Chapter 9

EXPERIMENTAL ANALYSIS OF HEURISTICS FOR THE STSP

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

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

1. Introduction

In this and the following chapter, we consider what approaches one should take when one is confronted with a real-world application of the TSP. What algorithms should be used under which circumstances? We are in particular interested in the case where instances are too large for optimization to be feasible. Here theoretical results can be a useful initial guide, but the most valuable information will likely come from testing implementations of the heuristics on test beds of relevant instances. This chapter considers the symmetric TSP; the next considers the more general and less well-studied asymmetric case.

For the symmetric case, our main conclusion is that, for the types of instances that tend to arise in practice, heuristics can provide surprisingly good results in reasonable amounts of time. Moreover the large collection of heuristics that have been developed for the STSP offers a broad range of tradeoffs between running time and quality of solution. The heuristics range from those that take little more time than that needed to read an instance and still get within 50% of optimum to those that get within a few percent of optimum for 100,000-city instances in seconds to those that get within fractions of a percent of optimum for instances of this size in a few hours.

The relevant level of performance will of course vary, depending on the application. This chapter provides a tentative characterization of the most promising approaches at many levels of the tradeoff hierarchy. In this way we hope to put previous theoretical and experimental work into a practical perspective.

In order to provide an up-to-date picture of the state of the art, the authors of this chapter, together with Fred Glover and Cesar Rego, organized a DIMACS Implementation Challenge[1] on the STSP. The Challenge began in June 2000 and continued through November, with additional data collected through June 2001. Researchers from all over the world, including all the current top research groups, ran their codes on instances from a collection of test suites, reporting running times and the lengths of the constructed tours. They also reported running times for a special benchmark code distributed by the organizers. These times allowed us to estimate the speeds of their machines (as a function of instance size) and thus to normalize running times to what they might (approximately) have been had the codes all been run on the same fixed machine. We thus can provide detailed comparisons between a wide variety of heuristics and implementations with specific attention to robustness, scalability, and solution quality/running time tradeoffs. In this way we hope to improve on earlier studies, such as those of Golden and Stewart [388], Bentley [103], Reinelt [710, 711], and Johnson-McGeoch [463], which covered fewer heuristics and instances and did not provide as convenient mechanisms for future comparability.

The remainder of this chapter is organized as follows. In Section 2 we provide more details about the Challenge, the testbeds of instances it covered, its participants, our scheme for normalizing running times, and our methods for evaluating tour quality. In Section 3, we describe the various heuristics studied, divided into groups on the basis of approach and speed, and summarize the experimental results obtained for them. Section 4 then presents some overall conclusions and suggestions for further research.

We should note before proceeding that certain heuristics described elsewhere in this book are for various reasons not covered in this chapter. Perhaps our foremost omission is the approximation schemes for geometric STSP's of Arora, Mitchell, et al. [35, 601, 696], as described in Chapter 5. These heuristics, despite their impressive theoretical guar-

[1]DIMACS is the Center for Discrete Math and Theoretical Computer Science, a collaboration of Rutgers and Princeton Universities with Bell Labs, AT&T Labs, NEC Labs, and Telcordia Technologies. This was the 8th in the DIMACS Implementation Challenge series. For more information, see `http://dimacs.rutgers.edu/Challenges/`.

antees, have significant drawbacks compared to the competition we shall be describing. Because of the perturbation of the instances that they initially perform, the versions of the heuristics guaranteeing $1+\epsilon$ worst-case ratios are likely to be off by a significant fraction of ϵ even in the average case. Thus, to be competitive with heuristics that typically get within 1 or 2 percent of optimum in practice, one probably must choose $\epsilon < 0.05$. This is likely to make the running times prohibitive, given the large constant factor overheads involved and the fact that the running times are exponential in $1/\epsilon$. It would be interesting to verify that this is indeed the case, but as of this date we know of no attempt at a serious implementation of any of the schemes.

A second hole in our coverage concerns local-search heuristics based on polynomial-time searchable exponential-size neighborhoods, one of the subjects of Chapter 6. We have results for only one such heuristic. Empirical study of such heuristics is still in its infancy, and so far very little has emerged that is competitive with the best traditional STSP heuristics.

The final hole in our coverage is rather large – much of the burgeoning field of metaheuristics is not represented in our results. Although we do cover one set of tabu search implementations, we cover no heuristics based on simulated annealing, neural nets, classical genetic algorithms, GRASP, etc. The Challenge was advertised to the metaheuristic community and announcements were sent directly to researchers who had previously published papers about heuristics of these sorts for the TSP. For various reasons, however, little was received. Fortunately, we may not be missing much of practical value in the context of the STSP. As reported in the extensive survey [463], as of 1997 all metaheuristic-based codes for the STSP were dominated by 3-Opt, Lin-Kernighan, or Iterated Lin-Kernighan. Metaheuristics, if they are to have a role in this area, are more likely to be useful for *variants* on the TSP. For example, they might well adapt more readily to handling side constraints than would more classical approaches.

2. DIMACS STSP Implementation Challenge

For a full description of the DIMACS Implementation Challenge, see the website at `http://www.research.att.com/~dsj/chtsp/`. In addition to providing input for this chapter, the Challenge is intended to provide a continually updated picture of the state of the art in the area of TSP heuristics (their effectiveness, robustness, scalability, etc.). This should help future algorithm designers to assess how their approaches compare with already existing TSP heuristics.

To this end, the website currently makes a variety of material available for viewing or downloading, including instances and instance generators, benchmark codes, raw data from the participants, and statistics and comparisons derived therefrom. Our intent is to maintain the website indefinitely, updating it as new results are reported and adding new instances/instance classes as interesting ones become available.

This chapter presents a summary and interpretation of the available results as of June 2001, representing code from 15 research groups. Many of the groups reported on implementations of more than one heuristic or variant, thus providing us with fairly comprehensive coverage of the classical heuristics for the STSP along with several promising new approaches. For additional details, the reader is referred to the website and to a forthcoming DIMACS Technical Report that will present the data in a more linear fashion. In the remainder of this section, we describe the Challenge testbeds in more detail, as well as our scheme for normalizing running times.

2.1. Testbeds

In designing the Challenge testbeds, we have chosen to ignore instances with fewer than 1,000 cities. This was done for several reasons. First, as we shall see, currently available optimization codes, in particular the publicly available `Concorde` package[2] of Applegate, Bixby, Chvátal, and Cook [29], seem to be able to solve typical STSP instances with fewer than 1,000 cities in quite feasible running times. Indeed, `Concorde` was able to solve all the 1,000-city instances in our random testbeds using its default settings. Normalized running times were typically in minutes, and the longest any such instance took was just a little over two hours. Second, if one is only willing to spend seconds rather than minutes, the best of the current heuristics are hard to beat. For instance, the $N/10$-iteration version of the publicly available LKH code of Keld Helsgaun [446] can get within 0.2% of optimum in no more than 20 seconds (normalized) for each of our 1,000-city random instances and for the six `TSPLIB`[3] instances with between 1,000 and 1,200 cities. Thus it is not clear that heuristics are needed at all for instances with fewer than 1,000 cities, and even if so, high quality solutions can be obtained in practical running times using publicly available codes. The real research question is how heuristic performance scales as instance sizes grow, es-

[2]Currently available from `http://www.math.princeton.edu/tsp/concorde.html`.

[3]`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 [709].

pecially since many modern applications generate instances with 10,000 cities or more.

Our second decision was to concentrate primarily on geometric instances in two dimensions. Most experimental research on the STSP to date has concentrated on such instances. This is largely because the major applications for the STSP, both industrial and academic, have been of this sort. Consequently, many codes have been written to exploit such structure and only work for such instances. (Limited experimentation with higher dimensional instances suggests that the lessons learned in two dimensions carry over at least to three or four [461, 465].)

A third decision, based on common practice in the literature and an assumption made by many codes, was to restrict attention to instances in which inter-city distances are all integral.

The Challenge test suite contains three classes of geometric instances:

- Random Uniform Euclidean Instances ("Uniform"). Here the cities are points whose two coordinates are each integers chosen randomly from the interval $[0, 10^6)$, with instance sizes increasing roughly by factors of $\sqrt{10}$ from $N = 1,000$ to $N = 10,000,000$. Distances are the Euclidean distance rounded to the nearest integer. There are ten instances with 1,000 cities, five with 3,162, three of size 10,000, two each of sizes 31,623 and 100,000, and one each of sizes 316,228, 1,000,000, 3,162,278, and 10,000,000. Instances of this type have been widely studied and yield an interesting view on asymptotic performance.

- Random Clustered Euclidean Instances ("Clustered"). Here we choose $N/100$ cluster centers with coordinates chosen uniformly in $[0, 10^6)$, and then for each of the N cities we randomly choose a center and two normally distributed variables, each of which is then multiplied by $10^6/\sqrt{N}$, rounded, and added to the corresponding coordinate of the chosen center. Distances are again the Euclidean distance rounded to the nearest integer. For this class there are ten instances with 1,000 cities, five with 3,162, three of size 10,000, two each of sizes 31,623 and 100,000, and one of size 316,228. These were designed to be challenging for local search heuristics.

- All 33 geometric instances in TSPLIB with 1,000 or more cities as of June 2001. These instances range in size from 1,000 cities to 85,900. All are 2-dimensional with rounded Euclidean distances (either rounded up or to the nearest integer). Most come either from geography (coordinates of actual cities, with the earth viewed as planar) or from industrial applications involving circuit boards, printed circuits, or programmable gate arrays.

As to applications with non-geometric instances, these tend to be asymmetric as well as non-geometric and hence will be covered in the next chapter. For the STSP Challenge, our main source of non-geometric instances consisted of random symmetric distance matrices, the only non-geometric class that has previously been widely studied in the context of the STSP. Although such instances have no direct relevance to practice, they do offer a substantial challenge to many heuristics and thus are useful in studying the robustness of various approaches. For our Random Matrix testbed, distances were independently chosen integers distributed uniformly in the interval $[0, 10^6)$. We include four instances of size 1,000, two of size 3,162, and one of size 10,000. (Since an instance of this type consists of roughly $N^2/2$ integers, storage can become a problem for larger N.) In addition, our testbed contains the one instance from TSPLIB that is given by its distance matrix and has 1,000 or more cities (si1032).

Although participants were encouraged to run their codes on as many of the testbed instances as possible, this was not always possible. There were four main reasons why participants could not handle the entire test suite:

1. The participant's code was too slow to handle the largest instances on the participant's machine.
2. The code was fast enough, but required too much memory to handle the largest instances on the participant's machine.
3. The participant's code was designed to handle geometric instances and so could not handle instances given by distance matrices.
4. The participant's code was not designed to handle instances with fractional coordinates. Despite the fact that the TSPLIB instances all have integral inter-city distances, 13 of the 33 geometric TSPLIB instances in our test suite have fractional coordinates.

Not all of these need be defects of the underlying *heuristic*. In particular, (4) can typically be circumvented by additional coding, as several of our participants have shown, and (1) and (2) can often be ameliorated by cleverer code-tuning and memory management (or more powerful machines). Reason (3) may be less forgiving, however: Some heuristics are geometric by definition (e.g., Convex Hull Cheapest Insertion), and others will experience substantial slowdowns if they are unable to exploit geometric structure. In any case, we can only report on the results for the implementations we have, although where relevant we will try to identify those heuristics for which faster or more robust implementations may well be possible.

2.2. Running Time Normalization

Running time comparisons are notoriously difficult to make with any precision, even when all codes are compiled using the same compiler and compiler options and run on the same machine. By allowing participants to compile their own codes and run them on their own machines, we have made the problem substantially more difficult. However, since we did not wish to restrict participation to those who were willing to share their source codes, and we wanted to establish a record of the state of the art that might still be meaningful after the machines we currently have are obsolete and forgotten, there seemed to be no other choice.

In order to provide some basis for comparison, we thus have distributed a benchmark STSP code, an implementation of the Greedy (or Multi-Fragment) heuristic that uses K-d trees to speed up its operation on geometric instances. Participants were asked to run this code on their machines for a set of instances covering the whole range of sizes in the Challenge test suite and to report the resulting running times. Note that one cannot accurately quantify the difference in speeds between two machines by a single number. Because of various memory hierarchy effects, the relative speeds of two machines may vary significantly as a function of the size of the input instances. Figure 9.1, which graphs the running time of the benchmark code as a function of instance size for a variety of machines, shows how widely relative machine speeds can vary as a function of N. (In the chart, running times are divided by $N \log N$, the approximate number of basic operations performed by the heuristic.)

Using these reports, we can normalize running times to approximately what they would have been on a specific benchmark machine: a Compaq ES40 with 500-Mhz Alpha processors and 2 Gigabytes of main memory. The basic plan is to compute a normalization factor as a function of N. For N equal to one of the instance sizes in our Uniform test suite, we simply use the ratio between the benchmark code's time on the source machine and on the ES40 for the test instance of that size (assuming the benchmark code could be run on the source machine for instances that large). For other values of N, we interpolate to find the appropriate normalization factor.

There are multiple sources of potential inaccuracy in this process. Linear interpolation is an inexact approach to getting intermediate normalization factors. A particular code may require more (or less) memory for a given value of N than does the benchmark `Greedy` code. It may make more (or less) efficient use of instruction and data caches than the benchmark code. Also, our normalization process de-emphasizes the time to read an instance. Reading times do not necessarily differ by the

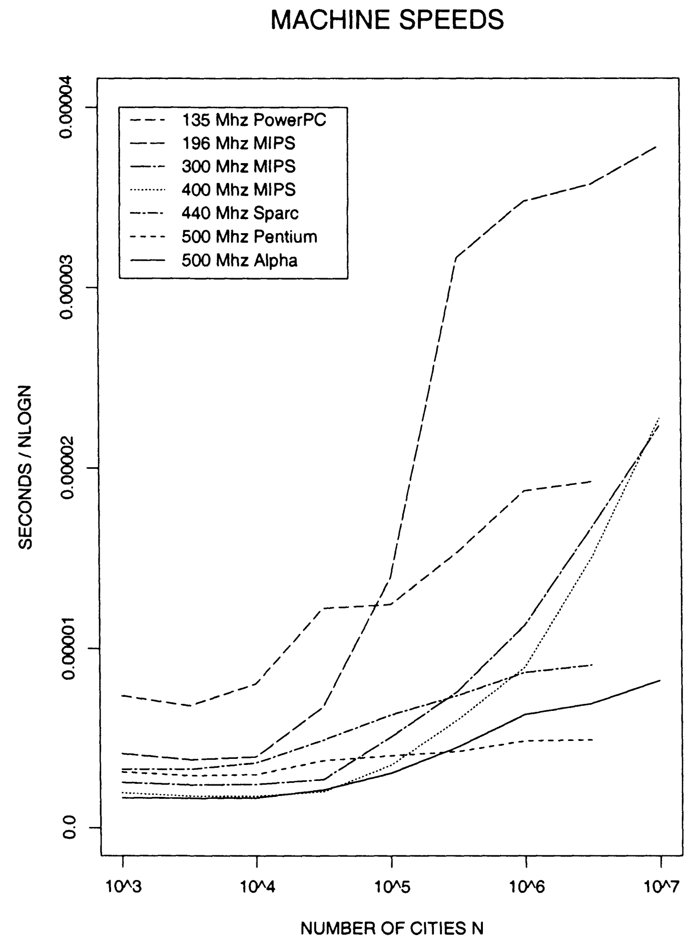


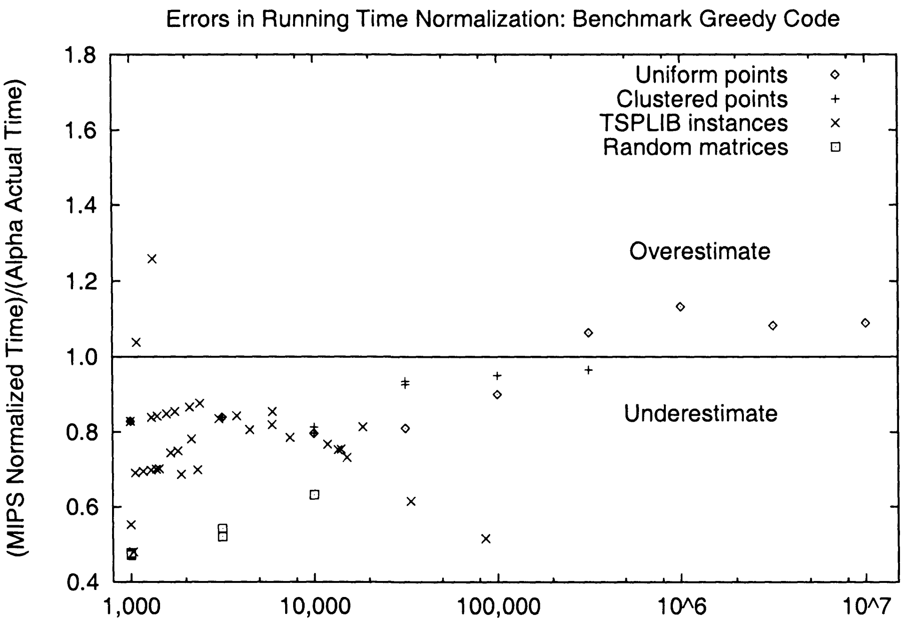
Figure 9.1. Running times for benchmark greedy code as a function of instance size for a variety of microprocessor-based machines. The microprocessors are listed in the order of their average times for 1000-city instances.

same factors as do CPU times, and they can be a significant component of the running time for the faster codes in our study, especially on smaller instances and on instances given by full distance matrices.

The de-emphasis arises because of the way we deal with the fact that systems typically only report running times in increments of 0.01 seconds. The benchmark `Greedy` code is so fast that its running time is typically 0.00, 0.01, or 0.02 for 1,000-city instances. This makes it difficult to derive precise normalization factors based on a single run. Thus, when we perform the benchmark runs on the smaller instances, we report the total time over a series of runs on the same instance. The number of runs is chosen so that the product of the number of runs and N is roughly 1,000,000, with just one run performed for each instance with 1,000,000 or more cities. Although the basic data structures are rebuilt in each run, the instance itself is read only once. A single read makes sense since the heuristics we are testing only read the instance once. However, by reducing the proportion of the total time devoted to reading, this approach may misrepresent the impact of reading time on heuristics for which it is a major component of the total time.

To get a feel for the typical accuracy of our normalization procedure, see Figure 9.2 which charts, for the benchmark `Greedy` code and a Johnson-McGeoch implementation of the Lin-Kernighan heuristic the ratio between the actual time on the target ES40 machine and the normalized time based on compiling the code and running it on a 196-Mhz MIPS R10000 processor in an SGI Challenge machine. Note that for each heuristic, the error is somewhat systematic as a function of N, but the error is not consistent between heuristics. For `Greedy` the tendency is to go from underestimate to overestimate as N increases, possibly reflecting the reading time underestimate mentioned above. For Lin-Kernighan, on the other hand, read time is not a major component of running time on geometric instances, and for these the tendency is to go from overestimate to underestimate, possibly because this code needs substantially more memory than `Greedy` and because the MIPS machine has larger 2nd level caches than does the ES40. It is worth noting, however, that for both codes the estimate is still typically within a factor of two of the correct time.

Unfortunately, even if we can estimate running times for specific codes to within a factor of two, this may not imply anything so precise when talking about *heuristics.* Differing amounts of low-level machine-specific code tuning can yield running time differences of a factor of two or more, even for implementations that supposedly use the same data structures and heuristic speedup tricks. And the latter can cause even greater changes in speed, even though they are not always specified in a high-level description of a heuristic. Thus, unless one sees order-of-magnitude differences in running times, or clear distinctions in running time growth rates, it is difficult to draw definitive conclusions about the relative efficiency of heuristics implemented by different people on different machines. Fortunately, there *are* orders-of-magnitude differences in running time within the realm of TSP heuristics, so some conclusions about relative efficiency will be possible.



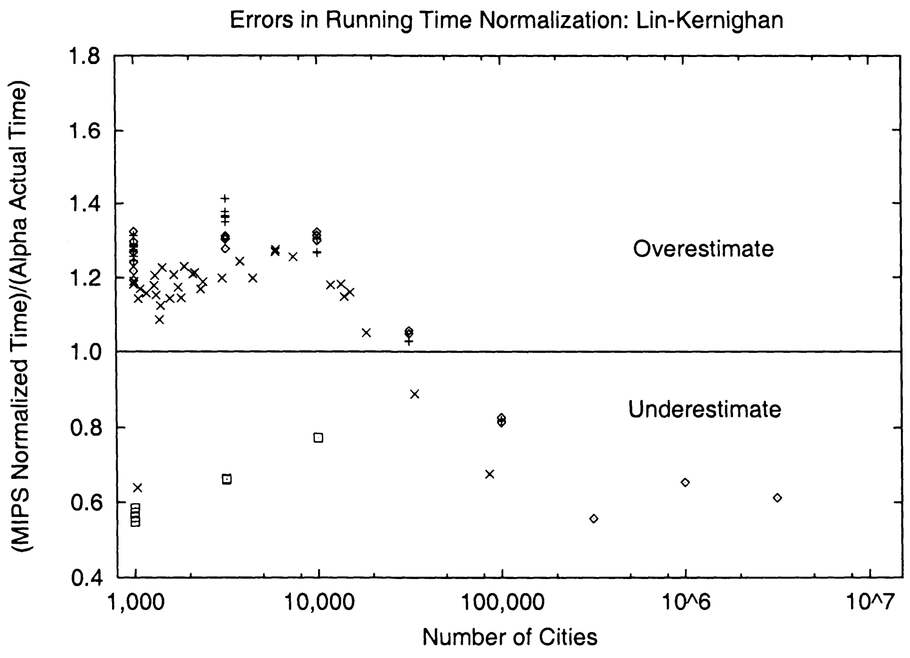


Figure 9.2. Ratios of predicted running time to actual running time on a Compaq 500-Mhz Alpha processor for the Benchmark Greedy code and for the Johnson-McGeoch implementation of Lin-Kernighan.

2.3. Evaluating Tour Quality

The gold standard for tour quality is of course the distance from the optimal solution, typically measured as the percentage by which the tour's length exceeds the length of an optimum tour. In order to use this standard, one unfortunately must *know* the optimal solution value.

Modern optimization technology is surprisingly effective: provably optimal solutions have been found for all but one of the instances in TSPLIB with 15,112 cities or fewer, and the Concorde code of Applegate, Bixby, Chvátal, and Cook [29] is able to solve all the random instances in the Challenge test suite with 3,162 or fewer cities (and the one 10,000-city Random Matrix instance). However, a prime reason for using heuristics is to get reasonable results for instances that are too difficult for current optimization algorithms to work. For this reason our test suite contains many instances for which optimal tour lengths are not yet known.

In order to provide a point of reference that is similar across all instances, our default comparison is thus to the Held-Karp lower bound on the optimal solution [444, 445]. This is the linear programming relaxation of the standard integer programming formulation for the STSP, as described in Chapter 2. Johnson et al. [465] argue that this bound is a good surrogate for the optimal solution value. For Random Uniform Euclidean instances in particular, they conjecture that the expected gap between the optimal tour length and the Held-Karp bound is asymptotically less than 0.65% and they provide extensive experimental evidence supporting this conjecture.

Table 9.1 shows the percent by which the optimal tour length exceeds the Held-Karp bound for all the instances in our test suite where the optimal is known. Note that the typical excess is less than 1% and the maximum excess observed is 1.74%. Moreover, although the optimal tour length is not yet known for four of the largest TSPLIB instances, for each one a tour is known that is within .54% of the Held-Karp bound.

The table also includes the normalized running times for computing the optimal tour lengths and for computing the Held-Karp bounds, which is typically much easier. When a running time is reported for an optimal tour length computation, it represents the time taken by Concorde using its default settings. For the random instances Concorde was run on our 196-Mhz MIPS processors. Times for the TSPLIB instances are those reported by Applegate et al. on their TSP webpage (`http://www.math.princeton.edu/tsp/`) for the same 500-Mhz Alpha processor used in our benchmark machine. For those instances with known optima but no quoted running time, additional expertise was needed (and running time – more than 22 CPU years for d15112).

Random Uniform Euclidean				TSPLIB			
Name	%Gap	Opttime	HKtime	Name	%Gap	Opttime	HKtime
E1k.0	0.77	1406	2.13	dsj1000	0.61	410	3.68
E1k.1	0.64	3855	2.15	pr1002	0.89	34	2.40
E1k.2	0.72	1211	2.02	si1032	0.08	25	11.32
E1k.3	0.62	956	1.92	u1060	0.65	571	3.62
E1k.4	0.69	330	1.69	vm1084	1.33	605	2.40
E1k.5	0.59	233	2.42	pcb1173	0.96	468	1.70
E1k.6	0.79	2940	1.67	d1291	1.18	27394	4.54
E1k.7	0.94	8003	1.95	rl1304	1.55	189	4.08
E1k.8	1.01	4347	1.65	rl1323	1.65	3742	4.49
E1k.9	0.61	189	2.14	nrw1379	0.43	578	2.40
E3k.0	0.71	533368	9.57	fl1400	1.74	1549	9.83
E3k.1	0.67	425631	10.54	u1432	0.29	224	2.42
E3k.2	0.74	342370	9.41	fl1577	1.66	6705	38.19
E3k.3	0.67	147135	10.30	d1655	0.94	263	6.51
E3k.4	0.73	–	8.07	vm1748	1.35	2224	4.43
Random Clustered Euclidean				u1817	0.90	449231	5.01
C1k.0	0.54	337	9.83	rl1889	1.55	10023	11.45
C1k.1	0.41	534	10.84	d2103	1.44	–	8.19
C1k.2	0.42	320	8.79	u2152	0.62	45205	8.10
C1k.3	0.53	214	7.63	u2319	0.02	7068	3.16
C1k.4	0.58	768	9.36	pr2392	1.22	117	5.75
C1k.5	0.58	139	9.29	pcb3038	0.81	80829	7.26
C1k.6	0.73	1247	7.07	fl3795	1.04	69886	123.66
C1k.7	0.58	449	13.24	fnl4461	0.55	–	12.47
C1k.8	0.34	140	10.40	rl5915	1.56	–	42.00
C1k.9	0.66	703	9.61	rl5934	1.38	–	56.15
C3k.0	0.62	16009	53.03	pla7397	0.58	–	55.42
C3k.1	0.61	17754	126.49	rl11849	1.02	–	102.41
C3k.2	0.70	18237	80.39	usa13509	0.66	–	120.20
C3k.3	0.57	6349	71.57	d15112	0.52	–	90.13
C3k.4	0.57	4845	44.02				
Random Matrices							
M1k.0	0.01	60	5.47	M3k.0	0.00	612	40.35
M1k.1	0.03	137	5.51	M3k.1	0.01	546	39.52
M1k.2	0.01	151	5.63	M10k.0	0.00	1377	367.84
M1k.3	0.01	169	5.26				

Table 9.1. For instances in the Challenge test suite that have known optimal solutions, the percent by which the optimal tour length exceeds the Held-Karp bound and the normalized running times in seconds for computing each using Concorde with its default settings. ("–" indicates that the default settings did not suffice.) For random instances, suffixes 1k, 3k, and 10k stand for 1,000, 3,162, and 10,000 cities respectively. The number of cities in a TSPLIB instance is given by its numerical suffix.

The Held-Karp times reported are also for `Concorde`, which contains a flag for computing the Held-Karp bound. Although the linear program that defines the bound involves an exponential number of "subtour" constraints, there are simple routines for finding violated constraints of this type, and typically not many of these need be found in order to solve the LP exactly. This is much more effective (and accurate) than the Lagrangean relaxation approach originally suggested by Held and Karp. `Concorde` was able to compute the bound using its default settings on our local SGI machine for all the instances in the Challenge test suite with less than a million cities, with the maximum normalized running time being roughly 4 hours for the 316,228-city random clustered Euclidean instance. Using more powerful machines at Rice University, Bill Cook used the code to compute the bound for our million-city instance. For the two instances in our test suite with more than a million cities, we relied on the empirical formula derived in [465], which was off by less than .02% for the million-city instance.

3. Heuristics and Results

As noted in the Introduction, currently available heuristics for the STSP provide a wide variety of tradeoffs between solution quality and running time. For example, Figure 9.3 illustrates the average performance of a collection of heuristics on the three 10,000-city instances in our testbed of Uniform instances. The underlying data is presented in Table 9.2. Details on the heuristics/implementations represented in the chart and table will be presented later in this section.

The normalized running times range from 0.01 seconds to over 5 hours, while the percentage excess over the Held-Karp bound ranges from about 35% down to 0.69% (which is probably within 0.1% of optimum). There is not, however, a complete correlation between increased running time and improved quality. Some heuristics appear to be *dominated*, in that another heuristic can provide equivalently good tours in less time or can provide better tours in the same time or less. For example, Bentley's implementation of Nearest Insertion (`NI`) from [103] is dominated by his implementation of Nearest Neighbor (`NN`), and the Tabu Search implementation `Tabu-SC-SC` is dominated by three sophisticated iterated variants on Lin-Kernighan (`MLLK-.1N`, `CLK-ABCC-N`, and `Helsgaun-.1N`).

In this chapter, we shall separately consider groups of heuristics clustered in different regions of this trade-off spectrum, attempting to identify the most robust undominated heuristics in each class. Although we shall concentrate primarily on undominated heuristics, we will not do so exclusively. Dominated heuristics for which theoretical results have

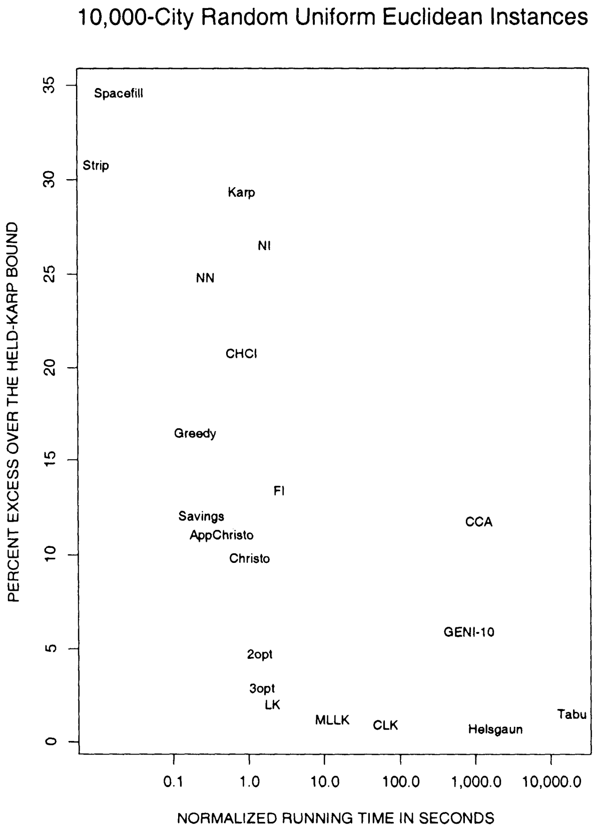


Figure 9.3. Average tradeoffs between tour quality and normalized running time for a variety of heuristics applied to 10,000-city Random Uniform Euclidean Instances. The full abbreviations for the heuristic names are given in Table 9.2 below and explained in Table 9.19 at the end of this chapter.

Heuristic	Excess Over HK	Time (Seconds)	Heuristic	Excess Over HK	Time (Seconds)
Spacefill	34.56	0.02	AppChristo	11.05	0.44
Strip	30.75	0.01	Christo-S	9.81	1.04
Karp-20	29.34	0.85	GENI-10	5.89	823.00
NI	26.50	1.71	2opt-JM	4.70	1.41
NN	24.79	0.28	3opt-JM	2.88	1.50
CHCI	20.73	0.83	LK-JM	2.00	2.06
Greedy	16.42	0.20	Tabu-SC-SC	1.48	18830.00
FI	13.35	2.59	MLLK-.1N	1.18	12.75
Savings	12.03	0.24	CLK-ABCC-N	0.90	63.91
CCA	11.73	1129.00	Helsgaun-.1N	0.69	1840.00

Table 9.2. Average tour quality and normalized running times for various heuristics on the 10,000-city instances in our Random Uniform Euclidean testbed.

been proven or which have received significant publicity will also be covered, since in these cases the very fact that they are dominated becomes interesting. For example, assuming that the triangle inequality holds, Nearest Insertion can never produce a tour longer than twice optimum, whereas NN can be off by a factor of $\Theta(\log N)$ [730], which makes the fact that the latter can be better in practice somewhat surprising.

Moreover, domination for one class of instances need not tell the full story. Table 9.3 summarizes the relative performances of Bentley's implementations of Nearest Insertion and Nearest Neighbor as a function of instance size for our three geometric instance classes, represented by the shorthands U (Random Uniform Geometric instances), C (Random Clustered Geometric instances) and T (TSPLIB instances). Figure 9.4 presents a more detailed picture, with charts that depict the relative solution quality and running times of the two implementations for each of the geometric instances in our testbeds to which both could be applied. (The implementations were designed to exploit geometry as much as possible, but do not handle fractional coordinates.) Analogous tables and charts for other pairs of heuristics can be generated and viewed online via "Comparisons" page at the Challenge website. One can also generate charts in which the running time for a single heuristic is compared to various growth rates, just as the running times for Greedy were compared to $N \log N$ in Figure 9.1.

Average Percent Excess: NI over NN

	N=1000	3162	10K	31K	100K	316K	1M	3M	10M
U	-0.55	-0.15	1.37	2.22	2.86	2.85	2.95	3.31	3.31
C	-4.47	-3.89	-2.42	-2.91	-2.60	-3.04			
T	-1.41	-3.44	-2.73	-1.27	0.35				

Average Running Time Ratio: NI/NN

U	5.5	5.2	5.6	5.6	5.7	5.7	5.1	4.9	5.0
C	8.5	9.7	13.0	11.5	13.1	11.9			
T	6.2	5.7	6.5	8.5	10.0				

Table 9.3. Average comparisons between Nearest Insertion (NI) and Nearest Neighbor (NN) on our geometric testbeds. Bentley's implementations [103] of both heuristics were used. A positive entry in the "Excess" table indicates that the NI tours are longer by the indicated percentage on average. As in all subsequent tables of this sort, the TSPLIB averages are over the following instances: pr1002, pcb1173, rl1304, and nrw1379 for $N = 1,000$, pr2392, pcb3038, and fnl4461 for $N = 3162$, pla7397 and bfd14051 for $N =$ 10K, pla33810 for $N =$ 31K, and pla85900 for $N =$ 100K. These may not be completely typical samples as we had to pick instances that most codes could handle, thus ruling out the many TSPLIB instances with fractional coordinates.

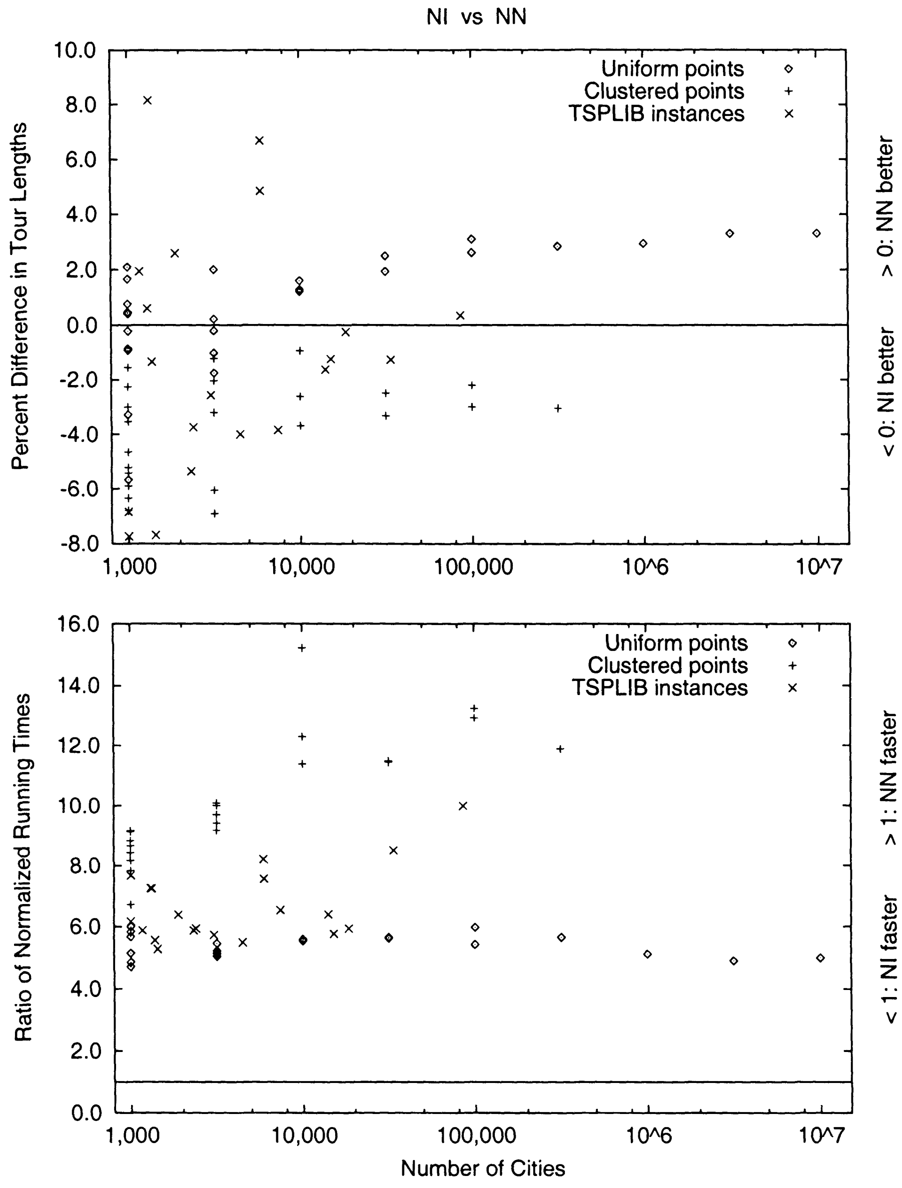


Figure 9.4. Tour length and normalized running time comparisons: Nearest Insertion versus Nearest Neighbor.

As to the Nearest Insertion versus NN comparison, we see that the tour length results for 10,000-city Uniform instances are echoed for larger instances from that class, but do not predict results for the other instance classes. Indeed, NI consistently provides better tours than NN for Clustered instances and is also better for a majority of the TSPLIB instances. NI *does* remain slower than NN (by a factor of 4 or more on the same machine for all instances), but for certain instance classes one might be willing to pay this price to get better tours. We thus cannot say that NI is consistently dominated by NN, although we will see many examples of consistent domination in what follows.

The body of this section is divided into seven parts, each covering a group of related heuristics. The first three subsections cover what are typically called *tour construction* heuristics, i.e., heuristics that incrementally construct a tour and stop as soon as a valid tour is created. The remaining sections concern heuristics with a *local search* component, i.e., heuristics that repeatedly modify their current tour in hopes of finding something better.

In Section 3.1, we consider tour construction heuristics designed more for speed than for quality. The Strip heuristic and the Spacefilling Curve heuristic, for example, do little more than read the instance and sort. Sections 3.2 and 3.3 cover the remainder of the classical tour construction heuristics, divided somewhat arbitrarily into those that build tours by adding edges one at a time, as in NN (Section 3.2), and those where the augmentation may involve *replacing* edges, as in Nearest Insertion and Christofides (Section 3.3). Since tour construction heuristics for the STSP are not covered in detail elsewhere in this book, we shall in these sections summarize what is known theoretically about these heuristics as well as discussing their empirical behavior.

The remaining sections cover local search heuristics, the subject of Chapter 8. Section 3.4 covers simple local search heuristics like 2-Opt and 3-Opt. Section 3.5 covers the famous Lin-Kernighan heuristic and its variants. Section 3.6 discusses various heuristics that involve repeated calls to a local search heuristic as a subroutine, such as the Chained Lin-Kernighan heuristic introduced by [563]. It also covers our one set of Tabu Search implementations, which operate in a similar fashion. The final Section 3.7 considers heuristics that take this one step further and use a heuristic like Chained Lin-Kernighan as a subroutine.

Although we do not have room to provide full descriptions of all the heuristics we cover, we present at least a high-level summary of each, mentioning relevant theoretical results and, where possible, pointers to sources of more detailed information. If implementation details can have a major impact on performance, we say something about these as well.

3.1. Heuristics Designed for Speed

In this section we cover heuristics for geometric instances of the STSP that are designed for speed rather than for the quality of the tour they construct. In particular, we restrict attention to heuristics whose observed total running time is within a small factor (10 or less) of the time simply to read the (x, y) coordinates for N cities using standard formatted I/O routines. The normalized times for the latter are shown in Table 9.4. Note that one can read instances much faster than this by using lower-level routines to exploit the fact that coordinates come in a known format. Using such an approach, one can speedup the reading of our 10,000,000-city instance by a factor of 80 or more [26]. This would have a significant impact on the overall speed of our fastest heuristics, which currently do not take this approach. The restriction to geometric instances, i.e., ones given by tuples of coordinates, is important: If one required the instance to be given by its full distance matrix, *many* of our heuristics would satisfy the above speed criterion, but could hardly be called "fast" given that instance reading itself would take $\Theta(N^2)$ time.

At present, three heuristics meeting the above criteria have received significant coverage in the literature: the Strip, Spacefilling Curve, and Fast Recursive Partitioning heuristics. In this section we cover all three, plus an obvious enhancement to the first. All are defined in terms of 2-dimensional instances but could in principle be generalized to geometric instances in higher dimensions. The results we report were all obtained on the same machine (as were the reading times mentioned above), which removes running-time normalization as an extra source of inter-heuristic variability. All the heuristics begin by making one pass through the data to determine minimum and maximum x and y coordinates and thus the minimum enclosing rectangle for the point set.

Strip. In this heuristic, we begin by dividing the minimum enclosing rectangle into $\sqrt{N/3}$ equal-width vertical strips and sorting the cities in each strip from top to bottom. We then construct a tour by proceeding from the leftmost strip to the rightmost, alternately traveling up one

Average Normalized Running Time in Seconds: Read

	N=1000	3162	10K	31K	100K	316K	1M	3M	10M
U	0.00	0.01	0.02	0.06	0.13	0.25	1.0	3.4	12
C	0.00	0.01	0.03	0.06	0.12	0.25			
T	0.00	0.00	0.03	0.06	0.12				

Table 9.4. Average normalized times for reading instances using the standard I/O routines of C, compiled for MIPS processors using gcc.

strip and down the next, with one final (long) edge back from last city in the rightmost strip to the first in the leftmost.

This heuristic can be traced back to 1959, when Beardwood, Halton, and Hammersley [94] introduced it as a tool in a proof about the average-case behavior of the optimal tour length. It is easy to see that `Strip`'s tours can be as much as $\Omega(\sqrt{N})$ times optimum in the worst case. However, for points uniformly distributed in the unit square (a continuous version of our Uniform instance class), the expected length of the `Strip` tour length can be shown to be no more than $0.93\sqrt{N}$ [500]. Given that the expected Held-Karp bound for such instances is empirically asymptotic to $0.71\sqrt{N}$ [465], this means that `Strip`'s expected excess for such instances should be less than 31%.

Results for `Strip` are summarized in Table 9.5. Note that for Uniform instances, the upper bound on average case excess mentioned above is close to `Strip`'s actual behavior, but `Strip`'s tours are much worse for the other two classes. `Strip` *is* fast, however: Even for the largest instances, its running time averages less than 3.5 times that for just reading the instance, and the time is basically independent of instance class (as are the times for all the heuristics covered in this section). Since most of `Strip`'s computation is devoted to sorting, this implementation uses a variety of sorting routines, depending on instance size. For the largest instances, a 2-pass bucket sort using 2^{16} buckets is used. This means that theoretically the implementation should run in linear time for our instances, although in practice it appears to be a bit slower, presumably because of memory hierarchy effects.

It is fairly easy to see why `Strip`'s tours for Clustered instances are poor: They jump between clusters far too frequently. For instances in `TSPLIB` something similar might be going on, but one might wonder whether some of its poor performance is just an artifact of the fact that

Average Percent Excess over the HK Bound: `Strip`

	N=1000	3162	10K	31K	100K	316K	1M	3M	10M
U	31.94	32.23	30.75	30.16	30.36	30.22	30.10	30.10	30.09
C	115.61	160.82	174.39	190.62	198.05	201.76			
T	61.33	36.26	73.03	91.86	73.28				

Average Normalized Running Time in Seconds

U	0.00	0.02	0.03	0.09	0.20	0.53	2.8	10.4	41
C	0.00	0.02	0.04	0.09	0.19	0.53			
T	0.01	0.01	0.04	0.09	0.18				

Table 9.5. Average performance of the `Strip` heuristic.

we chose *vertical* strips. To examine this question, we implemented a composite heuristic that applies both the original Strip heuristic and the variant that uses *horizontal* strips and returns the better result (*2-Way Strip*). Given that reading time is amortized across the two runs of `Strip`, the overall running time only goes up by a factor of 1.3 to 1.7. Unfortunately the average improvements are minor, with a few individual exceptions, such as an improvement from an excess of about 119% to one of 52% for the `TSPLIB` instance `rl5915`. Details can be explored on the Challenge website. A more promising competitor to `Strip` is the following.

Spacefilling Curve (`Spacefill`). This heuristic was invented by Platzmann and Bartholdi [671]. The cities are visited in the order in which they would be encountered while traversing a spacefilling curve for the minimum enclosing rectangle. As with `Strip`, most of the time is spent simply in sorting. For full details see [671]. Platzmann and Bartholdi prove that the Spacefilling Curve heuristic can never produce tours that are worse than $O(\log N)$ times optimum. Bertsimas and Grigni [108] exhibit pointsets for which `Spacefill` is this bad. Again, however, one can get bounded average-case ratios. A probabilistic analysis in [671] shows that when cities are uniformly distributed in the unit square the asymptotic expected tour length is approximately 35% above the empirical estimate of the expected Held-Karp bound. (Interestingly, the ratio of the heuristic's tour length to $\sqrt{N}$ does not go to a limit as $N \to \infty$, although the lim inf and lim sup are extremely close [671].) Table 9.6 presents results for the inventors' implementation.

As with `Strip`, the overall running time for `Spacefill` stays within a factor of 3.5 of that for merely reading an instance. Moreover, although `Spacefill`'s average excess for Uniform instances matches the theoretical prediction and hence is 4-5 percentage points worse than that for

Average Percent Excess over the HK Bound: `Spacefill`

	N=1000	3162	10K	31K	100K	316K	1M	3M	10M
U	32.25	33.40	34.56	34.71	34.94	35.00	35.09	35.06	35.08
C	41.08	60.74	72.85	95.48	76.81	51.68			
T	45.36	40.27	36.03	40.97	37.39				

Average Normalized Running Time in Seconds

U	0.00	0.01	0.04	0.11	0.24	0.64	3.0	10.6	39
C	0.00	0.01	0.04	0.11	0.24	0.62			
T	0.00	0.01	0.04	0.11	0.23				

Table 9.6. Average performance of the Spacefilling Curve heuristic.

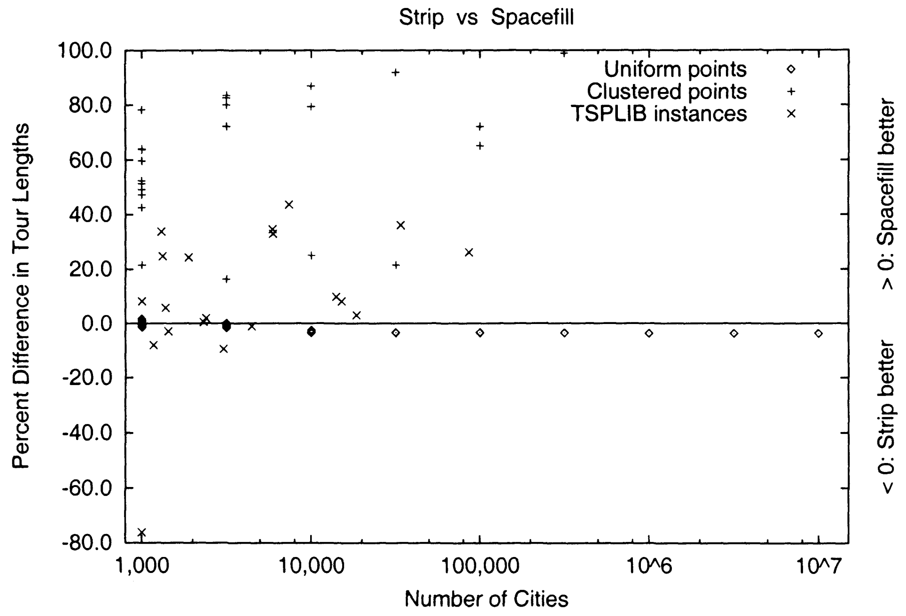


Figure 9.5. Tour quality comparisons for the Strip and Spacefilling Curve heuristics.

Strip, it is substantially better for the other two classes. Figure 9.5 provides a more detailed picture of the comparison. Based on these results, the Spacefilling Curve heuristic would seem to be the preferred choice, if one must choose only one of the two heuristics. It also would be preferred over our final candidate.

Fast Recursive Partitioning (FRP). In this heuristic, proposed by Bentley in [103], we begin by hierarchically partitioning the cities as in a K-d tree. This starts with the minimum enclosing rectangle and then recursively splits each rectangle containing more than $B = 15$ cities into two rectangles with roughly half as many cities. If the parent rectangle is longer than it is wide, the median x-coordinate for cities in the rectangle is found and a vertical split is made at this x value; otherwise the median y-coordinate is found and a horizontal split is made at this value of y. Call the final rectangles, all containing 15 or fewer cities, *buckets*. Nearest Neighbor tours are constructed for all the buckets, and these are then patched together to make an overall tour. FRP is effectively dominated by Spacefill, which is on average 2.5-3 times faster and is better for all but 4 instances in our testbed, usually by more than 10%. (The four exceptions are three of the 23 clustered instances and the TSPLIB instance dsj1000 which is itself a clustered instance, produced by an earlier version of our generator.)

3.2. Tour Construction by Pure Augmentation

In this section we cover heuristics that build their tours by adding one edge at a time, making each choice based on the length of the edge to be added. This is in contrast to `Strip` and `Spacefill`, which can be viewed as building their tours one edge at a time, but with choices based only on simple directional constraints. The class includes the Nearest Neighbor heuristic as well as the Greedy heuristic and several variants, including the lesser-known but quite effective "Savings" heuristic of [201].

As in the previous section, all the results we report were generated on same machine (using 196-Mhz R10000 MIPS processors), thus ensuring that running time comparisons will not be biased by normalization errors. However, it still may be dangerous to draw conclusions about the relative speeds of closely matched heuristics, since these may be highly implementation-dependent. We will illustrate this by presenting results for multiple implementations of the same heuristics (implemented by different programmers). These differ significantly in constant factors and asymptotic growth rates even though all follow the recommendations of Bentley's influential papers [102, 103] that promoted the use of K-d trees (short for "K-dimensional binary search tree" [101, 102]) and lazily updated priority queues for exploiting the geometric structure of instances and avoiding unnecessary work. These two are such a significant component of many of the implementations described in this section and later that they are worth a few more words.

***K-d* Trees.** In defining the FRP heuristic in the previous section, we introduced the fundamental hierarchical partition of the instance space that underlies the K-d tree. (For K-d trees, however, we typically split any rectangle that contains more than 8 cities, as opposed to the bound of 15 used in FRP.) This partition is represented by a tree, with a vertex for each rectangle. For each vertex that represents a split rectangle, we store the coordinate of the median point that was used in splitting the rectangle (x if the split was left-right, y if the split was top-bottom), together with pointers to the vertices representing the two subrectangles into which it was split. (In *bottom-up* K-d trees, we also store a pointer to the parent of the given rectangle.) For a vertex corresponding to a final unsplit rectangle, we store a list of the cities in that rectangle. The partition and associated tree can be constructed in $O(N \log N)$ time.

Simple recursive routines can be used to search a K-d tree in various ways. We here mention three important ones. These all assume the existence of an auxiliary array `present[]` that tells us which of the cities are relevant to the current search. First, there is the *nearest neighbor* search: Given a city c, find the present city that is nearest to c. Second, there is the *fixed-radius near neighbor* search: given a city c and a radius r, return (in some order) all those present cities c' such that $d(c, c') \leq r$. The third is *ball* search from city c, which assumes an additional array `rad[]` of radii for all the cities and returns all those present cities c' for which $d(c, c') \leq$ `rad[`c'`]`, i.e., all those cities c' for which the ball of radius `rad[`c'`]` around c' contains c. For details on how these can be efficiently implemented, see [101, 102]. The first two searches involve the execution of $O(\log N)$ computer instructions for most data sets, while the third may take somewhat longer, depending on the number of relevant balls. The speed of all three can vary depending on the sophistication of the implementation and its interaction with the memory hierarchy of the machine on which the heuristic is run.

This section's simple heuristics require only the first of these three operations (or a slight variant on it). The others come into play for the more complicated heuristics of the next section. (An alternative to K-d trees, the Delaunay triangulation, was exploited by Reinelt in [710, 711]. This appears to be a competitive approach, but the results presented in [710, 711] are not sufficiently comparable to ours to yield firm conclusions. K-d trees, at any rate, offer substantially more power and flexibility.)

Lazily Updated Priority Queues. The use of this data structure in TSP heuristics was first suggested in [102, 103]. A priority queue contains items with associated values (the *priorities*) and supports operations that (1) remove the highest priority item from the queue and deliver it to the user (a "pop"), (2) insert a new item, (3) delete an item, and (4) modify the priority of an item (an "update"). Algorithms textbooks contain a variety of implementations for this data structure, most of which support all the operations in time $O(\log N)$ or less, but with different tradeoffs and constant factors. The choice can have a significant effect on running time. A major additional savings is possible if we can reduce the number of updates actually performed, which is what happens with lazy evaluation. This technique can be used if we know that no update will ever *increase* a priority. In this case, we need not perform an update when it first takes effect, but only when the popped (highest priority) item has an outdated priority. In this case, that item's priority is reevaluated and it is reinserted into the queue.

We are now prepared to describe this section's heuristics and how they are implemented.

Nearest Neighbor (`NN`). We start by picking a initial city c_0. Inductively, suppose $i < N - 1$ and $c_0, c_1 \ldots, c_i$ is the current partial tour. We then choose c_{i+1} to be the nearest city to c_i among all those cities not yet present in the tour. If $i = N - 1$ we add the edge $\{c_{N-1}, c_0\}$, thus completing the tour. For non-geometric instances, this heuristic would take time $\Theta(N^2)$, but for geometric instances the use of K-d trees can reduce this to something like $O(N \log N)$ in practice.

Double-Ended Nearest Neighbor (DENN). We start by picking an initial city c_0. Inductively, suppose a and b are the endpoints of the current partial tour and that it does not yet contain all the cities. If a' and b' are the nearest non-tour cities to a and b respectively, we choose the one that is closest to its respective endpoint and add the corresponding edge to the tour. If all the cities *are* in the tour, we add the edge $\{a, b\}$ and are done. This heuristic can be implemented to run almost as fast as NN in practice, since we only need to compute a nearest neighbor when the tour gains a new endpoint or when an endpoint's previous nearest neighbor is added to the other end of the tour.

Greedy. We start by sorting all the potential tour edges $\{c, c'\}$ in order of increasing length. We then build a tour, viewed as a set of edges, by going through the edges in order, starting with the shortest, and adding $\{c, c'\}$ so long as neither c nor c' already has degree 2 and the new edge does not complete a cycle with fewer than N vertices.

As described, the implementation would take time $\Theta(N^2 \log N)$, and that is the time that would be required for non-geometric instances. For geometric instances, this can be reduced by combining K-d trees and nearest neighbor searches with a lazily updated priority queue, as suggested by [102, 103]. This is done as follows. After first constructing the K-d tree, we find the nearest neighbor c' for each city c and put the ordered pair (c, c') in the priority queue with priority $-d(c, c')$. Thus the queue contains only N entries and the highest priority entry corresponds to the shortest edge, i.e., the first that Greedy would add to its tour. As we proceed, we will mark a city as *present* if it does not have degree 2 in the current tour. If c is a city with degree 1 in the tour, we will let end[c] denote the city at the other end of the tour path starting with c. Note that we could build the Greedy tour in just $N - 1$ pops if we maintained the property that at all times the priority queue contained, for each city c that is currently present, the nearest *eligible* neighbor, i.e., the nearest present city other than end[c].

Unfortunately, maintaining this property might require many updates after each pop. A single city c' can be the nearest eligible neighbor for many other present cities. When c' attains degree 2, it will no longer be eligible and each city c that thought c' was its nearest eligible neighbor will have to find a new partner. Note, however, that whenever the nearest neighbor of a city c needs to be updated, it will be replaced by a new city that is at least as far away from c as the city it replaced. So we can do lazy updating. When we pop the highest priority item in the queue (c, c'), there are two cases. If c already has degree 2 in the tour, we simply discard this pair and pop the next one. If c has degree at most 1 in the tour and c' is present and not equal to `end[`c`]`, we can add edge $\{c, c'\}$ to the tour. Otherwise, we temporarily mark `end[`c`]` (if it exists) as "not present," find a new nearest present neighbor c'' for c, insert (c, c'') in the queue, reset `end[`c`]` (if it exists) to "present," and pop the new highest priority pair.

Boruvka. This heuristic is a variant on `Greedy` devised by Applegate, Bixby, Chvátal, and Cook in analogy with the classic minimum spanning tree of O. Borůvka [132]. As with the above `Greedy` implementation, we start by computing the nearest neighbor for each city. Instead of putting the resulting pairs into a priority queue, however, we simply sort them in order of increasing edge length. We then go through the list, edge by edge, adding each to the tour we are building so long as we legally can do so. In other words, when a pair (c, c') is encountered where c' is no longer eligible, we discard the pair without updating even if c still hasn't attained degree 2. After we have gone through the whole list, we probably won't yet have a tour, so we repeat the process again, this time restricting attention to cities that do not yet have degree 2 and eligible neighbors. We continue to repeat the process until a tour is constructed. In comparison to `Greedy`, this heuristic replaces priority queue overhead with simple sorting, but may have to do more nearest neighbor searches. It is not *a priori* evident whether tours should be better or worse.

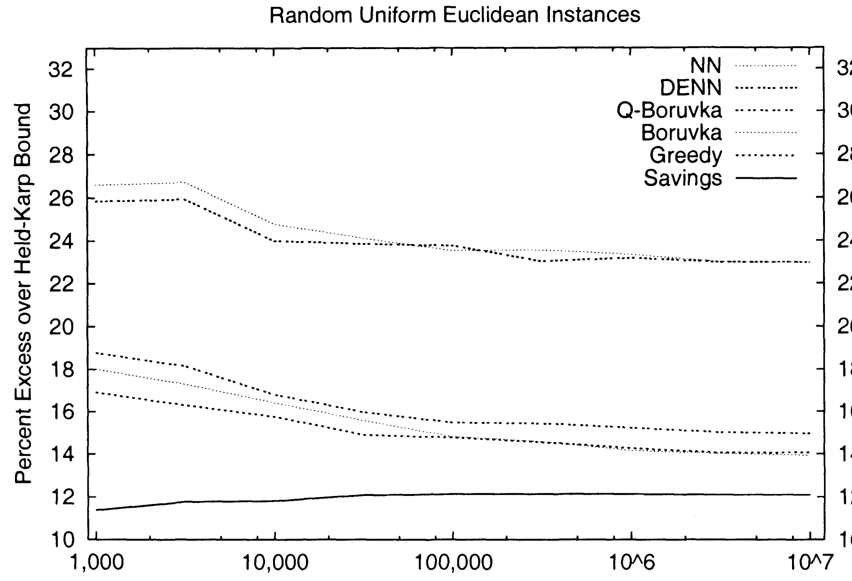
Quick Boruvka (`Q-Boruvka`). This variant, also due to Applegate, Bixby, Chvátal, and Cook, dispenses with the sorting step in `Boruvka`, presumably trading tour quality for an increase in speed. We go through the cities in some arbitrary fixed order, skipping a city if it already has degree 2 and otherwise adding an edge to the nearest eligible city. At most two passes through the set of cities will be required.

Savings. This is a specialization to the STSP of a more general vehicle routing heuristic proposed by Clarke and Wright in [201]. Informally, it works by starting with a pseudo-tour, consisting of a multigraph that has two edges from an arbitrary *central city* c_0 to each of the other cities. We then successively look for the best way to "shortcut" this graph by replacing a length-2 path from one (non-central) city to another by a direct link. In practice, the Savings heuristic works like `Greedy`, except with a surrogate distance function. For any pair of cities c, c' other than c_0, the surrogate distance function is $D(c, c') = d(c, c') - d(c, c_0) - d(c_0, c')$. Given a K-d tree, nearest neighbors under this surrogate distance function can be computed using a slightly more complicated version of the standard nearest neighbor search, as shown in [461]. The only other difference from Greedy is that c_0 is not put in the priority queue, and we stop growing the tour when it contains $N-2$ edges, at which point it must be a path, which we can complete to a tour by adding the edges from c_0 to its two endpoints.

Theoretical worst-case results have been proved for several of these heuristics, assuming the triangle inequality, i.e., that for all triples of cities (c_1, c_2, c_3), $d(c_1, c_2) \leq d(c_1, c_3) + d(c_3, c_2)$. For none of these heuristics are the tours guaranteed to be within a constant factor of optimum, but we can provide some bounds. NN can be shown never to produce a tour longer than $(1 + \lceil \log N \rceil)/2$ times optimum, and there are instances that force it to generate tours roughly 2/3 that long [730]. Greedy never produces a tour longer than $(1 + \log N)/2$ times optimum [633] (roughly the same upper bound as for NN), but the worst instances known only cause it to produce tours that are $(\log N/3 \log\log N)$ times optimum [325]. Savings never produces a tour longer than $(1 + \log N)$ times optimum [633] (a weaker bound than for the other two heuristics), but the worst examples known produce tours that are again only $(\log N/3 \log\log N)$ times optimum [325]. We are unaware of worst-case results for the relatively more recent Boruvka variants, but the bounds for these are likely to be no better than those for Greedy.

Figure 9.6 graphs the average tour quality as a function of N for the six heuristics described in this section and our two classes of random geometric instances. For both classes it is typical of most of the heuristics we cover that the average percentage excess over the Held-Karp bound appears to approach an asymptotic limiting value, although those limits are usually different for the two classes. For Uniform instances, the limiting values for NN and DENN appear to be roughly 23% above the Held-Karp bound, compared to 15% for Q-Boruvka, 14% for Greedy and Boruvka, and 12% for Savings. DENN appears to yield slightly better averages than NN for the smaller instances but its advantage vanishes once $N > 10,000$. (Variations after that point are attributable to the small number of instances in our samples). Greedy appears to be slightly better than Boruvka for the smaller instances, but this advantage disappears by the time $N = 100,000$. All the heuristics perform significantly more poorly for the Clustered instances, but the relative asymptotic ranking remains the same. For TSPLIB instances, the tour quality tends to lie between these two extremes, except that Savings is typically 1-2% better on the larger TSPLIB instances than it is even for Uniform instances of similar size.

When we order the heuristics by running time, they appear in roughly reverse order, which implies that no one of them is dominated by any of the others. Table 9.7 lists normalized running times for Uniform in-



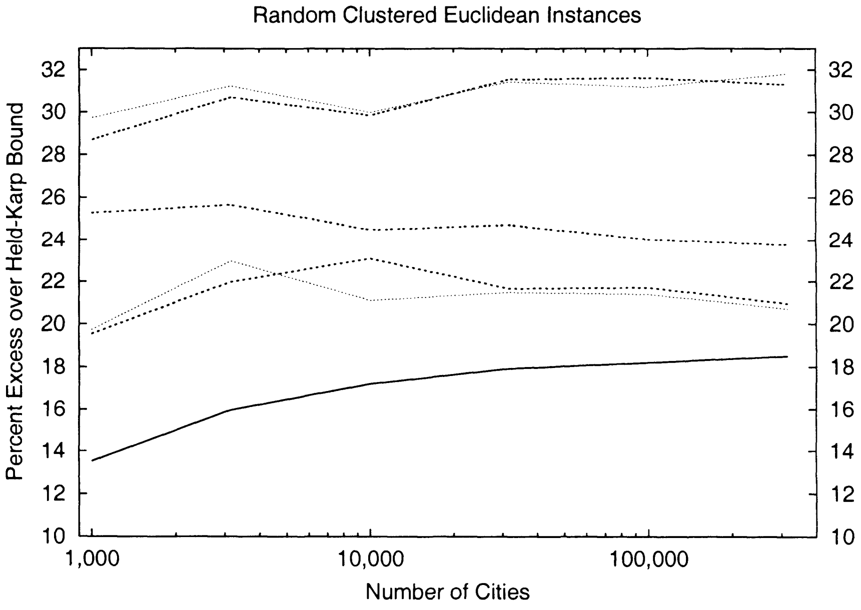


Figure 9.6. Average percentage excess for pure augmentation heuristics. (For an explanation of the abbreviations, see the text or Table 9.19.) Note that the ranges of N are different for the two classes of instances.

stances as a function of N. Times for Clustered instances are roughly the same. Those for TSPLIB instances tend to be faster, possibly because the added structure of these instances limits the breadth of the K-d tree nearest-neighbor searches. The table covers three families of implementations: Bentley's (-B) implementations of NN and Greedy, the Johnson-McGeoch (-JM) implementations of the same heuristics plus Savings, and the Concorde (-ABCC) implementations of those two plus the two Boruvka variants. (The suffixes are the implementers' initials.)

For codes with common implementers, the code that produces better tours typically takes longer time for all values of N, with two exceptions: DENN takes about the same time as NN (as predicted), and the Johnson-McGeoch implementation of Savings sometimes beats their implementation of Greedy. Although the nearest-neighbor searches are more complicated under Savings than under Greedy, this is balanced by the fact that far fewer of them need to be made and the two heuristics end up taking roughly the same overall time. Generally, the time for Greedy/Savings is 2 to 5 times that for NN, with the biggest differences occurring for the Concorde implementations.

Cross-family comparisons are more problematic, presumably because of implementation differences. The Bentley and Concorde implementations exploit up-pointers in their K-d trees, whereas the Johnson-McGeoch implementations do not. Up-pointers add constant-factor overhead but can greatly reduce the depth of searching. As a result, the Johnson-McGeoch implementations are faster than other two when $N \leq 100,000$ but slower when N is larger. Bentley's implementations are in C++ while the other two are in C, which might explain in part why Bentley's implementations lose to Concorde on both NN and Greedy.

The observed running times for all the implementations appear to have two components: one that grows more slowly than $N \log N$ and

N =	1000	3162	10K	31K	100K	316K	1M	3M	10M
NN-ABCC	0.01	0.03	0.10	0.27	0.88	2.73	14.2	58.6	247
NN-B	0.02	0.10	0.32	0.82	2.17	5.13	25.5	100.9	400
NN-JM	0.01	0.02	0.08	0.22	0.53	5.34	25.6	103.5	453
DENN-B	0.02	0.10	0.32	0.83	2.08	5.18	25.8	102.2	405
Q-Boruvka-ABCC	0.01	0.04	0.12	0.31	1.13	4.02	22.4	96.2	404
Boruvka-ABCC	0.02	0.05	0.18	0.65	2.60	7.46	36.9	151.4	597
Greedy-ABCC	0.02	0.07	0.27	1.12	4.55	12.64	59.2	221.8	863
Greedy-B	0.05	0.19	0.62	1.77	5.14	12.77	60.4	232.5	930
Greedy-JM	0.02	0.06	0.20	0.81	4.05	21.28	100.8	357.3	1450
Savings-JM	0.02	0.08	0.26	0.83	3.13	21.02	99.6	385.5	1604

Table 9.7. Normalized running times in seconds for Pure Augmentation heuristics and Random Uniform Euclidean instances.

dominates when $N \leq 100,000$, and one that grows faster than $N \log N$ and dominates once $N > 100,000$. This latter component in fact appears to be growing faster than $N \log^2 N$, although no worse than $O(N^{1.25})$. The relative importance of these two components and their crossover point depend on the heuristic and the implementation. Determining the causes of this behavior is an interesting question for future research.

With respect to tour quality, there is no appreciable difference between the various implementations of NN and Greedy. This is as should be expected, given the well-defined nature of those heuristics. Different implementations do not, however, always yield the same tours. This is because of different tie-breaking rules and because the output of NN depends on the starting city chosen.

Our overall conclusion is that, although there are few cases of pure domination here, three of the six heuristics adequately cover the range of trade-offs: DENN, Boruvka, and Savings (with the K-d tree implementation chosen based on the expected size of the instances to be handled). In most real-world applications, we would expect Savings to be fast enough to supplant the other two. The above conclusions assume that one is looking for a stand-alone heuristic. As we shall see in Sections 3.4 and 3.5, different conclusions may hold if one is choosing a method for generating starting tours in a local search heuristic.

The story for non-geometric applications may also differ, and we are less able to provide insight here. The only non-geometric implementations we have are for Greedy and NN, and our testbed of non-geometric instances consists mostly of Random Matrices, whose relevance to practice is suspect. For what it is worth, Greedy continues to provide substantially better tours than NN for these instances and now takes roughly the same time. Unfortunately, that time is now $\Theta(N^2)$, and both heuristics produce tour lengths whose average ratio to the optimum appears to grow with N and exceeds 2 by the time $N = 10,000$. See [461].

3.3. More Complex Tour Construction

In this section we consider somewhat more complicated heuristics, but ones that still build tours incrementally. Many of these heuristics, even ones with appealing theoretical performance guarantees, are dominated by Savings. Results for the dominated heuristics will not be covered in full detail here, although they can be viewed at the Challenge website.

Nearest Insertion and its Variants (NI,NA,NA$^+$). Start with a partial tour consisting of some chosen city and its nearest neighbor. Then repeatedly choose a non-tour city c whose distance to its nearest neighbor among the tour cities is minimum, and insert it as follows:

- **Insertion Rule** (NI). Insert c between the two consecutive tour cities for which such an insertion causes the minimum increase in tour length.
- **Addition Rule** (NA). Insert c next to its nearest neighbor in the tour on the side (before or after) that yields the minimum increase in the tour length.
- **Augmented Addition Rule** (NA$^+$). Insert c as in NI, but restrict attention to pairs of consecutive tour cities at least one of which is no further from c than twice the distance to c's nearest neighbor in the tour.

NI NA, and NA$^+$ are all guaranteed to produce tours that are no longer than $(2-\frac{2}{N})$ times optimum assuming the triangle inequality holds, and all can produce tours that bad [730]. As explained by Bentley in [103], which introduced the augmented addition rule, they can all be implemented to exploit geometry, although the process is complicated. (NI requires a ball search and NA$^+$ requires a fixed-radius near neighbor search.) As might be expected, this added complexity (even for NA) means that Bentley's implementations of all three heuristics are substantially slower than Savings. They also produce worse tours for all instances in our geometric testbeds. For Uniform instances NI and NA$^+$ have an average percentage excess over the Held-Karp bound that approaches 27% as compared to 12% for Savings, while the limiting percentage for NA is 32.5%. Thus all three variants are dominated by Savings. The same holds for the following family of theoretically interesting heuristics.

Cheapest Insertion and its Variants (CI,CHCI). In Cheapest Insertion (CI), we start with a partial tour consisting of a chosen city and its nearest neighbor. We then repeatedly choose a triple a, b, c of cities such that a and b are adjacent in the current tour, c is a non-tour city, and the increase in tour length that would occur if c were inserted between a and b is minimized, and perform that insertion. Assuming the triangle inequality, CI obeys the same $2-(2/N)$ times optimum bound as Nearest Insertion. The "Convex Hull" variant CHCI starts by computing the convex hull of the cities and creating a starting tour consisting of these in radial order. CHCI trivially obeys a $3-(2/N)$ bound, given the result for CI.

Implementations can again take advantage of geometry, as explained in [461]. For CHCI, the convex hull can be found by a linear-time algorithm such as Graham's [394]. Even so, the Johnson-McGeoch implementations of CI and CHCI remain substantially slower than their implementation of Savings and are almost universally worse. (CHCI is slightly better than `Savings` on one 1,000-city Clustered instance.) `CHCI` tends to produce better tours than `CI`, but the advantage shrinks as N grows. For Uniform instances, the average percentage excess over the Held-Karp bound for both `CHCI` and `CI` tends to about 22% versus 12% for `Savings`.

Double Minimum Spanning Tree (`DMST`). Construct a multigraph consisting of two copies of a minimum spanning tree for the cities. This graph must have an Euler tour, i.e., a (not necessarily simple) cycle that includes every edge exactly once. Construct one and convert it into a Hamiltonian cycle by taking shortcuts to avoid visiting cities more than once. Assuming the triangle inequality holds, this heuristic obeys the same $2-(2/N)$ worst-case bound as do Nearest and Cheapest Insertion.

Again, geometry can be exploited in implementing `DMST`, in particular for constructing the initial MST. The Euler tour can be found in linear time, as can the shortcuts needed to produce the tour. Unfortunately, we still end up slower than `Savings`, and even if we use the "greedy shortcut" procedure described below in the context of the Christofides heuristic, `DMST` still produces substantially worse tours than those for `Savings`. For Uniform instances the average percentage excess tends toward 40%, and the results for the other classes are comparable.

Karp's Partitioning Heuristic (`Karp`). As in K-d tree construction (and in the `FRP` heuristic of Section 3.1), we begin by recursively partitioning the cities by horizontal and vertical cuts through median cities, although now the median city is included in *both* of the subsets of cities created by the cut through it. This process is continued until no more than C cities are in any set of the partition (C is a parameter). Using the dynamic programming algorithm of Bellman [95], we then optimally solve the subproblems induced by the sets of cities in the final partition. Finally, we recursively patch the solutions together by means of their shared medians. For fixed C, this takes $O(N \log N)$ time.

This heuristic was proposed by Karp in his paper [497], which analyzed the average-case behavior of a closely related, non-adaptive heuristic. For this non-adaptive variant and any $\epsilon > 0$, there exists a C_ϵ such that for Uniform instances the expected ratio of the heuristic's tour length to the optimal tour length is asymptotically no more than $1+\epsilon$. Unfortunately, C_ϵ grows linearly with $1/\epsilon$, and the running time and space requirements of the dynamic programming subroutine are both exponential in C. (See also [500] and Chapter 7.)

The adaptive version of the heuristic we test here is likely to produce better tours and be more robust in the presence of non-uniform data, but this has not been rigorously proved. It suffers from the same drawbacks as far as C is concerned, however, with the largest value that has proved feasible being $C = 20$. Given that the average number of cities in the final partitions can vary from 10 to 20 depending on the value of N, this heuristic has a wildly varying running time as a function of N. The average quality of the tours it produces for Uniform instances also fails to go to a limit as $N \to \infty$. As might be expected, the best results correspond to the worst running times, which themselves can be hundreds of times worse than those for `Savings`. However, even those best results are far worse than those for `Savings`: the lim inf of the Uniform instance excesses is larger than 20% and the results for Clustered and `TSPLIB` instances are substantially worse.

The failings of this approach can be ameliorated if one settles for near-optimal rather than optimal solutions to the final subproblems. This was the approach taken by `FRP`, but it used a small value for C and a poor heuristic (`NN`). If one instead uses a large value for C and one of the much more powerful heuristics we describe later in this chapter, one could do much better. Indeed, this might be a plausible first choice for coping with instances that are too big to handled all-at-once in main memory.

Many other tour construction heuristics that have been proposed in the literature are also dominated by `Savings` (for example, Litke's recursive clustering heuristic [564] and the Greatest Angle Insertion heuristic of Golden and Stewart [388], both implemented to exploit geometry in [461] and covered on the Challenge website). However, none of these are of independent theoretical interest. For the remainder of this section, we concentrate on heuristics that are *not* dominated by `Savings`. We first consider variants on Nearest Insertion that lack its strong theoretical guarantees but perform much better in practice.

Random and Farthest Insertion Variants (`RI`,`RA`,`RA`$^+$,`FI`,`FA`,`FA`$^+$). These heuristics differ from their "Nearest" variants mainly in the choice of city to add. In the "Random" variants the city is simply chosen randomly. In the "Farthest" variants we add the city c with the largest value of $\min\{d(c,c') : c' \text{ is in the tour}\}$. For both sets of variants, we start with a tour consisting of the two maximally distant cities.

The best guarantee currently provable for these heuristics (assuming the triangle inequality) is that all provide tours that are no more than $O(\log N)$ times optimum. At present we do not know whether this bound is tight. The worst examples known for the Random variants were obtained by Azar [50]: Euclidean instances for which with high probability the heuristics produce tours of length $\Theta(\log\log N/\log\log\log N)$ times optimum. The worst examples known for the Farthest variants were obtained by Hurkens [455] and only yield ratios to optimum that approach 6.5 (triangle inequality) or 2.43 (2-dimensional Euclidean).

As with the Nearest variants, these heuristics can be implemented to exploit *K-d* trees. The Random variants save work in identifying the city to insert and so are fastest. The Farthest variants require additional work in order to find the point to be inserted, but this can be done using a *K-d* tree on the tour cities and a lazily updated priority queue that for each non-tour city lists the distance to the closest tour city (at the time the entry was computed). As an indication of the relative asymptotic performance of all these variants, see Table 9.8 which summarizes the average results for Bentley's implementations of them on 100,000 Uniform instances. For comparison purposes the results for the Johnson-McGeoch implementation of `Savings` are also included.

Note first that in each family Augmented Addition takes less than twice as much time as Addition, but provides substantially better tours, especially in the cases of the Random and Farthest families. Second, note that in each family the Augmented Addition variants produce nearly as good tours as do their Insertion siblings at a fraction of the running time cost. Unfortunately, the only one of these heuristics that is clearly competitive with `Savings` in running time (`RA`) produces very poor tours.

Uniform instances, however, don't tell the full story. As an illustration of the total picture, see Figure 9.7, which for all instances in our geometric testbeds without fractional coordinates compares the tour lengths found by `FI` and `Savings`. Although `Savings` typically has an even greater advantage for `TSPLIB` instances than for Uniform ones, a different story holds for the Clustered instance class. For these instances, `RA`$^+$, `RI`, `FA`$^+$, and `FI` all find better tours on average than does `Savings`, ranging from roughly a 1.5% improvement under `RA`$^+$ to 3.0% under `RI` and 3.5% improvement under `FA`$^+$ and `FI`. We should also point out that `RA`$^+$ has another advantage. Although it is slower than `Savings` when $N \leq 100,000$, its running time is similar to that of Bentley's implementation of `Greedy`, in that it becomes faster than `Savings` for larger N (for roughly the same implementation-dependent reasons).

Heuristic	NA	NA$^+$	NI	RA	RA$^+$	RI	FA	FA$^+$	FI	Sav
Excess %	32.5	27.1	27.1	40.5	15.4	15.0	43.7	13.6	13.4	12.1
Seconds	6.6	8.6	12.3	3.2	5.7	20.3	10.7	13.6	27.7	3.1

Table 9.8. Average percentage excesses over the Held-Karp bound and normalized running times for Bentley's implementations of Insertion, Addition, and Augmented Addition heuristics applied to 100,000-city Random Uniform Euclidean instances. For comparison purposes, the last column gives results for the Johnson-McGeoch implementation of `Savings`.

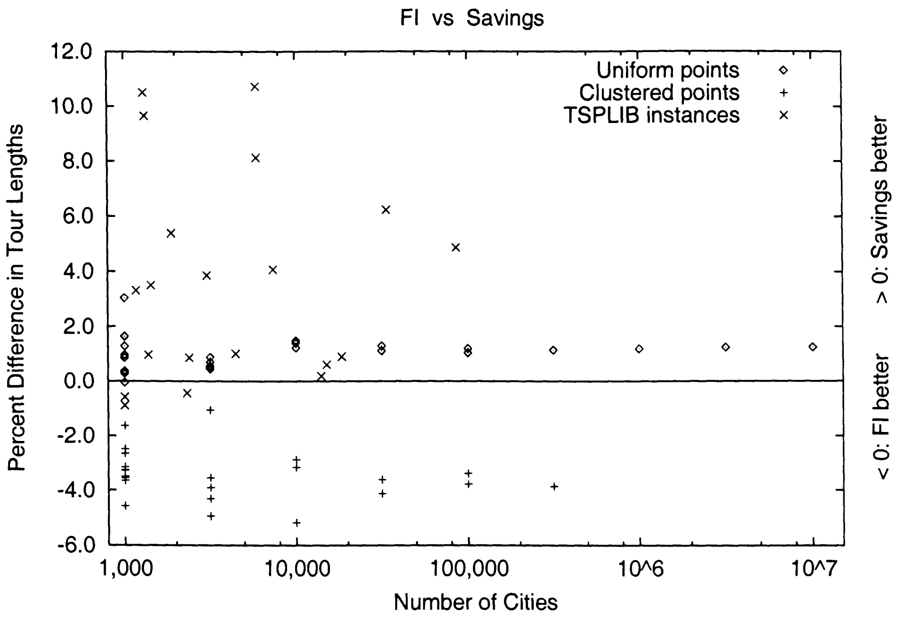


Figure 9.7. Tour quality comparisons for Farthest Insertion and Savings heuristics.

We thus can conclude that these heuristics and Savings are all technically incomparable in that none totally dominates any of the others. Given the artificial nature of the Clustered instances, however, one would probably still choose Savings if one could use only one heuristic.

CCA. This is an abbreviation for "Convex Hull, Cheapest Insertion, Angle Selection," a heuristic proposed by Golden and Stewart in [388] and claimed to be the best tour construction heuristic in that study. As in CHCI, one starts by constructing the convex hull of the cities and proceeds by successively inserting the remaining cities. The choice of insertion is more complicated however. For each non-tour city c, one determines the pair (a_c, b_c) of adjacent tour cities between which c could be inserted with the least increase in overall tour length. We then select that c that maximizes the angle between the edges $\{a_c, c\}$ and $\{c, b_c\}$ and insert it between a_c and b_c in the tour.

Nothing is known theoretically about this heuristic, and its complexity makes it difficult (if not impossible) to exploit geometry when implementing it. However, the results reported in [388] were impressive, even if the largest instance considered had only 318 cities. To see how it handles larger instances, Johnson and McGeoch [461] constructed a non-geometric implementation, which we tested. Running times as expected are non-competitive, growing at a rate somewhere between $\Theta(N^2)$ and

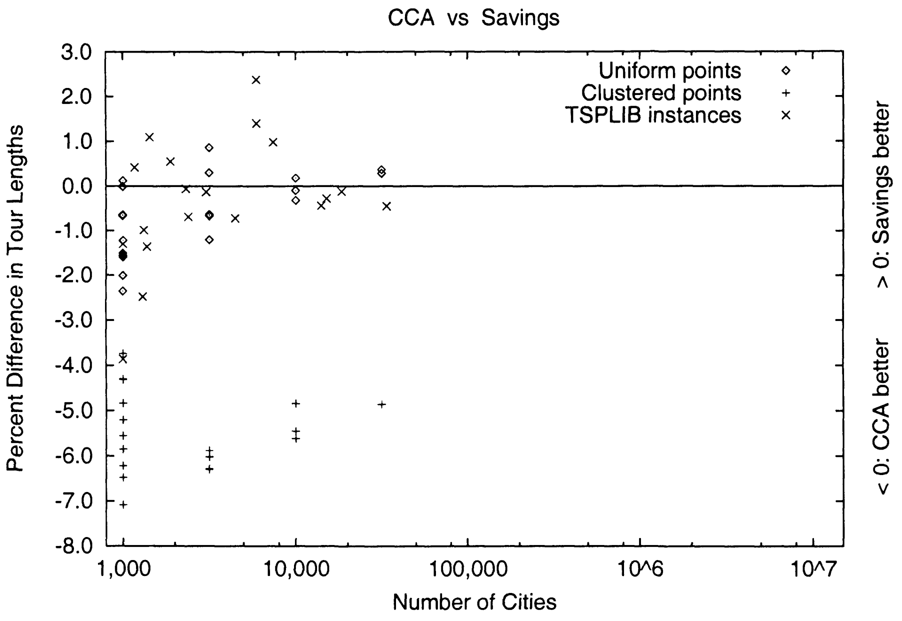


Figure 9.8. Tour quality comparisons for CCA and Savings heuristics.

$\Theta(N^{2.5})$ and taking over 3 normalized hours for $N = 33,810$ cities (the largest instance we tried) versus 0.44 seconds for Savings. However, as seen in Figure 9.8, CCA does find better tours for instances of all three types, even if its advantage for Uniform and TSPLIB instances seems to be vanishing as N grows. In particular, the limiting value for the average percentage excess on Uniform instances seems likely to exceed 12.5% (its value at $N = 31,623$), whereas that for Savings is 12.1%

The Christofides Heuristic and its Variants. For our final collection of tour construction heuristics, we consider variants on the famous heuristic of Christofides [189], which currently has the best worst-case guarantee known for any polynomial-time TSP heuristic, assuming only the triangle inequality.

The Christofides heuristic is a clever improvement on the Double Minimum Spanning tree (DMST) heuristic described earlier. In the standard version of Christofides (Christo-S), we start by computing a minimum spanning tree, as in DMST. However, instead of adding a second copy of the MST to get an Eulerian multigraph, here we add a minimum weight matching on the odd-degree vertices of the MST, which optimally grows the MST to an Eulerian multigraph. We then find an Euler tour and traverse it, shortcutting past previously visited vertices as in DMST. This leads to an improved guarantee: assuming the triangle inequality, the tour produced will never be more than 3/2 times the optimal tour length

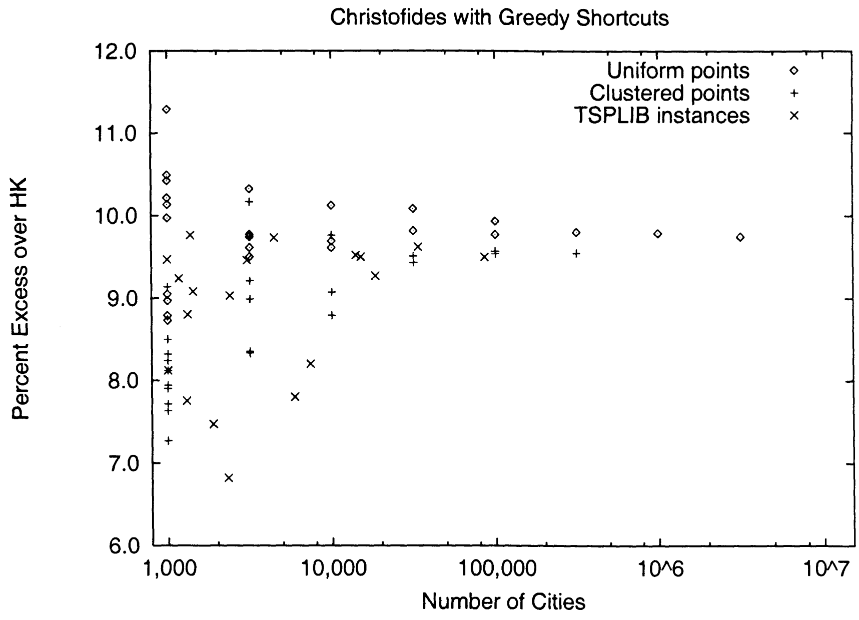


Figure 9.9. Tour quality for Christofides with greedy shortcuts.

(a bound that is asymptotically attainable by 2-dimensional Euclidean instances, as shown by Cornuejols and Nemhauser [222]).

Unfortunately, this improvement in worst-case behavior comes at a price. The running time for Christofides is dominated by that for computing the minimum weight matching, and the best algorithms known for this have $O(N^3)$ running times, as compared to the (non-geometric) worst-case running time of $O(N^2 \log N)$ for Savings. Fortunately, in practice we can use matching codes that exploit geometry to run much more quickly. For the implementations of Christofides studied here, we used the code of Cook and Rohe [207], together with a K-d tree based minimum spanning tree algorithm. With these, the observed running time appeared to be $O(N^{1.25})$ and we were able to handle instances with 3 million cities in normalized time of about an hour, only 10 times longer than for Savings. (Memory problems prevented us from successfully running the Christofides code on larger instances.)

The tour quality results for this standard version of Christofides are disappointing, however. Like Farthest Insertion it beats Savings on the Clustered instances, but it does worse for Uniform instances and most of the TSPLIB instances. Indeed, its average percentage excess for Uniform instances appears to approach 14.5%, which is worse than the limits for FA and FI as well as the 12.1% for Savings.

This is not the end of the story, however. A natural question is whether we might somehow do a better job of shortcutting in the final phase of the heuristic. As shown by Papadimitriou and Vazirani in [657], it is NP-hard to find the optimal way to shortcut the Euler tour. However, there are heuristics that do significantly better than the naive approach taken in the standard implementation. The "greedy shortcut" version of Christofides (`Christo-G`) examines the multiply visited cities in some arbitrary order and for each chooses the current best of the possible shortcuts. This version runs in essentially the same time as the standard one and yet finds better tours than both `Savings` and Farthest Insertion on all but one instance each (and on those two instances, it is only 0.06% worse). It is also more consistent. Figure 9.9 plots the percentage excess above the Held-Karp bound for `Christo-G` on all the integral-coordinate geometric instances in our testbeds (except the 10,000,000-city instance which we couldn't run). For all three classes `Christo-G`'s excesses for larger instances lie between 9 and 10%. The limiting percentage for Uniform instances appears to be about 9.8%, a substantial improvement over any of the other heuristics we have covered. `Christo-G` also outperforms `CCA` on all instances with more than 3,162 cities and performs better on average except in the case of Clustered instances with 1,000 or 3,162 cities (and is of course *much* faster).

Another modification of Christofides that has been proposed is to replace the initial MST with the one-tree obtained in the process of computing the Held-Karp bound using Lagrangean relaxation [444, 445]. This approach was combined with greedy shortcuts by Andre Rohe [729] in an implementation we shall call `Christo-HK`. The Lagrangean relaxation scheme used by Rohe involves many spanning tree computations over weighted sparse graphs derived from the Delaunay triangulation of the cities. Although no attempt is made to run this process to convergence, it still takes substantially longer than simply computing a single MST for the cities, so that asymptotically `Christo-HK` seems to be some 4-8 times slower than `Christo-G`. It does find significantly better tours, however: Its average excess over the HK bound appears to go toward 6.9% for Uniform instances and 8.6% for Clustered instances.

If one is unwilling to pay the running time penalty of `Christo-G` (much less that of `Christo-HK`), it is natural to ask how well one might do if one sped up the bottleneck matching phase of Christofides' algorithm by using a fast heuristic to get a good but not-necessarily-optimal matching (while still using greedy shortcuts). We have implemented such a heuristic using a K-d tree based greedy matching procedure followed by 2-opting, i.e., looking for pairs $\{a,b\}$, $\{c,d\}$ of matched cities for which changing partners (to $\{a,c\}$, $\{b,d\}$) shortens the matching, until

Average Percent Excess over the HK Bound: Uniform Instances

N =	1000	3162	10K	31K	100K	316K	1M	3M	10M
`RA`$^+$	13.96	15.25	15.04	15.49	15.43	15.42	15.48	15.47	15.50
`Chr-S`	14.48	14.61	14.81	14.67	14.70	14.49	14.59	14.51	–
`FI`	12.54	12.47	13.35	13.44	13.39	13.43	13.47	13.49	13.49
`CCA`	10.11	11.47	11.73	12.46	–	–	–	–	–
`Sav`	11.38	11.78	11.82	12.09	12.14	12.14	12.14	12.10	12.10
`ACh`	11.13	11.00	11.05	11.39	11.24	11.19	11.18	11.11	11.11
`Chr-G`	9.80	9.79	9.81	9.95	9.85	9.80	9.79	9.75	–
`Chr-HK`	7.55	7.33	7.30	6.74	6.86	6.90	6.79	–	–

Average Normalized Running Time in Seconds

`RA`$^+$	0.06	0.23	0.71	1.9	5.7	13	60	222	852
`Chr-S`	0.06	0.26	1.00	4.8	21.3	99	469	3636	–
`FI`	0.19	0.76	2.62	9.3	27.7	65	316	1301	5345
`CCA`	4.88	82.09	1129.85	14015	–	–	–	–	–
`Sav`	0.02	0.08	0.26	0.8	3.1	21	100	386	1604
`ACh`	0.03	0.12	0.44	1.3	3.8	28	134	477	2036
`Chr-G`	0.06	0.27	1.04	5.1	21.3	121	423	3326	–
`Chr-HK`	1.00	3.96	14.73	51.4	247.2	971	3060	–	–

Table 9.9. Results for the more powerful tour construction heuristics on Random Uniform Euclidean instances. `Sav`, `ACh`, and `Chr` stand for `Savings`, `AppChristo`, and `Christo`, respectively.

no more can be found. Our 2-opting procedure uses the speedup tricks for the 2-opt TSP heuristic described in the next section. The resulting "Approximate Christofides" heuristic (`AppChristo`) is from 2 to 7 or more times faster than `Christo-G`, with average tour lengths increasing between 1 and 2%, the higher figure being for Clustered instances. For Uniform instances, the limiting percentage excess for `AppChristo` appears to be 11.1%, compared to 12.1% for `Savings`, and `AppChristo` is typically only 1.2 to 3 times slower.

Table 9.9 summarizes the tour quality and running time results on Uniform instances for the best of the "more complex tour construction heuristics" of this section, with `Savings` included for comparison purposes. Times for similarly sized instances of the other two geometric classes are roughly the same except in the case of Clustered instances. `AppChristo` is typically almost twice as slow for such instances, while `Christo-S` and `Christo-G` are almost 3 times slower for the smaller ones, improving to 25-50% slower when $N = 316,228$.

3.4. Simple Local Search Heuristics

In this and the next three sections we cover various local search heuristics for the STSP, many of which are described in more detail in Chapter 8. In a local search heuristic for the TSP, one defines a neighborhood structure on the set of tours, where a tour T' is declared to be a *neighbor* of a tour T if it differs from it in some specified way. The classic neighborhoods of this type are the k-Opt neighborhoods, where T' is obtained from T by deleting k edges and replacing them with a different set of k edges (a *k-Opt move*). For $k > 2$, the sets need not be disjoint, so in particular a k-Opt move is a special case of a $(k+1)$-Opt move.

Given a neighborhood structure, a standard local search heuristic operates in two phases. First, it uses some tour construction heuristic to generate a *starting tour*. Then it repeatedly replaces its current tour by a neighboring tour of shorter length until no such tour can be found (either because none exists, in which case the tour is "locally optimal," or because the heuristic does not explore its neighborhoods exhaustively). A local search heuristic that uses the k-Opt neighborhood is usually called simply "k-Opt," and in this section we study various pure and restricted heuristics of this kind.

Currently, 2-Opt and 3-Opt are the main k-Opt heuristics used in practice, introduced respectively by Flood and Croes [314, 228] and by Bock [115]. In Shen Lin's influential 1965 study of 3-Opt [562], he concluded that the extra time required for 4-Opt was not worth the small improvement in tour quality it yielded, and no results have appeared since then to contradict this conclusion. In contrast, there have been several attempts to trade tour quality for improved running time in 3-Opt by exploiting restricted versions of the 3-Opt neighborhood, as in the *Or-Opt* heuristic of Or [635] and the 2.5-Opt heuristic of Bentley [103].

Implementation Details. Simply stating the neighborhood structure used does not completely specify a local search heuristic. In order to determine the tours generated by the heuristic one needs to provide such additional details as (a) the tour construction heuristic used, (b) the rule for choosing the improving move to make when there are more than one, and (c) the method used to look for improving moves (when the rule specified in (b) depends on the order in which moves are examined). Moreover, the heuristic's running time will depend on additional implementation details. Although naively one might expect 2-Opt and 3-Opt to require $\Omega(N^2)$ and $\Omega(N^3)$ time respectively, in practice they can be implemented to run much more quickly for geometric instances. 3-Opt can be implemented to run much more quickly than $\Omega(N^3)$ even for non-geometric instances. Although many factors are involved in these speedups, there are perhaps four key ones.

1. Avoiding Search Space Redundancy
2. Bounded Neighbor Lists
3. Don't-Look Bits
4. Tree-Based Tour Representation

The second and third of these trade a potential slight degradation in tour quality for improvements in running time. In particular, their use leaves open a slight possibility that the final tour may not be 2-Optimal (3-Optimal), i.e., it may have a better neighboring tour that we failed to notice. We now describe each of the four factors individually, as they are relevant not only the simple local search heuristics of this section but to the more sophisticated heuristics of later sections as well.

Avoiding Search Space Redundancy. We illustrate this in the context of 2-Opt. Each possible 2-Opt move can be viewed as corresponding to a 4-tuple of cities $\langle a, b, c, d\rangle$, where $\{a, b\}$ and $\{c, d\}$ are tour edges deleted and $\{a, c\}$ and $\{b, d\}$ are the edges that replace them. Suppose we intend to search through all the possibilities as follows: Let t_1 range over the N possibilities for a; given t_1, let t_2 range over the two possibilities for b; and given t_1 and t_2, let t_3 range over the possibilities for c (the choice of d is then forced). Note that, as stated, a given move would be examined four times, depending on whether $\langle t_1, t_2\rangle$ is $\langle a, b\rangle$, $\langle b, a\rangle$, $\langle c, d\rangle$, or $\langle d, c\rangle$. This redundancy can be exploited as follows: Never consider a city t for t_3 unless $d(t_1, t) < d(t_1, t_2)$. Note that if $\langle a, b, c, d\rangle$ is never examined under this regimen, we must have both $d(a, c) \geq d(a, b)$ and $d(b, d) \geq d(c, d)$, and so it cannot be an improving move. Hence no improving move will be missed. This restriction typically strongly limits the possibilities for t_3 as the heuristic proceeds. A generalization to 3-Opt limits choices for both t_3 and the analogous final choice t_5.

For geometric instances, this restriction can be implemented using K-d trees and a fixed-radius near neighbor search, as described in [103]. For non-geometric instances, one could simply precompute for each city an ordered list of the other cities by increasing distance. However, a quicker option is the following.

Bounded Neighbor Lists. Instead of creating for each city an ordered list of *all* the other cities, create a truncated list of the nearest k cities, ordered by increasing distance, on the assumption that more distant cities are unlikely to yield improving moves. In general, such lists can be computed in overall time $O(N^2 \log k)$. For geometric instances this can be reduced to something more like $O(N \log N)$ using K-d trees. A possibly more robust version of this approach is to include for each city c the $\lfloor k/4 \rfloor$ cities closest to c in each of the four quadrants of the coordinate system with c at (0,0). If these total fewer than k cities, we augment the set by the nearest remaining cities overall to bring the total up to k. This will be referred to in what follows as a *quad-neighbor* list. Another possibility for geometric instances, suggested by Reinelt [710, 711], is to construct a neighbor list from the cities closest to c in the Delaunay triangulation of the city set.

Don't-Look Bits. This idea was introduced by Bentley in [103] to help avoid the repetition of fruitless searches. Suppose our search for improving moves is as described above, with an outer loop that considers all N possible choices for t_1. Suppose we are considering the case where $t_1 = a$ and that (i) the last time we searched with $t_1 = a$, we didn't find an improving move and (ii) a has the same tour neighbors as it had that last time. Then it might seem unlikely that we will find an improving move this time either. Bentley proposed not searching in this case, being willing to risk the possibility that occasionally an improving move might be missed. In order to keep track of the cities for which searches could be skipped, he suggested maintaining an array of *Don't-Look* bits. Initially, the bits are all set to 0. Thereafter, the bit for a is set to 1 whenever a search with $t_1 = a$ is unsuccessful. Conversely, if an edge of the tour is deleted when an improving move is made, both its endpoints get their don't-look bits set back to 0. Note that as the local search procedure continues, the number of bits that are set to 0 will decline, so it may make sense to simply keep the cities with 0-bits in a queue, ordered by the length of time since they were last examined, rather than keeping an explicit array of don't-look bits. In this way we not only avoid searches, but spend no time at all considering cities that are to be skipped.

Tree-Based Tour Representation. Empirical measurements reported in [103, 322] suggest that for Uniform instances both 2-Opt and 3-Opt typically make $\Theta(N)$ improving moves. Bentley in [103] observed that as N increases, the time spent performing these moves came to dominate the overall time for his implementations. This was because of the way he represented the tour. A 2-Opt move basically involves cutting the tour in two places and reversing the order of one of the two resulting segments before putting them back together. If the tour is stored in a straightforward way (either as an array or a doubly linked list), this means that the time for performing the 2-Opt move must be at least proportional to the length of the shorter segment. Bentley's empirical data suggested that for Uniform instances the average length of this shorter segment was growing roughly as $N^{0.7}$, as was the average work for performing each move. If we consider alternative tree representations, this can be reduced to $\sqrt{N}$ using the 2-level trees of [322] or to $\log N$ using the splay tree data structure of [764]. For a study of the tradeoffs involved and the crossover points between various tour representations, see [322].

Results for 2-Opt, 2.5-Opt, and 3-Opt. Implementation choices can make a difference, both in tour quality and running time. We consider three sets of implementations.

- 2-Opt and 3-Opt implementations by Johnson and McGeoch (`-JM`).
- 2-Opt, 3-Opt and "2.5-Opt" implementations by Bentley (`-B`). The third heuristic is a restricted version of 3-Opt in which the 2-Opt neighborhood is augmented only by those 3-Opt moves that delete a single city from the tour and reinserted it elsewhere.
- 2-, 2.5-, and 3-Opt implementations by Applegate, Bixby, Chvátal, and Cook (`-ABCC`). These are included as options in the `edgegen` program of the `Concorde` software release.

The three sets of implementations are similar in that all exploit don't-look bits, but differ in many other respects. `Concorde`'s implementations use Nearest Neighbor to generate starting tours, whereas the Bentley and Johnson-McGeoch implementations both use the Greedy heuristic (although Johnson and McGeoch use a randomized variant that picks the shortest edge with probability 2/3 and the second shortest with probability 1/3). For all three heuristics `Concorde` considers only one of the two neighbors of t_1 as a choice for t_2 whereas the Bentley and Johnson-McGeoch implementations consider both. Another difference has to do with move selection. For each choice of t_1, the Johnson-McGeoch and `Concorde` implementations apply the first improving move found (except that in the `JM` implementations an improving 2-Opt move is not performed in 3-Opt unless no way is found to extend it to an even better 3-Opt move). In contrast, for each choice of t_1, the Bentley implementations keep looking for improving moves until it has seen 8 (or run out of possibilities) and then performs the best of these. The Bentley and `Concorde` implementations also have more chance of finding improving moves, since they use fixed-radius near-neighbor searches to find all possible candidates for t_3 (and t_5 in the case of 3-Opt), whereas the `JM` implementations restrict the choices to quad-neighbor lists of length 20. On the other hand, in the case of 3-Opt, Bentley's implementation examines fewer classes of potential 3-Opt moves, omitting for example those 3-Opt moves that permute but do not reverse any of the three

Algorithm	Percent Excess			Time (Seconds)		
	U	C	T	U	C	T
`Christo-G`	9.9	9.6	9.5	21.3	37.8	29.5
`Christo-HK`	6.9	8.4	7.4	247.2	197.0	177.9
`2opt-B`	5.7	9.6	5.8	8.8	9.7	5.6
`2opt-JM`	4.8	10.7	6.0	10.7	12.4	6.5
`2opt-ABCC`	14.4	19.6	14.7	3.7	2.9	1.9
`2.5opt-B`	4.7	8.2	4.8	10.2	12.0	7.5
`2.5opt-ABCC`	12.6	17.3	13.0	4.3	3.2	2.4
`3opt-B`	3.6	5.5	3.8	15.5	176.8	17.8
`3opt-JM`	3.0	6.9	4.2	12.3	14.9	6.9
`3opt-ABCC`	8.4	11.2	9.2	6.1	12.5	3.7

Table 9.10. Average percent excess over the HK bound and normalized running times for 100,000-city Uniform and Clustered instances and for `TSPLIB` instance `pla85900`. All codes were run on the same machine.

segments created when 3 tour edges are deleted. A final difference is that the Bentley and `Concorde` implementations represent the tour with an array whereas Johnson and McGeoch use the 2-level tree of [322].

Table 9.10 summarizes average heuristic performance of these implementations on the instances of approximately 100,000 cities in our three geometric classes. A first observation is that the `Concorde` (`-ABCC`) implementations are much faster and produce much worse tours than their `-B` and `-JM` counterparts. This is a reasonable tradeoff in the context of the intended use for the `Concorde` implementations, which is to quickly generate sets of good edges for use with other `Concorde` components. It does, however, illustrate the danger of `Concorde`'s restriction to one choice for city t_2, which is the primary cause for this tradeoff: Applying the same restriction to the `JM` implementations yields similar improvements in running time and degradations in tour quality.

As to comparisons between the Bentley and `JM` implementations, the latter produce better results for Uniform instances but are worse for Clustered instances. For these, Bentley's more complete examination of candidates for t_3 may be paying off, although there is a substantial running time penalty in the case of 3-Opt. For `TSPLIB` instances, the results are mixed, with the `JM` implementations more often producing better tours, although not for `pla85900` as shown in the table. Also not evident in the table is the running time penalty that the Bentley and `Concorde` implementations experience once $N > 100,000$, due to their use of the array representation for tours. Their observed running times have $\Omega(N^{1.5})$ growth rates, whereas the `JM` implementations, with their tree-based tour representations, have observed running times that appear to be $O(N \log^2 N)$. As a consequence, `2.5opt-B` is 5 times slower than the `JM` implementation of full 3-Opt when $N = 3,162,278$ and

	Percent Excess			Time (Seconds)		
Algorithm	1,000	3,162	10,000	1,000	3,162	10,000
NN-ABCC	224	321	337	0.8	9.7	112
Greedy-JM	163	198	250	0.7	8.8	107
2opt-JM	66	92	114	1.0	12.2	157
3opt-JM	31	43	63	1.1	12.3	150

Table 9.11. Average percent excesses over the HK bound and normalized running times for Random Matrix instances of sizes from 1,000 to 10,000 cities. All codes were run on the same machine.

2opt-ABCC and 3opt-ABCC are both 5-10 times slower than the corresponding JM implementations once $N = 1,000,000$.

Table 9.10 also addresses the question of how these simple local search heuristics compare to the best of the tour construction heuristics in our study: greedy-shortcut Christofides (Christo-G) and its Held-Karp-based variant Christo-HK. Based on the results in the table, it would appear that both are dominated by the 2.5opt-B and 3opt-JM, and the first is also dominated by 2opt-B. The situation is a bit more complicated, however, if one looks at the Challenge testbeds as a whole. Christo-G tends to produce better tours than 2opt-B for many Clustered instances and to be faster than all the Bentley and JM implementations for Uniform and TSPLIB instances with 10,000 or fewer cities. 2.5opt-B and 3opt-JM, however, produce better tours than Christo-G and Christo-HK for almost all the instances in the Challenge testbeds on which the latter two could be run. (3opt-JM loses only on two instances.) Since the running time advantage for Christo-G is never more than a factor of 3 on the smaller instances, all of which can be handled by the 3opt-JM in normalized time of less than 10 seconds (usually less than 2), and since the running time for Christo-HK is substantially worse than that for 3opt-JM across the board, there is probably no real reason to use either Christofides variant in practice, assuming one has a good implementation of 3-Opt.

The JM implementations of 2-Opt and 3-Opt can handle non-geometric instances, and so we also ran them on our Random Matrix testbed. The results are summarized in Table 9.11, which for comparison purposes also includes results for two tour construction implementations that can handle such instances: the benchmark Greedy code, which provides a good estimate for the lengths of the starting tours used by 2opt-JM and 3opt-JM, and Concorde's implementation of NN.

Observe that all the heuristics produce much poorer tours for Random Matrix instances than they do for geometric instances. Even the best of them, 3-Opt, has percentage excesses that are worse by a factor of 10 or more. Also note that tour quality declines substantially as N

increases. The results for all four heuristics are consistent with the conjecture that the average percentage excess grows as $\log N$, whereas for the geometric instances in our testbeds, all of our heuristics seem to have average percentage excesses that are bounded, independent of N. The running time for Random Matrices is also much worse, growing somewhat more rapidly than N^2, the nominal growth rate for simply reading the instance. An interesting side effect of this is that the time for performing local search becomes a much less significant component of the overall running time. `3opt-JM` takes only about 50% more time than `NN-ABCC` and the running time difference between `2opt-JM` and `3opt-JM` is inconsequential. (This is because most of the local search speedup tricks mentioned above continue to be applicable, so that the local search phase does not take much longer for Random Matrices than it did for geometric instances.) Given how much better `3opt-JM`'s tours are than those of the other heuristics, it is the obvious choice among the four, should one want to solve this kind of instance.

More on Starting Tours and Neighbor Lists. As noted above, all the Bentley and the `JM` implementations used the Greedy heuristic to generate starting tours. This decision was based on the extensive experiments reported in [103, 461], which showed that `Greedy` tours tended to yield the best results, both in comparison to worse tour construction heuristics such as `NN` or the infamous "generate a random tour" heuristic, and to better ones such as `FI` or `Savings`. It appears that the starting tour needs to have some obvious defects if a simple local search heuristic is to find a way to make major improvements, but it can't be too bad or else the heuristic will not be able to make up the full difference. Random starting tours have the additional disadvantage that they lead to increased running times because more moves need to be made to reach local optimality.

The `JM` implementations for which results were reported above used neighbor lists of length 20. Many authors have suggested using substantially shorter lists, but at least in the context of these implementations, 20 seems a reasonable compromise. Using lists of length 10 saves only 10-20% in running time but on average causes tour lengths to increase by 1% or more for both `2opt-JM` and `3opt-JM` and all four instance classes. Increasing the list length to 40 increases running time by 40-50% and for `3opt-JM` on average improves tour length by less than 0.3% on all but the Clustered instances. The average improvements for Clustered instances are more variable, but appear to average roughly 0.6% overall. For `2opt-JM` the tour length improvements due to increasing the neighbor list length to 40 are slightly larger, but the running time becomes greater than that for 3opt-JM with 20 neighbors, and the latter finds much better tours. For full details, see the Challenge website.

Other Simple Local Search Heuristics. 2.5-Opt is not the only restricted version of a k-Opt heuristic that has been seriously studied. Much early attention was devoted to the Or-Opt heuristic of [635], which uses a neighborhood intermediate between that of 2.5-Opt and full 3-Opt. In 2.5-Opt, we consider those 3-Opt moves in which one of the three segments into which the tour is initially broken contains just one city; Or-Opt expands this to segments of as many as 3 cities. It was originally proposed as a way of reducing the running time overhead of the naive $\Omega(N^3)$ implementation of 3-Opt, before the existence of the speedup tricks mentioned above was widely known. Although these tricks should be adaptable to Or-Opt as well, it seems unlikely that the latter would retain much speed advantage over a more complete 3-Opt implementation. Thus the probable tour degradation due to the much smaller Or-Opt neighborhood is not likely to be justified, and Or-Opt no longer appears to be a serious competitor. No implementations were submitted to the Challenge.

Researchers have recently also considered putting restrictions on the k-Opt neighborhood when $k \geq 4$, with the intent of getting better tours than 3-Opt without paying the full running time penalty for k-Opt itself. Two families of heuristics of this type were submitted to the Challenge: the GENI/GENIUS heuristics of Gendreau, Hertz, and Laporte [351] and the HyperOpt heuristics of Burke, Cowling, and Keuthen [149].

From one point of view GENI can be viewed as a tour construction heuristic, in that the tour is augmented one city at a time. However, each augmentation is equivalent to a simple insertion followed by a 4- or 5-Opt move, so we have chosen to consider it here in the section on local search. GENIUS uses GENI to construct its starting tour, and then attempts to improve it by a "stringing-unstringing" procedure that technically is a restricted version of 8-, 9-, or 10-Opt. In addition, the heuristics use truncated nearest neighbor lists to restrict their choices, with the heuristic being parameterized by the length p of these lists.

We tested implementations of GENI and GENIUS provided to us by Gendreau et al., which we fine-tuned by improving their handling of memory allocation and by removing some redundant operations. This fine-tuning did not change the output tours but does result in substantial running-time improvements. The implementations still do not, however, exploit the full set of speedup tricks listed above, and with $p = 10$ GENI is 100 times slower for 10,000-city instances than 3opt-JM (on the same machine), and GENIUS is over 300 time slower. Moreover, although both GENI and GENIUS find better tours than 3-Opt for Clustered instances, they are worse for Uniform instances and most TSPLIB instances. Increasing p to 20 does not yield a significant improvement in the tours but causes substantial increases in running time. It is possible that a reimplementation of the heuristics to take advantage of more speedup tricks might make them more competitive timewise. It seems unlikely, however, that we could speed up the heuristics sufficiently to make them competitive with implementations of Lin-Kernighan to be described in the next section, and those find better tours across the board. (Based on comparisons to GENIUS in [713], the I^3 heuristic of Renaud, Boctor, and Laporte [713], which uses a restricted 4-Opt neighborhood, would not seem to be competitive either.)

The HyperOpt heuristics of [149] also come in parameterized form, with the the neighborhood structure for k-HyperOpt being a restricted version of the $2k$-Opt neighborhood. To construct a neighboring tour, one first deletes two disjoint sets of k consecutive tour edges. This breaks the tour into two subtours and $2(k-1)$ isolated cities. These are then recombined optimally into a tour using dynamic programming. The implementations of Burke et al. are substantially faster than those for GENI/GENIUS, even taking into account possible normalization errors. The normalized times for 2-HyperOpt are comparable to those for the 3opt-JM until they start to degrade around $N = 100,000$. Unfortunately, 2-HyperOpt's average tour quality is worse than that of 3opt-JM for all three geometric instance classes. 4-HyperOpt might be slightly better than 3opt-JM on Clustered instances, but it is a close call, and 3opt-JM is 50 times faster.

Thus as of now it does not appear that restricted k-Opt heuristics, $k \geq 4$, offer a promising avenue to cost-effective improvements on 3-Opt. As was first shown in 1973 by Lin and Kernighan [563], a much better generalization is the concept of *variable-depth* search. We cover heuristics based on this concept in the next section.

3.5. Lin-Kernighan and Variants

The Lin-Kernighan heuristic of [563] does not limit its search to moves that change only a bounded number of edges. In principle it can change almost all the edges in the tour in a single move. However, the moves have a specific structure: each can be viewed as a 3-Opt move followed by a sequence of 2-opt moves, although only the full final move need actually improve the tour. The heuristic keeps running time under control by restricting the "LK-Search" to moves that are grown one 2-Opt move at a time, without backtracking beyond a fixed level. In addition, it uses neighbor lists to restrict the number of growth alternatives it considers, locks edges that have already been changed by the move so that the changes won't subsequently be undone, and aborts the search if a strict gain criterion based on the best tour seen so far is not met. More details can be found in Chapter 8.

In implementing Lin-Kernighan, one has far more choices to make than for simple heuristics like 2-Opt and 3-Opt, and the literature contains reports on many implementations of Lin-Kernighan with widely varying behavior. In addition to such standard choices as (a) what tour construction heuristic to use, (b) when to continue looking for better improving moves after one has been found, (c) how long neighbor lists should be and how they should be constituted, (d) what tour representation should be used, and (e) whether to use don't look-bits, here are some of the new choices:

- How broad should the search be? The original Lin-Kernighan implementation restricted attention to neighbor lists of length 5 at all levels of the search.
- How much backtracking is allowed? The original Lin-Kernighan implementation considered all 3-Opt moves that met the gain criterion (together with a restricted class of 4-Opt moves) as starting points for the LK-search.
- Should one consider just one or the standard two choices for t_2?
- Does one lock both deleted and added edges, or only one of the two classes? (The original LK implementation locked both, but locking either of the two classes separately is enough to insure that the search runs in polynomial time.)
- Does one tighten the gain criterion in the middle of the LK-search if a better tour is discovered along the way?
- Should one extend the partial move that yields the best tour, or the one that has the best gain criterion?
- Should one impose a constant bound on the depth of the LK-search?
- Should one augment the moves constructed by LK-search with non-sequential "double-bridge" 4-Opt moves, as proposed by Lin and Kernighan (but not used in their own implementation)?

Moreover, one can consider topological variants on the way that moves are grown and the reference structure maintained at each level of the search. Lin and Kernighan's reference structure is a path with one end fixed, and the changes considered in growing the move all consist of adding an edge from the unfixed end of the path to a city on its neighbor list, and then deleting the one edge that will take us back to a path. Other possible reference structures have been implemented, such as the unanchored path of Mak and Morton [576] or the "stem-and-cycle" of Glover [373, 374]. In addition, alternative ways of extending an LK-search (besides the standard 2-Opt move) have been considered, as in the variant due to Helsgaun [446] that augments via 5-Opt moves.

We do not have space to go into full detail on how all the tested implementations differ. Indeed, many implementers' written descriptions of their implementations do not provide all the answers. So we concentrate in what follows on key differences (and similarities) and our conclusions about the effects of various choices are necessarily tentative.

Basic Lin-Kernighan. We start with implementations that do not depart in major ways from the original heuristic, i.e., implementations that use a path as reference structure, use 2-Opt moves as the augmentation method in LK-search, and do not perform double-bridge moves. Four such implementations were submitted to the Challenge:

1. `LK-JM` (Johnson and McGeoch [461, 463]). The main results reported here for this implementation use `Greedy` starting tours, length-20 quad-neighbor lists for all levels of the search, don't-look bits, and the 2-Level Tree tour representation [322]. In the LK-search, this `C` implementation uses an anchored path as its reference structure, locks only the added edges, updates the gain criterion when a better tour is found in mid-search, does not bound the depth of the searches, and otherwise follows the original Lin-Kernighan `Fortran` code, from which it is derived.

2. `LK-Neto` (Neto [626]). This implementation is based on the original Lin-Kernighan paper [563] and on the Johnson-McGeoch chapter [463]. It differs from the Johnson-McGeoch implementation in that neighbor lists consist of 20 quadrant neighbors *unioned* with the 20 nearest neighbors, LK-searches are bounded at 50 moves, and special *cluster compensation* routines are used with the hopes of improving performance for instances in which the cities are grouped in widely separated clusters, presumably as in our Clustered instances. Source code for this implementation (in `CWEB`) is available from `http://www.cs.toronto.edu/~neto/research/lk/`.

3. `LK-ABCC` (Applegate, Bixby, Chvátal, and Cook [27]). This is the default Lin-Kernighan implementation in `Concorde`. Based on remarks in [27], it would appear that this implementation differs

from the Johnson-McGeoch implementation mainly as follows: It uses Q-Boruvka starting tours, length-12 quad-neighbor lists, an unanchored path as reference structure, a narrower (but slightly deeper) backtracking strategy with just one choice for t_2, and LK-searches bounded at 50 moves. Source code is available from the Concorde website.

4. LK-ACR (Applegate, Cook, and Rohe [32]). Based on remarks in [27, 32], it would appear that this implementation differs from Concorde's in that it uses a slightly broader and deeper backtracking strategy and bounds the depth of the LK-search at 25 moves.

Both of the latter two implementations were optimized for use in the context of Chained Lin-Kernighan, where the speed of a single invocation of LK may be more important than saving the last few fractions of a percent in tour length. Results confirm this. See Table 9.12 which presents average tour qualities and running times for the four implementations. For comparison purposes, the corresponding results for 3opt-JM are also included.

Note that LK-JM and LK-Neto provide roughly equivalent tours except for the Clustered instances. These tours are typically substantially better than those for LK-ABCC and LK-ACR, although LK-ABCC and LK-ACR are significantly faster for Clustered and TSPLIB instances and for Uniform instances when $N \leq 10,000$. As in the case of the ABCC implementations of 2-, 2.5-, and 3-Opt, a major reason for this disparity is probably the restriction in LK-ABCC and LK-ACR to a single choice for t_2. A second but lesser reason is the fact that the latter two heuristics construct shorter neighbor lists and do so more quickly. The fact that they bound the depth of the LK-search would not seem to be a major factor, however, since LK-Neto also bounds the depth. Moreover, if one imposes a depth bound of 50 on LK-JM, neither tour quality nor running time is typically affected significantly. As to asymptotic growth rates, the observed running times for LK-JM and LK-ACR appear to be $O(N^{1.25})$, while those for LK-Neto and LK-ABCC may be somewhat worse.

Between LK-ABCC and LK-ACR, the former tends to yield slightly better tours and take slightly longer, but the results for both are closer to those for 3opt-JM than to those for LK-JM. Indeed, for the three smallest TSPLIB sizes, 3opt-JM on average finds better tours than LK-ACR. This becomes less surprising when we look at running times, since LK-ACR actually appears (subject to normalizing errors) to use less time than 3opt-JM for these instances. LK-ABCC is almost as fast.

Average Percent Excess over the HK Bound

N =		1000	3162	10K	31K	100K	316K	1M	3M	10M
Random Uniform Euclidean Instances										
LK:	JM	1.92	1.99	2.02	2.02	1.97	1.96	1.96	1.92	–
	Neto	1.91	1.97	1.99	1.89	1.95	1.97	1.92	1.88	–
	ABCC	2.22	2.43	2.60	2.48	2.54	2.67	2.68	2.55	2.54
	ACR	2.36	2.90	2.72	2.73	2.74	2.75	2.77	2.67	2.49
3opt:	JM	2.96	2.84	3.06	3.02	2.97	2.93	2.96	2.88	–
Random Clustered Euclidean Instances										
LK:	JM	1.75	2.95	3.41	3.71	3.63	3.67			
	Neto	2.52	4.19	4.76	4.42	4.78	–			
	ABCC	3.77	6.23	5.70	6.38	5.31	5.45			
	ACR	3.89	6.13	5.93	6.28	5.54	5.54			
3opt:	JM	4.08	6.06	6.89	7.48	6.88	7.08			
TSPLIB Instances										
LK:	JM	2.38	2.16	1.92	1.73	1.61				
	Neto	2.40	2.32	1.88	2.00	–				
	ABCC	3.54	3.29	2.39	2.16	1.60				
	ACR	4.48	3.48	3.72	2.91	2.40				
3opt:	JM	3.93	3.67	3.17	3.99	4.20				

Average Normalized Running Time in Seconds

Random Uniform Euclidean Instances										
LK:	JM	0.20	0.69	2.32	7.2	22.8	61	323	1255	–
	Neto	0.19	0.87	3.35	14.4	89.6	574	3578	17660	–
	ABCC	0.09	0.34	1.49	6.0	21.4	61	307	1330	6980
	ACR	0.07	0.29	0.93	3.0	16.4	76	318	1290	5760
3opt:	JM	0.13	0.45	1.44	4.2	12.3	33	162	600	–
Random Clustered Euclidean Instances										
LK:	JM	1.66	4.97	15.37	59.3	173.1	495			
	Neto	4.35	15.04	51.17	138.6	558.1	–			
	ABCC	0.19	0.72	2.55	11.0	37.9	108			
	ACR	0.10	0.45	1.40	4.5	25.0	114			
3opt:	JM	0.15	0.54	1.77	5.0	14.9	38			
TSPLIB Instances										
LK:	JM	0.34	0.64	4.29	13.0	24.3				
	Neto	0.40	1.08	10.26	47.1	–				
	ABCC	0.10	0.29	1.21	3.5	8.8				
	ACR	0.08	0.23	0.74	1.7	5.4				
3opt:	JM	0.14	0.38	1.42	3.4	6.9				

Table 9.12. Results for 3-Opt and four implementations of Lin-Kernighan. Averages for TSPLIB are taken over the same instances as in Figure 9.3.

Turning now to the relationship between LK-JM and LK-Neto, a first observation is that the algorithmic descriptions in [563, 463] (on which the LK-Neto implementation was based) seem to be adequate to define a reproducible Lin-Kernighan implementation, at least as far as tour quality on Uniform and most TSPLIB instances is concerned. The two implementations do differ significantly on Clustered instances, but this is presumably because Neto has added innovations designed to handle clustered instances more effectively. Unfortunately, for this particular class of Clustered instances, the innovations do not appear to be effective. LK-Neto provides significantly worse tours than LK-JM and the ratio of its normalized running time to that of LK-JM is higher for Clustered instances than it is for Uniform instances. There also appears to be some as-yet-unidentified difference in the two implementations that has an asymptotic effect on running times: For Uniform instances the normalized running times for LK-Neto start to diverge from those for LK-JM when N gets large.

All the above LK implementations use variants on the Greedy heuristic to generate starting tours and so do not provide much of an opportunity to analyze the effects of different starting heuristics on the final LK results. (Analyses of this sort can be found in [32, 461].) However, one question is hard to resist: What would happen if we combined Lin-Kernighan with the very best (and slowest) of our tour generation heuristics, Christo-HK? Andre Rohe submitted results for just such a combination, and although his Lin-Kernighan implementation was an early one that lacks the detailed tuning of LK-ABCC and LK-ACR, the results are intriguing. For $N > 1,000$ the total running time is dominated by that simply to generate the starting tour and for Uniform and TSPLIB instances averages from 3 to 10 times slower than that for LK-JM. However, the final tours are significantly better. For Uniform instances the limiting ratio to the HK bound appears to be about 1.5% versus the 1.9% for LK-JM and the average improvement over LK-JM on TSPLIB instances ranges from 0.2% to 0.5%. For the larger Clustered instances the improvement over LK-JM is roughly 1.3% and there is essentially *no* running time penalty. Although even bigger improvements are possible using the repeated local search heuristics of the next section, those heuristics typically take even more time. Thus combining Lin-Kernighan with Christo-HK could sometimes be an appealing option.

To complete the Lin-Kernighan picture, Table 9.13 covers Random Matrix instances. Since the results for LK-Neto do not cover the 10,000-city instance, we also include a version of LK-JM with LK-search depth bounded at 50 (LK-JM-BD). (Note that because the triangle inequality does not hold, Christo-HK starting tours are no longer relevant.)

Algorithm		Percent Excess			Time (Seconds)		
		1,000	3,162	10,000	1,000	3,162	10,000
LK:	JM	3.5	4.4	5.9	1.2	12.8	161
	JM-BD	3.5	4.8	6.2	1.1	12.4	154
	Neto	3.0	4.1	–	1.2	16.9	–
	ABCC	4.0	6.0	9.0	1.1	12.6	151
	ACR	5.3	7.6	10.3	0.3	3.6	32
3opt:	JM	31.2	42.6	62.7	1.1	12.3	150

Table 9.13. Average percent excesses over the HK bound and normalized running times for Random Matrix instances of sizes from 1,000 to 10,000 cities.

Once again LK-ACR is faster than 3opt-JM (in this case significantly so), but now it also finds much better tours. Its tours are however worse than those for LK-ABCC, and both produce significantly worse tours than LK-JM and LK-Neto. Once again it is not clear if any of this difference can be attributed to bounding the depth of the LK-search. Indeed, LK-Neto, with bounding, finds significantly better tours than the LK-JM, with or without bounding (for reasons unknown to us and in contrast to the results for the geometric case). Running times for all the implementations except LK-ACR are comparable, being dominated by the time to read and preprocess the instance. As to the difficulty of this instance class in general, note that for all the implementations, the percentage above the Held-Karp bound again appears to be growing with N (possibly at a $\log N$ rate), something that doesn't happen for the geometric classes.

We now turn to variants on basic Lin-Kernighan that involve more substantial changes to the basic design of the heuristic.

Stem-and-Cycle Variants. These variants differ primarily in their choice of reference structure for the LK-search. Here the structure consists of a cycle with a path connected to one of its vertices, as described in more detail in Chapter 8. The implementation for which we report results is due to Rego, Glover, and Gamboa and is based on the heuristic described in [704]. It makes use of the 2-Level Tree tour representation, but does not use don't-look bits. (As we shall see, this probably exacts a severe running time penalty.) Variants using random starting tours (SCLK-R) or Boruvka starting tours (SCLK-B) yield roughly comparable tours, although the latter is usually at least twice as fast.

Figure 9.10 shows the relative tour-quality performance on our geometric testbed for SCLK-B and LK-JM. Note that the two are consistently close, usually well within 1% of each other. Neither implementation has tours that are consistently better than the other's, although LK-JM wins more often than it loses. The normalized running time for SCLK-B ranges from 4 times slower than LK-JM on 1,000-city Clustered instances

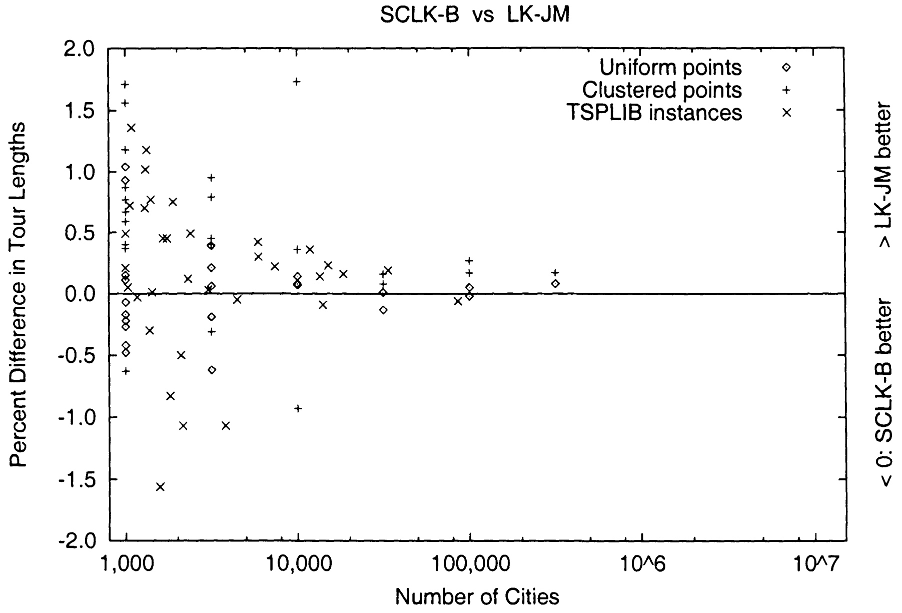


Figure 9.10. Tour quality comparisons between the Rego-Glover-Gamboa implementation of the Stem-and-Cycle variant of Lin-Kernighan with Boruvka starting tours and the Johnson-McGeoch implementation of basic Lin-Kernighan.

to 1300 times slower on the 316,228-city Uniform instance (the largest on which SCLK-B was run). It is possible that incorporating don't-look bits into the implementation will make up much of this gap, but for now there does not seem to be any significant advantage to this approach over basic Lin-Kernighan.

The Helsgaun Variant (Helsgaun). This variant, described in [446], offers several major innovations. First, the augmentation step in the LK-search is not a 2-Opt move but a sequential 5-Opt move. To keep running time under control, search at all levels is limited to length-5 neighbor lists. These are constructed in an innovative fashion.

We begin as if we were going to compute an estimate of the Held-Karp bound using the Lagrangean relaxation approach of [444, 445] augmented with techniques from [230, 442]. This yields a vector of "π-values" $(\pi_1, \ldots, \pi_N)$ such that the minimum one-tree (spanning tree plus an edge) under the distance function $d_\pi(c_i, c_j) = d(c_i, c_j) + \pi_i + \pi_j$ is a close lower bound on the Held-Karp bound. Based on this new distance function, the "α-value" $\alpha(i, j)$ for each edge $\{c_i, c_j\}$ is defined to be the difference between the length of the minimum one-tree (under d_π) that is required to contain $\{c_i, c_j\}$ and the length of the minimum unconstrained one tree (under d_π). Given the vector of π-values, the

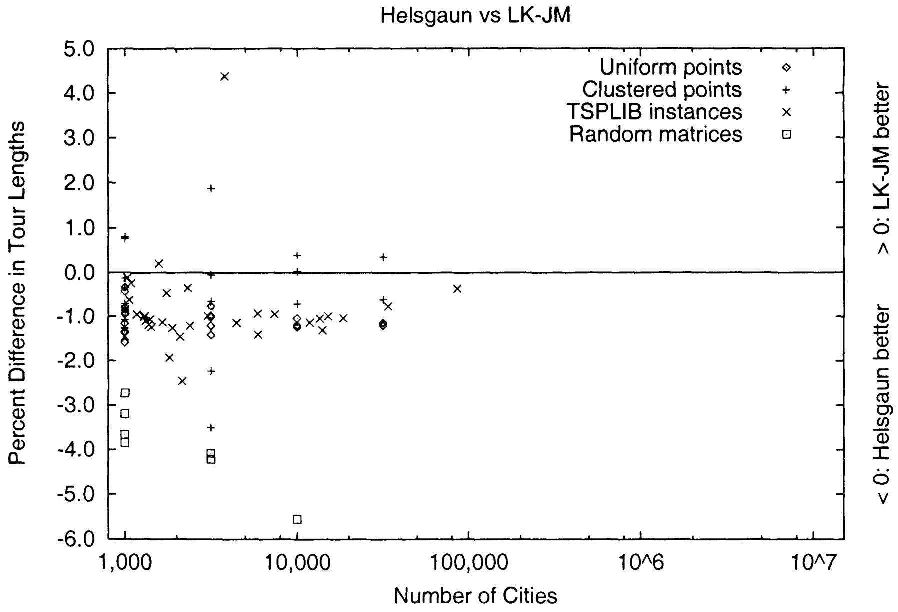


Figure 9.11. Tour quality comparisons between the Helsgaun variant on Lin-Kernighan and the Johnson-McGeoch implementation of basic Lin-Kernighan.

α-values for all edges can be computed in linear space and $O(N^2)$ time [446]. The neighbor list for c_i then consists of the 5 cities c_j with smallest values of $\alpha(i,j)$. These are initially ordered by increasing α-value, but are subsequently dynamically reordered to give priority to edges shared by the current best tour and its predecessor.

In addition, the implementation alternates between searching its "LK-search-with-sequential-5-Opt-augmentations" neighborhood and searching a second neighborhood defined by a set of non-sequential moves that includes not only double-bridge 4-Opt moves but also some specially structured 5-Opt moves as well. Given the power of its search strategy, the implementation needs only to backtrack at the first (t_1) level. For its starting tours it uses a special heuristic that exploits the α-values. The reader interested in this and the other details of the heuristic is referred to Helsgaun's paper [446]. The C source code for `Helsgaun` is currently available from `http://www.dat.ruc.dk/~keld/`.

The tour-quality results for this variant are impressive. See Figure 9.11, which compares `Helsgaun` to `LK-JM`. Note that `Helsgaun` finds better tours for a large majority of the instances in our testbed, usually better by 1% or more for Uniform and TSPLIB instances, a large amount in the context of the differences between LK-variants. The difference is even greater for our Random Matrix instances. The detailed

Average Percent Excess over the HK Bound

	N =	1000	3162	10K	31K	100K	316K	1M	3M	10M
U	LK-JM	1.92	1.99	2.02	2.02	1.97	1.96	1.96	1.92	–
	Helsgaun	0.90	0.89	0.83	0.83	–	–	–	–	–
C	LK-JM	1.75	2.95	3.41	3.71	3.63	3.67			
	Helsgaun	1.25	2.00	3.32	3.58	–	–			
T	LK-JM	2.38	2.16	1.92	1.73	1.61				
	Helsgaun	1.20	1.01	0.77	0.96	1.25				
M	LK-JM	3.52	4.35	5.91						
	Helsgaun	0.04	0.02	0.03						

Average Normalized Running Time in Seconds

U	LK-JM	0.20	0.7	2	7	23	61	323	1255	–
	Helsgaun	5.64	71.5	862	7820	–	–	–	–	–
C	LK-JM	1.66	5.0	15	59	173	495			
	Helsgaun	7.02	70.3	768	12800	–	–			
T	LK-JM	0.34	0.6	4	13	24				
	Helsgaun	7.82	73.3	1060	7980	48200				
M	LK-JM	1.15	12.8	161						
	Helsgaun	7.78	100.6	1270						

Table 9.14. Results for the Helsgaun variant on Lin-Kernighan as compared to those for the Johnson-McGeoch implementation of basic Lin-Kernighan.

averages are shown in Table 9.14, which indicates that for Random Matrices, Helsgaun finds tours that are essentially optimal. The table also reports on average running times, which show that Helsgaun pays a significant running time penalty for its improved tour quality. In some cases Helsgaun takes as much as 1,000 times longer than LK-JM, and its running time growth rate appears to be $\Omega(N^2)$ versus the observed $O(N^{1.25})$ rate for LK-JM. In certain applications such a price may be worth paying of course. The most active and interesting research on STSP heuristics today concerns how best to use a large amount of computation time to help narrow the small gap above optimal tour length left by Lin-Kernighan. In the next section we consider other ways of buying better tours by spending more time.

3.6. Repeated Local Search Heuristics

One way to invest extra computation time is to exploit the fact that many local improvement heuristics have random components, even if only in their initial tour construction. Thus if one runs the heuristic multiple times one will get different results and can take the best. Unfortunately, as noted by many authors and aptly illustrated in [463], the value of the straightforward repeated-run approach diminishes as instance size increases.

A much more effective way to use repeated runs has received wide attention in the last decade. The idea is basically this: Instead of using independently generated starting tours, derive the starting tours by perturbations of the output tours of previous runs. This idea was suggested by Baum in 1986 [93] and is in a sense implicit in the operation of the Tabu Search approach of Glover [369, 370]. However, its first effective realization in the context of the TSP is due to Martin, Otto, and Felten, who in [587, 588] proposed generating a new starting tour by perturbing the current best tour with a "kick" consisting of a random double-bridge 4-Opt move. Their original implementation used 3-Opt as the local search engine, but this was quickly upgraded to Lin-Kernighan in their own work and that of Johnson [460].

If the underlying Lin-Kernighan (or 3-Opt) variant uses don't-look bits, this approach has an additional advantage. At the end of a run of Lin-Kernighan, all the don't-look bits will typically be on since no improving moves have been found for any choice of t_1. Performing one 4-Opt move to get a new starting tour only changes the tour neighbors for at most 8 cities. This suggests that, instead of starting with all the don't-look bits off, as in a stand-alone run, we might want to start with the don't-look bits on for all but these 8 or fewer cities, i.e., we place only these cities in the initial priority queue of candidates for t_1. In practice, this can lead to sublinear time per iteration if other data structures are handled appropriately, which more than makes up for any loss in the effectiveness of individual iterations.

Martin, Otto, and Felten referred to their approach as "chained local optimization" since one could view the underlying process as a Markov chain. They also attempted to bias their choice of 4-Opt move toward better than average moves, and incorporated a fixed-temperature simulated annealing component into the heuristic. Results for five descendants of this approach were submitted to the Challenge, differing both in the underlying local search heuristic used and in the methods used for generating the double-bridge kicks. (Variants using more elaborate kicks have been studied [205, 452], but none have yet proved competitive on large instances with the best of the heuristics presented here.) Together, these five include all the current top performers known to the authors for this region of the time/quality tradeoff space. None of the implementations use simulated annealing, and the first three call themselves "Iterated" 3-Opt/Lin-Kernighan to mark this fact.

1. **Iterated 3-Opt (Johnson-McGeoch)** (`I3opt`) [463]. This uses `3opt-JM` and random double-bridge kicks.

2. **Iterated Lin-Kernighan (Johnson-McGeoch)** (`ILK-JM`) [463]. This uses `LK-JM-BD` and random double-bridge kicks.

3. **Iterated Lin-Kernighan (Neto)** (`ILK-Neto`) [626]. This uses `LK-Neto` with random double-bridge kicks.

4. **Chained Lin-Kernighan (Applegate-Bixby-Chvátal-Cook)** (`CLK-ABCC`). This uses `LK-ABCC` and the sophisticated method for generating promising kicks described in [27].

5. **Chained Lin-Kernighan (Applegate-Cook-Rohe)** (`CLK-ACR`) [32]. This uses `LK-ACR` together with a new method of generated biased kicks based on random walks, with the number of steps in the random walk doubling after the Nth iteration.

Given the huge running times for Helsgaun's variant on Lin-Kernighan, the above variants on Chained Lin-Kernighan can all perform many iterations and still not take as much time as Helsgaun does. However, if one has time to spare, one can also perform the latter repeatedly. For this, Helsgaun [446] has devised a different way of exploiting information about previous runs. Instead of using a kick to perturb his current champion into a new starting tour, he uses his standard tour construction heuristic to generate the new starting tour, but biases its choices based on the edges present in the current champion tour. He gets a per-iteration speedup because he doesn't have to recompute the π-vector and the α-values after the first iteration, although his running times remain substantial. We shall refer to this heuristic as `Helsgaun-`k, where k is the number of iterations.

Tables 9.15 and 9.16 summarize results for the various repeated-run heuristics described above. Results for `ILK-Neto` are omitted because of its similarity to the `ILK-JM` implementation and the fact that the results reported for it were less complete. As with the corresponding base implementations, `ILK-JM` and `ILK-Neto` seem to produce similar results, except that `ILK-Neto` is somewhat slower and does better on Random Matrices. The tables are divided into four sections, one for each class of instances in the Challenge testbed. Within each section, heuristics are ordered and grouped together according to the tour quality they provide. The grouping is somewhat subjective, but attempts to reflect performance over all instance sizes within a class. Thus for example for Clustered instances (`C`) we group `CLK-ABCC-N` and `I3opt-10N` together even though the latter is much better for small N, because it is worse for large N. These groupings are carried over from the tour quality table to the running time table, so that the most cost-effective heuristic in each group can be identified.

The performance of a repeated-run heuristic naturally depends on how many iterations are performed. In practice one may simply run for as many iterations as can be handled within some fixed time limit or until a satisfactory solution is reached, and several of these implementations were designed with this usage in mind. For a scientific study, however, one needs results that are more readily reproducible, and hence a combinatorial stopping criterion is to be preferred. Here we bound the number of iterations by a function of the number N of cities. Typical bounds are $N/10$, N, or $10N$, as indicated by the suffix on the heuristic's name in the tables. For comparison purposes, the tables also include the results for both `LK-JM` and `Helsgaun`.

A first observation is that none of the heuristics in the table is consistently dominated by any other in both tour quality and running time. However, if running time is no object, then the the iterated versions of Helsgaun's heuristic appear to be the way to go. Although Table 9.15 does not include rows for the optimal solution quality (which itself has a gap above the Held-Karp bound), the row for `Helsgaun-N` serves that purpose fairly well. The average excess for the optimal solution is known for all testbed instances with 3,162 or fewer cities, and all `TSPLIB` and Random Matrix instances with 10,000 cities or fewer. `Helsgaun-N`'s average excess is within 0.01% of this value in all these cases except for the "1,000-city" `TSPLIB` instances, for which it is 0.04% above the optimal excess. Moreover, for no instance in the Challenge testbed with a known optimal solution is the `Helsgaun-N` tour more than 0.18% longer than optimum. Much of this quality is retained by the $N/10$-iteration version, which uses substantially less time and can feasibly be applied to larger instances (although for instances with 10,000 or fewer cities even `Helsgaun-N` never takes more than four hours of normalized running time, which should be feasible in many applications). Both variants have running time growth rates that appear to be $\Omega(N^{2.5})$ or worse, however, and so computers will have to be a lot faster before they can be applied to million-city instances.

The basic Helsgaun heuristic is itself a strong performer in three of the four classes, although it does fall down seriously on Clustered instances and is probably not cost-effective compared to `CLK-ACR-N` on Uniform and `TSPLIB` instances. For Random Matrix instances, none of the other heuristics come close to `Helsgaun` and its iterated versions, and their running times are moreover quite reasonable. If there were ever a reason to solve instances like these in practice, however, optimization should be considered an option. `Concorde` was able to solve all the Random Matrix instances in our testbed to optimality using its default settings. The average normalized running times were also quite reasonable: For

Average Percent Excess over the HK Bound

	N =	1000	3162	10K	31K	100K	316K	1M	3M
U	LK-JM	1.92	1.99	2.02	2.02	1.97	1.96	1.96	1.92
	I3opt-10N	1.16	1.21	1.29	1.37	1.35	1.37	1.37	–
	ILK-JM-.1N	1.25	1.19	1.24	1.36	1.29	1.31	1.32	–
	CLK-ABCC-N	1.02	0.98	0.90	0.89	0.92	0.95	0.91	–
	CLK-ACR-N	0.84	0.93	0.92	0.91	0.92	0.95	0.90	0.86
	ILK-JM-N	0.90	0.87	0.89	1.01	0.94	0.94	–	–
	CLK-ABCC-10N	0.91	0.82	0.83	0.79	0.81	0.84	–	–
	Helsgaun	0.90	0.89	0.83	0.83	–	–	–	–
	Helsgaun-.1N	0.75	0.72	0.69	0.68	–	–	–	–
	Helsgaun-N	0.74	0.71	0.68	0.67	–	–	–	–
C	LK-JM	1.75	2.95	3.41	3.71	3.63	3.67		
	Helsgaun	1.25	2.00	3.32	3.58	–	–		
	CLK-ABCC-N	1.63	2.83	2.09	2.37	1.93	1.96		
	I3opt-10N	0.82	1.43	1.62	2.19	2.02	2.29		
	CLK-ABCC-10N	1.21	1.93	1.83	1.75	1.68	–		
	CLK-ACR-N	0.62	1.27	1.82	1.73	1.78	1.70		
	ILK-JM-.1N	0.87	1.33	1.19	1.91	1.69	1.79		
	ILK-JM-N	0.60	0.78	0.91	1.33	1.16	–		
	Helsgaun-.1N	0.57	0.65	1.05	0.54	–	–		
	Helsgaun-N	0.54	0.62	0.83	0.53	–	–		
T	LK-JM	2.38	2.16	1.92	1.73	1.61			
	Iopt-10N	1.29	1.53	1.04	1.12	1.18			
	ILK-JM-.1N	1.62	1.33	1.05	1.07	0.96			
	CLK-ABCC-N	1.40	1.31	1.01	0.92	0.69			
	CLK-ACR-N	1.12	1.21	0.76	0.88	0.59			
	ILK-JM-N	1.20	1.06	0.73	0.76	0.66			
	CLK-ABCC-10N	1.47	1.12	0.71	0.89	0.52			
	Helsgaun	1.20	1.01	0.77	0.96	1.25			
	Helsgaun-.1N	1.05	0.87	0.55	0.60	0.57			
	Helsgaun-N	1.00	0.86	0.53	–	–			
M	I3opt-10N	6.25	11.85	16.81					
	LK-JM	3.52	4.35	5.91					
	ILK-JM-.1N	2.12	2.88	4.62					
	CLK-ABCC-N	1.82	3.26	4.89					
	CLK-ACR-N	1.50	3.04	3.92					
	ILK-JM-N	1.11	2.30	3.60					
	CLK-ABCC-10N	1.18	2.39	3.74					
	Helsgaun	0.04	0.02	0.03					
	Helsgaun-.1N	0.02	0.01	0.01					
	Helsgaun-N	0.02	0.01	0.01					

Table 9.15. Tour quality results for repeated local search heuristics, with Helsgaun and LK-JM included for comparison purposes.

Average Normalized Running Time in Seconds

	N =	1000	3162	10K	31K	100K	316K	1M
U	LK-JM	0.20	0.7	2	7	23	61	323
	I3opt-10N	4.41	17.3	73	255	868	2660	16200
	ILK-JM-.1N	0.88	4.2	22	117	553	2240	18600
	CLK-ABCC-N	2.22	11.1	64	267	912	2590	16000
	CLK-ACR-N	3.19	14.6	49	164	933	4230	19200
	ILK-JM-N	6.16	32.4	169	920	4530	18500	–
	CLK-ABCC-10N	20.09	98.4	597	2480	8500	24300	–
	Helsgaun	5.64	71.5	862	7820	–	–	–
	Helsgaun-.1N	6.89	113.7	1830	35400	–	–	–
	Helsgaun-N	18.00	468.0	9390	229000	–	–	–
C	LK-JM	1.66	5.0	15	59	173	495	
	Helsgaun	7.02	70.3	768	12800	–	–	
	CLK-ABCC-N	9.32	42.1	206	899	3110	9230	
	I3opt-10N	7.32	27.1	102	327	948	2380	
	CLK-ABCC-10N	90.31	387.0	2010	8810	29900	–	
	CLK-ACR-N	9.99	45.0	142	490	2830	13000	
	ILK-JM-.1N	17.03	74.7	258	1260	4160	14100	
	ILK-JM-N	167.93	698.4	2320	9600	33500	–	
	Helsgaun-.1N	12.0	119.0	1780	26700	–	–	
	Helsgaun-N	67.47	727.0	11900	89500	–	–	
T	LK-JM	0.34	0.6	4	13	24		
	I3opt-10N	5.52	16.5	74	216	582		
	ILK-JM-.1N	1.70	3.7	50	358	663		
	CLK-ABCC-N	2.39	8.3	53	173	442		
	CLK-ACR-N	3.18	9.3	35	73	222		
	ILK-JM-N	11.43	31.5	360	2590	6130		
	CLK-ABCC-10N	22.12	73.9	489	1740	4180		
	Helsgaun	7.82	73.3	1060	7980	48200		
	Helsgaun-.1N	9.61	104.9	4900	120000	303000		
	Helsgaun-N	21.57	404.0	17600	–	–		
M	I3opt-10N	8.12	57.0	394				
	LK-JM	1.15	12.8	161				
	ILK-JM-.1N	3.30	32.0	354				
	CLK-ABCC-N	33.90	377.0	3820				
	CLK-ACR-N	140.29	1410.0	9650				
	ILK-JM-N	13.14	119.2	1250				
	CLK-ABCC-10N	303.00	3040.0	29800				
	Helsgaun	7.78	100.6	1270				
	Helsgaun-.1N	8.11	101.3	1540				
	Helsgaun-N	13.67	237.3	4580				

Table 9.16. Normalized running times for repeated local search heuristics. The "3M" column was omitted so the table would fit on a page. The missing entries are 1255 seconds for LK-JM and 94700 seconds for CLK-ACR-N.

10,000 cities the normalized optimization time was 1,380 seconds, only marginally more than that for Helsgaun.

At the other end of the spectrum, consider I3opt-10N. Although it does very poorly on Random Matrix instances, it finds significantly better tours than LK-JM for the other three classes, with its much greater running time balanced by the fact that it should be much easier to implement than any of the other heuristics in the table. Moreover, it substantially outperforms Helsgaun on large Clustered instances, beats CLK-ABCC-N for the smaller ones, and is fairly close to ILK-JM-.1N for Uniform and TSPLIB instances, with similar running times when N is large. However, if one looks at the details of its runs, it would appear that not much would be gained by providing I3opt more iterations. On average there were no further improvements in the last 15% of the $10N$ iterations. Thus it would appear that the tour quality achieved by ILK-JM-N, CLK-ACR-N and CLK-ABCC-10N is beyond the capabilities of I3opt. Note that the running time growth rate for I3opt-10N appears to be $O(N^{1.25})$, indicating that the use of don't-look bits is paying off asymptotically. A similar effect is observed for Lin-Kernighan based variants, and so faster computers will extend the range of these heuristics more readily than they will Helsgaun and its repeated-run variants.

Turning to the Lin-Kernighan-based variants, we see the effect of the fact that ILK-JM is based on a more powerful but slower Lin-Kernighan engine than CLK-ABCC and CLK-ACR. Even though ILK-JM-N performs only one tenth as many iterations as CLK-ABCC-10N, the running times for the two are (roughly) comparable. So are the tour lengths, with the exception of the Clustered instances. Here ILK-JM-N finds distinctly better tours, possibly because of its use of longer neighbor lists. As to CLK-ACR, note that CLK-ACR-N produces distinctly better tours than CLK-ABCC-N in all but the Uniform class (where they are comparable), so it is possible that results for CLK-ACR-10N, if we had them, would show better tours than ILK-JM-N, again in comparable time. Results for ILK-JM-10N are not included in the table for space reasons, but tend to yield an average tour-length improvement of 0.1% over ILK-JM-N for the three geometric classes (at the price of taking 10 times as long). For Random Matrix instances the improvement is closer to 1%.

At some point, however, simply running for more iterations may not be the best use of computation time. In our next section we consider other possibilities for getting the last ϵ improvement in tour quality: heuristics that use Chained Lin-Kernighan as a subroutine. But first, a brief digression to follow up our earlier remark about the relation between Tabu Search and Chained Lin-Kernighan.

Tabu Search. In its simplest form, a Tabu Search heuristic operates as follows. As with other local search variants, it assumes a neighborhood structure on solutions. Given the current solution, we find the the best neighbor (or the best of a random sample of neighbors) subject to certain "Tabu" restrictions needed to avoid cycling. One then goes to that neighbor, whether it is an improving move or not. Note that in effect this breaks the search into alternating phases. First we make improving moves until a local optimum (or at least an apparent local optimum) is found, then we perform a "kick" consisting of one or more uphill moves until we reach a solution from which we can once again begin a descent. A full Tabu Search heuristic is usually much more complicated than this, using various strategies for "diversification" and "intensification" of the search, as well as parameters governing lengths of Tabu lists and "aspiration levels," etc., for which see [369, 370]. However, the underlying similarity to chained local optimization remains.

And note that, in the case of the STSP at least, Tabu Search comes with extra overhead: Whereas in most of the local search implementations studied above, one performs one of the first improving moves seen, in a Tabu Search heuristic one typically must generate a sizable collection of moves from which the best is to be picked. This perhaps partially explains the results observed for the one collection of Tabu Search variants submitted to the challenge, implemented by Dam and Zachariasen [238]. Their implementations allow for the possibility of using different neighborhood structures for the downhill and uphill phases with the choices being the 2-Opt neighborhood, the double bridge neighborhood (`DB`), the standard LK-search neighborhood (`LK`), and the Stem-and-Cycle variant on it, called "flower" in [238] (`SC`). Interestingly, the best tours are most often found by the variants closest to Chained Lin-Kernighan: those that use standard LK-search or the Stem-and-Cycle variant for the downhill phase and double-bridge moves for the uphill phase, with the LK-search variant being substantially faster.

Unfortunately the running times even for this version are sufficiently slow that it is almost totally dominated by `ILK-JM-N` and `CLK-ACR-N`. The latter almost always finds better tours and averages between 35 and 200+ times faster (normalized running time) depending on instance class and size. Details can be viewed at the Challenge website. Although the Tabu Search implementations did not use all the available speedup tricks and is not as highly optimized as `ILK-JM-N` and `CLK-ACR-N`, it seems unlikely that more programming effort would bridge this gap.

3.7. Using Chained LK as a Subroutine

Just as we were able to get improved results out of Lin-Kernighan by using it as a subroutine in Chained Lin-Kernighan, one might consider improving on Chained LK by using *it* in more complicated procedures. One simple possibility would be to use Chained LK to generate starting tours for a second local search heuristic. Given how effective Chained LK already is, the second heuristic would probably need a neighborhood structure that is quite different from that used by Chained LK. Balas and Simonetti have proposed a likely candidate in [80].

Balas and Simonetti Dynamic Programming. This approach starts by identifying a *home* city c_1 and without loss of generality represents all tours as ordered sequences of the cities starting with c_1. For any fixed k, we say that such a tour T' is a *k-bounded neighbor* of a tour $T = c_{\pi[1]}, c_{\pi[2]}, \ldots, c_{\pi[N]}$ if for no i, j with $i \geq j + k$ does city $c_{\pi[j]}$ occur after $c_{\pi[i]}$ in T'. Note that in this neighborhood structure, two tours can be neighbors even if they have no edges in common. This is in sharp contrast to the k-Opt neighborhood structure, in which neighboring tours differ in at most k edges. Moreover, the new neighborhood can be searched much more effectively. For each k there is a linear-time dynamic programming algorithm that for any tour T finds its *best k-bounded neighbor* [80]. The running time is exponential in k, but this is not a major problem for small k. One thus can use this algorithm to perform steepest-descent local search under the k-bounded neighborhood structure.

Simonetti submitted results to the Challenge using this approach with $k \in \{6, 8, 10\}$ and with CLK-ABCC-N used to generate starting tours. The combination was run on all benchmark instances with one million or fewer cities except the 10,000-city Random Matrix instance. Improvements over the starting tour were found on 20 of these 87 instances when $k = 6$, on 22 when $k = 8$ and on 25 when $k = 10$. On larger instances, improvements were found at a significantly higher rate. For problems over 30,000 nodes, improvements over the starting tour were found on 11 of 13 instances when using $k = 6$, and on 12 of 13 instances with $k = 8$ and 10. Improvements were small, however. Even for $k = 10$ there were only six improvements larger than 0.01% and only one larger than 0.02%. This was an improvement of 0.07% for rl1323 and was already found with $k = 6$. In a sense this last improvement is quite substantial, since it reduced the tourlength from 0.60% above optimum to 0.53%. Moreover, even tiny improvements can be worthwhile if they can be obtained relatively inexpensively, which in this case they can. For $k = 8$ the added overhead for applying this algorithm to the tour output by CLK-ABCC-N is minimal, ranging from 5 to 20%, and even for $k = 10$ the total running time was increased by at most a factor of 2.4. Thus this approach might make a worthwhile low-cost general post-processor to any heuristic, especially for the larger instances where it appears to be more successful. The code is currently available from Simonetti's TSP webpage, `http://www.contrib.andrew.cmu.edu/~neils/tsp/index.html`.

The second heuristic we cover also uses Chained LK to generate starting tours, but in a more sophisticated framework.

A Multi-Level Approach (Walshaw [818]). The idea is to recursively apply Chained LK (or any other local search heuristic) to a smaller "coalesced" instance in order to generate a starting tour for the full instance. A coalesced instance is created by matching nearby cities and requiring that the edge between them be included in all generated tours (*fixing* the edge). If the instance to be coalesced already has fixed edges, then the only cities that can be matched must have degree 0 or 1 in the graph of fixed edges and can't be endpoints of the same path in that graph. Note that a coalesced instance containing a fixed path can be modeled by a smaller instance in which that path is replaced by a single fixed edge between its endpoints and the internal cities are deleted. Starting with the original instance, we thus can create a sequence of smaller and smaller instances by repeatedly applying this approach until we reach an instance with four or fewer cities.

Having constructed this hierarchical decomposition, we then proceed as follows, assuming our base heuristic has been modified so that it always outputs a tour containing all fixed edges. We start by applying the heuristic to the last (smallest) instance created, using the heuristic's native starting tour generator. Thereafter, as we progress back up the hierarchy, we use the result of running the heuristic on the previous instance as the starting tour for the current instance. We end up running the base heuristic $\Theta(\log N)$ times, but most of the runs are on small instances, and overall running time is no more than 2 to 3 times that for running the base heuristic once on the full instance. See [818] for more details, including the geometry-based method for matching cities during the coalescing phase. Walshaw submitted results for two instantiations of the Multi-Level approach. The first (`MLLK`) used `LK-ABCC` as its base heuristic and the second (`MLCLK-N`) used `CLK-ABCC-N`. The base heuristics were forced to obey fixed-edge constraints by setting the costs of required edges to a large negative value.

`MLLK` could well have been discussed in the previous section. Although it can't compete with `CLK-ABCC-N` on average tour quality, for large instances it is 15 to 35 times faster, and it did find better tours for a significant number of Clustered instances. When compared to

LK-ABCC, it found better tours on average for all three geometric classes, with the advantage averaging 3% for Clustered instances. MLCLK-N did not obtain significant tour-length improvements over its base heuristic (CLK-ABCC-N) for Uniform instances, but it did find better tours for over half the TSPLIB instances and almost all the Clustered instances. Its tours averaged 1% better for this latter class.

The two approaches considered so far in this section needed little more than twice the time for running Chained LK. In the remainder of the section, we consider what one might do if one wants even better tours and is willing to spend substantially more computational resources to get them. Currently the people willing to pay this price are mainly researchers interested in testing the ultimate limits of heuristic techniques and in generating better upper bounds on unsolved testbed instances, but this is an active community. Work in this area has also followed the paradigm of using Chained LK as a subroutine, but now the key factor being exploited is the randomization inherent in the heuristic.

Multiple Independent Runs. Chained Lin-Kernighan is a randomized heuristic, not only because of possible randomization in its starting tour generation, but also because the kicks are randomly generated. If one runs it several times with different seeds for the random number generator, one is likely to get different tours. This fact can be

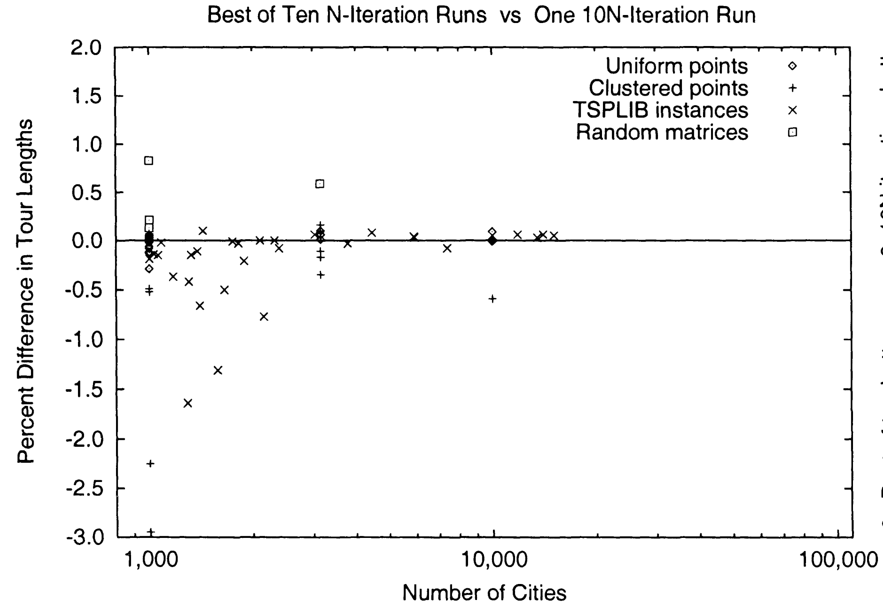


Figure 9.12. Tour quality comparisons between taking the best of ten runs of CLK-ABCC-N and taking the result of a single run of CLK-ABCC-10N.

exploited once one reaches the point where added iterations no longer seem to be helping, that is, it might be more effective to take the best of ten N-iteration runs instead of just performing one $10N$-iteration run, which would take roughly the same time. See Figure 9.12, which for all the instances in our testbeds with 100,000 or fewer cities compares the best tour length over ten runs of CLK-ABCC-N (a composite heuristic we shall denote by CLK-ABCC-N-b10) with the tour length for one run of CLK-ABCC-10N. Although the "best of ten" strategy falls down for Random Matrix instances, it never loses by much on any instance from the other three classes, and it wins by significant amounts on many of the smaller Clustered and TSPLIB instances.

Note that taking the best of ten runs of a heuristic may not be enough to make up the gap between that base heuristic and a better heuristic. For example, Walshaw's Multi-Level Approach using CLK-ABCC-N does even better on Clustered instances than does CLK-ABCC-N-b10 and takes only 1/5 as much time. Furthermore, Helsgaun's variant on Chained LK is so good that, even with just $0.1N$ iterations it consistently outperforms CLK-ABCC-N-b10, as shown in Figure 9.13. Moreover, Helsgaun-.1N is typically faster for the smaller instances, although its running time grows much more rapidly with N, so that by the time one reaches 30,000 cities it is more than 10 times slower than the best-of-ten approach.

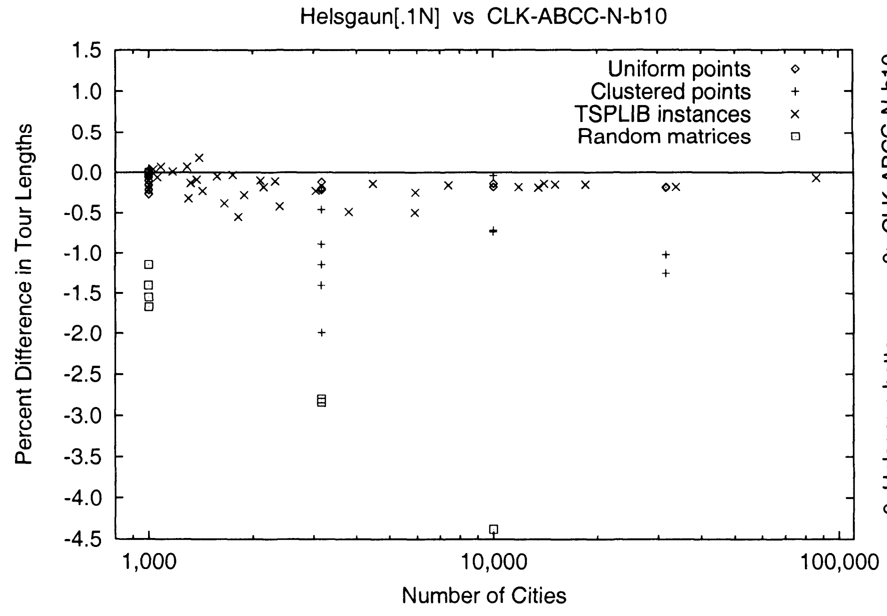


Figure 9.13. Tour quality comparisons between a $0.1N$ iteration run of Helsgaun's LK variant and taking the best of ten runs of CLK-ABCC-N.

This doesn't, however, totally rule out the value of the simple best-of-ten approach. The best of ten runs of `Helsgaun-.1N` might be an interesting competitor for `Helsgaun-N`, assuming the running time is available. There are, however, more creative approaches to exploit the randomization involved in our heuristics.

Tour Merging (Applegate, Bixby, Chvátal, and Cook [27]). This approach is based on the observation that tours that are very close to optimal share many common edges. Moreover, if one takes the union of all the edges in a small set of such tours, one typically obtains a graph with low "branch-width," a graph parameter introduced by Robertson and Seymour [724] that in a sense quantifies how "treelike" a graph is.

This can be exploited algorithmically as follows. First, although it is NP-hard to determine the precise branch-width of a graph, there are effective heuristics for finding near-optimal branch-width decompositions of low branch-width graphs. Second, there is a dynamic programming algorithm due to Cook and Seymour [208] that, given an edge-weighted Hamiltonian graph G and a branch decomposition for it with branch-width k, finds the shortest Hamiltonian cycle in G in time linear in N. The running time is exponential in k but is quite feasible for small values. Thus one can typically take the results of several runs of Chained LK and find the optimal tour contained in the union of their edges. During the last few years, this technique has helped Applegate, Bixby, Chvátal, and Cook solve many previously unsolved `TSPLIB` instances. Note that in this approach one needn't restrict oneself to Chained LK tours. Applegate et al. typically performed many tour-merging runs, often using the result of one run as one of the tours to be merged in a subsequent run. Moreover, the recent solution of `d15112` was aided by improved upper bounds that were obtained by applying tour-merging to Iterated Helsgaun tours in addition to Chained LK tours.

Although tour-merging has most typically been used in a hands-on fashion, with manual choice of tours to be merged, etc., it is possible to run it in a stand-alone manner. Cook submitted results to the Challenge for a heuristic of this sort (`Tourmerge`), in which the tour-merging process is applied to the results of ten runs of `CLK-ABCC-N`. Five runs of `Tourmerge` were performed on the set of all testbed instance with fewer than 10,000 cities. The heuristic was not always successful. For none of the runs on Random Matrix instances did the combined graph for the 10 tours have low enough branch-width for tour-merging to be feasible. Similar failures occurred in all runs on the `TSPLIB` instances `u2319`, `fnl4461`, and `pla7397` and in one or two runs on each of the 3162-city Uniform instances. Nevertheless, results *were* generated for most of the instances attempted and were substantially better than those for simply taking the best of the ten runs of CLK-ABCC-N. Moreover, if we take the best of the five tour-merging runs (a composite heuristic we shall denote by Tourmerge-b5), we now become competitive with Helsgaun's heuristic, even if the latter is allowed N rather than $0.1N$ iterations.

Figure 9.14 compares the results of Tourmerge-b5 and Helsgaun-N on all instances for which Tourmerge-b5 generated tours. Note first the much narrower range in differences. Tourmerge-b5 is never more than 0.05% worse (when it actually produces a solution) and is never more than 0.20% better. This is in comparison to the −4.5% to +1.5% range in Figure 9.13. The biggest variation is on TSPLIB instances, where Tourmerge-b5 does better more often than it does worse.

The normalized running time for performing all five tour-merging runs can however be substantially worse than that for one run of Helsgaun-N, typically some 5 times slower but occasionally as much as 100. (For the instances where some of the runs had to be aborted because of high branch-width, no times were reported, so we estimated the running time for the failed run as the average of those for the successful runs.) Moreover, for 14 of the 54 instances on which Tourmerge-b5 was successfully run, its normalized time was greater than that for finding the *optimal* solution and proving optimality using Concorde's default settings. In five cases Concorde was faster by a factor of 4 or more. (Helsgaun-N was

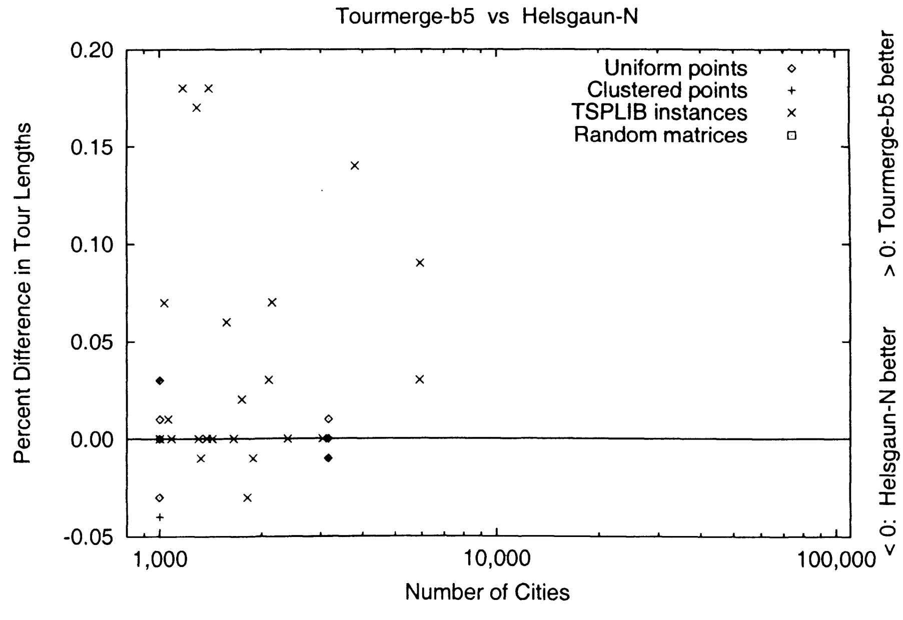


Figure 9.14. Tour quality comparisons between an N-iteration run of Helsgaun's LK variant and taking the best of five tour-merges, each involving ten CLK-ABCC-N tours.

slower than Concorde-optimization on only two instances.) Nevertheless, the time for Tourmerge-b5 would probably be manageable in many applications. With the exception of the 3,162-city Uniform instances, no instance with less than 10,000 cities required more than an hour of normalized time, and even the slowest 3,162-city instance took just 16 hours (and for this instance Tourmerge-b5 found an optimal solution, whereas full optimization via Concorde under default settings was still far from complete when terminated after several days).

The fact that neither Tourmerge-b5 nor Helsgaun-N beats the other on all instances suggests a final "heuristic" for getting improved results at the cost of increased running times: Run both and take the best tour found. Better yet, run both plus the several other effective heuristics from this and the previous section and take the best tour found. Table 9.17 summarizes how that approach would work on the subset of the 71 Challenge testbed instances for which optimal solution values are known, using the four heuristics Tourmerge-b5, Helsgaun-N, ILK-JM-10N, and ILK-JM-N-b10 (the best of ten runs of ILK-JM-N).

Running times for large instances would of course be huge, but this approach might well be the method of choice when Concorde is unsuccessful in optimizing within the large number of hours or days one is willing to spend. Recall from Table 9.1 that for a significant subset of the instances with fewer than 10,000 cities Concorde under its default settings took more than 100 (normalized) hours if it succeeded at all, and it failed for all our geometric instances with more than 4,000 cities.

4. Conclusions and Further Research

In this chapter we have discussed a wide variety of heuristics occupying many positions on the tradeoff curve for tour quality and running time. Moreover, we have compared their performance on a wider range of instance types and sizes than in previous studies. In order to get such broad coverage, however, we have had to accept some compromises, and it is appropriate to remind the reader about these.

First, for many of the codes we cover, the results we include come from unverified reports submitted by the implementers, based on runs on a variety of machines. As discussed in Section 2.2, our methodology for normalizing reported running times is necessarily inexact. Second, as seen in many places in this chapter, two implementations of the same heuristic can have markedly different running times, even on the same machine, depending on the data structures and coding expertise used. For local search heuristics, even the tour lengths can differ depending on implementation choices. Thus the conclusions we draw are often as

Instance	Percent Above Optimal	Heuristic	Instance	Percent Above Optimal	Heuristic
E1k.0	–	h	M3k.0	–	h
E1k.1	–	h,m	M3k.1	–	h
E1k.2	–	h,m	M10k.0	0.0033	h
E1k.3	–	m,i10	dsj1000	–	h
E1k.4	–	h,m	pr1002	–	h,m,i10
E1k.5	–	h,m,ib	si1032	–	m,ib
E1k.6	–	h	u1060	–	m,i10
E1k.7	–	m	vm1084	–	h,m
E1k.8	0.0012	m	pcb1173	–	m
E1k.9	–	h,m,ib	d1291	–	ib
E3k.0	0.0034	h	rl1304	–	h,ib,i10
E3k.1	–	h	rl1323	–	h,ib
E3k.2	–	h	nrw1379	–	h,m
E3k.3	0.0174	m	fl1400	–	m,ib,i10
E3k.4	–	m	u1432	–	h,m
C1k.0	–	h,m	fl1577	–	m,ib
C1k.1	–	h,m,ib,i10	d1655	–	h,m
C1k.2	–	h,ib	vm1748	–	m,ib,i10
C1k.3	–	h,m,ib,i10	u1817	–	h
C1k.4	–	i10	rl1889	0.0013	h
C1k.5	–	ib,i10	d2103	–	ib,i10
C1k.6	–	h	u2152	–	h
C1k.7	–	h,ib	u2319	–	m
C1k.8	–	h,m,ib,i10	pr2392	–	h,m
C1k.9	–	m,ib,i10	pcb3038	–	h
C3k.0	–	h,i10	fl3795	–	m,i10
C3k.1	–	h	fnl4461	0.0027	h
C3k.2	–	i10	rl5915	0.0057	m
C3k.3	–	m	rl5934	0.0023	m
C3k.4	–	h,m	pla7397	–	h
M1k.0	–	h	rl11849	0.0610	h
M1k.1	–	h	usa13509	0.0065	h
M1k.2	–	h	d15112	0.0186	h
M1k.3	–	h			

Table 9.17. Best results obtained for all testbed instances whose optimal solutions are known. Four heuristics sufficed to generate these results: `Helsgaun-N` (abbreviated in the table as `h`), `Tourmerge-b5` (`m`), `ILK-JM-10N` (`i10`), and best of ten runs of `ILK-JM-N` (`ib`). Where more than one heuristic found the best solution, we list all that did.

much about the implementation as about the heuristics implemented. Finally, most of our results only cover one run of the given heuristic on each instance, which for heuristics that incorporate randomization can lead to very noisy data. For this reason we have tried to concentrate on averages over similarly sized instances and on observable patterns in the data, rather than results for particular instances.

Time constraints have also had their effect, forcing us to concentrate on just four instance classes with an emphasis on 2-dimensional geometric instances, the type that occur most often in practice. We also have not considered in detail the dependence of heuristic performance on parameter settings (for example the lengths of neighbor lists, the choice of starting tour, or the nature of the "kick" in Chained Lin-Kernighan). Such questions should more naturally fall in the domain of individual papers about the heuristics in question, and we hope the Challenge testbeds and benchmarks will facilitate such work in the future. We intend to provide just such detailed experimental analysis for the Bentley and Johnson-McGeoch implementations discussed here in the forthcoming monograph [461]. As mentioned earlier, we also plan to maintain the Challenge website indefinitely as a resource and standard of comparison for future researchers.

If the reader is to take one lesson away from this chapter, it should be the high level of performance that can be attained using today's existing heuristics, many of them with publicly available implementations. A second major lesson concerns the large extent to which a heuristic's performance (both running time and tour quality) can depend on implementation details, as we have seen many times in this chapter.

As a final lesson, let us review once more the wide range of trade-offs to which we referred above. Recall that we have already provided one illustration of this in Section 3. In that section, Figure 9.3 and Table 9.2 showed the range of performance possibilities for Random Uniform Geometric instances. For such instances and most heuristics, the percent excess over the Held-Karp bound appears to approach a rough limiting bound, which typically has almost been reached by the time $N = 10,000$. The table and figure presented average results for selected heuristics on Uniform instances of that size, covering a wide range of behavior.

Results can be more instance-dependent for structured instances such as those in TSPLIB. To put the earlier table in perspective (while partially ignoring the above-mentioned proviso about drawing detailed conclusions from single runs on individual instances), we conclude our discussion with Table 9.18, which considers the currently largest solved TSPLIB instance d15112, and gives both the percentage excess above optimum and the normalized running time obtained for it by all the key implementations covered in the chapter, as well as a few related ones. The abbreviated heuristic names used in the table and elsewhere in this chapter are explained in Table 9.19. Analogous tables for other instances can be generated from the Comparisons page at the Challenge website.

Acknowledgment. The authors thank Jon Bentley, Gregory Gutin, and César Rego for helpful comments on drafts of this chapter and all the Challenge participants for their participation and feedback.

Percent Above Optimal	Running Time (Seconds)	Heuristic	Percent Above Optimal	Running Time (Seconds)	Heuristic
-0.5215	90.13	HeldKarp	2.8124	3.27	3opt-B
0.0000	–	Optval	3.0729	16.12	Hyper-3
0.0186	26322.93	Helsgaun-N	3.1389	2773.49	GENIUS-10
0.0236	7896.88	Helsgaun-.1N	3.7261	1.94	2.5opt-B
0.1051	1515.99	Helsgaun	4.0234	2.53	2opt-JM-40
0.1266	980.66	CLK-ABCC-10N	4.2518	1.73	2opt-JM-20
0.1358	8738.10	ILK-JM-10N	4.2704	1.81	2opt-JM-20a
0.1744	1046.75	CLK-ABCC-N-b10	4.3852	2.53	2opt-JM-40a
0.1810	102.55	CLK-ABCC-N	4.5620	2.47	Hyper-2
0.1952	154.79	MLCLK-N	4.7515	325.66	GENI-10
0.1988	3408.07	ILK-JM-N-b10	5.1630	1.56	2opt-JM-10
0.2013	65.36	CLK-ACR-N	5.2668	1.81	2opt-B
0.2114	89.59	MLCLK-.5N	6.2298	14.14	Christo-HK
0.2259	542.61	ILK-Neto-N	8.9277	1.65	Christo-G
0.2447	332.16	ILK-JM-N	10.4116	0.73	AppChristo
0.3359	108.29	ILK-JM-.3N	10.7438	2578.46	CCA
0.4451	21.53	MLCLK-.1N	11.0510	0.41	Savings
0.5232	42.07	ILK-JM-.1N	11.7303	4.67	FI
0.6138	6.55	CLK-ACR-1000	11.8092	2.33	FA$^+$
0.6282	69.52	ILK-Neto-.1N	12.9387	3.38	RI
0.6464	112.96	I3opt-10N	13.6291	1.65	Christo-S
0.6747	15.64	LK-HK	13.6803	1.13	RA$^+$
0.7579	38531.28	Tabu-SC-SC	13.6803	0.17	Q-Boruvka
0.9739	2203.88	Tabu-LK-DB	14.5281	0.26	Boruvka
1.0929	4.96	LK-JM-40	14.7968	0.44	Greedy-ABCC
1.1038	3.03	LK-JM-20	15.9946	31.09	Litke-15
1.1418	5.70	LK-Neto-20	17.1529	1.31	CI
1.1443	11.74	LK-Neto-40	17.3106	1.38	CHCI
1.1620	2.98	LK-JM-20a	22.3039	2.84	NI
1.1975	35660.20	Tabu-SC-DB	22.3094	2.01	NA$^+$
1.2170	2.61	LK-JM-10	23.1644	0.49	DENN
1.2549	4.23	LK-Neto-12	24.6647	0.14	NN-ABCC
1.2678	498.12	SCLK-R	28.5153	1.59	NA
1.3357	134.58	SCLK-B	31.0319	100.03	Karp-20
1.3422	2.01	CLK-ACR-100	32.0524	0.06	Spacefill
1.4427	4.97	MLLK	36.1238	0.55	DblMST
1.8178	2.38	LK-ABCC	37.0620	0.60	RA
1.8456	1.53	CLK-ACR-10	42.1799	1.77	FA
1.8761	1.46	LK-ACR	42.8104	0.05	Strip
2.1715	1.90	3opt-JM-20	42.8104	0.07	Strip2
2.3504	2.73	3opt-JM-40	49.2882	0.17	FRP
2.5974	136.37	Hyper-4	59.2344	0.26	Karp-15
2.7254	1.70	3opt-JM-10	–	0.03	Read

Table 9.18. Tour quality and normalized running times for TSPLIB instance d15112. General conclusions should not be drawn from small differences in quality or time.

Abbrev	Short Description of Heuristic
`AppChristo`	Approximate Christofides (`JM`)
{`Q-`}`Boruvka`	`Concorde`'s implementation of (Quick) Boruvka
`CCA`	The Golden-Stewart CCA heuristic (`JM`)
{`CH`}`CI`	The `JM` implementations of (Convex Hull) Cheapest Insertion
`CLK-`X`-`k	The X version of Chained LK with k iterations
`Christo-`{`S`,`G`}	The Christofides heuristic with {standard,greedy} shortcuts (`JM`)
`Christo-HK`	Christofides using Held-Karp one-trees instead of MST's (Rohe)
`Concorde`	Concorde used for optimization with default settings
`DblMST`	The Double Minimum Spanning Tree heuristic (`JM`)
`F`{`I`,`A`,$\mathtt{A}^+$}	Bentley's farthest {insertion,addition,augmented addition}
`FRP`	Bentley's implementation of the Fast Recursive Partitioning heuristic
`GENI`{`US`}`-`p	The Gendreau-Hertz-Laporte GENI(US) heuristic with p neighbors
`Greedy`{`-`X}	The X implementation of Greedy (Default: `JM` implementation)
`HeldKarp`	Held-Karp bound as computed by `Concorde`
`Helsgaun`{`-`k}	Helsgaun's heuristic (with k iterations)
`Hyper-`k	The Burke-Cowling-Keuthen implementation of k-Hyperopt
`I3opt-`k	The `JM` implementation of Iterated 3-Opt with k iterations
`ILK-`X`-`k	The X version of Iterated LK with k iterations
`Karp-`n	Karp's Partitioning heuristic for maximum subproblem size k (`JM`)
k`opt-`X	The X implementation of k-Opt, $k \in \{2, 2.5, 3\}$
k`opt-JM`{`-`p}	The `JM` implementation of k-Opt with p quad neighbors
`Litke-`k	Litke's Clustering heuristic for maximum subproblem size k (`JM`)
`LK-`X`-`p	The X version of basic Lin-Kernighan with p quad neighbors
`LK-`X{`-BD`}	The X version of basic Lin-Kernighan with default neighbor lists (and bounded depth LK-searches if that is not the default)
`LK-HK`	Lin-Kernighan using `Christo-HK` starts (Rohe)
`MLCLK-`k	Walshaw's implementation of Multi-Level k-iteration Chained LK
`MLLK`	Walshaw's implementation of Multi-Level Lin-Kernighan
`N`{`I`,`A`,$\mathtt{A}^+$}	Bentley's nearest {insertion,addition,augmented addition}
`NN`{`-`X}	The X implementation of Nearest Neighbor (Default: `B`)
`Optval`	Optimal solution lengths (from a variety of sources)
`R`{`I`,`A`,$\mathtt{A}^+$}	Bentley's random {insertion,addition,augmented addition}
`Read`	Time to simply read the instance using standard I/O routines
`Savings`	The `JM` implementation of the Clarke-Wright "Savings" heuristic
`SCLK-`{`R`,`B`}	The Glover-Rego implementation of a Stem-and-Cycle variant of Lin-Kernighan with {random,`boruvka`} starts
`Spacefill`	The Bartholdi-Platzmann implementation of Spacefilling Curve
`Strip`{`2`}	The `JM` implementation of the Strip (2-Way Strip) heuristic
`Tabu-`D`-`U	The Dam-Zachariasen implementation of Tabu Search using the D (U) neighborhood for downhill (uphill) moves
`Tourmerge`	The tour-merging heuristic of `ABCC` applied to 10 runs of `CLK-ABCC-N`

Table 9.19. Abbreviated names used in this chapter. The symbol X stands for an abbreviation of the implementers' names: "`ABCC`" for Applegate, Bixby, Chvátal, and Cook (`Concorde`), "`ACR`" for Applegate, Cook, and Rohe, "`B`" for Bentley, "`JM`" for Johnson-McGeoch, "`Neto`" for Neto, and "`R`" for Rohe. Adding the suffix "`-b`n" to any name means that one is taking the best of n runs.