DRAFT (approximately final) of a chapter that appeared in
"The Traveling Salesman Problem and its Variations,"Gutin
and Punnen (eds), Kluwer Academic Publishers, 2002, 445-487

Chapter 1

EXPERIMENTAL ANALYSIS OF HEURISTICS FOR THE ATSP

David S. Johnson
AT&T Labs – Research, Room C239, Florham Park, NJ 07932, USA
dsj@research.att.com

Gregory Gutin
Department of Computer Science, Royal Holloway, University of London
Egham, Surrey TW20 0EX, UK, G.Gutin@rhul.ac.uk

Lyle A. McGeoch
Department of Mathematics and Computer Science,
Amherst College, Amherst, MA 01002, USA, lam@cs.amherst.edu

Anders Yeo
Department of Computer Science, Royal Holloway, University of London
Egham, Surrey TW20 0EX, UK, anders@cs.rhul.ac.uk

Weixiong Zhang
Department of Computer Science, Washington University, Box 1045
One Brookings Drive, St. Louis, MO 63130, USA, zhang@cs.wustl.edu

Alexei Zverovitch
Department of Computer Science, Royal Holloway, University of London
Egham, Surrey TW20 0EX, UK, A.Zverovitch@rhul.ac.uk

1. Introduction

In this chapter, we consider what approaches one should take when confronting a real-world application of the asymmetric TSP, that is, the general TSP in which the distance $d(c, c')$ from city c to city c' need not equal the reverse distance $d(c', c)$. As in the previous chapter on heuristics for the symmetric TSP, we will discuss the performance of a variety of heuristics exhibiting a broad range of tradeoffs between running time and tour quality.

The situation for the ATSP is different from that for the STSP in several respects. First, no single type of instance dominates the applications of the ATSP in the way that two-dimensional geometric instances dominate the applications of the STSP. General-purpose ATSP heuristics must thus be prepared to cope with a much broader range of instance structures and are less likely to be robust across all of them. Moreover, the variety of possible instance structures rules out the possibility of a common linear-size instance representation. Thus general-purpose codes rely on instance representations that simply list the $N(N-1)$ inter-city distances, where N is the number of cities. This makes it difficult to handle large instances: if each distance requires say four bytes of memory, a 10,000-city instance would require roughly 400 megabytes of memory. Thus the experimental study of asymptotic behavior is much less advanced than in the case of the STSP. Whereas `TSPLIB`[1] contains STSP instances with as many as 85,900 cities, the largest ATSP instance it contains has only 443 cities and over half of its ATSP instances have fewer than 100 cities.

A second significant difference between the STSP and the ATSP is that the latter appears to be a more difficult problem, both with respect to optimization and approximation. Whereas all the randomly generated and `TSPLIB` instances with 1000 cities or less covered in the chapter on the STSP could be optimally solved using the `Concorde`[2] optimization code under its default settings, many 316-city ATSP instances from [10] remain unsolved. Moreover, as we shall see, there are plausibly realistic instance classes where *none* of our heuristics get as close to optimal as several STSP heuristics did for all the geometric instances in the testbeds covered in the STSP chapter.

[1]`TSPLIB` is a database of instances for the TSP and related problems created and maintained at `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/` by Gerd Reinelt and described in [29].
[2]`Concorde` is a publicly available software package written by Applegate, Bixby, Chvátal, and Cook, described in [2].

Finally, the experimental study of heuristics for the ATSP is much less advanced. Until recently there has been no broad-based study covering a full range of ATSP heuristics and what ATSP studies there have been, such as [13, 30, 31, 32], have concentrated mostly on the `TSPLIB` instances and randomly generated instances with no obvious connection to applications. Nor has there been a formal ATSP Challenge like the STSP Challenge that served as the main resource for Chapter 9. Fortunately, a first attempt at a more comprehensive study has recently appeared: a 2001 paper by Cirasella et al. [10]. This chapter will build on the results in [10] by looking at them in new ways and augmenting them with results for several additional heuristics.

The chapter is organized as follows. In Section 2 we describe the standards we use for measuring tour quality and comparing running times and the instance classes that make up our testbeds. Section 3 describes the various heuristics we cover, divided into groups based on approach taken (modified STSP tour construction heuristics, heuristics that solve assignment problems as a subroutine, and local search heuristics). Section 4 then describes the results for the various heuristics on our testbeds and how they compare, and attempts to derive meaningful insights based on the wealth of disparate data provided. We conclude in Section 5 with a summary of our major conclusions and suggestions for future research.

For a look at ATSP heuristics from a more theoretical point of view, see Chapter 6.

2. Methodology

In this chapter we evaluate heuristics on the basis of their performance on the collection of benchmark instances used in [10]. We begin in Section 2.1 by discussing the various measures of tour quality that we use in our comparisons (the optimal tour length, the Held-Karp lower bound, and the assignment problem lower bound) and the relations between them. In Section 2.2 we describe how running times were measured and the normalization process by which running times on different machines were compared (a variant on the approach taken in Chapter 9). In Section 2.3 we describe the benchmark instances and how they were obtained or generated.

2.1. Evaluating Tour Quality

The most natural standard for tour quality is the distance from the optimal solution, typically measured as the percentage by which the tour length exceeds the length of an optimal tour. In order to use this standard, one unfortunately must *know* the optimal solution value, and as remarked above, optimization technology for the ATSP is even less advanced than that for the STSP. Whereas optimal tour lengths were known for all the STSP testbed instances with 3,162-city cities or less, there are quite a few 316-city instances in our ATSP testbeds for which we have been unable to determine the optimal tour length.

Thus, as was the case for the STSP, we must settle for some computable lower bound on the optimal tour length as our standard of comparison. We shall actually consider two such bounds. Until recently, the most commonly studied lower bound for the ATSP was the *Assignment Problem* (AP) lower bound: Treat the distance matrix as representing the edge weights for a complete (symmetric) bipartite graph on $2N$ vertices, and compute the minimum-cost perfect matching for this graph, a computation that can be done in worst-case $\Theta(N^3)$ time but usually can be performed much more quickly. It is not difficult to see that this matching corresponds to a minimum-cost cover of the cities by vertex-disjoint directed simple cycles, and hence is a lower bound on the optimal tour length, which is the minimum cost cover of the cities by a single directed simple cycle. As we shall see, the AP bound is quite close to the optimal tour length for many of our instance classes. However, this is not always true. In the worst case the percent by which the optimal tour length exceeds it is not bounded by any constant even for $N = 4$. For five of our random instance classes the average gap is well over 10%.

To obtain a better lower bound, [10] proposed using the generalization of the Held-Karp STSP lower bound (HK bound) [15, 16] to the asymmetric case. As in the symmetric case, the asymmetric HK bound is the solution to the linear programming relaxation of the standard integer programming formulation of the problem. For the ATSP, that formulation is presented in Chapter 4. As in the symmetric case, we can use `Concorde` to compute it, although to compute the bound for an ATSP instance I, we first need to transform the instance to an STSP instance I' to which `Concorde` can be applied. We use the standard transformation that replaces each city c_i by three cities c_i^1, c_i^2, c_i^3 with an (undirected) path of 0-cost edges linking them in the given order. The directed arc (c_i, c_j) then corresponds to the undirected edge $\{c_i^3, c_j^1\}$. All edges not specified in this transformation are given lengths longer than the tour computed by Zhang's heuristic for the ATSP instance. It is easy to see that the optimal undirected tour for I' has the same length as the optimal directed tour for I and that the HK bounds likewise coincide. The (normalized) time to compute the HK bound by this process (including running Zhang's heuristic and performing the transformation) ranges from seconds for 100-city instances to 35 minutes for 3,162-city instances. For full running time details, see the tables of Section 4.6.

This same approach can be used to attempt to find optimal tours, simply by setting `Concorde`'s flags to make it do optimization instead of computing the Held-Karp bound. For this a good upper bound is needed, for which we would now recommend running Helsgaun's heuristic (different heuristics were used in [10]). When the optimum could be computed, the maximum average gap between it and the HK bound for any of our instance classes was 1.86%, with most averages being well under 1%. The maximum gap for any of the real-world instances in our testbed was 1.79%. Thus the HK bound appears to be a much more robust estimate of the optimal solution than the AP bound. Moreover, as we shall see, the gap between the AP and HK bounds has interesting correlations with algorithmic behavior.

2.2. Running Time Normalization

As in the chapter on STSP heuristics, this chapter summarizes results obtained on a variety of machines. Although we rely heavily on the results reported in [10], which were all obtained on the same machine (an SGI Power Challenge using 196 Mhz MIPS R10000 processors), we have augmented these with results from additional researchers, performed on their home machines. We thus again need a mechanism for comparing running times on different machines.

As in the STSP chapter, we base our comparisons on normalization factors derived from runs of a standard benchmark code. For the ATSP, our benchmark code is an implementation of the "Hungarian method" for solving the Assignment problem (available as part of the ATSP download from the TSP Challenge website [19]). It is intended to reflect ATSP computations more accurately than the Greedy geometric benchmark code used for the STSP. Contributors ran this code on 100-, 316-, 1,000-, and 3,162-city instances from the `rtilt` class described below on the same machine they used to test their own heuristics and reported the benchmark's running times along with the results for their own heuristics. The benchmark times were then used to construct size-dependent normalization factors by which their heuristics' running times could be multiplied to produce a rough estimate of what they would have been on our target machine. To facilitate comparisons with results from the STSP chapter, we use the same target machine: a Compaq ES40 with 500 Mhz Alpha processors and 2 gigabytes of main memory.

For more details on the conversion process (and its potential weaknesses), see Chapter 9. Based on limited tests, it appears that our normalization process is likely once again to introduce no more than a factor-of-two error in running time. Indeed, for the ATSP and its bench-

		N =	100	316	1000	3162
Class	Heuristic		Time in Seconds			
disk	Zhang	Normalized	.05	.56	6.4	105
		Actual	.09	.54	7.0	111
		Discrepancy	-44%	+4%	-9%	-5%
coin	Helsgaun	Normalized	1.40	17.45	205.4	4975
		Actual	.90	13.20	230.7	4522
		Discrepancy	+55%	+32%	-11%	+10%

Table 1.1. Comparisons between normalized 196 Mhz MIPS processor time and actual running time on the 500 Mhz Alpha target machine.

mark code, we may be slightly more accurate than that. See Table 1.1, where the predicted and actual running times on the target machine are summarized for two pairs of (heuristic,instance class). As in the symmetric case, we get both under- and over-estimates, but here the times are typically within 33% of each other except for the 100-city instances where running times are too small to estimate accurately.

2.3. Testbeds

Thirteen classes of instances were covered in [10]. Twelve of those classes were randomly generated. The generators for seven of the twelve classes were designed to produce instances with some of the structure that might arise in particular applications. The other five generators produced less-structured instances of types previously studied in the literature, three of which were actually symmetric instances viewed as asymmetric ones. For each class there were ten instances with 100 and 316 cities, three instances with 1,000 cities, and one instance with 3,162 cities. The thirteenth class consisted of "real-world" instances from TSPLIB and other sources, ranging in size from 43 to 1,000 cities.

For space reasons, in this chapter we consider a slightly restricted set of testbeds. First, we ignore all real-world instances with fewer than 100 cities, just as STSP instances with fewer than 1,000 cities were ignored in Chapter 9. (The lower threshold in the case of the ATSP reflects the fact that this problem appears to be harder in general.) All the real-world instances with 100 cities or less from [10] and all the 100-city randomly generated instances in our testbed can be solved to optimality in reasonable time using the technique described in Section 2.1. Moreover, the publicly available implementation of Helsgaun's heuristic [17] should suffice for most practical situations when instances are this small. It gets within 0.3% of optimal within a (normalized) second for all the real-world instances with less than 100 cities from [10], and as we shall see it averages two seconds or less to get within 0.1% of optimal for the 100-city instances in each of our random classes.

One can conceive of applications with a greater need for speed, for example the phylogenetic inference problem discussed by Moret et al. in [27, 28], where millions of ATSP instances with 37 cities or less need to be solved as part of the overall process. However, in such cases it typically becomes necessary to use extensive problem-specific algorithm engineering, which is well beyond the scope of this chapter. (In fact, Moret et al. ended up using a highly tuned branch-and-bound optimization code for their application [27].) Thus our lower bound of 100 cities seems quite reasonable.

In addition, we restrict ourselves to just two of the three symmetric instance classes from [10]: the ones corresponding to the Random Uniform Euclidean instances and Random Matrix instances of Chapter 9. We consider these classes mainly to illustrate the loss in tour quality that ATSP heuristics experience in comparison to their STSP counterparts. The third class, consisting of the closure under shortest paths of Random Matrix instances, adds little additional insight to this question.

We now say a bit about each of the twelve classes we do cover. For more details the interested reader is referred to [10] or to the code for the generators themselves, which together with associated `README` files is available from the DIMACS Implementation Challenge website [19].

Random Symmetric Matrices (`smat`). These are the Random Matrix instances of Chapter 9 written in asymmetric format. For this class, $d(c_i, c_j)$ is an independent random integer x, $0 \leq x \leq 10^6$ for each pair $1 \leq i < j \leq N$, and $d(c_i, c_j)$ is set to $d(c_j, c_i)$ when $i > j$. (Here and in what follows, "random" actually means pseudorandom, using an implementation of the shift register random number generator of [23, pages 171–172].) Such instances have no plausible application, but have commonly been studied in this context and at least provide a ground for comparison to STSP heuristics.

Random 2-Dimensional Rectilinear Instances (`rect`). These correspond to the Random Uniform Euclidean instances of Chapter 9 written in asymmetric format, except that we use the Rectilinear rather than the Euclidean metric. That is, the cities correspond to random points uniformly distributed in a 10^6 by 10^6 square, and the distance between points (x_1, y_1) and (x_2, y_2) is $|x_2 - x_1| + |y_2 - y_1|$. For STSP heuristics, tour quality and running times for the Euclidean and Rectilinear metrics are roughly the same [18].

Random Asymmetric Matrices (`amat`). The random asymmetric distance matrix generator chooses each distance $d(c_i, c_j)$ as an independent random integer x, $0 \leq x \leq 10^6$. For these instances it is known that both the optimal tour length and the AP 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 [8, 26]). Surprisingly large instances of this type can be solved to optimality, with [26] 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. We include this class to provide a measure of comparability with past studies.

Shortest-Path Closure of `amat` (`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, i.e., there can be three cities c_1, c_2, c_3 such $d(c_1, c_3) > d(c_1, c_2) + d(c_2, c_3)$. Thus heuristics designed to exploit the triangle inequality, such as the Repeated Assignment heuristic of [11] may perform horribly on them. A somewhat more reasonable instance class can be obtained by taking Random Asymmetric Matrices and closing them 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.

Tilted Drilling Machine Instances, Additive Norm (`rtilt`). These instances correspond to the following potential 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. The generator starts with an instance of `rect` and modifies it based on three parameters: u_x, the multiplier on $|\Delta x|$ that tells how much time the first motor takes, and u_y^+ and u_y^-, the multipliers on $|\Delta y|$ when the direction is up/down. For this class, the parameters $u_x = 1$, $u_y^+ = 2$, and $u_y^- = 0$ were chosen, which yields the same optimal tour lengths as the original symmetric `rect` instances because in a cycle the sum of the upward movements is precisely balanced by the sum of the downward ones. Related classes based on perturbing the Euclidean metric were proposed by Kataoko and Morito in [22] to model the problem of scheduling temperature-dependent operations in a factory where cooling was faster than heating.

Tilted Drilling Machine Instances, Sup Norm (`stilt`). For many drilling machines, the motors operate in parallel and so the proper metric is the maximum of the times to move in the x and y directions rather than the sum. This generator has the same three parameters as for `rtilt`, 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). For this class, the parameters $u_x = 2$, $u_y^+ = 4$, and $u_y^- = 1$ were chosen.

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. The generator has a single parameter $u \geq 1$, and constructs its source-destination pairs as follows. The sources are generated uniformly in the same way that cities are generated in an instance of `rect`. Then for each source s we pick two integers x and y uniformly and independently from the interval $[-10^6/u, 10^6/u]$. The destination is the vector sum $s + (x, y)$. In order to preserve a sense of geometric locality, we let u vary as a function of N, choosing values so that the expected number of other sources that are closer to a given source than its destination is roughly a constant, independent of N. For more details see [10]. These instances do not necessarily obey the triangle inequality since the time for traveling from source to destination is not counted.

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. 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 again generated as in `rect` instances, but now destinations have the same y-coordinates as their sources. To determine the x-coordinate of a destination, we generate a random integer $x \in [0, 10^6/u]$ and add it to the x-coordinate of its source 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 the y direction is proportional to the distance traveled 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 more details see [10].

Pay Phone Coin Collection Instances (`coin`). 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 with 2-way streets running between them and a 1-way street running around the exterior boundary of the city. The pay phones are uniformly distributed over the boundaries of the blocks. We can only collect from a pay phone if it is on the same side of the street as we are currently driving on, and we cannot make "U-turns" either between or at street corners. Finding the shortest route is trivial if there are so many pay phones that most blocks have one on all four of their sides. This class is thus generated by letting k grow with N, in particular as the nearest integer to $10\sqrt{N}$.

No-Wait Flowshop Instances (`shop`). The no-wait flowship was the application that inspired the local search heuristic of Kanellakis and Papadimitriou [20]. 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 performed by a fixed sequence of machines. The processing of u_{i+1} must start on machine $i+1$ as soon as processing of u_i is complete on machine i. This models the processing of heated materials that must not be allowed to cool down and situations where there is no storage space to hold waiting jobs. These instances have $k = 50$ processors and task lengths are independently chosen random integers between 0 and 1000. The distance from job $\bar{v}$ to job $\bar{u}$ is the minimum possible amount by which the finish time for u_k can exceed that for v_k if $\bar{u}$ is the next job to be started after $\bar{v}$.

Approx. Shortest Common Superstring Instances (`super`). This class is intended to capture some of the structure in a computational biology problem relevant to genome reconstruction. Given a collection of strings C, we wish to find a short superstring S in which all are (at least approximately) contained. If we did not allow mismatches the distance from string A to string B would be the length of B minus the length of the longest prefix of B that is also a suffix of A. Here we add a penalty equal to twice the number of mismatches, and 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$\}$. The generator uses this metric applied to random binary strings of length 20.

Specific Instances: `TSPLIB` and Other Sources (`realworld`). This collection includes the 13 ATSP instances with 100 or more cities currently in `TSPLIB` plus 16 new instances from additional applications. Applications covered include stacker crane problems (the `rbg` instances), vehicle routing (the `ftv` instances), coin collection (`big702`), robotic motion planning (`atex600`, called `atex8` in [10]), tape drive reading (the `td` instances), code optimization (the `code` instances), and table compression (the `dc` instances). We omit the `TSPLIB` instance `kro124p` since it is simply a perturbed random Euclidean instance, rather than from an application. For more details, see [10].

In what follows we shall consider which measurable properties of instances correlate with heuristic performance. Likely candidates include (1) the gap between the AP and HK bounds, (2) the extent to which the distance metric departs from symmetry, and (3) the extent to which it violates the triangle inequality. The specific metrics we use for these properties are as follows. For (1) we use the percentage by which the AP bound falls short of the HK bound. For (2) we use the ratio of the average value of $|d(c_i,c_j) - d(c_j,c_i)|$ to the average value of $|d(c_i,c_j) + d(c_j,c_i)|$, a quantity that is 0 for symmetric matrices and has a maximum value of 1. For (3) we first compute, for each pair c_i, c_j of distinct cities, the minimum of $d(c_i,c_j)$ and $\min\{d(c_i,c_k) + d(c_k,c_j) : \ 1 \le k \le N\}$ (call it $d'(c_i,c_j)$). The metric is then the average, over all pairs c_i, c_j, of $(d(c_i,c_j) - d'(c_i,c_j))/d(c_i,c_j)$. A value of 0 implies that the instance obeys the triangle inequality. (The latter two metrics are different and perhaps more meaningful than the ones used in [10].)

Tables 1.2 and 1.3 report the values for these metrics on our randomly generated classes and real-world instances respectively. For the random instances, average values are given for $N = 100$, 316, and 1,000. In Table 1.2 the classes are ordered by increasing value of the HK-AP gap for the 1,000-city entry. In Table 1.3 instances of the same type are ordered by increasing gap, and types are ordered by increasing average gap. For the random instance classes, there seems to be little correlation between the three metrics, although for some there is a dependency on the number N of cities. For the `realworld` instances there are some apparent correlations between metrics for instances of the same type, e.g., for the `ftv` instances, increasing HK-AP gap seems to go with increasing asymmetry, whereas for the `dc` instances increasing HK-AP gap seems to go with *decreasing* asymmetry (and increasing triangle inequality violations).

Table 1.3 also displays the percentage by which the optimal tour length exceeds the HK bound for our `realworld` instances. (The analogous information for our random instance classes is shown in the tables of Section 4.6.) Note that for these instances the OPT-HK gap is 0.03% or less whenever the HK-AP gap is less than 1%, a first suggestion of the significance of the latter metric.

	% HK-AP			Asymmetry			Triangle		
	100	316	1,000	100	316	1,000	100	316	1,000
`tmat`	.34	.16	.03	.232	.189	.165	–	–	–
`amat`	.64	.29	.05	.333	.332	.333	.633	.752	.837
`shop`	.50	.22	.15	.508	.498	.515	–	–	–
`disk`	2.28	.71	.34	.044	.045	.046	.250	.313	.354
`super`	1.04	1.02	1.17	.076	.075	.075	–	–	–
`crane`	7.19	6.34	5.21	.061	.035	.020	.101	.087	.066
`coin`	15.04	13.60	13.96	.010	.007	.003	–	–	–
`stilt`	18.41	14.98	14.65	.329	.333	.336	–	–	–
`rtilt`	20.42	17.75	17.17	.496	.500	.503	–	–	–
`rect`	20.42	17.75	17.17	–	–	–	–	–	–
`smat`	19.83	20.73	19.66	–	–	–	.632	.751	.837

Table 1.2. For the 100-, 316-, and 1000-city instances of each class, the average percentage shortfall of the AP bound from the HK bound and the average asymmetry and triangle inequality metrics as defined in the text. "–" stands for .000.

Instance	N	%OPT-HK	%HK-AP	Asymmetry	Triangle
rbg323	323	.00	.00	.206	.3892
rbg358	358	.00	.00	.226	.4251
rbg403	403	.00	.00	.231	.4489
rbg443	443	.00	.00	.229	.4358
td100	101	.00	.00	.115	.0001
td316	317	.00	.00	.115	.0002
td1000	1001	.00	.00	.117	.0006
big702	702	.00	.00	.131	–
dc849	849	.00	.09	.010	–
dc563	563	.01	.33	.008	–
dc134	134	.01	.36	.005	–
dc895	895	.03	.68	.003	–
dc176	176	.02	.81	.005	–
dc112	112	.02	.87	.002	–
dc188	188	.02	1.22	.001	–
dc932	932	.01	2.48	.001	.0002
dc126	126	.01	3.78	.001	.0017
ftv170	171	1.47	3.10	.114	–
ftv150	151	.77	3.17	.115	–
ftv160	161	1.36	3.29	.114	–
ftv130	131	.89	3.61	.112	–
ftv140	141	.83	4.12	.118	–
ftv110	111	1.79	4.18	.124	–
ftv120	121	1.68	4.93	.122	–
ftv100	101	1.16	5.52	.123	–
code198	198	.00	.09	1.000	.0224
code253	253	.74	17.50	1.000	.0598
atex600	600	1.13	97.69	.118	–

Table 1.3. Metrics for `realworld` test instances.

3. Heuristics

In this section we briefly describe the heuristics covered in this study and their implementations. Substantially fewer implementations are covered than in the STSP chapter. This is in part because we did not announce a formal challenge covering the ATSP, but mainly because there has been far less research on this more general problem. Thus a smaller sample of implementations can still include top-performing representatives of all the effective approaches. As was the case with the STSP, these effective approaches do not currently include any representatives from the world of metaheuristics (e.g., simulated annealing, tabu search, neural nets, etc.). The best ATSP heuristic from that field that we know of is a "memetic" algorithm of [5], but it is not yet fast enough to be competitive, given the quality of tours that it produces for most of our instance classes. We cover four basic groups of heuristics: Classical tour construction heuristics derived from those for the STSP, heuristics based on solutions to the Assignment Problem (these have no effective analogue in the case of the STSP), local search heuristics and repeated local search heuristics. We unfortunately do not have space to present all the algorithmic and implementation details for the heuristics (especially the more complicated local search and repeated local search heuristics), but we provide pointers to where more details can be found.

3.1. Classical Tour Construction Heuristics

Nearest Neighbor (`NN`) and Greedy (`Greedy`). These are ATSP versions of the classical Nearest Neighbor and Greedy heuristics for the STSP. In `NN` one starts from a random city and then successively goes to the nearest as-yet-unvisited city, returning at last to the starting city to complete the tour. In `Greedy`, we view the instance as a complete directed graph with arc lengths equal to the corresponding inter-city distances. Sort the arcs in order of increasing length. Call an arc *eligible* if it can be added to the current set of chosen arcs without creating a (non-Hamiltonian) cycle or causing an in- or out-degree to exceed one. The implementation covered here works by repeatedly choosing (randomly) one of the two shortest eligible arcs until a tour is constructed. Randomization is important as the implementations we cover are both part of a Johnson-McGeoch local search code to be described below, and a common way of using the latter is to perform a set of randomized runs and output the best tour found. The running times reported for these `NN` and `Greedy` implementations may be somewhat inflated from what they would be for stand-alone codes, as they include some preprocessing steps not strictly needed for tour construction alone.

3.2. Cycle Cover Heuristics

Recall from Section 2.1 that a solution to the Assignment Problem for the distance matrix of an ATSP instance corresponds to a minimum length cover of the cities by vertex-disjoint directed simple cycles. This section covers heuristics that solve such Assignment Problems as a subroutine. Analogous heuristics for the STSP are not particularly effective, but one might hope for better results in the case of the ATSP, especially for those instances classes with a small gap between the AP bound and the optimal tour length.

Cycle Patching, Two Largest Cycle Variant (`Patch`). This heuristic computes the minimum cycle cover for the full instance and then patches the cycles together into a tour using a heuristic analyzed by Karp and Steele in [21]: Repeatedly select the two cycles containing the most cities and combine them into the shortest overall cycle that can be constructed by breaking one arc in each cycle and linking together the two resulting directed paths. The running time for this heuristic is typically dominated by the time needed to construct the initial cycle cover. `Patch` provides better results than related variants, such as repeatedly patching together the two *shortest* cycles. The implementation for which we report results is due to Zhang.

Repeated Assignment (`RA`). This heuristic was originally studied by Frieze, Galbiati, and Maffioli in [11] 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 $\log_2 N$ 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 heuristic 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 constructed more than $\log_2 N$ cycle covers, and each cycle cover can be no longer than the optimal ATSP tour which is itself a cycle cover. Thus the total arc length for the connected graph is at most $\log_2 N$ 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 arc length of the graph. We report on a Johnson-McGeoch implementation that in addition uses heuristics to find good choices of representatives and performs "greedy" shortcutting as described in relation to the Christofides heuristic in Chapter 9. The combination of these measures yields substantial improvements in tour length with a negligible increase in running time, as reported in [10] (which called this enhanced version of the heuristic "`RA+`").

Contract or Patch (`COP`). This heuristic, as proposed by Glover, Gutin, Yeo, and Zverovich in [13] and refined by Gutin and Zverovich in [14], again begins by constructing a minimum length cycle cover. We then execute the following loop:

1. While the current cycle cover contains more than a single cycle and at least one cycle with fewer than t cities (*short* cycle):
   (1.1) Delete the longest arc in each short cycle and "contract" the resulting path into a single composite city whose out-arcs corresponds to the out-arcs of the path's head and whose in-arcs correspond to the in-arcs of the path's tail.
   (1.2) Compute a minimum length cycle cover from the resulting contracted graph.
2. Recursively expand the composite cities in the final cycle cover to obtain a cycle cover for the original instance.
3. Patch the cycle cover as in `Patch`.

The dependence of the performance of `COP` on the value of t was studied by Gutin and Zverovich in [14], who recommended setting $t = 3$, in which case only cycles of length two are contracted. The results presented here were obtained using their implementation with this choice of t.

Zhang's Heuristic (`Zhang`). Zhang's heuristic [31] works by truncating the computations of an AP-based branch-and-bound optimization algorithm that uses depth first search as its exploration strategy. We start 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 than 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* arcs, and X_0, the set of excluded arcs, be initially empty. The variant we cover in this chapter proceeds as follows.

Inductively, the incumbent cycle cover M_i is the minimum length cycle cover that contains all arcs from I_i and none of the arcs 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 in M_i that contains a minimum number of *free arcs* (arcs not in I_i). As pointed out in [9], there are k distinct ways of breaking this cycle: We 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 assignment problem for each of these possibilities that is not forbidden by the requirement that all arcs in I_i be included. In particular, for all h, $1 \leq h \leq k$, such that e_h is not in I_i, we construct a minimum cycle cover that includes all the arcs in $I_i \cup \{e_j : 1 \leq j < h\}$ and includes none of the arcs in $X_i \cup \{e_h\}$. (The exclusion of the arcs in this latter set is forced by adjusting their lengths to a value exceeding the initial champion tour length.) If we retain 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 other 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, we let the new incumbent M_{i+1} be a viable child of minimum length. We then patch M_{i+1} and if the result is better than the current champion tour, update the latter. Then we update I_i and X_i to reflect the sets of included and excluded arcs 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 arc 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 current champion tour.

The results reported here were obtained using Zhang's implementation. In [10], which called the above heuristic "`ZHANG1`," several variants were also considered. The most interesting of these, `ZHANG2`, differs from the above in that in each phase *all* viable children are patched to tours to see if a new champion can be produced. As reported there, this variant produces marginally better tours than does `ZHANG1`, but at a typical cost of roughly doubling the running time.

3.3. Local Search

Local search heuristics are typically defined in terms of a *neighborhood structure*, where tour B is a neighbor of tour A if it can be obtained from A by a specific type of perturbation or *move*. The standard local search heuristic uses a tour construction heuristic to generate an initial tour and then repeatedly looks for an improving move and, if it finds one, performs it to obtain a new and better tour. This process continues until no such move exists (or none can be found by the particular search process employed by the heuristic). We cover two local search heuristics.

3-Opt (`3opt`). In this heuristic, the neighborhood consists of all tours that can be obtained by deleting three arcs and permuting the three resulting paths. In contrast to the STSP version of `3opt`, we do not consider moves in which any of the three paths is reversed. Reversing a path potentially changes the lengths of all the arcs in the path, and hence is much more expensive to evaluate than in the symmetric case. We present results for a Johnson-McGeoch implementation of 3-Opt that generates its starting tours using `NN` and employs many of the speedup tricks exploited by their implementation of symmetric `3opt`. For details on these, see Chapter 9 and [18].

Kanellakis-Papadimitriou (`KP`). This heuristic, invented by Kanellakis and Papadimitriou in [20], attempts to mimic the Lin-Kernighan heuristic for the STSP [24], subject to the constraint that it does not reverse any tour segments. (The symmetric version of Lin-Kernighan is discussed in more detail in Chapters 8 and 9.) `KP` consists of two alternating search processes. The first process is a variable-depth search that tries to find an improving k-opt move (breaking k arcs and permuting the resulting k segments) for some odd $k \geq 3$ by a constrained sequential search procedure modeled on that in Lin-Kernighan. The second process is a search for an improving, non-sequential "double-bridge" 4-Opt move. In such a move, if (a, b, c, d) is the original ordering of the four tour segments formed by deleting four edges, the new tour contains them in the order (b, a, d, c), without reversals. The implementation on which we report here is that of Johnson and McGeoch described in [10] and called `KP4` there. The details are too complicated to go into here. Suffice it to say that many of the same speedup tricks as used in the Johnson-McGeoch implementation of Lin-Kernighan (see Chapter 9) are incorporated, and in the 4-Opt phase the *best* 4-Opt move is found using a dynamic programming approach suggested by [12] that takes only $\Theta(N^2)$ time.

3.4. Repeated Local Search Heuristics

Each of the heuristics in the previous section can be used as the engine for an "iterated" (or "chained") local search procedure that obtains significantly better tours, as originally proposed by [25]. (Other ways of improving on basic local search heuristics, such as the dynamic programming approach of [3], do not seem to be as effective here, although hybrids of this approach with iteration might be worth further study.) In an iterated procedure, one starts by running the basic heuristic once to obtain an initial champion tour. Then one repeats the following process some predetermined number of times:

> Apply a random double-bridge 4-Opt move to the current champion to obtain a new (and presumably worse) tour. Use this new tour as the starting tour for another run of the heuristic. If the resulting tour is better than the current champion, declare it to be the new champion.

If one is using the "don't-look bit" speedup trick of Bentley [4] in the basic local search heuristic, the fact that a double-bridge move changes the neighbors of only 8 cities can be exploited to obtain further speedups, as described in Chapter 9. All the iterated local search heuristics on which we report here use unbiased random double-bridge moves to perturb the tour, although better results might be obtainable if one biases the choice toward better moves, as is done for example in [1] for the STSP. In this chapter we present results for three iterated local search heuristics plus one additional heuristic that uses a different repetition mechanism.

Iterated 3-Opt (`I3opt`). A Johnson-McGeoch implementation that performs $10N$ iterations, where N is the number of cities.

Iterated Kanellakis-Papadimitriou (`IKP`). A Johnson-McGeoch implementation that performs N iterations. The basic local search heuristic used is a variant on `KP` that searches more broadly in the variable-depth search phase. This was called `IKP4F` in [10].

Iterated HyperOpt (`Ihyper-`k). This heuristic uses for its basic local search heuristic an asymmetric variant on the HyperOpt heuristic for the STSP introduced by Burke, Cowling, and Keuthen in [6] and described in Chapter 9. The HyperOpt neighborhood with parameter k is a restricted version of $2k$-Opt. In the variant used here and described more fully in [7], this neighborhood is augmented with (non-reversing) 3-Opt moves. Although that paper talks about using a "Variable Neighborhood Search" version of the above iteration process, the results we report are for an implementation from the authors that uses the standard double-bridge perturbation and performs N iterations. There is one added detail however: whenever a new champion tour is found in an iteration, the implementation attempts to improve it using 2-Opt moves. A 2-Opt move involves the reversal of a tour segment, so as remarked above this is expensive, but it does here yield a slight improvement in tour length for the near-symmetric instances. We report on the results for `Ihyper-3`. The tours found by `Ihyper-4` are slightly better on average, but the running times are 2–10 times longer.

Helsgaun's Heuristic (`Helsgaun`). This heuristic successfully applies the same sort of trick we used to compute Held-Karp bounds and optimal tour lengths for ATSP instances using `Concorde`: the ATSP instance is transformed into an equivalent STSP instances and then an STSP code is applied, in this case the repeated search version of Helsgaun's heuristic [17] as described in Chapter 9. The transformation used is somewhat simpler than the one used for `Concorde`. Here we replace each city c_i by a pair of cities c_i^+ and c_i^-, and set $d(c_i^+, c_j^-) = d(c_i, c_j)$ and $d(c_j^+, c_i^-) = d(c_j, c_i)$. In addition we set $d(c_i^-, c_i^+)$ to $-\infty$ and all the remaining distances to $+\infty$, where ∞ is a sufficiently large number that all arcs with length $-\infty$ *must* be included in the tour and none of the arcs with length $+\infty$ can be used. (This transformation doesn't work for `Concorde` because `Concorde` has trouble with large negative arc lengths and currently has no other facility for forcing arcs into the tour.)

The results we report are for version `LKH-1.1` of Helsgaun's code, with N iterations, called `Helsgaun[N]` in Chapter 9. (A significant number of iterations is needed just to get all the required arcs into the tour, so one *must* use the iterated version of `Helsgaun` heuristic.) In Helsgaun's approach, the starting tours for iterations after the first are not generated by double-bridge perturbations. Instead we run a full tour construction heuristic that biases its choices based on the arcs in the best tours seen so far. Note that since the ATSP to STSP transformation constructs an instance with $2N$ cities, the actual number of iterations is $2N$.

4. Results

In this section we summarize the results for the heuristics and instance classes describe above. We begin in Section 4.1 by using the two symmetric instance classes (`rect` and `smat`) to illuminate the question of how much more powerful symmetric codes can be than asymmetric ones. This is further examined in Section 4.2, which displays the best tour quality obtainable by any of our heuristics within given time bounds and compares this with what is possible in the symmetric case.

We then consider the relative advantages of the various heuristics. Section 4.3 looks at those heuristics that appear to be consistently fast (`NN`, `Greedy`, and `3opt`) and compares their performance. Section 4.4 considers the four heuristics based on the computation of minimum cycle covers. Section 4.5 then considers the remaining local and repeated local search heuristics `KP`, `Ihyper-3`, `I3opt`, `IKP`, and `Helsgaun`.

The final three "results" sections provide more detailed comparisons between heuristics. Section 4.6 gives a class-by-class comparison, presenting tables for each of the asymmetric classes ranking all the heuristics in the study. Section 4.7 compares the heuristics according to various robustness metrics. Section 4.8 then discusses the lessons learned from the experiments on our random instance classes and sees how well they apply to our testbed of real world instances.

4.1. Symmetric Instances and Codes

In this section we illustrate our claim that ATSP heuristics are in a sense less powerful than STSP heuristics. We have already seen that they are more limited, for instance by the fact that moves involving the reversal of a tour segment are typically too expensive to evaluate. We see how this translates into performance by looking at pairs of related ATSP/STSP heuristics and comparing their results on our two classes of symmetric instances (`rect` and `smat`). See Table 1.4.

The only heuristic that appears to be as powerful in its asymmetric form as in its symmetric form is Nearest Neighbor (`NN`). This is not surprising, given that for any given starting city it will construct the same tour in either case. Differences in running time come from the differences in input representation. In contrast, our other tour construction heuristic, the Greedy heuristic, completely falls apart in the asymmetric case. This is because two paths that will eventually be merged by the symmetric Greedy heuristic may be constructed with inconsistent orientations by the asymmetric Greedy heuristic and hence be unmergeable.

The results for the remaining local search and repeated local search pairs all show an advantage in tour quality for the symmetric members of each pair, reflecting the richer neighborhoods that they search. Note that this is true in the `KP` versus Lin-Kernighan comparison, even though the `KP` implementations includes a double-bridge 4-Opt phase absent from the `LK` implementation. Note also that the ATSP variants on repeated local search pay a substantial running time penalty (far greater than the simple difference attributable to the larger size of the asymmetric instance representation).

4.2. Asymmetric State of the Art

In this section we further address the differences in the states of the art for the STSP and ATSP by examining what can be accomplished within given time thresholds, assuming we can choose the most appropriate heuristic for each instance class and number of cities. See Table 1.5, which for $N = 316$, 1,000, and 3,162 and appropriate time thresholds reports the best average excess over the HK bound obtainable in normalized time less than that threshold and the heuristic that obtains it. For space reasons, we do not list results for our 100-city instances. We note, however, that for all nine classes `Helsgaun`'s average running time on 100-city instances is 2 seconds or less and it never averages more than .1% above optimal. More detailed results on the tradeoffs for each class can be found in the tables of Section 4.6.

`rect`

Heuristic	Percent above HK				Time in Seconds			
	100	316	1000	3162	100	316	1000	3162
`NN-S`	26.39	27.35	26.07	26.68	.00	.01	.0	0
`NN-A`	26.39	27.51	26.11	26.55	.04	.26	1.9	22
`Greedy-S`	18.72	19.35	17.91	16.46	.01	.01	.0	0
`Greedy-A`	153.78	292.21	547.94	1005.68	.04	.27	1.9	22
`3opt-S`	2.81	2.86	2.88	3.25	.04	.12	.2	0
`3opt-A`	7.89	8.30	8.37	8.76	.22	1.82	5.9	22
`I3opt-S`	.86	.94	1.11	1.29	.41	1.34	4.1	18
`I3opt-A`	2.02	2.57	2.66	3.09	.44	2.19	11.4	124
`LK-S`	1.32	1.55	1.77	1.92	.05	.14	.3	0
`KP-A`	5.11	5.06	5.00	5.17	.04	.31	2.0	23
`I3opt-S`	.86	.94	1.11	1.29	.41	1.34	4.1	18
`I3opt-A`	2.02	2.57	2.66	3.09	.44	2.19	11.4	124
`Hgaun-S`	.68	.67	.69	.62	.47	4.67	55.0	887
`Hgaun-A`	.68	1.00	1.62	2.35	2.00	21.50	247.3	6244

`smat`

Heuristic	Percent above HK				Time in Seconds			
	100	316	1000	3162	100	316	1000	3162
`NN-S`	139.28	180.75	235.51	298.05	.02	.13	1.0	12
`NN-A`	139.28	181.18	233.10	307.11	.03	.26	1.9	21
`Greedy-S`	103.48	136.78	177.17	213.90	.02	.16	1.0	12
`Greedy-A`	653.13	1842.80	5396.52	12073.47	.03	.31	1.9	21
`3opt-S`	11.38	19.18	31.29	46.59	.11	1.03	3.1	12
`3opt-A`	32.36	44.31	61.69	85.58	.19	1.76	5.8	21
`I3opt-S`	1.21	3.18	6.34	10.83	.45	1.84	7.8	58
`I3opt-A`	8.43	15.57	25.55	39.84	.50	2.63	14.0	121
`LK-S`	1.72	2.22	3.43	4.68	.12	.97	4.2	12
`KP-A`	10.59	11.50	13.13	15.31	.04	.36	2.2	26
`Hgaun-S`	.10	.04	.01	.00	.27	1.94	13.8	302
`Hgaun-A`	.88	2.16	3.78	5.72	1.40	16.45	180.9	4756

Table 1.4. Tour quality and normalized times for corresponding symmetric (`-S`) and asymmetric (`-A`) codes. `Hgaun` is an abbreviation for `Helsgaun`. `LK` is the Johnson-McGeoch implementation of (symmetric) Lin-Kernighan covered in Chapter 9.

Note that almost all our heuristics are represented in the table for some combination of class and N. Only `Greedy`, `RA`, and `Ihyper-3` are missing, and the latter would have made it into the table for `rtilt` and $N \geq 1,000$ had we chosen slightly different thresholds. Thus depending on the application and instance size, almost any of the heuristics we cover might be useful.

The tradeoffs between running time and tour quality are not as good as in the case of the STSP, however. We don't have the space here for a detailed comparison, but let us simply consider the behavior of the STSP heuristic `Helsgaun[.1N]` as discussed in Chapter 9. For the 1,000-city instances of all four classes discussed in that chapter, `Helsgaun[.1N]` averages within 1.05% of the HK bound in 12 seconds or less (normalized to the same target machine used in this chapter). Here that level of behavior can't be attained in 2 minutes for four of the classes and 10 minutes for three. For the 3,162-city instances, `Helsgaun[.1N]` averages within .87% of the HK bound in less than 2 minutes for each of the four STSP classes, whereas here that level of behavior cannot be attained in 3 hours for four of our nine classes. It would thus seem that at least some potential ATSP applications may be more difficult to handle than the current applications of the STSP. We will return to this question when we consider "real-world" instances in Section 4.8.

4.3. Consistently Fast Heuristics

In this section we consider our options if we need to obtain tours very quickly, i.e., within a small multiple of the time to read the instance. In the case of geometric instances of the STSP, this restriction was quite severe and only allowed us to use heuristics that did little more than sort. For the general ATSP, with its $\Theta(N^2)$ instance size, much more sophisticated heuristics satisfy this restriction, at least asymptotically: not only tour construction heuristics like `NN` and `Greedy`, but even simple local optimization heuristics like `3opt`. Table 1.6 presents the results for these three heuristics on our nine truly asymmetric instance classes, the classes ordered by increasing values of the gap between AP and HK bounds. In comparison, the time simply to read and store the distance matrix using standard `C` input/output routines is .02–.03 (normalized) seconds for $N = 100$, .14–.18 seconds for $N = 316$, .9–1.2 seconds for $N = 1,000$, and 10–14 seconds for $N = 3,162$, depending on the class. Observe that `NN` and `Greedy` typically use at most 2.5 times the read time, and although `3opt` has higher multiples for small N, it appears to be settling asymptotically to a similar multiple. Also, for none of the three heuristics does running time vary markedly from class to class.

316 Cities

Class	< .33 Seconds		< 2 Seconds		< 10 seconds		< 30 seconds	
tmat	.01	Zhang	.01	Zhang	.01	Zhang	.00	Helsgaun
amat	3.15	COP	.16	Zhang	.08	Helsgaun	.08	Helsgaun
shop	14.65	NN	.08	Zhang	.08	Zhang	.02	Helsgaun
disk	1.13	COP	.27	Zhang	.06	Helsgaun	.06	Helsgaun
super	1.20	COP	.15	IKP	.04	Helsgaun	.04	Helsgaun
crane	4.45	KP	4.29	Zhang	1.79	IKP	1.30	Helsgaun
coin	6.59	KP	6.59	KP	2.99	IKP	1.47	Helsgaun
stilt	22.79	Patch	8.79	KP	3.95	I3opt	2.33	Helsgaun
rtilt	18.91	Patch	18.91	Patch	7.35	KP	2.76	Helsgaun

1000 Cities

Class	< 5 Seconds		< 30 Seconds		< 2 Minutes		< 10 Minutes	
tmat	.00	Zhang	.00	Zhang	.00	Zhang	.00	Zhang
amat	2.66	COP	1.29	IKP	.03	Helsgaun	.03	Helsgaun
shop	13.29	NN	.03	Helsgaun	.03	Helsgaun	.01	Helsgaun
disk	.88	Patch	.02	Zhang	.01	Helsgaun	.01	Helsgaun
super	1.22	COP	.21	Zhang	.05	Helsgaun	.05	Helsgaun
crane	4.78	KP	1.27	IKP	1.27	IKP	1.02	Helsgaun
coin	6.15	KP	2.66	IKP	2.66	IKP	1.61	Helsgaun
stilt	8.80	KP	4.31	I3opt	4.31	I3opt	2.61	Helsgaun
rtilt	18.38	Patch	8.33	KP	8.33	KP	1.50	Helsgaun

3,162 Cities

Class	< 1 minute		< 5 minutes		< 30 Minutes		< 3 hours	
tmat	.00	Zhang	.00	Zhang	.00	Zhang	.00	Zhang
amat	1.01	COP	.04	Zhang	.04	Zhang	.03	Helsgaun
shop	10.88	3opt	.24	Patch	.01	Zhang	.01	Zhang
disk	25.64	3opt	.01	Zhang	.01	Zhang	.01	Helsgaun
super	1.88	I3opt	.52	IKP	.43	Zhang	.13	Helsgaun
crane	4.26	KP	1.36	IKP	1.36	IKP	.92	Helsgaun
coin	6.34	KP	2.87	IKP	2.87	IKP	2.16	Helsgaun
stilt	8.15	KP	4.28	I3opt	4.28	I3opt	3.39	Helsgaun
rtilt	19.83	3opt	9.68	KP	9.68	KP	2.76	Helsgaun

Table 1.5. Best average percentage above the HK bound obtained within various normalized running time thresholds for the 316-, 1,000-, and 3162-city instances of the nine asymmetric classes.

As for tour quality, observe first that the `Greedy` heuristic continues to perform poorly, being significantly worse than `NN` on all but three classes (`tmat` and `super`, where it is better, and `crane`, where it is roughly equivalent). Moreover, on four classes (`amat`, `disk`, `stilt`, and `rtilt`) `Greedy` might be said to self-destruct, as the ratio of its tour length to the HK bound grows rapidly as N increases, whereas `NN` only self-destructs on two of these and actually improves as N gets larger on the other two (`stilt` and `rtilt`). Consequently, although symmetric `Greedy` typically outperforms symmetric `NN`, their roles are reversed in the asymmetric case. This is why the Johnson-McGeoch local search implementations for the ATSP use `NN` to generate their starting tours rather than `Greedy`, which was the default for their STSP implementations. Although for three of the asymmetric instances classes (`tmat`, `super`, and `crane` again) `Greedy` manages to provide more effective (although not necessarily shorter) starting tours than `NN`, it is not sufficiently robust to be used as a general-purpose starting tour generator.

Consider now `3opt`. Note that the ratio of its overall running time to that for its tour generator `NN` declines from a factor of 6 or more when $N \leq 316$ to little more than a factor of 1 when $N = 3,162$. On the other hand, the amount by which its average tour length improves over its `NN` starting tour also declines as N increases for all but two of the classes, those being the two classes where `NN` self-destructs (`amat` and `disk`). The improvements over `NN` do however remain significant, even for $N = 3,162$, although the amount of significance varies with class. `3opt` would thus be the heuristic of choice among these three unless high speed on small instances is a requirement. Indeed, as we shall see, it might well be a best overall choice in many situations, given that it averages within 27% of the HK bound in less than 30 seconds for all classes except the totally unstructured instances of `amat`.

Let us now consider the extent to which these results correlate with the abstract instance metrics in Tables 1.2 and 1.3. A first observation is that for none of the three heuristics is there much of a correlation with the gap between AP and HK bounds, which is less than 2% for the first five classes in the table and greater than 13% for the last three. Such correlations will not be evident until we consider the heuristics of the next section. On the other hand, the two classes where both `NN` and `Greedy` self-destruct are those with the biggest triangle inequality violations (`amat` and `disk`), and `Greedy` self-destructs on all but one of the four classes with the highest asymmetry metrics (`amat`, `shop`, `stilt`, and `rtilt`). Whether such possible correlations are meaningful can only be determined once we have a more extensive set of instance classes to study.

Greedy

	Percent above HK				Time in Seconds			
Class	100	316	1000	3162	100	316	1000	3162
`tmat`	31.23	29.04	26.53	26.25	.03	.26	1.7	20
`amat`	243.09	362.86	418.56	695.29	.04	.27	1.9	21
`shop`	49.34	56.07	61.55	66.29	.03	.26	2.1	40
`disk`	188.82	307.14	625.76	1171.62	.03	.28	2.7	23
`super`	6.03	5.40	5.16	5.79	.03	.22	1.5	18
`crane`	41.86	44.09	39.70	41.60	.03	.27	1.9	21
`coin`	48.73	46.76	42.33	35.94	.04	.24	1.7	20
`stilt`	106.25	143.89	178.34	215.84	.04	.28	1.9	23
`rtilt`	350.12	705.56	1290.63	2350.38	.03	.28	2.0	23

NN

	Percent above HK				Time in Seconds			
Class	100	316	1000	3162	100	316	1000	3162
`tmat`	38.20	37.10	37.55	36.66	.03	.24	1.7	20
`amat`	195.23	253.97	318.79	384.90	.03	.26	1.9	21
`shop`	16.97	14.65	13.29	11.87	.03	.23	2.5	20
`disk`	96.24	102.54	115.51	161.99	.04	.27	1.9	23
`super`	8.57	8.98	9.75	10.62	.03	.21	1.5	18
`crane`	40.72	41.66	43.88	43.18	.03	.26	1.9	21
`coin`	26.08	26.71	26.80	25.60	.03	.23	1.7	20
`stilt`	30.31	30.56	27.62	24.79	.03	.30	1.9	22
`rtilt`	28.47	28.28	27.52	24.60	.04	.26	1.9	22

3opt

	Percent above HK				Time in Seconds			
Class	100	316	1000	3162	100	316	1000	3162
`tmat`	6.44	9.59	12.66	16.20	.19	1.71	5.5	20
`amat`	39.23	58.57	83.77	112.08	.19	1.75	5.8	21
`shop`	3.02	7.25	10.22	10.88	.23	1.78	5.6	21
`disk`	12.11	16.96	20.85	25.64	.19	1.82	6.1	23
`super`	3.12	4.30	5.90	7.94	.15	1.43	4.8	18
`crane`	9.48	9.41	10.65	10.64	.19	1.76	7.3	22
`coin`	8.06	9.39	9.86	9.92	.18	1.62	5.3	20
`stilt`	11.39	12.65	12.62	12.27	.19	1.80	8.2	22
`rtilt`	10.04	13.09	18.00	19.83	.19	2.05	6.6	23

Table 1.6. Tour quality and normalized running times for fast heuristics.

4.4. Cycle Cover Heuristics

Table 1.7 presents the results for three of the four heuristics of Section 3.2: the simple Cycle Patching heuristic (`Patch`), the Contract-or-Patch heuristic (`COP`), and Zhang's heuristic (`Zhang`). We omit the Repeated Assignment heuristic (`RA`) as for every class it finds worse tours and takes more time than `Patch`, despite the fact that it is the only polynomial-time heuristic known to have a reasonable worst-case guarantee for instances obeying the triangle inequality. (Its domination by `Patch` is shown in the class-based summary tables of Section 4.6.)

Note that for all three heuristics in the table, there is a strong correlation between good average tour length and the gap between the HK and AP bounds. (Once again, the classes in the table are ordered by increasing gap.) The only exceptions are the `amat` class, whose lack of structure appears to cause trouble for `Patch` and `COP` on the smaller instances, and the $N = 100$ entry for `disk`. Note, however, that for $N = 100$ `disk` has a bigger average gap than `super`. All three heuristics find better tours than does `3opt` on the first five classes and the larger instances of the sixth, while `Patch` and `COP` are worse than `3opt` on the last three classes. `Zhang`, although better than `3opt` on one of the last three classes (`rtilt`), has a running time that is over 400 times longer.

The running times for all three cycle cover heuristics are highly dependent on instance class. For `PATCH` and to a lesser extent `COP`, this primarily reflects the fact that the Assignment Problem code used as a subroutine has substantially larger running times for some instance classes than others. See Table 1.8, which gives the normalized running times for the Assignment Problem codes used by `COP` and by `Patch` (and by `Zhang`). Note first that the times for the latter code represents a major proportion of the times reported for `Patch` in Table 1.7.

The speeds of both AP codes vary substantially with instance class, and although the first code tends to be faster, the two are slowest on the same instance classes: `shop`, `disk`, and `rtilt`, with the times for the 3162-city instances of `shop` being from 18 to 87 times slower than those for `coin`, depending on the subroutine used. As to running time growth rates, note that if the rate for a class were merely quadratic, then the average times would go up roughly by a factor of 10 from one instance size to the next, while if the rate were cubic (the worst-case bound) it would go up by factors of about 32. The times on the `shop` and `disk` classes for both codes are thus consistent with a cubic growth rate, whereas the times for `amat`, `crane`, `coin`, and `stilt` are much closer to quadratic, at least in the case of the AP code used by `Patch`.

Patch

	Percent above HK				Time in Seconds			
Class	100	316	1000	3162	100	316	1000	3162
`tmat`	.84	.64	.17	.00	.03	.22	1.8	29
`amat`	10.95	6.50	2.66	1.88	.03	.22	1.9	18
`shop`	1.15	.59	.39	.24	.04	.48	8.4	260
`disk`	9.40	2.35	.88	.30	.03	.26	2.9	75
`super`	1.86	2.84	3.99	6.22	.02	.19	1.7	29
`crane`	9.40	10.18	9.45	8.24	.03	.21	1.5	23
`coin`	16.48	16.97	17.45	18.20	.02	.18	1.4	17
`stilt`	23.33	22.79	23.18	24.41	.03	.24	2.2	29
`rtilt`	17.03	18.91	18.38	19.39	.03	.28	2.9	54

COP

	Percent above HK				Time in Seconds			
Class	100	316	1000	3162	100	316	1000	3162
`tmat`	.57	.36	.16	.00	.01	.12	.7	15
`amat`	9.31	3.15	2.66	1.01	.01	.15	.6	26
`shop`	.68	.36	.19	.10	.08	1.41	29.1	1152
`disk`	6.00	1.13	.51	.15	.03	.31	8.7	297
`super`	1.01	1.20	1.22	2.06	.03	.24	4.6	243
`crane`	10.32	9.08	7.28	6.21	.04	.44	3.5	53
`coin`	16.44	17.68	16.23	16.06	.02	.10	1.2	22
`stilt`	22.48	23.31	22.80	22.90	.07	.94	8.1	105
`rtilt`	19.62	22.86	20.95	20.37	.05	.33	5.6	117

Zhang

	Percent above HK				Time in Seconds			
Class	100	316	1000	3162	100	316	1000	3162
`tmat`	.06	.01	.00	.00	.03	.27	2.5	30
`amat`	.97	.16	.04	.04	.04	.47	7.6	296
`shop`	.20	.08	.03	.01	.06	1.02	19.6	460
`disk`	1.51	.27	.02	.01	.05	.56	6.4	105
`super`	.27	.17	.21	.43	.04	.61	20.4	995
`crane`	4.36	4.29	4.05	4.10	.07	1.96	66.7	3176
`coin`	8.20	11.03	11.14	11.42	.10	3.82	168.4	9610
`stilt`	10.75	13.99	12.66	12.86	.11	4.11	163.7	4184
`rtilt`	9.82	12.20	11.81	11.45	.13	4.37	178.0	9594

Table 1.7. Tour quality and normalized running times for Cycle Cover Heuristics.

	COP AP Code				Patch and Zhang AP Code			
	Time in Seconds				Time in Seconds			
Class	100	316	1000	3162	100	316	1000	3162
tmat	.00	.05	.6	12	03	.22	1.8	26
amat	.01	.06	.5	9	.03	.22	1.8	17
shop	.02	.37	8.1	262	.03	.47	8.3	251
disk	.00	.08	1.7	63	.03	.26	2.8	72
super	.00	.03	.5	9	.02	.18	1.7	26
crane	.02	.09	.8	12	.03	.21	1.5	19
coin	.00	.02	.1	3	.02	.17	1.2	14
stilt	.01	.26	2.0	27	.03	.23	2.0	26
rtilt	.01	.15	2.1	43	.03	.27	2.7	49

Table 1.8. Normalized running times for reading an instance and solving the corresponding Assignment Problem with the AP codes used by COP, Patch, and Zhang.

These growth rates should be compared to those for 3opt in Table 1.6, which seems to be subquadratic, at least within this range. (Asymptotically it must be at least quadratic, because it needs to read the instance, but within this range reading time is not the dominant running time component.) Thus, although Patch is faster than 3opt for all classes when $N \leq 316$, it begins losing ground thereafter. It is significantly slower on shop when $N = 1,000$, and on five additional classes when $N = 3,162$.

When we turn to COP and Zhang, additional factors affect running times. Most importantly, these heuristics may call the AP code more than once, and the number of calls also varies among instance classes, adding further to overall running time variability. For Patch the ratio between slowest and fastest class running times at 3,162 cities is about 15, whereas for COP it is 77 and for Zhang it is 320.

Because of its extra AP calls and despite its often-faster AP code, COP is typically slower than Patch. The only exceptions are those classes where the initial cycle cover typically has no 2-cycles, as appears to be the case for tmat. In such cases COP is just a version of PATCH with different tie-breaking rules. COP does, however, get better results than PATCH on all but one of the classes (rtilt), although there isn't much room for improvements on Patch's results for tmat, shop, and the larger instances of disk.

Zhang finds better tours than Patch and COP for all classes. (In the first case this is unsurprising, since the first step of Zhang is to perform Patch.) Where there is room for substantial improvements, Zhang's improvements are comparatively larger (although still not enough to beat 3opt on the coin and stilt instances). Zhang's better tour quality often comes at a significant running time cost, however. For example, Zhang is over 100 times slower than COP on the larger coin instances. Its running time growth rates look worse than quadratic for all classes except possibly tmat, and worse even than cubic for amat, super, crane, coin, and rtilt. Only two of these five (amat and super) have small HK-AP gaps, however, and in neither of these two cases is the time at $N = 3,162$ nearly as bad as for the last four classes. Moreover, for the shop and larger disk instances Zhang is actually faster than COP. Indeed, for the five classes with small HK-AP gaps and instances with 1,000 or fewer cities, Zhang gets its good results while never averaging more than 20.4 normalized seconds. Thus assuming one is not dealing with larger instances and one has small gaps, Zhang might well be the heuristic of choice. For instance classes that have larger HK-AP gaps, however, there are better alternatives, as we shall see in the next section.

4.5. Local and Repeated Local Search Heuristics

Table 1.9 presents the results for the Johnson-McGeoch implementation of the Kanellakis-Papadimitriou heuristic (KP) and iterated versions of two simpler local search heuristics: the Johnson-McGeoch implementation of Iterated 3-Opt (I3opt) and the Burke, Cowling, and Keuthen implementation of their Iterated HyperOpt heuristic (Ihyper-3). All three implementations perform 3-Opt as part of their search, so it is no surprise that they find better tours than does 3opt for all classes. For KP, this often comes with only a small running time penalty. For five of the classes (amat, super, crane, and stilt), its running time is within 50% or less of that for 3opt.

Comparisons to the cycle cover heuristics are more complicated, but reflect the correlation for the latter between good behavior and small HK-AP gaps. For the first four classes, KP is outperformed by the faster Patch heuristic and the comparably fast COP heuristic on all but the 100-city instances of amat and disk, where KP finds significantly better tours, although not ones as good as those found by Zhang. Interestingly, for all four classes KP's tours get progressively worse as N increases, while those of the cycle cover heuristics get progressively better. On the other hand, for the last three classes KP finds much better tours than Zhang in much less time, with the time advantage growing dramatically with N. For crane it finds tours that are almost as good as Zhang's with a time advantage that grows to a factor of over 100 for $N = 3,162$.

I3opt is slower than KP for all classes except disk. For most classes its running time is about 10 times slower for 100 cities, but the ratio declines as N increases. On the other hand, its tours for classes crane, coin, and stilt are the best we have seen so far, and its tours for

KP

	Percent above HK				Time in Seconds			
Heur	100	316	1000	3162	100	316	1000	3162
tmat	1.41	2.23	3.09	4.03	.11	.64	4.2	49
amat	5.82	6.95	8.99	11.51	.04	.36	2.3	27
shop	1.57	2.92	3.88	4.54	2.08	17.46	118.2	1005
disk	2.99	3.81	5.81	9.17	.05	.55	8.2	324
super	1.05	1.29	1.59	2.10	.04	.24	1.7	19
crane	4.58	4.45	4.78	4.26	.05	.32	2.0	22
coin	5.74	6.59	6.15	6.34	.04	.27	1.8	20
stilt	8.57	8.79	8.80	8.15	.09	.56	3.0	31
rtilt	6.06	7.35	8.33	9.68	.27	2.15	14.9	133

I3opt

	Percent above HK				Time in Seconds			
Heur	100	316	1000	3162	100	316	1000	3162
tmat	.30	.85	1.63	2.25	1.02	3.93	15.5	98
amat	5.10	13.27	27.12	45.53	.50	2.59	14.3	116
shop	.49	4.02	9.23	10.76	9.90	37.73	102.9	401
disk	.97	2.32	3.89	5.29	.49	3.14	24.1	293
super	.28	.61	1.06	1.88	.40	1.77	7.8	59
crane	1.98	2.27	1.95	2.12	.45	2.22	12.1	115
coin	2.98	3.37	3.48	3.83	.43	2.06	10.9	110
stilt	3.29	3.95	4.32	4.28	.92	5.21	29.1	256
rtilt	1.69	4.22	12.97	18.41	3.08	24.22	98.2	412

Ihyper-3

	Percent above HK				Time in Seconds			
Heur	100	316	1000	3162	100	316	1000	3162
tmat	1.29	2.71	4.96	7.29	2.82	15.45	62.7	1048
amat	15.29	34.74	66.66	103.10	1.93	15.78	81.9	529
shop	.46	1.85	5.05	9.36	10.07	75.38	300.3	1077
disk	2.02	6.53	10.52	14.77	2.02	21.95	166.5	1812
super	.62	1.53	2.81	4.69	1.35	9.09	47.6	741
crane	1.83	2.46	2.66	3.28	1.58	11.37	75.8	920
coin	2.22	3.09	3.83	4.60	1.44	9.99	62.6	675
stilt	3.20	4.32	5.67	6.45	2.38	18.35	91.0	869
rtilt	1.31	2.16	4.63	8.10	4.33	42.26	300.2	2531

Table 1.9. Tour quality and normalized running times for KP, I3opt, and Ihyper-3.

`super` have only been bested by `Zhang` which took more than 10 times as long. It does perform relatively poorly, however, on the four classes with smallest HK-AP gap and on `rtilt`.

Turning to `Ihyper-3`, we see substantially greater running times than for either `KP` or `I3opt`. For the four classes with smallest HK-AP gaps, it is also significantly slower than `Zhang` and produces worse results. For the remaining classes it is outperformed by `I3opt` except for the `rtilt` class and the 100-city `crane` and `stilt` instances, where it produces the best results seen so far.

Table 1.10 presents results for the two most sophisticated repeated local search codes in this study: The Johnson-McGeoch implementation of Iterated Kanellakis-Papadimitriou (`IKP`) and Helsgaun's `Helsgaun` variant on Lin-Kernighan.

Note that `IKP` finds better tours than does `I3opt` for all classes, but at much greater running time cost. It performs poorly in the same

`IKP`

	Percent above HK				Time in Seconds			
Heur	100	316	1000	3162	100	316	1000	3162
`tmat`	.09	.14	.41	.65	5.91	31.21	189.1	1934
`amat`	.56	.74	1.29	2.43	.34	3.19	29.6	488
`shop`	.49	2.36	3.66	4.55	214.48	1585.19	7449.1	31738
`disk`	.56	.48	.96	1.77	.43	12.57	479.4	26277
`super`	.13	.15	.28	.52	.17	1.33	10.2	138
`crane`	1.46	1.79	1.27	1.36	.44	3.52	23.5	297
`coin`	2.71	2.99	2.66	2.87	.35	2.35	16.9	231
`stilt`	3.00	3.54	3.96	4.13	4.78	89.67	1577.0	30916
`rtilt`	1.80	4.12	7.29	8.89	25.93	657.80	7420.4	61939

`Helsgaun`

	Percent above HK				Time in Seconds			
Heur	100	316	1000	3162	100	316	1000	3162
`tmat`	.03	.00	.00	.00	1.00	10.22	91.8	1634
`amat`	.29	.08	.03	.03	.80	7.89	75.6	1802
`shop`	.05	.02	.01	.02	.87	20.17	483.0	13671
`disk`	.24	.06	.01	.01	1.00	9.39	93.9	1914
`super`	.06	.04	.05	.13	1.07	7.67	77.3	1828
`crane`	1.21	1.30	1.02	.92	1.20	14.89	162.2	4040
`coin`	1.15	1.47	1.61	2.16	1.40	17.45	205.4	4975
`stilt`	1.95	2.33	2.61	3.39	1.67	18.67	221.0	5863
`rtilt`	.69	.99	1.50	2.76	2.00	23.45	318.2	8574

Table 1.10. Tour quality and normalized running times for `IKP` and `Helsgaun`.

places, however. For the four classes with smallest HK-AP gap Zhang finds better tours in less time. For the intermediate class super, the two heuristics are roughly equal in tour quality while IKP takes twice as long (on the same machine). On the problematic rtilt class, IKP is marginally beaten by Ihyper-3, which takes much less time.

The final heuristic Helsgaun has the best average tour length for all classes and all N (except the 3162-city shop instance, where it was beaten by Zhang, which was 0.1% over HK as opposed to Helsgaun's 0.2%). Helsgaun also has substantial running times, although ones that are exceeded by those for Zhang on two classes and by IKP on four. This is clearly the general-purpose algorithm of choice when time is not an issue or when instances are small.

As a final point of comparison, Table 1.11 presents results for the "heuristic" that computes the optimal tour length using Concorde's STSP optimization code via our ATSP-to-STSP transformation, with Zhang or Helsgaun run to provide an initial upper bound. Note that for the four classes with smallest HK-AP gap, the running time for optimization, if exponential, is not yet showing strong evidence of that behavior, although the data suggests such behavior for coin. Indeed, optimization is actually faster than IKP for all sizes of classes tmat and shop and for the larger sizes of disk. It also beats Helsgaun on the larger tmat instances, and is even faster than computing the HK bound for these and the largest disk instance. Average times should not be trusted, however, especially for the last five classes, where running times are far more variable than were the times for our heuristics. The times

Optimization via Concorde

	Percent above HK				Normalized Time in Seconds			
Heur	100	316	1000	3162	100	316	1000	3162
tmat	.03	.00	.00	.00	4.14	11.50	38.0	568
amat	.29	.08	.02	.01	16.60	109.00	399.0	40317
shop	.05	.02	.01	.00	54.80	315.00	1039.0	20234
disk	.24	.06	.01	.01	15.20	40.00	160.0	1176
super	.05	.03	.01	–	4.30	33.60	1629.0	–
crane	1.21	1.30	–	–	117.00	7081.00	–	–
coin	1.05	1.36	–	–	193.00	83285.00	–	–
stilt	1.86	–	–	–	1119.00	–	–	–
rtilt	.68	.67	–	–	82.00	1734.00	–	–

Table 1.11. Results for optimization via Concorde. A "–" entry indicates that optimization was not feasible via this approach. Running times include the time for running Zhang (first four classes) or Helsgaun (last five) to obtain an initial upper bound, but not the (relatively small) time for the ATSP-to-STSP transformation.

for the 100-city `stilt` instances ranged from 5 seconds to 3500 seconds. The gap between the optimal tour length and the HK bound for a given class and value of N was relatively consistent, however. Note that the optimal solution value is quite close to the HK bound for classes with small HK-AP gap, but somewhat farther away when the gap is larger.

4.6. Results Organized by Instance Class

In the last three "Results" sections, we more directly address the question of which heuristics one should use in practice. Here we consider what one should do if one knows a great deal about the application in question, as we now do in the case of our nine random asymmetric instance classes. The choice of what to use of course depends on the relative importance one places on tour quality and running time. In Tables 1.12 through 1.16 we present for each of those nine classes the average tour quality and normalized running times for all the heuristics in this study. This expands on the more restricted class-based results of Section 4.2 by showing the full range of quality/time tradeoffs for each class. We include the previously-ignored results for the Greedy and Repeated Assignment heuristics and data on the HK-AP and Opt-HK gaps and the time for computing the associated bounds. The AP bounds were computed using the AP code of `Patch`. All codes except `COP` and `Ihyper-3` were run on the same 196 Mhz MIPS R10000 processors, and running time comparisons among all but those two are thus unaffected by normalization errors (although the actual values presented may be affected by their translation to the target machine).

In order to fit the tables to the width of the page, we use the abbreviations "`Hgaun`" for `Helsgaun` and "`Ihyper`" for `Ihyper-3`. The identities of the algorithms behind the names are all explained in Section 3. For each class we have sorted the heuristics by their average excess over the HK bound on 3,162-city instances. Note that the ordering would often have been significantly different had we sorted on the 100-city results. Beyond these remarks, we will let the results speak for themselves.

4.7. Robustness of the Heuristics

In this section we consider the question of which heuristics to choose for general purpose ATSP tour generation in cases where one may not yet know much about the instance structure. For such uses, one would want a heuristic that was relatively robust, producing good results in reasonable time for a wide variety of instance types, the meaning of "good" and "reasonable" of course depending on the situation. As a

tmat

	Percent above HK				Time in Seconds			
Heur	100	316	1000	3162	100	316	1000	3162
NN	38.20	37.10	37.55	36.66	.03	.24	1.7	20
Greedy	31.23	29.04	26.53	26.25	.03	.26	1.7	20
3opt	6.44	9.59	12.66	16.20	.19	1.71	5.5	20
Ihyper	1.29	2.71	4.96	7.29	2.82	15.45	62.7	1048
KP	1.41	2.23	3.09	4.03	.11	.64	4.2	49
I3opt	.30	.85	1.63	2.25	1.02	3.93	15.5	98
IKP	.09	.14	.41	.65	5.91	31.21	189.1	1934
RA	4.88	3.10	1.55	.46	.06	.63	7.8	278
Patch	.84	.64	.17	.00	.03	.22	1.8	29
COP	.57	.36	.16	.00	.01	.12	.7	15
Zhang	.06	.01	.00	.00	.03	.27	2.5	30
Hgaun	.03	.00	.00	.00	1.00	10.22	91.8	1634
OPT	.03	.00	.00	.00	4.14	11.50	38.0	568
HK	.00	.00	.00	.00	.77	4.46	43.8	968
AP	-.34	-.16	-.03	.00	.03	.22	1.8	26

amat

	Percent above HK				Time in Seconds			
Heur	100	316	1000	3162	100	316	1000	3162
Greedy	243.09	362.86	418.56	695.29	.04	.27	1.9	21
NN	195.23	253.97	318.79	384.90	.03	.26	1.9	21
RA	86.60	100.61	110.38	134.14	.07	.69	10.2	265
3opt	39.23	58.57	83.77	112.08	.19	1.75	5.8	21
Ihyper	15.29	34.74	66.66	103.10	1.93	15.78	81.9	529
I3opt	5.10	13.27	27.12	45.53	.50	2.59	14.3	116
KP	5.82	6.95	8.99	11.51	.04	.36	2.3	27
IKP	.56	.74	1.29	2.43	.34	3.19	29.6	488
Patch	10.95	6.50	2.66	1.88	.03	.22	1.9	18
COP	9.31	3.15	2.66	1.01	.01	.15	.6	26
Zhang	.97	.16	.04	.04	.04	.47	7.6	296
Hgaun	.29	.08	.03	.03	.80	7.89	75.6	1802
OPT	.29	.08	.02	.01	16.60	109.00	399.0	40317
HK	.00	.00	.00	.00	.70	4.33	36.7	703
AP	-.65	-.29	-.04	-.04	.03	.22	1.8	17

Table 1.12. Tour quality and normalized running times for classes `amat` (Random Asymmetric Matrices) and `tmat` (Random Asymmetric Matrices closed under shortest paths).

shop

	Percent above HK				Time in Seconds			
Heur	100	316	1000	3162	100	316	1000	3162
Greedy	49.34	56.07	61.55	66.29	.03	.26	2.1	40
NN	16.97	14.65	13.29	11.87	.03	.23	2.5	20
3opt	3.02	7.25	10.22	10.88	.23	1.78	5.6	21
I3opt	.49	4.02	9.23	10.76	9.90	37.73	102.9	401
Ihyper	.46	1.85	5.05	9.36	10.07	75.38	300.4	1077
IKP	.49	2.36	3.66	4.55	214.48	1585.19	7449.1	31738
KP	1.57	2.92	3.88	4.54	2.08	17.46	118.2	1005
RA	4.77	2.77	1.69	1.05	.10	1.68	28.7	1103
Patch	1.15	.59	.39	.24	.04	.48	8.4	260
COP	.68	.36	.19	.10	.08	1.41	29.1	1152
Hgaun	.05	.02	.01	.02	.87	20.17	483.0	13671
Zhang	.20	.08	.03	.01	.06	1.02	19.6	460
OPT	.05	.02	.01	.00	54.80	315.00	1039.0	20234
HK	.00	.00	.00	.00	1.53	10.12	104.3	1581
AP	-.50	-.22	-.15	-.07	.03	.47	8.3	251

disk

	Percent above HK				Time in Seconds			
Heur	100	316	1000	3162	100	316	1000	3162
Greedy	188.82	307.14	625.76	1171.62	.03	.28	2.7	23
NN	96.24	102.54	115.51	161.99	.04	.27	1.9	23
3opt	12.11	16.96	20.85	25.64	.19	1.82	6.1	23
RA	86.12	58.27	42.45	25.32	.07	.92	18.9	658
Ihyper	2.02	6.53	10.52	14.77	2.02	21.95	166.5	1812
KP	2.99	3.81	5.81	9.17	.05	.55	8.2	324
I3opt	.97	2.32	3.89	5.29	.49	3.14	24.1	293
IKP	.56	.48	.96	1.77	.43	12.57	479.4	26277
Patch	9.40	2.35	.88	.30	.03	.26	2.9	75
COP	6.00	1.13	.51	.15	.03	.31	8.7	297
Zhang	1.51	.27	.02	.01	.05	.56	6.4	105
Hgaun	.24	.06	.01	.01	1.00	9.39	93.9	1914
OPT	.24	.06	.01	.01	15.20	40.00	160.0	1176
HK	.00	.00	.00	.00	.85	4.84	53.7	1929
AP	-2.28	-.71	-.34	-.11	.03	.26	2.8	72

Table 1.13. Tour quality and normalized running times for classes `shop` (50 Processor No-Wait Flowshop Scheduling) and `disk` (Random Disk Head Motion Scheduling).

`super`

	Percent above HK				Time in Seconds			
Heur	100	316	1000	3162	100	316	1000	3162
`Greedy`	48.73	46.76	42.33	35.94	.04	.24	1.7	20
`NN`	8.57	8.98	9.75	10.62	.03	.21	1.5	18
`RA`	4.24	5.22	6.59	8.34	.01	.47	2.1	167
`3opt`	3.12	4.30	5.90	7.94	.15	1.43	4.8	18
`Patch`	1.86	2.84	3.99	6.22	.02	.19	1.7	29
`Ihyper`	.62	1.53	2.81	4.69	1.36	9.09	47.6	741
`KP`	1.05	1.29	1.59	2.10	.04	.24	1.7	19
`COP`	1.01	1.20	1.22	2.06	.03	.24	4.6	243
`I3opt`	.28	.61	1.06	1.88	.40	1.77	7.8	59
`IKP`	.13	.15	.28	.52	.17	1.33	10.2	138
`Zhang`	.27	.17	.21	.43	.04	.61	20.4	995
`Hgaun`	.06	.04	.05	.13	1.07	7.67	77.3	1828
`OPT`	.05	.03	.01	–	4.30	33.60	1629.0	–
`HK`	.00	.00	.00	.00	.68	4.52	40.4	900
`AP`	-1.04	-1.02	-1.17	-1.61	.02	.18	1.7	26

`crane`

	Percent above HK				Time in Seconds			
Heur	100	316	1000	3162	100	316	1000	3162
`RA`	40.80	50.33	53.60	54.91	.07	.67	7.0	251
`NN`	40.72	41.66	43.88	43.18	.03	.26	1.9	21
`Greedy`	41.86	44.09	39.70	41.60	.03	.27	1.9	21
`3opt`	9.48	9.41	10.65	10.64	.19	1.76	7.3	22
`Patch`	9.40	10.18	9.45	8.24	.03	.21	1.5	23
`COP`	10.32	9.08	7.28	6.21	.04	.44	3.5	53
`KP`	4.58	4.45	4.78	4.26	.05	.32	2.0	22
`Zhang`	4.36	4.29	4.05	4.10	.07	1.96	66.7	3176
`Ihyper`	1.83	2.46	2.66	3.28	1.58	11.37	75.8	920
`I3opt`	1.98	2.27	1.95	2.12	.45	2.22	12.1	115
`IKP`	1.46	1.79	1.27	1.36	.44	3.52	23.5	297
`Hgaun`	1.21	1.30	1.02	.92	1.20	14.89	162.2	4040
`OPT`	1.21	1.30	–	–	117.00	7081.00	–	–
`HK`	.00	.00	.00	.00	.95	7.11	357.4	669
`AP`	-7.19	-6.34	-5.21	-4.43	.03	.21	1.5	19

Table 1.14. Tour quality and normalized running times for classes `super` (Approximate Shortest Common Superstring) and `crane` (Stacker-Crane Scheduling).

coin

	Percent above HK				Time in Seconds			
Heur	100	316	1000	3162	100	316	1000	3162
RA	52.74	64.95	68.78	71.20	.06	.56	6.3	140
Greedy	48.73	46.76	42.33	35.94	.04	.24	1.7	20
NN	26.08	26.71	26.80	25.60	.03	.23	1.7	20
Patch	16.48	16.97	17.45	18.20	.02	.18	1.4	17
COP	16.44	17.68	16.23	16.06	.02	.10	1.2	22
Zhang	8.20	11.03	11.14	11.42	.10	3.82	168.4	9610
3opt	8.06	9.39	9.86	9.92	.18	1.62	5.3	20
KP	5.74	6.59	6.15	6.34	.04	.27	1.8	20
Ihyper	2.22	3.09	3.83	4.60	1.43	9.99	62.5	674
I3opt	2.98	3.37	3.48	3.83	.43	2.06	10.9	110
IKP	2.71	2.99	2.66	2.87	.35	2.35	16.9	231
Hgaun	1.15	1.47	1.61	2.16	1.40	17.45	205.4	4975
OPT	1.05	1.36	–	–	193.00	83285	–	–
HK	.00	.00	.00	.00	1.11	7.64	54.8	394
AP	-15.04	-13.60	-13.96	-13.09	.02	.17	1.2	14

stilt

	Percent above HK				Time in Seconds			
Heur	100	316	1000	3162	100	316	1000	3162
Greedy	106.25	143.89	178.34	215.84	.04	.28	1.9	23
RA	55.79	62.76	65.03	71.48	.07	.61	8.5	241
NN	30.31	30.56	27.62	24.42	.03	.30	1.9	48
Patch	23.33	22.79	23.18	24.41	.03	.24	2.2	29
COP	22.48	23.31	22.80	22.90	.07	.94	8.1	105
Zhang	10.75	13.99	12.66	12.86	.11	4.11	163.7	4184
3opt	11.39	12.65	12.62	12.27	.19	1.80	8.2	22
KP	8.57	8.79	8.80	8.15	.09	.56	3.0	31
Ihyper	3.20	4.32	5.67	6.45	2.38	18.35	91.0	870
I3opt	3.29	3.95	4.32	4.28	.92	5.21	29.1	256
IKP	3.00	3.54	3.96	4.13	4.78	89.67	1577.0	30916
Hgaun	1.95	2.33	2.61	3.39	1.67	18.67	221.0	5863
OPT	1.86	–	–	–	1119.00	–	–	–
HK	.00	.00	.00	.00	1.24	8.63	67.4	368
AP	-18.41	-14.98	-14.65	-14.04	.03	.23	2.0	26

Table 1.15. Tour quality and normalized running times for classes `coin` (Coinbox Collection Routing) and `stilt` (Tilted Supnorm Routing).

rtilt

	Percent above HK				Time in Seconds			
Heur	100	316	1000	3162	100	316	1000	3162
Greedy	350.12	705.56	1290.63	2350.38	.03	.28	2.0	23
RA	61.95	73.47	78.27	82.03	.08	1.06	19.3	607
NN	28.47	28.28	27.52	24.60	.04	.26	1.9	22
COP	19.62	22.86	20.95	20.37	.05	.33	5.6	117
3opt	10.04	13.09	18.00	19.83	.19	2.05	6.6	23
Patch	17.03	18.91	18.38	19.39	.03	.28	2.9	54
I3opt	1.69	4.22	12.97	18.41	3.08	24.22	98.2	412
Zhang	9.82	12.20	11.81	11.45	.13	4.37	178.0	9594
KP	6.06	7.35	8.33	9.68	.27	2.15	14.9	133
IKP	1.80	4.12	7.29	8.89	25.93	657.80	7420.4	61939
Ihyper	1.31	2.16	4.63	8.10	4.32	42.26	300.2	2531
Hgaun	.69	.99	1.50	2.76	2.00	23.45	318.2	8574
OPT	.68	.67	.68	–	82.00	1734.00	–	–
HK	.00	.00	.00	.00	1.37	8.88	87.5	373
AP	-20.42	-17.75	-17.17	-16.84	.03	.27	2.7	49

Table 1.16. Tour quality and normalized running times for class `rtilt` (Tilted Rectilinear Routing). Optima for 1000-city instances were computed by symmetric code applied to equivalent `rect`.

first attempt at providing such advice, we propose using the following empirical metric.

In Table 1.17 we summarize the worst average results (both excess over the HK bound and normalized running time) obtained over three different subsets of the classes, with heuristics ordered by their tour quality for 3,162-city instances. The first subset consists of all of the nine asymmetric instance classes except the Random Distance Matrix class `amat`. (We omit `amat` because such structureless instances are unlikely to arise in any real world application, and if one wants to study them for some mathematical reason, we already have provided detailed information about which heuristics to use in the previous section.) This is the robustness criterion that might apply if one knows nothing about the instance(s) for which tours are desired. The second and third subsets yield robustness criteria that might be relevant if we at least knew something about the HK-AP gaps for the instances in question. The second subset consists of the four classes (other than `amat`) for which the average HK-AP gap is less than 2% for all values of N we cover, and the third consists of the remaining four classes, for which the gaps range from 4% to over 20%.

The table omits the `Greedy` and `NN` tour construction heuristics because both are dominated with respect to the robustness metrics by

Worst Results over All Asymmetric Classes except amat

Heur	Percent over HK				Normalized Time in Seconds			
	100	316	1000	3162	100	316	1000	3162
3opt	12.11	16.96	20.85	25.64	.23	2.05	8.2	23
Patch	23.33	22.79	23.18	24.41	.04	.48	8.4	260
COP	22.48	23.31	22.80	22.90	.08	1.41	29.1	1152
I3opt	3.29	4.22	12.97	18.41	9.90	37.73	102.9	412
Ihyper-3	3.20	6.53	10.52	14.77	10.07	75.38	300.3	2531
Zhang	10.75	13.99	12.66	12.86	.13	4.37	178.0	9610
KP	8.57	8.79	8.80	9.68	2.08	17.46	118.2	1005
IKP	3.00	4.12	7.29	8.89	214.48	1585.19	7449.1	61939
Helsgaun	1.95	2.33	2.61	3.39	2.00	23.45	483.0	13671

Worst Results over {tmat, shop, disk, super}

3opt	12.11	16.96	20.85	25.64	.23	2.05	8.2	23
Ihyper-3	2.02	6.53	10.52	14.77	10.07	75.38	300.3	1812
I3opt	.97	4.02	9.23	10.76	9.90	37.73	102.9	401
KP	2.99	3.81	5.81	9.17	2.08	17.46	118.2	1005
Patch	9.40	2.84	3.99	6.22	.04	.48	8.4	260
IKP	.56	2.36	3.66	4.55	214.48	1585.19	7449.1	31738
COP	6.00	1.20	1.22	2.06	.08	1.41	29.1	1152
Zhang	1.51	.27	.21	.43	.06	1.02	20.4	995
Helsgaun	.24	.06	.05	.13	1.07	20.17	483.0	13671

Worst Results over {crane, coin, stilt, and rtilt}

Patch	23.33	22.79	23.18	24.41	.03	.28	2.9	54
COP	22.48	23.31	22.80	22.90	.07	.94	8.1	117
3opt	11.39	13.09	18.00	19.83	.19	2.05	8.2	23
I3opt	3.29	4.22	12.97	18.41	3.08	24.22	98.2	412
Zhang	10.75	13.99	12.66	12.86	.13	4.37	178.0	9610
KP	8.57	8.79	8.80	9.68	.27	2.15	14.9	133
IKP	3.00	4.12	7.29	8.89	25.93	657.80	7420.4	61939
Ihyper-3	3.20	4.32	5.67	8.10	4.33	42.26	300.2	2531
Helsgaun	1.95	2.33	2.61	3.39	2.00	23.45	318.2	8574

Table 1.17. Robustness metrics for the heuristics in this study.

3opt. We also omit the cycle cover heuristic RA, which is dominated for all instance classes by Patch.

A first remark is that many of the values in the table would improve if we could simply delete the classes shop and rtilt from the subsets, as these tend to cause the most trouble for many of the heuristics. However, as the results stand, certain heuristics do jump out as good performers. `Helsgaun` is of course the best tour generator if running time is not an issue or N is small. `Zhang` and `Patch` both might offer appealing trade-offs for instances with small HK-AP gaps when less time is available, as would `KP` for instances with large gaps. Two other contenders are `COP` for small-gap instances and `Ihyper-3` for large-gap instances, but the first is perhaps not sufficiently faster than `Zhang` to justify its poorer performance, and the second is not sufficiently faster than `Helsgaun`.

In the next section we shall see how well this advice serves us, by applying `Patch`, `Zhang`, `KP`, and `Helsgaun` to our suite of `realworld` instances.

4.8. Performance on Real-World Instances

In Table 1.18 we present the excesses over optimal and the normalized running times for the codes `Patch`, `KP`, `Zhang`, and `Helsgaun` on our testbed of real world instances. Within groups, the instances are ordered by increasing value of HK-AP gap, and groups are ordered by increasing *average* HK-AP gap. Note that here we are presenting the excess over the *optimal* tour length, not the HK bound, since the former has been successfully computed for all these instances (often in time less than that required by `Helsgaun`). All codes were run on the same machine so normalization errors do not affect running time comparisons. However, since instance sizes do not align precisely with the sizes for which our benchmark normalization runs were performed we settled for a single normalization factor of 0.5. This is a reasonable compromise between the actual factors for the 196 Mhz MIPS R10000 to 500 Mhz Alpha normalization, which were .6667, .5556, .3863, and .4262 for $N = 100$, 316, 1,000, and 3,162 respectively.

Note that the `Zhang` does much better than might have been expected based on our results for random instance classes, getting within 1% of optimal for all but one instance (`atex600`), for which it is still only 2.6% over optimal. Moreover, it never takes more than a minute on any instance. This is in contrast to `Helsgaun` which, although it is within 1% of optimum for `all` the instances, has gigantic running times for many of them (over 20 hours for `dc932`). Of our two choices for fast heuristics, `Patch` is by far the faster overall, often a factor of 10 faster than `KP`, and provides comparable results, with the exception of `atex600`, where it is almost 35% over optimal compared to 4.25% for `KP`. Moreover, `Patch` is itself within 1% of optimal for all the instances through the `dc` class, and is often significantly faster than `Zhang`. (The

	% Excess over Optimal				Normalized Running Time				
Instance	P	KP	Z	H	P	KP	Z	H	OPT
rbg323	.00	.78	.00	.00	.22	3.71	.22	46.5	35.5
rbg358	.00	1.50	.00	.00	.27	3.33	.27	120.5	19.5
rbg403	.00	.22	.00	.00	.48	9.00	.48	221.5	22.9
rbg443	.00	.11	.00	.00	.53	11.74	.52	164.0	20.6
td100	.00	.00	.00	.00	.02	.20	.02	.5	4.1
td1000	.00	.01	.00	.00	1.87	7.29	1.87	140.5	370.5
td316	.00	.00	.00	.11	.20	3.87	.21	24.5	123.5
big702	.00	2.10	.00	.41	.98	6.04	.97	119.0	149.6
dc849	.04	.62	.00	.23	2.67	114.80	26.33	2180.0	378.3
dc563	.39	.79	.09	.12	1.49	111.95	10.23	2231.5	1449.7
dc134	.25	.57	.18	.02	.04	13.43	.14	6.5	63.7
dc895	.55	.60	.42	.25	4.74	144.43	56.54	22077.0	35926.1
dc176	.81	.67	.09	.49	.07	20.48	.19	105.0	195.1
dc112	.26	.39	.13	.28	.03	15.47	.11	4.5	76.2
dc188	.57	.59	.23	.13	.08	12.98	.19	28.5	122.8
dc932	.30	.26	.13	.26	4.98	119.17	43.86	72373.0	14722.4
dc126	.94	.65	.21	.54	.03	22.69	.15	5.5	48.3
ftv170	1.38	4.44	.36	.00	.06	.09	.10	2.5	66.3
ftv150	2.57	4.43	.00	.00	.04	.07	.05	1.0	17.4
ftv160	.60	5.89	.11	.00	.04	.07	.07	2.0	40.7
ftv130	3.64	2.16	.00	.00	.03	.06	.07	1.0	26.5
ftv140	2.77	3.15	.00	.00	.03	.06	.05	1.5	29.5
ftv110	3.32	4.04	.00	.00	.02	.04	.04	1.0	24.9
ftv120	3.09	3.12	.00	.00	.02	.05	.05	1.0	33.5
ftv100	2.69	3.11	.00	.00	.02	.04	.03	1.0	22.6
code198	.00	.00	.00	.00	.05	.54	.06	6.5	9.0
code253	3.52	.10	.28	.28	.09	1.09	.23	24.5	22.6
atex600	34.94	4.25	2.60	.82	.54	3.38	24.03	123.0	–

Table 1.18. Results for `realworld` instances. `Patch`, `Zhang`, and `Helsgaun` are abbreviated by `P`, `Z`, and `H` respectively. The running time for `OPT` includes both the time to obtain an initial upper bound using `Zhang` and the time to run `Concorde` using its default settings on the transformed instance. The relatively small time needed to perform the ATSP to STSP transformation is not included. Instance `atex600` could not be optimized in reasonable time using `Concorde`'s default settings.

two have roughly equivalent running times for the first eight instances, with `Zhang` occasionally appearing slightly faster because of fluctuations in running times from run to run.)

This suggests various hybrid approaches. For example, one could use `Patch` unless the solution gets too far above the AP bound, in which case `KP` could be run as a backup. For the current testbed, this strategy would always get within 4.25% of optimal and never take more than 8 normalized seconds per instance. An analogous `Zhang`/`Helsgaun` combination would always get with in 1% in no more than about 2 minutes.

5. Conclusions and Further Research

In this chapter we have evaluated implementations of a broad range of heuristics for the ATSP, including the best ones currently available. We have seen wide varieties of behavior (in tour quality and/or running time) for the same heuristic, depending on instance class, and we have reached some tentative conclusions about what strategies to try when confronting a real-world application, depending on the trade-off one is willing to make between tour quality and running time.

Our conclusions should be viewed only as preliminary, however. First, we do not yet know how typical are the random instance classes and real-world instances in the testbeds covered in this study. As we have seen, there can be large performance differences depending on instance structure, and we cannot claim to have contemplated all likely applications. Based on the results of Section 4.8, one might suppose that the real world is actually somewhat easier than the hardest of our random instance classes (e.g. `shop` and `rtilt`) assuming one chooses the appropriate heuristic. However, there is no guarantee this will always be true. Conversely, we have been using codes with their default settings, and it may well be that better performance (e.g., faster times for the same tour quality) might be obtained by general or class-specific tweaking.

One definite challenge for the future concerns large instances. Here we set the bound at 3,162 cities, because of the memory constraints for storing instances using the $\Theta(N^2)$ full distance matrix representation. For many applications (including the ones behind most of our classes), there is actually an application-specific linear-space representation for instances, and there might well be much larger instances for which tours are needed. Given that essentially all the heuristics we studied here have running time growth rates of $\Theta(N^2)$, $\Theta(N^3)$, or worse, new algorithmic ideas may well be needed if we are to deal effectively with such situations.

Acknowledgment. The research of Gregory Gutin was supported in part by an EPSRC grant. The research of Anders Yeo was supported in part by the grant "Research Activities in Discrete Mathematics" from the Danish Natural Science Research Council. The research of Weixiong Zhang was supported in part by NSF grants #IRI-9619554, #IIS-0196057, and #E1A-0113618 and by DARPA cooperative agreements F30602-00-2-0531 and F33615-01-C-1897. The authors thank Pablo Moscato for helpful comments on an early draft of the chapter.

References

[1] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Finding tours in the TSP, 1998. Draft available from `http://www.math.princeton.edu/tsp/`.

[2] D. Applegate, R.E. Bixby, V. Chvátal, and W. Cook. On the Solution of Traveling Salesman Problems. *Documenta Mathematica*, Extra Volume ICM III:645–656, 1998. The 12/15/1999 release of the `Concorde` code is currently available from `http://www.math.princeton.edu/tsp/concorde.html`.

[3] E. Balas and N. Simonetti. Linear time dynamic programming algorithms for new classes of restricted TSPs: A computational study. *INFORMS Journal on Computing*, 13:56–75, 2001. The code is currently available from `http://www.contrib.andrew.cmu.edu/~neils/tsp/index.html`.

[4] J. L. Bentley. Fast algorithms for geometric traveling salesman problems. *ORSA Journal on Computing*, 4:387–411, 1992.

[5] L. Buriol, P. M. França, and P. Moscato. A new memetic algorithm for the asymmetric traveling salesman problem. Submitted for publication, 2001.

[6] E. K. Burke, P. I. Cowling, and R. Keuthen. Embedded local search and variable neighborhood search heuristics applied to the travelling salesman problem. Unpublished manuscript, 2000.

[7] E. K. Burke, P. I. Cowling, and R. Keuthen. Effective local and guided variable neighborhood search methods for the asymmetric travelling salesman problem. In E. J. W. Boers, J. Gottlieb, P. L. Lanzi, R. E. Smith, S. Cagnoni, E. Hart, G. R. Raidl, and H. Tijink, editors, *Applications of Evolutionary Computing, Proceedings of the EvoWorkshops 2001*, Lecture Notes in Computer Science **2037**, pages 203–212, Berlin, 2001. Springer-Verlag.

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

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

[10] J. Cirasella, D.S. Johnson, L.A. McGeoch, and W. Zhang. The asymmetric traveling salesman problem: Algorithms, instance generators, and tests. In A.L. Buchsbaum and J. Snoeyink, editors, *Algorithm Engineering and Experimentation, Third International Workshop, ALENEX 2001*, Lecture Notes in Computer Science **2153**, pages 32–59. Springer-Verlag, Berlin, 2001.

[11] A. 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.

[12] 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.

[13] F. Glover, G. Gutin, A. Yeo, and A. Zverovich. Construction heuristics and domination analysis for the asymmetric TSP. *European J. Oper. Res.*, 129:555–568, 2001.

[14] G. Gutin and A. Zverovich. Evaluation of the Contract-or-Patch Heuristic for the Asymmetric TSP. Submitted.

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

[16] M. Held and R.M. Karp. The Traveling Salesman Problem and Minimum Spanning Trees: Part II. *Mathematical Programming*, 1:6–25, 1971.

[17] K. Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operations Research*, 12:106–130, 2000. Source code currently available from the author's `http://www.dat.ruc.dk/~keld/`.

[18] D. S. Johnson, J. L. Bentley, L. A. McGeoch, and E. E. Rothberg. Near-optimal solutions to very large traveling salesman problems. Monograph, in preparation, 2003.

[19] D.S. Johnson, L.A. McGeoch, F. Glover, and C. Rego. Website for the DIMACS Implemenation Challenge on the Traveling Salesman Problem: `http://www.research.att.com/~dsj/chtsp/`.

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

[21] R.M. Karp and J.M. Steele. Probabilistic analysis of heuristics. In A.H.G. Rinnooy Kan E.L. Lawler, J.K. Lenstra and D.B. Shmoys,

editors, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization.* Wiley, Chichester, 1985.

[22] S. Kataoka and S. Morito. Selection of relaxation problems for a class of asymmetric traveling salesman problem instances. *J. Oper. Res. Soc. of Japan*, 34:233–249, 1991.

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

[24] S. Lin and B.W. Kernighan. An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21:972–989, 1973.

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

[26] D.L. Miller and J.F. Pekny. Exact solution of large asymmetric traveling salesman problems. *Science*, 251:754–761, 1991.

[27] B. M. Moret, D. A. Bader, and T. Warnow. High-performance algorithmic engineering for computational phylogenetics. In *Proc. 2001 Int'l Conf. Computational Science (ICCS 2001), San Francisco.* Lecture Notes in Computer Science 2073-2074, Springer Verlag, 2001.

[28] B. M. Moret, S. Wyman, D. A. Bader, T. Warnow, and M. Yan. A new implementation and detailed study of breakpoint analysis. In *Proc. 6th Pacific Symp. on Biocomputing (PSB 2001), Hawaii*, pages 583–594. World Scientific Pub., 2001.

[29] G. Reinelt. TSPLIB — a traveling salesman problem library. *ORSA Journal on Computing*, 3:376–384, 1991. `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`.

[30] 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.

[31] 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.

[32] 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.