the latter case it is still bested by `ZHANG1`.

Class	Winner	% Excess	Seconds	% HK-AP	Symmetry	Triangle
`amat`	`ZHANG1`	.04	19.8	.05	.9355	.163
`tmat`	`ZHANG1`	.00	6.4	.03	.9794	1.000
`smat`	`iKP4F`	5.26	67.0	19.70	1.0000	.163
`tsmat`	`i3opt`	2.42	40.7	17.50	1.0000	1.000
`rect`	`iKP4F`	2.20	43.5	17.16	1.0000	1.000
`rtilt`	`KP4`	8.33	38.7	17.17	.8760	1.000
`stilt`	`i3opt`	4.32	75.3	14.60	.9175	1.000
`crane`	`iKP4F`	1.27	60.8	5.21	.9998	.934
`disk`	`ZHANG1`	.02	16.6	.34	.9990	.646
`coins`	`iKP4F`	2.66	43.7	14.00	.9999	1.000
`shop50`	`ZHANG1`	.03	50.7	.15	.8549	1.000
`super`	`ZHANG1`	.21	52.9	1.17	.9782	1.000

Table 1. For the 1000-city instances of each class, the algorithm producing the best average tour among those that finish in less than 80 seconds, the average percent by which that tour exceeds the HK bound, and the average running times. In addition we list the average percentage shortfall of the AP bound from the HK bound, and the average symmetry and triangle inequality metrics as defined in the text below.

The table also includes three instance measurements computed in hopes of finding a parameter that correlates with algorithmic performance. The first is the percentage by which the AP bound falls short of the HK bound. The second is a measure of the asymmetry of the instance. For this we construct the "symmetrized" version I' of an instance I with distances $d'(c_i, c_j) = d'(c_j, c_i)$ set to be the average of $d(c_i, c_j)$ and $d(c_j, c_i)$. Our symmetry metric is the ratio of the standard deviation of $d'(c_i, c_j)$, $i \neq j$, to the standard deviation for $d(c_i, c_j)$, $i \neq j$. A value of 1 implies the original graph was symmetric. Our third measure attempts to quantify the extent by which the instance violates the triangle inequality. For each pair c_i, c_j of distinct cities we let $d'(c_i, c_j)$ be the minimum of $d(c_i, c_j)$ and $\min\{d(c_i, c_k) + d(c_k, c_j) : 1 \leq k \leq N\}$. Our measure is then the average, over all pairs c_i, c_j, of $d'(c_i, c_j)/d(c_i, c_i)$. A value of 1 implies that the instance obeys the triangle inequality.

Based on the table, there is a clear correlation between the presence of a small HK-AP gap and the superiority of `ZHANG1`, but no apparent correlations with the other two instance metrics. Note that when the HK-AP gap is close to zero and `ZHANG1` is the winner, it does extremely well, never being worse than .21% above the HK bound. A plausible explanation is that when the AP bound is close to the HK bound it is also close to optimal, which means that an optimal tour is not much longer (and hence perhaps not very different) from a minimum cycle cover. Algorithms such as `ZHANG1` (and `RA+` and `PATCH`) that start by computing a minimum cycle cover are thus likely to perform well, and `ZHANG1`, by doing a partial branch-and-bound search for a tour, is most likely to do best. Conversely, when the HK-AP gap is large, `ZHANG1` is at a disadvantage, for example producing tours that are more than 11% above the HK bound for classes `rect`, `rtilt`, `stilt`, and `coins`. (See Tables 6, 7, 8, and 11.)

Here are some more tentative conclusions based on the results reported in Tables 2 through 13 and additional experiments we have performed.

1. Simple ATSP tour construction heuristics such as Nearest Neighbor (`NN`) and Greedy (`GR`) can perform abysmally, with `NN` producing tours for some classes that are over 300% above the HK-bound. (`GR` can be even worse, although `PATCH` is only really bad for symmetric random distance matrices.)
2. Reasonably good performance can be consistently obtained from algorithms in both local search and cycle-patching classes: `KP4` averages less than 15.5% above HK for each instances size of all twelve of our instance classes and `ZHANG1` averages less than 13% above HK for each. Running times for both are manageable when $N \leq 1000$, with the averages for all the 1000-city instance classes being 8 minutes or less for both algorithms.
3. Running time (for all algorithms) can vary widely depending on instance class. For 3162 cities, the average time for `ZHANG1` ranges from 71 to roughly 22,500 seconds. For `KP4` the range is a bit less extreme: from 43 to 2358.
4. For the instance classes yielding the longer running times, the growth rate tends to be substantially more explosive for the Zhang variants than for the local search variants, suggesting that the latter will tend to be more usable as instance size increases significantly beyond 3162 cities. As an illustration, see Figures 1 and 2 which chart the change in solution quality and running time as N increases for four of our instance classes. Running times are normalized by dividing through by N^2, the actual size of the instance and hence a lower bound on the asymptotic growth rate. Although these charts use the same sort of dotted line to represent both local search algorithms `KP4` and `iKP4F`, this should be easy to disambiguate since the former always produces worse tours in less time. Similarly, the same sort of solid line is used to represent both AP-based algorithms `PATCH` and `ZHANG1`, with the former always producing worse tours more quickly. By using just two types of lines, we can more clearly distinguish between the two general types of algorithm. Figure 1 covers classes `amat` and `super`, both with small HK-AP gaps, although for the latter the gap may be growing slightly with N as opposed to declining toward 0 as it appears to do for classes `amat`, `tmat`, `disk`, and `shop50`. The tour quality chart for `amat` is typical of these four, with both `PATCH` and `ZHANG1` getting better as N increases, and both `KP4` and `iKP4F` getting worse. The difference for class `super` is that `PATCH` now gets worse as N increases, and `ZHANG1` does not improve. As to running times (shown in the lower two charts of the figure), `KP4` seems to be running in roughly quadratic time (its

normalized curve is flat), whereas `iKP4F` is not only slower but seems to have a slightly higher running time growth rate. `ZHANG1` has the fastest growth rate, substantially worse than quadratic, and on the `super` class has already crossed over with `iKP4F` by 1000 cities. Figure 2 covers two classes where the HK-AP gap is substantial, and we can see marked differences from the previous classes.

5. As might be expected, the Repeated Assignment Algorithm (`RA+`) performs extremely poorly when the triangle inequality is substantially violated. It does relatively poorly for instances with large HK-AP gaps. And it always loses to the straightforward `PATCH` algorithm, even though the latter provides no theoretical guarantee of quality.
6. Although the Kanellakis-Papadimitriou algorithm was motivated by the no-wait flowshop application, both it and its iterated variants are outperformed on these by *all* the AP-based algorithms, including `RA+`.
7. For all instances classes, optimal solutions are relatively easy to obtain for 100-city instances, the maximum average time being less than 30 minutes per instance (less than 6 minutes for all but one class). For 7 of the 12 classes we were able to compute optimal solutions for all our test instances with 1000 cities or less, never using more than 5 hours for any one instance. For all classes the time for optimization was an order of magnitude worse than the time for `ZHANG1`, but for three classes (`tmat`, `disk`, and `shop50`) it was actually faster than that for `iKP4F`.
8. The currently available ATSP heuristics are still not as powerful in the ATSP context as are the best STSP heuristics in the STSP context. The above times are far in excess of those needed for similar performance levels on standard instance classes for the STSP. Moreover, for symmetric instances, our ATSP codes are easily bested by ones specialized to the STSP (both in the case of approximation and of optimization).
9. It is difficult to reproduce sophisticated algorithm implementations exactly, even if one is only interested in solution quality, not running time. Although our two implementations of `ZHANG1` differ only in their tie-breaking rules and one rarely-invoked halting rule, the quality of the tours they produce can differ significantly on individual instances, and for some instance classes one or the other appears to dominate its counterpart consistently. Fortunately, we can in essence be said to have “reproduced” the original results in that the two implementation do produce roughly the same quality of tours overall.
10. The task of devising random instance generators that produce nontrivial instances (ones for which none of the heuristics consistently finds the optimal solution) is a challenge, even when one builds in structure from applications. One reason is that without carefully constraining the randomization, it can lead you to instances where the AP bound quickly approaches the optimal tour length. A second reason, as confirmed by some of our real-world instances, is that many applications *do* give rise to such easy instances.

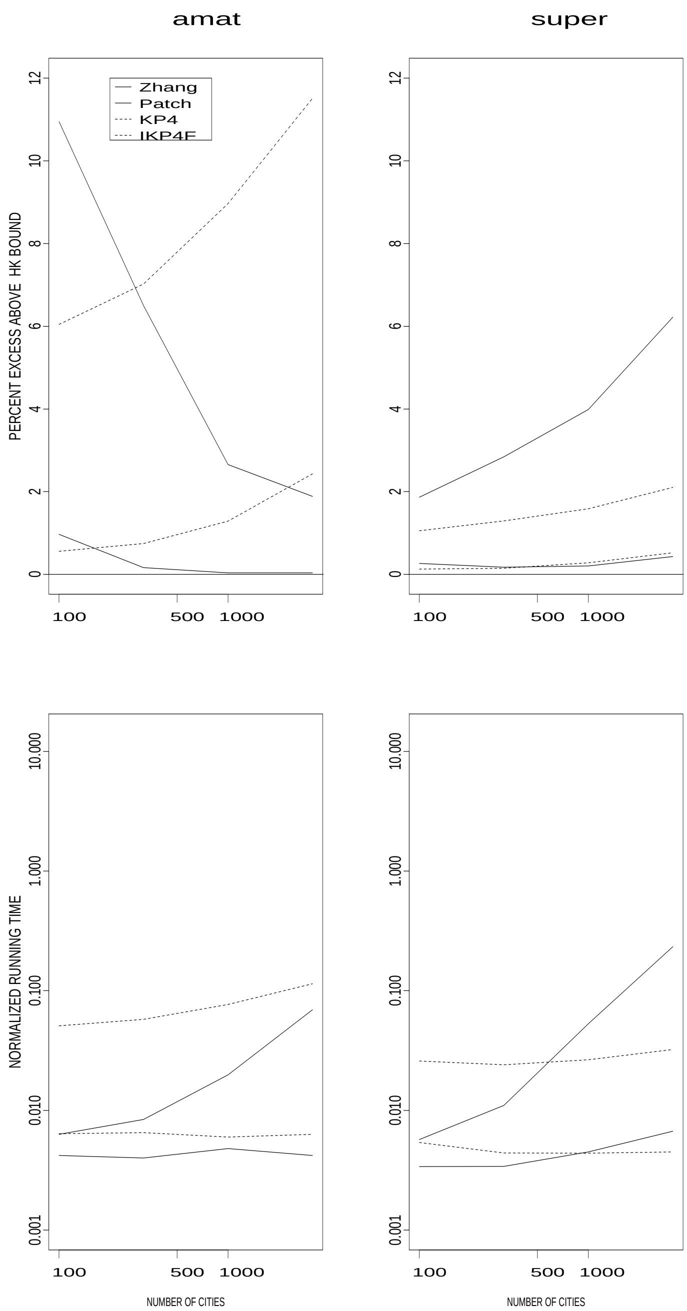

Fig. 1. Tour quality and running time as a function of the number of cities for classes `amat` and `super` and algorithms `KP4`, `iKP4F`, `PATCH`, and `ZHANG1`. The same type lines are used for the local search algorithms `KP4` and `iKP4F` and for the AP-based algorithms `PATCH` and `ZHANG1`, but the correspondence between line and algorithm should be clear: Algorithm `iKP4F` is always better and slower than `KP4`, and algorithm `ZHANG1` is always better and slower than `PATCH`.

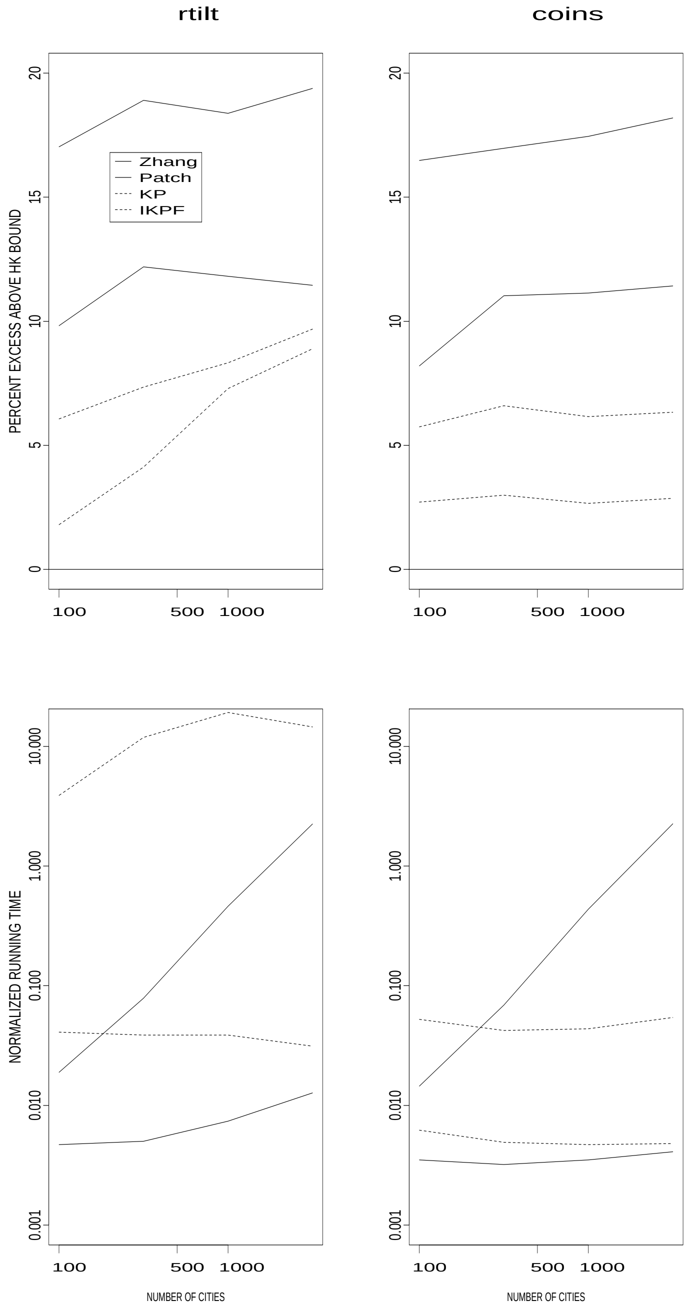

Fig. 2. Tour quality and running time as a function of the number of cities for classes `rtilt` and `coins` and algorithms `KP4`, `iKP4F`, `PATCH`, and `ZHANG1`. For these classes the HK-AP gap exceeds 10%.

Table 14 presents results for our `realworld` testbed. For space reasons we restrict ourselves to reporting the three instance parameters and the results for `ZHANG1` and for `iKP4F`, our local search algorithm that produces the best tours. Here we report the excess over the optimal tour length rather than over the HK bound, as we were able to determine the optimal tour length for all 47 instances. (In some cases this took less time than it took to run `iKP4F`, although `dc895` took almost 20 hours and `atex8` required special handling from David Applegate of the `Concorde` team, as well as much more time.) Instances are grouped by class, within classes they are ordered by number of cities, and the classes are ordered roughly in order of increasing HK-AP gap.

A first observation is that, true to form, `ZHANG1` does very well when the HK-AP gap is close to zero, as is the case for the `rbg` stacker crane instances, the `td` tape drive instances, and the `dc` table compression instances. Surprisingly, it also does well on many instances with large HK-AP gaps, even ones with gaps larger than 97%. Indeed, on only two instances is it worse than 3.5% above optimal, with its worst performance being roughly 11% above optimal for `ft53`. Note that `ZHANG1` beats `iKP4F` more often than it loses, winning on 25 of the 47 instances and tying on an additional 5 (although many of its wins are by small amounts). If one is willing to spend as much as two hours on any given instance, however, then `iKP4F` is a somewhat more robust alternative. It stays within 3.02% of optimal for all 47 instances and within 1% for all but four.

5 Future Work

This is a preliminary report on an ongoing study. Some of the directions we are continuing to explore are the following.

1. **Running Time.** In order to get a more robust view of running time growth for the various algorithms, we are in the process of repeating our experiments on a variety of machines. We also intend to compute detailed counts of some of the key operations involved in the algorithms, in hopes of spotting correlations between these and running times.
2. **Variants on Zhang's Algorithm and on Kanellakis-Papadimitriou.** As mentioned in Section 2, there are several variants on Zhang's algorithm not covered in this report, and we want to study these in more detail in hopes of determining how the various algorithmic choices affect the tradeoffs between tour quality and running time. A preliminary result along this line is that `ZHANG2`, which patches all viable children, typically does marginally better than `ZHANG1`, although at a cost of doubling or tripling the overall running time. Similar issues arise with our local search algorithms, where we want to learn more about the impact of the 4-Opt moves in the Kanellakis-Papadimitriou algorithm, and the differences in behavior between our two versions of sequential search.
3. **Other Patching Algorithms.** We intend to more completely characterize the advantages of Zhang's algorithm versus the best of the simpler patching algorithms by performing more head-to-head comparisons.

4. **Starting Tours for Local Search.** This is an important issue. We chose to use Nearest Neighbor starting tours as they provided the most robust results across all classes. However, for several classes we could have obtained much better results for `KP4` and `iKP4F` had we used Greedy or `PATCH` starts, and for those classes where `ZHANG1` is best but still leaves some room for improvement, it is natural to consider using `ZHANG1` as our starting heuristic. Preliminary results tell us that there is no universal best starting algorithm, and moreover that the ranking of heuristics as to the quality of their tours is for many instance classes quite different from their ranking as to the quality of the tours produced from them by local optimization.
5. **More, and More Portable, Generators.** Our current instance generators use a random number generator that does not produce the same results on all machines. For the final paper we plan to rerun our experiments on new test suites created using a truly portable random number generator. This will help confirm the independence of our conclusions from the random number generator used and will also provide us with a testbed that can be distributed to others by simply providing the generators, seeds, and other input parameters. We also hope to add more instance classes and grow our realworld test set.

References

[ABCC98] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. On the solution of traveling salesman problems. *Doc. Mathematica J. der Deutschen Mathematiker-Vereinigung*, ICM III:645–656, 1998. The `Concorde` code is available from the website `http://www.keck.caam.rice.edu/concorde.html`.

[ACR] D. Applegate, W. Cook, and A. Rohe. Chained Lin-Kernighan for large traveling salesman problems. To appear. A postscript draft is currently available from the website `http://www.caam.rice.edu/~bico/`.

[CDT95] G. Carpaneto, M. Dell'Amico, and P. Toth. Exact solution of large-scale, asymmetric traveling salesman problems. *ACM Trans. Mathematical Software*, 21(4):394–409, 1995.

[CT80] G. Carpaneto and P. Toth. Some new branching and bounding criteria for the asymmetric traveling salesman problem. *Management Science*, 26:736–743, 1980.

[FGM82] A. M. Frieze, G. Galbiati, and F. Maffioli. On the worst-case performance of some algorithms for the asymmetric traveling salesman problem. *Networks*, 12:23–39, 1982.

[FT92] M. Fischetti and P. Toth. An additive bounding procedure for the asymmetric travelling salesman problem. *Math. Programming A*, 53:173–197, 1992.

[FT97] M. Fischetti and P. Toth. A polyhedral approach to the asymmetric traveling salesman problem. *Management Sci.*, 43:1520–1536, 1997.

[FTV94] M. Fischetti, P. Toth, and D. Vigo. A branch and bound algorithm for the capacitated vehicle routing problem on directed graphs. *Operations Res.*, 42:846–859, 1994.

[GGYZ] F. Glover, G. Gutin, A. Yeo, and A. Zverovich. Construction heuristics for the asymmetric TSP. *European J. Operations Research.* to appear.

[Gia] R. Giancarlo. Personal communication, September 2000.

[Glo96] F. Glover. Finding a best traveling salesman 4-opt move in the same time as a best 2-opt move. *J. Heuristics*, 2(2):169–179, 1996.

[GZ] G. Gutin and A. Zverovich. Evaluation of the contract-or-patch heuristic for the asymmetric TSP. Manuscript, 2000.

[HK70] M. Held and R. M. Karp. The traveling salesman problem and minimum spanning trees. *Operations Res.*, 18:1138–1162, 1970.

[HK71] M. Held and R. M. Karp. The traveling salesman problem and minimum spanning trees: Part II. *Math. Prog.*, 1:6–25, 1971.

[JBMR] D. S. Johnson, J. L. Bentley, L. A. McGeoch, and E. E. Rothberg. Near-Optimal Solutions to Very Large Traveling Salesman Problems. Monograph, to appear.

[JM97] D. S. Johnson and L. A. McGeoch. The traveling salesman problem: A case study in local optimization. In E. H. L. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley and Sons, Ltd., Chichester, 1997.

[JMR96] D. S. Johnson, L. A. McGeoch, and E. E. Rothberg. Asymptotic experimental analysis for the Held-Karp traveling salesman bound. In *Proc. 7th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 341–350. Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[Kar79] R. M. Karp. A patching algorithm for the nonsymmetric traveling-salesman problem. *SIAM J. Comput.*, 8(4):561–573, 1979.

[Knu81] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms (2nd Edition)*. Addison-Wesley, Reading, MA, 1981. See pages 171–172.

[KP80] P. C. Kanellakis and C. H. Papadimitriou. Local search for the asymmetric traveling salesman problem. *Oper. Res.*, 28(5):1066–1099, 1980.

[KS85] R. M. Karp and J. M. Steele. Probabilistic analysis of heuristics. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B Shmoys, editors, *The Traveling Salesman Problem*, pages 181–205. John Wiley and Sons, Chichester, 1985.

[LK73] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Res.*, 21:498–516, 1973.

[MOF92] O. Martin, S. W. Otto, and E. W. Felten. Large-step Markov chains for the TSP incorporating local search heuristics. *Operations Res. Lett.*, 11:219–224, 1992.

[MP96] D. L. Miller and J. F. Pekny. Exact solution of large asymmetric traveling salesman problems. *Science*, 251:754–761, 15 September 1996.

[Rei91] G. Reinelt. TSPLIB – A traveling salesman problem library. *ORSA J. Comput.*, 3(4):376–384, 1991. The `TSPLIB` website is `http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95/`.

[Rei94] G. Reinelt. *The Traveling Salesman: Computational Solutions of TSP Applications*. LNCS 840. Springer-Verlag, Berlin, 1994.

[Rep94] B. W. Repetto. *Upper and Lower Bounding Procedures for the Asymmetric Traveling Salesman Problem.* PhD thesis, Graduate School of Industrial Administration, Carnegie-Mellon University, 1994.

[SB96] N. Simonetti and E. Balas. Implementation of a linear time algorithm for certain generalized traveling salesman problems. In *Integer Programming and Combinatorial Optimization: Proc. 5th Int. IPCO Conference*, LNCS 840, pages 316–329, Berlin, 1996. Springer-Verlag.

[Wil92] D. Williamson. Analysis of the Held-Karp lower bound for the asymmetric TSP. *Operations Res. Lett.*, 12:83–88, 1992.

[YJKS97] C. Young, D. S. Johnson, D. R. Karger, and M. D. Smith. Near-optimal intraprocedural branch alignment. In *Proceedings 1997 Symp. on Programming Languages, Design, and Implementation*, pages 183–193. ACM, 1997.

[Zha93] W. Zhang. Truncated branch-and-bound: A case study on the asymmetric TSP. In *Proc. of AAAI 1993 Spring Symposium on AI and NP-Hard Problems*, pages 160–166, Stanford, CA, 1993.

[Zha00] W. Zhang. Depth-first branch-and-bound versus local search: A case study. In *Proc. 17th National Conf. on Artificial Intelligence (AAAI-2000)*, pages 930–935, Austin, TX, 2000.

	Percent above HK				Time in Seconds			
Alg	100	316	1000	3162	100	316	1000	3162
`NN`	195.23	253.97	318.79	384.90	.05	.46	4.8	50
`RA+`	86.60	100.61	110.38	134.14	.10	1.25	26.3	621
`i3OPT`	5.10	13.27	27.12	45.53	.75	4.67	36.9	273
`KP4`	5.82	6.95	8.99	11.51	.06	.65	6.0	63
`PATCH`	10.95	6.50	2.66	1.88	.04	.40	4.8	42
`iKP4F`	.56	.74	1.29	2.43	.51	5.75	76.7	1146
`ZHANG1`	.97	.16	.04	.04	.06	.84	19.8	694
`OPT`	.29	.08	.02	–	24.73	194.58	1013.6	–
`HK`	.00	.00	.00	.00	1.05	7.80	95.0	1650
`MATCH`	-0.65	-0.29	-0.04	-0.04	.04	.40	4.6	40

Table 2. Results for class `amat`: Random asymmetric instances.

	Percent above HK				Time in Seconds			
Alg	100	316	1000	3162	100	316	1000	3162
`NN`	38.20	37.10	37.55	36.66	.05	.44	4.4	46
`KP4`	1.41	2.23	3.09	4.03	.17	1.16	10.9	115
`i3OPT`	.30	.85	1.63	2.25	1.53	7.08	40.1	231
`RA+`	4.88	3.10	1.55	.46	.09	1.13	20.2	653
`iKP4F`	.09	.14	.41	.65	8.87	56.17	489.4	4538
`PATCH`	.84	.64	.17	.00	.04	.39	4.7	68
`ZHANG1`	.06	.01	.00	.00	.05	.49	6.4	71
`OPT`	.03	.00	.00	–	6.16	20.29	91.9	–
`HK`	.00	.00	.00	.00	1.15	8.03	113.3	2271
`MATCH`	-0.34	-0.16	-0.03	.00	.04	.39	4.6	62

Table 3. Results for class `tmat`: `amat` instances closed under the shortest paths.

	Percent above HK				Time in Seconds			
Alg	100	316	1000	3162	100	316	1000	3162
RA+	693.90	2263.56	7217.91	24010.30	.10	1.21	10.0	703
PATCH	126.39	237.59	448.48	894.17	.04	.40	4.9	64
NN	139.28	181.18	233.10	307.11	.05	.46	4.8	50
i3OPT	8.43	15.57	25.55	39.84	.75	4.73	36.2	285
KP4	10.59	11.50	13.13	15.31	.06	.64	5.8	62
ZHANG1	5.52	5.84	5.97	5.67	.12	4.01	200.4	9661
iKP4F	3.98	4.34	5.26	6.21	.52	5.44	67.0	1007
LK	1.72	2.22	3.43	4.68	.18	1.75	11.0	29
iLK	.15	.42	1.17	2.35	.45	5.22	31.4	258
OPT	.10	.04	.01	–	16.30	378.09	1866.5	–
HK	.00	.00	.00	.00	1.44	12.90	267.4	7174
MATCH	-19.83	-20.73	-19.66	-20.21	.04	.39	4.6	56

Table 4. Results for class `smat`: Random symmetric instances.

	Percent above HK				Time in Seconds			
Alg	100	316	1000	3162	100	316	1000	3162
RA+	32.75	40.97	47.35	49.44	.10	1.22	26.3	557
PATCH	14.46	19.37	23.68	26.42	.04	.36	3.7	45
NN	23.64	23.18	22.83	22.15	.04	.44	4.4	47
KP4	3.12	3.71	4.15	4.71	.23	1.43	10.9	122
ZHANG1	3.49	4.06	3.38	3.51	.17	6.82	373.2	16347
i3OPT	1.12	1.88	2.42	3.18	2.05	8.45	40.7	244
LK	.62	1.17	1.50	2.09	.41	2.68	10.8	42
iKP4F	.91	1.19	1.44	1.98	20.69	88.26	479.1	5047
iLK	.12	.27	.44	.85	8.12	42.19	135.7	696
OPT	.10	.12	.11	–	28.00	549.04	12630.3	–
HK	.00	.00	.00	.00	1.83	15.15	217.8	885
MATCH	-17.97	-19.00	-17.50	-17.92	.04	.35	3.4	37

Table 5. Results for class `tsmat`: `smat` instances closed under shortest paths.

	Percent above HK				Time in Seconds			
Alg	100	316	1000	3162	100	316	1000	3162
RA+	64.02	73.93	78.90	85.72	.10	1.14	20.2	494
NN	26.39	27.51	26.11	26.55	.06	.47	4.9	51
PATCH	17.90	18.73	18.46	19.39	.04	.37	3.9	45
ZHANG1	9.75	12.40	12.23	12.19	.18	7.78	466.4	22322
KP4	5.11	5.06	5.00	5.17	.06	.55	5.2	53
i3OPT	2.02	2.57	2.66	3.09	.66	3.95	29.6	292
iKP4F	1.75	2.16	2.20	2.32	.48	4.41	43.5	483
LK	1.32	1.55	1.77	1.92	.07	.26	.7	1
iLK	.69	.72	.79	.82	.42	2.47	17.8	109
OPT	.68	.67	.68	–	94.82	–	–	–
HK	.00	.00	.00	.00	1.93	14.51	205.0	1041
MATCH	-20.42	-17.75	-17.17	-16.84	.04	.36	3.6	37

Table 6. Results for class `rect`: Random 2-dimensional rectilinear instances. Optima for 316- and 1000-city instances computed by symmetric code.

	Percent above HK				Time in Seconds			
Alg	100	316	1000	3162	100	316	1000	3162
RA+	61.95	73.47	78.27	82.03	.13	1.90	49.9	1424
NN	28.47	28.28	27.52	24.60	.06	.47	5.0	51
PATCH	17.03	18.91	18.38	19.39	.05	.50	7.4	127
i3OPT	1.69	4.22	12.97	18.41	4.61	43.59	254.3	967
ZHANG1	9.82	12.20	11.81	11.45	.19	7.86	460.8	22511
KP4	6.06	7.35	8.33	9.68	.41	3.87	38.7	312
iKP4F	1.80	4.12	7.29	8.89	38.90	1183.94	19209.0	145329
OPT	.68	.67	.68	–	152.05	–	–	–
HK	.00	.00	.00	.00	2.06	15.98	226.6	875
MATCH	-20.42	-17.75	-17.17	-16.84	.05	.49	7.1	116

Table 7. Results for class `rtilt`: Tilted drilling machine instances with additive norm. Optima for 316- and 1000-city instances computed by symmetric code applied to equivalent symmetric instances.

Alg	Percent above HK				Time in Seconds			
	100	316	1000	3162	100	316	1000	3162
RA+	55.79	62.76	65.03	71.48	.10	1.10	22.1	566
NN	30.31	30.56	27.62	24.42	.05	.54	4.9	112
PATCH	23.33	22.79	23.18	24.41	.04	.44	5.6	69
ZHANG1	10.75	13.99	12.66	12.86	.17	7.39	423.7	9817
KP4	8.57	8.79	8.80	8.15	.14	1.01	7.7	73
i3OPT	3.29	3.95	4.32	4.28	1.38	9.38	75.3	601
iKP4F	3.00	3.54	3.96	4.13	7.17	161.40	4082.4	72539
OPT	1.86	–	–	–	1647.46	–	–	–
HK	.00	.00	.00	.00	1.86	15.53	174.5	864
MATCH	-18.41	-14.98	-14.65	-14.04	.04	.42	5.2	62

Table 8. Results for class `stilt`: Tilted drilling machine instances with sup norm.

Alg	Percent above HK				Time in Seconds			
	100	316	1000	3162	100	316	1000	3162
RA+	40.80	50.33	53.60	54.91	.10	1.20	18.1	590
NN	40.72	41.66	43.88	43.18	.05	.46	4.8	50
PATCH	9.40	10.18	9.45	8.24	.04	.38	4.0	53
KP4	4.58	4.45	4.78	4.26	.07	.57	5.3	52
ZHANG1	4.36	4.29	4.05	4.10	.10	3.52	172.6	7453
i3OPT	1.98	2.27	1.95	2.12	.67	3.99	31.2	270
iKP4F	1.46	1.79	1.27	1.36	.66	6.33	60.8	696
OPT	1.21	1.30	–	–	216.54	11290.42	–	–
HK	.00	.00	.00	.00	1.43	12.80	925.2	1569
MATCH	-7.19	-6.34	-5.21	-4.43	.04	.38	4.0	44

Table 9. Results for class `crane`: Random Euclidean stacker crane instances.

Alg	Percent above HK				Time in Seconds			
	100	316	1000	3162	100	316	1000	3162
NN	96.24	102.54	115.51	161.99	.06	.48	5.0	54
RA+	86.12	58.27	42.45	25.32	.11	1.66	48.9	1544
KP4	2.99	3.81	5.81	9.17	.07	.99	21.2	760
i3OPT	.97	2.32	3.89	5.29	.73	5.66	62.4	687
iKP4F	.56	.48	.96	1.77	.64	22.62	1240.9	61655
PATCH	9.40	2.35	.88	.30	.04	.47	7.5	176
ZHANG1	1.51	.27	.02	.01	.08	1.00	16.6	247
OPT	.24	.06	.01	–	22.66	70.99	398.2	–
HK	.00	.00	.00	.00	1.27	8.71	139.1	4527
MATCH	-2.28	-0.71	-0.34	-0.11	.04	.46	7.2	168

Table 10. Results for class `disk`: Disk drive instances.

	Percent above HK				Time in Seconds			
Alg	100	316	1000	3162	100	316	1000	3162
RA+	52.74	64.95	68.78	71.20	.09	1.00	16.3	329
NN	26.08	26.71	26.80	25.60	.05	.42	4.4	46
PATCH	16.48	16.97	17.45	18.20	.04	.32	3.5	41
ZHANG1	8.20	11.03	11.14	11.42	.14	6.88	435.9	22547
KP4	5.74	6.59	6.15	6.34	.06	.49	4.7	48
i3OPT	2.98	3.37	3.48	3.83	.64	3.71	28.3	257
iKP4F	2.71	2.99	2.66	2.87	.52	4.23	43.7	542
OPT	1.05	1.49	–	–	356.69	67943.26	–	–
HK	.00	.00	.00	.00	1.66	13.75	141.9	924
MATCH	-15.04	-13.60	-13.96	-13.09	.03	.31	3.1	33

Table 11. Results for class `coins`: Pay phone coin collection instances.

	Percent above HK				Time in Seconds			
Alg	100	316	1000	3162	100	316	1000	3162
NN	16.97	14.65	13.29	11.87	.04	.42	6.4	46
i3OPT	.49	4.02	9.23	10.76	14.85	67.91	266.5	940
KP4	1.57	2.92	3.88	4.54	3.12	31.43	306.1	2358
iKP4F	.49	2.36	3.66	4.51	321.70	2853.11	19283.1	74617
RA+	4.77	2.77	1.69	1.05	.15	3.03	74.4	2589
PATCH	1.15	.59	.39	.24	.05	.86	21.7	611
ZHANG1	.20	.08	.03	.01	.09	1.83	50.7	1079
OPT	.05	.02	.01	–	82.03	565.69	2638.2	–
HK	.00	.00	.00	.00	2.29	18.22	269.9	3710
MATCH	-0.50	-0.22	-0.15	-0.07	.05	.85	21.6	590

Table 12. Results for class `shop50`: No-wait flowshop instances.

	Percent above HK				Time in Seconds			
Alg	100	316	1000	3162	100	316	1000	3162
NN	8.57	8.98	9.75	10.62	.04	.38	3.9	43
RA+	4.24	5.22	6.59	8.34	.02	.84	5.4	393
PATCH	1.86	2.84	3.99	6.22	.03	.34	4.5	67
KP4	1.05	1.29	1.59	2.10	.05	.44	4.4	45
i3OPT	.28	.61	1.06	1.88	.59	3.19	20.2	138
iKP4F	.13	.15	.28	.52	.26	2.40	26.5	323
ZHANG1	.27	.17	.21	.43	.06	1.10	52.9	2334
OPT	.05	.03	.01	–	10.74	130.40	4822.2	–
HK	.00	.00	.00	.00	1.02	8.14	104.6	2112
MATCH	-1.04	-1.02	-1.17	-1.61	.03	.33	4.3	61

Table 13. Results for class `super`: Approximate shortest common superstring instances.

Class	N	% HK-AP	Symm.	Triangle	% Above Opt iKP4F	ZHANG1	Time in Seconds iKP4F	ZHANG1
rbg358	358	.00	.9517	.5749	.79	.00	342.26	.20
rbg323	323	.00	.9570	.6108	.30	.00	414.53	.17
rbg403	403	.00	.9505	.5511	.11	.00	743.99	.54
rbg443	443	.00	.9507	.5642	.09	.00	908.12	.54
big702	702	.00	.9888	1.0000	.21	.00	1136.52	.61
td100.1	101	.00	.9908	.9999	.00	.00	39.90	.01
td316.10	317	.00	.9910	.9998	.00	.00	1950.07	.12
td1000.20	1001	.00	.9904	.9994	.00	.00	720.57	.88
dc112	112	.87	.9996	1.0000	.31	.13	904.31	.20
dc126	126	3.78	.9999	.9983	.64	.21	1533.84	.25
dc134	134	.36	.9973	1.0000	.55	.18	714.98	.23
dc176	176	.81	.9982	1.0000	.59	.09	1448.29	.31
dc188	188	1.22	.9997	1.0000	.52	.23	674.15	.28
dc563	563	.33	.9967	1.0000	.78	.09	4379.74	19.86
dc849	849	.09	.9947	1.0000	.61	.00	5666.96	51.06
dc895	895	.68	.9994	1.0000	.58	.42	7214.80	111.65
dc932	932	2.48	.9999	.9998	.27	.13	4611.56	85.68
code198	198	.09	.7071	.9776	.00	.00	114.24	.03
code253	253	17.50	.7071	.9402	.00	.28	267.14	.30
kro124p	100	5.61	.9992	.9724	1.28	3.29	.34	.06
ry48p	48	12.40	.9994	.9849	1.92	2.17	.16	.01
ft53	53	14.11	.8739	1.0000	.01	10.96	7.99	.02
ft70	70	1.75	.9573	1.0000	.02	.47	4.83	.02
ftv33	34	7.85	.9838	1.0000	3.02	3.50	.11	.01
ftv35	36	5.24	.9823	1.0000	.03	1.09	.11	.00
ftv38	39	5.04	.9832	1.0000	.00	1.05	.12	.00
ftv44	45	4.03	.9850	1.0000	1.30	.00	.12	.00
ftv47	48	5.53	.9832	1.0000	.49	.68	.17	.01
ftv55	56	9.41	.9853	1.0000	.00	.75	.25	.02
ftv64	65	4.79	.9850	1.0000	.33	.00	.37	.01
ftv70	71	7.49	.9844	1.0000	.08	.00	.30	.03
ftv90	91	5.77	.9849	1.0000	.08	.44	.47	.01
ftv100	101	5.52	.9873	1.0000	.13	.00	.62	.02
ftv110	111	4.18	.9872	1.0000	.12	.00	1.00	.04
ftv120	121	4.93	.9877	1.0000	.33	.00	1.70	.05
ftv130	131	3.61	.9884	1.0000	.27	.00	1.52	.08
ftv140	141	4.12	.9883	1.0000	.21	.00	1.23	.04
ftv150	151	3.17	.9887	1.0000	.40	.00	1.18	.04
ftv160	161	3.29	.9890	1.0000	.24	.11	1.43	.07
ftv170	171	3.10	.9890	1.0000	.45	.36	1.52	.10
br17	17	100.00	.9999	.8474	.00	.00	.11	.01
p43	43	97.36	.7565	.8965	.01	.05	23.81	.02
atex1	16	98.23	.8569	1.0000	.44	6.07	.07	.00
atex3	32	98.37	.9498	1.0000	.03	.41	1.69	.02
atex4	48	96.92	.9567	1.0000	.00	.12	2.22	.02
atex5	72	97.49	.9629	1.0000	.15	.38	7.77	.06
atex8	600	97.69	.9895	1.0000	.99	2.60	2112.70	47.43

Table 14. Results for `realworld` test instances.