Efficient cluster compensation for Lin-Kernighan heuristics

by

David M. Neto

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Efficient cluster compensation for Lin-Kernighan heuristics

David M. Neto
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
1999

For certain problems such as the Traveling Salesman Problem, the Lin-Kernighan heuristic and its derivatives are among the most successful algorithms known to optimization practice. It usually runs quickly, producing nearly optimal answers. Unfortunately, run times are usually longer on sharply clustered instances than on more uniform instances. This dissertation introduces *efficient cluster compensation*, an algorithmic technique designed to reduce the performance penalty Lin-Kernighan suffers on clustered inputs. The technique aims to decrease running times while maintaining the quality of the answers produced. The strategy is to prune unfruitful portions of the search space by incorporating extra lookahead into the guiding utility function. The lookahead takes the form of the *cluster distance* between two points, a value computed in constant time given modest preprocessing.

Efficient cluster compensation reduces running times on nearly all inputs tested, not just sharply clustered instances. When it increases running times, the slowdown is not severe. Efficient cluster compensation therefore delivers overwhelming benefit at little or no cost.

Heuristics are notorious for behaving unpredictably. Making algorithmic choices and tuning parameters is rightly described as a black art. A broader contribution of this thesis is a better understanding of the qualitative behaviour of the Lin-Kernighan heuristic. The key features of a worst-case result, Papadimitriou's proof that Lin-Kernighan solves a PLS-complete problem, are conjectured to be the cause of bad behaviour observed by

Johnson. This intuition is validated by the performance improvement seen when cluster compensation is used, and by comparing several of the results with perturbed data.

This thesis also introduces several novel instance generation algorithms. They are used to test the generalizability and the robustness of the experimental results reported in earlier chapters, and to test our intuition about the behaviour of the heuristic.

The Lin-Kernighan heuristic is best known as a heuristic for the Traveling Salesman Problem. Yet the TSP heuristic is only one instance of the more general Lin-Kernighan strategy to which cluster compensation applies. We test cluster compensation using it in the Lin-Kernighan heuristic for the TSP and in a Lin-Kernighan heuristic for the minimum weight perfect matching problem.

Acknowledgements

Only a small part of an iceberg shows above the surface. The same is true of a doctoral thesis.

Professor Derek Corneil supervised the work presented in this thesis. I cannot say enough about him in this regard. In addition to bringing his considerable technical skill and depth of knowledge to bear, he was also a tremendous grounding force. He was as interested in my ordinary life as my research and technical life. His no nonsense attitude is much appreciated.

Throughout my doctoral career I received excellent advice from the rest of my advisory committee, Professors Allan Borodin, Ken Sevcik, Mike Molloy, Ken Sevcik, and Faith Fich. They helped me keep the big picture in mind and steered me away from dead ends and tar pits. When necessary, they also tended to the little things, like fixing my spelling and pointing out unclear passages.

Professor Vassos Hadzilacos graciously stepped in at a late date to read the thesis and ask me tough questions at my departmental exam. Professors William Cook and Mike Carter served as external examiners for my final defense. Their truly constructive criticism helped me clarify several crucial points in both my work and its relation to the work of others. Of course, any faults remaining in the thesis are entirely my own responsibility.

Early in my doctoral career Professor Tarek Abdelrahman introduced me to heuristics for the Traveling Salesman Problem, and advised me to keep my "hands dirty" by always returning to experiments as the final authority on the performance of heuristics.

I thank the Department of Computer Science at the University of Toronto as a whole. It fosters excellence in a collegial and multidisciplinary environment. There's usually someone who knows the answer, someone who knows there is no answer yet, someone who knows there won't likely be an answer, or someone knows there is no answer at all. (Life is complex on the cutting edge.) The Department of Computer Science also provided financial support for a portion of my doctoral studies. As well, four years of my graduate career were funded by the Natural Sciences and Engineering Research Council of Canada.

As a student I have had many wonderful teachers, but I will single out two. Professor Stephen Cook showed me time and again that difficult ideas can be made easier if they are expressed clearly. John Watkins, my high school history teacher, taught me invaluable skills for a complex world: to think critically, to weigh evidence, and to judge information together with its source.

As a long-time teaching assistant, I also thank my students. Those blank stares were a powerful invitation to re-evaluate the way I was thinking about an idea, or the way I was expressing it. They taught me to do more than just brute-force it, and that thinking about a problem in multiple ways can be very useful. It was very rewarding to see those blank stares turn to flashes of understanding.

I am lucky to have the school friends I do. Their long-time support and encouragement are irreplaceable. They know who they are. (More importantly, they have my email address, and I use theirs.) More recent friends were also excellent sounding boards for my embryonic technical ideas.

I thank Rob and Nita Hill for funding the last part of my studies through a flexible work arrangement. Mercifully, the work provided much needed cognitive relief, being quite different from my research. My new friends in industry also supported me with encouragement and the latitude to pursue my graduate work.

Dr. Jack Birnbaum was a wonderful coach, giving me the tools to put many things into perspective.

I thank my parents João and Madelena Neto for valuing education, giving me the freedom to study what I wanted, and for much appreciated financial support throughout my university career. My siblings John, George, Cecilia, Gene, and Teresa were my earliest teachers and supporters.

My in-laws Cheryl and Dr. Michael Irving and their son Zachary are truly a second family to me. For six years they shared their home while my wife and I worked through graduate school. Their moral and physical support have been much appreciated.

Finally, I thank my wife, Dr. Robyn Irving. Her unconditional love and uncompromising grace have seen me through the worst and the best of my whole graduate career. I could not have had a more deeply committed supporter. She knows me like no other.

This thesis is dedicated to my first child, who is only weeks away from being born. Daddy finally got it out of the way, and can go on to more important things.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since the 1970s and the advent of the theory of NP-completeness, deciding if a particular problem is difficult or easy to solve has naturally rested on determining whether it can be solved in polynomial time or whether it is NP-hard [24, 50, 35, 25]. For approximation problems in combinatorial optimization, more recent work in the theory of MAX-SNP-completeness also encourages the "easy" *vs.* "hard" determination to be based on membership in certain complexity classes [69, 11].

Theory can guide practitioners needing to produce answers for their problems. It informs them of the nature of algorithms likely to do the job. On the flip side, hardness results can suggest the problem be relaxed or reformulated.

Yet even when polytime algorithms are known for an exact or relaxed problem, they may run too slowly on the instances one cares about. In that case the practitioner turns to any algorithm that works in their problem domain, regardless of worst case or even average case results. One's own problems are hardly ever "average."

Ironically, often the algorithms providing the best answers in reasonable time and space give only weak guarantees. The Lin-Kernighan heuristic for the Traveling Salesman Problem (TSP) [61] has for a long time been the champion heuristic for that problem [44, 46]. It is also a key ingredient in the most successful exact algorithms for the TSP [6, 4, 26]. Lin-Kernighan for the TSP is a local search heuristic, taking a perhaps suboptimal tour and improving it through sets of edge exchanges. Its performance guarantees are rather weak; for an n vertex instance the resulting tour is not more than $4\sqrt{n}$ times longer than an optimal tour [22],[1] and also the final result is no worse than the initial

[1] The $4\sqrt{n}$ approximation ratio holds for the 3-Opt local search heuristic, and Lin-Kernighan subsumes 3-Opt. For geometric instances the approximation ratio improves to $O(\log n)$.

tour it was given. As for running times, one can construct a family of graphs forcing Lin-Kernighan for the TSP to take an exponential number of steps [67][2].

In practice Lin-Kernighan usually runs quickly. Unfortunately, run times can be much longer on sharply clustered instances than on more uniformly distributed instances [44] [46, p. 296]. This dissertation introduces *efficient cluster compensation*, an algorithmic technique designed to reduce the performance penalty Lin-Kernighan suffers on clustered inputs. The technique aims to decrease overall running times while maintaining the quality of the answers produced by the heuristic. The strategy is to prune unfruitful portions of the search space by incorporating extra lookahead into the utility function guiding the search. The lookahead takes the form of the *cluster distance* between two points, a value which can be computed very quickly given modest one-time preprocessing requirements.

In fact, efficient cluster compensation reduces Lin-Kernighan running times on nearly all classes of inputs tested, not just sharply clustered instances. In the rare cases cluster compensation increases running times, the slowdown is not severe. Efficient cluster compensation therefore earns a place alongside many other algorithmic techniques in the practitioner's toolbox, delivering overwhelming benefit at little or no cost.

Heuristics are notorious for behaving unpredictably. Making algorithmic choices and tuning parameters is rightly described as a black art. At a broader level, this thesis contributes a better understanding of the qualitative behaviour of the Lin-Kernighan heuristic. The key qualitative features of a worst-case analysis [67] are conjectured to be the cause of bad behaviour observed by Johnson [44], and Johnson and McGeoch [46]. This intuition is validated by the performance improvement seen when cluster compensation is used, and by comparing several of the results with perturbed data.

This thesis also introduces several novel instance generation algorithms. They are used to test the generalizability and the robustness of the experimental results reported in earlier chapters, and to test our intuition about the behaviour of the heuristic.

The Lin-Kernighan heuristic is best known as a heuristic for tackling the Traveling Salesman Problem. Yet the TSP heuristic is only one instance of the more general Lin-Kernighan strategy to which cluster compensation applies. We test the general suitability of cluster compensation by demonstrating its use in the Lin-Kernighan heuristic for the

[2] A technical requirement of Papadimitriou's argument prevents the result from applying to current implementations of the Lin-Kernighan heuristic. See also Section 4.3 and Appendix A.

TSP and in a Lin-Kernighan heuristic for the minimum weight perfect matching problem.

1.1 The two problems

The two concrete optimization problems considered in this thesis are the Traveling Salesman Problem and the minimum weight perfect matching problem.

The Traveling Salesman Problem is perhaps the most famous combinatorial optimization problem of them all. Let $G = (V, E)$ be a complete undirected graph with edge weight function ω. Throughout this thesis, n denotes the number of vertices in G. A *tour* T of G is a cycle in G visiting all the nodes exactly once. Each tour thus specifies an ordering σ of the vertices $1, \ldots, n$. The *length* of a tour is the sum of the weights of its edges: $\omega(T) = \omega(\sigma_n, \sigma_1) + \sum_{i=1}^{n-1} \omega(\sigma_i, \sigma_{i+1})$. The Traveling Salesman Problem asks us to find a shortest tour for the graph. See Lawler *et al.* [58] for an encyclopaedic account of the TSP up to 1985.

Given an arbitrary graph G, a set M of edges of G is a *matching* if each vertex of G appears in at most one member of M. Matching M is *perfect* if each vertex of G appears in *exactly one* member of M. As with the TSP, we are interested in the case where G is a complete undirected weighted graph. Under those conditions, G has a perfect matching if and only if n is even. The minimum weight perfect matching problem asks us to find a perfect matching with least total cost, *i.e.*, minimize $\sum_{e \in M} \omega(e)$.

An important subclass of instances are those with geometric structure. Each vertex has a set of coordinates, and the distance between two vertices is some metric on that coordinate space. For example, in a two-dimensional Euclidean instance each vertex has x and y coordinates and the distance between two vertices is the ordinary Euclidean distance. Commonly used metrics include [73]:

- the Euclidean metric, or L_2 metric: $\omega(u, v) = \sqrt{(u_x - v_x)^2 + (u_y - v_y)^2}$,
- the Manhattan metric, or L_1 metric: $\omega(u, v) = |u_x - v_x| + |u_y - v_y|$,
- the maxnorm metric, or L_∞ metric: $\omega(u, v) = \max(|u_x - v_x|, |u_y - v_y|)$.

To improve the comparability of results, researchers often force distance functions to be integral, *e.g.*, by rounding to the nearest integer, or rounding up to the next larger integer. (Simply rounding to the nearest integer does not always preserve the triangle inequality property of a metric, while rounding upward does.) We return to these considerations in Section 5.8.

Theory tells us much about the worst-case nature of the TSP [48]. For instance, the decision-problem variant of the TSP is NP-complete [50, 35], even for Euclidean instances [48]. For the general TSP, it is hard to approximate an optimal solution to within a factor of $(1 + 1/c)$ if c is given as an arbitrary parameter [79]. This is so even when the triangle inequality is assumed [70]. Recent work shows the Euclidean TSP can be approximated to within a factor of $(1+1/c)$ with $O(n(\log n)^{O(c)})$ running time [9, 10], although to date no experiments with this scheme have been reported. Exact polytime algorithms are known for several special cases of the TSP [36, 21].

As for weighted perfect matching, it can be solved with general weighted matching algorithms [30, 68]. These can be coded to run in $O(n^3)$ time for general instances [34] and in $O(n^{2.5} \log^4 n)$ time for geometric instances [84]. Asymptotically faster algorithms computing approximately optimal matchings are also known [83, 71]. Asymptotic running times for approximate algorithms can also be reduced if the input is known to be a geometric instance [43, 10]. High performance implementations of general weighted matching are also now available [8, 27]. We use Cook and Rohe's Blossom IV implementation to produce optimal perfect matchings against which the output of the Lin-Kernighan heuristic can be compared.

1.2 Local search

The Lin-Kernighan strategy is itself part of the larger optimization paradigm known as *local search* [1]. A local search algorithm takes as input a feasible but possibly suboptimal solution to the problem. It repeatedly tries to find a better solution than the current one by modifying the current one in some way. If a better solution is found then the process starts again, with the newly found solution being the starting point for the new search. Otherwise, the process terminates, and the current solution is returned as the result. Local search is also known as hill climbing. For minimization problems, hill *descending* would perhaps be a more appropriate term. See Johnson and McGeoch [46] for an in-depth account of local search for the TSP.

The set of solutions reachable from a given solution in one step is called the *neighbourhood* of that solution. Local search heuristics are characterized by the neighbourhoods they define. A large neighbourhood may give better answers but increases search time.

A small neighbourhood may give poor answers, but gives them very quickly.

A simple example of a local search heuristic is the 2-Opt heuristic for the TSP [32, 28]. Given a current tour T, all tours differing from T in exactly two edges are examined. If one these other tours has lower cost, then we take it as our new tour and begin again. The neighbourhood for 2-Opt is the set of all 2-changes, *i.e.*, the exchange of two old edges for two new edges.

The idea behind 2-Opt extends to larger neighbourhoods. For example, the 3-change neighbourhood, where up to three edges may be exchanged at a time, defines the 3-Opt heuristic. 3-Opt usually runs longer than 2-Opt, but it also produces better tours [60].

In the extreme, one can use the k-change neighbourhood, where up to k edges are exchanged at a time. But for even modest k, searching the k-change neighbourhood takes too much time, $\Theta(n^k)$ steps. The Lin-Kernighan heuristic is also known as "variable-depth-Opt" because it can find improving exchanges involving many edges, theoretically up to $\Omega(n)$ at a time. It uses several techniques to reduce the time required to search its neighbourhood. In the worst case the time to search the Lin-Kernighan neighbourhood is at most $O(n^7)$. In practice the *total* running time of the heuristic appears to grow subquadratically [46].

Two of the techniques used by Lin-Kernighan to keep running times low are *the cumulative gain criterion* and *greedy selection*. They are discussed in detail in Chapter 2. Both use a utility function, the cumulative gain, to guide and prune the search. Cluster compensation is an adjustment to the cumulative gain function in certain contexts. It is *efficient* cluster compensation because we also describe implementation techniques to compute the adjustment very quickly, in small constant time.

Cluster compensation is orthogonal to the many other techniques used to make Lin-Kernighan fast and effective in practice. In particular, efficient cluster compensation does not preclude or require any of the other techniques. Furthermore, the impact of cluster compensation on legacy Lin-Kernighan code is small; on the order of 10 lines must be changed. Of course, efficient cluster compensation requires new code as well. See Appendix B for the availability of our implementation. The code implementing cluster compensation itself is well separated from the code implementing the rest of the heuristic.

1.3 Clusteredness

Intuitively, an instance is clustered if its vertices can be grouped so that the distances between vertices in distinct groups are large in comparison to the distances between near neighbour vertices within a group. Clusteredness is a characteristic of the distance function ω. We do not offer a precise definition of clusteredness, but the arguments in Section 3.3.1 suggest the following quantitative yardstick. Let $T = (V, E_T)$ be a minimum spanning tree for G. Removing the longest edges in T breaks it into several components, partitioning the vertices of G into clusters. Let γ be the ratio of the length of the longest edge in T to the length of the median length edge in T. When γ is large, the minimum distance between points in distinct clusters is large in comparison to the minimum distance between points within any single cluster. When γ is small, the gaps between clusters are relatively small.

The γ function on graphs is only a rough measure of clusteredness. It and its obvious variations can be easily fooled by a clever adversary. We therefore avoid defining clusteredness in quantitative terms and instead rely on the reader's visual intuition. Many of the instances used in this thesis are geometric, and we depict them in Appendix D. For example, instance `fl3795` (Figure D.2(d)) is very sharply clustered, whereas instance `fl4461` (Figure D.2(e)) is relatively uniform, and instance `pr1002` (Figure D.1(f)) is only moderately clustered.

1.4 Related work

We direct the reader to Johnson and McGeoch's comprehensive survey of local search techniques for the TSP [46]. There the Lin-Kernighan heuristic is compared to basic polytime construction algorithms, to weaker local search heuristics such as 2-Opt and 3-Opt, and to more general local search metaheuristics such as general tabu search, genetic algorithms, and simulated annealing. Lin-Kernighan and Iterated Lin-Kernighan (see Section 4.9) are found to occupy a large segment of the undominated time-quality tradeoff curve; they provide the best quality output from among all algorithms running within their time bounds. Furthermore, together they occupy the higher-quality end of that tradeoff curve, while still running in observed subquadratic time.

1.4.1 Decomposition heuristics

Decomposition heuristics are very natural algorithms to try, especially when considering clustered inputs. The idea is to break up the instance into subgroups, solve the problem on the subgroups, and finally patch the pieces together. Theoretical work in this vein for the TSP includes Karp's average case results for partitioning schemes in the plane [51], and culminates with Arora's polynomial time approximation scheme for Euclidean TSP and other geometric problems [9, 10]. The latter work describes an algorithm that partitions space into hyperboxes so that near-optimal tours cross the boundaries between hyperboxes only a small number of times. Dynamic programming is used to find good tour fragments at the base cases, and to combine tour fragments when unraveling the recursion.

Decomposition heuristics work well. However, in comparison with the very fast running times of high quality Lin-Kernighan implementations, they can suffer from the difficulty in the time required to find a good partition of the vertices, and to recombine the subsolutions. The best practical work appears to be due to Rohe [76, 75], who implemented TSP and minimum weight perfect matching upper and lower bounding heuristics for very large instances, up to 18 million cities. Subproblems were chosen as random contiguous subsets, and were worked on in parallel. Iterated Lin-Kernighan was used on the subproblems, and the results were patched using a variant of k-opt. Runs of up to several weeks long were used. Cluster compensation is orthogonal to these techniques, and may be applied in the Lin-Kernighan runs on the subproblems.

1.4.2 Adjusting the distance function

We are aware of only one other Lin-Kernighan-based heuristic that changes the cost function during execution. Gu [38] (see also Section 6.3 in [46]) implements a variant of what one might call "Iterated 2-Opt", with the intent of improving the quality of output at the expense of longer running times. Ahead of time one computes the average distance between vertices, ω^*, and decides on α, the number of iterations of 2-Opt to be run. Iteration i of 2-Opt is run on an instance with distance function ω_i defined as follows:

$$\omega_i(u,v) = \begin{cases} \omega^* + (\omega(u,v) - \omega^*)^{\alpha-i+1} & \text{if } \omega(u,v) > \omega^* \\ \omega^* - (\omega^* - \omega(u,v))^{\alpha-i+1} & \text{otherwise} \end{cases}$$

The tour found at the end of one 2-Opt run is used as the start tour for the next iteration, *i.e.*, the next 2-Opt run. In the first few iterations the distance function is smooth and therefore not sharply clustered. With more iterations ω_i approaches ω, with equality on the last iteration. Gu used this strategy only on random distance matrix instances with $n \leq 100$, and with 2-Opt and Or-opt (a 2-Opt variant).

However, changing the whole edge weight function in this way conflicts with many of the key speedup techniques required to make Lin-Kernighan run fast in practice on larger instances. For example, the candidate sets (near neighbour lists) as described in Section 4.5 would be invalidated with each change in the underlying distance function. They would therefore have to be recomputed before each new iteration.

Let us compare Gu's "search space smoothing" technique with efficient cluster compensation. Our goals for cluster compensation are complementary; Gu aims to improve quality while increasing running times, while cluster compensation is designed to reduce running times while maintaining quality. Cluster compensation does not change the underlying distance function, so it does not impose as stiff a runtime penalty. Furthermore, cluster compensation is designed specifically for the Lin-Kernighan heuristic, taking advantage of the heuristic's use of the cumulative gain function and its long search sequences. Finally, cluster compensation is useful even with only single iteration runs of Lin-Kernighan.

1.4.3 The Kernighan-Lin heuristic

The general Lin-Kernighan heuristic is an outgrowth of the Kernighan-Lin heuristic for graph partitioning [61, 52]. The Kernighan-Lin graph partitioning algorithm is the basis for practical algorithms central to VLSI design automation [29, 59, 42]. Lin-Kernighan differs from Kernighan-Lin in two ways relevant to our work. First, Lin-Kernighan employs the cumulative gain criterion (see Chapter 2), whereas Kernighan-Lin does not. Second, the intermediate structures within the Kernighan-Lin heuristic are always feasible, so there are no hidden closing-up costs that must be taken into account. As we shall see in Chapter 3, cluster compensation is a way of adjusting the cumulative gain criterion with the aim of partially accounting for closing up costs. The Kernighan-Lin heuristic is therefore outside the scope of the work in this thesis.

1.5 Outline of the thesis

The remainder of the thesis is organized as follows. Chapter 2 describes the general Lin-Kernighan strategy. Using clues from worst-case theoretical results, it also suggests reasons for the slow running times on clustered inputs. Chapter 3 introduces the proposed partial remedy, cluster compensation, and describes efficient algorithms for the technique.

Chapter 4 describes how the general Lin-Kernighan strategy is applied to the Traveling Salesman Problem. The emphasis is on the many details and techniques developed since the original Lin and Kernighan paper [61] appeared 25 years ago. Those techniques are incorporated into the high quality implementations of today because they extend the heuristic's scalability without significantly harming its ability to find very good tours. Chapter 5 describes the experimental methodology: how the experiments were designed, a description of the test bed, what was measured, and how the measurements are presented and evaluated. It is written in the context of Lin-Kernighan for the TSP, but almost all the considerations carry over to Lin-Kernighan for minimum weight perfect matching. Chapter 6 presents the results for most of the experiments for the TSP. It compares the quality of tours, overall running times, and probe and move depth profiles. The probe and move depth profiles are measures of the overall work performed by the optimization phase of the heuristic, and the progress it makes during that time, respectively. They corroborate the running time and quality of output findings while being independent of programmer skill and platform speed.

Chapter 7 does for the minimum weight perfect matching problem what Chapters 4, 5, and 6 do for the Traveling Salesman Problem. That is, it describes how the Lin-Kernighan strategy can be applied to the minimum weight perfect matching problem, how experiments were run for that problem, and the results of those experiments. The Lin-Kernighan heuristics for the two problems are similar enough that much of their implementations are shared, as are the experimental methodology and test data. The experimental outcomes are quite similar as well.

Naturally, an experimental study cannot hope to test all possible inputs. Chapter 8 describes several algorithms that transform one instance into many randomly generated similar instances. The idea is to use a structural fingerprint of the input instance as the basis for a random distribution from which new instances can be drawn. The structural fingerprints and the distributions built upon them are chosen to either emphasize or downplay the clustering features affecting the running time of the Lin-Kernighan heuristic. This approach does three things for us. First, it lets us test the validity of our intuition about the factors affecting the performance of the heuristic. Second, it gives us a general way of creating an infinite variety of test instances. By doing this we can hope that our results will not be skewed by the idiosyncrasies of any finite sample of instances. Third, it allows a practitioner to generate a test bed having the characteristics that might recur in their own application domain. Chapter 9 reports the results of experiments on data generated by the algorithms of Chapter 8.

Chapter 10 summarizes the thesis and suggests further work.

This dissertation is primarily an experimental work. Theorems and lemmas are presented along the way to motivate ideas and to prove the correctness and efficiency of the algorithms involved in cluster compensation. Appendix A discusses a worst-case result, that the problem solved by the Lin-Kernighan heuristic using cluster compensation is PLS-complete. It is a slight modification of Papadimitriou's argument showing that the Lin-Kernighan heuristic solves a problem that is PLS-complete [67]. Papadimitriou's argument is very robust, making our job quite easy.

Appendix B briefly describes the implementation used for the experiments performed in this thesis, and where to get it. Appendix C describes the computing platform used for the experiments. Appendix D depicts the main geometric instances used in this thesis.

Chapter 2

The Lin-Kernighan heuristic

Lin and Kernighan defined both a general strategy for optimization and its application to the symmetric Traveling Salesman Problem (TSP)[61]. Efficient cluster compensation applies to a large subclass of algorithms described by their general strategy. This subclass includes both their algorithm for the TSP and the approximation algorithm for minimum weight perfect matching described in Chapter 7.

This chapter describes the subclass to which efficient cluster compensation can be applied. Chapter 4 gives more detail about the original Lin-Kernighan algorithm for the symmetric Traveling Salesman Problem (TSP). Chapter 7 describes a Lin-Kernighan heuristic for weighted perfect matching.

2.1 The generic problem and strategy

Lin-Kernighan heuristics approximately solve optimization problems of the following form [61]: From a space S find a subset T satisfying some feasibility criteria and minimizing some cost function.

Two main strategies are used. Like many other heuristics, at a high level Lin-Kernighan heuristics use local search [1]. Given a feasible structure T, the heuristic tries to find a feasible structure T' with lower cost by modifying T. If the search succeeds, then the new structure T' takes the place of the original structure T and the step is repeated. This continues until no better T' can be found. The set of feasible structures examined from starting point T during each step is called the *neighbourhood* of T. The process is depicted in Algorithm 1.

Algorithm 1 Local search

Require: T is a feasible structure

Ensure: T is a locally optimal feasible structure

```
repeat
  Search the neighbourhood of T
  if We find a lower cost feasible structure T' then
    T := T'
  end if
until No lower cost feasible T' is found
```

2.1.1 Sequential changes and the cumulative gain criterion

The lower level strategy takes the following shape. Let us assume that all feasible sets of interest have the same size. If the current solution T is not optimal, then it differs from some optimal solution T_{opt} by k elements; we can evolve T into T_{opt} by removing k elements and adding k elements. Such a modification to a set is called a k-change. Considering all feasible sets differing from T by k elements is expensive even for small k; such a neighbourhood is too large. Even worse, it is unlikely that we know the value of k.

At this lower level, Lin-Kernighan heuristics distinguish themselves by applying the following two ideas. First, the k-change can be built sequentially, swapping one pair of elements at a time. (The intermediate structures need not be feasible sets.) Second, we maintain the *cumulative gain*, a running score of the improvement over the cost of T via the current sequence of moves. We use the cumulative gain to reduce the number of pairs considered for swapping.

Let us write g_i for the gain made by the i'th swap, *i.e.*, the amount by which the i'th swap decreases the cost of the current structure. The cumulative gain is the sum of each of the gains made by swapping a pair of elements: the cumulative gain after j swaps is $cum\_gain(2j) = \sum_{i=1}^{j} g_i$. We use the following rule, the *cumulative gain criterion*, to bound the depth of the search:

Criterion 2.1 *Extend the sequential change by one more swap only if the cumulative gain will remain positive.*

That is, after j swaps we add a $j+1$'st swap with gain g_{j+1} only if $cum\_gain(2j) + g_{j+1} = cum\_gain(2(j+1)) > 0$. (The new swap may actually increase the cost of the structure, *i.e.*, we can accept swaps with negative gain ($g_{j+1} < 0$). This allows the search to escape some minima that would otherwise trap it.) The pruning action provided by the cumulative gain criterion is central to the success of the heuristic.

2.1.2 The cumulative gain criterion does no harm

The cumulative gain criterion is reasonable because it prunes the search without discarding any improving sequential k-changes. This is a consequence of the following fact about sums: If a sum is positive, then its terms may be cyclically shifted so that all the partial sums are positive. That is, if $\sum_{i=0}^{n-1} a_i > 0$ then there is an integer s so that $\forall 1 \leq j < n$, $\sum_{i=0}^{j} a_{(i+s) \bmod n} > 0$.

(The proof is simple once one draws a picture. Let M^- be the least partial sum in the original list, and let m be the largest index at which that partial sum occurs (hence $M^- = \sum_{i=0}^{m} a_i$), and let $M^+ = \sum_{i=m+1}^{n} a_i$ be the remainder of the sum. Considered in isolation, all the partial sums in $M^+ = \sum_{i=m+1}^{n} a_i$ are positive. Choose $s = n - m$, shifting the terms making up the least partial sum to the end of the list. All the partial sums in the new list are positive since the least partial sum in the new list occurs at the end, and it is equal to the total sum, $M^+ + M^-$, which we know is positive.)

The cumulative gain criterion embodies the following interpretation of this fact about sums. A single search is trying to construct an unknown set of k swaps with positive total gain by building sequentially from the first swap forward. If during any step the cumulative gain becomes negative, then we have taken the wrong path. We should stop now and instead hope to find the same improving set of swaps by beginning the search with a different swap. The fact about sums guarantees that if there is an improving set of swaps, then there is a build sequence of those swaps which always maintains a positive cumulative gain.

2.1.3 Greedy choice

At each step in the build sequence, many swaps might satisfy the cumulative gain criterion. The natural choice is the greedy one:

Criterion 2.2 *Choose the swap with maximum gain. In case no improving swaps can be found, choose the one with least loss. Symbolically, maximize* g_{i+1}.

This greedy strategy carries with it the following implicit interpretation on the cumulative gain:

Interpretation 2.3 *The cumulative gain is a measure of the promise of the magnitude of improvements to be found along the current search path.*

This interpretation becomes pivotal in trying to improve the heuristic, as we shall see in Section 3.1.

We eventually come to a point where the swap sequence can go no further: all allowable swaps force the cumulative gain to become negative. We are free to choose any of the feasible structures evolved from T using some prefix of the now-terminated sequence of swaps, perhaps together with one of the swaps previously eliminated by Criterion 2.2. We are greedy again and choose T' to be the one with least cost; it will be the basis for the next search.

2.1.4 Feasibility rules

Each optimization problem has its own feasibility rules defining the set of valid outputs. The intermediate structures examined during a search need not be feasible. But it helps if they are "not too infeasible", so that the search space is reduced, and so that the algorithm can easily find a sequence of swaps repairing it to feasibility. Such considerations are problem-dependent. We shall see examples in Chapters 4 and 7.

2.1.5 Tabu rules

It might be easy for the heuristic to get caught in an endless cycle of swaps, never terminating its search. To prevent this from happening, the heuristic employs *tabu rules*: restrictions on which elements can participate in a swap. A common tabu rule is to never let an element participate in more than one swap per sequence, *i.e.*, never delete an added element and never add a deleted element. This rule bounds the length of any single sequence of swaps by the size of the original structure T. In Lin-Kernighan heuristics the tabu rules act as a safety, and not as a primary guide for the search [61, 67].

2.1.6 Limited backtracking and candidate sets

Of course, the set of structures discovered during the construction of a sequence of swaps will not always contain a feasible structure with lower cost. In that case we would like to examine different sequences of swaps. Lin and Kernighan suggest using limited backtracking:

Feature 2.4 *While no lower cost feasible structure is found, examine greedy swap sequences using all possible choices for the first few (e.g., two) swaps.*

The alternatives allowed by this rule are subject to the cumulative gain criterion, and the feasibility and tabu rules.

There is usually a tradeoff between richness of the search space and the running time of the search. Backtracking over a greater depth forces longer running times; backtracking over a lesser depth confines the search space.

Instead of just reducing the backtracking depth, one may confine the search to more likely territory by using *candidate sets* [61, 74, 46]. Considering the nature of the optimization problem, the structure of the input, and possibly user-supplied parameters, we *a priori* define the candidate set for element $e \in S$ to be those elements e' that are likely to improve a structure locally when swapped with e. Then we can refine the backtracking rule as follows:

Feature 2.5 *While no lower cost feasible structure is found, examine greedy swap sequences using all possible choices for the first few swaps; each swap in the sequence is the replacement of an element by some member of its associated candidate set.*

The sizes of the candidate sets are a primary factor in the running time of the heuristic, as the main work of the algorithm is to scan the candidate sets repeatedly for feasible and non-tabu swaps. Therefore, the user is usually given a degree of control over the size of the candidate sets, a point to which we shall return in Chapters 4 and 7.

2.1.7 Features in perspective

All these features of the generic heuristic were described by Lin and Kernighan [61]. Even at this level of abstraction it is easy to lose the forest for the trees. To know where to focus our efforts to improve the heuristic, it is useful to put its features in perspective.

The primary features of Lin-Kernighan are the building of the k-change one element at a time, and the use of the cumulative gain criterion. The rest can be viewed as tweaks for practicality or correctness. Choosing swaps greedily is less important although it is a natural choice. The feasibility rules stem from the optimization problem itself, and cannot be changed. The tabu rules are simply a safety against non-termination. In the case of Lin-Kernighan for the TSP, the current hypothesis is that tinkering with the tabu rules does not change the basic behaviour of the algorithm [46, 67].

2.2 A more specific problem and strategy

Efficient cluster compensation applies to Lin-Kernighan heuristics that specialize the above strategy for use in a particular setting. This section describes this specialized setting and strategy. The original Lin-Kernighan algorithm for the TSP uses this specialized strategy, as does the algorithm for weighted matching described in Chapter 7.

Given a complete undirected graph $G = (V, E)$ with non-negative edge cost function ω, we want to find a low-cost feasible set of edges. In this setting the space S is just the set of all edges E. In the case of the Traveling Salesman Problem, the feasible structures are the tours of G. In the case of weighted perfect matching, the feasible sets are the perfect matchings of G. Let n be the number of vertices in G, *i.e.*, $n = |V|$.

To transform feasible set T into T' we use a k-change of edges: the removal of k edges and the addition of k edges. As before, the swaps are done sequentially. More importantly, a completed sequence of swapped edges forms an even-length cycle in G alternating between removed edges and added edges. A completed k-change can therefore be encoded as a sequence of $2k$ vertices of the original graph, $t_1, \ldots, t_{2k}$. Odd-numbered edges $\{e_{2i-1} = (t_{2i-1}, t_{2i})\}_{i=1}^{k}$ are removed edges, and even-numbered edges $\{e_{2i} = (t_{2i}, t_{2i+1})\}_{i=1}^{k-1}$ and $e_{2k} = (t_{2k}, t_1)$ are added edges. Figure 2.1 shows how a particular 4-change on a tour is encoded. Figure 2.1(a) shows the tour before the 4-change is applied; vertices $t_1, \ldots, t_8$ are marked. Figure 2.1(b) shows the new tour formed by applying the 4-change with t vertices as marked in part (a). Figure 2.1(c) shows the edges that differ between the two tours of (a) and (b); odd-numbered edges e_1, e_3, e_5 and e_7 are edges removed from tour (a), while even-numbered edges e_2, e_4, e_6 and e_8 are edges added to tour (a). Figure 2.1(d) unravels (c) to show t_1 through t_8 as a sequence, with added edges drawn above the sequence, and deleted edges drawn below the sequence. Both (c) and (d) show the edges as directed, indicating a natural orientation associated with building the 4-change. The search for an improving k-change begins with the deletion of edge $e_1 = (t_1, t_2)$, and is structured around extending the t sequence of vertices.

Since the edge cost function ω is non-negative, the cumulative gain criterion need only be checked when considering adding an edge, *i.e.*, when considering extending the

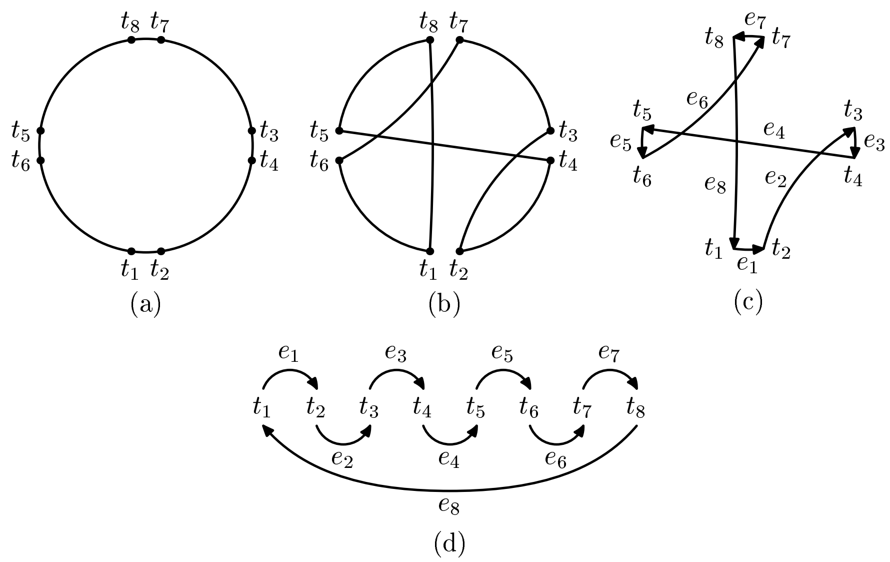


Figure 2.1: Encoding a sample 4-change on a tour. Vertices $t_1, \ldots, t_8$ are shown, as are the associated removed edges e_1, e_3, e_5, e_7, and the associated added edges e_2, e_4, e_6, e_8.

t sequence with t_{2i+1} we require $\mathit{cum\_gain}(2i+1) > 0$. Furthermore, given a choice for t_{2i+1}, the feasibility constraints make it easy to select the next edge; vertex t_{2i+2} is either uniquely determined or drawn from a small set. So we take advantage of this easy lookahead and consider candidates for t_{2i+1} and t_{2i+2} as pairs, choosing the pair maximizing the new cumulative gain $\mathit{cum\_gain}(2i+2)$. This is not much more work, and this extra lookahead leads to more intelligent choices [61]. Thus the t sequence begins as t_1, t_2 and then grows by pairs, t_{2i+1}, t_{2i+2}.

Note that the first edge in the sequence is a deleted edge and it is not immediately paired with an added edge. Afterward, edges are appended to the sequence in pairs. To complete the alternating cycle, we need to append an added edge (t_{2i}, t_1). These specifics of constructing the alternating cycle force a slight conceptual shift from the general setting in which all swaps, even the first, are atomic.

2.2.1 The concrete cumulative gain criterion

We now can give a more concrete definition of the cumulative gain function.

Definition 2.6 *The cumulative gain function is defined as*

$$\mathit{cum\_gain}(j) = \sum_{1 \le l < j,\, l \text{ odd}} \omega(t_l, t_{l+1}) - \sum_{1 < l < j,\, l \text{ even}} \omega(t_l, t_{l+1}).$$

In these terms, the basic cumulative gain criterion from above requires that $cum\_gain(j) > 0$ for all $j \geq 2$, and need only be checked for odd j.

Criterion 2.7 (Basic Cumulative Gain [61]) *Maintain the invariant* $cum\_gain(j) > 0$.

While constructing a particular $\{t_j\}$ sequence we may have the good fortune of finding some way of closing up the alternating cycle with a net improvement of, say, *net_gain* units. (Value *net_gain* necessarily includes the closing up cost of adding edge (t_{2i}, t_1). That is, $net\_gain = cum\_gain(2i) - \omega(t_{2i}, t_1)$.) We continue the search by extending the $\{t_j\}$ sequence beyond t_{2i} in hope of finding even better net improvements. However, Lin and Kernighan suggest tightening the cumulative gain criterion as follows.

Criterion 2.8 (Cumulative Gain [61]) *Maintain the invariant*

$$cum\_gain(j) > best\_net\_gain(j)$$

where $best\_net\_gain(j)$ *is the best net improvement provided by some alternating cycle discovered during the construction of* $t_1, \ldots, t_j$.

This tighter version of the criterion is more common—the basic version is used only rarely.

Note that the cumulative gain criterion deliberately ignores the immediate cost of closing up the cycle with edge (t_{2i}, t_1) [61]. That is, we do not require $cum\_gain(2i) - \omega(t_{2i}, t_1) > best\_net\_gain(2i)$. We ignore the immediate closing up cost because it is too pessimistic [61]. We might be able to extend the t sequence by a few more vertices and find a good way to close up the cycle, a way with net loss much less than $\omega(t_{2i}, t_1)$. In fact, we might get lucky and find an alternating path back to t_1 with a big net gain.

2.2.2 The concrete greedy selection criterion

In this concrete setting, the greedy selection criterion is cast as follows. With vertices $t_1, \ldots, t_{2i}$ given, we may have a choice among feasible non-tabu candidate pairs for vertices t_{2i+1}, t_{2i+2} satisfying the cumulative gain criterion.

Criterion 2.9 (Greedy Selection [61]) *We choose to extend the* t *sequence with the candidate pair* t_{2i+1}, t_{2i+2} *that maximizes* $cum\_gain(2i + 2)$.

Since $cum\_gain(2i+2) = cum\_gain(2i) - \omega(t_{2i}, t_{2i+1}) + \omega(t_{2i+1}, t_{2i+2})$, and $cum\_gain(2i)$ is already determined by $t_1, \ldots, t_{2i}$, this is the same as selecting the candidate pairs maximizing $-\omega(t_{2i}, t_{2i+1}) + \omega(t_{2i+1}, t_{2i+2})$.

2.3 Summary

This chapter gives a high level description of both the abstract Lin-Kernighan strategy and a slightly more concrete version of that strategy. The concrete strategy is based on three ingredients: constructing improving alternating cycles, using a cumulative gain criterion to prune searches, and using a greedy selection criterion to choose among alternative moves.

With a few extra details, the concrete strategy can be used to build heuristics for many optimization problems. The original Lin-Kernighan heuristic for the Traveling Salesman Problem uses the concrete strategy, and is described in Chapter 4. Chapter 7 describes how the concrete strategy can be applied to minimum weight perfect matching.

Efficient cluster compensation, the technique motivated and described in the next chapter, applies to any Lin-Kernighan heuristic using the concrete strategy.

Chapter 3

Efficient cluster compensation

The Lin-Kernighan heuristic and its derivatives are champion heuristics in some domains [44, 46]. That is, they provide better solutions for a given amount of computational effort, especially when looking for nearly optimal solutions. However, clustered instances are more difficult for the Lin-Kernighan heuristic, producing longer running times than for uniform instances [46].

This chapter motivates and describes *efficient cluster compensation*, a technique aimed at reducing the performance degradation caused by clustering in the input, but without adversely affecting the quality of the output. This chapter describes the technique. Later chapters show how it affects the heuristic, both in practice and in theory.

3.1 How Lin-Kernighan stumbles on clustered instances

The Lin-Kernighan heuristic for the TSP performs remarkably well on most instances because the cumulative gain criterion effectively prunes the search space. Furthermore, choosing to ignore the immediate closing up cost is important in the discovery of high quality solutions.

However, choosing to ignore immediate closing up costs also makes the heuristic vulnerable to long running times. Papadimitriou exploits this vulnerability in his proof that the Lin-Kernighan heuristic for the TSP solves a problem that is PLS-complete [67]. Roughly speaking, PLS is the class of problems solved by local search algorithms in which each step takes time polynomially bounded in the size of the input. PLS stands for Polynomial Local Search. Just as with NP-completeness, a theory of reductions among PLS problems may be used to define the class of PLS-complete problems [49, 85]. It is conjectured that a search analog of NP properly includes PLS, and that PLS properly includes a search analog of P. One consequence of Papadimitriou's theorem is that one can construct a family of graphs that force the Lin-Kernighan heuristic, with certain relaxed tabu rules, to perform an exponential number of steps.

Papdimitriou's proof uses a constructed graph with a small number of *bait edges* whose sole purpose are to pump the cumulative gain function to optimistically large values. This large cumulative gain, together with limited backtracking, force a deep search of alternative paths in the constructed graph.

In fact, the graph is constructed so that all possible paths back to t_1 must end in an edge at least as large as the bait edges removed at the beginning of the search. In this case the heuristic is fooled into believing that a large improvement is in the works, and spends much effort looking for one.

We will revisit this graph in Appendix A, where we adjust Papadimitriou's proof to show that Lin-Kernighan solves a PLS-complete problem, even if cluster compensation is used.

Experiments show that the Lin-Kernighan heuristic for the TSP spends much more time on clustered instances than it does on relatively uniform instances [46]. We hypothesize that the main problem presented by clustered instances is that edges crossing relatively large inter-cluster gaps serve as bait edges. When removing such a large *bridge* edge, the cumulative gain grows optimistically large. This large cumulative gain "budget" allows the heuristic to spend much effort wandering in the newly visited cluster, with little or no incentive to return to the originating cluster containing start vertex t_1. The option of returning to the originating cluster usually involves adding to the structure some necessarily long edge from the current cluster back to the originating cluster. The cost of adding such a bridge edge is large, and this option is likely to be repeatedly discarded by the greedy criterion the heuristic applies in choosing the next pair of vertices t_{2i+1} and t_{2i+2}.

Of course, we might get extremely lucky by finding a path back into the originating cluster that *removes* a long bridge edge. However, as we find increasingly better solutions, the likelihood of encountering this very lucky case diminishes greatly over time. That is, a decent solution typically contains few edges that bridge clusters. Furthermore, in the case of the TSP, solutions are tours and hence connected subgraphs; that is, bridge edges must always exist.

3.2 Cluster compensation

The lesson of the previous section is the following: both the cumulative gain criterion and the greedy selection criterion ignore immediate and future closing up costs. Although this allows the heuristic much freedom to explore, the heuristic can be tricked into long fruitless searches when it removes an edge bridging a large inter-cluster gap.

We would like to reduce the baiting impact of bridge edges. The key lies in the implicit use of the cumulative gain as a measure of the promise of the magnitude of the improvements likely to be found. We want to make the removal of inter-cluster edges appear less promising. As it stands, the heuristic pays attention only to relatively local factors in the selection of t_{2i+1} (a candidate on vertex t_{2i}'s list) and t_{2i+2} (forced by feasibility). The heuristic does not know whether the edge it is considering for removal, (t_{2i+1}, t_{2i+2}), is a long inter-cluster bridge edge, or whether it is a long edge over densely populated territory. If the edge is a long inter-cluster bridge then we want to discourage its removal. If the edge is a long edge over a densely populated territory, then it is a good candidate for removal and so we don't want to discourage its removal.

We propose to refine the heuristic by giving both the cumulative gain and the greedy selection criteria knowledge of the global cluster structure of the instance. Cluster compensation is the use of the adjusted cumulative gain and greedy selection criteria in a Lin-Kernighan heuristic in place of their standard versions. The standard versions are Criteria 2.8 and 2.9. The adjusted versions are Criteria 3.3 and 3.4, given below.

3.2.1 Cluster distance

The knowledge of the cluster structure is given in the form of the cluster distance between two vertices. The cluster distance between two vertices is just the length of the "longest hop" one *must* take in traveling from one vertex to the other.

Definition 3.1 (Cluster Distance) *Let the bottleneck cost of path P be the weight of a heaviest edge on P. Then the* cluster distance *between vertices u and v, written as $c_G(u, v)$, is the minimum bottleneck cost of any path from u to v in graph G.*

When the underlying graph G is understood, we shall drop the subscript and simply write $c(u, v)$.

3.2.2 Cluster distance as an estimate of future closing up costs

The cluster distance between two endpoints t_1 and t_{2i} can be viewed as an approximate lower bound on all possible future closing up costs.

Suppose vertices t_1 through t_{2i} are fixed, and that we are trying to complete an alternating k-cycle by appending vertices t_{2i+1} through t_{2k} to our list. The unknown portion t_{2i+1} through t_{2k} encodes an odd-length alternating path beginning with added edge $e_{2i} = (t_{2i}, t_{2i+1})$, removed edge $e_{2i+1} = (t_{2i}, t_{2i+1})$, and on through added edge $e_{2k} = (t_{2k}, t_1)$. The cluster distance between t_{2i} and t_1 is a lower bound on the length of the longest hop on this path. The net closing up cost of this yet to be discovered path is

$$\omega(e_{2i}) - \omega(e_{2i+1}) + \cdots + \omega(e_{2k-2}) - \omega(e_{2k-1}) + \omega(e_{2k}).$$

There are $k-i+1$ positive terms and $k-i$ negative terms in this sum. Let us temporarily make the unreasonable assumption that the largest term in magnitude is the positive term $\omega(e_{2L})$ and that all the other terms cancel. It follows that the net closing up cost of this path is $\omega(e_{2L})$, and hence that the cluster distance from t_{2i} to t_1 is a lower bound on the future closing up cost.

Of course, the terms will not usually cancel as assumed. So the cluster distance is not an *actual* lower bound on closing up costs. The assumption embodies the point of view that the longest hop from t_{2i} to t_1 is all that matters, that the lengths of edges before and after that biggest hop are noise. But this first-order approximation to future closing up costs motivates the adjustments suggested in the following sections.

3.2.3 Adjusting the cumulative gain criterion

We want to tune the cumulative gain criterion so it does something in between always being pessimistic by counting the immediate closing up costs (*i.e.*, $\mathit{cum\_gain}(2i) - \omega(t_{2i}, t_1) > \mathit{best\_net\_gain}(2i)$) and always being optimistic and never counting future closing up costs (*i.e.*, $\mathit{cum\_gain}(2i) > \mathit{best\_net\_gain}(2i)$). The first alternative is rejected because it reduces the quality of the solutions [61]. The second alternative suffers from fruitless searches caused by bait edges.

We propose discounting the cumulative gain by the cluster distance between the start vertex t_1 and the current end vertex t_{2i}. The basic cumulative gain criterion is adjusted as follows.

Criterion 3.2 (Basic Cluster Compensating Cumulative Gain) *Maintain the invariant*

$$cum\_gain(2i) - c(t_{2i}, t_1) > 0$$

where $c(t_{2i}, t_1)$ *is the cluster distance from* t_{2i} *to* t_1.

The common cumulative gain criterion is adjusted as follows.

Criterion 3.3 (Cluster Compensating Cumulative Gain) *Maintain the invariant*

$$cum\_gain(2i) - c(t_{2i}, t_1) > best\_net\_gain(2i),$$

where $c(t_{2i}, t_1)$ *is the cluster distance from* t_{2i} *to* t_1*, and* $best\_net\_gain(2i)$ *is the maximum net improvement provided by any alternating cycle discovered during the construction of* $t_1, \ldots, t_{2i}$.

The cluster distance $c(u, v)$ is bounded below by 0 as long as ω is non-negative. Since the edge (u, v) is a trivial path, $c(u, v)$ is bounded above by the length of the edge from u to v, namely $\omega(u, v)$. The adjusted cumulative gain criterion thus provides a middle ground between the pessimistic and optimistic alternatives.

3.2.4 Adjusting the greedy selection criterion

The greedy selection criterion helps us decide between many possible choices for the pair t_{2i+1}, t_{2i+2}. Being greedy, we seek to maximize the promise of improvement. Interpretation 2.3 tells us the promise of improvement for the pair t_{2i+1}, t_{2i+2} is just $cum\_gain(2i+2)$. As with the cumulative gain criterion, we introduce some pessimism by discounting by the cluster distance.

Criterion 3.4 (Cluster Compensating Greedy Selection) *We choose to extend the t sequence with the candidate pair* t_{2i+1}, t_{2i+2} *that maximizes* $cum\_gain(2i+2) - c(t_{2i+2}, t_1)$*, where* $c(t_{2i}, t_1)$ *is the cluster distance from* t_{2i} *to* t_1*,*

Again, $cum\_gain(2i + 2) = cum\_gain(2i) - \omega(t_{2i}, t_{2i+1}) + \omega(t_{2i+1}, t_{2i+2})$. With the prefix $t_1, \ldots, t_{2i}$ fixed, $cum\_gain(2i)$ is fixed. So maximizing $cum\_gain(2i+2) - c(t_{2i+2}, t_1)$ means maximizing $-\omega(t_{2i}, t_{2i+1}) + \omega(t_{2i+1}, t_{2i+2}) - c(t_{2i+2}, t_1)$.

3.3 Efficiently computing cluster distance

Cluster compensation attempts to speed up the heuristic by adding code to the inner loop: for each pair of vertices considered for t_{2i+1}, t_{2i+2} we must compute the cluster distance $c(t_{2i+2}, t_1)$. We had better make sure we can compute c quickly. This section describes how, with modest preprocessing time and space, online queries of c can be computed in constant time.

One might be tempted to precompute the cluster distance function and then use a naïve table lookup for online queries. This takes space quadratic in the number of vertices of the instance. But with appropriate supporting data structures, Lin-Kernighan heuristics can effectively be applied to very large instances. For example, certain million-city instances of the TSP can be tackled with linear space (about 250 megabytes) and in under an hour of CPU time [44, 46]. We would like cluster compensation to be effective whenever Lin-Kernighan is effective. In particular, space consumption should be no worse than linear in the number of vertices in the instance.

3.3.1 Minimum spanning trees encode cluster structure

The main ingredient is the observation that a minimum spanning tree for the underlying graph embodies all the cluster structure we require.

Lemma 3.5 *Let $G = (V, E)$ be a connected weighted undirected graph, and let $T = (V, E_T)$ be any minimum spanning tree for G. Then the cluster distance in G between any two vertices u and v is equal to the bottleneck cost of the* unique *path from u to v in T. Symbolically, $c_G(u, v) = c_T(u, v)$.*

Proof: The cluster distance $c_G(u, v)$ is the minimum bottleneck cost over all paths from u to v in G. Since T is a subgraph of G, we immediately have $c_G(u, v) \leq c_T(u, v)$.

We use contradiction to prove $c_G(u, v) \geq c_T(u, v)$. Let P be the unique path in T from u to v, and let b be a longest edge on P; in particular, the weight of b is $c_T(u, v)$. Removing edge b from T would separate T into two connected components: component T_u containing u, and component T_v containing v. Now assume $c_G(u, v) < c_T(u, v)$. Then there is a "sneaky" path P' in G from u to v, all of whose edges have length less than $c_T(u, v)$. Let b' be an edge in P' with one endpoint in T_u and the other endpoint in T_v. Then we can form spanning tree T' from T by replacing b with b'. Tree T has lower cost than does T. But this contradicts the fact that T is a minimum spanning tree for G. Hence our assumption that $c_G(u,v) < c_T(u,v)$ must be wrong, and therefore $c_G(u,v) \geq c_T(u,v)$. ■

The task then becomes computing a minimum spanning tree and then efficiently computing cluster distances in that tree. Implementations of cluster compensation should allow the user to supply their own tree. The user often has special knowledge about the instance allowing them to compute minimum spanning trees more quickly than the more generic algorithms included with the cluster compensation software package. There are many efficient algorithms for computing minimum spanning trees, each tailored for particular instance classes [55].Without special knowledge of the input, Prim's $O(|V|^2)$ algorithm is optimal since we assume our graphs are complete. If we know the instance has geometric structure, then we can use Bentley's k-dimensional search trees together with Prim's algorithm to compute minimum spanning trees in observed $O(|V| \log |V|)$ time [18, 19]. The MST problem is well-solved, so we shall assume we already have a minimum spanning tree in hand.

3.3.2 Computing cluster distance in a tree

Computing cluster distances in a graph is the same as computing cluster distances in a minimum spanning tree for the graph. The main work in efficiently computing cluster distances in any tree is the efficient computation of least common ancestors in a related binary tree.

Definition 3.6 *The least common ancestor of vertices u and v in a rooted tree is the ancestor w of both u and v with the property that any ancestor of both u and v is either w or an ancestor of w. We write $LCA(u,v)$ for the least common ancestor of u and v.*

Computing cluster distances in T amounts to computing least common ancestors in an associated topology tree $Top(T)$. We construct rooted binary tree $Top(T)$ from T by recording the execution history of running Kruskal's minimum spanning tree algorithm on T. Since T is already a minimum spanning tree, this can be viewed as a *reconstruction* of T, with extra output on the side.

The vertices of T label the leaves of $Top(T)$, and the edges of T label the internal nodes of $Top(T)$: $V_{Top(T)} \cong V \cup E_T$. Each subtree of $Top(T)$ corresponds to a connected component of T formed during the execution of Kruskal's algorithm on T: adding edge $e = (u,v)$ during reconstruction of T corresponds to adding internal node R_e to $Top(T)$

with its two children being the current roots of the components containing u and v. The process is laid out in Algorithm 2. The algorithm is simpler than Kruskal's algorithm in some ways because we know T is acyclic, so we know *a priori* that no edges of T will be discarded.

Algorithm 2 Construct $Top(T)$ from T. (Modified Kruskal's minimum spanning tree algorithm.)

Require: $T = (V, E_T)$ is a weighted tree.
Ensure: $\forall u, v \in V$, $u \neq v$, cluster distance $c_T(u, v)$ equals the weight of edge labeling $LCA(u, v)$ in $Top(T)$.
  // Forest F contains the current set of components as rooted trees.
  // We identify a subtree with its root.
  $F := \{\mathsf{Leaf}(v) \mid v \in V\}$ // Initialize with singletons.
  for $e = (u, v)$ in E_T in non-decreasing weight order **do**
    Find $root(u)$, the root of the tree containing $\mathsf{Leaf}(u)$ in forest F.
    Find $root(v)$, the root of the tree containing $\mathsf{Leaf}(v)$ in forest F.
    Form internal node R_e with children $root(u)$ and $root(v)$.
    $F := F \cup \{R_e\} \setminus \{root(u), root(v)\}$
  end for
  $Top(T) := F$

The correctness of Algorithm 2 is given by Lemma 3.7.

Lemma 3.7 (Correctness of Algorithm 2) *For any distinct vertices $u, v \in V$, cluster distance $c_T(u, v)$ equals the weight of the edge (of T) that labels the internal node $LCA(\mathsf{Leaf}(u), \mathsf{Leaf}(v))$ in $Top(T)$.*

Proof: Let distinct $u, v \in V$ be given. Let P be the unique path in T from u to v, and let b be the last edge of P to appear in the sorted list used by Algorithm 2. In particular, $c_T(u, v) = \omega(b)$. Let R be the root of the smallest subtree of $Top(T)$ containing all nodes labeled by the edges and vertices of path P.

Node R is an ancestor of both u and v, so R is an ancestor of $LCA(u, v)$. If R is a proper ancestor of $LCA(u, v)$, then path P would be disconnected. Since P is not disconnected, we have $R = LCA(u, v)$.

Finally, tree $Top(T)$ is built bottom-up, with later edges on the sorted list being closer to the root. Since R is minimal, b must label R, and hence $c_T(u, v)$ equals the weight of the edge labeling of $LCA(u, v)$. ■

3.3.3 Resource requirements for preprocessing

The running time of Algorithm 2 is dominated by two components: sorting the edges, and finding roots. Sorting the edges takes $O(|V| \log |V|)$ time. The only other super-linear time factor is contributed by finding roots of trees. Just as Algorithm 2 is a modified version of Kruskal's minimum spanning tree algorithm, we can adapt the standard implementation of a part of Kruskal's algorithm for the purpose of finding roots within Algorithm 2.

The standard version of Kruskal's minimum spanning tree algorithm needs a way of quickly finding out whether two vertices are in the same tree in the current forest. One solution uses a union-find abstract data type (ADT) over the space of all n vertices [41]. Initially all n vertices are in distinct sets. Operation $\mathsf{find}(u)$ answers some representative element of the set to which u belongs. Given representatives r and s from distinct sets, operation $\mathsf{union}(r, s)$ forms a new set: the union of the sets containing r and s. The answer to a $\mathsf{find}(u)$ query may not change between union operations involving the set containing u. We apply this abstract data type in Kruskal's algorithm as follows. At each stage in the execution of Kruskal's algorithm, the vertices are partitioned into subsets according to the connected components in existence at that time. Vertices u and v are in the same component if and only if their respective subsets have the same representative, *i.e.*, $\mathsf{find}(u) = \mathsf{find}(v)$. When a new edge (u, v) is added to the forest, we merge the components containing u and v by performing operation $\mathsf{union}(\mathsf{find}(u), \mathsf{find}(v))$.

A standard implementation for the union-find abstract data type uses a forest of trees to represent a partition of the vertices into subsets [41] [3, Section 5.5]. Each vertex in the forest points to its parent vertex; if the vertex is a root, it points to some distinguished nil value. Each root records the number of vertices in its own subtree. The representative element of a set is always the root of the tree corresponding to that set. When a $\mathsf{find}(u)$ query is issued, we follow parent pointers from node u to its root. To save time on future find queries, we perform path compression: all pointers on the path from u to its root are updated to point to the root. When a $\mathsf{union}(r, s)$ operation is issued, we compare the sizes of the trees rooted at r and s; we join the trees by making the root of the smaller tree point to the root of the larger tree; we also update the recorded size of the tree of the remaining root. This implementation of the union-find ADT runs in almost linear time: a sequence of $O(n)$ operations on a space of n elements takes $O(n \cdot \alpha(n))$ time, where α is an extremely slow-growing sub-logarithmic function [41].

Algorithm 2 requires that we be able to find roots quickly. (It does not need to know whether two vertices are in the same component since T is already known to be a tree.) The implementation of the union-find ADT given above is almost perfect for the job. The only complication is that Algorithm 2 specifies which nodes should be roots at any particular time, while for efficiency reasons the union-find implementation chooses the roots based on tree size. However, we can modify the union-find implementation so that its runtime characteristics are preserved while answering root-finding queries according to Algorithm 2's needs.

In the modified union-find implementation, we include all of $V_{Top(T)} = \{\mathsf{Leaf}(u) \mid u \in V\} \cup \{R_e \mid e \in E_T\}$ as singletons. In processing edge $e = (u, v)$ we form internal node R_e, performing $\mathsf{union}(\mathsf{find}(\mathsf{Leaf}(u)), \mathsf{find}(\mathsf{Leaf}(v)))$ and then $\mathsf{union}(\mathsf{find}(\mathsf{Leaf}(u)), R_e)$. (We could have used v in place of u in this last step.) We need to ensure that subsequent find operations in this subtree (before the next union involving this tree) end up finding node R_e. Since node R_e has not yet been merged with anything else, it is either the root or a child of the root. If it is a child of the root, then we swap it with the root in only one more step. This modified implementation keeps the total cost of the sequence of $O(|V|)$ find and union operations bounded by $O(|V| \cdot \alpha(|V|))$, the same asymptotic bound as for the standard union-find implementation. Since α grows sub-logarithmically, the running time for all the root-finding of Algorithm 2 is in $O(|V| \log |V|)$.

The total runtime for Algorithm 2 is therefore dominated by the time to sort the edges, and is therefore in $O(|V| \log |V|)$.

Topology tree $Top(T)$ has $2|V|-1$ vertices and $2|V|-2$ edges, so it may be stored in linear space. Algorithm 2 can be made to use only linear space.

3.3.4 Computing least common ancestors

Now let us turn to finding least common ancestors. Let R be any rooted binary tree having n leaves. With $O(n \log n)$ preprocessing time and $O(n)$ extra space, online least common ancestor queries in R may be answered in constant time [81]. This technique is applied to tree $Top(T)$, requiring the same asymptotic time and space bounds for preprocessing as used by Algorithm 2.

3.3.5 Cluster distance query summary

Taken altogether, we have the following theorem.

Theorem 3.8 (Cost of computing cluster distance) *Suppose we are given a weighted connected undirected graph $G = (V, E)$, and a minimum spanning tree for G. With $O(|V| \log |V|)$ preprocessing time and $O(|V|)$ extra space for data structures, we can compute arbitrary cluster distance queries in G in constant time.*

All these arguments can be extended to the case where G is possibly disconnected, with the following adjustments. First, the cluster distance between a pair of vertices from different components would be infinite. Second, we would use a minimum spanning forest instead of a minimum spanning tree. Finally, the least common ancestor for a pair of vertices from different components would need to be some distinguished non-edge value, *e.g.*, nil, having infinite weight. The resource bounds would stay the same.

3.4 When to apply efficient cluster compensation

Are the overheads imposed by efficient cluster compensation worth its benefits? When should we apply the technique?

In the worst case, cluster compensation does not help. The technique was inspired in part by the proof for the PLS-completeness of the problem solved by the Lin-Kernighan heuristic. That proof is quite robust, needing little modification to extend it in the case the cluster compensation is employed by Lin-Kernighan. Appendix A gives a sketch of the changes required.

In practice, the answer is more encouraging. The following chapters describe experiments with implementations of the Lin-Kernighan heuristic for both the Traveling Salesman Problem and the minimum weight perfect matching problem. In the experiments, the benefits of efficient cluster compensation almost always outweigh its overheads. These results suggest efficient cluster compensation should be used by default for geometric instances. For non-geometric instances, the extra overhead for computing minimum spanning trees may be significant enough to outweigh any advantage achieved by cluster compensation during the optimization phase. More precisely, these judgements are for Lin-Kernighan heuristics that use the concrete strategy described in Section 2.2. Other options are outlined in Section 10.1.

Cluster compensation provides us with only a loose estimation on future closing up costs (see Section 3.2.2). Because the estimate is loose, we do not expect cluster compensation to take away completely the running time penalty Lin-Kernighan experiences on clustered instances. That is, we do not expect the heuristic with cluster compensation to run as quickly on clustered instances as it does on uniform instances. Rather, our aim in applying cluster compensation is to make the heuristic run faster on a particular instance than it would if we had not applied cluster compensation. We compare using cluster compensation versus not using cluster compensation, rather than one instance versus another.

3.5 Summary

We have described the difficulty Lin-Kernighan heuristics have with clustered instances, suggesting the baiting effect of long inter-cluster edges as the primary culprit.

This chapter introduces the technique of cluster compensation, a way of modifying the cumulative gain and greedy selection criteria of Lin-Kernighan heuristics. Our hope is that cluster compensation reduces the baiting effect of inter-cluster edges. We have seen how to compute a cluster distance in constant time, with one-time preprocessing taking modest time and space.

But as with all heuristics, the real test is whether the technique works well in practice. The following chapters describe experiments showing the effect efficient cluster compensation has on an implementation of the Lin-Kernighan heuristic for both the Traveling Salesman Problem and the minimum weight perfect matching problem.

Chapter 4

Lin-Kernighan for the Traveling Salesman Problem

A great deal of work goes into making a high quality implementation of the Lin-Kernighan heuristic. This chapter describes the details of the Lin-Kernighan heuristic for the Traveling Salesman Problem within the framework given in Chapter 2. We cover the details as specified by Lin and Kernighan and the quarter century of work that followed to improve its scalability and effectiveness.

Johnson and McGeoch note that by omitting certain key features, one is liable to end up with a heuristic producing worse tours than even 3-Opt. Indeed, "Other authors seem to think that 'Lin-Kernighan' is a synonym for 3-Opt or even 2-Opt!" [46, p. 254]. We have taken great care to implement the key features so that these warnings do not apply to our work. We also therefore take great care to explain those details properly. We estimate that a high quality re-implementation would take upwards of eight months full time work.

Many of the key features are directly usable and sensible in the context of the minimum weight perfect matching. Chapter 7 shows how the features translate into that setting.

4.1 Historical context

The Lin-Kernighan heuristic was originally devised for the Traveling Salesman Problem. Most aspects of modern high quality implementations of the heuristic were described in the original paper by Lin and Kernighan [61]. Modifications for extending the heuristic's practical applicability to ever larger instances include the use of data structures with better asymptotic behaviour [33], taking advantage of geometric structure of the input [17], and miscellaneous shortcuts (such as Bentley's "don't look" bits) that have proven their worth through experimentation [16, 17].

The implementation used in our experiments is based on the description given by Johnson and McGeoch in their survey of local search techniques for the Traveling Salesman Problem [46]. Their survey reports results from experiments with a state of the art implementation of the Lin-Kernighan heuristic. The following sections describe the essential details of our implementation within the framework given in Chapter 2. Of course, the last word on the details of the implementation is the implementation itself (See Appendix B).

4.2 Feasibility

The first essential detail is how the heuristic maintains feasibility and near feasibility of intermediate structures. When the first edge (t_1, t_2) is removed from a tour, we are left with a Hamiltonian path (a path meeting all the vertices exactly once) beginning at t_1 and ending at t_2. When extending the t list of the Lin-Kernighan strategy (see Section 2.2), we choose pairs of vertices so that we are always left with a Hamiltonian path. That way we need only add one edge to re-form form a tour.

The mechanism for maintaining a Hamiltonian path works as follows. If we extend the t sequence with vertex t_3 (with $t_1 \neq t_3$), we are adding edge (t_2, t_3), and therefore creating a cycle. We select t_4 as the neighbour of t_3 (on the Hamiltonian path) that is nearer to t_2 than to t_1. This removes edge (t_3, t_4), which breaks the cycle to form a new Hamiltonian path. In general, we constrain t_{2i+2} as being a particular path neighbour of t_{2i+1} so that after building sequence $t_1, \ldots, t_{2i+2}$, we are always left with a Hamiltonian path. (We relax this rule over a small prefix of the t sequence for the sake of the backtracking rules. See Section 4.4.) The process is diagrammed in Figure 4.1.

Of course, in a complete graph any Hamiltonian path starting at vertex u and ending at vertex v can be transformed into a tour by adding edge (u, v). So if we have a Hamiltonian path after adding a pair of vertices to the t sequence, we can easily construct a tour by adding an edge between the two endpoints of the path, t_1 and t_{2i}.

In fact, each pair of candidates for t_{2i+1} and t_{2i+2} is checked to see if it would allow us to make a better tour. That is, if u and v are possibilities for t_{2i+1} and t_{2i+2}, we check

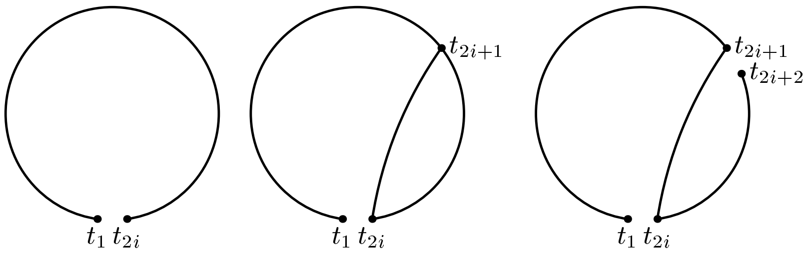


Figure 4.1: We maintain a Hamiltonian cycle by constraining t_{2i+2}.

to see if $net\_gain(t_1, \ldots, t_{2i}, u, v) > best\_net\_gain$. The value

$$net\_gain(t_1, \ldots, t_{2i}, u, v) = cum\_gain(2i) - \omega(t_{2i}, u) + \omega(u, v) - \omega(v, t_1)$$

is the net tour improvement we would attain if we extended the t sequence to $t_1, \ldots, t_{2i}, u, v$ and then closed up the Hamiltonian path to form a tour by adding edge (v, t_1). If in fact stopping with u and v represents a better alternative than the best tour improvement found so far, then we update $best\_net\_gain$ to be $net\_gain(t_1, \ldots, t_{2i}, u, v)$ and continue the search.

Although u and v might give us the best net tour improvement, some other candidate pair u' and v' might give us a better cumulative gain. (For this to happen, the closing up cost when using u and v must be less than the closing up cost when using u' and v'. That is, $\omega(v, t_1) < \omega(v', t_1)$.) As always, we extend the t sequence with the pair maximizing the cumulative gain. In this case it means using pair u' and v', *i.e.*, we choose $t_{2i+1} = u'$ and $t_{2i+2} = v'$.

4.3 Tabu rules

If left unrestricted, some choices for vertices t_{2i+1} and t_{2i+2} don't make sense because they don't make progress. For example, in the previous section we required that t_3 be distinct from t_1: if $t_1 = t_3$ then we'd be adding an edge we had just removed. Similar behaviour can occur with multiple edges at a time. That is, some sequences of choices can force the algorithm into endlessly swapping the same set of edges in and out of the current Hamiltonian path.

To avoid these problems, Lin and Kernighan devised two *tabu rules*: never delete an added edge; and never add a deleted edge. These eliminate the possibility of cycling, and also have the side effect of limiting the depth of the search. The first rule limits the search to at most $|V|$ swaps. The second rule limits the search to at most $O(|V|^2)$ swaps, since there are up to $O(|V|^2)$ non-tour edges available at the beginning of each search sequence. Modern implementations use only the first rule, *i.e.*, never delete an added edge [44, 46]. Our implementation does the same.

Papadimitriou's proof that a stylized version of the Lin-Kernighan heuristic for the TSP solves a PLS-complete problem requires that only the "never add a deleted edge" rule rule be used by the algorithm. The result is conjectured to hold if the "never delete an added edge" rule is used either alone or in conjunction with "never add a deleted edge." [67]

4.4 Limited backtracking

With a tour T in hand, we construct a t sequence to try to find a better tour T'. Most t sequences we construct do not lead us to a better tour. We use a limited backtracking scheme so that many alternative t sequences can be tried. The backtracking can be decomposed into a high level and a low level.

4.4.1 High level backtracking

At a high level, we iterate t_1 through all the vertices in V as long as no improvement is found. For each choice for t_1 we build a t sequence starting with t_1 and use the low level backtracking scheme until it finds a better tour. We can choose the best improvement found by the low level backtracking scheme, if any.

There are two shortcuts we can apply to this backtracking scheme. First, we don't need to examine all possible tour improvements the low level backtracking scheme can find. Second, we don't need to re-examine all the vertices every time we start with another tour.

The first shortcut is as follows. The first tour improvement answered by the low level backtracking scheme is declared the winner. We need not iterate t_1 through the rest of the vertex set V. Instead we apply the improvement to T to form T' and restart the local search, iterating t_1 anew over V.

The second shortcut is more involved. The 2-opt and 3-opt local search heuristics use the same restart mechanism described here. When experimenting with 2-opt and 3-opt, Bentley noticed that after a while the local search stops making progress in much of the graph [16]. That is, large portions remain the same between one tour and the next in the local search process. The vertices in these stable regions are examined again and again without finding improvements. Bentley therefore introduced "don't-look" bits for those heuristics [16, 17], and they apply equally well to the Lin-Kernighan heuristic [46].

Don't-look bits work as follows. Each vertex has a bit that determines whether it should be used as vertex t_1 in constructing a t sequence. If the bit for vertex v is off, then v should be used as a choice for t_1; if the bit is on, then v shouldn't be used as a choice for t_1. At the beginning of the local search, all vertices have their bits turned off.

At each stage of the local search, the high level backtracking considers all vertices as candidates for t_1. For a given choice $t_1 = v$, there are two outcomes: either we find a tour improvement or we don't, even while trying all t sequences according to the low level backtracking scheme. If we find a tour improvement with $t_1 = v$, then all the vertices involved in the edge exchange have their don't-look bits turned off. Those vertices are considered to be in the "active" region of the graph. If we don't find a tour improvement with $t_1 = v$ then we turn v's don't-look bit on; v is considered to be in the "stable" region of the graph.

Don't-look bits save work by focusing the algorithm's attention on the actively changing portions of the graph. The number of vertices having their don't-look bits turned off decreases greatly over time.

We can represent don't-look bits as membership in a set *Active*. A vertex is in the *Active* set if and only if its don't-look bit is off. Iterating over the set, and inserting and removing elements from the set can be done efficiently.

Our implementation of the Lin-Kernighan heuristic for both the TSP and the minimum weight perfect matching uses don't-look bits. The local search process, including the high level backtracking scheme, is depicted in Algorithm 3.

In our implementation, the *Active* set is a first-in-first-out queue supporting a constant time membership test, similar in spirit to the sparse set representation of Briggs and Torczon [20]. At initialization time, line 1 of Algorithm 3, all the vertices are inserted in random order. Thereafter when a vertex is to be made active again, in line 6 of Algorithm 3, it is inserted at the end of the *Active* queue only if it isn't already in the queue. The implementations of Johnson *et al.* and Applegate *et al.* handle the active queue similarly [45, 4].

Algorithm 3 Local search and high level backtracking, with don't-look bits.

Require: T is a tour on graph $G = (V, E)$

Ensure: T is a LK-optimal tour on G

$Active := V$
while $Active \neq \emptyset$ **do**
    Choose $v \in Active$
    Run low level backtracking on T with $t_1 = v$
    if We find a better tour T' using $t_1, \ldots, t_{2i}$ **then**
        $Active := Active \cup \{t_1, \ldots, t_{2i}\}$
        $T := T'$
    else
        $Active := Active \setminus \{v\}$
    end if
end while

4.4.2 Low level backtracking

When a search sequence from a given vertex t_1 fails to find a tour improvement, the algorithm backtracks. Lin and Kernighan specify that, as long as searches fail, all alternatives for t_2 through t_6 should be tried. As always, the backtracking choices are constrained by feasibility criteria, tabu rules, and candidate sets.

However, the choices for t_2 through t_6 are also constrained so that a Hamiltonian path would remain if the changes specified by the *whole* prefix are applied. Allowable choices may be determined by the relative order of vertices t_1 through t_6 (and sometimes through t_8) on the start tour *before* any changes are applied. Table 4.1 shows the case analysis tree. Subcases are indicated by nesting within the table. Recall that t_1 and t_2 are always tour neighbours, so they are always adjacent in the relative orderings. The same is true for the pairs (t_3, t_4), (t_5, t_6), and (t_7, t_8). However, any number of vertices may occupy the positions marked by the precedence symbol ($\prec$). In fact, the t vertices on either side of a precedence marker may denote the same tour vertex. Each possible ordering among the first few t vertices is marked in the table as either allowable or not allowable. Allowable configurations are depicted in Figure 4.2; the start tour would be shown as a circle.

We allow choices for t_1 through t_6 that result in a Hamiltonian path even though

Relative order of $t_1, \ldots, t_4$ on start tour / Relative order of $t_1, \ldots, t_6$ / Relative order of $t_1, \ldots, t_8$	Allowable, *i.e.*, forms a Hamiltonian path?	Diagram, if allowable
$t_1t_2 \prec t_3t_4 \prec t_1$	no	
$t_1t_2 \prec t_6t_5 \prec t_3t_4 \prec t_1$	yes	Fig. 4.2(a)
$t_1t_2 \prec t_5t_6 \prec t_3t_4 \prec t_1$	yes	Fig. 4.2(b)
$t_1t_2 \prec t_3t_4 \prec t_6t_5 \prec t_1$	no	
$t_1t_2 \prec t_7t_8 \prec t_3t_4 \prec t_6t_5 \prec t_1$	yes	Fig. 4.2(c)
$t_1t_2 \prec t_8t_7 \prec t_3t_4 \prec t_6t_5 \prec t_1$	yes	Fig. 4.2(d)
$t_1t_2 \prec t_3t_4 \prec t_5t_6 \prec t_1$	no; no for all 4-change subcases	
$t_1t_2 \prec t_4t_3 \prec t_1$	yes	Fig. 4.2(e)
$t_1t_2 \prec t_6t_5 \prec t_4t_3 \prec t_1$	no	
$t_1t_2 \prec t_6t_5 \prec t_8t_7 \prec t_4t_3 \prec t_1$	yes	Fig. 4.2(f)
$t_1t_2 \prec t_6t_5 \prec t_7t_8 \prec t_4t_3 \prec t_1$	yes	Fig. 4.2(g)
$t_1t_2 \prec t_5t_6 \prec t_4t_3 \prec t_1$	yes	Fig. 4.2(h)
$t_1t_2 \prec t_4t_3 \prec t_6t_5 \prec t_1$	yes	Fig. 4.2(i)
$t_1t_2 \prec t_4t_3 \prec t_5t_6 \prec t_1$	no	
$t_1t_2 \prec t_8t_7 \prec t_4t_3 \prec t_5t_6 \prec t_1$	yes	Fig. 4.2(j)
$t_1t_2 \prec t_7t_8 \prec t_4t_3 \prec t_5t_6 \prec t_1$	yes	Fig. 4.2(k)
$t_1t_2 \prec t_4t_3 \prec t_7t_8 \prec t_5t_6 \prec t_1$	yes	Fig. 4.2(l)
$t_1t_2 \prec t_4t_3 \prec t_8t_7 \prec t_5t_6 \prec t_1$	yes	Fig. 4.2(m)

Table 4.1: Case analysis for low level backtracking portion of Lin-Kernighan for the TSP. Allowable choices are determined by the relative ordering of t_1 through t_8 on the starting tour. Subcases are nested.

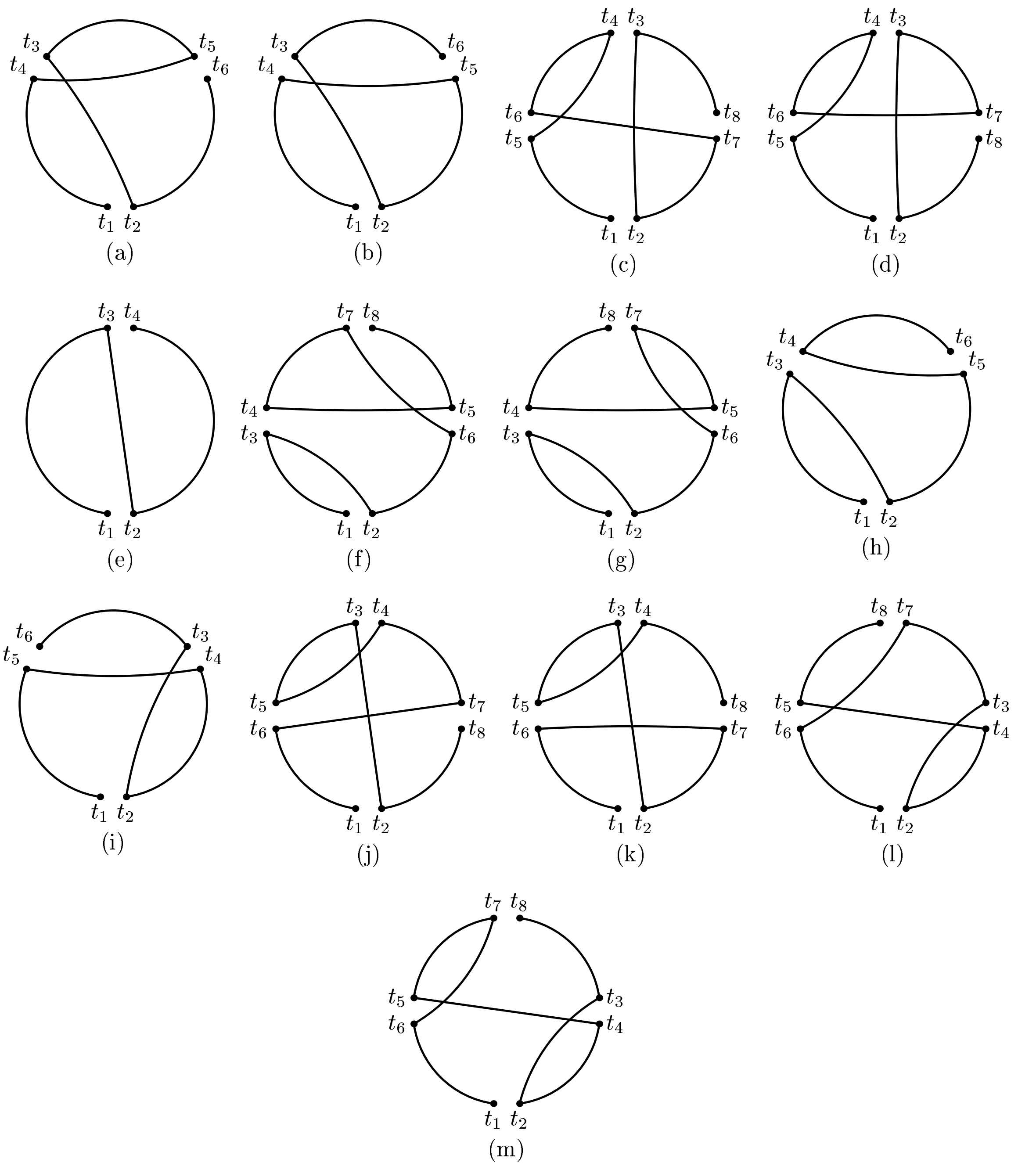


Figure 4.2: Allowable backtracking cases for Lin-Kernighan for the TSP. The relative ordering of vertices t_1 through t_6 (and sometimes t_8) on the start tour determine whether a Hamiltonian path results when those changes are applied. Shown are possibilities when edges (t_1, t_2), (t_3, t_4), (t_5, t_6), and (t_7, t_8) are removed from the start tour while edges (t_2, t_3), (t_4, t_5), (t_6, t_7) are added. See also Table 4.1.

those choices for t_1 through t_4 do not. Parts (a) and (b) of Figure 4.2 show such cases.

Some choices for t_1 through t_6 do not result in a Hamiltonian path, but may be extended with choices for t_7 and t_8 so that t_1 through t_8 specifies a change that results in a Hamiltonian path. All such alternatives for t_1 through t_6 are tried, but for each of them, only the first pair (t_7, t_8) is tried. Diagrams (c), (d), (f), (g), (j), (k), (l), and (m) in Figure 4.2 depict these cases.

The original Lin and Kernighan implementation uses this backtracking schema, as does our own, and the Johnson *et al.* implementation described in [46]. In the interest of saving computational work, some researchers suggest using a more limited backtracking schema [74]. However, Johnson and McGeoch argue that reducing the backtracking schema is a poor tradeoff [46]. In this they agree with Lin and Kernighan, that backtracking over a larger prefix incurs "a considerable time penalty," while backtracking to only shallower levels "weakens the procedure" [61, p. 506]. In practice, the more extensive backtracking schema used here leads to better tours and does not require much more computational work.

4.5 Candidate sets

The low level backtracking schema specifies the allowable *relative* orderings of t_1 through t_6 (or t_8). If t_1 and t_2 are fixed, there may be many choices for t_3. Similarly, if t_1 through t_4 are fixed, there may be many choices for t_5. In general, with t_1 through t_{2i} fixed, there can be many choices for t_{2i+1}. As a time-saving measure, these choices are constrained with candidate lists, though without appreciable loss in tour quality [61, 44, 74]. Candidate lists are also known as *neighbour lists* [46].

When discussing the general Lin-Kernighan scheme, we described the candidate set for an element $e \in S$ as those elements $e' \in S$ that are likely to improve the solution locally when they are swapped with e. In the TSP, the elements e are edges of the graph. The union of all the candidate sets is the candidate subgraph of the instance. We represent the candidate subgraph as a list of adjacencies; each vertex knows its neighbours in the candidate subgraph. Vertex t_{2i+1} is chosen from one of the neighbours of t_{2i} in the candidate subgraph.

What candidate sets are appropriate for the TSP? Lin and Kernighan suggest using the 15 nearest neighbour subgraph as the candidate subgraph. That is, in the candidate subgraph each vertex is adjacent to its 15 nearest neighbours in the whole graph.

Naturally, one can use any number of nearest neighbours.

For geometric instances, Johnson suggests using quadrant-based neighbours [44]. (Recall that a geometric TSP instance is one in which the vertices are given coordinates, and the distance function is some metric over that coordinate space.) For example, in a 2-d Euclidean TSP instance, the quadrant-based 40 nearest neighbours of a vertex v is the union of the 10 nearest neighbours to v that lie to the north-east of v, together with the 10 nearest to the north-west, the 10 nearest to the south-east, and the 10 nearest to the south-west. Each vertex would thus have up to 40 vertices on its candidate list. (Vertices on the boundaries of the instance might not have enough neighbours in a particular direction to fill up the quota for that quadrant.) This candidate subgraph provides a richer interconnection network to explore, especially when clustering is present in the input. Quadrant-based lists can be built efficiently using Bentley's k-d trees [15]. The quadrant-based idea easily extends to higher dimensions.

For 2-d geometric instances, Reinelt suggests using a Delauney triangulation as the basis for the candidate subgraph [74]. The Voronoi region of a vertex v in G is the set of points on the plane that are closer to v than to any other vertex of G. Taken together, the Voronoi regions of all the vertices in a graph form a tiling of the plane. The Delauney triangulation of G is a graph with the same vertices as G. The pair (u, v) is an edge of the triangulation if and only if the Voronoi region of u is adjacent to the Voronoi region of v. The Delauney triangulation can be computed in $O(n \log n)$ time. Furthermore, it contains at most $3n$ edges and includes a minimum spanning tree for G as a subgraph [72, 55, 74].

Another kind of candidate subgraph is based on *2-matchings*. Each vertex in a tour is touched by two edges, *i.e.*, a tour is a 2-matching of the graph. In light of this, some researchers suggest using the edges present in a minimum weight 2-matching as a candidate subgraph. One can formulate the minimum weight 2-matching problem as a linear program, where each edge e in the graph is associated with an indicator variable x_e forced to take values of either 0 or 1. An edge e is included in the 2-matching when the solution to the linear program has $x_e = 1$, and is not included otherwise [68]. One can also relax the integrality constraints on the indicator variables, allowing them to take on all values between 0 and 1 inclusive. The solution to the relaxed linear program is a minimum weight *fractional 2-matching*. A candidate subgraph for Lin-Kernighan may consist of the set of edges whose indicator values are positive (though not necessarily as large as 1) in a minimum weight fractional 2-matching [4, 26].

Finally, one can always take the union of multiple candidate subgraphs to produce a (possibly) larger candidate subgraph. The Concorde software suite has the ability to do so [4].

Candidate subgraphs are designed to focus the attention of the search heuristic on fruitful territory. But there is a tradeoff. Making the candidate subgraph larger enriches the search space, providing possibly better tours at the expense of longer running times. Making the candidate subgraph smaller impoverishes the search space, reducing running times at the expense of providing possibly worse tours. However, simple choices for candidate subgraphs often serve well for a variety of situations. On any instance, the 15- or 20-nearest neighbour subgraph offers a good tradeoff; on Euclidean graphs, the quadrant-based 20 nearest neighbour subgraph is a good choice [44, 46].

Our implementation of the Lin-Kernighan heuristic can find and use unrestricted nearest neighbours lists, quadrant-based neighbour lists, and the unions of these two kinds of lists.

4.6 Bounding search sequence lengths

With the tabu rule "never delete an added edge," search sequences are bounded. For an instance with n vertices, only n edges can be added to the tour, and hence the t sequence must end by t_{2n}, representing up to n edge exchanges.

However, most search sequences are much shorter than n exchanges, and most improving moves are shorter still [46, p. 258]. For a wide range of TSP instances, the average depth of a search sequence is 7 edge exchanges (to t_{14}), only three or four edge exchanges beyond the backtracking depth [46, p. 258]. A long search sequence is an indication of either the rare case of a very extensive improvement being built, or is an unfruitful waste of effort. In order to catch the long search sequences, some researchers suggest forcibly stopping a search sequence at a fixed depth. For example, Applegate *et al.* suggest stopping at 50 edge exchanges beyond the backtracking depth [6], *i.e.*, stop at t_{106} or t_{108}. Johnson and McGeoch follow this suggestion for their results on "production mode Iterated Lin-Kernighan" [46, p. 293]. (See also section 4.9.) Reinelt suggests the sharper limit of stopping after just 15 edge exchanges in total [74], *i.e.*, stop at t_{30}.

Cluster compensation itself is designed to shorten search sequences. We shall see that it reduces the frequency and impact of unfruitful searches, but that it does not eliminate them. For our own experiments, we follow Applegate *et al.* and bound search sequences to at most 50 edge exchanges beyond the backtracking depth.

4.7 Start tour

The Lin-Kernighan heuristic is a local search algorithm, and therefore requires a start tour that it tries to improve. For our experiments, we follow Johnson *et al.* in using a randomized greedy algorithm for the start tour [45].

A pure greedy tour for instance $G = (V, E)$ is constructed as follows, with n being the number of vertices, $n = |V|$. The algorithm is similar in spirit to Kruskal's minimum spanning tree algorithm. In n steps we grow a set of edges from the empty set to a whole tour on the graph. Set T_i is the set of i edges after i steps of the algorithm; T_0 is empty, T_{n-1} is a Hamiltonian path, and T_n is a tour. For steps i from 0 through $n - 1$, set T_i consists of disjoint simple paths containing a total of i edges. During the execution of the algorithm, we examine the edges in E in order from shortest to longest. During step i, with $i < n - 1$, we form T_{i+1} from T_i by adding the next shortest edge that maintains the invariant, *i.e.*, the edge does not create a cycle and it does not force some vertex to be of degree 3. In the last step we make the tour T_n from the Hamiltonian path T_{n-1} by adding the edge between the path's two endpoints.

There is only one difference between this greedy tour algorithm and Kruskal's minimum spanning tree algorithm. In Kruskal's minimum spanning tree algorithm, an edge is discarded if it would create a cycle. In this greedy tour algorithm, an edge is discarded if it would create a cycle prematurely (before the n'th edge is to be added), or if it would force some vertex to be of degree 3.

Tours discovered by this greedy algorithm are suitable for use by local search heuristics because they consist of mostly short edges and have a few large "defects" that can be exploited by the local search technique [17] [46, p. 236]. The short edges are added at the beginning stages of the greedy algorithm, and the defects are the long edges added in the latter stages of the greedy algorithm.

The synergy between the output of the greedy algorithm and local search heuristics is intensified whenever Bentley's "don't-look" bits are used, *i.e.*, if we use an *Active* queue as described in Section 4.4.1. In this case the heuristic will quickly find that it cannot improve the tour within the vicinity of the short edges. The vertices in those areas will tend to drop out of the *Active* queue quickly. The local search heuristic will therefore focus its attention on the large defects in the tour, where wholesale improvements can be made. This is especially true for the Lin-Kernighan heuristic with its ability to find changes involving many edges, *i.e.*, changes encoded by a long t vertex sequence.

In order to introduce some variation between executions of the Lin-Kernighan heuristic, we use a randomized version of the pure greedy algorithm in our experiments. In the pure greedy algorithm we always inserted the next allowable edge. In the randomized version, during the first $n-1$ steps we determine the two next allowable edges and choose randomly between them. (One must ensure that the two edges are distinct. A naïve implementation can get caught always choosing between (u, v) and (v, u). Within the implementation these two edges might have a distinct representation, but they will have the same weight. Since the graph is undirected, choosing between these two does not introduce any variation at all. This problem is especially pronounced if all the edge weights are distinct; the forward representation of an undirected edge will always be paired with its backward mate.) We follow Johnson *et al.* in choosing the shorter of the two edges with probability of 2/3 [45], and the longer of the two with the remaining probability of 1/3. The edge not chosen goes back into the pool of available edges.

The extra time taken by the randomized greedy algorithm to find distinct allowable edges is not significant. The penalty is roughly ten percent or less extra running over that of the pure greedy algorithm. To put this into perspective, within our implementation the time taken by the randomized greedy algorithm is roughly one tenth the time it takes to build a 20 nearest-neighbour candidate subgraph, using Bentley's k-d trees for both algorithms.

4.8 Data structures

We need good data structures to achieve low running times, especially as instance sizes grow. Equally important, the data structures should be well-matched to the heuristic. We want to make the common cases fast, and if possible, make the fast cases common [53]. For example, if the heuristic uses some operation on an abstract data type (ADT) many more times than the other operations of that ADT, then we tune the implementation of that ADT to make the common operation fast. Similarly, we try to tune the implementation to be faster on commonly used argument patterns.

In terms of execution time, the main work of the Lin-Kernighan heuristic is examining the vertices that might become t_{2i+1} and t_{2i+2}, given vertex t_{2i}. We should therefore pay close attention to the data structures supporting these operations.

4.8.1 Candidate lists

Candidates for t_{2i+1} appear on the candidate list for t_{2i}. Since we will revisit each candidate list many times, we compute all candidate lists once as a preprocessing step. The list for vertex v is ordered from nearest to farthest neighbour of v in the hope that more promising moves are found first [46]. For geometric instances, Bentley's k-d search trees are useful for creating both ordinary and quadrant-based nearest neighbour lists since k-d trees support fast near-neighbour searching for geometric point sets [15, 17].

Sorted candidate lists are a general feature of Lin-Kernighan heuristics, so preparing them in advance is useful for all Lin-Kernighan heuristics. In particular, it is useful for Lin-Kernighan for minimum weight perfect matchings.

4.8.2 Oriented-tour abstract data type

Vertex t_{2i+2} is always a neighbour of vertex t_{2i+1}. Within the backtracking cases, *i.e.*, up to t_6 or t_8, vertex t_{2i+2} can often be either tour neighbour of t_{2i+1}. Beyond the backtracking cases, vertex t_{2i+2} can be only one neighbour of t_{2i+1} as discussed in Section 4.2. In the latter cases determining *which* tour neighbour depends on the relative ordering of vertices t_1, t_2, t_{2i}, and t_{2i+1} within the current working Hamiltonian path.

Another common operation is the exchange of edges in the current Hamiltonian path. As shown in Figure 4.1, adding t_{2i+1} and t_{2i+2} to the t list corresponds to reversing the portion of the path from t_{2i} to t_{2i+2}. That portion of the path can be very large, especially as instance sizes grow.

Fredman *et al.* [33] describe an oriented tour abstract data type supporting these queries (predecessor, successor, and relative ordering) and the updates (reverse a segment). A Hamiltonian path from t_1 through t_{2i} is represented as the tour consisting of that Hamiltonian path together with the edge (t_1, t_{2i}). We can just ignore edge (t_1, t_{2i}) when we need to think of the tour as a Hamiltonian path.

Fredman *et al.* report experiments on four kinds of implementations for the oriented tour: arrays, two-level trees [23], segment trees [7], and splay trees [33]. An array-based implementation is suitable for instances with up to about a few thousand cities, primarily because its simplicity makes it fast. For instances with between a few thousand and a million cities, implementations based on two-level trees, segment trees, and splay trees are better than arrays. In practice, two-level trees and splay trees are the most competitive in this range. Here we discuss the key features of the array, two-level tree, and splay tree implementations.

The array-based implementation represents the tour as two arrays. One array corresponds to one of the n permutations π that the tour defines over the set of vertices, and the other corresponds to the inverse permutation π^{-1}. For example, if vertex v is in position i on the tour, then $pi[i] = v$ and $pi\_inv[v] = i$. The predecessor, successor, and relative ordering queries can each be performed in constant time. Reversing a segment of the tour is just reversing the corresponding segment in the *pi* array, and updating the inverse array accordingly. This takes time proportional to the length of the segment being reversed. Reversing a segment of a tour is the same as reversing the complement of that segment, so we can always choose to reverse the shorter of the two. Unfortunately, this trick does not remove the linear worst-case time for the update operation. As instance sizes grow into the many thousands, the time for update operations begins to dominate the run time for the heuristic.

Two-level trees represent the tour as a tree with large arity, much like a flattened B-tree; both the root and its children can be of width $O(\sqrt{n})$ [23]. Two-level trees support the queries in constant-time and the updates in worst-case $O(\sqrt{n})$ time. However, experience shows that most updates are over short segments, so it is recommended that the children should each be about 200 wide, even for instances with up to a million cities [33].

Splay trees support each of the queries and updates in amortized $O(\log n)$ time. They represent the tour as a splay tree, where each node is labeled by a vertex of the tour, and each subtree represents a contiguous segment of the tour. To support fast reversal of tour segments, each node in the splay tree is also equipped with a reversal bit. When the reversal bit is off, the tour segment is read from the subtree with an in-order traversal; when the reversal bit is on, the tour segment is read from the subtree with a reversed in-order traversal.

Fredman *et al.* also prove a lower bound of worst-case amortized $\Omega(\log n / \log\log n)$ time per operation for the oriented tour abstract data type in the cell probe model of computation. Therefore implementations taking $O(\log n)$ time per operation —splay trees (amortized only), implementations based on AVL trees [23], and a balanced nested ring implementation [62]— are nearly optimal in a theoretical worst-case sense.

Our implementation supports both array-based and two-level tree-based oriented tours. For our experiments, we use arrays for instances with up to 1000 cities, and two-level trees for instances with more than 1000 cities.

4.8.3 Tabu lists

Another common operation performed by the heuristic is checking the tabu conditions, *i.e.*, determining whether a certain edge has been removed or added during the current search sequence. (Recall that our implementation uses only the rule: "never delete an added edge.") The tabu-checking problem can be solved by storing the disallowed edges in a set and checking for membership.

There are many possible implementations, and we should use the one best suited to the usage pattern of the Lin-Kernighan heuristic, and whose data structures fit within memory. Recall from Section 4.6 that search sequences are usually very short—the average length is roughly 7 edge exchanges, or until t_{14}. Consequently, the tabu lists are also very short on average. This usage pattern favours simple tabu list implementations.

If large instances are to be tackled, then using a bit vector to record tabu edges is undesirable, since there are $O(n^2)$ edges in an instance with n vertices. For scalability, we require a data structure that incurs a space overhead of at most $O(n)$ cells.

Our implementation can use one of three data structures meeting this requirement: a linear array, a splay tree, and a hash table.

Our implementation stores the t sequence in a one-dimensional array. One choice for checking whether an edge is tabu is to scan the t array. Adding elements to the tabu list is automatic; a new element is added when the t array is extended. This scheme is simple and incurs no space overhead. Since it revisits space already being used by the heuristic, it takes full advantage of the processor's cache: it is likely that the t array already resides in the fastest portion of the processor's memory. However, this scheme can get quite expensive when a search sequence gets long. If the search sequence reaches a depth of l steps, then $\Omega(l^2)$ time has been spent checking tabu conditions for that sequence. This linear array scheme is best when the t sequences are short on average.

A splay tree can also be used to store the tabu edges. Both adding an edge to the tabu list, and checking for membership take amortized logarithmic time, *i.e.*, $O(\log l)$, where l is the size of the current tabu list. Splay trees have a nice locality property that might make them more suitable than other tree data structures: recently accessed elements always reside near the root of the tree, and so are found more quickly [82]. In the case of tabu checking, this means that an edge that actually is tabu is often found to be tabu quite quickly. We expect there to be some locality in the tabu queries for the following reason. For most of the execution of the heuristic, the edges being added to the tour (and hence checked for tabu-ness) are likely to be short relative to the entire tour. So a search sequence will often reexamine the same vertices and edges during a single search sequence. However, splay trees have a high constant of proportionality, and so are better than linear arrays only when search sequences often become quite long.

The third option, a hash table, can often provide constant time addition and membership checking. However, hashing is more involved than the linear scan. A simple hash function is likely to be best, such as the exclusive-or of the two vertex numbers [4]. When the search sequence is ended, the tabu list is emptied in preparation for the next sequence. Quickly clearing out the hash table becomes an issue. Since search sequences are usually short, the buckets are likely to be sparse relative to the entire table. Checking every bucket is too much work; only the "full" buckets should be touched. This problem can be handled by maintaining a linked list of the dirty buckets.

Both hash tables and splay trees incur a space overhead. So in comparison to the linear array tabu check, these more elaborate tabu data structures add pressure to the processor's first-level cache, possibly slowing down the entire heuristic. This would reduce the attractiveness of these fancier data structures in comparison to the linear array check. We leave cache pressure measurements to other researchers. In choosing which tabu list implementation to use, we instead rely on overall runtime measurements on a variety of instances.

One might combine schemes to form a hybrid tabu check. We can use the linear array approach when the t sequence is short, and switch over to either the hash table or some other scheme with good asymptotic behaviour when the t sequence grows long. This hybrid may slow down the tabu checking for *both* the short and long cases since one must check *which* search to perform on each search. This might overwhelm any savings expected from the hybrid approach. We leave such investigations to other researchers, with the recommendation that probing depths for a wide variety of instance classes be observed before settling on a crossover point. Note that cluster compensation is designed to shorten search sequences, and therefore skews the distribution of search sequence depths. Furthermore, we conjecture that cluster compensation provides a greater time saving than any hybrid tabu set implementation would give on its own.

For our implementation, the simple linear array check provides the fastest running times, regardless of whether cluster compensation is used. We therefore always use the linear array tabu check in our experiments.

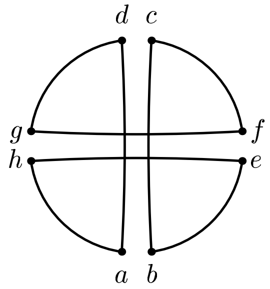


Figure 4.3: A double-bridge 4-change. Edges (a, b), (c, d), (e, f), and (g, h) are removed, while (a, d), (b, c), (e, h), and (f, g) are added.

4.9 Iterated Lin-Kernighan

One can run any local search heuristic multiple times in sequence, applying mutations to the current solution between iterations. This technique is called *chained local optimization* and can be viewed as a genetic algorithm with a population size of 1 [63] [46, p.291]. Martin, Otto & Felten showed how performing mutations in between runs of the 3-opt heuristic could be a powerful technique for finding short tours [64, 65].

The tour mutations used by the Martin *et al.* heuristic are randomly chosen double-bridge 4-changes, biased to result in adding some short edges. A double-bridge move is depicted in Figure 4.3. It consists of two 2-changes performed in parallel. If the 2-changes are run in sequence, then the intermediate result is two subtours. So double-bridge moves cannot be discovered by the 2-opt heuristic. Double-bridge moves remove four edges and add four edges, so they cannot be discovered directly by 3-opt heuristics. Improvements found by the Lin-Kernighan heuristic always form an even-length cycle alternating between added edges and removed edges, so double-bridge moves elude the Lin-Kernighan heuristic as well [61]. Double-bridge moves therefore make for good mutations on tours since they are not easily undone by the common TSP local search heuristics.

Martin *et al.* use an *Active* queue of cities to focus the search as described in Section 4.4.1. When starting a new iteration of 3-opt, they load the *Active* queue with the eight cities that were involved in the double bridge mutation, and no others. The new run of the 3-opt heuristic immediately sets to work trying to repair the damage caused by the mutation.

An improved solution found by a 3-opt search is always accepted as the new incumbent tour, *i.e.*, it becomes the start tour for the next step. If the tour in hand at the end of a 3-opt search is worse than the incumbent tour, then the Martin *et al.* heuristic allows this worse tour to become the new incumbent tour with some small probability. Over time the sequence of probabilities decay, similar to the way the probability for accepting uphill moves decays in simulated annealing heuristics [46, 2].

Following the ideas of Martin *et al.*, Johnson [44] combined the Lin-Kernighan heuristic with double-bridge mutations, and disallowed altogether the possibility of accepting a worse tour as the incumbent. The result is the *Iterated Lin-Kernighan* (ILK) heuristic. It is similar to Baum's *iterated descent* for the TSP [12, 13], which uses 2-opt in place of Lin-Kernighan, and a random 2-change in place of the double-bridge 4-change.

Johnson's initial implementation of Iterated Lin-Kernighan loaded all the cities into the *Active* queue at each iteration, and did not bias the double-bridge mutations. Iterated Lin-Kernighan proved to be very powerful. Given enough time, it found better tours than other heuristics run for the same time, often finding optimal tours for those instances with known optima. Since then, the Johnson *et al.* implementation has been updated to use the Martin *et al.* rule of loading the *Active* queue with only the eight cities involved in the mutation. For the same number of iterations, this change reduces running times substantially, but also slightly worsens the quality of the tours found. On a time-equalized basis (rather than on an iteration-equalized basis), the change is cost-effective [46, p. 293].

Our implementation follows the recent behaviour of the Johnson *et al.* implementation. It can perform any number of pre-specified iterations; it does not bias its 4-change mutations; between iterations it loads the *Active* queue only with the eight cities whose tour neighbours change with the mutation; and it never allows a worse tour to become the incumbent.

4.10 Summary

At over 25 years old, Lin-Kernighan is one of the older heuristics still in use for the TSP. This chapter describes many of the features that have been tried and accepted as mainstream additions to the heuristic over that period of time. With them, Lin-Kernighan has remained formidable competition for newer heuristics.

Lin-Kernighan has many features and tunable parameters. This chapter describes them in enough detail so that the reader can compare the implementation used for the results in this thesis with other implementations described in the literature. Just as important, the reader has the tools to compare the experiments reported here with those in the literature, for example by considering differences in the candidate subgraph or the backtracking cases used in both. The reader also has enough detail to begin a new implementation and replicate our work.

Of course, the final authority on our implementation is the source code itself. See Appendix B for details on obtaining the source code.

Chapter 5

Experimental methodology

This chapter describes our experimental methodology. We explain the purpose of the experiments and all factors affecting their outcomes. We describe the experimental environment well enough so that our results can be compared with those of other researchers. We discuss the values measured, including their interpretation and why they are measured. We also describe the test data, which consists largely of instances identical to or very similar to those used by other researchers.

5.1 Objectives

Is efficient cluster compensation a worthwhile technique? To answer this, we run experiments to compare two versions of the Lin-Kernighan heuristic: one with cluster compensation, and the other without it. The two main performance criteria are running time and tour quality.

For each instance we perform many kinds of experiments on both versions of the heuristic. We perform a single run of the heuristic, and equal numbers of Iterated Lin-Kernighan iterations. We consider the two versions on a time-equalized basis, to determine whether adding cluster compensation to Lin-Kernighan provides better tours in the same time.

5.2 The computer code and execution environment

The computer program code is a Literate C program written by the author using Knuth and Levy's CWEB system [54, 56]. A single code base is used for both versions. The different versions are generated by specifying different options to the compiler. By using the same code base we remove variability due to differences in programmer skill.

The baseline version of the code follows as closely as possible the description of the Johnson *et al.* implementation given by Johnson and McGeoch [46]. For brevity, we will often call it *LK*. The important details of our implementation have been given in the previous chapters. The version employing efficient cluster compensation is the same as the baseline version, except that the ordinary cumulative gain and greedy selection criteria (Criteria 2.8 and 2.9) are replaced by their cluster compensating versions (Criteria 3.3 and 3.4, respectively). We will often call it *LKcc*, for "Lin-Kernighan with cluster compensation."

The experiments were run on the author's home computer, a system with a 300MHz Pentium II CPU,[1] and 128 megabytes of random access memory. The system runs version 5.1 of the Red Hat Linux operating system.[2] We used the GNU C compiler, version 2.7.2.3, with optimization level `-O2`. More details on the hardware and software environment can be found in Appendix C.

The experiments were performed when the system was unloaded; only the minimal set of system processes were running in addition to the code being measured. Run times are always in user CPU seconds, the time spent executing the user's code, and do not include the system time, the time spent executing the operating system code (*e.g.*, for servicing page faults or performing context switches). We use the *getrusage* library call to determine the user CPU time spent by the program. As with most Unix[3] and Unix-like systems, time measurements are meaningful to a resolution of no better than one hundredth of a second.

Benchmarks, though imperfect, can offer some idea of the speed of a computer system. Our system runs the DIMACS benchmark `wmatch` test 1 in 0.13 CPU seconds and `wmatch` test 2 in 1.07 CPU seconds. The `wmatch` benchmark is Rothberg's implementation of Gabow's $O(n^3)$ time algorithm for weighted matching. The two versions of the benchmark are executions of the `wmatch` code on two particular inputs [78, 66]. In comparison, the experiments reported by Johnson and McGeoch for the Johnson *et al.* implementation use a single 150MHz MIPS R4400 processor of an SGI Challenge machine [46]. That

[1] "Pentium" and "Pentium II" are trademarks of Intel Corporation.

[2] "Red Hat" is a registered trademark of Red Hat Software Incorporated. "Linux" is a registered trademark of Linus Torvalds.

[3] Unix is a trademark of the X/Open Company, Ltd.

system is about nearly four times slower than ours, running the `wmatch` test 1 in 0.5 seconds and the `wmatch` test 2 in 4.0 seconds.

5.3 Items included in run time

Time measurements for the heuristic do not include the time taken to read the instance from a file, nor to output the results. However, they do include the time for all preprocessing required by the heuristic, including the building of candidate sets, and construction of a start tour. In particular, run times for the cluster compensating version include the extra preprocessing that efficient cluster compensation requires. The cluster compensating version adds a few items to the baseline version, as described in Chapter 3: building a minimum spanning tree (MST); re-processing the MST into a topology tree; preparing data structures for fast least common ancestors on the topology tree; and finally, when deciding on eligibility and prioritization of prospective edge exchanges, the cumulative gain function is discounted by the cluster distance between the two endpoints of the Hamiltonian path. To maximize the fairness of the comparison, the baseline version does not pay runtime costs for cluster compensation.

Our experiments for weighted perfect matching follow the same rules about running times.

5.4 Measuring tour quality

We follow Johnson and McGeoch in measuring tour quality in terms of percentage over the Held-Karp lower bound [39, 40, 46]. For any instance, the Held-Karp bound provides a lower bound on the length of an optimal tour. The bound is usually very tight. For a wide variety of instances, it averages less than 0.8% below the optimal tour length [47]. Furthermore, the Held-Karp bound can be computed or approximated in a reasonable amount of time—much less time than required to compute the length of an optimal tour.

The Held-Karp lower bound is based on the 1-tree relaxation with Lagrangean multipliers. A 1-tree on a graph G with n nodes is a connected subgraph of G containing all n vertices and exactly n edges. Since a tour for G is a 1-tree for G, a minimum weight 1-tree is no heavier than a minimum weight tour. We can restrict our attention to 1-trees in which vertex n is of degree two. In this case, a minimum weight 1-tree is just a minimum spanning tree on the first $n-1$ nodes, together with the two shortest edges incident upon node n.

The 1-tree relaxation alone provides a rough estimate on optimal tour length. However, the use of real-valued Lagrangean multipliers improves the estimate considerably. Lagrange multipliers are used in the following way. Instead of computing minimum 1-trees with the basic distance function ω, compute minimum 1-trees with cost function $\omega_\lambda(i,j) = \omega(i,j) + \lambda_i + \lambda_j$, where λ is an arbitrary n-dimensional vector over the reals. How does this help us? For a given graph, let $L(\lambda)$ be the length of a shortest 1-tree using cost function ω_λ and let $T_{opt}(\lambda)$ be the length of a shortest tour using the same cost function ω_λ. Then for any real vector λ, we have from before that

$$L(\lambda) \leq T_{opt}(\lambda).$$

Since each tour is a 1-tree in which each vertex is of degree 2, we have

$$T_{opt}(0) + 2\sum_i \lambda_i = T_{opt}(\lambda).$$

In particular, a shortest tour under the modified cost function ω_λ is therefore also a shortest tour under the the original cost function ω. Quantity $T_{opt}(0)$ is just the length of a shortest tour with the original cost function. Substituting for $T_{opt}(\lambda)$ in the first equation, we get the inequality

$$L(\lambda) - 2\sum_i \lambda_i \leq T_{opt}(0). \tag{5.1}$$

The Lagrange multipliers give us an extra n degrees of freedom in finding lower bounds for the optimal tour length. The task then becomes finding a λ vector that makes inequality 5.1 as tight as possible. Held and Karp prove that the objective function is convex over the space of possible values for λ, and prove convergence for an iterative ascent algorithm with certain parameters [39, 40]. At each step the λ vector is adjusted, and a new 1-tree is computed.

An optimal λ vector can also be computed directly via linear programming. Held and Karp formulate the problem of maximizing $L(\lambda)$ as a linear program. Furthermore, they show this formulation to be equivalent to a version of the standard linear programming formulation of the symmetric TSP where the integrality constraints have been removed [39, Sections 1 and 2]. The direct linear programming solution is complicated by the fact that the number of constraints grows exponentially with n. However, with care the solution can still be constructed in polynomial time. In fact, instances with n exceeding 30 000 have been solved directly using code due to Applegate *et al.* [6, 47].

When possible, we use directly computed Held-Karp lower bounds. These are reported for TSPLIB instances by Reinelt [74], by Johnson *et al.* [47][4], and by Johnson [45][5].

When directly computed Held-Karp bounds are not known, we use approximate bounds. One can approximate the Held-Karp bound by using the iterative ascent technique. Once the bound converges or some other stopping criterion is met, we take the largest bound as an approximation to the Held-Karp lower bound. The approximate bounds discovered in this way are conservative: they are never higher than the real Held-Karp lower bound. Furthermore, they are usually quite close to the directly computed Held-Karp lower bound, often within 0.01% [46, p. 224]. We use approximate bounds computed by our own implementation of these ideas, using up to 10000 1-trees for a single instance. Although our estimates might not be as tight as those of others in an absolute sense, they are still useful for comparing between the two versions of the Lin-Kernighan heuristic.

5.5 Measuring search sequence lengths

Efficient cluster compensation is designed to shorten Lin-Kernighan search sequence lengths, especially on clustered instances. To test this hypothesis, our experiments also measure these search sequence lengths. Two kinds of depths are of interest. The first is the depth at which a search sequence stops. We call this the *probe depth*. The second is the depth at which the best tour improvement found during this sequence ends. We call this the *move depth*. If no tour improvement is found during a sequence, then the move depth is zero.

For example, during a single search sequence we might find a tour improvement with a net gain of 40 units using t_1 through t_{12}, and another with a net gain of 35 units using t_1 through t_{18}. We might continue the search until reaching t_{26}, at which point all alternatives force a negative cumulative gain. The search stops at t_{26}, and we roll back

[4] Johnson *et al.* note a discrepancy between themselves and Reinelt for the Held-Karp lower bounds for two TSPLIB instances. One is a misprint in Reinelt's book, and they diagnose the other as likely being due to floating point precision differences between the two implementations.

[5] Johnson reports the directly computed Held-Karp lower bound for `fl3795` to be 28478, when rounded up to the next integer.

and commit to the changes encoded in the best net improvement found, encoded with t_1 through t_{12}. In this case, the probe depth is 26 and the move depth is 12.

The heuristic gets into trouble when it has many deep probes without an accompanying number of deep moves. The move depth histograms alone are useful for finding a good cutoff point for bounding probe lengths as discussed in Section 4.6. Each execution records a histogram of probe depths and move depths. In Sections 6.7 and 7.7.6 we shall see examples of probe depth and move depth histograms for executions on selected instances.

5.6 Kinds of test data

We use five kinds of test instances: TSPLIB instances, random geometric instances, random distance matrices, ten of Bentley's geometric classes, and generated data based on selections from each of the first four kinds of instances. The first three classes are those used by Johnson and McGeoch. Bentley's geometric classes are synthetic Euclidean instances used to test various tour construction heuristics and the 2-opt and 3-opt local search heuristics [17]. The last class is the result of the work reported in Chapter 8. We shall see in Chapter 7 that TSP instances can also be used for the weighted perfect matching problem, and we use them in this way.

TSPLIB is a library of TSP instances collected by Reinelt from the literature and from applications [73]. Most instances are symmetric, and most are even two-dimensional Euclidean instances. There are 101 symmetric instances in TSPLIB, but only 34 of them have 1000 vertices or more, and only 7 have more than 10000 vertices. We use the subset of 12 instances used by Johnson and McGeoch, the largest having 7397 vertices.

Randomly generated Euclidean instances are also commonly used by researchers. For an n vertex instance, n points are drawn from a uniform distribution over a square, and the distance between two points is the Euclidean distance. These instances typically show almost no clustering at all. In fact these instances are so regular that the ratio of the optimal tour length to $\sqrt{n}$ converges almost surely to a constant C_{OPT} as n grows without bound [14]. Furthermore, the ratio of the Held-Karp lower bound to $\sqrt{n}$ converges almost surely to a constant C_{HK} as n grows without bound [37]. Johnson *et al.* give empirical formulae for both the expected Held-Karp bound and the expected optimal tour length for this class of instances.

In the class of *random distance matrix* instances, the distance between any two vertices is randomly drawn from a uniform distribution over the unit interval $[0, 1]$. (All the distributions are independent.) They "do not have any apparent relevance, [but] they are interesting from a theoretical point of view."[46, p. 225] Even so, it is the second most popular class of instances used by researchers [47]. As long as the edge lengths are drawn from a fixed interval, the expected length of an optimal tour is bounded by a constant *independent of the number of vertices* [47].

The random Euclidean and the random distance matrix instances are typically very uniform. We therefore do not expect efficient cluster compensation to dramatically improve Lin-Kernighan on them. Equally important, the performance of Lin-Kernighan using cluster compensation should not seriously degrade on them.

Bentley implemented a suite of direct tour construction heuristics and the 2-opt and 3-opt local search heuristics (and some in between 2-opt and 3-opt). Each took advantage of k-d trees to make them run fast on geometric data [17]. To test the suite of algorithms, Bentley used eleven classes of synthetic geometric instances. Instances in each class are either similar to those found in some application for the TSP, or are designed to evoke bad behaviour in some construction heuristics. Some instance classes are easy to solve for both construction heuristics and Lin-Kernighan, and other instance classes are difficult to solve. The classes are:

uni Points randomly drawn from a uniform distribution within a square. ($U[0, 1]^2$.)

annulus Points randomly drawn from uniform distribution on a circle; a circle is a zero-width annulus.

arith x coordinates are perfect squares $(0, 1, 4, 9, 16, \ldots)$, and the y coordinate is always 0. (The distances between successive points increases arithmetically.)

ball Points randomly drawn from a uniform distribution within a circular disk.

clusnorm Choose 10 points randomly from a uniform distribution over a unit square. Then draw points randomly from normal distributions centred on those points; use a standard deviation of 0.05.

cubediam Coordinate x chosen randomly from a uniform distribution over $[0, 1]$, and then set $y = x$.

cubeedge Coordinate x chosen randomly from a uniform distribution over $[0, 1]$, and $y = 0$.

corners Four separated `uni` distributions: $U[0,1]^2$ centred at $(0,0)$, $(2,0)$, $(0,2)$, and $(2,2)$.

grid Choose n points from a square grid that contains about $1.3n$ points.

normal Each coordinate is drawn independently from a normal distribution with mean 0 and standard deviation of 1.

spokes Half the points drawn from $(U[0,1], 0.5)$ and the other half drawn from $(0.5, U[0,1])$. (This looks like a big "plus" sign (+).)

We omit class `cubeedge` from our experiments because from the perspective of the Lin-Kernighan heuristic, it is very similar to class `cubediam`. (Preliminary experiments showed that the heuristic behaves almost identically on a geometric instance and on rotations of that instance through any angle.) We expect efficient cluster compensation to be most valuable on distributions `corners` and `clusnorm` since they are the most dramatically clustered.

The last class of instances are those generated by the algorithms described in Chapter 8. Those algorithms are designed to randomly generate new instances from old instances. Certain of the algorithms are designed to preserve the cluster structure of the seed instance, while others are designed to destroy it. Each generating algorithm pays attention to the aspects of the seed instance's cluster structure to which Lin-Kernighan reacts. See Chapter 8 for more.

5.7 Details of the test bed

Our test bed includes the following instances from each of the classes described in the previous section.

We follow the TSPLIB convention of including the number of vertices in the instance as a suffix on the name of the instance. The prefix either names the source of the instance, or describes the instance class. Extra parameters describing a generated instance, such as the seed used to create a random instance, are given between the prefix and the suffix. When discussing aggregated statistics about a class of generated instances, we drop the parameters from the name. For example, an individual uniform Euclidean instance with 1000 vertices and seed 820 is named `uni.820.1000`, but results for all uniform Euclidean instances with 1000 vertices are listed under `uni.1000`.

Of the TSPLIB instances, we use `lin318`, `gr431`, `pcb442`, `att532`, `gr666`, `dsj1000`, `pr1002`, `pr2392`, `pcb3038`, `fl3795`, `fnl4461`, and `pla7397`. The number of vertices in each is encoded as a suffix of its name. Instance `gr666` uses great-circle distances on the earth, and the rest are two-dimensional Euclidean instances in the plane. The length of a shortest tour is known for each, as is the directly computed Held-Karp lower bound [6, 5, 73, 46, 45]. The source of these instances includes positions of world cities, applications in printed circuit board manufacturing, and applications in VLSI customization. Instance `dsj1000` is synthetic, designed to be difficult for local search heuristics [44]. Instance `fl3795` is singled out by Johnson and McGeoch as particularly difficult for Lin-Kernighan because it is so "pathologically clustered," even more so than instance `dsj1000` [46, p. 296, Fig. 8.7].

We use five random uniform Euclidean instances of sizes 316, 1000, and 3162. The sizes are chosen to be increasing half-powers of 10, so we can easily see whether running times increase quadratically with the number of vertices. These instances are grouped according to size, and the outcomes for each group are presented as arithmetic means.

From the random distance matrix class, we use five instances each of size 316, 1000, and 3162 vertices. As with the random Euclidean instances, the results are grouped according to size and the outcomes for each group are presented as arithmetic means.

We use five instances of size 1000 from nine of the Bentley distributions. The `arith` distribution is deterministic, producing only one instance for each size, so we only use one instance of size 1000 from it. The results are presented as arithmetic means.

We also use generated instances. See Chapter 8 for a description of the instances and for the results on those instances.

For non-geometric instances, the 20 nearest neighbour subgraph is used as the candidate subgraph. Geometric instance `gr666` uses the 20 nearest neighbour candidate subgraph as well. All other geometric instances are 2-dimensional Euclidean instances, and the candidate subgraph used for each is the union of the 20 nearest neighbour subgraph with the 20 quadrant-based nearest neighbour subgraph (5 nearest neighbours in each of the four directions).

When computing averages, we are not trying to estimate expected values. Our test beds are too small for that. Rather, we merely want to be able to highlight distinctions that might exist for those kinds of instances between using or not using efficient cluster compensation. This follows in the spirit of Johnson and McGeoch [46], where entirely different heuristics are compared.

5.8 Representation of lengths

Many instances use Euclidean distances, and hence involve square root terms. One might therefore wonder about problems in computing sums and performing comparisons between length values. Should we use algebraic methods to keep represented values exact? Since data generated from applications are usually approximate themselves, going to such measures is likely to be overkill.

We might instead use floating point numbers to represent lengths. But then how much precision should be used? Furthermore, floating point number systems are notorious for breaking the laws of arithmetic such as the associativity of addition. How are we to safely compare results from different implementations? Are the optima even well-defined?

An early version of our software once found itself in an endless loop when optimizing a particular Euclidean instance where edge lengths were not forced to be integers. On this instance the code would find an improving 6-change resulting in a small positive net gain. In the next search sequence, it computed the inverse 6-change as *also* having a small positive net gain. In effect the alternating sum of edge weights computed in the forward direction was positive, and in the reverse direction was negative! Therefore the code ended up doing and undoing the same 6-change, *ad infinitum*.

One can ameliorate floating point problems by paying attention to certain characteristics of the floating point hardware. For example, we fixed the problem from the previous paragraph by imposing two changes when using an inexact type for representing lengths. The first change prevents the algorithm from considering too-small improvements. The heuristic maintains a value we call *instance epsilon*, the length of the incumbent tour multiplied by machine epsilon. (Machine epsilon is the smallest positive value ε such that the floating point hardware distinguishes between $1+\varepsilon$ and 1.) Comparisons between the cumulative gain and the best net gain are made conservative by adding instance epsilon to one side. That is, the comparison in the Cumulative Gain Criterion 2.8 is changed from

$$cum\_gain(j) > best\_net\_gain(j)$$

to

$$cum\_gain(j) > best\_net\_gain(j) + instance\_epsilon.$$

This change protects us from finding improvements that would not even register as a change in the current tour length.

The second change splits the cumulative gain variable. Since alternating sums are notoriously difficult for floating point hardware to compute accurately, we keep the sum of the positive terms in one variable, and the sum of the negative terms in another variable. The actual cumulative gain is the difference of the two.

Such precautions are helpful, though certainly non-standard. Most researchers avoid these problems altogether by mandating that all edge lengths be integers [45]. In fact, all TSPLIB instances are specified to have integral edge weight functions. Our implementation can use either a floating point type or an integer type to represent length values; the option is selected at compile-time.

Forcing integrality has some consequences. As mentioned in Section 1.1, merely rounding lengths to the nearest integer does not preserve the triangle inequality. So some instances specify that lengths needing rounding be rounded to the next larger integer; this preserves the triangle inequality.

For our synthetic instances, distributions over a unit interval $[0, 1]$ are scaled by a factor of a million to become distributions over $[0, 10^6]$. Normal distributions are scaled in a similar way. In the case of Bentley's **corners** distribution, the four corners also have their coordinates scaled by a million, to become $(0, 0)$, $(2 \cdot 10^6, 0)$, $(0, 2 \cdot 10^6)$, and $(2 \cdot 10^6, 2 \cdot 10^6)$. In the random distance matrices edge lengths are drawn from the uniform distribution over $1, \ldots, 10^6$; note that zero distances are excluded. These rules follow the practice of Johnson *et al.* and of Bentley [47, 17].

5.9 Summary

This chapter describes our experiments. They are designed so that their outcomes can provide a fair assessment of the utility of efficient cluster compensation. The experiments follow the methodology of other researchers' work so that the results may be meaningfully compared. Lastly, enough detail is given so that our experiments are repeatable by others.

Chapter 6

Experimental results for the TSP

This chapter presents the results of the experiments with the Lin-Kernighan heuristic for the TSP, both with (LKcc) and without (LK) efficient cluster compensation. The results are based on experiments with the implementation described in Chapter 4. The experimental methodology was given in Chapter 5, including a description of the test bed of instances. The results for instances generated from other instances are postponed until Chapter 8 where the generating algorithms themselves are described.

We discuss both the quality of the tours produced by the two variants of the heuristic, and the running times required to produce those tours. We also show sample probe and move depth profiles for selected instances, and describe how they correlate with observed running times and tour quality.

Finally, we draw some conclusions about the effectiveness of efficient cluster compensation.

6.1 Breakdown of run times

All the reported run times include the time required for preprocessing. Table 6.1 gives an idea of how much time is spent during each phase of a run of Lin Kernighan for the TSP. It lists the times for each phase in a typical n iteration run by Iterated Lin-Kernighan when using cluster compensation. The two instances are the geometric TSPLIB instance `pcb3038` having 3038 vertices, and the random distance matrix instance `dsjr.55.3162` having 3162 vertices. The candidate subgraph used for `dsjr.55.3162` is the 20 nearest neighbour subgraph. The candidate subgraph use for `pcb3038` is the union of the 20 nearest neighbour subgraph and the 20 quadrant-based nearest neighbours (For each

Phase	Time (sec) pcb3038	Time (sec) dsjr.55.3162
Build a 2-d tree	0.01	n/a
Build a minimum spanning tree	0.19	2.51
Process the minimum spanning tree in preparation for later cluster distance queries	0.04	0.02
Build a candidate subgraph	0.83	8.16
Construct a randomized greedy tour	0.19	8.25
Lin-Kernighan optimization phase (first iteration)	1.29	2.25
$n-1$ subsequent iterations of Iterated Lin-Kernighan	109.30	272.16

Table 6.1: Breakdown of running time into phases for Iterated Lin-Kernighan with cluster compensation. The times are taken for a 3038 iteration run on geometric instance `pcb3038`, and a 3162 iteration run on random distance matrix instance `dsjr.55.3162`.

vertex v, 5 neighbours are chosen in each of the four quadrants surrounding v.).

Iterated Lin-Kernighan without cluster compensation skips two of these initial phases: it does not build a minimum spanning tree, nor does it preprocess that tree for subsequent online cluster distance queries. However, we will see that the base Lin-Kernighan heuristic generally spends more time in the Lin-Kernighan optimization phases.

The preprocessing steps are fast because they use the k-d tree. Runs on non-geometric instances avoid building a k-d tree, but they spend more time enumerating the edges when building a minimum spanning tree, a candidate subgraph, and a greedy tour. Prim's algorithm for constructing a minimum spanning tree examines each edge once, so running time for that phase increases quadratically with the number of vertices. For an unstructured graph, the time to build a nearest neighbour candidate subgraph increases at least as fast as the number of edges, since all the edges must be enumerated in order to find the shortest ones. Both phases become bottlenecks when processing larger unstructured graphs. However, Prim's algorithm is fast enough so the time to find a minimum spanning tree is usually dominated by the time to build the candidate subgraph and the greedy tour.

The preprocessing phases, from building a 2-d tree to building the candidate subgraph, are performed once only, regardless of the number of Iterated Lin-Kernighan iterations used. The cost of preprocessing is therefore amortized to insignificant levels if we use a large number of iterations. Often the first Lin-Kernighan iteration takes a long time, making preprocessing time only a small concern.

6.2 Comparing performance

All tables comparing the performance of Lin-Kernighan with and without cluster compensation use differences to compare percent excess over Held-Karp, and ratios to compare running times. We use the differences to compare percentages because the differences are small, and because the Held-Karp lower bound depends upon the instance only. As noted in Chapter 5, some of the Held-Karp bounds are directly computed while others are approximated values computed by iterative ascent.

We show the ratio of running times rather than difference for several reasons. First, the differences in running times can be quite large, often larger than the smaller running time itself. Second, the heuristic has many tunable parameters that greatly affect running times. The ratio of running times can be used as an estimate of the relative speeds of the two variations of the heuristic when using different parameter settings. Finally, in the context of ever-faster computing platforms, ratios of running times are more meaningful than differences in running times.

6.3 TSPLIB instances

Table 6.2 compares the quality of the tours generated by the two versions of the Lin-Kernighan heuristic on TSPLIB instances. Ten experiments were performed for a single iteration and $n/10$ iterations; six experiments were performed for n iterations. For comparison purposes, we show the difference between the average percentage excesses for each instance. A positive difference indicates that base Lin-Kernighan finds better tours than Lin-Kernighan with cluster compensation. Table 6.3 shows the average running times for the same experiments. For comparison purposes, we show the ratio of the running times. A ratio greater than 1 indicates that Lin-Kernighan with cluster compensation ran faster than base Lin-Kernighan.

Table 6.2 shows there is a small loss in the quality of the tours when cluster compensation is used. After one iteration of Lin-Kernighan, there is a loss of up to 0.5% in

Percentage excess

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
lin318	1.12	1.62	0.50	0.58	0.81	0.23	0.37	0.42	0.05
pcb442	1.52	1.49	-0.03	1.05	1.05	0.00	0.69	0.75	0.06
att532	1.81	1.89	0.08	1.30	1.32	0.02	1.06	1.05	-0.01
gr666	2.32	2.04	-0.28	1.23	1.24	0.01	0.74	0.73	-0.01
dsj1000	2.07	2.23	0.16	1.31	1.28	-0.03	0.94	0.91	-0.03
pr1002	2.28	2.60	0.32	1.73	1.76	0.03	1.15	1.17	0.02
pcb1173	2.39	2.38	-0.01	1.50	1.68	0.18	1.06	0.98	-0.08
pr2392	2.90	2.94	0.04	2.03	1.95	-0.08	1.40	1.50	0.10
pcb3038	2.05	2.06	0.01	1.38	1.43	0.05	1.01	1.02	0.01
fl3795	3.86	4.25	0.39	1.34	1.49	0.15	1.11	1.14	0.03
fnl4461	1.55	1.63	0.08	1.07	1.12	0.05	0.75	0.76	0.01
pla7397	2.06	2.01	-0.05	0.98	1.02	0.04	0.73	0.74	0.01

Table 6.2: Quality of output produced for TSPLIB instances by the Lin-Kernighan heuristic for the TSP. Quality is measured as the percent excess over Held-Karp lower bound. "LK" is the base Lin-Kernighan heuristic, and "LKcc" is the Lin-Kernighan heuristic using efficient cluster compensation.

Running time

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
lin318	0.72	0.19	3.79	3.25	0.57	5.70	25.78	3.83	6.73
pcb442	0.94	0.34	2.76	4.46	1.70	2.62	37.57	12.63	2.97
att532	1.06	0.49	2.16	4.97	2.29	2.17	37.99	15.28	2.49
gr666	17.13	6.53	2.62	94.46	22.50	4.20	745.36	138.83	5.37
dsj1000	16.99	2.70	6.29	166.98	21.84	7.65	1606.73	162.12	9.91
pr1002	4.15	1.45	2.86	18.98	6.31	3.01	132.58	48.33	2.74
pcb1173	1.41	0.94	1.50	7.13	4.12	1.73	51.55	25.64	2.01
pr2392	3.59	2.39	1.50	22.74	13.23	1.72	157.18	89.64	1.75
pcb3038	3.19	2.40	1.33	26.14	15.71	1.66	197.38	114.89	1.72
fl3795	148.06	51.37	2.88	3345.49	1598.92	2.09	30859.90	14139.33	2.18
fnl4461	4.05	4.08	0.99	38.07	27.80	1.37	286.95	215.42	1.33
pla7397	53.33	34.88	1.53	1057.16	535.95	1.97	10329.43	3802.79	2.72

Table 6.3: Time taken on TSPLIB instances by the Lin-Kernighan heuristic for the TSP. Time is measured as user CPU seconds.

Percentage excess

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
uni.316	1.82	1.90	0.08	1.27	1.35	0.08	1.04	1.07	0.02
uni.1000	1.78	1.81	0.03	1.19	1.23	0.04	0.89	0.90	0.00
uni.3162	1.93	1.99	0.06	1.30	1.34	0.03	0.93	0.94	0.01

Table 6.4: Quality of output produced by Lin-Kernighan for the TSP on uniform geometric instances. Quality is measured as the percent excess over Held-Karp lower bound.

excess over Held-Karp, but the difference is usually much smaller. By n iterations, the loss is very small, always less than 0.1%. In some cases cluster compensation helps the heuristic find better tours.

However, the running times tell the real story. Cluster compensation makes the heuristic run anywhere from 1.3 times faster to almost 10 times faster. The only exception is a slowdown of 1% for a single iteration of `fnl4461`, and that is a difference of only 0.03 seconds.

6.4 Uniform geometric instances

Tables 6.4 and 6.5 show the results of experiments on uniform geometric instances with 316, 1000, and 3162 vertices. This instance class is the same as Bentley's `uni` class. Instances of this type are not sharply clustered, and are very easy for the Lin-Kernighan heuristic. The main difficulty in running experiments on larger instances is computing the Held-Karp lower bound, although one can rely on expected Held-Karp values reported by Johnson *et al.* [47]

The results are similar here as for the TSPLIB instances. Cluster compensation causes only a very small loss in tour quality, 0.02% or less, and it speeds up the heuristic by roughly 50% to 100%. Indeed, the speedup ratio increases with the number of iterations. This effect is cannot be accounted for by the amortization of the extra preprocessing costs over more iterations. Cluster compensation is saving work.

The instances sizes are increasing half-powers of 10: $316 \approx 10^{2.5}$, $1000 = 10^3$, and $3162 \approx 10^{3.5}$. If the running time for Lin-Kernighan heuristic increases faster than

Running time

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/LKcc	LK	LKcc	LK/LKcc	LK	LKcc	LK/LKcc
uni.316	0.33	0.21	1.56	1.36	0.72	1.88	9.70	4.79	2.02
uni.1000	1.17	0.76	1.54	7.96	4.07	1.96	61.18	30.47	2.01
uni.3162	3.52	2.56	1.37	29.81	17.90	1.67	236.51	138.99	1.70

Table 6.5: Time taken on uniform geometric instances by the Lin-Kernighan heuristic for the TSP.

quadratically, then we would expect the running times to increase by 10 times or more from one size to the next. That is not the case here, even for n iterations with increasing n.

6.5 Uniformly generated distance matrices

Table 6.6 compares the quality of the tours generated by the two versions of the Lin-Kernighan heuristic on instances generated by drawing distances randomly from a uniform distribution. Three instance sizes were used; 316, 1000, and 3162 vertices. Five instances were generated for each size, and ten experiments were performed on each instance.

Table 6.7 shows the average running times for the same experiments. For comparison purposes, the ratio of the average running times are listed.

Cluster compensation consistently allows Lin-Kernighan to provide slightly *better* tours than usual, by up to 0.18%. For larger instances, the extra preprocessing costs for cluster compensation initially dominates the time savings cluster compensation provides during the optimization phase. With more iterations the savings eventually outweigh the startup costs, and even the speed up *ratio* increases with the number of iterations.

The uniform geometric case of the previous section showed that the transition point where cluster compensation becomes favourable time-wise is less than a single Iterated Lin-Kernighan iteration. In the non-geometric case the transition point is still below a single iteration for smaller instances; as instances grow larger a larger number of iterations are needed before cluster compensation is a time win.

Percentage excess

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
dsjr.316	2.21	2.16	-0.05	1.03	1.02	-0.01	0.47	0.46	-0.01
dsjr.1000	3.29	3.11	-0.18	1.75	1.72	-0.03	0.96	0.92	-0.04
dsjr.3162	4.22	4.07	-0.15	2.76	2.66	-0.10	1.93	1.82	-0.11

Table 6.6: Quality of output produced for the class of uniformly generated distance matrices by the Lin-Kernighan heuristic for the TSP. Quality is measured as the percent excess over an estimated Held-Karp lower bound.

Running time

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
dsjr.316	0.51	0.44	1.17	2.28	1.51	1.51	16.40	10.11	1.62
dsjr.1000	4.51	3.53	1.28	32.99	18.78	1.76	231.21	125.91	1.84
dsjr.3162	19.54	21.36	0.91	67.76	56.98	1.19	397.56	301.17	1.32

Table 6.7: Time taken on the uniform distance matrix class of instances by the Lin-Kernighan heuristic for the TSP. Time is measured as user CPU seconds.

As with the uniform geometric instances, the running times do not increase quadratically with the number of iterations, at least over the upper range of instance sizes. Again, this holds true even for n iteration runs with increasing n. However, we expect the preprocessing time for cluster compensation to eventually dominate the time used in the optimization phase as instance sizes increase.

Table 6.6 also displays the beginnings of the trend noted by Johnson and McGeoch [46, p. 297]. As instance sizes increase, Lin-Kernighan in general produces worse tours for random distance matrices. This is the case for both versions of the heuristic. We shall see that this trend is repeated and more obvious for weighted perfect matchings.

6.6 Bentley's distributions

Table 6.8 compares the quality of the tours generated by the two versions of the Lin-Kernighan heuristic on Bentley's classes of instances. Class `arith` is deterministic, always specifying the same instance, so only one instance was generated for it, and five experiments were performed on that instance. Five instances were generated from class `cubediam`, and one experiment was performed on each. Five instances were generated from each of the other classes, and three experiments were run on each of those instances. Table 6.9 shows the average running times for the same experiments. For comparison purposes, the ratio of the average running times are listed.

Classes `arith`, `cubediam`, and `spokes` are peculiar and are discussed in the next section. The results from the remaining classes are unsurprising. Both versions of the heuristic find good tours, usually less than 2 percent above optimal. Only in two cases (`uni` and `clusnorm`) does cluster compensation make Lin-Kernighan find tours more than a tenth of a percent worse than it would otherwise. Even then, that occurs not after a single iteration, but after $n/10$ iterations.

The running times vary a great deal both across instance classes and between the two versions of the heuristic. The big surprise is the running time for class `cubeedge`, topping out at three hours or more for 100 iterations on a 1000 city problem. The next section gives possible explanations for these unusually long runs. For a large number of iterations, cluster compensation occasionally makes Lin-Kernighan slower, by up to 9 percent on classes `annulus` and `grid`. Those two have a very regular structure spread out in two dimensions, and hence insignificant clustering. The most impressive result for cluster compensation is that it speeds up Lin-Kernighan by up to 30 times on the very

Percentage excess

Instance	1 iteration			$n/10$ iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
uni.1000	1.77	1.85	0.07	1.14	1.25	0.11
annulus.1000	0.08	0.08	0.00	0.08	0.08	0.00
arith.1000	90.22	90.22	0.00	90.22	90.22	0.00
ball.1000	1.88	1.88	0.00	1.24	1.26	0.02
clusnorm.1000	2.65	2.37	-0.28	1.10	1.21	0.11
cubediam.1000	48.16	48.16	0.00	48.16	48.16	0.00
corners.1000	2.58	2.66	0.08	1.88	1.86	-0.03
grid.1000	1.04	1.08	0.04	0.63	0.63	0.00
normal.1000	2.02	1.95	-0.08	1.17	1.16	-0.02
spokes.1000	5.73	5.73	0.00	5.73	5.73	0.00

Table 6.8: Quality of output produced for Bentley's classes of instances by the Lin-Kernighan heuristic for the TSP. Quality is measured as the percent excess over an estimated Held-Karp lower bound.

Running time

Instance	1 iteration			$n/10$ iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
uni.1000	1.11	0.77	1.45	7.52	4.16	1.81
annulus.1000	0.49	0.34	1.45	103.56	114.37	0.91
arith.1000	5.32	5.71	0.93	683.33	292.27	2.34
ball.1000	1.14	0.75	1.51	7.70	3.71	2.08
clusnorm.1000	24.93	5.76	4.33	265.71	69.86	3.80
cubediam.1000	312.10	246.62	1.27	14239.84	10819.75	1.32
corners.1000	32.92	0.90	36.74	481.05	16.50	29.16
grid.1000	0.69	0.85	0.81	3.44	3.72	0.92
normal.1000	3.17	0.96	3.30	14.42	4.30	3.35
spokes.1000	14.69	15.84	0.93	230.26	228.12	1.01

Table 6.9: Time taken on Bentley's classes of instances by the Lin-Kernighan heuristic for the TSP. Time is measured as user CPU seconds.

sharply clustered `corners` distribution. Other timing results show cluster compensation speeding up the heuristic by between 30 and 300 percent.

There is an interesting correlation between the two tables. The running time for many-iteration Lin-Kernighan with cluster compensation is longest when it fails to improve the tour after the first iteration. (Base Lin-Kernighan runs a long time on class `corners`, but that is explained by its sharply clustered structure; in contrast, Lin-Kernighan with cluster compensation runs quickly on `corners`.) Either both phenomena have the same root cause, such as the peculiar structure of those inputs, or one causes the other. Long running times have several causes: deep searches, much backtracking, and large *Active* queues. Since the *Active* queue contains only eight cities between iterations, we eliminate the possibility that failing to improve the tour causes long running times. Instead, we suspect that the peculiar structure of the instances is the root cause of both. First, optimal and near-optimal tours are easy to find. Second, any perturbation is quickly undone, but perhaps involving large numbers of vertices, and therefore blowing up the size of the *Active* queue. These two features of an instance can cause the observed correlation between the two tables. The next section analyzes classes `arith` and `cubediam` in more detail.

6.6.1 Points on a line segment

Several of the Bentley classes evoke unusual results. The most glaring differences are the large percent excess over Held-Karp for classes `arith`, `cubediam`, and to a lesser extent, `spokes`. Those three classes are also outliers when it comes to running times.

Classes `arith` and `cubediam` are peculiar: each such instance has all its vertices lying on a line segment. Any tour passing the position of each vertex exactly twice is optimal.[1] This is a special case of how one can construct an optimal tour for a set of points on a convex hull by following the convex hull path either forward or backward. Any other ordering induces a tour with path crossings; such tours can always be improved by applying a 2-change to uncross the path.

The Lin-Kernighan heuristic takes a long time on these instances. The greedy tour for points on a line segment consists of all the edges between consecutive points on the line segment and closes up that Hamiltonian path by adding the edge between the two end points. The randomized greedy tours are likely to also include the long edge. That last edge is very long, half the optimal tour length. Any time the Lin-Kernighan heuristic breaks that long edge, the cumulative gain rises enormously, and the heuristic is therefore encouraged to perform very deep searches. When the tour is locally optimal —as the greedy tour is— the problem is compounded since the heuristic ends up backtracking over all its cases, each one becoming a deep search. Cluster compensation does not help much in these cases since the cluster distance between the two endpoints is likely to be very short relative to the actual distance between the endpoints.

The greedy heuristic finds an optimal tour for these instances. Randomized greedy tours are likely to be very close to optimal. In our experiments, the Lin-Kernighan optimization phase is given a randomized greedy tour to try to improve. The optimal tour length for the `arith` class can be computed analytically.[2] Examining the raw data shows that the randomized greedy algorithm finds an optimal tour roughly 80% of the time, and the suboptimal tours it finds are very close to optimal, usually to within one part in a million. In all the runs for this instance, the Lin-Kernighan heuristic finds an optimal tour. We strongly suspect the same behaviour occurs for the `cubeedge` classes. The

[1] We are ignoring the effect of rounding that might make a difference in the `cubediam` class of instances.

[2] The optimal tour length for an n vertex `arith` instance is $2 \cdot (n-1)^2$.

high percentage excesses listed in Table 6.8 for classes `arith` and `cubediam` are therefore a result of poor lower bounds provided for these instances by the iterative ascent approximation to the Held-Karp lower bound. There are two possibilities for this failure. First, the parameters used for the iterative ascent may be insufficient for convergence, and a Held-Karp bound discovered through direct linear programming methods would find a tighter bound. The alternative is that the "Held-Karp gap," the relative difference between the exact Held-Karp lower bound and the optimal tour length for a given instance, is significantly larger for these line segment instances than observed so far for other instance classes [47].

We suspect that the `spokes` distribution evokes similar behaviour from the iterative ascent Held-Karp heuristic, the greedy tour heuristic, and the Lin-Kernighan optimization heuristic. It has a similar line-oriented structure, two line segments that cross as in a plus sign (+). A second clue is that both versions of the Lin-Kernighan heuristic, with and without cluster compensation, fail to improve the tour after the end of the first iteration. Furthermore, the individual instances used in the experiments are very similar, but the approximated Held-Karp bounds computed for each vary a great deal while the individual upper bounds computed by the Lin-Kernighan heuristic do not vary much. The same behaviour is observed for the `cubediam` distribution.

The large excess over the lower bounds are a red herring, but the long running times are still worrisome. It is ironic that such simply structured instances with easily computed optimal (or near-optimal) tours could force the Lin-Kernighan heuristic to run so long.

6.6.2 Summary

Overall, cluster compensation speeds up Lin-Kernighan the most when clustering is sharpest, on classes `corners`, `clusnorm`, and `normal`. (One might not think of `normal` instances as clustered, but the cities on the outside fringe are actually quite far apart. A minimum spanning tree for such an instance therefore has long edges toward the outside of the instance, and the cluster distances therefore grow relatively large.) The highly uniform `grid` class is the only case where cluster compensation makes Lin-Kernighan run more slowly both after a single iteration and many iterations. Even then, the slowdown is not more than nine percent.

The quality of the tours is good, usually within two percent of optimal, once adjustments to the approximated Held-Karp bound are taken into account. Both Lin-Kernighan variations are well matched with respect to each other, producing tours of similar quality.

6.7 Probe depth and move depth profiles of sample cases

Recall from Section 5.5 that the probe depth of a search is the maximum t index examined during that search, and that the move depth is the maximum t index of the improving move, if any, discovered during that search. Aggregate probe depths measure the amount of work performed by the heuristic, while aggregate move depths measure the progress made by the heuristic. Cluster compensation is designed to reduce the overall amount of work performed by the heuristic by shortening search sequences on average.

Figures 6.1, 6.2, and 6.3 are histograms of probe and move depths for typical $n/10$ iteration runs for instances `grid.828.1000`, `uni.820.1000`, and `dsj1000`, respectively. These instances were chosen because they are all the same size yet are examples forcing the heuristic to display behaviours from across the spectrum. Instance `grid.828.1000` is one of the few instances causing cluster compensation to slow down Lin-Kernighan for the TSP, by about 10%. Instance `uni.820.1000` evokes more typical behaviour, where Lin-Kernighan with cluster compensation runs between 50 and 80 percent faster than base Lin-Kernighan. Instance `dsj1000` makes Lin-Kernighan with cluster compensation run 6 to 10 times faster than base Lin-Kernighan. Absolute running times are also much longer on `dsj1000` than the other two.

In each of the three cases, the move depth histogram for LKcc closely tracks the move depth histogram for LK. In other words, Lin-Kernighan with cluster compensation is finding improving k-edge exchanges for similar k values with the same frequency that ordinary Lin-Kernighan does. We presume that overall the improvements are of similar magnitude as well since the ending tour lengths are also very close.

However, the probe depth profiles tell the tale on running times.

For `grid.828.1000`, the probe depths for LKcc closely match LK. That is, cluster compensation does not greatly change the amount of search work performed by the heuristic, as measured by depths alone. On closer inspection we notice that LKcc performs fewer short (depth 10) searches and more long (depth 106 and 108) searches than LK does. Note that the amount of work for a search of depth l increases at least linearly with l, so the peaks at the upper end are quite significant. The extra running time on

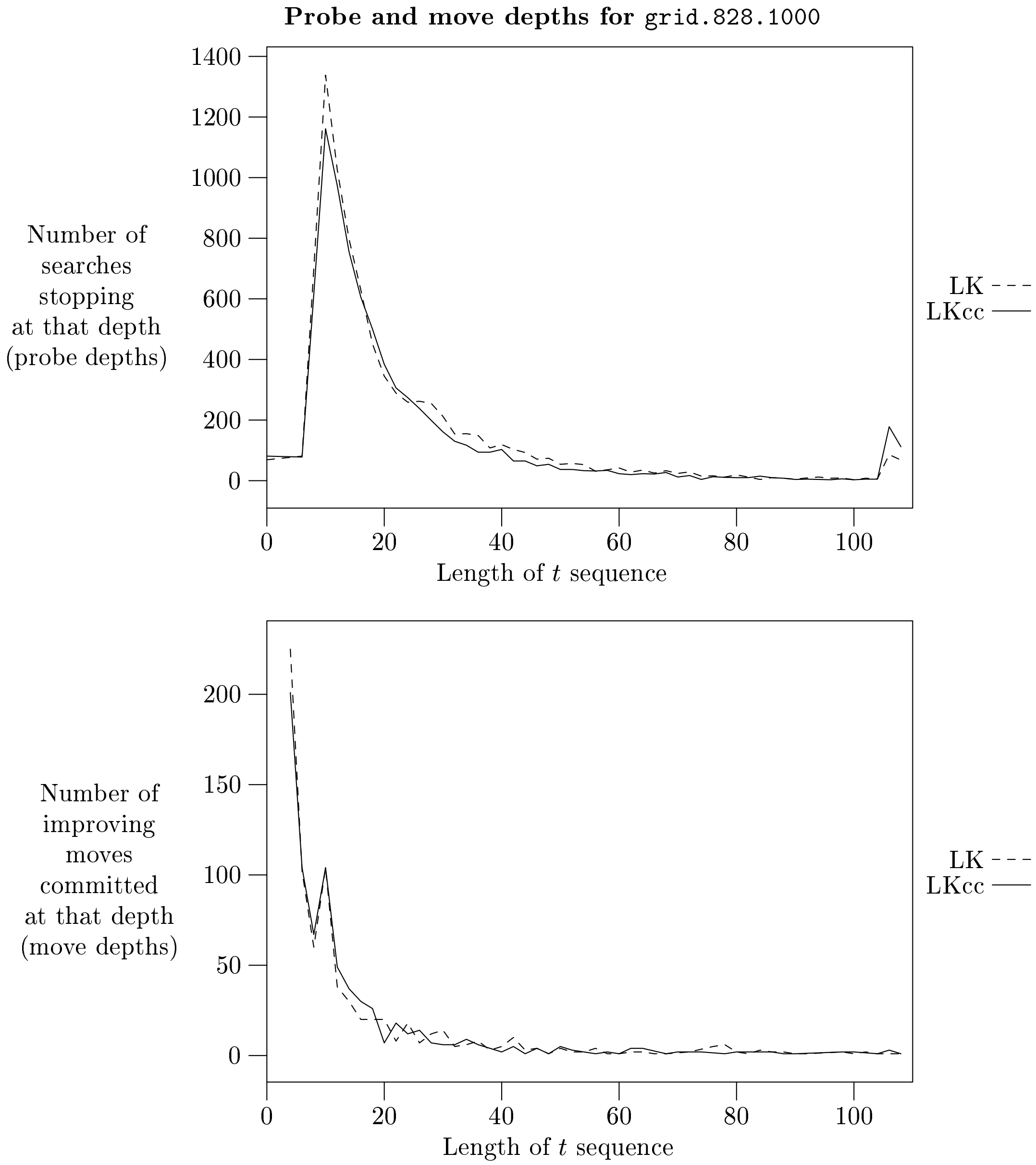


Figure 6.1: Probe and move depth profiles for a $n/10$ iteration run on instance `grid.828.1000`.

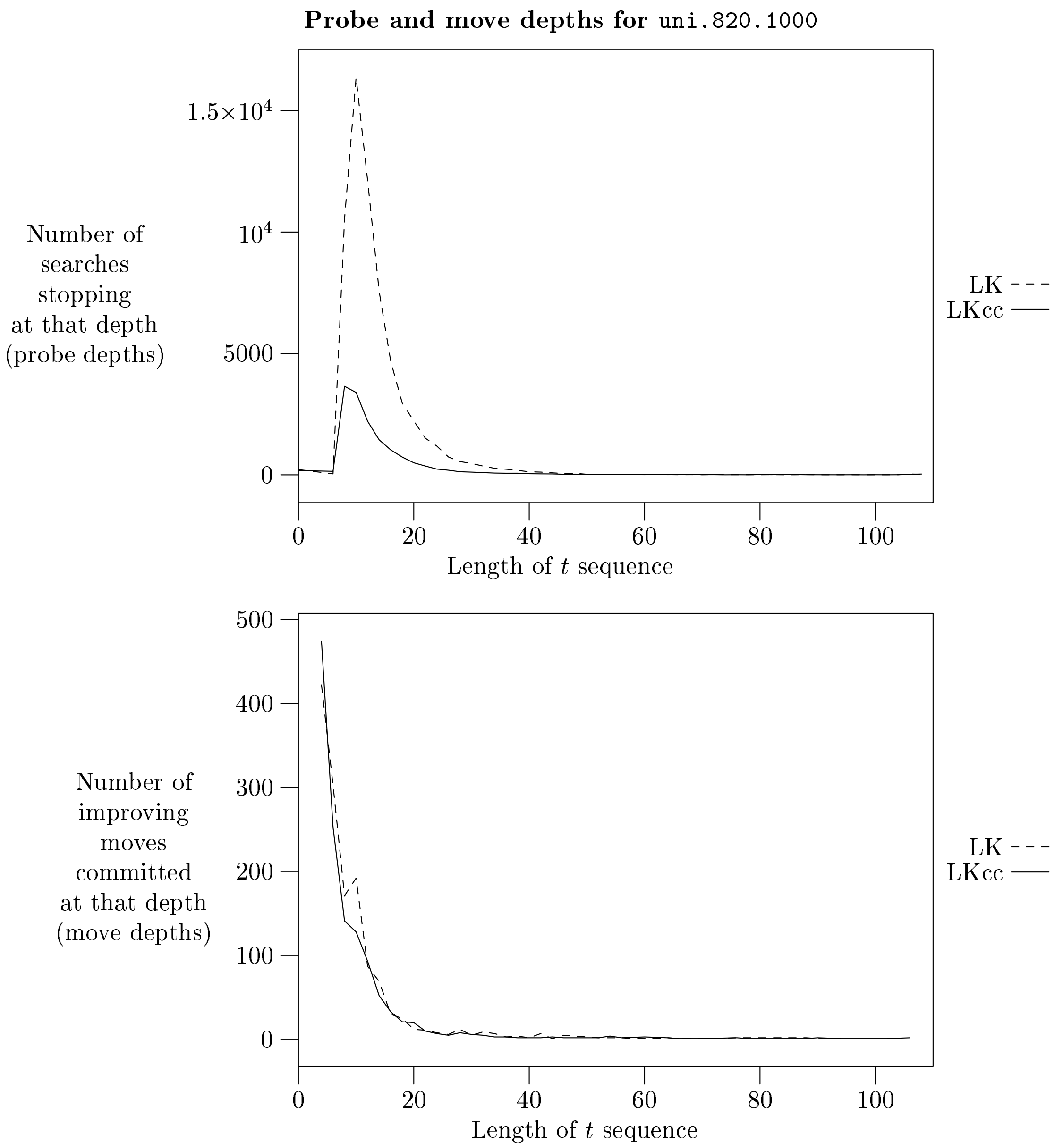


Figure 6.2: Probe and move depth profiles for a $n/10$ iteration run on instance `uni.820.1000`.

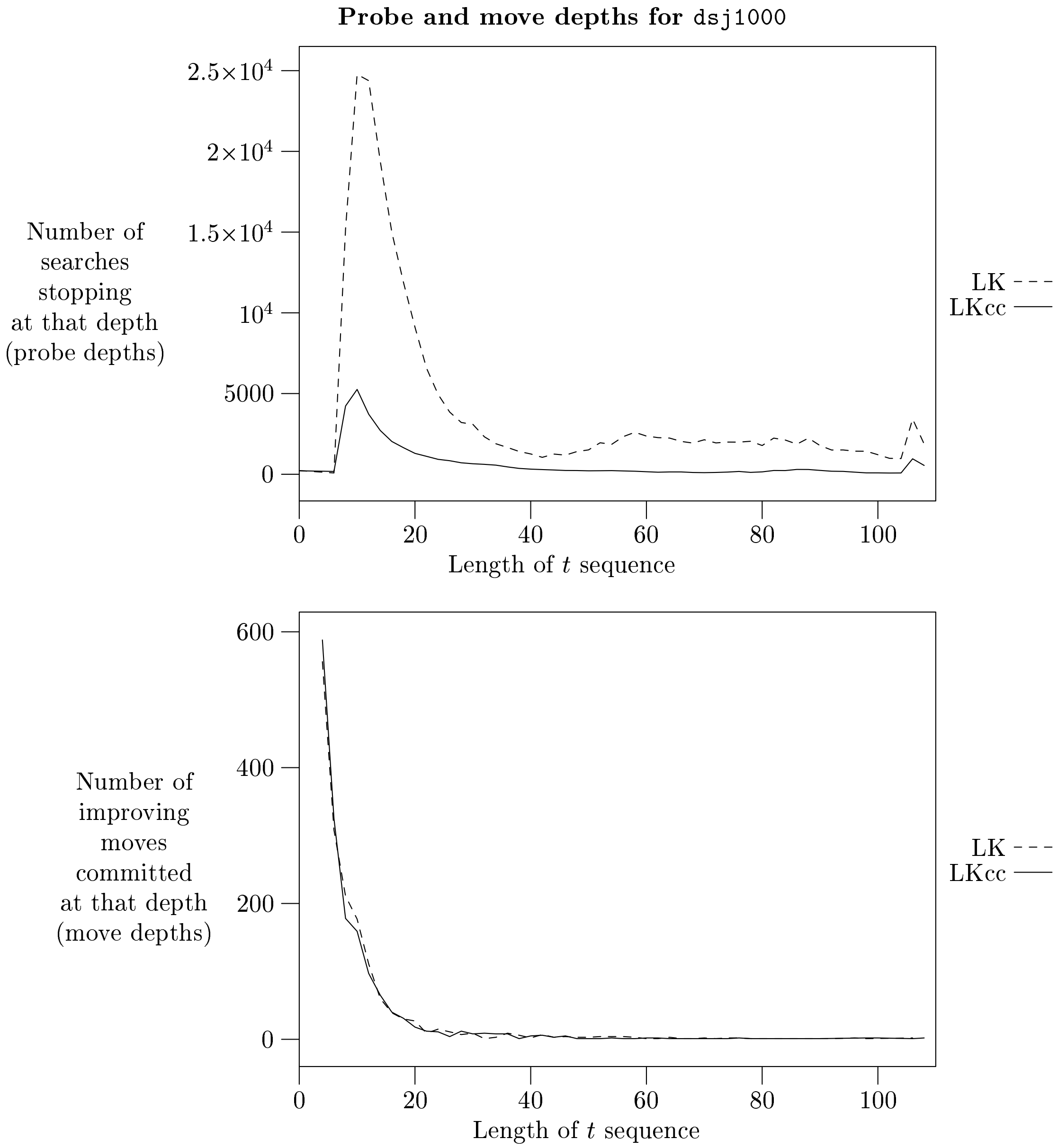


Figure 6.3: Probe and move depth profiles for a $n/10$ iteration run on instance `dsj1000`.

this instance is therefore explained by three factors: preprocessing overhead, the added work of computing and subtracting cluster distance inside the heuristic's inner loop, and the extra deep searches. With search depths closely matched overall, the extra overhead inside the inner loop may become especially significant.

For `uni.820.1000` the probe depths are always lower for LKcc than for LK. Cluster compensation is dramatically reducing the frequency of searches, especially below a depth of 20 cities (or 10 edge exchanges). This accounts for the time savings produced by cluster compensation.

For `dsj1000` the probe depths are always lower for LKcc than for LK. On this instance LK performs roughly 2000 or more probes at *all* depths, explaining the extraordinarily long running times for an instance of that size. Cluster compensation reduces the probe depth profile, and therefore running times, to more ordinary levels. LKcc performs roughly only 30% more searches at depths of 60 cities and fewer on `dsj1000` than on `uni.820.1000`. However, LKcc still displays a small peak at depths of 106 and 108 cities, enough to explain why LKcc takes 21 seconds for a $n/10$ iteration run on `dsj1000` but only 4 seconds for a $n/10$ iteration run on `uni.8.1000`.

There are a few common trends across the instances. First, Lin-Kernighan with cluster compensation produces similar improvements to those produced by ordinary Lin-Kernighan. Second, improving moves are found on only a small fraction of searches. Third, higher probe depth profiles correlate with longer absolute running times. Finally, cluster compensation really does reduce search depths overall as it was designed to do, and that this reduction in work translates into shorter running times.

6.8 Summary

Efficient cluster compensation often greatly speeds up the Lin-Kernighan heuristic. Over many kinds of instances, running time improvements of 50% to 100% are common. The speedup ratios are data dependent. Cluster compensation provides greater time savings on more sharply clustered instances. Only on the most regular and uniformly structured instances does cluster compensation slow down the heuristic. Even then, the lost time amounts to less than 10% after a single Iterated Lin-Kernighan iteration. With more iterations cluster compensation again proves to be a time saver on those instances.

The sample probe and move depth profiles show that cluster compensation does reduce the work performed by the heuristic. In contrast with raw running times, the depth profiles are independent of computing platform, programmer skill, and the amount of effort put into tuning the implementation code.

There is very little or no loss at all in the quality of the tours found when cluster compensation is used. The differences in tour lengths are less than a tenth of a percent, for an equal number of iterations.

Cluster compensation affects both running time and tour quality simultaneously. From a practical point of view, cluster compensation gives us the luxury of performing more iterations in the same allotted time. Since the quality of the incumbent tours improves with more iterations, cluster compensation holds the promise of more "bang for the buck" when using Lin-Kernighan. This can only improve Lin-Kernighan's reputation as a formidable optimization heuristic.

Chapter 7

Lin-Kernighan for minimum weight perfect matching

This chapter describes how the general Lin-Kernighan strategy described in Chapter 2 and in particular Section 2.2 can be adapted for finding low-weight perfect matchings. That is, we describe a Lin-Kernighan heuristic for minimum weight perfect matchings. Rohe [76] also reports experience with a similar Lin-Kernighan heuristic for minimum weight perfect matching.

Many of the features of Lin-Kernighan for the TSP can be carried over to Lin-Kernighan for matching. In many ways the matching heuristic is much simpler.

7.1 Why choose a problem in P?

Minimum weight perfect matchings can be found in polynomial time. Algorithms based on a primal-dual approach to a linear programming formulation of the problem can find optimal solutions for an n node weighted graph in $O(n^3)$ time [68]. Even faster times are possible for geometric instances [84]. Why bother with a heuristic when we have exact algorithms that run in polynomial time? In particular, why look at the effect of cluster compensation on the behaviour of Lin-Kernighan for perfect matching? There are two main reasons, one practical and the other theoretical.

7.1.1 Practical reasons

When instances get very large, cubic or even quadratic running times can be considered impractical. Even for Lin-Kernighan for the TSP, run times grow sub-quadratically on uniform geometric inputs [46, p. 295]. Trading poorer quality answers for less computational effort can be useful when tackling polytime solvable problems as much as it is useful when tackling problems with NP-hard analogues. For example, Bentley used 2-Opt to find low-weight perfect matchings in an approximate version of the Christofides construction algorithm for the TSP [17]. Lin-Kernighan may provide a useful tradeoff for practical problems, even in comparison to highly scalable implementations [8, 27] of the exact algorithms. We want to know whether cluster compensation improves the behaviour of Lin-Kernighan in this domain.

7.1.2 Theoretical reasons

Using Lin-Kernighan for weighted perfect matching is interesting from a theoretical perspective as well. Polytime algorithms for matching can often be viewed as local search algorithms, where each step consists of finding an alternating *path* in the graph to improve the current matching [68].

In this sense these matching algorithms are very similar to the concrete Lin-Kernighan heuristic of Section 2.2, a local search algorithm that finds improving alternating *cycles*. The PLS-completeness of Lin-Kernighan for the TSP shows us that relying only on improving alternating cycles is not entirely satisfying from a theoretical perspective. A consequence of this fact is that one can construct a family of TSP instances and associated starting tours that force Lin-Kernighan to take an exponential number of steps. Hill climbing (or descending, as the case may be) in the space of tours is difficult if only "improving alternating cycle" moves are allowed. The space of tours is rather ornery from this perspective. Yet polytime algorithms for matching are based on just this kind of downhill move. So in broad terms we know the space of matchings is structured rather more nicely than is the space of tours.

Weighted perfect matching thus provides an interesting domain in which cluster compensation can prove itself. Does cluster compensation provide benefits to Lin-Kernighan in the well-structured space of matchings, similar to the benefits it provides in the more ornery space of tours? Using both kinds of test beds allows us to demonstrate the versatility of cluster compensation.

7.2 An observation about perfect matchings

The applicability of Lin-Kernighan to finding low-weight perfect matchings rests on the following observation about perfect matchings: the difference between two perfect matchings is just a set of disjoint cycles. This is very similar in spirit to a preliminary result in the theory of general matchings of graphs, namely that a matching M in a graph G is maximum if and only if there is no augmenting path in G with respect to M. (For example, see Lemma 10.1 and Theorem 10.1 in [68].) The point is not that the result about perfect matchings is novel, but that the proof gives us insight into the applicability of Lin-Kernighan to matching problems.

Lemma 7.1 *Let M_α and M_β be distinct perfect matchings for the complete graph $G = (V, E)$, $M_i \subset E$. Then the symmetric difference between the two matchings, $M_\alpha \oplus M_\beta = (M_\alpha \setminus M_\beta) \cup (M_\beta \setminus M_\alpha)$, consists of disjoint alternating cycles. Each such cycle alternates between edges in M_α and edges in M_β.*

Proof: The proof is by strong induction on the number of elements in the symmetric difference. The proof is constructive, yielding an algorithm for transforming M_α into M_β one alternating cycle at a time. The tactic for finding each cycle is akin to finding a loose thread in a sweater and just pulling.

Let $e_1 = (t_1, t_2)$ be some edge in $M_\alpha \setminus M_\beta$. There is a unique edge $e_2 = (t_2, t_3) \in M_\beta$. Each vertex has exactly one mate in any perfect matching, thus $e_2 \in M_\beta \setminus M_\alpha$. Continuing from t_3, we can find the unique edge $e_3 = (t_3, t_4) \in M_\alpha$ and can similarly show that $e_3 \in M_\alpha \setminus M_\beta$.

In general, given vertex t_j we can always find a unique mate t_{j+1} in the appropriate perfect matching: if j is odd, we find the unique edge $e_j = (t_j, t_{j+1})$ in M_α; if j is even, we find the unique edge $e_j = (t_j, t_{j+1})$ in M_β. In particular, edges e_h and e_i are in the same matching if and only if h and i are either both even or both odd. Since the graph is finite, we must eventually loop back onto the t sequence, *i.e.*, we find a smallest L for which $t_L = t_i$ for some $i < L$. We will show that L is odd and $t_L = t_1$.

We show that $t_i \neq t_j$ whenever $1 < i < j \leq L$. First, $t_j \neq t_{j+1}$ by construction since there are no self-loops in the graph. Second, $t_j \neq t_{j+2}$ for otherwise both $e_j = (t_j, t_{j+1})$ and $e_{j+1} = (t_{j+1}, t_{j+2}) = (t_{j+1}, t_j)$ would be in the same matching. Now suppose $j - i > 2$ and $t_j = t_i$. If $j - i$ is odd then $t_i = t_j$ would have two distinct mates in the same matching, namely t_{i+1} and t_{j-1}, violating the perfect matching property. (We know t_{i+1} and t_{j-1}

are distinct because $i+1 < j-1 < L$ and L is minimal.) If $j-i$ is even and $i > 1$, then $t_i = t_j$ would have two distinct mates in the same matching, namely t_{i-1} and t_{j-1}, again violating the perfect matching property. (We know t_{i-1} and t_{j-1} are distinct because $i-1 < j-1 < L$ and L is minimal.) We conclude that $t_L = t_1$. The argument also shows that $L-1$ cannot be odd, and therefore that L must itself be odd.

The vertex sequence $t_1, \ldots, t_{L-1}$ encodes a cycle of edges $e_1, \ldots, e_{L-1}$ in the symmetric difference $M_\alpha \oplus M_\beta$, alternating between members of M_α and members of M_β. That is, the sequence encodes a k-change δ (with $k = (L-1)/2$) that, when applied to M_α, forms a perfect matching δM_α that is closer to M_β than M_α is. That is, $|\,\delta M_\alpha \oplus M_\beta\,| < |\,M_\alpha \oplus M_\beta\,|$. We can then apply strong induction to complete the proof. ∎

The proof shows how to transform any perfect matching M_α into any perfect matching M_β by applying a set of changes in sequence, where each change is encoded by an alternating cycle. Furthermore, applying any alternating cycle to a perfect matching yields another perfect matching. That is, if M is a perfect matching for a complete graph G, and δ is an even-length cycle in G alternating between edges in M and edges in not in M, then $\delta M = M \oplus \delta$ is also a perfect matching on G.

7.3 A Lin-Kernighan heuristic for weighted perfect matching

The previous section describes a process for transforming one perfect matching into another by applying changes encoded by alternating cycles. This is precisely how the concrete Lin-Kernighan strategy operates. The notation in the proof is suggestive: the t and e sequences in the proof are exactly the t and e sequences of Section 2.2, immediately yielding a Lin-Kernighan heuristic for weighted perfect matching.

There are two main differences between the algorithm used in the proof of the lemma and the Lin-Kernighan heuristic for minimum weight perfect matching. First, the heuristic does not know the destination toward which it is progressing, an LK-optimal perfect matching M_β. The heuristic has to discover the destination for itself. Second, only improving k-changes are accepted.

It is interesting to note that given *any* initial perfect matching M_0 and an *optimal* perfect matching M_{opt}, Lin-Kernighan can potentially find its way from M_0 to M_{opt}. Each step is the discovery of an improving alternating cycle δ_i. That is, there are no *struc-*

tural barriers stopping Lin-Kernighan from finding optimal perfect matchings. Contrast this situation with that for the TSP. Lin-Kernighan for the TSP cannot construct an improving double-bridge 4-change (see Section 4.9).

However, Lin-Kernighan for weighted perfect matching is not *guaranteed* to find optimal matchings. To guarantee optimal perfect matchings with Lin-Kernighan, one would have to dispense with the usual shortcuts—candidate sets and only limited backtracking. The greedy selection criterion would become less prominent because backtracking would have to occur at all depths. (The cumulative gain criterion could stay because of the fact about positive partial sums.) Dispensing with these shortcuts would make Lin-Kernighan impractical in comparison with the exact polytime algorithms. The genius of the polytime algorithms for matching is in finding the right improvements, and proving that they lead to optimality in reasonable time.

Still, the observation gives hope that the perfect matchings found by Lin-Kernighan may be rather good. Furthermore, the fast running times of Lin-Kernighan for the TSP gives hope that Lin-Kernighan for weighted perfect matching will also have fast running times.

7.4 Applying Lin-Kernighan to minimum-weight perfect matching

This section summarizes the details required to fully specify the Lin-Kernighan heuristic for minimum-weight perfect matching. We borrow as much as possible from details for the TSP setting, given in Chapter 4.

7.4.1 Feasibility, candidate sets, backtracking, and data structures

The heuristic has no problem maintaining feasibility and near-feasibility. Applying changes encoded by *any* alternating cycle to a perfect matching always results in a perfect matching. So Lin-Kernighan for weighted perfect matching need not do anything special in this regard. This simplifies the code a great deal in comparison to the TSP case.

We use the same tabu rule as in the TSP case, namely "never delete an added edge."

This prevents endless cycling in a single search.

Because we seek low-weight solutions, we can use the same candidate sets as for the TSP. In our implementation, the candidate set for a vertex is either its nearest few neighbours in the graph, or the nearest few quadrant-based neighbours, or the unions of these two.

We use limited backtracking as in the TSP case. We adopt the same high-level backtracking scheme using don't-look bits. See Algorithm 3 in Section 4.4.1, but substitute "perfect matching" for "tour". The low-level backtracking code benefits greatly from the simpler feasibility requirements, making it is easy to specify an arbitrary depth over which low-level backtracking should occur. In keeping with the TSP case, we backtrack over all possibilities for t_1 through t_6, subject to the candidate sets. That is, for each choice of t_1, all candidates for t_3 and t_5 are tried; recall that t_{2i+2} is fixed once t_{2i+1} is chosen. Compare this with the TSP case where given t_1, there are always two choices for t_2, and given t_1 through t_3 there are two choices for t_4, and finally given t_1 through t_5 there are often two choices for t_6 (see Figure 4.2). Even with the same candidate sets, backtracking in the matching heuristic involves fewer possibilities than does backtracking in the TSP heuristic.

As in the TSP case, we bound the search sequence lengths as a safety measure. This prevents the heuristic from wasting time on extremely deep searches that may not yield an improving move. Search sequences are limited to at most 50 edge exchanges beyond the backtracking depth, *i.e.*, they can extend no farther than t_{106}.

In comparison to the TSP case, the simpler feasibility requirements for perfect matching lead to simpler data structures. The only operations we need to perform on a matching are to ask a vertex for its current mate, and to mate a pair of vertices. We can use an array *mate* to record the perfect matching: if (u, v) is in the matching, then $mate[u] = v$ and $mate[v] = u$. The query and the update each take constant time using this scheme. The scheme is also space-efficient, using $O(n)$ memory cells for an n vertex graph.

We also adopt the same data structures from the TSP case for the candidate subgraph (adjacency lists), the tabu list (a simple array).

7.4.2 Initial perfect matching

The heuristic needs to be seeded with a perfect matching. For this we use a randomized greedy perfect matching. It is similar in spirit to Kruskal's algorithm for minimum spanning trees, and the greedy heuristic for the TSP.

A pure greedy perfect matching for $G = (V, E)$ is constructed as follows [17]. We begin with an empty matching $M_0 = \emptyset$. In each of $|V|/2$ stages we add one more edge to the matching to form M_i, answering $M_{|V|/2}$ as our greedy perfect matching. Given a matching M_i on G, we say edge (u, v) is *allowable* if $M_i \cup \{(u, v)\}$ is also a matching, *i.e.*, if neither u nor v have mates in M_i. We examine the edges E in non-decreasing order, always adding the next allowable edge.

A randomized greedy perfect matching is built in the same way, except that we choose randomly between the next two distinct allowable edges. With probability 2/3 we add the lighter of the two edges, and otherwise choose to add the heavier of the two. The allowable edge not added to the matching goes back into the pool of available edges.

We can use the same data structures for these greedy matching algorithms as we used for the greedy tour algorithms. In particular, we use a priority queue to hold, for each vertex v, a small number of the shortest remaining edges incident upon v. In the geometric case, we can apply k-d trees for semi-dynamic point sets to feed the priority queue. The "semi-dynamic" aspect of k-d trees allows vertices to be hidden from future nearest neighbour queries. Within the greedy matching algorithm, a vertex becomes hidden as soon as it is matched.

We expect greedy perfect matchings to be good starting points for Lin-Kernighan for minimum weight perfect matching for the same reasons greedy tours are good for Lin-Kernighan for the TSP. First, many of the edges in a greedy perfect matching will be between mutual nearest neighbours. Those portions of the graph should rapidly quiesce, *i.e.*, those vertices will quickly be removed from the *Active* queue. Second, there are a few large defects in the matching, namely the last few edges added during the greedy algorithm. Those long edges will provide a large cumulative gain when removed from the matching, driving the heuristic to make large gains through a few deep searches.

7.5 Iterated Lin-Kernighan for weighted perfect matching

Iterated Lin-Kernighan can certainly be applied in the perfect matching case. In the TSP case we use a random double-bridge 4-change to try to kick the heuristic out of a local optimum. Since a double-bridge 4-change cannot be encoded as an alternating cycle, the heuristic is not easily led back to the same local optimum. However, Lemma 7.1 tells us that a similarly bad construction does not exist for perfect matchings. Any legal 4-change on a perfect matching is undone either by a single alternating cycle with eight edges total, or by two alternating cycles with four edges each. Indeed any k-change is undone by a collection of disjoint alternating cycles.

Between iterations, our implementation of Iterated Lin-Kernighan for perfect matching applies a random 4-change "kick" in the form of a single alternating cycle with eight edges. We hypothesize that the heuristic has a more difficult time undoing a single eight-edge cycle than it does undoing two four-edge cycles. We therefore hope to achieve a greater kick out of local minima even while using a kick with such small cardinality. Rohe uses kicks in the form of k-changes with much larger k, *e.g.*, k=10 and k=50 [76, 77]. We use k=4 so we may more easily compare the iterated matching heuristic with the iterated TSP heuristic.

As with the TSP case, the choice of perturbations is unbiased; only the eight vertices involved in the 4-change are loaded into the *Active* queue; and finally, we never allow a heavier perfect matching to become the incumbent.

7.6 Experimental methodology

It may be interesting to measure Lin-Kernighan for weighted perfect matching with an eye toward comparing it with other heuristics and exact algorithms for the problem. However, our purpose is to measure the effectiveness of cluster compensation. Our experiments are structured around this goal, and are therefore very similar to those we perform in the TSP setting. There are only two differences from the TSP case: some of the data is modified, and we measure the quality of the answers differently.

7.6.1 TSP instances can be weighted perfect matching instances

A complete graph has a perfect matching if and only if it has an even number of vertices. Every TSP instance with an even number of vertices can therefore be used as an instance of the weighted perfect matching problem. If a TSP instance has an odd number of vertices, then we can remove one vertex to make it a legal instance for the weighted perfect matching problem. For geometric instances, we follow Applegate and Cook and remove the most "northeast" point [8]. More precisely, we remove the vertex with the greatest x coordinate; break ties by removing the vertex which also has the greatest y coordinate. We do not adapt non-geometric TSP instances having an odd number of vertices for use as weighted perfect matching instances.

7.6.2 Measuring matching quality

We measure the weight of the matchings produced by the Lin-Kernighan heuristic against the optimal values produced by an exact algorithm for the problem. We use the highly scalable Blossom IV implementation due to Cook and Rohe [27] to compute optimal matchings.

7.7 Results

The following sections describe the results of the experiments for weighted perfect matchings. The results for instances generated from other instances are postponed until Chapter 8 where the generating algorithms themselves are discussed.

7.7.1 Breakdown of run times

Table 7.1 shows how running times break down for typical runs on two instances, geometric instance `pcb3038` and random distance matrix `dsjr.55.3162`. Table 7.1 is analogous to Table 6.1 describing the time for the phases involved in Lin-Kernighan for the TSP.

The preprocessing steps are the same for both the TSP and perfect matching, except that the TSP heuristic constructs a greedy tour, while the matching heuristic constructs a greedy perfect matching. The preprocessing times are therefore similar to the TSP case.

The optimization phase is much shorter for perfect matching than it is for the TSP. This has an effect on the conditions under which cluster compensation makes the heuristic run faster. There is less work to optimize away.

For comparison, Table 7.1 also lists the time used by the Blossom IV code to find optimal perfect matchings on those instances. The times vary a great deal with the kind of instance being optimized, much more than for the Lin-Kernighan heuristic. Clearly, finding an optimal perfect matching for `pcb3038` in less than one second is preferable to finding a suboptimal one in 5 seconds. But we shall see that Lin-Kernighan for weighted perfect matching is robust in the face of different inputs, and scales very well. In our

Phase	Time (sec) `pcb3038`	Time (sec) `dsjr.55.3162`
Build a 2-d tree	0.01	n/a
Build a minimum spanning tree	0.19	2.55
Process the minimum spanning tree in preparation for later cluster distance queries	0.04	0.01
Build a candidate subgraph	0.84	8.23
Construct a randomized greedy perfect matching	0.10	8.44
Lin-Kernighan optimization phase (first iteration)	0.14	1.30
$n-1$ subsequent iterations of Iterated Lin-Kernighan	4.03	9.18
Time for Blossom IV to find an optimal perfect matching	0.97	199.62

Table 7.1: Breakdown of running time into phases for Iterated Lin-Kernighan with cluster compensation. The times are taken for a 3038 iteration run on geometric instance `pcb3038`, and a 3162 iteration run on random distance matrix instance `dsjr.55.3162`. Also shown is the time taken for the Blossom IV code to find optimal perfect matchings.

Percentage excess

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
lin318	0.56	1.38	0.82	0.38	1.22	0.84	0.06	0.75	0.69
pcb442	0.98	1.48	0.50	0.87	1.48	0.61	0.71	1.35	0.64
att532	0.71	0.71	0.00	0.68	0.68	0.00	0.58	0.58	0.00
gr666	0.66	0.91	0.25	0.52	0.81	0.29	0.33	0.53	0.20
dsj1000	0.75	0.61	-0.14	0.63	0.53	-0.10	0.53	0.30	-0.23
pr1002	1.35	1.53	0.18	1.31	1.43	0.12	1.26	1.00	-0.26
pcb1173	1.21	1.38	0.17	1.15	1.35	0.20	1.08	1.27	0.19
pr2392	1.41	1.53	0.12	1.36	1.53	0.17	1.34	1.49	0.15
pcb3038	1.00	1.28	0.28	0.98	1.22	0.24	0.93	1.18	0.25
fl3795	3.38	4.44	1.06	2.74	3.10	0.36	0.74	1.29	0.55
fnl4461	0.79	0.96	0.17	0.77	0.92	0.15	0.75	0.90	0.15
pla7397	1.51	2.04	0.53	1.41	2.00	0.59	1.32	1.81	0.49

Table 7.2: Quality of output produced for TSPLIB instances by the Lin-Kernighan heuristic for weighted perfect matching. Quality is measured as the percent over optimal. "LK" is the base Lin-Kernighan heuristic, and "LKcc" is the Lin-Kernighan heuristic using efficient cluster compensation.

experience the time and space required for Lin-Kernighan for matching increases more slowly than the time and space required for the Blossom IV code.

7.7.2 TSPLIB instances

Table 7.2 compares the quality of the tours generated on TSPLIB instances by the two versions of the Lin-Kernighan heuristic for weighted perfect matching. Ten experiments were performed for each instance. Each experiment on an n vertex instance is an n iteration Iterated Lin-Kernighan run. For comparison purposes, the differences in the average percentage excess is also shown. Table 7.3 shows the average running times for the same experiments. For comparison purposes, the ratio of the average running times are listed.

As with the results for the TSP, the Lin-Kernighan heuristic in general produces very

Running time

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
lin318	0.23	0.14	1.64	0.37	0.18	2.06	1.44	0.53	2.72
pcb442	0.34	0.23	1.48	0.50	0.28	1.79	1.65	0.78	2.12
att532	0.28	0.31	0.90	0.42	0.44	0.95	1.68	1.70	0.99
gr666	5.71	5.00	1.14	7.00	5.26	1.33	17.54	7.74	2.27
dsj1000	2.58	0.66	3.91	4.76	0.86	5.53	23.37	3.08	7.59
pr1002	0.93	0.62	1.50	1.37	0.78	1.76	5.12	2.24	2.29
pcb1173	0.86	0.68	1.26	1.15	0.85	1.35	3.72	2.44	1.52
pr2392	1.95	1.53	1.27	2.59	1.87	1.39	8.41	5.07	1.66
pcb3038	2.52	2.01	1.25	3.37	2.48	1.36	10.98	6.62	1.66
fl3795	19.04	8.30	2.29	34.19	14.65	2.33	144.77	47.97	3.02
fnl4461	3.05	2.96	1.03	3.89	3.58	1.09	11.20	9.10	1.23
pla7397	12.71	9.84	1.29	19.24	12.45	1.55	76.19	36.26	2.10

Table 7.3: Time taken on TSPLIB instances by the Lin-Kernighan heuristic for weighted perfect matching. Time is measured as user CPU seconds.

good solutions. The weighted matchings it finds are usually within 2% of optimal. This is the case after only the first iteration for all instances except the most sharply clustered instance, `fl3795`. By iteration 3795, the relative quality of the matchings found for it are as good as for the other instances.

However, cluster compensation usually forces Lin-Kernighan to produce matchings of poorer quality, its percentage excess over optimal being usually between 0.15% and 0.60% greater than base Lin-Kernighan. This is true over all iteration ranges. There are outliers on both the worse and better side. In some cases, cluster compensation helps Lin-Kernighan find matchings that are roughly 0.25% better than found by base Lin-Kernighan. In general, the degradation in the output caused by cluster compensation on these instances is more accentuated than is the case for the TSP.

As for running times, Lin-Kernighan for weighted perfect matching runs far more quickly than Lin-Kernighan for the TSP, between roughly 20 and 200 times faster, depending on the variation and the instance used for comparison.

Cluster compensation usually reduces running times, with the speedups being consistent with the results for the TSP. Only on instance `att532` does cluster compensation increase running times. That instance uses the special `ATT` metric, and our implementation treats it as a general distance matrix. The increased running time is likely caused by the use of a less efficient MST algorithm than is available for Euclidean instances. By n iterations the difference is only 0.02 seconds, near the edge of the accuracy Linux provides on the Intel Pentium architecture, and representing a relative difference of less than 1%.

7.7.3 Uniform geometric instances

Table 7.4 shows the percent excess over optimal of the matchings generated by the two versions of the Lin-Kernighan heuristic on uniform geometric instances. Three instance sizes were generated, with 316, 1000, and 3162 vertices. Five instances were generated for each size, and ten experiments were performed on each instance.

Both versions of the heuristic find very good answers, less than 1.4% above optimal even after only the first iteration. On the larger size inputs, cluster compensation forces the heuristic to find answers about 0.18% greater excess than the base version of the heuristic.

Table 7.5 shows the average running times for the same experiments. For comparison

Percentage excess

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
uni.316	0.70	0.68	-0.02	0.66	0.59	-0.07	0.55	0.52	-0.03
uni.1000	1.13	1.32	0.19	1.10	1.28	0.18	1.02	1.20	0.18
uni.3162	1.05	1.24	0.19	1.03	1.21	0.18	1.01	1.18	0.16

Table 7.4: Quality of output produced for the class of uniform geometric instances by the Lin-Kernighan heuristic for the TSP. Quality is measured as the percent excess over optimal.

Running time

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
uni.316	0.16	0.13	1.26	0.23	0.17	1.35	0.81	0.52	1.57
uni.1000	0.62	0.51	1.21	0.86	0.65	1.32	2.95	1.90	1.55
uni.3162	2.18	1.87	1.16	2.91	2.30	1.26	9.48	6.18	1.53

Table 7.5: Time taken on the uniform geometric instances by the Lin-Kernighan heuristic for the TSP. Time is measured as user CPU seconds.

purposes, the ratio of the average running times are listed.

Running times were very low for both versions of the heuristic. Still, cluster compensation's advantage grows from roughly 20% overall speed improvement to over 50%. These speedups are consistent across all the instance sizes. Again, times for weighted perfect matching are much faster than for the TSP (Cf. Table 6.5).

7.7.4 Uniformly generated distance matrices

Table 7.6 compares the quality of the matchings generated by the two versions of the Lin-Kernighan heuristic on instances generated by drawing distances randomly from a uniform distribution. Three instance sizes were generated, with 316, 1000, and 3162 vertices. Five instances were generated for each size, and ten experiments were performed

Percentage excess

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
dsjr.316	2.03	1.53	-0.50	1.57	1.30	-0.27	0.61	0.51	-0.10
dsjr.1000	3.67	3.01	-0.65	3.08	2.59	-0.49	2.06	1.69	-0.38
dsjr.3162	5.14	4.45	-0.69	4.61	3.98	-0.63	3.74	3.02	-0.72

Table 7.6: Quality of output produced for the class of uniformly generated distance matrices by the Lin-Kernighan heuristic for the TSP. Quality is measured as the percent excess over optimal.

Running time

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
dsjr.316	0.30	0.31	0.96	0.39	0.39	1.01	1.10	1.00	1.10
dsjr.1000	2.18	2.34	0.93	2.53	2.60	0.97	5.14	4.71	1.09
dsjr.3162	18.36	20.65	0.89	19.49	21.58	0.90	27.94	28.70	0.97

Table 7.7: Time taken on the uniform distance matrix class of instances by the Lin-Kernighan heuristic for the TSP. Time is measured as user CPU seconds.

on each instance.

Table 7.7 shows the average running times for the same experiments. For comparison purposes, the ratio of the average running times are listed.

As with the TSP, both variations of Lin-Kernighan find worse answers as instance sizes grow. The effect is more marked than in the TSP case. At these sizes, however, the excess above optimal is still at most roughly 5%. Cluster compensation consistently helps Lin-Kernighan find better tours, roughly 0.5% closer to optimal, with the advantage growing as instance sizes grow.

Cluster compensation increases the running time of the heuristic in most cases. Two related factors explain the slowdown. First, as in the TSP case LKcc must pay the extra costs for building and processing a minimum spanning tree. On a general matrix, the time to find a MST grows quadratically in the number of vertices. Second, the optimization phase on perfect matchings runs very quickly. Preprocessing time therefore becomes significant, moreso than it was in the TSP case. The preprocessing times for weighted matching are similar to those for the TSP. On a 3162 vertex instance, LK takes roughly 17 seconds to build 20 nearest neighbour candidate subgraph and to find a greedy matching; LKcc does the same as LK and also takes an extra 2.5 seconds to build a minimum spanning tree and process it for later cluster distance queries. Yet LK takes less than 2 seconds to complete the first iteration of the optimization phase. There is simply no manœuvreing room: the first iteration of LK ends before the first iteration of LKcc begins! However, cluster compensation makes iterations faster on average. LKcc eventually pays back its preprocessing cost. The iterations required to completely amortize the extra costs increases with the size of the instance: $n/10$ iterations suffices for $n = 316$, n iterations suffices for $n = 1000$, and we expect that just over n iterations suffices for $n = 3162$.

Even without the aid of specialized data structures, the absolute running times are quite reasonable, being comparable to many of the TSPLIB instances tested.

Again, we have used instance sizes increasing in half-powers of 10: $316 \approx 10^{2.5}$, $1000 = 10^3$, and $3162 \approx 10^{3.5}$. If the running time for the Lin-Kernighan heuristic increases faster than quadratically, then we would expect the running times to increase by a factor of ten or more from one instance size to the next. This is roughly the case for single-iteration Lin-Kernighan, but only because preprocessing times increase quadratically for both variations of the heuristic.

7.7.5 Bentley's distributions

Table 7.8 compares the quality of the matchings generated by the two versions of the Lin-Kernighan heuristic on Bentley's classes of instances. Five instances were generated from each class, and ten experiments were performed on each beginning from randomized greedy matchings. Because the running times were so fast, we had the luxury and the patience to include class `cubeedge` in our experiments. As expected, Lin-Kernighan behaves much the same on class `cubeedge` as it does on class `cubediam`. This supports our decision to skip class `cubeedge` in the TSP experiments.

Table 7.9 shows the average running times for the experiments. For comparison purposes, the ratio of the average running times are listed.

Percentage excess

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
uni.1000	1.13	1.32	0.19	1.10	1.28	0.18	1.02	1.20	0.18
annulus.1000	0.99	0.90	-0.09	0.99	0.90	-0.09	0.99	0.90	-0.09
arith.1000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ball.1000	0.92	1.08	0.16	0.89	1.06	0.17	0.86	1.02	0.17
clusnorm.1000	1.06	1.10	0.04	0.88	0.95	0.06	0.64	0.62	-0.01
cubediam.1000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
cubeedge.1000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
corners.1000	0.77	0.94	0.16	0.75	0.92	0.17	0.71	0.84	0.14
grid.1000	0.83	0.80	-0.03	0.82	0.78	-0.03	0.74	0.70	-0.04
normal.1000	1.07	0.99	-0.08	1.04	0.96	-0.08	0.95	0.92	-0.03
spokes.1000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 7.8: Quality of output produced for Bentley's classes of instances by the Lin-Kernighan heuristic for the TSP. Quality is measured as the percent excess over optimal.

Running time

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
uni.1000	0.62	0.51	1.21	0.86	0.65	1.32	2.95	1.90	1.55
annulus.1000	1.02	0.60	1.69	1.56	0.78	2.00	6.30	2.43	2.59
arith.1000	1.40	0.53	2.64	2.21	0.59	3.74	9.60	1.19	8.06
ball.1000	0.58	0.49	1.19	0.81	0.63	1.30	2.81	1.81	1.55
clusnorm.1000	1.30	0.74	1.76	2.21	1.10	2.02	7.70	2.92	2.64
cubediam.1000	1.25	0.76	1.64	1.84	0.96	1.91	7.18	2.74	2.62
cubeedge.1000	1.23	0.79	1.56	1.77	0.98	1.82	6.71	2.66	2.52
corners.1000	0.60	0.53	1.13	0.82	0.66	1.24	2.87	1.89	1.52
grid.1000	0.51	0.65	0.79	0.65	0.81	0.81	1.86	2.17	0.86
normal.1000	0.81	0.54	1.50	1.16	0.68	1.69	4.26	1.96	2.18
spokes.1000	1.23	0.85	1.44	1.80	1.03	1.74	7.08	2.67	2.65

Table 7.9: Time taken on Bentley's classes of instances by the Lin-Kernighan heuristic for the TSP. Time is measured as user CPU seconds.

The matchings produced by Lin-Kernighan are very good, with excesses over optimal of less than 1%. Across the different distributions, cluster compensation does not significantly impair the quality of the matchings produced by Lin-Kernighan.

An interesting pattern emerges if we look individually at the distributions `uni`, `corners`, and `ball`. These are the three classes where cluster compensation makes Lin-Kernighan produce worse matchings than the base version of the heuristic. The effect is small, a degradation of less than 0.2%. However, in all three cases the matchings produced by LKcc after 1000 iterations are worse than those produced by base LK after only the first iteration. We are at a loss to explain why these three classes force cluster compensation to produce worse matchings: classes `uni` and `ball` are not sharply clustered, while class `corners` is the most sharply clustered. However, this quality of output comparison does point out how little of a "kick" the 4-change perturbation gives between iterations. In particular, the heuristic has a hard time improving a poor matching with extra iterations. This experience affirms Rohe's choice of using k-changes with k much larger than 4.

Running times are much faster for matching than for the TSP. Cluster compensation makes Lin-Kernighan have very uniform running times indeed on these classes: 1000 iteration runs all averaged less than three seconds of CPU time. This is consistent with the times for TSPLIB instances of similar size. The run times for the base heuristic vary more, from just under 2 seconds for 1000 iterations to just under 10 seconds for 1000 iterations.

Classes `arith`, `cubediam`, `cubeedge`, and `spokes` are peculiar for weighted perfect matching as they were for the TSP. Both variations of Lin-Kernighan found an optimal matching on the first iteration. This is the same behaviour as we surmise for the TSP runs on these instances. (Section 6.6.1 explained the high excess over Held-Karp on this class of instances on a poor lower bound.) A pleasant surprise is that matching run times remain reasonable for these classes. The run times for base Lin-Kernighan are usually in the same ballpark as for the other Bentley classes; with the aid of cluster compensation, the run times for these classes are essentially indistinguishable from the run times for the others. Contrast these results with the TSP results, where these line-oriented classes were very tough for Lin-Kernighan.

In only one case does cluster compensation slow down Lin-Kernighan. At 1000 iterations on 1000 city instances from the `grid` class, cluster compensation slows down Lin-Kernighan by 14%. The preprocessing overhead for cluster compensation does not fully account for this slowdown. For 1000 city instances, extra preprocessing overhead for cluster compensation is roughly 0.1 seconds. Online computation of cluster distances during the optimization phase proper likely accounts for the other lost 0.2 seconds. Still, these extra overhead times are miniscule, at least at this instance size.

7.7.6 Probe depths and move depths of sample cases

Probe and move depths helped us understand the speedups attained in the case of the TSP. This section discusses the probe and move depths for typical runs of the matching heuristic on the same three instances.

Figures 7.1, 7.2, and 7.3 are histograms of probe and move depths for typical n iteration runs for instances `grid.828.1000`, `uni.820.1000`, and `dsj1000`, respectively.

Our first observation is about the success of the heuristic in finding improvements. The move depth graphs show the number and depth of improving exchanges actually found by both heuristics. For each instance, the move depth graphs for both variants of the heuristic track each other very closely, and in many places are perfectly superimposed. This shows that cluster compensation does not significantly impair the ability of Lin-Kernighan to find improving move sequences at any depth.

The probe depths reflect the amount of search work performed by the heuristic. Cluster compensation slows down Lin-Kernighan for weighted perfect matching by an average of 14% after n iterations on `grid` instances. In contrast, cluster compensation produces a speedup of 1.55 for `uni` instances, and a speedup of nearly 8 for `dsj1000`. These running time results should be reflected in the probe depths.

The probe depths for both variants of the heuristic on instance `grid.828.1000` are almost identical. Also, relatively few probes go deeper than $t[15]$. So cluster compensation does not reduce search work on this instance. In fact it only slows things down because it requires that least common ancestors be computed in the inner loop of the heuristic, and it also requires extra time for preprocessing. This agrees with the running time results, a slowdown of 14% for this instance class.

The probe depths for both variants on `uni.820.1000` are also quite close to each other. This time, cluster compensation slightly reduces the number of searches ending between 8 and 40 cities deep. This is apparently enough of a work savings to reduce running times by a third over the base Lin-Kernighan. We must admit being surprised at how such a small difference in probe depths can translate into that large of a speedup.

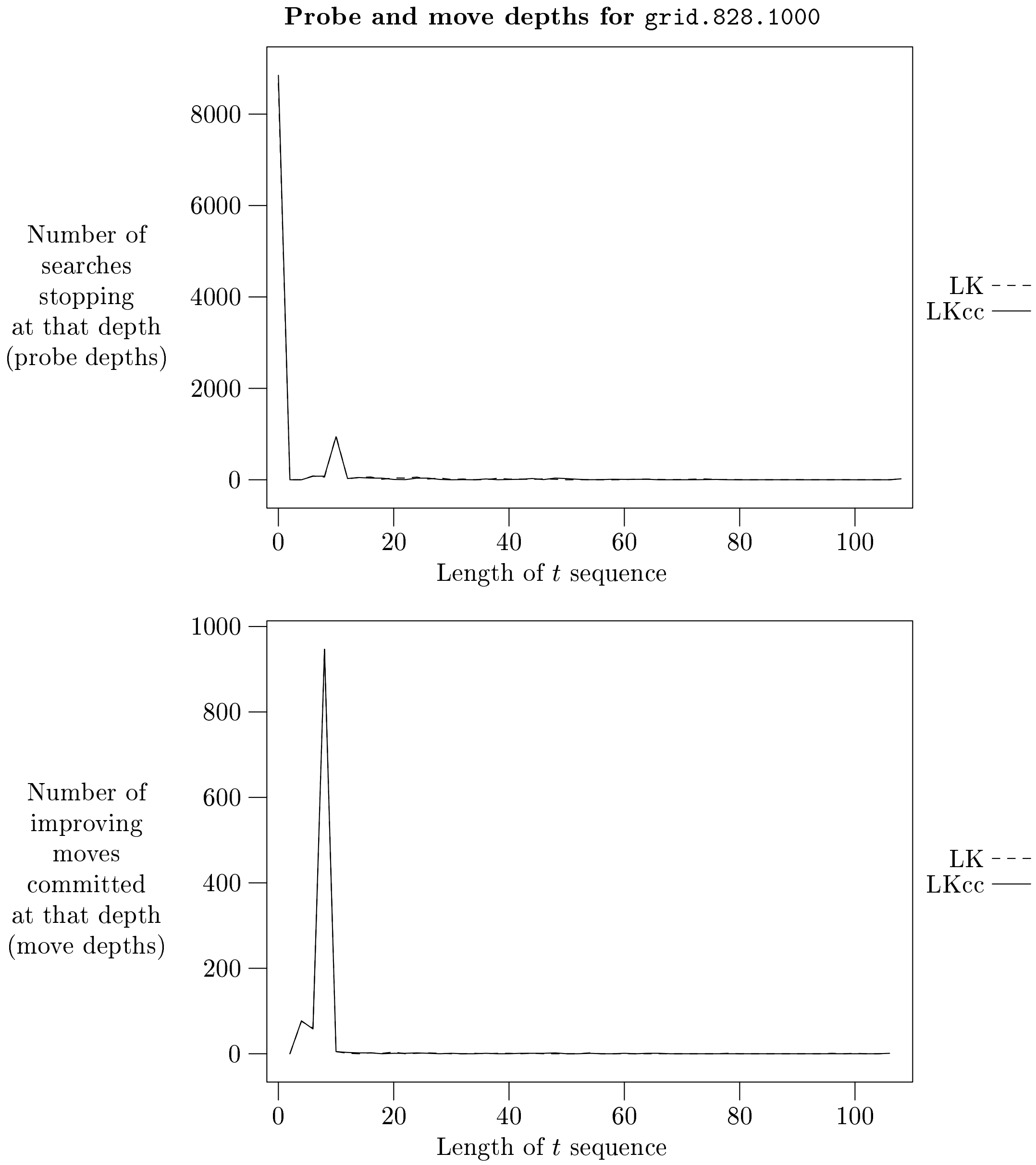


Figure 7.1: Probe and move depth profiles for a n iteration run of Lin-Kernighan for weighted perfect matching on instance `grid.828.1000`.

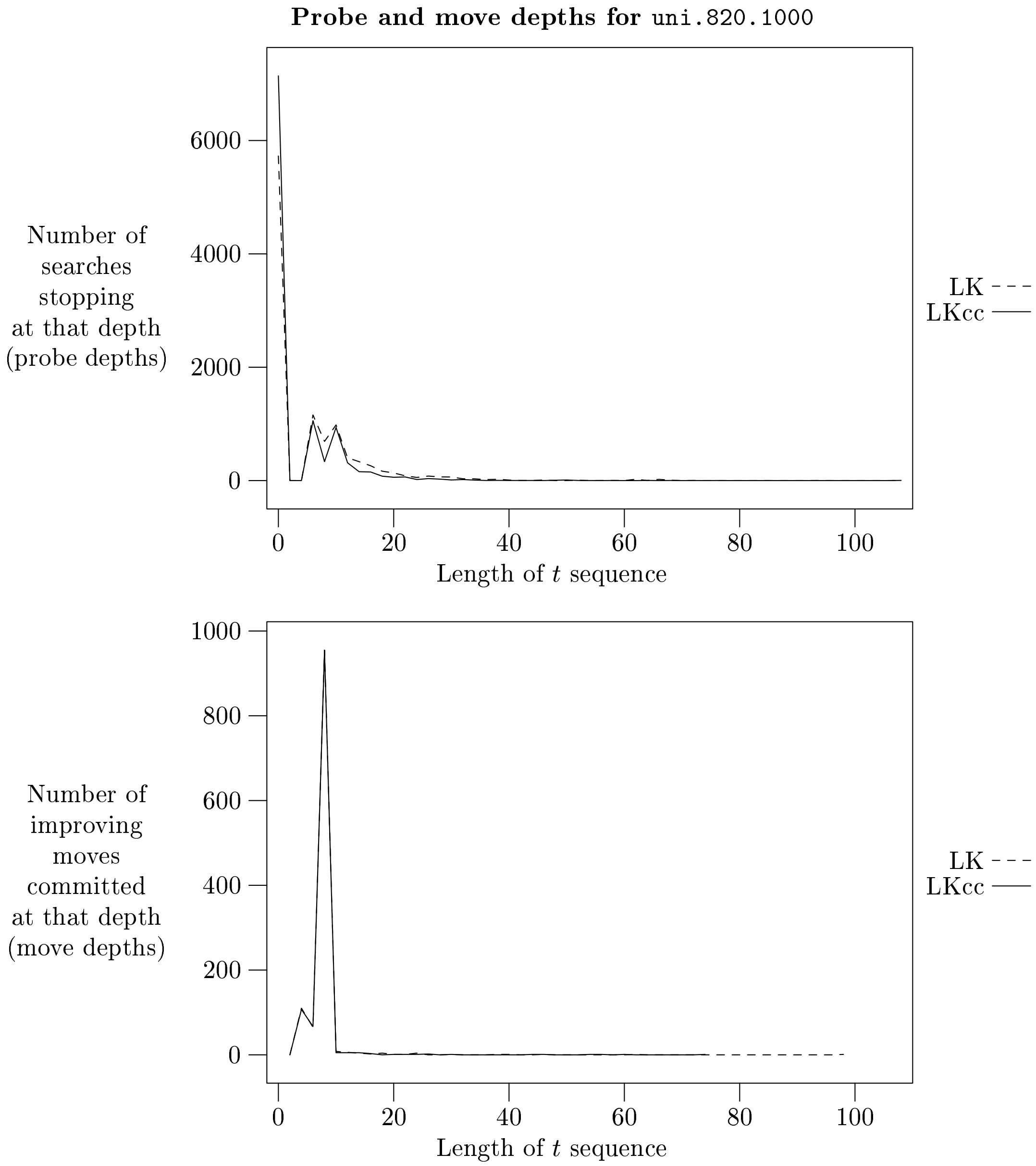


Figure 7.2: Probe and move depth profiles for a n iteration run of Lin-Kernighan for weighted perfect matching on instance `uni.820.1000`.

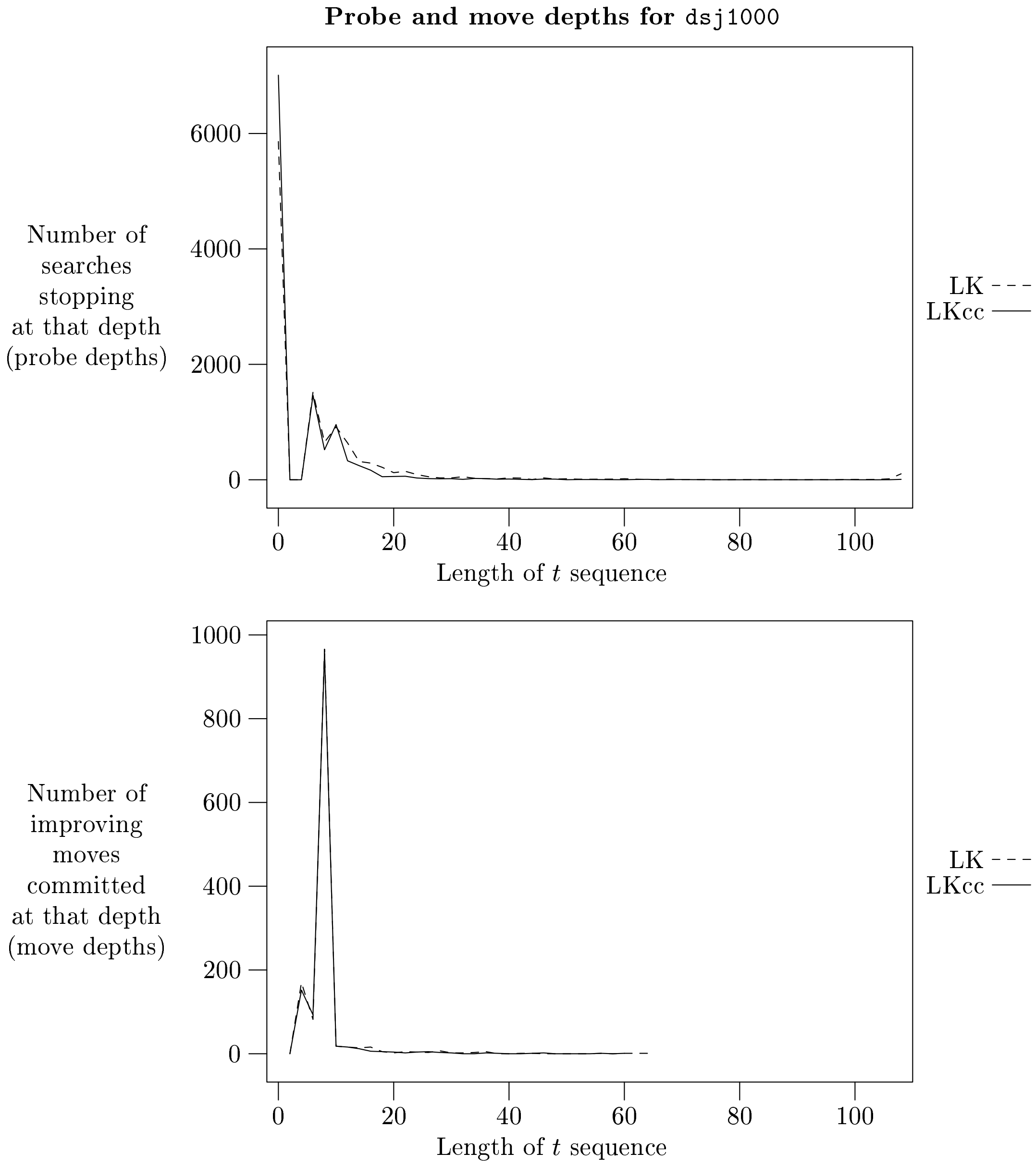


Figure 7.3: Probe and move depth profiles for a n iteration run of Lin-Kernighan for weighted perfect matching on instance `dsj1000`.

Still, the results are consistent between the running time speedup and the probe depths.

At first blush the probe depth trend for instance `dsj1000` appears the same as for `uni.820.1000`, with cluster compensation reducing the number of probes ending in the middle depths, between 10 and 40 cities deep. But a closer look reveals that base Lin-Kernighan has a small peak at depths of over 100 cities, while Lin-Kernighan with cluster compensation does not. Examining the raw data for this plot shows that base Lin-Kernighan performs 137 searches ending 100 cities or deeper, while Lin-Kernighan with cluster compensation performs only 8 searches that deep. Recall that the work involved for a search of depth l grows at least linearly with l. (If the simple linear time tabu check is used in adding each new city to the t list, it grows at least quadratically with l.) So the small peak at the large depths are more significant than they might first appear. The 137 to 8 margin at those depths is certainly enough to explain the speedup of almost 8 times provided by cluster compensation, especially given that there is very little other search work being performed.

Overall, the move depth graphs show that cluster compensation does not reduce the improvement-finding power of the Lin-Kernighan heuristic for weighted perfect matching. The probe depth graphs show that even slightly less work translates into lower running times. For instance `grid.828.1000` cluster compensation does not reduce overall work and it therefore increases the running time for the heuristic.

7.8 Summary

This chapter motivates the use of a Lin-Kernighan style heuristic for minimum weight perfect matching. It describes the details of such a heuristic, noting that it can share much of the same implementation as a Lin-Kernighan heuristic for the TSP. We have given the results of experiments with the heuristic on the same data as the experiments for the TSP.

If an application absolutely requires optimal matchings, then the Blossom IV code will find them, given enough time and space. However, if getting within 2% of optimal is good enough, then a single iteration run of Lin-Kernighan will usually find it, and in very short order.

When preprocessing costs are small, such as for geometric instances or on small instances, cluster compensation usually speeds up Lin-Kernighan by between 25% and 300%. However, the quality of the output degrades slightly, increasing the excess above optimal by about 0.2% to 0.5%. Except for the very difficult instances, the extra time saved by cluster compensation can be spent on more iterations, but the matchings it produces with new iterations improves only very slowly when using 4-changes as kicks. For better quality answers and more progress over time, we suggest following Rohe [76] and using k-changes with k much larger than 4. For more difficult instances such as `fl3795`, the rate of improvement for both variations of Lin-Kernighan is much better, with more iterations producing much better matchings. In that case, cluster compensation allows 3795 iterations in only 50% more time than base Lin-Kernighan uses for 380 iterations, and finds matchings only 1.3% above optimal compared to 2.75% for base Lin-Kernighan.

For large non-geometric instances over the range of iterations studied here, cluster compensation may not produce faster Lin-Kernighan runs. The preprocessing overhead can dominate the time saved during the optimization phase. The main component in the preprocessing phase is finding a minimum spanning tree. Every edge must be examined at least once, and already that $O(n^2)$ time dominates the worst-case asymptotic time for the rest of the cluster compensation preprocessing phase and the observed subquadratic behaviour of the optimization phase. However, there are three bright spots. First, Lin-Kernighan with cluster compensation tends to catch up to base Lin-Kernighan as more iterations are used. Second, an application may have special knowledge of its instances, allowing minimum spanning trees to be computed much more quickly than by the general methods used here (Prim's algorithm). In that case, cluster compensation would not be dragged down so badly by the extra preprocessing time it requires. Third, cluster compensation helps Lin-Kernighan find better matchings. For the `dsjr` classes of inputs, that advantage is a reduction in the excess above optimal of between 0.4% and 0.7%.

The rate of improvement in the quality of the output with extra iterations is slower for weighted perfect matching than for the TSP. This may have two causes. First, the first iteration of Lin-Kernighan usually finds much better matchings than Lin-Kernighan for the TSP finds tours. Second, the double-bridge mutation gives Iterated Lin-Kernighan for the TSP a greater "kick" out of local minima than the 4-change does for Iterated Lin-Kernighan for weighted perfect matching. Lemma 7.1 suggests that only very elaborate perturbations can give a much greater kick than the 4-change used here.

We come to similar conclusions for weighted perfect matching as for the TSP. Cluster compensation usually speeds up Lin-Kernighan, with only a slight loss in the quality of the output. Cluster compensation provides a good tradeoff for a wide variety of instance classes. When non-geometric instances are used, more iterations are required before cluster compensation saves overall running time. For geometric data, only very rigidly uniform and therefore unclustered instances such as those drawn from the `grid` class make Lin-Kernighan with cluster compensation take more time than base Lin-Kernighan.

Chapter 8

Instance generators

This chapter describes algorithms for generating new instances from old. There are a few reasons one might want to do this.

First, we want to assure ourselves that the experimental results reported thus far are not flukes, dependent upon some peculiar characteristic of the test data used. That is, we want to know if the results presented so far are robust.

Second, practitioners deciding whether to apply efficient cluster compensation for their own problem take several factors into account, including applicability of the technique and ease of coding. But perhaps the most important factor is whether the performance win applies to their own problem domain. Testing only on a small number of standard instances and on uniform instances of one kind or another does not help in this respect. Since we cannot report experiments on all possible instance classes, we describe tools whereby the reader can generate many test instances based on their own data. Some of the algorithms described here for generating test data are designed to preserve the main characteristics of the input instance to which Lin-Kernighan reacts. That is, the Lin-Kernighan heuristic, both with and without cluster compensation, should behave similarly on the generated data as it does on the original data. The practitioner can therefore build their own domain-specific test bed of instances. This test bed can be used to decide whether to apply Lin-Kernighan in their own ongoing work with similarly structured inputs.

Third, the tools in this chapter let us test our intuition about why Lin-Kernighan performs less well on clustered inputs and why cluster compensation helps in those cases. The algorithms described here divide naturally into two categories. The first category includes algorithms designed to preserve the essential cluster structure of the input instances. If our intuition is correct, then Lin-Kernighan should behave similarly on output instances as it does on the input instance to these kinds of algorithms. The second category includes algorithms designed to destroy cluster structure by moving vertices into the gaps between widely separated clusters. If our intuition is correct, the base Lin-Kernighan heuristic should perform better on the derived instance than on the original instance, and cluster compensation should give less of an advantage over baseline on the derived instance than it does on the original instance.

The following sections describe the distill and generate paradigm for data generation, and several algorithms within this paradigm. The next chapter reports on experiments with the Lin-Kernighan heuristic on selected instances generated in this way.

8.1 Distill and generate paradigm

We generate new instances from old seed instances in two steps. We first distill the important features of the seed instance into a few key values, throwing much information away. The characterization of the seed instance is used to define an implicit random distribution over instances. We can then generate as many new instances as we like from the distribution.

Hutton [42] shows the effectiveness of the characterize and generate paradigm for creating benchmark data in the context of optimization problems for very large scale integrated circuit design. Hutton calls the generated instances *clones* because for the most part the optimization algorithms were unable to distinguish between the original seed instance and the generated instances.

Our goals here are narrower. Our context is a particular optimization *algorithm* (the Lin-Kernighan heuristic), not an optimization *problem*. The characterization and generation algorithms are tailored to the behaviour of the heuristic and how it responds to the quality and severity of clustering in the input. They are also designed to highlight or mute any performance differences efficient cluster compensation provides to the Lin-Kernighan heuristic. This narrower focus, together with the results of Section 3.3.1 motivates the use of a minimum spanning tree of the instance as the primary characterization of the seed instance in many of the algorithms described below.

8.2 Preserving cluster structure

The first category of algorithms tries to preserve the essential cluster structure of the seed instance. The algorithms have different restrictions on the form of the input: some may be used only on geometric inputs, while others may be applied to any instance.

8.2.1 Jitter

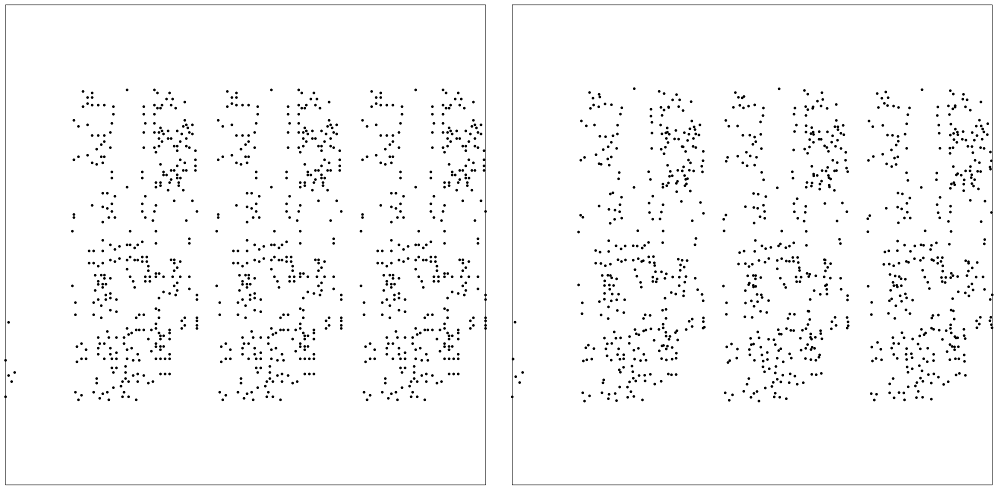
The first generation algorithm, jitter, is an obvious one: it moves each vertex of the instance a random distance in a random direction. It requires the seed instance to be geometric in nature, and produces geometric outputs. If the range of distances is small, then the generated instances will have almost exactly the same structure as the seed instance. This algorithm can therefore be quite conservative. It is also conservative in the sense that it preserves the triangle inequality.

This generation algorithm models errors in the measurement of coordinates of some physical quantity. It can therefore be used to observe the sensitivity of an optimization heuristic to small perturbations of the input.

The jitter algorithm moves each node u in a random direction by a distance r where r is drawn from a normal distribution with mean 0 and a standard deviation of $d \cdot l(u)$. Parameter d is a scaling factor that may be set arbitrarily: perturbations grow with d. Length $l(u)$ is a *benchmark length* for node u, providing a standard "yardstick" length in the context either of the local area of the graph near u, or globally. There are two factors that go into $l(u)$, context scope and reducing function. Let T be a minimum spanning tree for the seed instance. The global context for node u is all the edges in T, and the local context for u is all the edges in T incident upon u. The reducing function is used to derive the benchmark length $l(u)$ from the set of edges in the chosen context for u. Reasonable reducing functions include the minimum edge length, the maximum edge length, or the mean edge length.

In our experiments, scaling factor d is $1/8$ and the benchmark length for any node is always the average length of an edge in the minimum spanning tree T, *i.e.*, $l = \omega(T)/(n-1)$. (That is we always use the global context and the "arithmetic mean" reducing function.) Figure 8.1 shows TSPLIB instance `pr1002` on the left and, on the right, an instance on the generated by algorithm jitter with these parameters.

One can imagine ways to extend jittering to non-geometric inputs. For example,



(a) (b)

Figure 8.1: (a) Seed instance `pr1002`, and (b) an instance generated by jitter from `pr1002`.

we could add a random number to each edge length. Note that such a transformation would not necessarily preserve the triangle equality. We leave investigations into jitter algorithms for non-geometric instances to other researchers.

8.2.2 MST shake

Algorithm jitter is often fairly conservative. We can make more radical but still cluster-appropriate change in an instance by transformations whose nature are more closely tied to its cluster structure. The minimum spanning tree of an instance encodes its cluster structure, and we can even view it as the skeleton of the instance. Taking the biological analogy further, we can generate new instances by articulating that structure, *i.e.*, bending the skeleton at key points. We call this tree shaking, after the mental image of a tree waving in the wind. More specifically, we call the algorithm mst-shake because it is a minimum spanning tree that we manipulate. As with jitter, algorithm mst-shake takes a geometric instance as input and produces a geometric output.

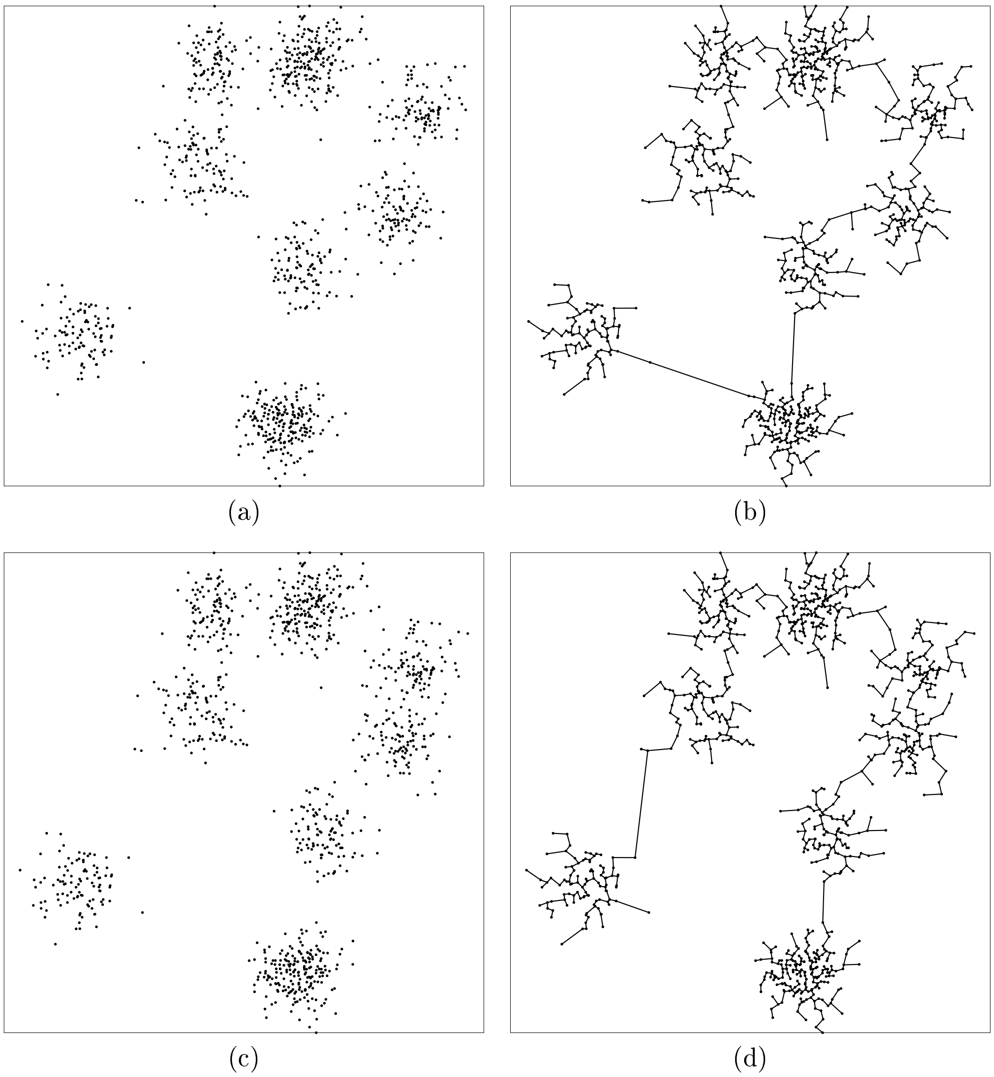
Algorithm mst-shake operates as follows. We use two parameters, a scaling factor, d, and the number of branches to shake, b. Let T be any minimum spanning tree for the seed instance. We perform a rotation for each of the b longest branches (edges) of T. Each of these long branches $e = (u, v)$ breaks T into two components: one component, T_u, contains u and the other end, T_v, contains v. Each vertex in T_u is at least $\omega(e) = \omega(u, v)$ units away from every vertex in T_v, with equality at points u and v. Without loss of generality suppose there are at least as many vertices in component T_u as there are in T_v. We choose an angle θ from a normal distribution with mean 0 and standard deviation $d \cdot \pi/16$, and rotate all the vertices in the smaller component T_v by angle θ around node v. This rotation can be likened to the smaller end of a branch being the one more likely to bend under the force of the wind. Note that the branch e in question does not move during this transformation. The rotation might change the cluster distance between T_u and T_v, but it is less likely to do so using this scheme than if we allowed the branch to rotate as well (by using vertex u as the centre of rotation instead of v), or if we rotated the larger component T_u instead of T_u.

In our experiments, parameter b is always 20, and d is 1. That is, we shake the 20 longest branches in some MST for the seed instance, and the rotation angles are drawn from a normal distribution with mean of 0 and a standard deviation of $\pi/16$.

Figure 8.2 shows instance `clusnorm.904.1000` (a 1000 node instance from Bentley's `clusnorm` distribution using seed 904) and one of the instances generated by mst-shake using these parameters. Part (a) shows instance `clusnorm.904.1000` itself; part (b) shows the minimum spanning tree of `clusnorm.904.1000` used by mst-shake; part (c) shows an instance generated from the MST in part (b); for comparison, part (d) shows a MST of the generated instance in part (c). The length of the minimum spanning trees differ by 0.9%. In fact, the generated instance is different enough from the seed instance that its MST exhibits an abrupt difference: the cluster in the bottom left originally joined the rest of the instance via a long edge to the bottom-most cluster, but in the generated instance it joins via the upper set of clusters. More subtlely, the upper set of clusters moves toward a more horseshoe-shaped arrangement from the original circle-like arrangement.

In general, if d is not too large and b is small relative to the number of vertices in the seed instance, then the generated instances have much of the same cluster structure as the seed instance. In fact, the fine-grain detail is often locally identical; it can only be corrupted if one cluster gets rotated into the same space as an existing cluster, which is rare. Because of these clustering similarities between seed and generated instances, algorithm mst-shake is useful for testing whether cluster compensation behaves as advertised on similarly clustered instances.

The results produced by mst-shake can vary a great deal. There are two reasons for this. First, there may be many choices for the heaviest b edges in a particular MST. Second, there may be many MSTs for the seed instance, with widely varying topological



(a) (b) (c) (d)

Figure 8.2: (a) Seed instance `clusnorm.904.1000`; (b) a minimum spanning tree (MST) for `clusnorm.904.1000`; (c) an instance generated from the MST by mst-shake; (d) a MST for the generated instance in part (c).

structures. This effect is more pronounced for instances where there are many equal-length edges that compete to become branches in a minimum spanning tree. The next section describes an algorithm that overcomes this dependence on which MST is chosen, but at the expense of throwing away a great deal of structural information about the instance.

8.2.3 MST explode and construct

The previous algorithms required a geometric seed instance. Algorithm jitter applies perturbations in a rectilinear coordinate space, and algorithm mst-shake can be viewed as applying perturbations in a polar coordinate space. Furthermore, the action of mst-shake can be highly dependent upon which particular minimum spanning tree we choose to shake. Algorithm mst-explode-construct has neither of these traits, but its output is often radically different from its input. It takes an arbitrary instance for a seed, and produces a two-dimensional instance as output.

In fact, the characterization of an instance G is only the multiset of edge lengths of a minimum spanning tree for G; all other information is thrown away before generating the new instance. The new instance G' is constructed so that it has the same characterization as G: if T and T' are minimum spanning trees for G and G' respectively, then we have the unordered list equality $[\,\omega(e) \mid e \in T\,] = L(T) = L(T') = [\,\omega(e) \mid e \in T'\,]$. An inductive argument shows that multiset $L(T)$ depends only on the graph G and not on the particular minimum spanning tree T. (Prove the result for minimum spanning *forests*, using strong induction on maximum edge length.) We may therefore unambiguously write $L(G)$ in place of $L(T)$. In summary, algorithm mst-explode-construct takes any instance G and produces a Euclidean instance G' so that $L(G') = L(G)$.

Algorithm mst-explode-construct takes a seed instance G on n vertices, and computes the list $L(G)$. This is the explode part. The construct part makes Euclidean instance G' from $L(G)$. We use a snowballing process, gluing vertices and components together until only one component remains,

The construct process is similar to Kruskal's minimum spanning tree algorithm. It begins with pre-determined edge lengths and unknown coordinates (and therefore unknown overall cost function), and over time fixes coordinates so that the edge lengths are edge lengths for a minimum spanning tree for those coordinates. In contrast, Kruskal's algorithm begins with a known overall cost function but unknown edge lengths, and over time fixes edge lengths in a minimum spanning tree for the cost function.

The construction process begins with a pool of n isolated vertices; each vertex has undetermined coordinates and makes up a component by itself. We proceed in $n-1$ steps. After i steps there are $n-i$ components and only the $n-1-i$ largest lengths remain in $L(G)$. During step i (counting from 0) we withdraw the smallest remaining length l from $L(G)$, draw two components a and b at random, and form a new component c by joining a point a_h on the convex hull of a to a point b_h on the convex hull of b by an edge of length l. The new component c is put back into the pool of components.

There may be a range of ways to join the two hull points with an edge of length l; we choose some way in which all the points of component a are at least l units away from all the points of b, with equality for at least one pair. See Figure 8.3. We also have freedom in deciding which hull vertices a_h and b_h to join—any corner vertex will do.

The joining of components a and b to form c does not fix the final positions of the vertices in a and b, but it does fix the positions of the points in c relative to each other. After $n-1$ steps, the list of edge lengths is exhausted, and there is only one component c left in the pool. The positions of all the vertices are fixed relative to each other. All the coordinates are determined once we place an arbitrary vertex at the origin, and we form Euclidean instance G' from the set of coordinates. Algorithm 4 summarizes mst-explode-construct in pseudo-code.

Why does this algorithm perform as promised, namely that $L(G') = L(G)$? Consider the spanning tree T' for G' formed by the edges added during gluing in step 7 of Algorithm 4. One can see that T' is in fact a *minimum* spanning tree for G. Recall that in a minimum spanning tree T, each branch $(u,v) \in T$ can be seen as dividing T into two components T_u and T_v, containing u and v, respectively. The key property of a minimum spanning tree is that branch (u,v) is no heavier than any edge in G having one endpoint in T_u and the other endpoint in T_v. Since we join components only on their convex hulls, and by lengths l that do not decrease, tree T' has this property for G'. It therefore follows that $L(G) = L(G')$.

There is no conceptual difficulty in extending these ideas to higher dimensions. We would maintain convex hulls in higher dimensions, but the gluing process would remain the same. The coordinates and rotations would also be in higher dimensions.

Instances generated by mst-explode-construct can vary a great deal. The output can be rather sparse if all choices are resolved randomly from uniform distributions, *i.e.*, if hull vertices a_h and b_h are chosen uniformly and the join angles are chosen uniformly over

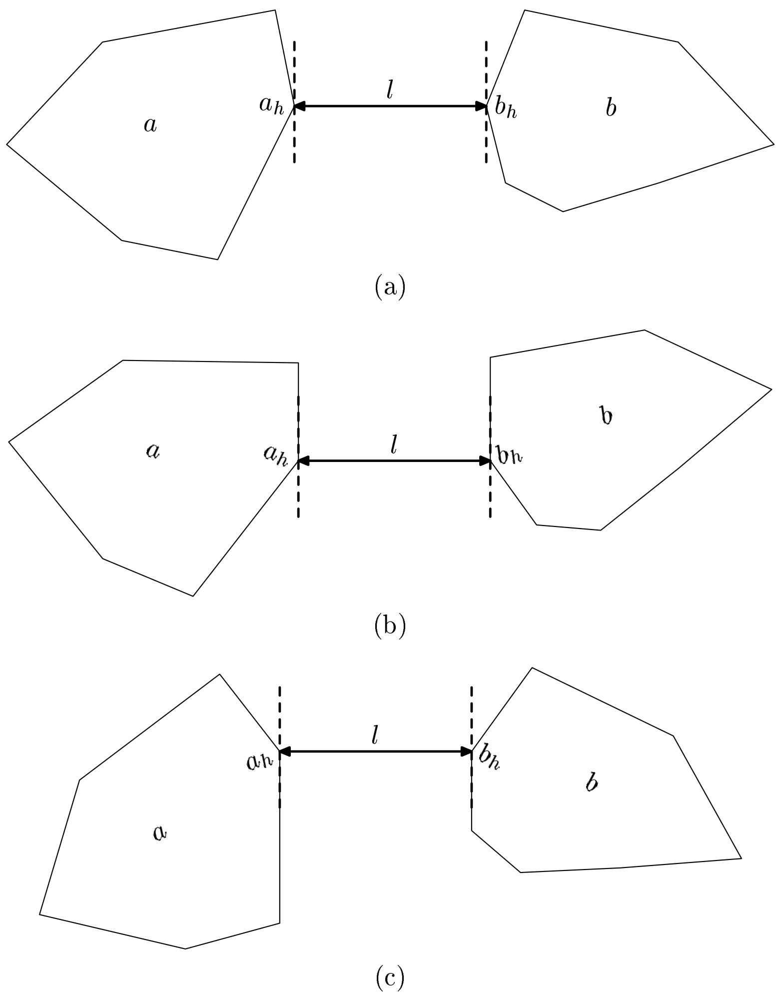


Figure 8.3: There may be a range of ways to join the hull points a_h and b_h by an edge of length l. We choose some way in which all the vertices of a are at least l units from all the vertices of b. Part (a) shows a typical join. Parts (b) and (c) show components a and b rotated to extremes before joining.

Algorithm 4 mst-explode-construct

Require: G is a graph with edge cost function ω.

Ensure: G' is a Euclidean instance with $L(G') = L(G)$.

Let T be a minimum spanning tree for G.
Let $L = [\,\omega(e) | e \in T\,]$
$C :=$ a set of $|L|$ +1 points with undetermined coordinates
// C is the current set of components.
for $l \in L$ from smallest to largest **do**
    Choose a and b at random from C.
    Choose vertices a_h and b_h on the convex hulls of a and b, respectively.
    Form component c by joining a_h to b_h by an edge of length l; use the packing factor to bias the join angles. Ensure that $\omega(a_i, b_j) \geq l$ for all vertices a_i in a and b_j in b.
    $C := C \setminus \{a, b\} \cup \{c\}$
end for
Only one component c remains in C. Place one vertex at the origin, thus fixing all the coordinates.
Form 2-d Euclidean instance graph G from the coordinates of points in c.

the allowable range. We can bias the choices to make more compact instances: when joining two components, try to line up long edges of the convex hulls so that they face each other. We use two parameters to specify the bias. A *join length bias* ranging from 0 through 1 can be used to specify which face on each component is lined up against the other component, with 0 being the shortest face and 1 being the longest face. (A random uniform face choice is specified with a negative join length bias.) A positive *packing factor* is used to weight our choice of join angle: large values make the chosen face run parallel to the other component, while low values make the chosen face turn as far away from the other component as allowable. In the extreme —a join length bias of 1 and a packing factor of around 100— the algorithm chooses hull vertices a_h and b_h and the join angles so that the longest edges on the convex hulls run parallel l units apart. See Figure 8.4.

Figure 8.5 shows some possible outputs for a single seed instance, `dsj1000` from TSPLIB. Part (a) shows `dsj1000` itself.

Parts (b) through (f) show five instances generated by mst-explode-construct from `dsj1000`. Part (b) was generated without any bias in choosing hull vertices or join angles.

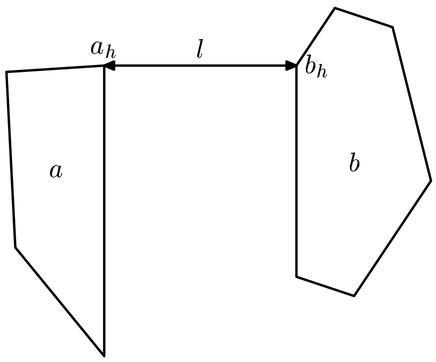


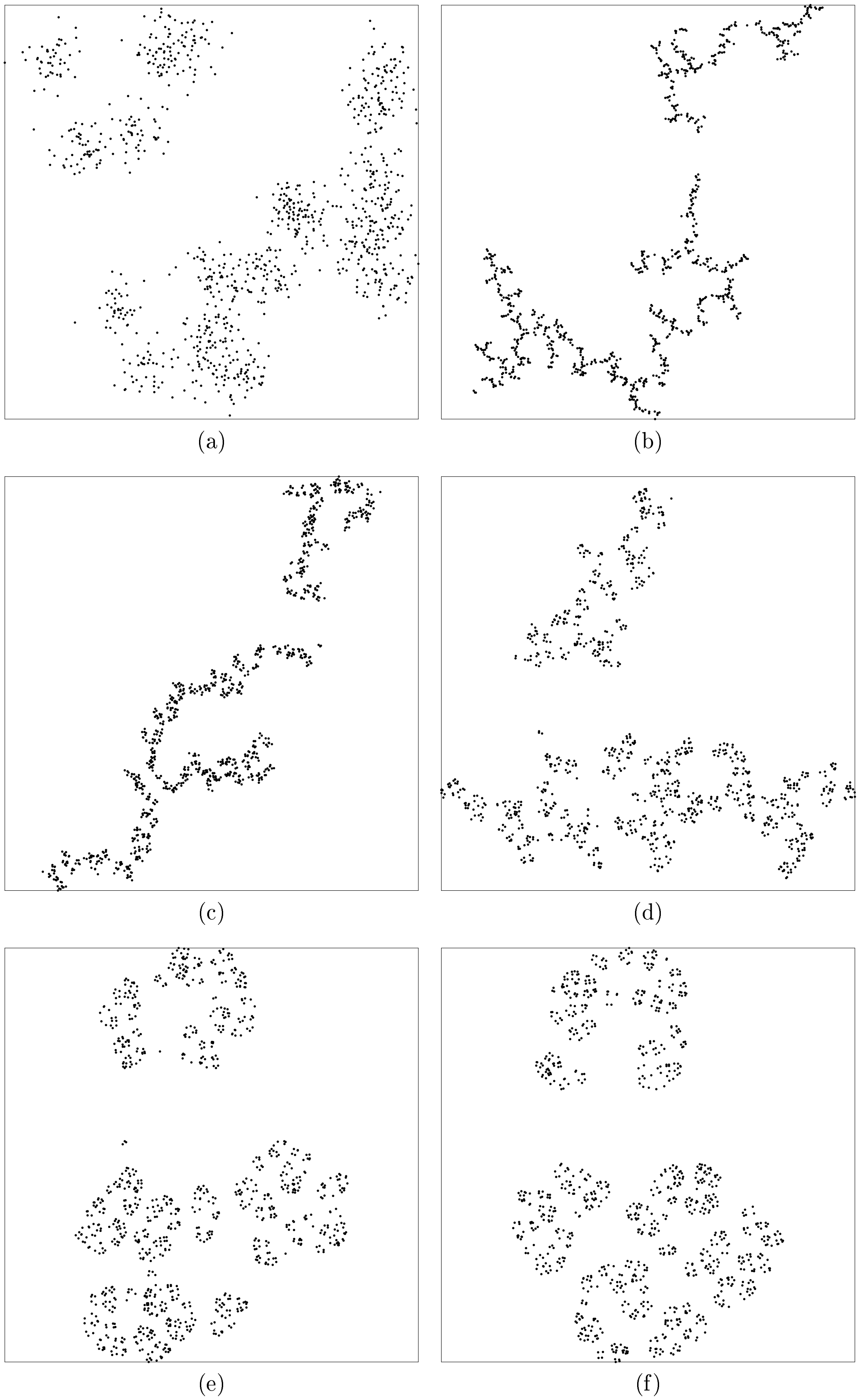
Figure 8.4: More compact instances are often formed if the longest edges of hulls a and b are made to run parallel in the new component.

Its sparseness is typical of such instances, as noted above. Parts (c) through (e) were generated using the same seed (and hence the same sequence of choices for components a and b), but with bias parameters chosen to make progressively more compact instances. Part (f) uses the same bias parameters as part (e), but uses a different random number seed and therefore shows some of the variability we can get given fixed bias parameters. Table 8.6 records the parameter values for each of the figures.

8.2.4 MST dangle and construct

Generally, the outputs of mst-explode-construct are rather "clumpy". There are many small clusters of similar size separated by increasingly large gaps. This is caused in large part by the way components are chosen for gluing: step 5 of Algorithm 4 chooses two components a and b at random from the remaining pool of components. In the early stages it is unlikely that many of the components in the pool will have many vertices—the random withdrawals would have been favouring the now-large components at the expense of the smaller components. So all the small edges are dispersed among a great number of components that then get separated by the larger edges. We are building random trees, changing only the lengths of the branches.

We would like to take an arbitrary minimum spanning tree and from it build a geometric instance that retains as much as possible of the tree structure of the original. If we are lucky, the whole MST structure is preserved, and therefore the cluster distance *function* is the same on the generated instance as it is on the seed instance, modulo node renaming. This is the goal of algorithm mst-dangle-construct. It uses the same bottom-up



(a) (b)

(c) (d)

(e) (f)

Figure 8.5: Seed instance `dsj1000` (part (a)), and five instances generated from it by mst-explode-construct. See also Table 8.6.

Figure	Seed	Join length bias	Packing factor
8.5(b)	1021	-1	(uniform) 1
8.5(c)	1021	0	0.1
8.5(d)	1021	0.75	10
8.5(e)	1021	1	100
8.5(f)	1099	1	100

Figure 8.6: Parameters used by mst-explode-construct to generate parts (b) through (f) of Figure 8.5 from instance `dsj1000`. A negative join length bias indicates a hull vertex and face chosen at random from a uniform distribution.

gluing idea that is the essence of mst-explode-construct, but it is more careful about how components are chosen and glued.

Algorithm mst-dangle-construct retains the entire minimum spanning tree T of the seed instance and tries to build an isomorphic copy of it, T'. The idea of saving the structure of the minimum spanning tree is reflected in the mental picture of picking up the MST skeletal structure of the original instance, dangling it to shake off the residual geometry, and then trying to build a new instance around the mostly intact skeleton.

In the beginning, the isolated vertices that become the vertices of T' are given identities matching them with vertices in T. In `mst-explode-construct` we examined the *lengths* of edges in T and from them formed edges in T' having those lengths. In `mst-dangle-construct` we examine each *edge* (u, v) from T from shortest to longest, and build new edges in T', keeping the identities of vertices u and v and retaining the length $\omega(u, v)$ of the edge as well. The two components C_u and C_v containing u and v are withdrawn from the pool and joined by an edge of length $\omega(u, v)$; the new component is returned to the pool. If possible, u and v themselves are chosen as the vertices to be joined, thus recreating the edge (u, v) as a branch in the new minimum spanning tree T'. For this to happen, however, both u and v must appear on the convex hull of their respective components. This is not always possible, as we will see below. If u or v is not on its component's convex hull, then in the new tree u will be more than $\omega(u, v)$ units away from v. We try to minimize this deviation in some sense by choosing a hull join point and face for that component according to the maximum join length bias (value 1). That way the component's longest face is adjacent to the joining edge and the hull join point is fixed. In other respects mst-dangle-construct proceeds as mst-explode-construct does. Algorithm 5 is the pseudo-code for mst-dangle-construct.

Vertices u and v will not always be on the convex hulls of their respective components. We call such an occurrence a "force failure" because the algorithm failed to force the new tree to be isomorphic to the seed tree. There is an infinite supply of minimum spanning trees that cause force failures. We can use our knowledge of sphere close-packings to create such a family, noting that the output of mst-dangle-construct is always a geometric instance. In a given dimension d, there is a bound $s(d)$ on the number of spheres of a fixed radius which can be made to touch a central sphere of equal radius without overlapping either the central sphere or each other. For example, a honeycomb-like packing shows that 6 circles may touch a central circle, and so $s(2) \geq 6$; one can show that 7 circles is too many, and therefore $s(2) = 6$. There is simply "not enough space" in d dimensions to pack more than $s(d)$ spheres around a central sphere. We can always force failures in mst-dangle-construct by feeding it a seed tree with its shortest $s(2) + 1$ edges being of equal length, each connecting one of $s(2) + 1$ vertices to a central hub vertex. Given this kind of input, mst-dangle-construct will inevitably place the hub in the interior of its component before all the edges incident upon the hub are consumed. Even if we extend mst-dangle-construct to d dimensions, we could always use $s(d) + 1$ equal length edges.

The output of mst-dangle-construct is disappointing. On typical geometric seed instances, we get a force failure rate of 15 to 25 percent. Sometimes the force failure rate is as high as 65 percent for regular grid-like instances; regular grids are efficient packings, concurring with our intuition about sphere packings being difficult for mst-dangle-construct. On the bright side, for strung-out thread-like seed instances such as Bentley's `arith`, `cubediam`, or `spokes` classes, the force failure rate is very low at 2 percent or less.

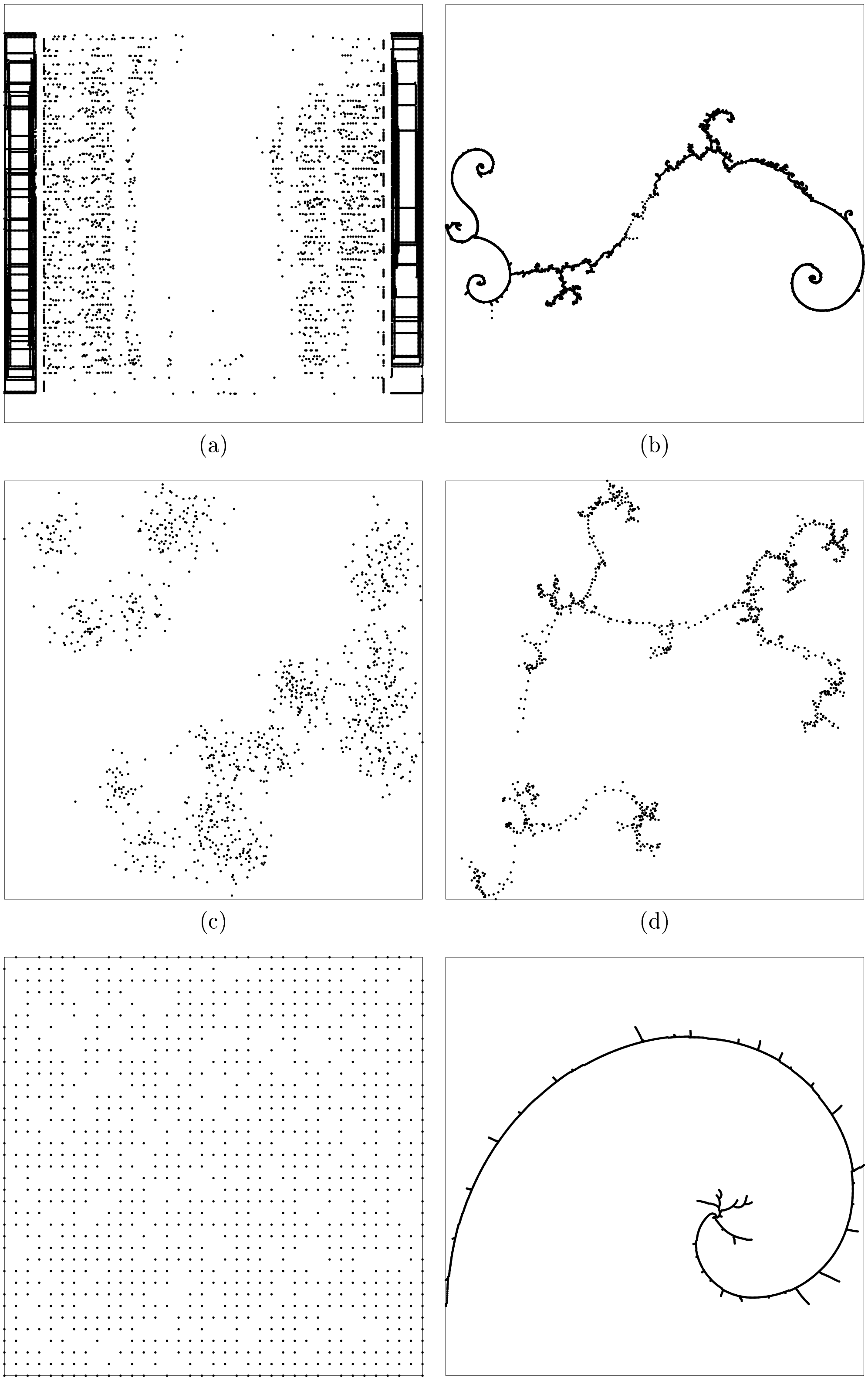
However, visual inspection is the most damning: the instances just don't look right. No matter the seed instance, the outputs are dominated by spiral-like structures. One is reminded of the rough traces ions leave in cloud chambers; of fractal s-sets [31]; or of visuals from Tim Burton's *The Nightmare Before Christmas*. Figure 8.7 shows several seed instances and a corresponding output for each.

Algorithm 5 mst-dangle-construct

Require: G is a graph with edge cost function ω.

Ensure: G' is a Euclidean instance with $L(G') = L(G)$. Try to make G' have a MST T' with as much of the structure as T as possible.

  Let $T = (V, E_T)$ be a minimum spanning tree for G.
  $C :=$ a copy of V, each vertex having undetermined coordinates. C is the current set of components.
  force_failures $:= 0$
  for $(u, v) \in E_T$ from smallest weight to largest weight **do**
    Choose C_u and C_v from C, containing u and v respectively.
    if u is on the convex hull of C_u **then**
      $a_h := u$
    else
      Choose a_h adjacent to a longest edge on the hull of C_u
      force_failures $+= 1$
    end if
    if v is on the convex hull of C_v **then**
      $b_h := v$
    else
      Choose b_h adjacent to a longest edge on the hull of C_v
      force_failures $+= 1$
    end if
    Form component c by joining a_h to b_h by an edge of length $\omega(u, v)$; use the packing factor to bias the join angles. Ensure that $\omega(a_i, b_j) \geq \omega(u, v)$ for all vertices a_i in C_u and b_j in C_v.
    $C := C \setminus \{C_u, C_v\} \cup \{c\}$
  end for
  Only one component c remains in C. Place one vertex at the origin, thus fixing all the coordinates.
  Form 2-d Euclidean instance graph G from the coordinates of points in c.



(a) (b) (c) (d) (e) (f)

Figure 8.7: Output of mst-dangle-construct: (a) TSPLIB seed instance `pla7397`; (b) output of mst-dangle-construct on `pla7397`; (c) TSPLIB seed instance `dsj1000`; (d) output of mst-dangle-construct on `dsj1000`; (e) Bentley instance `grid.828.1000`; (f) output of mst-dangle-construct on `grid.828.1000`.

8.2.5 Cluster and noise

The algorithms described so far generate only two-dimensional Euclidean instances. There may be a need to generate non-geometric instances. Algorithm cluster-noise is one possibility. Its output is a symmetric distance matrix.

We are trying to preserve the essential clustering of the seed instance. One way to do so is to just output the cluster function as the distance matrix. To add variety, we add noise to each edge weight. Specifically we set,

$$\omega_{G'}(u,v) = c_G(u,v) \cdot (1 + x(u,v))$$

where x is drawn uniformly from $[0,1]$. The triangle equality is not necessarily preserved by this transformation.

Unfortunately, we cannot depict non-geometric instances as we did with the geometric instances. However, we can imagine that this transformation collapses much of the structure of the seed instance: between two vertices u and v, all that matters is the longest hop between them, their cluster distance. Long chains of vertices are jumbled together, separated only by the white noise of the x function.

8.3 Corrupting cluster structure

The previous sections describe instance generating algorithms that hope to preserve the essential cluster structure of the seed instance. They vary from the very conservative jitter to the very radical mst-explode-construct and cluster-noise. If our intuition about the behaviour of Lin-Kernighan is correct, then given a seed instance G, Lin-Kernighan will behave similarly on the instances generated by those algorithms from G as it does on G itself. That would prove that our intuition about clustering and the heuristic is "right."

But we would also like to know that our intuition is "not wrong." We would like to tweak an instance G to produce another instance G' having less severe clustering. One would hope that if G is sharply clustered then the baseline Lin-Kernighan heuristic would perform better on G' than it does on G. Furthermore, we would hope that any performance advantage given by cluster compensation would be reduced on instance G' when compared with the advantage it gives on instance G.

8.3.1 Cluster discount

A natural way to produce a less-clustered instance G' from G is to make G' identical to G except that we discount its edge weight function. Weight function $\omega_{G'}$ is weight function ω_G discounted by the cluster distance function c_G. That is, we set

$$\omega_{G'}(u, v) = \omega_G(u, v) - c_G(u, v)$$

for all vertices u and v. We call this transformation cluster-discount. It takes any instance as input and produces a non-geometric instance as output. It effectively eliminates the largest cluster gap from G, and reduces the impact of many of the others. Note that a minimum spanning tree for G will almost certainly not be a minimum spanning tree for G'.

Unfortunately, this transformation forces some edges to have zero weight. These boundary cases may be unrealistic for many applications, and may break many assumptions implicit in an application. To mitigate these problems, we can soften the discount by redefining the transformation to

$$\omega_{G'}(u, v) = \omega_G(u, v) - \alpha \cdot c_G(u, v)$$

where α is some value between 0 and 1. For variety, we can choose a new α randomly in $[0, 1]$ for each edge (u, v).

Another drawback of this transformation is that even if its input is a geometric instance, its output is not. It may therefore be difficult to compare the running time performance of the heuristic on G and its associated output G' because the preprocessing algorithms (*e.g.*, minimum spanning tree and candidate set generation) differ for the two classes of instances. However, the Lin-Kernighan optimization phase should be comparable between the different executions.

8.3.2 Cluster infill

Infilling is a less dramatic way to produce a less clustered instance G' from a seed instance G. Infilling puts new vertices into the largest inter-cluster gaps. Let T be a minimum spanning tree for G. Algorithm infill produces G' from G by placing a new vertex uv "somewhere between" vertices u and v, for each edge (u, v) in T.

If G is a geometric instance, then "somewhere between" has several obvious interpretations, one of which is "exactly halfway between." Bisecting each MST edge cuts existing cluster gaps in half. Unless G is very sharply clustered, a minimum spanning tree for the new instance will not resemble T.

If G is not geometric, then we must be more creative in interpreting "somewhere between." The difficulty of course lies in the fact that vertices do not have coordinates, yet we still must define edge weights appropriately so that cluster distances are in fact smaller in the new instance. For example, we can set the distances from new vertex uv to other vertices as follows. We can set

$$\omega_{G'}(w, uv) = \frac{1}{2}\omega_G(u, v) + \begin{cases} \min(\omega_{G'}(w, u), \omega_{G'}(w, v)) & \text{if } w \text{ is a new vertex} \\ \min(\omega_G(w, u), \omega_G(w, v)) & \text{otherwise} \end{cases}$$

and further declare that $\omega_{G'}$ is symmetric. (This definition recurses one level deep when w is a new vertex, and does not recurse otherwise.) If the original weight function ω_G satisfies the triangle inequality, then the new function $\omega_{G'}$ does as well.

If G has n vertices, then its minimum spanning trees each have $n - 1$ edges, and so the base infill algorithm creates a new instance with $2n - 1$ vertices. It may be difficult to judge the relative performance of an optimization heuristic on such widely differing instance sizes. Of course, we can generate instances with only $n + k$ vertices by inserting new vertices only for the longest k edges in a MST for G. We fill in the *longest* edges because they bridge the largest cluster gaps.

8.4 Variations

All the instance generating algorithms described so far produce instances with the same number as or not many more vertices than the seed instances given to them. To test the scalability of optimization algorithms, we would like to automatically generate ever-larger instances. This is straightforward for randomly generated instances such as the random distance matrices and Bentley's geometric classes of instances. In those cases, one can simply specify that more vertices be generated. But again, we would like to generate instances with clustering characteristics similar to a seed instance.

8.4.1 Distill, expand, and generate

One way to generate larger instances is to leverage the two-phase distill and generate nature of the algorithms given earlier. We can add a middle phase, *expansion*. Given an instance G, we produce as usual a distillation $\mathit{Distill}(G)$ of the essential cluster structure of G. We can then expand *Distill*(G) into *Expand*(*Distill*(G)) producing a distillation of a larger instance. We can then feed *Expand*(*Distill*(G)) to the generating phase to produce an instance G' with more vertices than G has. The trick of course is in defining an appropriate expansion function *Expand*. In many ways this just renames the problem. We must find a way to expand a distillation, instead of finding a way to expand the graph itself. However, this approach does represent a savings in conceptual work.

Some of the distill and generate algorithms may be more amenable to expansion than others. Those with a less structured characterization of an instance are perhaps the most amenable since expansion is then simplified. Algorithm mst-explode-construct is one of the easier algorithms to extend. Its characterization of graph G is the unordered list $L(G)$ of edge lengths used in a minimum spanning tree T for G. We can view $L(G)$ itself as having been drawn from a probability distribution $\tilde{L}(G)$. If we can model $\tilde{L}(G)$ appropriately, then we can draw other lists of any desired size from it. These lists can then be passed on to the generating phase of mst-explode-construct to create new instances of any size.

For example, Figure 8.8 shows the edge lengths in minimum spanning trees for instances `dsj1000`, `pr1002`, and `uni.820.1000` (a uniform geometric instance with 1000 vertices and using seed 820). For any plot $l(e)$ of this kind, we can define a probability distribution function as follows. Find a smooth curve $f(t)$ through the discrete plot $l(e)$ and rescale the domain of f to $[0, 1]$. We can then generate a new set L' of n edge lengths by taking uniform samples x_i from $[0, 1]$ and using the values $f(x_i)$, *i.e.*,

$$L' = \{f(x_i) \mid x_i \text{ sampled uniformly from } [0,1],\ i = 1, \ldots, n\}.$$

Note that the largest cluster gaps are generated by the longest edges. There may be only a few relatively long edges, in which case they can easily be overlooked by the random draw. To ensure that the large-scale cluster gaps are retained in the new instance, we may want to force a small number of long edges to be present in any list generated from the distribution.

The distill-expand-generate paradigm is quite general, with many ways to implement any of the three phases. We have described only one way of applying it to mst-explode-construct, which itself is only one algorithm in the distill-generate paradigm. We leave further examination of the expansion paradigm to future work.

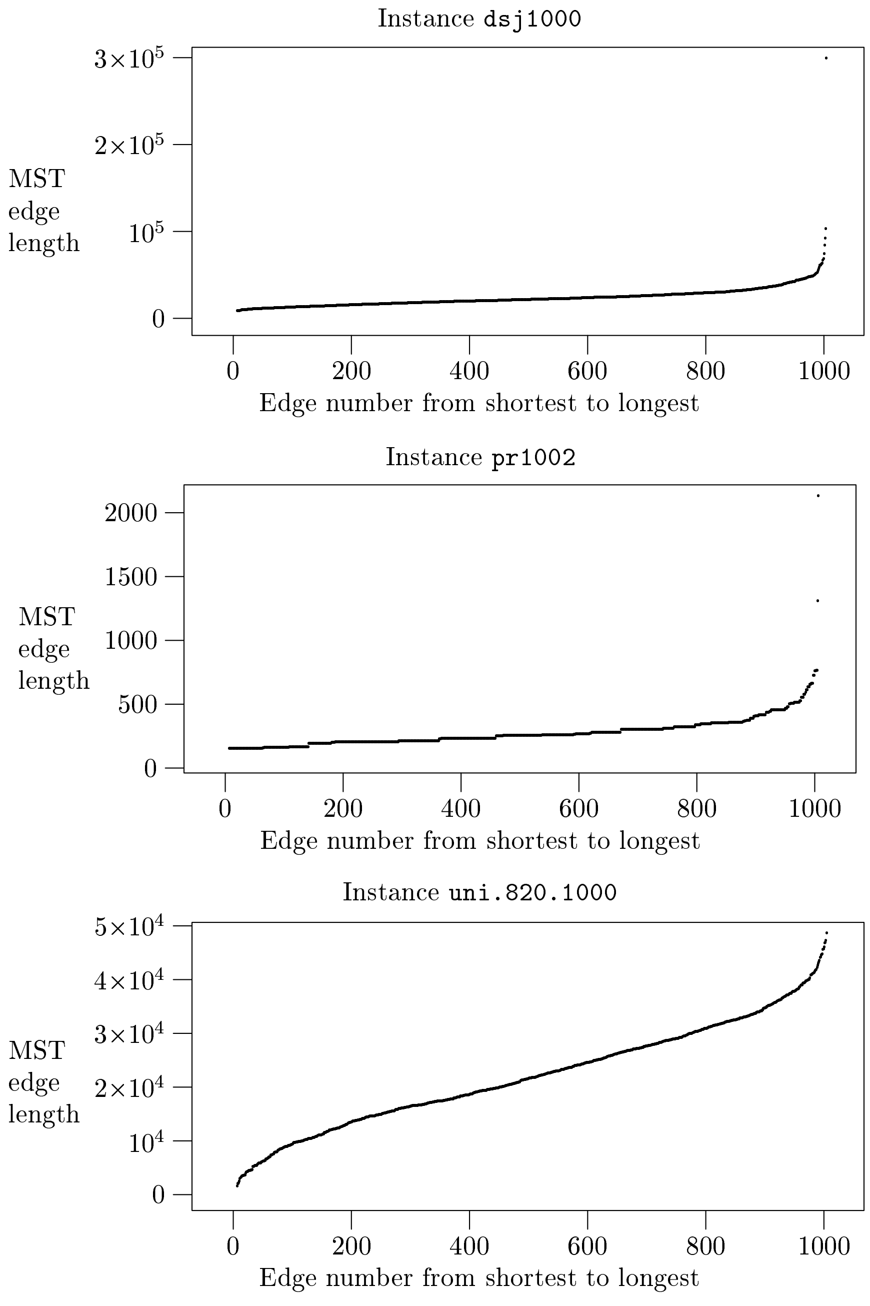


Figure 8.8: MST edge lengths in `dsj1000`, `pr1002`, and `uni.820.1000`, plotted from shortest to longest.

8.5 Summary

This chapter motivates the use of algorithms for producing new instances from old. The main reason for doing so is to allow us to circumvent the methodological limitations of working with only a small number of test cases.

A secondary reason is that instance generating algorithms let us test our intuition about the behaviour of heuristics. We can tailor the generating algorithms to emphasize or diminish those features of the input that we believe cause bad behaviour on the part of the optimization heuristic.

There are endless possibilities for instance generation algorithms. This chapter describes seven of them. They differ in how radically they transform seed instances, and on the restrictions they place on the form of the input. The algorithms fall into two categories. Algorithms of the first kind try to mimic in the new instances the basic cluster structure of the seed instance. Those of the second kind try to generate new instances with less severe cluster structure than the seed instance.

There are of course motivations for generating synthetic instances other than just testing how cluster structure affects a heuristic and its modifications. There are certainly other suitable data generation algorithms for those purposes as well.

The next chapter presents the results of experiments on instances generated by the seven generating algorithms.

Chapter 9

Testing the instance generators

The previous chapter describes seven instance generation algorithms, many of which allow almost unlimited variation through random number seeds and in some cases continuous parameters. It is of course impossible to test all of the possibilities. We instead perform a small number of experiments for a few variations on a small number of seed instances.

The following sections describe the seed instances and the instances generated from them, and the results of experiments on the generated data with the Lin-Kernighan heuristic for both the Traveling Salesman and the minimum weight perfect matching problems.

9.1 Selected seed instances

We want to see the effect of cluster structure on the behaviour of Lin-Kernighan on data generated by the different generation algorithms, so we choose seed instances representing a variety of clustering characteristics. Instances `uni.820.1000` and `dsjr.45.1000` are representatives for uniform instances; instance `pr1002` is a mildly clustered instance, and instances `dsj1000` and `corners.827.1000` are sharply clustered instances. Instance `dsjr.45.1000` is a random distance matrix, so some of the generation algorithms are not applicable. All the others are two-dimensional geometric instances.

Except for infill, all the algorithms are randomized. Each randomized generation algorithm was used to produce two new instances from each seed instance. Algorithm infill was used to create one new instance (with $1.1 \cdot n$ vertices) from each seed instance. For the TSP, both versions of Lin-Kernighan were run twice on each generated instance. The percentage excess over an approximate Held-Karp bound and the running times are presented as arithmetic means over all experiments for each seed instance. For matching, both versions of the heuristic were run three times, and the percentage excess over optimal and the running times are presented as arithmetic means over all experiments for each seed instance. All other parameters were set as for the experiments reported in Chapters 6 and 7.

9.2 Results for the TSP

This section summarizes the results of running both variants of the Lin-Kernighan heuristic for the TSP on the seed instances and on the instances produced from those seeds by the seven instance generating algorithms.

The quality of output is always measured as percentage excess over an approximated Held-Karp bound computed using the iterative ascent method described in Section 5.4.

For the sake of comparison, the TSP results for the seed instances are collected in Tables 9.1 and 9.2. The results for TSP runs on Bentley's instance distributions, Tables 6.8 and 6.9, did not include results for n iteration runs. The results given here for instance `corners.827.1000` are the averages over 6 runs.

The rows in Tables 9.1 and 9.2 are arranged in order from least to greatest time advantage given by cluster compensation. That is, later rows have larger ratios of LK running time to LKcc running time. The instance ordering also correlates with the severity of clustering: more sharply clustered instances come later. Uniform distance matrix instances and uniform geometric instances are not at all sharply clustered, so we order them solely by the running time advantage cluster compensation provides.

9.2.1 Jitter

Tables 9.3 and 9.4 show the results for the instances generated by `jitter` on the geometric seed instances. The results are quite consistent with the results for the seed instances themselves. The percentage excesses over Held-Karp are quite similar, as are the absolute running times and their ratios. In particular, the absolute runtimes rarely differ by more than 20% from those on the seed instances.

We should not be surprised by these results because `jitter` is a very minor perturbation in rectangular coordinate space. At least we have the satisfaction that the behaviour of the Lin-Kernighan heuristic in general and our results in particular do not appear to be

Percentage excess

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
dsjr.45.1000	3.25	2.96	-0.29	2.07	2.24	0.17	1.36	1.06	-0.30
uni.820.1000	1.74	1.79	0.05	1.21	1.26	0.05	0.93	0.89	-0.04
pr1002	2.28	2.60	0.32	1.73	1.76	0.03	1.15	1.17	0.02
dsj1000	2.07	2.23	0.16	1.31	1.28	-0.03	0.94	0.91	-0.03
corners.827.1000	2.65	2.59	-0.06	1.83	1.85	0.02	1.45	1.50	0.05

Table 9.1: Quality of output produced by Lin-Kernighan for the TSP on seed instances.

Running time

Instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
dsjr.45.1000	4.61	3.56	1.29	37.51	19.97	1.88	258.59	133.32	1.94
uni.820.1000	1.19	0.83	1.43	7.87	4.09	1.92	62.40	31.00	2.01
pr1002	4.15	1.45	2.86	18.98	6.31	3.01	132.58	48.33	2.74
dsj1000	16.99	2.70	6.29	166.98	21.84	7.65	1606.73	162.12	9.91
corners.827.1000	33.73	0.80	42.16	522.76	16.26	32.15	4241.17	101.64	41.73

Table 9.2: Time taken on seed instances by the Lin-Kernighan heuristic for the TSP.

Percentage excess

Input to jitter	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
uni.1000	1.81	1.95	0.14	1.00	1.20	0.20	0.78	0.81	0.03
pr1002	2.52	2.71	0.19	1.81	1.75	-0.06	1.33	1.37	0.04
dsj1000	2.21	2.22	0.01	1.20	1.27	0.07	0.92	0.92	-0.01
corners.1000	2.44	2.50	0.06	1.81	1.88	0.06	1.46	1.48	0.02

Table 9.3: Quality of output produced by Lin-Kernighan for the TSP on jitter instances.

Running time

Input to jitter	1 iteration LK	1 iteration LKcc	1 iteration LK/LKcc	$n/10$ iterations LK	$n/10$ iterations LKcc	$n/10$ iterations LK/LKcc	n iterations LK	n iterations LKcc	n iterations LK/LKcc
uni.1000	1.15	0.74	1.55	7.36	3.15	2.34	53.16	26.36	2.02
pr1002	4.22	1.75	2.40	18.22	7.14	2.55	139.99	50.61	2.77
dsj1000	18.07	2.48	7.30	202.22	17.52	11.55	1700.13	153.81	11.05
corners.1000	34.79	1.16	30.12	513.19	11.18	45.90	4192.26	154.66	27.11

Table 9.4: Time taken on jitter instances by the Lin-Kernighan heuristic for the TSP.

very sensitive to small perturbations in the input.

9.2.2 MST shake

Tables 9.5 and 9.6 show the TSP results for instances generated by mst-shake.

Percentage excess

Input to mst-shake	1 iteration LK	1 iteration LKcc	1 iteration LKcc −LK	$n/10$ iterations LK	$n/10$ iterations LKcc	$n/10$ iterations LKcc −LK	n iterations LK	n iterations LKcc	n iterations LKcc −LK
uni.1000	1.58	1.63	0.04	1.21	1.29	0.07	0.90	0.87	-0.03
pr1002	2.58	2.70	0.12	1.81	1.60	-0.21	1.23	1.19	-0.04
dsj1000	2.25	2.17	-0.09	1.25	1.43	0.17	0.72	0.72	0.00
corners.1000	2.92	2.88	-0.04	1.56	2.18	0.62	1.28	1.23	-0.06

Table 9.5: Quality of output produced by Lin-Kernighan for the TSP on mst-shake instances.

The quality of the tours produced by mst-shake is close to that produced by the heuristics on the seed instances. The running times are similar for those based on `uni.820.1000`, and are longer for those based on the other seeds. In comparison to jitter, the mst-shake transformation introduces much more variance in the running times. Even so, the advantage given by cluster compensation on mst-shake instances is for the most part consistent with the advantage given on the seed instances themselves.

Running time

Input to mst-shake	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
uni.1000	1.10	0.71	1.54	7.12	3.94	1.81	62.98	31.32	2.01
pr1002	4.39	2.81	1.56	22.70	12.06	1.88	151.23	92.75	1.63
dsj1000	20.24	3.72	5.44	251.66	46.98	5.36	1951.05	323.44	6.03
corners.1000	38.27	4.57	8.37	439.93	48.96	8.98	3765.49	179.68	20.96

Table 9.6: Time taken on mst-shake instances by the Lin-Kernighan heuristic for the TSP.

9.2.3 MST explode and construct

Tables 9.7 and 9.8 show the TSP results for instances generated by mst-explode-construct. Three sets of experiments were performed, corresponding to three pairs of choices for the join length bias and the packing factors. The tables show the results of the experiments with all three pairs of settings. Join length bias of -1 (random hull point) was paired with a packing factor of 1 (prefix `j-1.p1`), join length bias of 0.75 was paired with packing factor 10 (prefix `j0.75.p10`), and join length bias of 1 (the maximum) was paired with packing factor 100 (prefix `j1.p100`).

The percentage excesses after a single iteration are higher for the generated instances as compared with the seed instances. The pattern continues for all the instances generated with join length bias -1 and packing factor 1. The pattern also appears for all the instances generated from the `corners.827.1000` seed instance. For the other instances and other join length bias and packing factors the excesses are similar to the excesses on the respective seed instances.

On these instances the percentage excess difference between the variations of Lin-Kernighan is usually less than 0.5%. (There are a couple of outlier cases where Lin-Kernighan with cluster compensation produces answers more than 1% closer to the Held-Karp lower bound.)

As for running times, the three pairings of join length bias and packing factors fare quite differently. The first pair have join length bias -1 and packing factor 1. Recall that they create rather strung-out instances, no matter the structure of the seed instance.

Percentage excess

Input to mst-explode-construct	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
j-1.p1.dsjr.1000	6.55	6.12	-0.44	3.43	3.86	0.43	2.87	2.87	0.00
j-1.p1.uni.1000	6.69	5.84	-0.85	2.96	3.56	0.60	2.83	2.81	-0.02
j-1.p1.pr1002	8.75	7.43	-1.32	5.77	4.58	-1.19	5.77	4.42	-1.35
j-1.p1.dsj1000	3.54	3.41	-0.12	2.46	2.55	0.09	2.19	2.19	0.00
j-1.p1.corners.1000	11.38	9.59	-1.79	7.18	7.56	0.38	6.77	7.22	0.46
j0.75.p10.dsjr.1000	2.13	2.62	0.48	1.10	1.10	0.00	0.99	0.99	0.01
j0.75.p10.uni.1000	4.25	2.75	-1.50	2.16	1.44	-0.73	0.90	0.90	0.00
j0.75.p10.pr1002	2.22	2.17	-0.05	0.83	0.96	0.12	0.80	0.80	-0.01
j0.75.p10.dsj1000	4.49	4.07	-0.42	1.32	1.25	-0.07	0.93	0.92	-0.01
j0.75.p10.corners.1000	5.01	5.12	0.12	3.61	3.58	-0.03	3.30	3.36	0.06
j1.p100.dsjr.1000	2.90	2.91	0.01	1.25	1.61	0.37	0.90	0.94	0.05
j1.p100.uni.1000	2.25	2.68	0.43	1.42	1.42	0.00	0.94	1.04	0.11
j1.p100.pr1002	2.73	2.94	0.21	1.40	1.71	0.32	0.89	0.97	0.08
j1.p100.dsj1000	3.06	3.14	0.08	1.58	1.33	-0.26	1.16	1.16	0.00
j1.p100.corners.1000	4.93	4.99	0.06	3.76	3.86	0.10	3.53	3.55	0.02

Table 9.7: Quality of output produced by Lin-Kernighan for the TSP on `mst-explode-construct` instances. Join length biases are -1 (random choice), 0.75, and 1, and corresponding packing factors are 1, 10, and 100.

Running time

Input to mst-explode-construct	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/LKcc	LK	LKcc	LK/LKcc	LK	LKcc	LK/LKcc
j-1.p1.dsjr.1000	55.02	44.66	1.23	487.51	506.17	0.96	5273.18	4427.51	1.19
j-1.p1.uni.1000	33.35	43.06	0.77	495.26	437.93	1.13	5067.86	5203.75	0.97
j-1.p1.pr1002	33.91	48.04	0.71	387.43	527.27	0.73	3329.13	4519.94	0.74
j-1.p1.dsj1000	50.37	40.41	1.25	508.67	464.60	1.09	5589.41	4664.69	1.20
j-1.p1.corners.1000	52.95	41.62	1.27	592.49	388.09	1.53	5273.90	3016.52	1.75
j0.75.p10.dsjr.1000	17.03	15.50	1.10	161.09	119.53	1.35	1574.38	793.15	1.98
j0.75.p10.uni.1000	16.84	9.26	1.82	106.37	76.47	1.39	591.47	503.15	1.18
j0.75.p10.pr1002	15.21	8.62	1.76	150.34	81.02	1.86	1180.04	589.97	2.00
j0.75.p10.dsj1000	29.65	15.29	1.94	145.79	73.59	1.98	1151.10	584.59	1.97
j0.75.p10.corners.1000	48.30	12.86	3.75	454.48	76.91	5.91	3787.13	525.18	7.21
j1.p100.dsjr.1000	14.12	6.25	2.26	126.46	36.95	3.42	701.61	220.75	3.18
j1.p100.uni.1000	2.38	5.25	0.45	18.86	22.37	0.84	128.37	96.16	1.33
j1.p100.pr1002	11.93	7.56	1.58	75.93	35.55	2.14	515.27	162.02	3.18
j1.p100.dsj1000	20.04	8.98	2.23	148.19	45.52	3.26	1093.19	328.40	3.33
j1.p100.corners.1000	30.87	11.72	2.63	332.33	95.59	3.48	3011.26	771.68	3.90

Table 9.8: Time taken on `mst-explode-construct` instances by the Lin-Kernighan heuristic for the TSP. Join length biases are -1 (random choice), 0.75, and 1, and corresponding packing factors are 1, 10, and 100.

The running times on those instances are nearly uniform and quite long. The uniformity comes despite the highly varied behaviour of the Lin-Kernighan heuristic on the seed instances.

Interestingly, cluster compensation forces n-iteration Lin-Kernighan to run 25 percent longer on the instances generated from `pr1002` with join length bias -1 and packing factor 1. This is an average; one of the instances ran forced Lin-Kernighan with cluster compensation to run four times longer than base Lin-Kernighan. This is by far the worst slowdown produced by cluster compensation that we have seen so far, and we have no explanation for it.

With the second pair of parameters, join length bias 0.75 and packing factor 10, there is still uniformity in the running times after n iterations. However, there is more differentiation with fewer iterations. Still, the absolute running times are significantly greater than for the seed instances themselves.

The last pair of parameters, join length bias 1 and packing factor 100, is the most faithful to the results on the seed instances. Run times are often significantly longer on the generated instances than they were on the seed instances. But the relative ratios of run times among the different distributions are similar for the generated data as they were for the seed data. Except for the data generated from the quite uniform seed instance `uni.820.1000`, the ratios of run times still show a significant advantage is given by cluster compensation. The ratios themselves do not venture above 4 for the generated data, yet they were as high as 42 on the seed data.

9.2.4 MST dangle and construct

Tables 9.9 and 9.10 show the TSP results for instances generated by mst-explode-construct. The pairs of settings for the join length bias and packing factor were the same as for algorithm `mst-explode-construct`.

The excess over the approximate Held Harp bounds are quite high, often 5% or more. We suspect the iterative ascent method is partly to blame. However, given the same standard to measure against, the two variations of the Lin-Kernighan heuristic produced similar quality answers. The differences in quality diminish with more iterations executed, eventually settling to less than 0.1% in most cases.

The running times are all much longer than for the seed instances. No consistent pattern emerges when comparing the results from different seed instances: dif-

Percentage excess

Input to mst-dangle-construct	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
p1.dsjr.1000	14.86	14.88	0.02	14.34	14.34	0.00	14.30	14.30	0.00
p1.uni.1000	13.47	13.09	-0.39	9.06	9.07	0.01	8.95	8.98	0.03
p1.pr1002	7.55	7.53	-0.03	6.03	6.44	0.41	4.88	4.88	0.00
p1.dsj1000	8.02	7.29	-0.73	5.00	4.96	-0.04	4.94	4.94	0.00
p1.corners.1000	11.27	11.56	0.29	8.32	8.11	-0.21	7.59	7.51	-0.08
p10.dsjr.1000	1.76	1.81	0.05	1.31	1.42	0.10	1.26	1.26	0.00
p10.uni.1000	9.49	9.60	0.11	7.96	7.92	-0.04	7.39	7.44	0.05
p10.pr1002	6.71	6.88	0.17	5.48	5.60	0.12	5.32	5.42	0.10
p10.dsj1000	5.63	4.87	-0.77	4.56	3.92	-0.65	3.87	3.77	-0.10
p10.corners.1000	8.01	7.04	-0.96	5.25	5.15	-0.10	4.80	4.80	0.00
p100.dsjr.1000	1.06	1.04	-0.02	0.86	0.75	-0.11	0.66	0.66	0.00
p100.uni.1000	7.21	7.37	0.16	6.47	6.68	0.21	6.23	6.02	-0.21
p100.pr1002	8.32	7.01	-1.31	6.17	6.00	-0.18	5.72	5.72	0.00
p100.dsj1000	3.72	4.03	0.31	3.38	3.35	-0.03	3.20	3.20	0.00
p100.corners.1000	6.18	6.79	0.61	4.38	4.10	-0.28	4.00	3.92	-0.08

Table 9.9: Quality of output produced by Lin-Kernighan for the TSP on `mst-dangle-construct` instances. Packing factors of 1, 10, and 100 were used.

Running time

Input to mst-dangle-construct	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
p1.dsjr.1000	21.62	22.54	0.96	387.86	414.64	0.94	3404.25	3648.75	0.93
p1.pr1002	34.02	38.65	0.88	375.81	409.51	0.92	4728.94	4961.03	0.95
p1.uni.1000	45.96	52.36	0.88	927.20	929.21	1.00	8129.25	8243.40	0.99
p1.dsj1000	43.91	35.00	1.25	565.73	533.51	1.06	5607.82	5985.10	0.94
p1.corners.1000	42.11	38.63	1.09	906.45	719.81	1.26	8593.83	6723.45	1.28
p10.dsjr.1000	17.30	17.72	0.98	532.54	652.35	0.82	4862.62	4911.69	0.99
p10.uni.1000	26.89	30.27	0.89	294.43	305.30	0.96	3084.82	2860.78	1.08
p10.pr1002	44.51	46.35	0.96	601.23	633.30	0.95	5943.72	4897.94	1.21
p10.dsj1000	42.57	41.13	1.04	674.64	552.68	1.22	4845.02	5757.48	0.84
p10.corners.1000	48.34	43.07	1.12	560.33	430.10	1.30	6985.12	3867.87	1.81
p100.dsjr.1000	15.36	15.65	0.98	388.90	437.46	0.89	4885.44	5056.41	0.97
p100.uni.1000	26.34	30.96	0.85	342.05	334.56	1.02	3211.03	3735.88	0.86
p100.dsj1000	40.39	43.12	0.94	421.31	468.34	0.90	3789.10	3816.22	0.99
p100.pr1002	32.80	35.50	0.92	420.14	452.12	0.93	4199.52	4514.23	0.93
p100.corners.1000	63.40	39.05	1.62	659.33	401.33	1.64	7303.75	3971.49	1.84

Table 9.10: Time taken on `mst-dangle-construct` instances by the Lin-Kernighan heuristic for the TSP. Packing factors of 1, 10, and 100 were used.

ferent distributions take turns being the most difficult for the heuristics. Algorithm `mst-dangle-construct` therefore is unsuitable for preserving or predicting the behaviour of Lin-Kernighan on classes of similarly-structured seed instances. This is consistent with our visual intuition that the instances generated by mst-dangle-construct "don't look right."

When viewed as instances in their own right, we see that cluster compensation slows down the heuristic as often as it speeds it up. The slowdowns are usually by less than 12%. The speedups are usually by less than 30%.

9.2.5 Cluster noise

Tables 9.11 and 9.12 show the results for instances generated by algorithm cluster-noise.

Percentage excess

Input to cluster-noise	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
dsjr.1000	1.00	0.99	-0.01	0.94	0.91	-0.03	0.74	0.77	0.03
uni.1000	0.21	0.38	0.17	0.12	0.21	0.10	0.09	0.12	0.04
pr1002	0.43	0.71	0.27	0.30	0.45	0.15	0.18	0.29	0.11
dsj1000	1.02	1.23	0.21	0.83	1.07	0.24	0.74	0.88	0.14
corners.1000	1.75	1.91	0.16	1.63	1.73	0.10	1.57	1.61	0.04

Table 9.11: Quality of output produced by Lin-Kernighan for the TSP on cluster-noise instances.

The percent excess over approximate Held-Karp is always less than 2%. Lin-Kernighan with cluster compensation produces slightly worse tours than the base heuristic. The differences between tours produced by the two variants diminish with more iterations, ending up less than 0.15% apart after n iterations. These results are consistent with the results on the seed instances.

The story told by the running times is quite dramatic. Lin-Kernighan with cluster compensation runs very quickly on these instances, always faster than it does on the seed instances themselves. However, base Lin-Kernighan runs slower (up to 5 times) on instances generated from `uni.820.1000`, `pr1002` than it does on the seed instances

Running time

Input to cluster-noise	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
dsjr.1000	190.75	1.59	119.97	905.80	4.61	196.49	6689.26	26.96	248.12
uni.1000	7.25	1.42	5.11	46.35	3.15	14.71	348.24	15.38	22.64
pr1002	8.91	1.35	6.57	46.17	2.28	20.30	329.08	9.67	34.03
dsj1000	30.85	1.42	21.65	171.99	3.16	54.43	1282.63	15.97	80.31
corners.1000	27.88	1.40	19.91	290.54	3.28	88.58	2610.29	18.52	140.91

Table 9.12: Time taken on cluster-noise instances by the Lin-Kernighan heuristic for the TSP.

themselves. It usually runs slightly faster on the instance generated from `dsj1000` and `corners.1000` than it does on `dsj1000` and `corners.1000` themselves. The big surprise is that base Lin-Kernighan runs much slower on the instance produced from `dsjr.45.1000` than it does on `dsjr.45.1000` itself. Putting together the results for both variants of the heuristic —LK the same or slower and LKcc faster— we get some spectacular results: cluster compensation makes Lin-Kernighan between 20 and 250 times faster.

But these results should not be so surprising, and in fact confirms our intuition about the behaviour of the heuristic. Algorithm cluster-noise is designed to throw away all structure *except* the cluster separation pattern itself. So, according to our intuition, base Lin-Kernighan is faced with all the "hard parts" of the instance (the cluster separation structure) while Lin-Kernighan with cluster compensation just ignores them.

9.2.6 Cluster discount

Tables 9.13 and 9.14 summarize the results for the cluster-discount instances. After one iteration, the percentage excess over Held-Karp is in the 4% range, significantly higher than for the seed instances. By n iterations the quality of the tours is more in line with those produced on the seed instances, usually less than 2% above Held-Karp. The differences between the two variants of the heuristic are larger for these instances as compared with the seed instances, with Lin-Kernighan with cluster compensation usually

Percentage excess

Input to cluster-discount	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
dsjr.1000	4.34	5.25	0.91	2.75	2.74	-0.01	1.50	1.65	0.15
uni.1000	3.57	3.94	0.36	2.39	2.68	0.29	1.67	1.70	0.03
pr1002	4.32	4.17	-0.15	2.83	2.88	0.04	2.08	2.13	0.05
dsj1000	3.47	4.36	0.89	2.19	2.31	0.11	1.06	1.61	0.54
corners.1000	3.46	3.79	0.33	2.20	2.10	-0.10	1.39	1.50	0.11

Table 9.13: Quality of output produced by Lin-Kernighan for the TSP on cluster-discount instances.

Running time

Input to cluster-discount	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
dsjr.1000	3.04	2.56	1.19	25.64	19.73	1.30	190.46	139.06	1.37
uni.1000	1.45	1.39	1.04	4.54	3.00	1.51	27.13	16.28	1.67
pr1002	1.39	1.40	0.99	3.36	3.36	1.00	20.09	15.68	1.28
dsj1000	2.27	2.98	0.76	14.33	11.26	1.27	168.82	61.06	2.76
corners.1000	1.44	1.39	1.04	5.79	3.83	1.51	43.99	23.25	1.89

Table 9.14: Time taken on cluster-discount instances by the Lin-Kernighan heuristic for the TSP.

lagging by less than 0.3% after $n/10$ or more iterations.

For the most part, running times on cluster-discount instances are shorter than on the corresponding seed instances. For Lin-Kernighan with cluster compensation, the speedups are by factors in the 2–5 range at $n/10$ iterations and more. Only the instances generated from `dsjr.45.1000` are exceptions, with running times for $n/10$ and n iteration runs being roughly the same as for `dsjr.45.1000` itself. For base Lin-Kernighan the speedups are more dramatic, with some n iteration runs being nearly 100 times faster than those on the seed instances.

It is interesting to compare the results for the most sharply clustered seed instance `corners.827.1000`. The n iteration base Lin-Kernighan runs on the instances produced by cluster-discount from `corners.827.1000` take 22 seconds on average. This is even shorter than the runs of either Lin-Kernighan variant on *any* of the seed instances, including the highly uniform `uni.820.1000`.

Even with such time reductions for both variants, cluster compensation makes the heuristic run faster on most runs. By n iterations, the speedup factor is in the 1.3 to 2.75 range. The worst slowdown is by 24% after one iteration on the instances produced from `dsj1000`, but by n iterations those instances display the greatest speedup for cluster compensation, a 2.76 times reduction in running time.

Our intuition is confirmed by these running time trends. That is, the large gaps between clusters is a primary cause of long running times for the Lin-Kernighan heuristic. When those gaps are factored out, being largely removed by cluster-discount, even the base Lin-Kernighan heuristic runs much faster. The bonus is that Lin-Kernighan with cluster compensation retains a running time advantage on those instances.

9.2.7 Cluster infill

Tables 9.15 and 9.16 show the results for instances generated by algorithm infill.

The quality of output produced on these instances is similar to the quality produced on the seed instances. By n iterations, both variants produce answers within 2% of the Held-Karp bound. Also, the quality of answers does not vary much between the two variants of Lin-Kernighan. The excess above Held-Karp usually differs by less than 0.2%.

When considering running times, recall that the instances examined here each have 10% more vertices than their corresponding seed instances. Even with the larger instance sizes, base Lin-Kernighan runs much *faster* on the instances generated by infill

Percentage excess

Input to infill	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
dsjr.1000	1.95	1.94	-0.01	0.96	1.27	0.31	0.48	0.75	0.27
uni.1000	2.57	2.77	0.20	1.67	1.99	0.32	1.28	1.26	-0.02
pr1002	3.17	3.36	0.19	2.39	2.61	0.22	1.45	1.64	0.19
dsj1000	3.01	2.94	-0.07	1.55	1.65	0.10	1.21	1.08	-0.13
corners.1000	2.46	2.27	-0.19	1.41	1.42	0.01	1.15	1.11	-0.04

Table 9.15: Quality of output produced by Lin-Kernighan for the TSP on infill instances.

Running time

Input to infill	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
dsjr.1000	2.17	2.03	1.07	12.50	8.88	1.41	93.44	55.21	1.69
uni.1000	1.29	1.01	1.28	7.88	4.67	1.69	48.84	32.44	1.51
pr1002	4.64	3.41	1.36	23.13	13.32	1.74	145.25	110.86	1.31
dsj1000	16.05	14.36	1.12	156.74	109.81	1.43	1049.30	839.17	1.25
corners.1000	16.35	5.79	2.82	220.29	52.55	4.19	1885.96	391.54	4.82

Table 9.16: Time taken on infill instances by the Lin-Kernighan heuristic for the TSP.

from the sharply clustered `dsj1000` and `corners.827.1000`. This confirms our suspicion that filling in the largest gaps between clusters should speed up base Lin-Kernighan. Both variants of Lin-Kernighan run much faster on the instance generated by infill from `dsjr.45.1000` than they did on the `dsjr.45.1000` itself. Interestingly, Lin-Kernighan with cluster compensation often runs much longer, up to five times longer, for the geometric infill instances than on their corresponding seed instances. Even with such a handicap, cluster compensation still provides a running time advantage on the instances generated by infill from any given seed instance. This too confirms our intuition that the time advantage given by cluster compensation should be diminished on instances generated by infill.

9.3 Results for minimum weight perfect matching

This section presents the results for minimum weight perfect matching experiments on the instances produced by the seven generation algorithms. Tables 9.17 and 9.18 collect the results on the five seed instances, and on an extra instance `corners.896.996`.

Percentage excess

Seed instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
dsjr.45.1000	4.01	3.21	-0.80	3.40	2.76	-0.64	2.40	1.73	-0.67
uni.820.1000	1.23	1.70	0.47	1.22	1.67	0.45	1.12	1.55	0.43
pr1002	1.35	1.53	0.18	1.31	1.43	0.12	1.26	1.00	-0.26
dsj1000	0.75	0.61	-0.14	0.63	0.53	-0.10	0.53	0.30	-0.23
corners.827.1000	0.68	0.84	0.16	0.62	0.83	0.21	0.57	0.82	0.25
corners.896.996	0.34	0.62	0.28	0.32	0.52	0.20	0.21	0.42	0.21

Table 9.17: Quality of output produced by Lin-Kernighan for weighted perfect matching on the seed instances, and on `corners.896.996`.

Running time

Seed instance	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
dsjr.45.1000	2.23	2.32	0.96	2.56	2.55	1.00	5.21	4.49	1.16
uni.820.1000	0.62	0.51	1.22	0.85	0.66	1.29	3.00	1.94	1.55
pr1002	0.93	0.62	1.50	1.37	0.78	1.76	5.12	2.24	2.29
dsj1000	2.58	0.66	3.91	4.76	0.86	5.53	23.37	3.08	7.59
corners.827.1000	0.58	0.51	1.14	0.82	0.65	1.26	2.85	1.87	1.52
corners.896.996	4.84	0.51	9.49	12.57	0.69	18.22	83.03	2.57	32.31

Table 9.18: Time taken by the Lin-Kernighan heuristic for weighted perfect matching. on the seed instances and on instance `corners.896.996`.

9.3.1 An anomaly with `corners`

Tables 9.17 and 9.18 display an anomaly with the `corners` distribution. The rows are ordered in the same way as they were in the previous section about the TSP. They are ordered according to increasing speedup provided by cluster compensation *for the TSP*. For weighted perfect matching the pattern is the same —increasing speedup provided by cluster compensation— until broken by `corners.827.1000` where the speedup is only 1.52 even after n iterations. It is the most sharply clustered instance and therefore should have the greatest speedup for cluster compensation. However, the speedup for the matching heuristic on the `corners` class of inputs is highly dependent upon the number of vertices in the input.

The tables also show the average results from five runs on a 996 vertex instance from the same `corners` distribution. The running times for base Lin-Kernighan are much longer, while Lin-Kernighan with cluster compensation runs are similar to the `corners` instance with 1000 vertices. The speedup on the 996 vertex instance is quite high, and more in keeping with what we might expect for so sharply clustered an instance.

What is going on? Recall that the `corners` distribution produces instances with four widely separated clumps, each containing $n/4$ vertices. A 1000 vertex instance therefore has four clusters of 250 vertices each. It is overwhelmingly likely that optimal and near-optimal perfect matchings will not contain any edges with endpoints in two distinct super-clusters. The same is likely true for even the greedy matchings with which our implementation begins. But a 996 vertex instance will have 249 vertices in each of the clumps, forcing at least two edges in any perfect matching on that instance to span the gaps between the super-clusters. Always having a long edge in a perfect matching forces the base Lin-Kernighan heuristic for matching to spend a lot of time on long fruitless searches, just as base Lin-Kernighan for the TSP does on `corners` instances. The running times on `corners.827.1000` are low in comparison mainly because the number of cities in the super-clusters has even parity.

We should expect *both* kinds of behaviours —very long and very short runs for base Lin-Kernighan— on the instances generated from `corners.827.1000`. The following sections show that the heuristic in fact does exhibit both extreme behaviours on those instances. The generating algorithms are thus performing the important service of highlighting the sensitivity of the Lin-Kernighan heuristic to seemingly minor details of the input.

9.3.2 Jitter

Tables 9.19 and 9.20 show the results for the instances produced by the jitter algorithm.

Percentage excess

Input to jitter	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
uni.1000	1.10	1.01	-0.09	1.10	1.00	-0.10	1.09	0.93	-0.17
pr1002	1.09	1.47	0.38	1.09	1.47	0.38	0.93	0.83	-0.10
dsj1000	1.37	1.28	-0.09	1.31	1.06	-0.25	0.78	0.70	-0.08
corners.1000	0.46	0.82	0.36	0.41	0.82	0.41	0.41	0.52	0.11

Table 9.19: Quality of output produced by Lin-Kernighan for weighted perfect matching on jitter instances.

The results here are to be expected, just as they were for the TSP. Both variants produce very good matchings, often within 1% of optimal. Cluster compensation often helps the heuristic to find better matchings than it would otherwise.

The running times on the jitter instances are similar to the running times on the seed

Running time

Input to jitter	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
uni.1000	0.59	0.49	1.19	0.82	0.62	1.31	2.95	1.80	1.64
pr1002	1.01	0.75	1.35	1.45	0.95	1.52	5.39	2.52	2.13
dsj1000	3.32	1.17	2.83	3.99	1.38	2.90	27.91	3.84	7.27
corners.1000	0.55	0.53	1.04	0.74	0.65	1.15	2.62	1.84	1.42

Table 9.20: Time taken on jitter instances by the Lin-Kernighan heuristic for weighted perfect matching.

instances, with cluster compensation providing a speedup of between roughly 1.5 and 7 after n iterations.

Again, jitter's conservative nature leads to results on generated instances differing little from the results on the seed instances.

9.3.3 MST shake

Tables 9.21 and 9.22 show the results for the instances produced by the mst-shake algorithm.

Percentage excess

Input to mst-shake	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
uni.1000	0.81	1.19	0.38	0.81	1.16	0.34	0.75	1.11	0.36
pr1002	1.22	0.77	-0.45	0.97	0.75	-0.22	0.81	0.71	-0.10
dsj1000	1.36	1.11	-0.25	1.35	0.89	-0.46	1.08	0.42	-0.67
corners.1000	0.92	0.81	-0.11	0.76	0.81	0.05	0.63	0.74	0.11

Table 9.21: Quality of output produced by Lin-Kernighan for weighted perfect matching on mst-shake instances.

Both variants produce good matchings, with the percentage excess above optimal

Running time

Input to mst-shake	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
uni.1000	0.60	0.52	1.17	0.81	0.66	1.24	2.72	1.92	1.41
pr1002	0.82	0.53	1.57	1.21	0.66	1.84	4.03	1.87	2.16
dsj1000	2.95	0.80	3.69	6.04	1.30	4.63	27.51	5.47	5.03
corners.1000	0.62	0.54	1.16	0.85	0.67	1.27	2.77	1.85	1.50

Table 9.22: Time taken on mst-shake instances by the Lin-Kernighan heuristic for weighted perfect matching.

always being less than 1.5%. Neither variant always produces better matchings, with the advantage being data-dependent though not apparently correlated to the severity of clustering. These results are consistent with the results on the seed instances.

The running times on these instances are also similar to those for the seed instances, staying within roughly 25% of the original timings. The speedup provided by cluster compensation is also similar, at n iterations they range from 1.4 to 5, compared with 1.5 to 7.5 on the seed instances.

9.3.4 MST explode and construct

Tables 9.23 and 9.24 show the weighted perfect matching results for the instances produced by mst-explode-construct. Again, join length bias -1 (random choice) was paired with packing factor 1, join length bias 0.75 with packing factor 10, and join length bias 1 with packing factor 100.

By n iterations, both variants of Lin-Kernighan produce matchings with weight within 2% of optimal. Each one takes turn producing better matchings than the other, with significant differences on both sides. For example, for three distributions Lin-Kernighan produces a matching after n iterations more than 0.5% closer to optimal than produced by Lin-Kernighan with cluster compensation. Going the other way, there is one case where cluster compensation helps the heuristic find matchings 0.14% above optimal instead of 1.76% above optimal.

The only instances that prove too difficult to get good answers in few iterations are the

Percentage excess

Input to mst-explode-construct	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
j-1.p1.dsjr.1000	2.94	1.58	-1.36	2.63	0.80	-1.83	1.76	0.14	-1.63
j-1.p1.uni.1000	0.15	0.30	0.14	0.15	0.30	0.14	0.15	0.30	0.14
j-1.p1.pr1002	0.48	0.51	0.03	0.48	0.51	0.03	0.48	0.51	0.03
j-1.p1.dsj1000	1.19	1.53	0.34	1.19	1.53	0.34	0.10	1.03	0.93
j-1.p1.corners.1000	4.94	4.92	-0.02	3.91	3.42	-0.50	0.93	0.64	-0.30
j0.75.p10.dsjr.1000	1.83	2.19	0.35	0.95	1.29	0.34	0.73	1.08	0.35
j0.75.p10.uni.1000	0.22	0.37	0.15	0.22	0.37	0.15	0.22	0.33	0.11
j0.75.p10.pr1002	2.58	0.70	-1.87	2.58	0.70	-1.87	0.78	0.68	-0.09
j0.75.p10.dsj1000	0.56	0.69	0.12	0.56	0.69	0.12	0.55	0.68	0.12
j0.75.p10.corners.1000	1.20	1.25	0.05	1.18	0.99	-0.19	0.82	0.88	0.06
j1.p100.dsjr.1000	1.30	1.84	0.54	1.14	1.67	0.53	0.94	1.56	0.62
j1.p100.uni.1000	1.42	1.88	0.46	1.42	1.84	0.43	0.82	0.88	0.05
j1.p100.pr1002	1.53	1.41	-0.12	1.53	1.35	-0.18	0.35	0.92	0.57
j1.p100.dsj1000	1.74	2.67	0.93	1.73	2.13	0.40	1.37	1.58	0.21
j1.p100.corners.1000	0.80	1.01	0.21	0.68	0.84	0.16	0.60	0.58	-0.02

Table 9.23: Quality of output produced by Lin-Kernighan for weighted perfect matching on `mst-explode-construct` instances. Join length biases are −1 (random choice), 0.75, and 1, and corresponding packing factors are 1, 10, and 100.

Running time

Input to mst-explode-construct	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
j-1.p1.dsjr.1000	3.56	2.82	1.26	6.93	4.84	1.43	28.66	12.29	2.33
j-1.p1.uni.1000	0.69	0.69	1.01	0.99	0.91	1.09	3.60	3.05	1.18
j-1.p1.pr1002	0.96	0.93	1.04	1.40	1.22	1.15	6.75	4.98	1.36
j-1.p1.dsj1000	2.61	1.43	1.83	3.64	1.79	2.03	13.24	7.90	1.68
j-1.p1.corners.1000	3.16	2.01	1.57	7.96	5.65	1.41	35.15	14.79	2.38
j0.75.p10.dsjr.1000	2.44	1.36	1.79	3.50	1.69	2.07	11.59	4.43	2.61
j0.75.p10.uni.1000	0.81	0.65	1.24	1.12	0.85	1.32	4.50	2.88	1.56
j0.75.p10.pr1002	1.59	0.86	1.84	2.62	1.07	2.44	8.93	3.12	2.87
j0.75.p10.dsj1000	2.11	0.85	2.47	3.34	1.22	2.74	13.14	3.85	3.41
j0.75.p10.corners.1000	4.72	0.98	4.79	10.12	1.82	5.56	51.97	7.27	7.15
j1.p100.dsjr.1000	2.34	1.14	2.06	3.92	1.67	2.36	15.37	5.51	2.79
j1.p100.uni.1000	1.06	0.96	1.10	1.41	1.25	1.13	6.30	4.95	1.28
j1.p100.pr1002	2.19	1.05	2.08	3.85	1.59	2.42	11.74	4.30	2.73
j1.p100.dsj1000	3.04	1.59	1.91	5.36	2.78	1.93	21.95	8.67	2.53
j1.p100.corners.1000	2.73	1.00	2.73	6.17	1.60	3.85	41.01	8.06	5.08

Table 9.24: Time taken on `mst-explode-construct` instances by the Lin-Kernighan heuristic for weighted perfect matching. Join length biases are -1 (random choice), 0.75, and 1, and corresponding packing factors are 1, 10, and 100.

ones produced from seed `corners.827.1000` using join length bias -1 and packing factor 1. After a single iteration, both variants produce matchings almost 5% above optimal. The excess drops to between 3 and 4% after $n/10$ iterations. But after n iterations the excess is in the normal range of less than 1% above optimal.

There are three main trends in the absolute running times given in Table 9.24. First, the running times for the instances generated from the random distance matrix `dsjr.45.1000` (29, 12, and 15 seconds) are significantly higher after n iterations than they were after n iterations on `djsr.45.1000` itself (5 seconds). Second, n iteration running times are significantly higher for Lin-Kernighan with cluster compensation, often 3 times higher. Base Lin-Kernighan often runs longer, but not on instances generated from `dsj1000`. Third, the instances generated from `corners.827.1000` force quite high running times. Running times for base Lin-Kernighan on those instances are significantly higher than for other instances and are more in line with what one might have predicted by looking at its running times on `corners.896.996`. It appears that on the instances generated from `corners.827.1000`, we have not only the wide separation of the clusters inherent in the construction but also those clusters have sizes with odd parity.

The ratios of running times still consistently favour cluster compensation. The smallest speedup is roughly 1.2, and the largest is more than 7. Nowhere do we see a repeat of the dramatic 40 times speedup that Table 9.18 shows on `corners.896.996`. However, without the instances generated from `dsjr.45.1000` the speedups increase with the sharpness of clustering, *i.e.*, with successive rows.

Overall, `mst-explode-construct` produces instances with weighted perfect matching heuristic results comfortably close to the results for its seed instances that are geometric in nature. It is most consistent in predicting the speedup offered by cluster compensation. Surprisingly, these trends do not appear to depend heavily on the join length bias nor on the packing factor used to generate the new instances.

9.3.5 MST dangle and construct

Tables 9.25 and 9.26 show the weighted perfect matching results for the instances produced by mst-dangle-construct. Packing factors 1, 10, and 100 were used. (Join length bias is not significant in algorithm mst-dangle-construct.)

Table 9.25 has several striking features. First, the quality of the matchings produced varies a great deal from one row instance to another. In particular, the instances gen-

Percentage excess

Input to mst-dangle-construct	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
p1.dsjr.1000	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
p1.uni.1000	0.70	0.61	-0.09	0.53	0.61	0.08	0.53	0.38	-0.15
p1.pr1002	0.73	0.35	-0.38	0.50	0.26	-0.24	0.02	0.02	0.00
p1.dsj1000	2.32	2.25	-0.06	2.32	2.25	-0.06	2.30	2.24	-0.06
p1.corners.1000	0.32	0.17	-0.15	0.32	0.17	-0.15	0.21	0.17	-0.05
p10.dsjr.1000	0.19	0.20	0.01	0.01	0.03	0.01	0.01	0.03	0.01
p10.uni.1000	2.17	1.94	-0.22	1.76	0.83	-0.92	0.65	0.71	0.06
p10.pr1002	0.20	0.17	-0.03	0.20	0.17	-0.03	0.15	0.17	0.01
p10.dsj1000	1.66	3.22	1.57	1.51	3.06	1.54	1.51	1.60	0.09
p10.corners.1000	0.09	0.12	0.03	0.09	0.12	0.03	0.03	0.12	0.09
p100.dsjr.1000	0.01	0.01	0.00	0.01	0.01	0.00	0.01	0.01	0.00
p100.uni.1000	2.75	1.09	-1.66	2.75	1.09	-1.66	0.96	0.46	-0.50
p100.pr1002	0.83	1.17	0.34	0.83	1.17	0.34	0.06	0.24	0.18
p100.dsj1000	5.25	3.36	-1.89	5.25	3.12	-2.13	5.15	1.31	-3.84
p100.corners.1000	3.23	3.23	0.00	3.22	3.23	0.01	3.22	1.68	-1.54

Table 9.25: Quality of output produced by Lin-Kernighan for weighted perfect matching on `mst-dangle-construct` instances. Packing factors of 1, 10, and 100 were used.

Running time

Input to mst-dangle-construct	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
p1.dsjr.1000	0.91	0.77	1.17	1.33	1.05	1.27	5.16	3.61	1.43
p1.uni.1000	0.98	0.83	1.19	1.49	1.26	1.18	5.55	4.36	1.27
p1.pr1002	1.04	0.79	1.32	1.50	0.97	1.53	4.38	2.52	1.73
p1.dsj1000	1.97	1.51	1.30	3.23	2.31	1.40	14.84	12.25	1.21
p1.corners.1000	0.78	0.76	1.04	1.12	0.97	1.16	4.76	3.11	1.53
p10.dsjr.1000	1.06	0.99	1.08	1.48	1.27	1.17	5.31	3.79	1.40
p10.uni.1000	1.26	1.00	1.25	1.90	1.96	0.97	8.43	6.95	1.21
p10.pr1002	0.76	0.66	1.16	1.03	0.83	1.23	3.79	2.56	1.48
p10.dsj1000	2.14	2.15	1.00	5.17	3.83	1.35	19.71	20.32	0.97
p10.corners.1000	0.66	0.60	1.09	0.93	0.79	1.18	3.50	2.58	1.36
p100.dsjr.1000	1.20	1.06	1.13	1.54	1.29	1.19	4.58	3.40	1.35
p100.uni.1000	1.92	1.30	1.47	2.93	2.00	1.47	12.34	5.93	2.08
p100.pr1002	0.96	1.41	0.68	1.52	3.44	0.44	4.49	6.29	0.71
p100.dsj1000	2.93	1.75	1.68	6.74	2.70	2.50	46.68	13.18	3.54
p100.corners.1000	2.25	1.36	1.65	3.75	1.98	1.90	17.86	5.77	3.09

Table 9.26: Time taken on `mst-dangle-construct` instances by the Lin-Kernighan heuristic for weighted perfect matching. Packing factors of 1, 10, and 100 were used.

erated with packing factor 1 from random distance matrix `dsjr.45.1000` are very easy to optimize: both variants of the heuristic find matchings 0.00% above optimal. The impressive "0.00" figure is not a result of rounding. Checking the output logs reveals that both Lin-Kernighan variants found perfect matchings with exactly optimal weight after the first iteration on all three runs on each of the two instances generated from `dsjr.45.1000`. (The initial randomized greedy matchings were not optimal in any case.) The instances generated from `dsjr.45.1000` with packing factors 10 and 100 are almost as easy, with percentage excesses over optimal falling to 0.03% or less by $n/10$ iterations.

Second, some instance distributions were quite difficult to optimize, especially those generated from `dsj1000`. For the instance generated with packing factor 100, base Lin-Kernighan fails to find a matching with weight less than 5% above optimal, even after n iterations. Lin-Kernighan with cluster compensation fares much better in that case, achieving an average excess of only 1.3%. A similar story holds for instances generated from `corners.827.1000` with packing factor 100: base Lin-Kernighan does no better than 3.2% above optimal while Lin-Kernighan with cluster compensation gets down to 1.7% above optimal. Still, both variants do equally "poorly", producing answers with excesses in the 2.25% range for instances generated from `dsj1000` with packing factor 1.

Running times are generally longer for the mst-dangle-construct instances than for the seeds from which they were generated. For both variants, the blowup factor is often 5 times or greater. The main statistic in common between Table 9.26 and Table 9.18 is that the longest running time for a seed instance is for base Lin-Kernighan on `dsj1000`, and the longest running time on generated data is for base Lin-Kernighan on instances generated from `dsj1000`. (We are not counting the running times for `corners.896.996` because we did not generate data from it.)

Cluster compensation usually reduces running times, but there is no consistent speedup trend across the rows. Certainly, the speedup trends across the rows could not be predicted from the trends displayed in Table 9.18 for the seed instances themselves.

Given the outcomes on the data it generates, algorithm mst-dangle-construct gives only a very broad indication of how the two Lin-Kernighan variants performed on the seed instances. It is less faithful to the original outcomes than algorithm mst-explode-construct.

9.3.6 Cluster and noise

Tables 9.27 and 9.28 show the weighted perfect matching results for the instances generated by algorithm `cluster-noise`.

Percentage excess

Input to cluster-noise	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
dsjr.1000	1.31	1.05	-0.26	1.21	0.98	-0.23	0.94	0.84	-0.09
uni.1000	0.14	0.44	0.30	0.10	0.39	0.29	0.04	0.18	0.14
pr1002	0.39	0.66	0.27	0.34	0.60	0.26	0.24	0.41	0.17
dsj1000†	0.33	0.70	0.37	0.23	0.54	0.32	0.14	0.28	0.14
corners.1000†	0.08	0.32	0.23	0.07	0.27	0.21	0.03	0.12	0.10

Table 9.27: Quality of output produced by Lin-Kernighan for weighted perfect matching on cluster-noise instances.

Of the ten instances generated by cluster-noise, three of them posed serious difficulty for the Blossom IV code. (Blossom IV was itself built upon the preliminary version of the Concorde software dated 1997/08/27, the latest available as of this writing.) For both instances generated from `dsj1000` and one of the two generated from `corners.827.1000`, the Blossom IV code failed to find a perfect matching at all, let alone an optimal perfect matching. In place of getting an optimal perfect matching from Blossom IV, for each instance we substituted the lightest perfect matching found during 10 n iteration runs of base Lin-Kernighan and 10 n-iteration runs of Lin-Kernighan with cluster compensation. We have therefore marked those rows with a dagger (†). Fortunately, the percentage excesses are not out of place in comparison with the other rows in the table, nor with the excesses for those seed instances using other generation heuristics.

Both variants of Lin-Kernighan find good matchings, always less than 1.5% on average above optimal after a single iteration, and always less than 1% above optimal on average after n iterations. Cluster compensation usually forced the heuristic to find slightly worse answers, but the differences in percentage excess after n iterations were small, less than 0.15% extra excess. These results are even more uniform than the percent excess results for the seed instances.

Running time

Input to cluster-noise	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
dsjr.1000	9.07	1.25	7.29	30.94	1.46	21.12	207.44	3.18	65.23
uni.1000	1.33	1.21	1.09	1.60	1.28	1.25	4.43	1.81	2.45
pr1002	1.73	1.23	1.41	2.33	1.28	1.82	8.04	1.77	4.53
dsj1000†	3.54	1.21	2.92	9.39	1.29	7.25	44.21	1.99	22.22
corners.1000†	1.35	1.17	1.15	1.84	1.25	1.48	5.42	1.84	2.95

Table 9.28: Time taken on cluster-noise instances by the Lin-Kernighan heuristic for weighted perfect matching.

The big surprise in Table 9.28 is the base Lin-Kernighan running time for the instances generated by cluster-noise from `dsjr.45.1000`. After n iterations, the time exceeds 200 seconds, by far the longest running time recorded in the perfect matching results in this chapter. (The closest runner-up is the 83 seconds recorded by n iteration base Lin-Kernighan on `corners.896.996`.)

Lin-Kernighan with cluster compensation runs longer on the instances generated from `dsjr.45.1000` than on `dsjr.45.1000` itself. Otherwise, the running times for LKcc are surprisingly uniform across all numbers of iterations. At n iterations the other four times range from 1.77 to 1.99 seconds.

In contrast, the trend in running times for base Lin-Kernighan on those four instance classes mimics the trend for base Lin-Kernighan on the seed instances. The main difference is that running times are significantly longer, by factors of between roughly 1.5 and 2.

Combining the trends for both kinds of heuristics, the speedup due to cluster compensation is more exaggerated on the generated data in comparison to the results on the seed instances. The speedup for the generated data ranges up to 22 and 65 times, whereas for the seed data it was limited to less than 8 times.

Algorithm cluster-noise strips away all significant structure *except* the cluster structure, so we should not be surprised by these results for absolute running times and for the speedups provided by cluster compensation. These results help confirm our intuition that sharply clustered structure forces long running times, and that cluster compensation is an appropriate remedy.

9.3.7 Cluster infill

Tables 9.29 and 9.30 show the weighted perfect matching results for the instances generated by algorithm infill.

Percentage excess

Input to infill	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
dsjr.1000	2.64	2.07	-0.57	2.20	1.78	-0.42	1.55	0.91	-0.64
uni.1000	1.12	1.55	0.43	1.05	1.54	0.49	1.05	1.49	0.44
pr1002	1.88	1.44	-0.44	1.88	1.41	-0.47	0.91	1.33	0.42
dsj1000	0.84	0.67	-0.17	0.81	0.67	-0.14	0.58	0.51	-0.07
corners.1000	1.97	3.51	1.54	1.93	1.10	-0.83	1.59	0.71	-0.88

Table 9.29: Quality of output produced by Lin-Kernighan for weighted perfect matching on infill instances.

Running time

Input to infill	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc	LK	LKcc	LK/ LKcc
dsjr.1000	1.63	1.70	0.96	1.91	1.95	0.98	4.27	3.86	1.11
uni.1000	0.67	0.59	1.14	0.93	0.75	1.24	3.13	2.24	1.40
pr1002	1.18	0.72	1.64	1.59	0.93	1.71	5.72	2.85	2.01
dsj1000	0.86	0.58	1.48	1.18	0.71	1.66	4.59	2.10	2.19
corners.1000	4.16	1.44	2.89	8.78	2.57	3.42	36.95	5.42	6.82

Table 9.30: Time taken on infill instances by the Lin-Kernighan heuristic for weighted perfect matching.

Both variants produce very good perfect matchings for these instances, with excesses never exceeding 1.6% above optimal after n iterations. Even after only a single iteration, the excesses never exceed 2.7%. Neither variant of Lin-Kernighan produces consistently better answers than the other.

Running times for instances generated from `dsjr.45.1000`, `uni.820.1000` and `pr1002` are quite similar to the running times on those seed instances themselves. The biggest differences come for the instances generated from `dsj1000` and `corners.827.1000`. Base Lin-Kernighan is 5 times faster (4.5 seconds) on the instances generated from `dsj1000` than on `dsj1000` itself (23 seconds). Base Lin-Kernighan on the instances generated from `corners.827.1000` after n iterations takes 37 seconds, somewhere between its running time on `corners.827.1000` and `corners.896.996`.

Cluster compensation consistently reduces running times, with speedup factors ranging from just over 1 to just under 7 times. The speedup factors are generally smaller here than they were on the seed instances themselves. This is expected because algorithm `infill` is designed to corrupt the cluster structure by bisecting the longest edges. Our intuition about cluster structure and the Lin-Kernighan heuristic and cluster compensation is therefore further confirmed by these results.

9.3.8 Cluster discount

Tables 9.31 and 9.32 show the weighted perfect matching results for the instances generated by algorithm `cluster-discount`.

Percentage excess

Input to cluster-discount	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK	LK	LKcc	LKcc −LK
dsjr.1000	6.64	5.20	-1.44	5.35	4.39	-0.96	3.12	2.78	-0.34
uni.1000	1.56	1.81	0.24	1.52	1.75	0.23	1.38	1.72	0.34
pr1002	2.91	2.88	-0.04	2.69	2.74	0.05	2.22	2.40	0.17
dsj1000	2.02	2.49	0.47	1.73	2.46	0.73	1.66	1.89	0.23
corners.1000	1.80	2.11	0.30	1.77	1.92	0.15	1.55	1.73	0.19

Table 9.31: Quality of output produced by Lin-Kernighan for weighted perfect matching on cluster-discount instances.

Overall, both variants of Lin-Kernighan have difficulty finding very good matchings.

Running time

Input to cluster-discount	1 iteration			$n/10$ iterations			n iterations		
	LK	LKcc	LK/LKcc	LK	LKcc	LK/LKcc	LK	LKcc	LK/LKcc
dsjr.1000	1.25	1.40	0.89	1.42	1.58	0.89	2.49	3.00	0.83
uni.1000	1.14	1.23	0.93	1.19	1.29	0.93	1.68	1.82	0.92
pr1002	1.08	1.13	0.95	1.15	1.19	0.96	1.65	1.71	0.97
dsj1000	1.25	1.38	0.91	1.33	1.45	0.91	2.08	2.33	0.89
corners.1000	1.09	1.17	0.93	1.16	1.25	0.93	1.75	1.83	0.95

Table 9.32: Time taken on cluster-discount instances by the Lin-Kernighan heuristic for weighted perfect matching.

The percentage excesses are higher on these instances than they were on the corresponding seed instances. For example, base Lin-Kernighan fails to find matchings closer than 3% above optimal after n iterations for the instances generated from cluster-discount. Lin-Kernighan with cluster compensation finds better matchings on those instances, but base Lin-Kernighan finds marginally better matchings for the other instance distributions.

The running times are both uniform and short. Cluster compensation always slows down the heuristic, by between 3 and 17 percent. However, the differences in absolute running times are very small, often by less than 0.25 seconds. That is roughly the time taken by the extra preprocessing steps, such as building a minimum spanning tree, required by efficient cluster compensation. As described in Section 7.7.4, the disadvantage is compounded by the fact that the optimization phase for weighted perfect matching runs so quickly and that the general matrix form of the input forces us to use a slower minimum spanning tree algorithm such as Prim's $O(n^2)$ time method.

We should not be surprised by these running time results since cluster discount is supposed to *remove* the features of the instance that slow down base Lin-Kernighan. So Lin-Kernighan with cluster compensation is left with no structure to take advantage of, and yet must try to make up for its extra preprocessing time.

9.4 Summary

The experiments reported in this chapter support our experimental findings from previous chapters. In general Lin-Kernighan usually finds very good tours and perfect matchings. The answers are routinely within 2% of lower bounds for those instances.

Quality-wise, the most difficult classes of instances were those generated by algorithm mst-dangle-construct. For the weighted perfect matching problem, a couple of the seed instances produced data that base Lin-Kernighan failed to find an answer with an excess of less than 3% above optimal, and in one case less than 5% above optimal.

Furthermore, sharply clustered data usually forces base Lin-Kernighan to take significantly more time. Cluster compensation usually helps to reduce the performance penalty incurred on clustered data.

The algorithms designed to preserve cluster structure —jitter, mst-shake, mst-explode-construct, mst-dangle-construct, and cluster-noise— generally maintained the running time advantage given by cluster compensation, and in particular the distinctions made between the speedups given the varying severity of clustering in the inputs. Algorithm mst-dangle-construct had the least success in doing so, especially for the TSP. Algorithms jitter and mst-shake were the most successful at producing instances that make the Lin-Kernighan heuristic behave similarly to the way it behaves on the seed instances.

The algorithms designed to damage the gross cluster structure —infill and cluster-discount— made running times more uniform and diminished the running time advantage given by using cluster compensation. For weighted perfect matching, cluster-discount produced instances that always made Lin-Kernighan with cluster compensation run longer than base Lin-Kernighan. It did this even though the reverse was true for the seed instances. Even so, the slowdowns were not excessive, never more than 17%.

Our intuition about the causes of long running times was confirmed by these experiments with generated data. Large gaps between clusters are a primary cause of long running times in the base Lin-Kernighan heuristic, and efficient cluster compensation goes a long way toward reducing the running time penalty without significantly damaging the quality of output.

Chapter 10

Conclusion

This thesis motivates and describes efficient cluster compensation, a technique designed to reduce running times of the Lin-Kernighan heuristic, especially on clustered inputs, without significantly degrading the quality of the output.

Cluster compensation adjusts the cumulative gain criterion and the greedy selection criterion. These two criteria serve as the Lin-Kernighan heuristic's primary means of pruning the search space and prioritizing avenues within the available search space, respectively. The adjustment consists of subtracting the cluster distance between two points from the standard cumulative gain function used by the heuristic. We have shown online cluster distance queries can be answered in constant time, with only modest preprocessing cost—effectively just the cost of computing a minimum spanning tree for the underlying instance. The number of extra words of storage required is only linear in the number of vertices of the graph.

Cluster compensation applies to a wide range of Lin-Kernighan heuristics. We have focused on two particular problems, the Traveling Salesman Problem and the minimum weight perfect matching problem. We report the results of experiments comparing our implementation of the Lin-Kernighan heuristic for both problems, both with and without efficient cluster compensation. The test data includes twelve of the standard TSPLIB instances (the same used by Johnson and McGeoch [46]), samples from eleven distributions Bentley reported as either standard or which stress TSP construction heuristics in some way [17], and both uniform geometric and uniform distance matrix classes.

Additionally, we introduce seven instance generating algorithms to produce new test data. Each generator takes a seed instance as input and generates a new instance from it, trying either to preserve or to destroy aspects of the cluster structure present in the seed instance. The algorithms vary according to their goals, how radically they transform the input, and the assumptions they place on the input. For example, algorithms jitter and mst-shake both require their inputs to be geometric, and jitter is primarily intended to test the sensitivity of the Lin-Kernighan heuristic to small perturbations in the coordinates of the input vertices. Algorithms mst-explode-construct and mst-dangle-construct take any complete undirected weighted graph as input and produce a two-dimensional geometric instance as output. They are radical transformations, and are intended to preserve the most basic aspects of cluster structure. Algorithm cluster-noise preserves *only* the gross cluster structure, together with some noise. Algorithms cluster-discount and infill destroy the main aspects of the cluster structure, by subtracting the cluster distance function from the base distance function or by inserting new points into the gaps between clusters, respectively. These last three algorithms help us test our intuition about the causes of long running times for the Lin-Kernighan heuristic, and whether cluster compensation is an effective means for reducing that performance penalty.

The experiments show that cluster compensation usually reduces running times on a given instance, often dramatically. A very rough rule is that the reductions are greater on more sharply clustered instances. For example, on the sharply clustered 1000 vertex instances from the `corners` distribution, the TSP heuristic has an average speedup of 29 times after 100 iterations. An average speedup of 1.8 is observed for 100 iteration runs on the `uni.1000` distribution, random uniform geometric instances with 1000 vertices. For n iteration TSP runs, running times are usually between 1.3 and 3 times shorter.

Overall running times for the weighted perfect matching heuristic are much shorter than for the TSP heuristic. Cluster compensation usually reduces running times for the matching heuristic as well. But the margins are often thinner, and there are more instances for which cluster compensation increases running times.

In the few cases for which cluster compensation increases running times, the slowdown is small, usually less than 15%. In by far the worst case, cluster compensation increased average running times for an instance by a factor of 4. On the flip side, the greatest speedup observed due to cluster compensation was by a factor of 248 times.

The experiments on data produced by the instance generators confirm our intuition that sharply clustered instances usually force longer running times, and support the claim that cluster compensation reduces that effect.

We also plotted probe and move depths for executions of the heuristic on selected instances. The probe depth plot correlates with the amount of search work performed by the heuristic, and the move depth plot can be used to see how much of that searching results in improvements being made. The probe and move depth plots concur with the observations for running times and quality of the output. First, cluster compensation does not significantly change the number and depth at which committed improving moves are found. Second, the probe depth plots show that the greatest reductions in running time correspond to the greatest reduction in search work performed. These effects are clearest for the TSP case, and less so for the weighted perfect matching case. Probe and move depth plots are valuable because, unlike absolute running times, they are independent of computing platform, programmer skill, and the amount of effort put into tuning the implementation. These plots are therefore the most convincing evidence that our results should carry over to other implementations of the Lin-Kernighan heuristic.

In the cases where efficient cluster compensation increases running times, often one or two factors can be blamed. First, the optimization phase runs very quickly. In these cases there is little slack: there is not much time that cluster compensation can cut out. Second, the time for preprocessing might be very high when compared to the time spent in the optimization phase. This is often the case for the weighted matching problem, where optimization times are typically very short. It is also true for larger non-geometric instances for either problem. The preprocessing stage for those inputs uses Prim's algorithm to find a minimum spanning tree. Prim's algorithm examines each edge a small constant number of times, so it is efficient. But on almost all the instance classes we tested, we observed subquadratic running time ($o(n^2)$) for the optimization phase of the heuristic. The time spent in the optimization phase is therefore asymptotically dominated by the time to build a minimum spanning tree since a complete graph has $\theta(n^2)$ edges. As instance sizes grow larger, the asymptotics take over and cluster compensation has no hope of recovering the time it spent in preprocessing. We find it ironic that computing a minimum spanning tree —one of the classic "easy" combinatorial optimization problems— should take more time than the optimization phase of the Lin-Kernighan heuristic, a heuristic useful for tackling the TSP —a classic "hard" combinatorial optimization problem. On the other hand this might only indicate that even though the problem is hard, the sample instances are tame.

When should cluster compensation be used? In the first bad case given above, the running time is short anyway, so we may not care about the small relative slowdown. The second bad case occurs only when n is large and we must use Prim's or any $\Omega(n^2)$ algorithm to compute a minimum spanning tree. We of course detect that case as soon as we examine the instance. The experimental evidence suggests that efficient cluster compensation should be applied in all other situations. Because it reduces running times with almost no loss in quality of the output, cluster compensation should be incorporated as a standard feature of any implementation of the Lin-Kernighan heuristic.

10.1 Future work

We conclude with some suggestions for further research.

10.1.1 Other parameter settings

The Lin-Kernighan heuristic has many tunable parameters. The experiments reported in this thesis use only a small number of the possible settings, most of which were found to be good choices by other researchers. It remains to be seen how cluster compensation affects the heuristic when using other settings, and whether cluster compensation makes other settings more attractive than the standard ones.

For example, in our experiments search sequences are forcibly terminated at 50 edge exchanges beyond the backtracking depth, *i.e.*, at t_{106} or at t_{108}. The probe depth graphs of Sections 6.7 and 7.7.6 show that Lin-Kernighan with cluster compensation only very rarely runs into that barrier. Cluster compensation may therefore provide an opportunity to either loosen or completely eliminate that depth bound. This might not be advisable for a general purpose implementation since the behaviour on the very worst input distributions, such as *e.g.*, Bentley's `arith`, `cubediam` and other line-like configurations, remains quite bad with many very deep and fruitless searches.

Another example is how efficient cluster compensation affects the choice of candidate sets. Cluster compensation makes Lin-Kernighan more selective in the paths it follows. The practitioner can perhaps afford the luxury of specifying a richer candidate set while remaining within the same time budget. With more search options offered by a rich candidate set, there is a chance of finding better tours.

We are unaware of any experimental work using the complementary tabu rule "never add a deleted edge" in place of "never delete an added edge." The first is used in Papadimitriou's proof that Lin-Kernighan solves a problem that is PLS-complete; the second is the only tabu rule used in the Johnson *et al.* implementation and in our own; Lin and Kernighan's original implementation used both. We suspect that switching the tabu rule would not significantly change the performance of the heuristic, since it appears that the cumulative gain criterion is the primary pruning mechanism in any case. However, cluster compensation might become more important for good performance with the new tabu rule.

10.1.2 Partial use of cluster compensation

One class of inputs for which efficient cluster compensation is not a performance win are extremely uniform instances with little or no clustering. One is tempted to detect those kinds of instances and turn cluster compensation off for them. Here's one way to do it; when we build a minimum spanning tree, if its longest edge is not much longer than its shortest edge, then by Lemma 3.5 the cluster distance is pretty much constant across the instance, and we can turn cluster compensation off. But there's a fly in the ointment; in performing this test, we have already spent most of the preprocessing time required for cluster compensation in just trying to detect the highly uniform case. So the cost of detecting the bad case for cluster compensation is about the same cost of performing the cluster compensation anyway.

There are several ways of softening cluster compensation. First, we could discount by $\beta \cdot c(u, v)$ with $\beta < 1$ instead of the full cluster distance $c(u, v)$. I would guess that when applying these techniques, the running times would increase with decreasing β down to 0, with the initial step being the most dramatic. Second, we could flip a weighted coin to determine whether to factor in the cluster distance. Third, we could discount by a particular cluster distance only if it was larger than some threshhold, say, the median of weights in a minimum spanning tree for the instance. Note that these solutions add extra overhead to the inner loop of the heuristic, so any benefits they provide would have to be balanced against extra runtime overhead. Each of these would be interesting experiments.

The other main situation in which cluster compensation increases running times occurs when computing a minimum spanning tree takes a relatively long time. Perhaps we can still capture the spirit of cluster compensation by basing our cluster distance computations on only approximately minimum spanning trees. Perhaps there is a tradeoff, with poorer spanning trees translating to less precise cluster distance computations and therefore less beneficial results when used by the Lin-Kernighan heuristic. Again, we find it ironic that a quintessentially easy combinatorial optimization problem should be a barrier to good performance of heuristics for a quintessentially hard problem.

10.1.3 Closing the gap

The quality of the answers produced by the Lin-Kernighan heuristic is consistent. For both minimum weight perfect matching and the TSP, the answers are usually within 2% of optimal. In contrast, its running times can vary greatly depending on the configuration of the distance function of the graph. Cluster compensation is designed to speed up the heuristic in the presence of sharp clustering. But even when cluster compensation is used, there is still a large difference between runs on instances that are sharply clustered and instances that are not. An obvious goal would be to close that gap, *i.e.*, reduce running times for all size n instances to be more in line with the fastest size n runs. In light of the hardness of approximating the TSP [70], we might never fully achieve this goal. But we may learn a lot in the attempt.

10.1.4 Modeling the behaviour of the heuristic

As a first step in improving the heuristic, it would be useful to more accurately model its behaviour so that running times could be predicted. An accurate model might go a long way toward understanding the performance problem and therefore focus algorithmic tuning effort. This is akin to how practitioners already use program execution profiling to find where their program spends its time, using that information to tune their algorithms. Much of Sections 3.1 and 3.2 is devoted to describing our own qualitative understanding of the broad behaviour of the Lin-Kernighan heuristic. The great difficulty in modeling many heuristics lies in the fact that their important behavioural patterns are emergent, and therefore cannot be seen by merely aggregating models of their small scale behaviour. The whole is usually greater than the sum of its parts.

On a more mundane level, a performance model can be useful in its own right. If the running time for the heuristic could be determined with some accuracy as a function of the algorithm's parameters, then one can use the model to pick appropriate parameter settings for a given situation. We would use the model to find parameter settings to get the best answers in a given time and space budget. Of course for the model to be useful in this way, less effort must be spent in evaluating and finding good settings from the model than would be spent just running the heuristic with standard settings.

To achieve significant success, modeling should always be closely tied to experimentation and instrumentation. Traditionally, the output of instrumentation has been tables of numbers and static graphical plots. Algorithm animation also holds promise as an avenue for qualitative discovery. For example, Bentley [15] describes how the idea for "don't-look" bits came out of viewing animations of the 2-Opt algorithm in action. Our own research benefited from taking an explicitly visual approach to understanding the heuristic. In particular, our implementation can automatically produce PostScript figures depicting data structures and the progress of the heuristic.[1] One way to sum up our attitude is that there is "sight" in "insight."

10.1.5 Maximization problems

From a theoretical perspective, maximization problems are the same as minimization problems. To convert one form into another, one need only choose a constant $M > \max_{e \in E} \omega(e)$ and then set $\omega'(e) = M - \omega(e)$.

In practice things are more complicated, especially in the context of the techniques used for good implementations of the Lin-Kernighan heuristics. For example, in the minimization form of the heuristic the candidate subgraphs are usually composed of edges between near neighbours, and are precomputed and stored as neighbour lists. Those edges are used because they are likely to be present in good solutions, and there are usually not very many such edges. In a maximization problem, a larger fraction of the edges may be considered candidates for good solutions. To understand the difference, consider the important case where ω obeys the triangle inequality. If $\omega(a,b)$ and $\omega(b,c)$ are small, then $\omega(a,c)$ is also small, in fact $\omega(a,c) \leq \omega(a,b)+\omega(b,c)$. But if both $\omega(a,b)$ and $\omega(b,c)$ are large relative to most other edges, then the $\omega(a,b)+\omega(b,c)$ bound on $\omega(a,c)$ is not a tight constraint: $\omega(a,c)$ is free to be very small or very large in comparison to all the other edges in the graph.

With this insight we might want to avoid using a fixed candidate subgraph. We could instead generate the candidate subgraph dynamically and randomly. That is, each time we search from vertex t_{2i} we could select candidates for t_{2i+1} at random instead of scanning a fixed list of near neighbours of t_{2i}. The qualitative argument above suggests

[1] Early in the development process, we knew our k-d tree code was buggy. Several days of symbolically-oriented code scrutiny failed to find the problem. We then inserted code to produce PostScript figures depicting the data structure. The bug was obvious once the first figure was displayed, and the code was fixed within minutes.

this would be more fruitful for maximization problems than for minimization problems. Unfortunately, this random sampling scheme makes termination more fuzzy. We would likely construct a different set if we resample the space of cities each time a candidate list is required. In that case, we might take a cue from Iterated Lin-Kernighan and terminate the search after each city has been examined a small fixed number of times without finding an improvement anchored at that city.

These ideas take some inspiration from the work of Johnson *et al.* as reported in [46]. They experiment with the use of k-d trees to dynamically generate exact neighbour lists. This saves space because neighbour lists need not be stored, but increases running time because the lists are regenerated anew at each search. For a million city uniform random geometric instance, space usage is reduced from 275 megabytes down to 72 megabytes, but running time is increased by a factor of four.

On a different note, cluster compensation itself has less intuitive appeal in the maximization context. Cluster distance would still be an estimate of possible future closing up costs, though we would want to *maximize* those costs. Cluster distance would be computed in the same way, *i.e.*, by least common ancestors in the topology tree constructed from a *minimum* spanning tree. The conflicting orientations, using a minimum spanning tree but wanting to maximize costs, jars our sensibilities.

We suspect that for graphs having a distance function obeying the triangle inequality, Lin-Kernighan for maximization problems would run much longer and find poorer answers than Lin-Kernighan for minimization problems. Part of this intuition is based on the arguments on the structure of the space of good solutions. We believe that light tours (and matchings) are similar to each other in some sense, while heavy tours can differ greatly. Lin-Kernighan for these minimization problems is successful because it easily jumps from one good tour (or matching) to another. This may not be the case for maximization problems with those distance functions. We are unaware of experimental work with Lin-Kernighan for maximization problems.

10.1.6 Other instance generators

Finally, there is always room for new instance generation algorithms. Each would be tailored according to the present need. In particular, instance generators are very useful for testing our intuition about behaviour patterns of the heuristic. This holds for all heuristic algorithms, and not just the Lin-Kernighan heuristic.

Bibliography

[1] E. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons Ltd., 1997.

[2] E. H. L. Aarts, J. H. M. Korst, and P. J. M. van Laarhoven. Simulated annealing. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, Wiley Interscience series in discrete mathematics and optimization, chapter 4, pages 91–120. John Wiley & Sons, 1997.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[4] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Concorde: A code for solving Traveling Salesman Problems. Published electronically at http://www.caam.rice.edu/~keck/concorde.html.

[5] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. (certificates of optimality for certain tsp solutions). Published electronically at ftp://netlib.att.com/netlib/att/math/applegate/TSP/proofs, August 1994.

[6] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. Finding cuts in the TSP (a preliminary report). Published electronically at ftp://netlib.att.com/netlib/att/math/applegate/TSP/tsp_aug23.ps.Z, August 1994.

[7] D. Applegate and W. Cook. (proposal for segment tree implementation of oriented tour data type). Referenced as a private communication by Fredman *et al.* [33], 1990.

[8] D. Applegate and W. Cook. Solving large-scale matching problems. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS*

Implementation Challenge, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 557–576. American Mathematical Society, 1993.

[9] S. Arora. Nearly Linear Time Approximation Schemes for Euclidean TSP and Other Geometric Problems. In *Proceedings of the Thirty-Eighth Annual Symposium on Foundations of Computer Science*, October 1997.

[10] S. Arora. Polynomial time approximation schemes for Euclidean TSP and other geometric problems. To appear in the *Journal of the ACM*. See also http://www.cs.princeton.edu/~arora, 1997.

[11] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and intractability of of approximation problems. In *Proceedings of the Thirty-Third Annual Symposium on Foundations of Computer Science*, pages 13–22, October 1992.

[12] E. B. Baum. Iterated descent: a better algorithm for local search in combinatorial optimization problems, 1986. Manuscript.

[13] E. B. Baum. Towards practical 'neural' computation for combinatorial optimization. In J. S. Denker, editor, *Neural Networks for Computing, Proceedings AIP Conference 151*, pages 53–58, New York, 1986. American Institute of Physics.

[14] J. Beardwood, J. H. Halton, and J. M. Hammersley. The shortest path through many points. In *Proceedings of the Cambridge Philosophical Society*, volume 55, pages 299–327, 1959.

[15] J. L. Bentley. Experiments on Traveling Salesman Heuristics. In *First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 91–99, January 1990. San Francisco, California.

[16] J. L. Bentley. K-d Trees for Semidynamic Point Sets. In *Proceedings of the 6th Annual Symposium on Computational Geometry*, pages 187–197, June 1990.

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

[18] J. L. Bentley and J. H. Friedman. Fast Algorithms for Constructing Minimal Spanning Trees in Coordinate Spaces. *IEEE Transactions on Computers*, C-27(2):97–105, 1973.

[19] J. L. Bentley and J. B. Saxe. An Analysis of Two Heuristics for the Euclidean Traveling Salesman Problem. In *18th Annual Allerton Conference on Communication, Control, and Computing*, pages 41–49, October 1980.

[20] P. Briggs and L. Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(4):59–69, March 1993.

[21] R. E. Burkard, V. G. Deineko, R. Van Dal, J. A. A. Van Der Veen, and G. J. Woeginger. Well-solvable special cases of the Traveling Salesman Problem: a survey. *Siam Review*, 40(3):496–546, September 1998.

[22] B. Chandra, H. Karloff, and C. Tovey. New results on the old k-opt algorithm for the TSP. In *Proceedings of the Fifth ACM-SIAM Symposium on Discrete Algorithms*, pages 150–159, 1994.

[23] M. Chrobak, T. Szymacha, and A. Krawczyk. A data structure useful for finding Hamiltonian cycles. *Theoretical Computer Science*, 71:419–424, 1990.

[24] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[25] S. A. Cook. An overview of computation complexity (1982). In *ACM Turing Award Lectures: The First Twenty Years, ACM Press Anthology Series*. ACM Press, New York, Addison-Wesley, 1987.

[26] W. Cook. The traveling salesman problem. Invited presentation at the Ninth SIAM Conference on Discrete Mathematics, July 13, 1998.

[27] W. Cook and A. Rohe. Computing minimum-weight perfect matchings. Describes the Blossom IV implementation written by the authors. The description and the implementation are available from http://www.or.uni-bonn.de/home/rohe/matching.html.

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

[29] A. E. Dunlop and B. W. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer-Aided Design*, 4:92–98, 1985.

[30] J. Edmonds. Matching and a polyhedron with 0-1 vertices. *Journal of Research of the National Bureau of Standards*, 69B:125–130, 1965.

[31] K. J. Falconer. *The geometry of fractal sets*. Cambridge tracts in mathematics. Cambridge University Press, Cambridge UK, 1985.

[32] M. M. Flood. The traveling-salesman problem. *Operations Research*, 4:61–75, 1956.

[33] M. L. Fredman, D. S. Johnson, L. A. McGeoch, and G. Ostheimer. Data structures for traveling salesmen. *Journal of Algorithms*, 18:432–479, 1995.

[34] H. N. Gabow. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *Journal of the ACM*, 23(2):221–234, April 1976.

[35] M. R. Garey and D. S. Johnson. *Computers and Intractabilty: A Guide to the theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[36] P. C. Gilmore, E. L. Lawler, and D. B. Shmoys. Well-solved special cases. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors, *The Traveling Salesman Problem*, chapter 4, pages 87–143. John Wiley & Sons Ltd., 1985.

[37] M. X. Goemans and D. J. Bertsimas. Probabilistic analysis of the Held and Karp lower bound for the Euclidean traveling salesman problem. *Mathematics of Operations Research*, 16(1):72–89, February 1991.

[38] J. Gu. Efficient local search with search space smoothing: a case study of the traveling salesman problem (TSP). *IEE Transactions on Systems, Man, and Cybernetics*, 24:724–735, 1994.

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

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

[41] J. E. Hopcroft and J. D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, December 1973.

[42] M. D. Hutton. *Characterization and Parameterized Generation of Digital Circuits*. PhD thesis, Department of Computer Science, University of Toronto, 1997. Available electronically at http://www.eecg.toronto.edu/~mdhutton/thesis.html.

[43] C. Imielinska and B. Kalantari. A generalized hypergreed algorithm for weighted perfect matching. *BIT*, 33:178–189, 1993.

[44] D. S. Johnson. Local optimization and the traveling salesman problem. In *ICALP '90*, pages 446–461. Springer-Verlag, 1990. Proceedings of the 17^{th} Colloquium on Automata, Languages, and Programming.

[45] D. S. Johnson, August 1998. Personal communication.

[46] D. S. Johnson and L. A. McGeoch. The traveling salesman problem: a case study. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, Wiley Interscience series in discrete mathematics and optimization, chapter 8, pages 215–310. John Wiley & Sons, 1997.

[47] D. S. Johnson, L. A. McGeoch, and E. E. Rothberg. Asymptotic experimental analysis for the Held-Karp Traveling Salesman bound. Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, January 28–30, 1996., 1996.

[48] D. S. Johnson and C. H. Papadimitriou. Computational complexity. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors, *The Traveling Salesman Problem*, chapter 3, pages 37–85. John Wiley & Sons Ltd., 1985.

[49] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. How easy is local search? *Journal of Computer and System Sciences*, 37:79–100, 1988.

[50] R. M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations (Proceedings of a Symposium on the Complexity of Computer Computations, March, 1972, Yorktown Heights, NY)*, pages 85–103. Plenum Press, New York, 1972.

[51] R. M. Karp. Probabilistic analysis of partitioning algorithms for the traveling salesman problem in the plane. *Mathematics of Operations Research*, 2:209–224, 1977.

[52] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49(2):291–307, 1970.

[53] B. W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. McGraw-Hill, New York, NY, USA, 1974.

[54] D. E. Knuth. *Literate Programming*. Stanford Center for the Study of Language and Information, Stanford, California, 1992. CSLI Lecture Notes No. 27.

[55] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley, 1993.

[56] D. E. Knuth and S. Levy. The CWEB System of Structured Documentaion, version 3.0. Published electronically at ftp://labrea.stanford.edu/pub/cweb., 1987. Also available in book form from Addison-Wesley, 1994.

[57] M. W. Krentel. On finding locally optimal solutions. *SIAM Journal on Computing*, 19:742–749, August 1990.

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

[59] T. Lengauer. *Combinatorial algorithms for integrated circuit layout*. John Wiley and Sons, Chicester, 1990.

[60] S. Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44:2245–2269, 1965.

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

[62] F. Margot. Quick updates for p-opt TSP heuristics. *Operations Research Letters*, 11:45–46, February 1992.

[63] O. C. Martin and S. W. Otto. Combining simulated annealing with local search heuristics. In I. H. Osman G. Laporte, editor, *Metaheuristics in Combinatorial Optimization*, number 63 in Annals of Operations Research, pages 57–75. Baltzer, Amsterdam, 1996.

[64] O. C. Martin, S. W. Otto, and E. W. Felten. Large-step Markov chains for the traveling salesman problem. *Complex Systems*, 5:299–326, 1991.

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

[66] C. C. McGeoch. Electronically available materials. In David S. Johnson and Catherine C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 577–582. American Mathematical Society, 1993.

[67] C. H. Papadimitriou. The complexity of the Lin-Kernighan heuristic for the traveling salesman problem. *SIAM Journal on Computing*, 21(3):450–465, June 1992.

[68] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, USA, 1982. Republished in 1998 by Dover Publications, Mineola, New York.

[69] C. H. Papadimitriou and M. Yannakakis. Optimization, approximation and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.

[70] C. H. Papadimitriou and M. Yannakakis. The traveling salesman problem with distances one and two. *Mathematics of Operations Research*, 18(1):1–11, February 1993.

[71] D. A. Plaisted. Heuristic matching for graphs satisfying the triangle inequality. *Journal of Algorithms*, 5(2):163–179, June 1984.

[72] F. P. Preparata and M. I. Shamos. *Computational geometry: An introduction*. Springer-Verlag, New York, 1985.

[73] G. Reinelt. TSPLIB — a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991. Updated versions of the library and the world-best values for tour lengths are available at http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB.html.

[74] G. Reinelt. *The traveling salesman: Computational solutions for TSP applications*. Springer Verlag, 1994. LNCS 840.

[75] A. Rohe. Parallel lower and upper bounds for large tsps. *ZAMM*, 77(Supplement 2):429–432, 1997. Also available from http://www.or.uni-bonn.de/home/rohe/.

[76] A. Rohe. Parallele heuristiken für sehr große traveling salesman probleme (parallel heuristics for very big traveling salesman problems), 1997. Diplomarbeit, in German. Also available from http://www.or.uni-bonn.de/home/rohe/.

[77] A. Rohe. Personal communication, april 15, 1999.

[78] E. Rothberg. DIMACS wmatch benchmark computer code and data. ftp://ftp.dimacs.rutgers.edu/pub/netflow/benchmarks.

[79] S. Sahni and T. Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23:555–565, 1976.

[80] A. A. Schäffer and M. Yannakakis. Simple local search problems that are hard to solve. *SIAM Journal on Computing*, 20(1):56–87, February 1991.

[81] B. Schieber. Parallel lowest common ancestor computation. In John H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 6, pages 259–273. Morgan Kaufmann, 1993.

[82] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[83] K. J. Supowit, D. A. Plaisted, and E. M. Reingold. Heuristics for weighted perfect matching. In *Conference Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, pages 398–419, Los Angeles, California, 28–30 April 1980.

[84] P. M. Vaidya. Geometry helps in matching. *SIAM Journal of Computing*, pages 1202–1255, 1989.

[85] M. Yannakakis. Computational Complexity. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, Wiley Interscience series in discrete mathematics and optimization, chapter 2, pages 19–55. John Wiley & Sons, 1997.

Appendix A

A completeness result

This thesis is primarily concerned with the experimental effect of cluster compensation on the Lin-Kernighan heuristic. In contrast, this appendix is concerned with the worst-case behaviour of the heuristic. Papadimitriou [67] proved that a stylized version of the Lin-Kernighan heuristic for the Traveling Salesman Problem solves a PLS-complete problem. PLS is a complexity class capturing all reasonable deterministic local search heuristics; we give more details below. The following sections sketch how to adapt that proof in the case that the stylized Lin-Kernighan heuristic employs cluster compensation.

A.1 Introduction

Papadimitriou showed that a stylized version of the Lin-Kernighan heuristic for the TSP solves a PLS-complete problem [67]. The problem in question, which we call *TSP-LK*, is the one naturally associated with the heuristic: find a tour that Lin-Kernighan for the TSP cannot improve. (Papadimitriou calls the problem *LIN-KERNIGHAN*, identifying the problem with the heuristic; we want to emphasize the underlying search problem as well.)

The version of the Lin-Kernighan heuristic considered by Papadimitriou differs slightly from the version used in practice. First, Papadimitriou's version uses a different set of tabu rules. The two tabu rules used by Lin and Kernighan are "never delete an added edge" and "never add a deleted edge." Most implementations use both tabu rules, and the recent Johnson *et al.* implementations use only the first [46]. Papadimitriou's version uses only the second tabu rule, "never add a deleted edge." Second, Papadimitriou's version of the heuristic uses the basic cumulative gain criterion, Criterion 2.7. That is, it requires the cumulative gain to be greater than zero, not greater than the best net gain. Third, it requires backtracking only to t_4 as opposed to at least t_6 in the standard version. However, as a first step it searches for all possible improving 2-changes and 3-changes. Fourth, no special candidate sets are used: given t_{2i}, all vertices are considered candidates for t_{2i+1}. Finally, Papadimitriou's version of the heuristic uses the same feasibility rules (see Section 4.2), and the same greedy selection criteria as the standard version.

Cluster compensation was motivated in part by Papadimitriou's proof that problem *TSP-LK* is PLS-complete. One of the main features of the proof is the use of heavy bait edges to drive up the cumulative gain to very large values. This forces a particular class of search sequences to explore large portions of the graph before finally returning to the start vertex. The final added edge is always at least as heavy as the first removed (bait) edge.

Cluster compensation is designed to mitigate the effects of bait edges. Does it always protect the heuristic from bad cases? The answer is "no." We will describe how Papadimitriou's proof can be adjusted to show that a cluster compensating version of his Lin-Kernighan heuristic for the TSP also solves a PLS-complete problem, *TSP-LKCC*. As before, this is a problem naturally associated with the heuristic: find a tour that cluster compensating Lin-Kernighan for the TSP cannot improve.

In the following, the heuristic is the same as described by Papadimitriou, except that for LK-style searches, cluster compensating versions of the basic cumulative gain and greedy selection criteria (Criteria 3.2 and 3.4) are used in place of their ordinary versions (Criteria 2.7 and 2.9). In particular, all 2-changes and 3-changes are searched without applying cluster compensation.

Papadimitriou's construction uses negative edge weights to force certain edges to always be examined first, or to force them to always end up in the tour. Having negative edge weights forces some cluster distances to be negative. Cluster compensation is supposed to shorten search sequence lengths by rejecting certain moves that would otherwise be accepted. But subtracting a negative-valued cluster distance makes *more* moves acceptable. We therefore disallow the cluster distance discount from being negative. That is, we use

$$cum\_gain(2i) - \max(0, c(t_{2i}, t_1)) > 0$$

for the cumulative gain criterion, and

$$\text{maximize } cum\_gain(2i+2) - \max(0, c(t_{2i+2}, t_1))$$

for the greedy selection criterion.

Fortunately, Papadimitriou's proof translates almost directly to the new setting. We need make only one very minor adjustment: we halve the weights of two edges in the constructed TSP instance. The change is small enough that we need not repeat the entire argument. The following sections introduce PLS and give the outline of Papadimitriou's proof, showing how to adjust it to the new setting.

A.2 PLS

The class PLS, standing for Polynomial Local Search, is a set of search problems. Each member of PLS is a pair (A, L), where A is an ordinary search problem and L is a local search algorithm for A. We say an algorithm solves (A, L) if it always finds local optima for A with respect to L. Schematically, each member of PLS can be stated as follows:

Problem:	(A, L)
Input:	Any input x of search problem A.
Output:	A local optimum of x with respect to local search algorithm L.

Among other restrictions, we require that each neighbourhood search performed by algorithm L take polynomial time.

Definition A.1 (PLS [49, 67]) *Problem (A, L) is in PLS if it is of the form "given input x to problem A, find $s \in F_x$ locally optimal with respect to L," and the following conditions are satisfied:*

1. *For each input x there is a set of feasible solutions F_x.*

2. *Given x and s, there is a polytime algorithm for deciding whether s is feasible, i.e., $s \in F_x$.*

3. *Given x, we can compute some member s_0 of F_x in polynomial time; s_0 is an initial solution.*

4. *Given x and $s \in F_x$ we can compute the cost of s in polynomial time.*

5. *Given x and $s \in F_x$ the neighbourhood search of algorithm L either finds a better solution $s' \in F_x$, or simply halts without finding one. If the improvement heuristic*

halts without finding a better solution, then we say that s is locally optimal. *Each neighbourhood search takes time bounded by a polynomial in the sizes of x and s.*

For example, problem *TSP-LK* is a pairing of the ordinary TSP search problem together with the (stylized) Lin-Kernighan heuristic for the TSP. Given a distance matrix as input, the goal is to find a locally optimal tour with respect to Lin-Kernighan.

Problem:	*TSP-LK*
Input:	An instance x of the symmetric TSP, *i.e.*, a complete undirected weighted graph $x = G = (V, E, \omega)$.
Output:	A tour for G that cannot be improved by the (stylized) Lin-Kernighan heuristic for the TSP.

TSP-LK is in PLS because it is of the proper form and it satisfies each of the other criteria:

1. In the case of the TSP, x is the distance matrix, and the feasible solutions F_x is the set of tours of the graph.
2. It is easy to determine whether an edge set is a tour.
3. Since our graphs are complete, finding a tour is trivial.
4. The cost of a tour is just the sum of its edge weights.
5. A single Lin-Kernighan neighbourhood search takes polynomial time. That is, in polynomial time it either finds a better solution or it gives up.

The natural local search algorithm associated with a PLS problem (A, L) is as follows: find a start solution s_0; repeatedly use the neighbourhood search of improvement heuristic L to try to find a better solution s' from the current solution s; when no better s' is found, stop and answer s. Each step takes polynomial time, but the algorithm is free to take exponentially many steps. Note that an algorithm solving a PLS problem need not be a local search algorithm, and need not use the improvement heuristic at all.

The reader may already be familiar with the idea of reducibilities among decision problems forming the basis for the theory of NP-completeness [24, 50, 35]. A problem A is NP-complete if (a) A is in NP, and (b) any problem B in NP is reducible to A. By the transitivity of the reducibility relation, every NP problem reduces to A if some NP-complete problem C reduces to A.

Similarly, the notion of PLS-reduction underlies the theory of PLS-completeness. Problem (B, L_B) in PLS reduces to problem (A, L_A) in PLS if problem (B, L_B) can be solved by an algorithm that invokes as a subroutine an algorithm solving problem (A, L_A). More precisely, (B, L_B) reduces to (A, L_A) if there are two polytime computable functions f and g so that for any instance x of B, $f(x)$ is an instance of A, and if s is a local optimum for $f(x)$ (with respect to L_A) then $g(s)$ is a local optimum for x (with respect to L_B). To show problem (A, L_A) is PLS-complete, we show that (A, L_A) is in PLS and that every PLS problem (B, L_B) PLS-reduces to (A, L_A). As before, showing that some PLS-complete problem (C, L_C) reduces to (A, L_A) shows that every PLS problem reduces to (A, L_A) [49].

TSP-LKCC is the problem of finding locally optimal tours with respect to the (stylized) Lin-Kernighan heuristic that uses cluster compensation.

Problem:	*TSP-LKCC*
Input:	An instance of the symmetric TSP, *i.e.*, a complete undirected weighted graph, $G = (V, E, \omega)$.
Output:	A tour for G that cannot be improved by the (stylized) Lin-Kernighan heuristic for the TSP that uses cluster compensation.

We show the PLS-completeness of *TSP-LKCC* in two steps. First, *TSP-LKCC* is in PLS. Second, a PLS-complete problem PLS-reduces to *TSP-LKCC*.

A.3 *TSP-LKCC* is in PLS

It is easy to show that *TSP-LKCC* is in PLS. It follows for almost exactly the same reasons that *TSP-LK* is in PLS.

Lemma A.2 *TSP-LKCC is in PLS.*

Proof: An algorithm solves *TSP-LKCC* if whenever it is given a distance matrix as input, it always finds a locally optimal tour with respect to the (stylized) Lin-Kernighan heuristic using cluster compensation. First, the problem is of the proper form. We satisfy the other requirements of Definition A.1 one at a time:

1. The input is a complete undirected graph $G = (V, E)$ with symmetric edge weight function ω represented, say, as a matrix of integers. Let us write $n = |V|$. Feasible set F_G is the set of all tours on G.

2. In polynomial time, we can determine whether edge set T is a tour.

3. A start tour T_0 can be found in polynomial time, since G is complete. In fact, any ordering of the vertices will do.

4. The cost of a tour T is just the sum of the weights of its edges, which can be computed in polynomial time.

5. One iteration of Lin-Kernighan using cluster compensation can be performed in $O(n^7)$ time. See below.

Finally, the problem is of the proper form: given G, find a tour that cluster compensating Lin-Kernighan cannot improve.

The last point is the most involved. Searching for an improving 2-change or 3-change takes $O(n^3)$ time. The deep Lin-Kernighan search sequences backtrack over all choices for t_1 thorough t_4. We have n choices for t_1; given t_1 we have two choices for t_2; given t_1 and t_2 we have roughly n choices for t_3; given t_1, t_2, and t_3 we have one choice for t_4. All told, there are $O(n^2)$ possibilities for t_1 through t_4. Each search sequence can extend up to $O(n^2)$ moves since we are prohibited only from adding a deleted edge. If we use a simple array for the tabu list, then lengthening the t sequence from t_{2i} to t_{2i+2} takes $O(i)$ time. We see that given a start vertex t_1, the low level backtracking takes $O(n^6)$ time. Iterating over all n choices for t_1 therefore takes $O(n^7)$ time. We can reduce this worst-case bound by using better data structures for the tabu list.

Efficient cluster compensation adds to this total but does not increase the asymptotic running time. Online cluster distance queries are performed in constant time and therefore are absorbed into the other bookkeeping. As for the required preprocessing, we have shown in Section 3.3 that it may be done in $O(n \log n)$ time plus the time needed to compute a minimum spanning tree, or at most $O(n^2)$.

This completes the proof that *TSP-LKCC* is in PLS. ∎

A.4 *TSP-LK* is PLS-hard

This section outlines Papadimitriou's proof that problem *TSP-LK* is PLS-hard. The proof consists of showing that problem *2-SATFLIP* PLS-reduces to *TSP-LK*. Since *2-SATFLIP* is PLS-complete [57, 80], this shows every PLS problem PLS-reduces to *TSP-LK*. This section summarizes the construction and the argument. For more detail see [67]. The next section shows how to modify the construction and the argument to show that *2-SATFLIP* PLS-reduces to *TSP-LKCC*.

The problem *2-SATFLIP* can be described as follows. We are given a boolean formula C in conjunctive normal form with clauses $C_1, \ldots, C_m$. Each clause C_i has exactly two literals and has an associated integer weight w_i. A literal is either a boolean variable or the negation of a boolean variable. We think of the weight of a clause as the penalty incurred for not satisfying that clause. The *2-SATFLIP* problem is to assign truth values to the variables so that the total weight of the unsatisfied clauses cannot be reduced by flipping the truth value of a single variable. Schematically, we have:

Problem:	*2-SATFLIP*
Input:	Boolean formula C in conjunctive normal form, each clause having exactly two literals, and a weight for each clause.
Output:	A truth assignment to variables of C so that flipping the value of any variable does not reduce the weight of the unsatisfied clauses.

A.4.1 Sketch of the construction

To reduce *2-SATFLIP* to *TSP-LK*, a set of clauses $C = C_1, \ldots, C_m$ is transformed into a complete weighted undirected graph $f(C)$. Locally optimal tours in $f(C)$ have a special structure corresponding to locally optimal weighted truth assignments for C. An assignment σ of truth values to variables corresponds to a particular tour $T(\sigma)$ of $f(C)$. All such tours are called *standard tours*.

The edges of graph $f(C)$ can be partitioned by weight into distinct classes. There is a large positive constant M depending only on the set of clauses C so that the weights of edges in the classes are: $-M^4$, -1, 0, at least M but less than M^2, and M^j for $j \in \{2, 3, 4, 5, 6, 7, 8, 9\}$. Each class serves a distinct purpose, and our sketch will detail all the classes excepting those with edges heavier than M^4.

Graph $f(C)$ has a sparse skeleton of light edges, depicted schematically in Figure A.1 (corresponding to Papadimitriou's Figure 2 [67]). Each logical variable x_i in C is associated in $f(C)$ with a pair of complex paths, or "ribs," and with a *start* edge from S to Y_i. The ribs are depicted as thick grey bars in the figure. Each variable has two ribs, corresponding to the two possible truth values it may be assigned. Rib x_i corresponds to the clauses satisfied when variable x_i is assigned the true value, and rib $\overline{x_i}$ corresponds

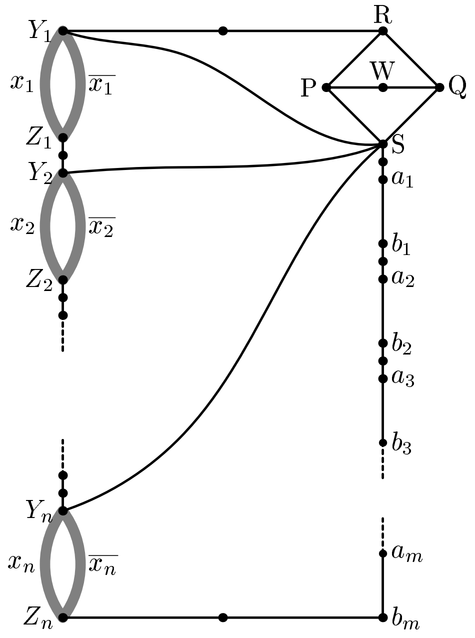


Figure A.1: Skeleton of light edges in the TSP instance constructed for the reduction from *2-SATFLIP* to *TSP-LK*.

to the clauses satisfied when x_i is assigned the false value. The standard tour $T(\sigma)$ of $f(C)$ traverses exactly one rib for each variable: rib x_i is traversed if $\sigma(x_i) = \mathsf{true}$ and rib $\overline{x_i}$ is traversed if $\sigma(x_i) = \mathsf{false}$.

Each clause $C_j = (L_{j1} \vee L_{j2})$ is represented by an edge (a_j, b_j) and by an *OR-device* straddling ribs L_{j1} and L_{j2}. An OR-device is depicted in Figure A.2 (a); it has 12 nodes and 14 edges. The weights of the edges are given in Figure A.2(b). The top row of nodes and edges corresponds to the first literal of the associated clause, and the bottom row to the second literal of the clause. If the four corner nodes are the only allowable entry and exit points then there are exactly three ways to traverse the OR-device, shown in Figures A.2(c), (d), and (e). The three ways of traversing an OR-device correspond to the three ways clause $(L_{j1} \vee L_{j2})$ can be satisfied: by literal L_{j1} alone, by literal L_{j2} alone, or by both literals simultaneously. In each of these three configurations, each of the diagonals is traversed in a consistent order. We can think of the bottom-left and top-right nodes as entry nodes, and the top-left and bottom-right nodes as exit nodes.

The rib for literal L strings together all the OR-devices corresponding to clauses in which L appears. Figure A.3 shows a rib corresponding to literal x_i appearing in three clauses. Either the bottom row or the top row from each device is incorporated into L's

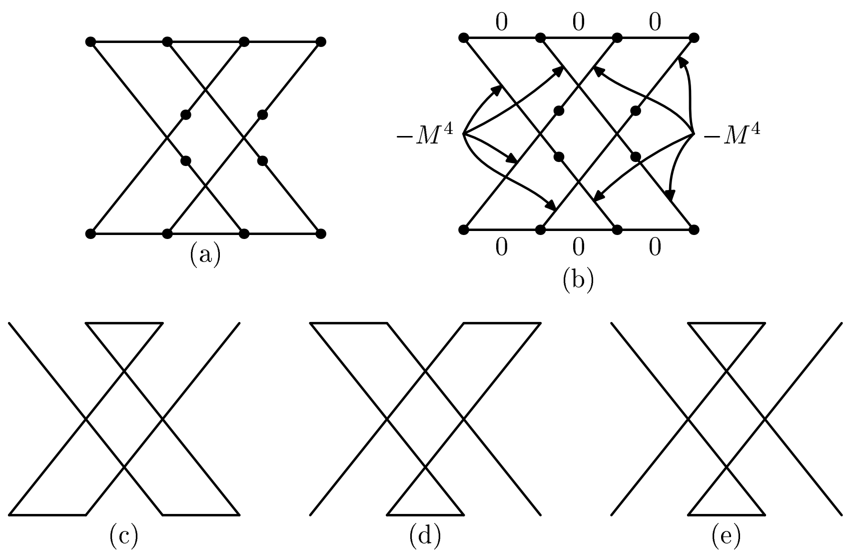


Figure A.2: An OR-device, corresponding to a clause with two literals. Parts (c), (d), and (e) show the three ways of traversing the device, corresponding to the three ways a two-literal clause can be satisfied.

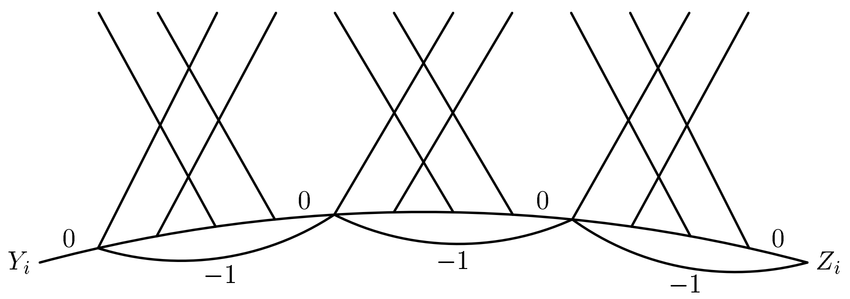


Figure A.3: A rib corresponding to a literal x_i appearing in three clauses.

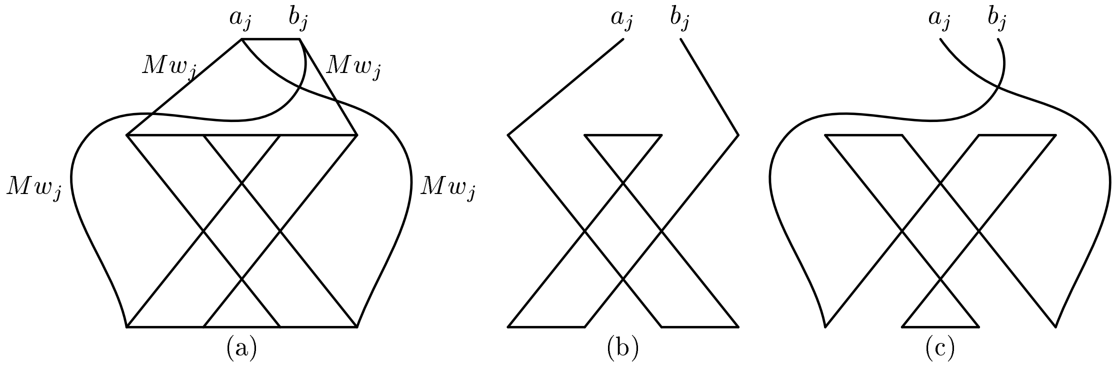


Figure A.4: (a) Edge (a_j, b_j) and penalty edges connecting to the OR-device for clause C_j; (b) OR-device picked up by top pair of penalty edges; (c) OR-device picked up by bottom pair of penalty edges.

rib. A zero-weight edge joins the exit node of one OR-device's row to the entry node of the next OR-device's row. Zero weight edges precede the first device and follow the last one. The entry node of one device is also joined by a *jump* edge to the entry node of the next device, and the entry node of the last device is joined by a jump edge to Z_j. Jump edges have weight -1 and are used to guide the Lin-Kernighan search.

The OR-device for clause C_j is also joined to edge (a_j, b_j) by four penalty edges as shown in Figure A.4(a). Vertex a_j is joined by penalty edges to the two exit nodes, and b_j is joined by penalty edges to the two entry nodes. Edge (a_j, b_j) has weight zero, and each of the penalty edges have weight Mw_j, *i.e.*, proportional to the cost w_j of not satisfying clause C_j. If $C_j = (L_{j1} \vee L_{j2})$ is satisfied by truth assignment σ, for example $\sigma(L_{j2}) = \mathsf{true}$, then in the standard tour $T(\sigma)$ the OR device for C_j is traversed during the traversal of the rib for L_{j2}, and edge (a_j, b_j) is also in $T(\sigma)$. If C_j is not satisfied by σ then the OR-device for C_j is not traversed with any rib. Instead, it is picked up by two of the penalty edges as in Figure A.4(b) or Figure A.4(c), and edge (a_j, b_j) does not appear in the tour.

All of the start edges have weight M^2, and none of them appear in a standard tour.

Figure A.1 also contains a series of two-edge paths: counterclockwise, they run from R to Y_1, from Z_i to Y_{i+1}, from Z_n to b_m, from a_i to b_{i-1}, from a_1 to S, and from P to Q. Each of the edges in those paths are assigned a weight of $-M^4$, just like the internal diagonal edges of the OR devices. Let us call all of these $-M^4$-weight edges "crazy glue" edges. They are so light that all of them appear in every standard tour. We will see that

Lin-Kernighan searches on standard tours never break crazy glue edges.

The four diagonal edges in the diamond-shape device in the upper right of Figure A.1, namely (R, P), (R, Q), (P, S), and (Q, S) are all assigned weight M^3. In a standard tour one of two cases occurs. In the first case, both (R, P) and (Q, S) appear and neither (R, Q) nor (P, S) appear. The second case is complementary: neither (R, P) nor (Q, S) appear and both (R, Q) and (P, S) appear.

This completes the description of standard tours, including the weights of the edges involved in them. Graph $f(C)$ is a complete graph, and the weights of all its other edges are fixed to very high values. Those heavy weights are polynomially bounded, ranging from M^4 through M^9. Setting an edge weight to be very large ensures that they never appear in standard tours, and are only removed and never added during the course of the execution of the Lin-Kernighan heuristic. They are never added because by construction, there are always better alternatives.

A.4.2 Proof sketch

The fact that *TSP-LK* is PLS-hard follows from the following two lemmas.

Lemma A.3 (Papadimitriou [67]) *A standard tour T is a local optimum only if $g(T)$ (the corresponding truth assignment) is optimum.*

Lemma A.4 (Papadimitriou [67]) *The only locally optimum tours are the standard tours.*

Let us first summarize the argument for Lemma A.3. Suppose the truth assignment $\sigma = g(T)$ is not a local optimum under algorithm *2-SATFLIP*. Then we can reduce the weight of the unsatisfied clauses by flipping the truth value of some variable, say x_i. Papadimitriou shows how a Lin-Kernighan improvement can be made to tour T. Essentially, a Lin-Kernighan improvement simulates the flip of the value of x_i.

Figure A.5 sketches the main steps taken by a Lin-Kernighan improvement in simulating the flip of the value assigned to x_3. For the sake of clarity, it omits the penalty edges and the steps required to remove them, and the many steps involved in reconfiguring individual OR devices. For all the details, see figures 6 through 14 of [67]. Part (a) shows a standard tour with variables x_1 and x_4 assigned the true value and x_2 and x_3 assigned the false value. Part (b) shows the removal of edge (Q, S), for a cumulative gain of M^3. Part (c) shows the addition of "start" edge (S, Y_i), for a new cumulative gain of

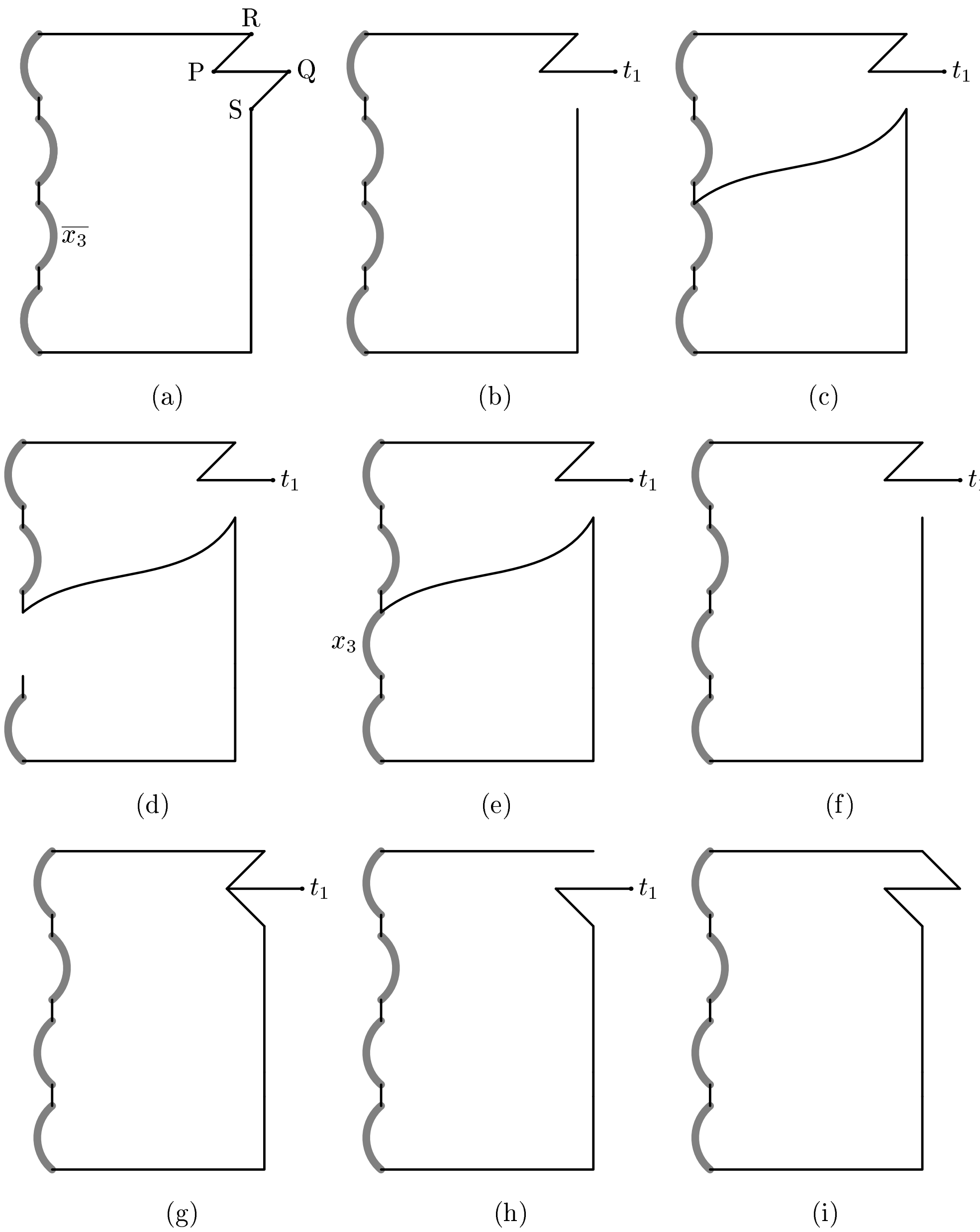


Figure A.5: Sketch of steps in Lin-Kernighan improvement on a standard tour.

$M^3 - M^2$. Part (d) shows the result of the many steps involved in changing the status of the OR devices along the rib for x_3. Any OR devices on that rib that were "satisfied" by $x_3 = \textsf{false}$ are now reconfigured to represent that they are no longer "satisfied" in that way. If any of those OR devices are no longer satisfied by either of their literals, then penalty edges are added, with the cumulative gain decreasing accordingly. Each penalty edge is of weight $M\omega_j$ for some j. Let us write $o(M^2)$ for those new penalty terms: no matter how they sum up, they are always dominated by M^2. The cumulative gain at (d) is therefore $M^3 - M^2 - o(M^2)$. Part (e) shows the result of processing the OR devices along the rib for literal x_3. That is, those OR devices are reconfigured to represent the fact the associated clauses are now satisfied by the assignment $x_3 = \textsf{true}$. This may involve the removal of some penalty edges, which increases the gain by an amount we represent by $o(M^2)$. Since flipping x_3 decreases the overall weight of unsatisfied clauses, it more than just cancels the previous $-o(M^2)$ term; we write the new cumulative gain as $M^3 - M^2 + o(M^2)$. Part (f) shows the removal of the start edge, and the cumulative gain is now $M^3 + o(M^2)$. Part (g) shows the addition of edge (S, P), resulting in the new cumulative gain of $o(M^2)$, a relatively small positive amount. The tabu rule "never add a deleted edge" prevented us from adding edge (Q, S), as it was the first edge removed in this search sequence. But part (g) depicts a non-tour, so we must continue. Part (h) shows the removal of edge (R, P), for a new cumulative gain of $M^3 + o(M^2)$. Finally, part (i) shows the new tour found by adding edge (R, Q), corresponding to the standard tour where the truth value of x_3 was flipped, *i.e.*, x_1, x_3, and x_4 are $\textsf{true}$, while x_2 is $\textsf{false}$. The net gain from (a) to (i) is the difference between the weights of the penalty edges removed and the weights of the penalty edges added (if any), a total in $o(M^2)$. This net gain (the net decrease in the tour length) is proportional to the reduction in the weight achieved by flipping the truth value of variable x_3.

Had edges (R, Q) and (P, S) been in the standard tour then we could have proceeded with a similar sequence, beginning with the removal of edge (P, S) and ending with the addition of edge (R, P). Such Lin-Kernighan improvements merely reverse the orientation of the traversal of that diamond in the graph.

The process, including the cumulative gains along the way, is summarized in Table A.1. There are a few key points we should observe. First, the cumulative gain is always positive, as required by the heuristic. Second, the removal of the first edge, the "bait" edge, starts the cumulative gain at a value of M^3. That value is high enough to ensure that the heuristic will continue searching through large portions of the graph.

Step	Figure	Cumulative gain
Remove (Q, S)	Fig.A.5(b)	M^3
Add start edge (S, Y_i)	Fig.A.5(c)	$M^3 - M^2$
Reconfigure OR devices along rib for literal $\overline{x_i}$, removing some penalty edges.	Fig.A.5(d)	$M^3 - M^2 - o(M^2)$
Reconfigure OR devices along rib for literal x_i, possibly adding some penalty edges.	Fig.A.5(e)	$M^3 - M^2 + o(M^2)$
Remove start edge (Y_i, S)	Fig.A.5(f)	$M^3 + o(M^2)$
Add edge (S, P)	Fig.A.5(g)	$o(M^2)$
Remove edge (P, R)	Fig.A.5(h)	$M^3 + o(M^2)$
Add edge (R, Q) to form the new tour	Fig.A.5(i)	$o(M^2)$

Table A.1: Summary of steps in a Lin-Kernighan improvement on a standard tour. In this example, the value of x_i is flipped from false to true.

Third, the backtracking portion of the search forces the heuristic to consider removing each of the "start" edges. Each of those weigh only M^2, much less than M^3. Trying all the alternative start edges is how the Lin-Kernighan heuristic discovers the appropriate truth variable to flip. Fourth, all the edge exchanges involved in reconfiguring OR devices and adding and removing penalty edges fall into the $o(M^2)$ terms, either positive or negative. The configurations of the OR devices never stray from the three allowable ones shown in Figure A.2 parts (c), (d), and (e) because the diagonal edges in each OR device weigh $-M^4$, and *removing* them would make the cumulative gain *negative*. (That is, it is crucial that those edges have negative weight with magnitude much larger than M^3.) Finally, the lowest value the cumulative gain attains is the final net gain in step (i), *i.e.*, the improvement in the weight of the tour, and two steps earlier at (g). This last point becomes critical later on when we consider applying cluster compensation. All the other intermediate values of the cumulative gain are higher, somewhere in $\theta(M^3)$, even for the steps we have omitted from our sketch.

In summary, the weights of the edges are engineered so that the bait edges are heavy enough to force the appropriate long searches ($M^3 >> M^2$), but small enough to force the heuristic to stay within the desired OR device and tour configurations ($M^3 << M^4$).

This completes the sketch proof of Lemma A.3.

Lemma A.4 says that the only locally optimum tours are the standard tours. Papadimitriou shows how the stylized variant of Lin-Kernighan improves any non-standard tour so that it becomes more like a standard tour. Most of the arguments are not of interest to us, since they involve only 2-changes or 3-changes.

The main point of interest to us is that only one kind of Lin-Kernighan improvement is used in the argument. Suppose an exit node of an OR device O_1 is connected to the entry node of an OR device O_2 that does not follow O_1 in O_1's rib. Those undesirable edges are given weight M^4. Papadimitriou shows how a Lin-Kernighan improvement can be made, beginning with the removal of such an undesirable edge.

As in the case of Lemma A.3, the large M^4 weight of the undesirable edge drives the heuristic down the appropriate search path. The cumulative gains are always dominated by a positive M^4 term. The improvement is constructed so that the following conditions hold. First, the exit node of O_1 is connected by the 0-weight edge to the next OR device in its rib or to the Z_i node that ends the rib. That is, the undesirable M^4-weight edge is removed. Second, possibly other M^4-weight edges are removed, and no edges of weight M^4 or higher are added.

This concludes the proof sketch of Lemma A.4.

Repeatedly applying the argument of Lemma A.4 shows that an arbitrary tour in $f(C)$ is improved by Papadimitriou's stylized Lin-Kernighan until a standard tour results. Then, we can repeatedly apply the argument of Lemma A.3 to find a standard tour that is locally optimal for stylized Lin-Kernighan and which represents a truth assignment that is locally optimal for *2-SATFLIP*. This concludes the proof sketch that *TSP-LK* is PLS-hard.

A.5 *TSP-LKCC* is PLS-hard

This section describes how to adjust the construction given in the previous section to show that *TSP-LKCC* is PLS-hard. Our work here is relatively easy because Papadimitriou's construction is robust, the margins are comfortably large. First we show why we cannot just reuse the construction for *TSP-LK*, and then we describe a small patch to the construction so that it will work in the new case.

Let us write G for the graph built from the set of clauses by the f function in Papadimitriou's construction, *i.e.*, $G = f(C)$. To understand how cluster compensation affects the stylized Lin-Kernighan search, we must know the structure of the cluster distance function on G. More precisely, the cluster distance discount is forced to be non-negative, so we must understand the structure of

$$\overline{c}_G = \max(0, c_G).$$

Fortunately, the weights of the edges of G are such that $\overline{c}$ has very simple structure. First, each of the (a_i, b_i) edges and the edges forming the main arc of every rib have weight 0. Second, the diagonal edges inside each OR device, and the edges joining the rib pairs in sequence and to R and b_m are crazy glue edges, and thus have negative weight. The two edges going from P to Q through W are also crazy glue edges. We can thus span G with two trees, neither of which has positive weight edges. The first tree is the two-edge path from P to Q through W, and the other tree covers all the other vertices of G. Finally, the shortest edges from the $\{P, Q, W\}$ component to the rest of the graph have weight M^3: they are (R, P), (R, Q), (P, S), and (Q, S). We have thus proven the following lemma.

Lemma A.5 *Let $G = f(C) = (V, E)$ as in Papadimitriou's construction, and let $\overline{c}_G = \max(0, c_G)$. Then*

$$\overline{c}_G(u, v) = \begin{cases} 0 & \text{if } u, v \in \{P, Q, W\}, \\ 0 & \text{if } u, v \in V \setminus \{P, Q, W\}, \\ M^3 & \text{otherwise.} \end{cases}$$

Now consider the Lin-Kernighan search described in the proof of Lemma A.3. By construction, $t_1 = P$ or $t_1 = Q$, and also $t_2 = S$. If cluster compensation is used, then the search sequence is never begun. The first edge to be removed, say (Q, S) has weight M^3, but this is exactly the cluster distance between Q and S. (The other cases are similar.) So the cumulative gain discounted by the cluster distance would not be strictly positive after the first step!

The fix is simple. We adjust f to f' so that $G' = f'(C)$ is identical to $f(G)$ except that edges (R, P) and (R, Q) are each assigned weight $M^3/2$, down from M^3. We therefore have

$$\overline{c}_{G'}(u, v) = \begin{cases} 0 & \text{if } u, v \in \{P, Q, W\}, \\ 0 & \text{if } u, v \in V \setminus \{P, Q, W\}, \\ M^3/2 & \text{otherwise,} \end{cases}$$

and the search is not stopped before it begins. Recall that the main point about the edge weights were that the bait edge weights, M^3, were much larger than the other weights in

Step	Figure	Discounted cumulative gain
Remove (Q, S)	Fig.A.5(b)	$M^3/2$
Add start edge (S, Y_i)	Fig.A.5(c)	$M^3/2 - M^2$
Reconfigure OR devices along rib for literal $\overline{x_i}$, removing some penalty edges.	Fig.A.5(d)	$M^3/2 - M^2 - o(M^2)$
Reconfigure OR devices along rib for literal x_i, possibly adding some penalty edges.	Fig.A.5(e)	$M^3/2 - M^2 + o(M^2)$
Remove start edge (Y_i, S)	Fig.A.5(f)	$M^3/2 + o(M^2)$
Add edge (S, P)	Fig.A.5(g)	$o(M^2)$
Remove edge (P, R)	Fig.A.5(h)	$M^3/2 + o(M^2)$
Add edge (R, Q) to form the new tour	Fig.A.5(i)	$o(M^2)$

Table A.2: Summary of steps in Lin-Kernighan improvement on a standard tour in the new construction. Cumulative gains shown include the cluster distance discount. See also Figure A.5 and Table A.1

the skeleton (at most M^2), and much smaller than the non-skeleton edges (at least M^4). This arrangement guides the Lin-Kernighan search appropriately. In the new setting, the M^3 term is replaced by an $M^3/2$ term, but it is still in the right range to guide the search in exactly the same way. (Of course, it helps that the cluster discount computations are always anchored at $t_1 = Q$.) Table A.2 shows the cumulative gains for the same search, but in graph G' instead of G. A search beginning with $t_1 = P$ would be similar. We have thus proven the analog of Lemma A.3 in the new setting. That is, a standard tour T of $G' = f'(C)$ is locally optimal only if $g(T)$ is locally optimal.

The analog of Lemma A.4 would be: the only locally optimal tours are the standard tours. We can reuse the arguments from the proof of Lemma A.4 itself. The edges heavier than M^4 are removed by 2-changes and 3-changes. In particular, we can assume the non-standard edges to P, Q, and W have been removed. Cluster compensation only affects the Lin-Kernighan searches, and Lin-Kernighan searches are applied such that t_1 is an exit point of an OR device and the first edge removed has weight M^4. In that case, the dominant and therefore driving term in the cumulative gain is M^4. But the maximum value of $\overline{c}_{G'}$ is $M^3/2$, which is completely dominated by the M^4 term. These Lin-Kernighan searches therefore proceed as in Papadimitriou's argument. This completes the proof that the only locally optimal tours of G' are the standard tours.

Combining both lemmas for the *TSP-LKCC* setting, we have the following theorem.

Theorem A.6 *2-SATFLIP reduces to TSP-LKCC.*

Corollary A.7 *TSP-LKCC is PLS-hard.*

That is, stylized Lin-Kernighan employing cluster compensation solves a PLS-hard problem.

A.6 Summary

Since *TSP-LKCC* is both in PLS and PLS-hard, we conclude that *TSP-LKCC* is PLS-complete.

The PLS-completeness of *TSP-LKCC* is both good news and bad news. The *bad* news is that cluster compensation does not eliminate the very worst case behaviour of the Lin-Kernighan heuristic. In particular, a consequence of the PLS-completeness of *TSP-LKCC* is that there is a family of graphs for which stylized Lin-Kernighan with cluster compensation takes an exponential number of steps. The *good* news is that cluster compensation does not fundamentally reduce the power of the Lin-Kernighan heuristic, at least from a PLS-completeness perspective.

Appendix B

Where to get the code

The software described in this thesis is publicly available from `http://www.cs.utoronto.ca/~neto/research/lk`. (The World-Wide Web is always in flux; in case that web site no longer exists, search for a web page containing "`Efficient Cluster Compensation for Lin-Kernighan Heuristics`".)

The main programs use ANSI C together with a small number of BSD Unix extensions for measuring resource usage (`getrusage` and friends). Perl (version 5 or later) and GNU `make` are required in order to run some of the scripts and create some of the graphical output. To develop the program further, you will need the CWEB literate C programming toolset. All these tools are freely available and licensed under liberal terms. LK itself is licensed under the GNU General Public License or the GNU Library General Public License, as appropriate. Some of the scripts are in the public domain.

Program `lk` implements the Lin-Kernighan heuristic for both the Traveling Salesman Problem and minimum weight perfect matching. One chooses at compile-time whether the program should use cluster compensation. Program `lk` also implements an iterative approximation algorithm for the Held-Karp lower bound, specified via a command-line option.

The instance generators are also included in the package. Program `jitter` implements algorithm jitter. Program `shake` implements the tree shaking algorithm mst-shake. Algorithms mst-explode-construct and mst-dangle-construct are split: we use program `lk` with option `-M` to produce a minimum spanning tree which is then fed to program `tspgen` to generate the geometric instance. By default, `tspgen` performs algorithm mst-explode-construct; if given option `-i`, it performs algorithm mst-dangle-construct. Program `tspbgen.pl` is a Perl script that is able to generate all 11 distributions described by Bentley [17].

Programs `lk`, `jitter`, `shake`, and `tspgen` display help when given option `-h`. Since they are all written in literate C using the CWEB toolset, full documentation can be obtained by using CWEB's `cweave` to produce a TEX file and then typesetting the result with TEX. For convenience, a copy of `cwebmac.tex` is included without modification in LK.

Appendix C

Computing platform details

This appendix describes the computer used for the experiments. It is more specific than the description given in Section 5.2.

The experiments reported in this thesis are compute-bound, so we describe only those components relevant to compute-bound processes.

C.1 Hardware

- Intel Pentium II 300MHz central processing unit. It has a 512KB second-level cache running at 150MHz.
- Asus P2L97 motherboard, with an Intel 440LX chipset and a separate video bus. The front-side bus operates at 66MHz.
- 128MB SDRAM main memory with a 12ns cycle time. However, memory access time is limited by the 66MHz front-side bus.

C.2 Software

- Red Hat Linux 5.1
- Linux Kernel 2.0.35
- GNU C Library version 2.0.7
- GNU C Compiler version 2.7.2.3 with optimization level `-O2`.

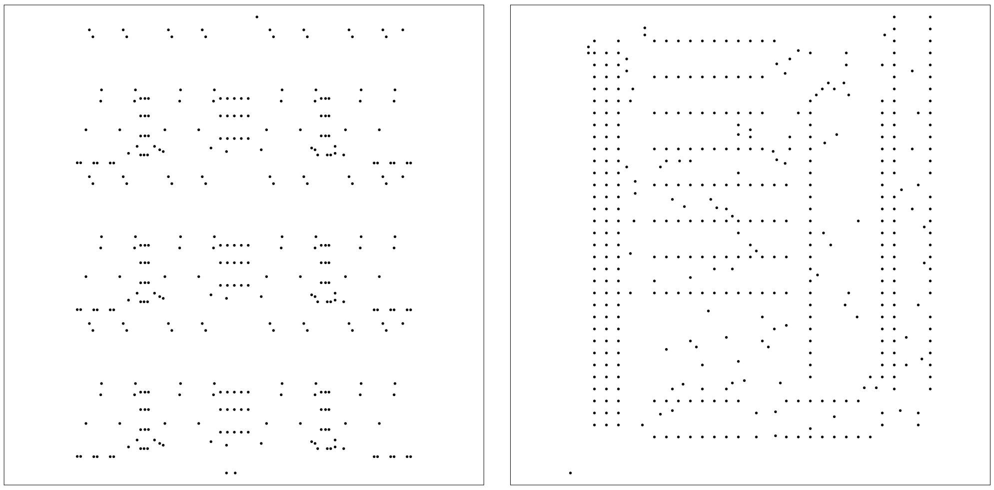
Appendix D

Depictions of geometric instances

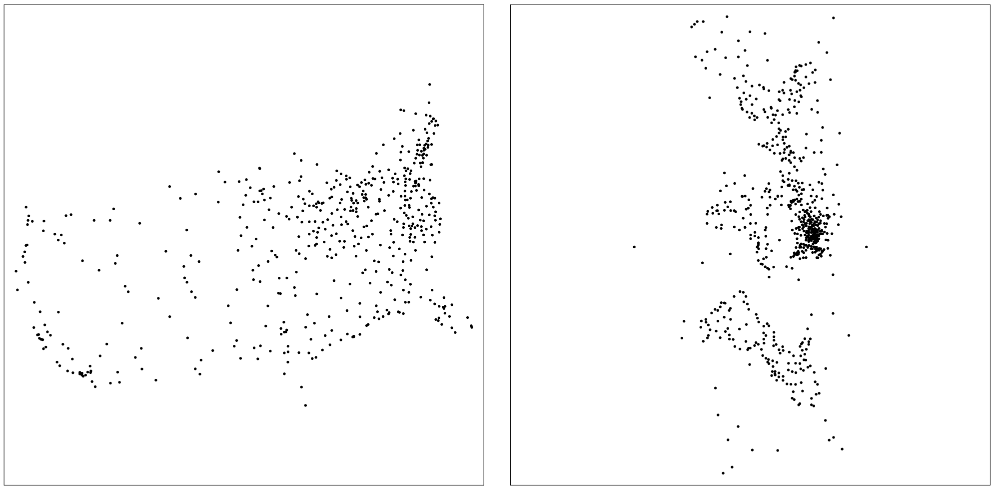
Many of the instances used in this thesis are geometric. This appendix depicts many of them.

Figures D.1 and D.2 show the TSPLIB instances. Note that instance `att532` uses the `ATT` cost function, a modified Euclidean metric. Instance `gr666` uses great circle distances, and we plot the latitude and longitude as ordinary y and x coordinates. All the other TSPLIB instances use Euclidean distance, rounding to either the nearest integer or rounding up.

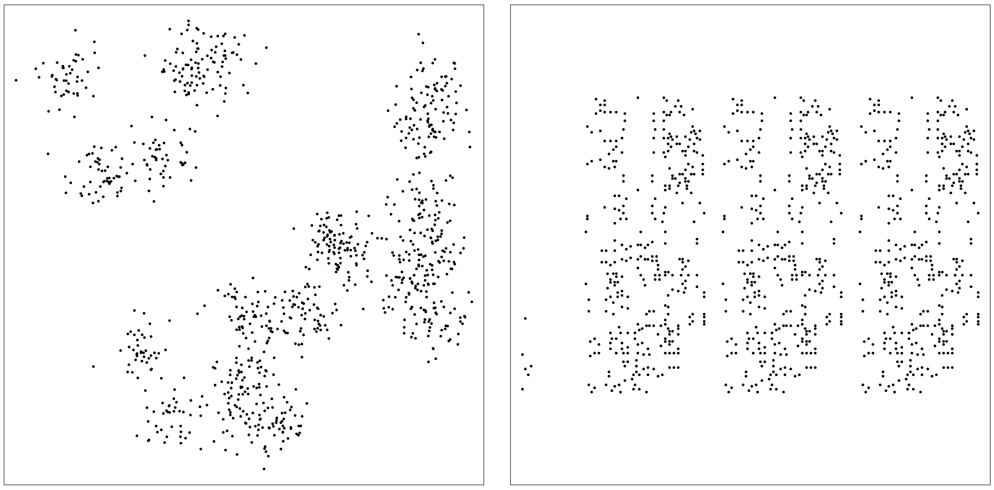
Figures D.3 and D.4 depict one sample from each of the 11 Bentley distributions. Each has 1000 vertices, except instance `arith.20` which has only 20 in order to show the structure of the arithmetic differences sequence.



(a) (b)

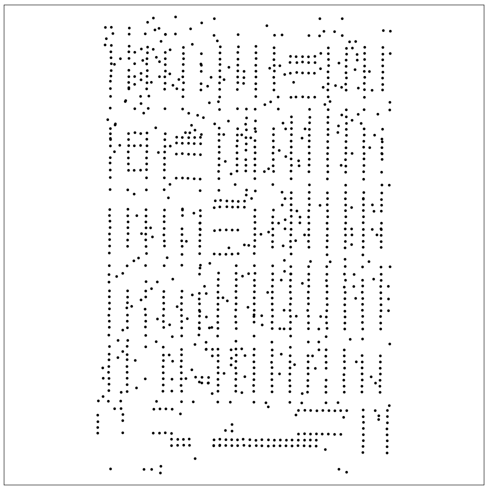


(c) (d)

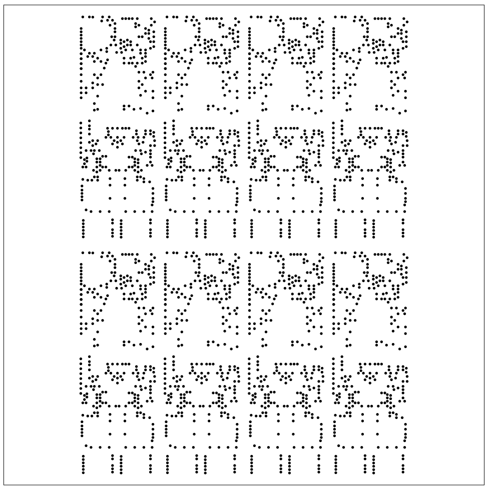


(e) (f)

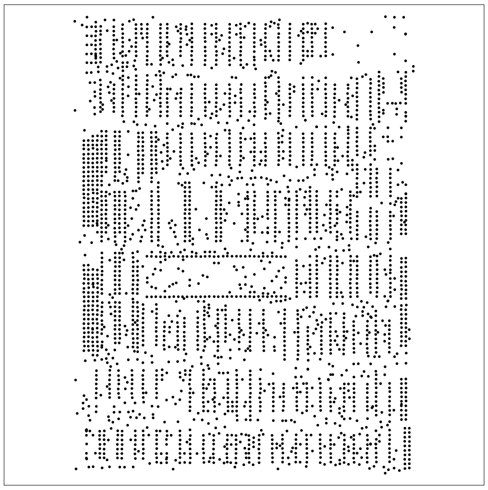
Figure D.1: TSPLIB instances: (a) `lin318`, (b) `pcb442`, (c) `att532`, (d) `gr666`, (e) `dsj1000`, (f) `pr1002`.



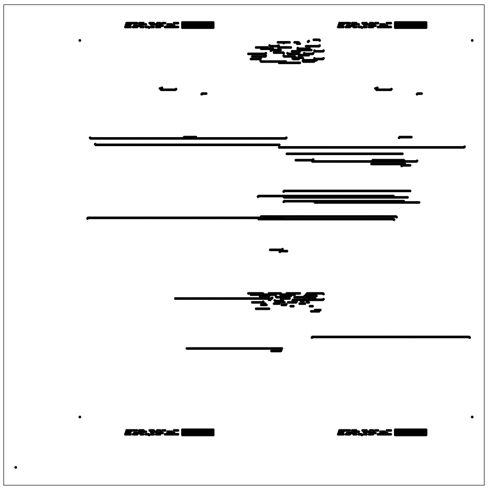
(a)



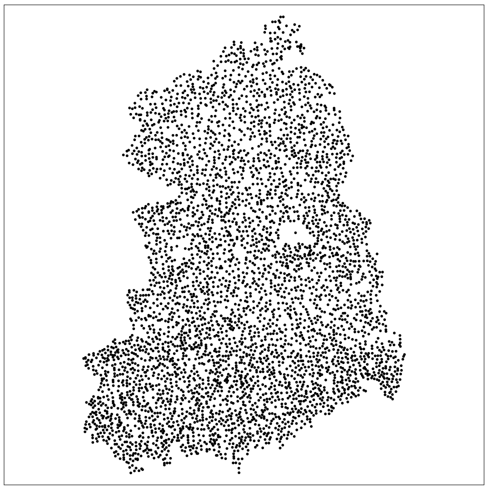
(b)



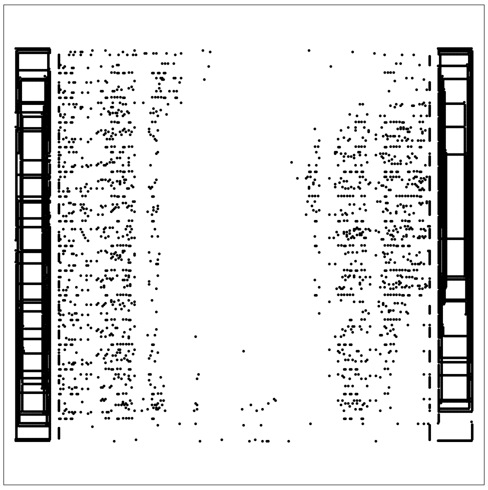
(c)



(d)



(e)



(f)

Figure D.2: TSPLIB instances: (a) `pcb1173`, (b) `pr2392`, (c) `pcb3038`, (d) `fl3795`, (e) `fnl4461`, (f) `pla7397`.

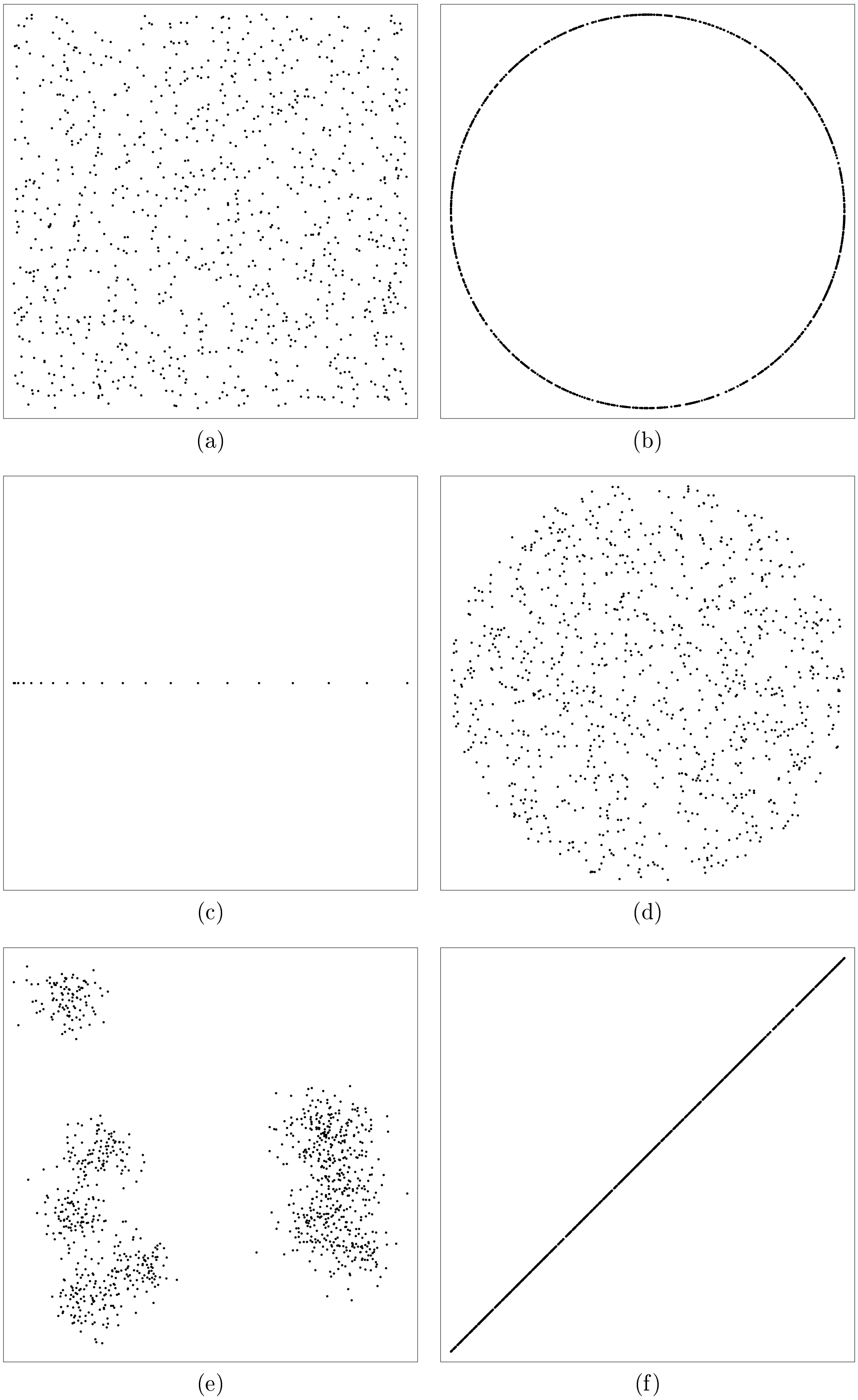


Figure D.3: Sample Bentley instances: (a) `uni.820.1000`, (b) `annulus.821.1000`, (c) `arith.20`, (d) `ball.823.1000`, (e) `clusnorm.824.1000`, (f) `cubediam.825.1000`.

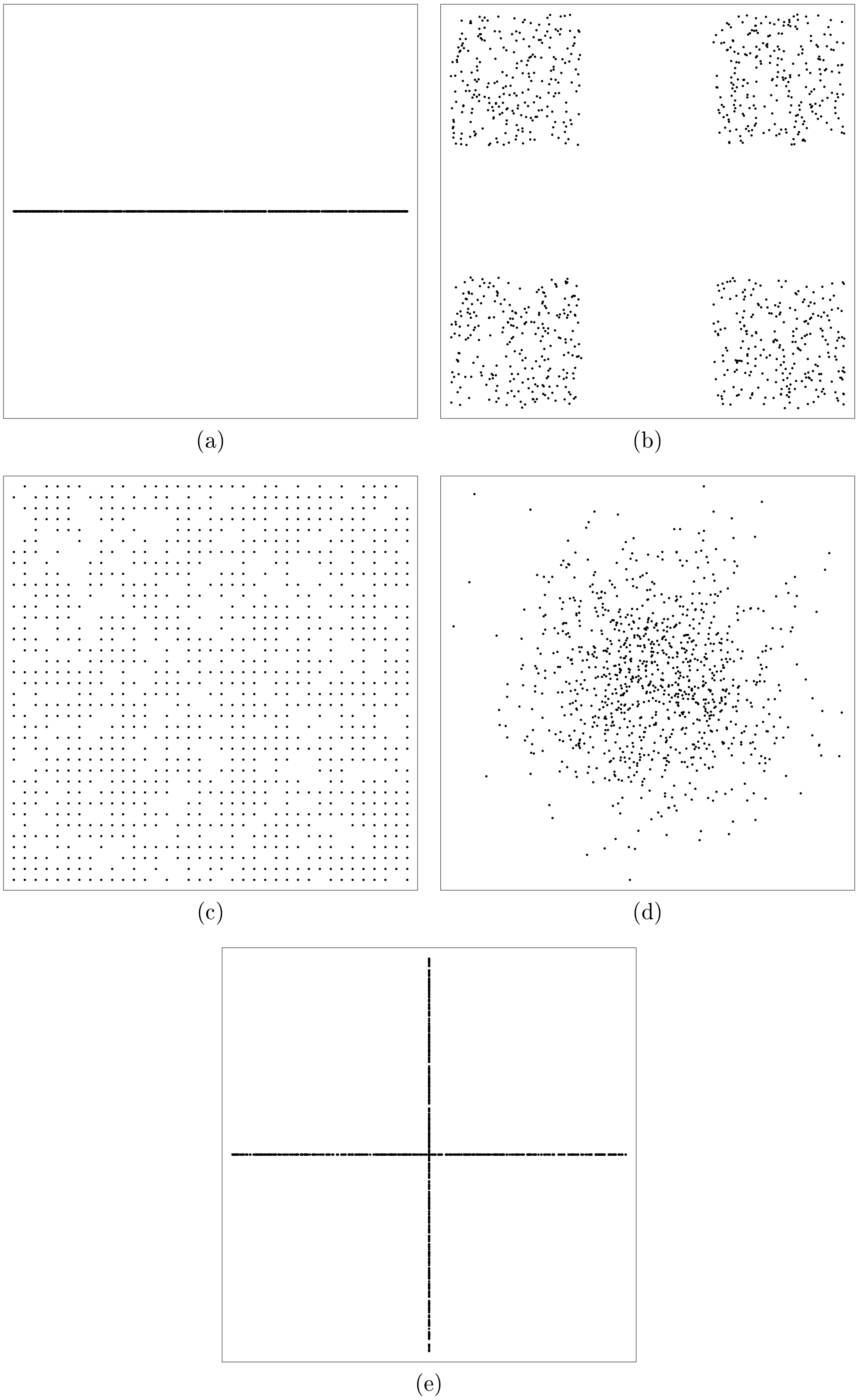


Figure D.4: Sample Bentley instances: (a) `cubeedge.826.1000`, (b) `corners.827.1000`, (c) `grid.828.1000`, (d) `normal.823.1000`, (e) `spokes.824.1000`.