Efficient cluster compensation for Lin-Kernighan heuristics

David M. Neto
University of Toronto
(currently at Altera Corporation)

June 2002

Abstract

This paper describes *cluster compensation*, a technique for adjusting the Lin-Kernighan heuristic for the Traveling Salesman Problem so that it is less prone to wasting effort in the presence of clustered inputs. We describe the essential features of the Lin-Kernighan heuristic, and argue how those features combine to force the heuristic to waste effort on clustered inputs. We then define cluster compensation, proposing it as a way to help Lin-Kernighan better estimate future costs in its gain function. We indicate how cluster compensation may be performed with low overhead. We provide experimental results over several classes of inputs. They show that showing that cluster compensation has very little impact on the quality of the tours produced. For geometric instances, cluster compensation reduces running time between 23 and 68 percent. For random distance matrix instances, cluster compensation does not reduce running time enough to cover its initialization costs, increasing overall running time by 14 percent on average.

Contents

1 Introduction

The Lin-Kernighan heuristic is a highly effective heuristic for finding low-cost tours for the Traveling Salesman Problem (TSP). For a survey of the problem and the undominated approaches to it, see [6] or [7].

Let us fix our terminology as follows. Let $G = (V, E, w)$ be a weighted complete graph where V is the set of "cities," E is the set of edges between cities, and w is weight function on edges. We will only consider the case where w is symmetric and positive-definite. Subgraph $T = (V, E_T)$ is a tour of G if it is a simple cycle in G that visits every vertex exactly once. The weight of tour T is the sum of the weights of its edges:

$$w(T) = \sum_{e \in E_T} w(e)$$

Given a graph G, the Traveling Salesman Problem asks us to find a *tour* of minimum weight.

The Lin-Kernighan heuristic is a local search algorithm, iteratively improving an initial tour. We describe its essential features in the next section. It has been known for some time that the heuristic can perform large amounts of unproductive work when it is presented with a clustered input [6]. Section 3 presents our working hypothesis as to why this is the case. Section 4 describes *cluster compensation*, which is our approach to avoiding the problem in the first place. The technique is centred on computing the *cluster distance* between two vertices. We indicate how cluster distance queries can be performed in constant time, given modest preprocessing time and extra space linear in the number of cities. Section 5 summarizes results from experiments with Lin-Kernighan for the TSP both with and without cluster compensation. The implementation is the one used in [9], which is based on the description of the details of the JBMR implementation of the Lin-Kernighan heuristic given in [6]. Section 7 briefly discusses other aspects of cluster compensation. First, we consider cluster compensation in the context of the Lin-Kernighan heuristic adapted for Minimum Weight Perfect Matching. Second, the section considers the PLS-completeness of the Lin-Kernighan TSP heuristic if cluster compensation is applied. The final section summarizes the results and give directions for further work.

2 Essential features of the Lin-Kernighan heuristic

Many variants of the Lin-Kernighan heuristic are possible. See [6] for a recent survey of implementation approaches. This section describes only the features of the heuristic that are essential to understanding the rest of this paper, *i.e.* why it can get stuck on clustered inputs, and the proposed solution. We will describe the heuristic in its standard context, finding good tours for the Traveling Salesman Problem. Later we will also show how it can be easily applied to the Minimum Weight Perfect Matching problem.

The Lin-Kernighan heuristic is a local search heuristic, *i.e.* it begins with a starting tour and iteratively improves the tour until some stopping criterion is met. One tour T in this sequence differs from the next tour T' in that T' is formed from T by applying a k-change: removing k edges and adding k edges (in some order). Tour T' is an improvement over T because the total weight of the removed edges exceeds the weight of the added edges.

The k-changes found by the heuristic are alternating cycles. Edges to be removed alternate with edges to be added, and each edge shares a vertex with the next edge in the cycle.[1] By convention, we express the k-change as a sequence of vertices $t_1, \ldots, t_{2k}$ where (t_{2j-1}, t_{2j}) is an edge to be removed from the tour and (t_{2j}, t_{2j+1}) is an edge to be added to the tour. Also, we add edge (t_{2k}, t_1), which we shall call the "closing up" edge. The reduction in the weight of the tour when applying a k-change is defined as:

$$net\_gain(k) = \sum_{j=1,\ldots,k} w(t_{2j-1}, t_{2j}) - \sum_{j=1,\ldots,k-1} w(t_{2j}, t_{2j+1}) - w(t_{2k}, t_1).$$

So a k-change is an improvement if and only if $net\_gain(k) > 0$. We also define the cumulative gain, which is the net gain but where we ignore the closing up edge:

$$cum\_gain(k) = \sum_{j=1,\ldots,k} w(t_{2j-1}, t_{2j}) - \sum_{j=1,\ldots,k-1} w(t_{2j}, t_{2j+1}).$$

The heuristic finds improving alternating cycles incrementally. First it proposes a 2-, 3-, or 4-change, which specifies t_1 through t_4, or t_6, or t_8.

[1] This is certainly true if you use both the tabu rules specified by Lin and Kernighan, namely "never delete an added edge" and "never add a deleted edge". However, when you drop one of the rules, as done by Johnson *et al.* [6] and Papadimitriou [10], then you have to be more careful. In those cases "removing" an edge means removing it from the intermediate tour formed by applying the intermediate $(k-1)$-change, and "adding" an edge means adding it to the intermediate tour formed by applying the intermediate $(k-1)$-change.

Thereafter the t sequence is extended by pairs of cities, making a $(k-1)$-change into a k-change. That is, the t sequence is extended by proposing a new edge (t_{2k-2}, t_{2k-1}) to be removed (along with the intermediate closing up edge (t_{2k-2}, t_1)) and a new edge (t_{2k-1}, t_{2k}) to be added (along with the new intermediate closing up edge (t_{2k}, t_1)). The extension must satisfy certain feasibility criteria:

- The result of applying the k-change must still be a tour.
- We never delete an already-added edge.
- We never add an already-deleted edge.

Furthermore, Lin and Kernighan [8] show that we can safely prune our search for an improving k-change by insisting that all the intermediate cumulative gains are positive. That is, we consider a $(k+1)$-change if and only if $cum\_gain(i) > 0$ for all $i \leq k+1$. Note that they deliberately ignore the closing-up edge in this criterion. They observe that forcing the *net* gains to always be positive would sometimes prevent the heuristic from finding a change that looks bad in the short run but is good in the long run. They prune the search even more by insisting that the new cumulative gain should be at least as good as the best net gain seen so far:

$$cum\_gain(k+1) \geq \max\{net\_gain(i) \mid i \leq k\} \tag{1}$$

Given these conditions, there may be feasible ways to extend the t sequence. To choose among them, Lin-Kernighan uses a greedy selection criterion:

$$\text{Choose } t_{2k+1}, t_{2k+2} \text{ to maximize } cum\_gain(k+1) \tag{2}$$

There are many other details that go into a good implementation of the heuristic. Those details include:

- (Backtracking) How much backtracking should be done. Typically one backtracks over the choices for the initial 2-, 3-, or 4-changes.
- (Candidates lists) Given vertex t_{2k-2}, what vertices are considered for t_{2k-1}? Typically t_{2k-1} would be chosen to be "near" to t_{2k-2}, *e.g.* one of its 20 nearest neighbours.
- (Initial tour) How do we pick the initial tour in the search? Lin-Kernighan is quite robust in that the final tour quality is not very sensitive to the initial tour, but one typically uses a tour constructed by some greedy algorithm. [6].

However, we already have enough of a specification of the heuristic to be able to propose a reason for its poor behaviour on clustered instances.

3 How Lin-Kernighan can get trapped

First let us describe what we mean by a clustered instance. An instance G is clustered if you can partition its vertices into clusters so that:

- Between any pair of vertices in the same cluster, you can find a path between them in which each edge is relatively short, *i.e.* has low weight.
- Any path between two vertices from different clusters must necessarily use an edge that is relatively long, *i.e.* has high weight.

Of course, this is a fuzzy definition, as the "sharpness" of the clusteredness can vary. An instance is sharply clustered if the "long" edges are much longer than the "short" edges.

We propose that the key to understanding why Lin-Kernighan gets trapped on clustered instances is that its cumulative gain criterion always ignores the closing-up costs. Let us consider what happens with the cumulative gain criterion and the greedy selection criterion when the heuristic is presented with a clustered instance. At some point the heuristic considers removing an edge that spans two clusters. That edge is "long" by definition, and so removing it looks very attractive to both the feasibility check and the greedy selection criterion. So it is likely that that is how the t sequence is extended.

However, choosing to remove that long inter-cluster edge has two effects. First, the excess of the cumulative gain over the best net gain so far becomes very large. You can think of this excess as a budget of losses that can be spent doing bad moves. When you remove a long edge, the budget becomes big and you can search for a long time before having to give up. Second, crossing back into the cluster containing t_1 is never attractive. Doing so would mean adding a long inter-cluster edge. That is almost always less attractive than just staying in the other cluster. The combination of the two effects can be devastating. The heuristic wants to spend a long time in the other cluster, and it is not likely that it will find an improvement by staying there.

If ignoring the closing-up costs really is the problem, then the solution would be to give the heuristic better visibility of closing up costs. Lin and Kernighan rightly rejected counting the *immediate* closing up cost, but perhaps we can get an estimate on *future* closing up costs. That is the idea behind cluster compensation.

4 Cluster compensation

Cluster compensation is a technique for estimating closing up costs, and for including that estimate in both the greedy selection criterion and in the cumulative gain criterion.

First, let us define the cluster distance between two vertices as follows. For a path P, let the *bottleneck cost of P* be the length of the longest edge in P:

$$\mathit{bottleneck}(P) = \max\{w(e) \mid e \in P\}$$

Then the *cluster distance between vertices u and v* is the minimum bottleneck cost for any path between u and v:

$$w_c(u,v) = \min\{\mathit{bottleneck}(P) \mid \text{path } P \text{ goes from } u \text{ to } v\ \}$$

This corresponds to our intution of clusteredness because the cluster distance between two vertices in the same cluster is "short", but the cluster distance between vertices in different clusters is "long".

Two elementary facts follow. The first one is important to understanding the functioning of cluster compensation, and the second one is only mentioned to aid the reader's intution. The first fact is that cluster distance between two vertices is bounded from below by 0 and from above by the actual distance between the two vertices:

$$0 \leq w_c(u,v) \leq w(u,v)$$

This lower bound holds because w is positive-definite, and the upper bound holds because u, v is one of the possible paths from u to v because G is a complete graph. The second fact is that w_c is a metric if w is symmetric and positive-definite, as assumed. The proof of this fact is less trivial, but still easy.

The cluster compensation technique modifies both the cumulative gain criterion and the greedy selection criterion to incorporate the cluster distance between the proposed t_{2k+2} and t_1. That is, the cumulative gain criterion becomes:

$$\mathit{cum\_gain}(k+1) - w_c(t_{2k+2}, t_1) \geq \max\{\mathit{net\_gain}(i) \mid i \leq k\} \qquad (3)$$

and the greedy selection criterion becomes:

$$\text{Choose } t_{2k+1}, t_{2k+2} \text{ to maximize } \mathit{cum\_gain}(k+1) - w_c(t_{2k+2}, t_1) \qquad (4)$$

Compare these with statements 1 and 2, respectively.

The bounds on w_c imply the following bounds:

$$\mathit{cum\_gain}(k) \leq \mathit{cum\_gain}(k) - w_c(t_{2k}, t_1) \leq \mathit{net\_gain}(k)$$

That is, cluster compensation at each step makes the heuristic more conservative than baseline Lin-Kernighan, and less conservative than the rejected alternative of always including the immediate closing up cost.

4.1 Cluster distance as an estimate of closing-up costs

This section gives some intuition as to how the cluster distance can be a good estimate for future closing-up costs.

Suppose the Lin-Kernighan heuristic has an intermediate k-change in hand, *i.e.* a sequence $t_1, \ldots, t_{2k}$ of vertices. It considers ways to extend this sequence so that it *eventually* gets from t_{2k} back to t_1. Consider a generic closing-up path $t_{2k}, t_{2k+1}, \ldots, t_1$. It is always an odd-length path alternating between edges to remove and edges to add, beginning with an edge to be added. In particular, there is one more edge to be added than there is to be removed.

Consider the very special case where all the edges on the closing-up path are of equal weight W, and that this is a good closing-up path in the sense that there are no short-cuts (so the cluster distance between any pair of vertices on the path is also W). In this case, cancellation makes the closing up cost for the entire path equal to the weight of just one added edge, *i.e.* W. Furthermore, the cluster distance happens to be equal to W. (Because we chose it to be so!)

Of course, this generalizes slightly to the case where the weights on the path are not necessarily all equal, but that the weights on all but the longest edge cancel out.

With suitable doses of imagination, we can generalize a little further to the case where the weight of the longest edge merely dominates the alternating sum of weights of the rest of the edges. Assuming the closing-up path is a good one, then in the interesting cases the longest edge is an edge to be added, and it will have weight similar to the cluster distance between t_{2k} and t_1. (The less interesting case is where the longest edge on the closing up path is an edge to be *removed*. But these cases are very easy for the heuristic to find and optimize away. Usually just a greedy 2-change will find it, and so such cases are eliminated early in the execution of the heuristic.)

This section has given some intuition as to how cluster distance can be a good estimate for future closing up costs. The scenarios constructed cover some of the more interesting cases encountered during a run on sharply clustered instances.

4.2 Computing cluster distance efficiently

Cluster compensation modifies the inner loop of the Lin-Kernighan heuristic. We must be assured that it incurs little overhead. This section indicates how cluster distances between arbitrary pairs of vertices can be computed efficiently, given modest preprocessing. We refer the reader to [9] for the full details.

Two ideas are instrumental to computing cluster distances quickly. The first key idea is that any minimum spanning tree for G encodes enough information to compute minimum bottleneck costs. Let M be a minimum spanning tree for G. Then

$$\begin{aligned} w_c(u,v) &= \min\{\mathit{bottleneck}(P) \mid \text{path } P \text{ goes from } u \text{ to } v\} \\ &= \min\{\mathit{bottleneck}(P) \mid P \text{ is a path in } M \text{ from } u \text{ to } v\} \end{aligned}$$

This can be proven by an argument similar to the one used to prove the correctness of Kruskal's algorithm for computing a minimum spanning tree. Essentially, if paths in M are insufficient to compute minimum bottleneck costs over all paths in G, then Kruskal's algorithm would make a mistake.

The second key idea exploits the minimum spanning tree. Because the minimum spanning tree M is a tree, there is exactly one path in M between any pair of vertices. Modulo ties, the bottleneck cost is achieved by the heaviest edge on that unique path. A preprocessing step must find a minimum spanning tree M and then arrange the data encoding M so that it is easy to find the heaviest edges on the path in M from u to v.

From M we construct its topology tree, $\mathit{Top}(M)$, which is related to the execution history of Kruskal's algorithm with M as input. (In particular, M does not need to have been *produced* by Kruskal's algorithm.) Tree

$Top(M)$ is a rooted binary tree of roughly twice the size of M itself. The leaves of $Top(M)$ are all the vertices V of G (and of M). The internal nodes of $Top(M)$ are the edges of M. For the edges of $Top(M)$ we must consider the execution history of Kruskal's algorithm on M. Kruskal's algorithm begins with a forest of $|V|$ connected components, each with one vertex in it and nothing else. It then considers each edge e in non-decreasing weight order. It adds edge e to the forest if doing so does not create any cycles. Adding an edge merges two connected components into one. Constructing the topology tree is almost exactly the same, except that we don't throw away information. In particular, the forest of connected components maintained by Kruskal's algorithm is represented in our construction by a forest of binary trees that become subtrees of $Top(M)$. That is, when Kruskal's algorithm adds edge $e = (u, v)$ to the forest, we add e as an internal *node* in $Top(M)$. The two children of e in $Top(M)$ are the roots of the subtrees representing the components containing the leaves u and v. Some minor technical issues are required to show that constructing $Top(M)$ from M takes asymptotically no more time and space than running Kruskal's algorithm itself on M [9].

The topology tree $Top(M)$ is just the encoding we need to quickly compute cluster distances. That's because finding the edge achieving the cluster distance $w_c(u, v)$ between u and v is just a least common ancestor computation in the topology tree $Top(M)$. Given $O(|V| \log |V|)$ preprocessing time and $O(|V|)$ space, least common ancestor queries in trees may be computed in constant time. On how to do this, see for example Schieber [11].

In summary, the preprocessing overhead for cluster compensation is construction of a minimum spanning tree M, construction of $Top(M)$ from M, and finally preprocessing $Top(M)$ for fast least common ancestor queries. Altogether, it is the cost of computing M plus $O(|V| \log |V|)$ time (most of which is sorting) and $O(|V|)$ space. The cluster distance queries themselves are constant time queries into a $O(|V|)$ size data structure.

5 Results for the TSP

This section presents experimental results for an implementation of the Lin-Kernighan heuristic for the TSP, comparing the heuristic compiled both with and without cluster compensation.

We describe the results for two broad classes of instances: geometric instances, and random distance matrices. The instances include most of the instances from the recent DIMACs Challenge on the TSP [5]. A small handful of instances from that suite were excluded because our parser does not know how to parse those intance types.

We report the quality of the tour found as the percent excess over the Held-Karp bound as distributed with the DIMACS data. The running times include all preprocessing and optimization time, but exclude the time to read the instance.

5.1 Implementation details

The implementation is based on the description of the JBMR implemenation as given by Johnson and McGeoch [6]. The rest of this section lists the more important details.

For non-geometric instances, the candidate lists are the 20 nearest neighbours. (That is, the choice of t_{2k+1} is taken from a pre-computed list of the 20 nearest neighbours of t_{2k}.) For 2-D geometric instances we used two kinds of candidate lists. In the first option we used just the 20 nearest neighbours as in the non-geometric case. In the second option the candidate list for a given city is the union of the 20 nearest neighbours together with the 5 nearest neighbours in each of the four quadrants surrounding that city (northeast, northwest, southwest, and southeast). This addition significantly affects only those cities that are at the boundary of a cluster, giving them more opportunity to add a new edge to cross the intercluster gap. We did this to get better tours for both variants of the heuristic. The conclusions regarding running times are not greatly affected by this choice. That is, the ratios of running times comparing the base heuristic with the cluster compensation are not very different.

The implementation uses a mechanism equivalent to Bentley's don't-look bits [1]. This helps focus the heuristic on the parts of the tour that are as-yet unexplored or that have changed recently.

Given a starting vertex t_1, we backtrack over all feasible 2-, 3-, and 4-changes starting at t_1, subject to the candidate lists. If beginning the k change with a particular 2-, 3-, or 4-change finds a tour improvement, then we go on to the next choice for t_1. Otherwise we backtrack and try the next 2-, 3-, or 4-change starting with the same t_1. With candidate lists of 20 nearest neighbours, we can attempt up to roughly 800 different $\{2, 3, 4\}$-changes from the same starting city. Thereafter we use the two criteria to search greedily without backtracking at those greater depths.

We use the same tabu rules as JBMR, namely "never delete an added edge". This is sufficient to bound the number of edges to be removed by $|V|$. However, we also use a rule that limits the absolute depth of the search. After the initial $\{2, 3, 4\}$-change, we allow ourselves to remove at most 50 more edges. (So the most extensive change possible is a 54-change, encoded as $t_1, \ldots, t_{108}$.) We found that longer probes were rarely worth the effort.

The initial tour was computed by a Greedy algorithm (this is Johnson and McGeoch's name, while Bentley calls it the Multi-fragment heuristic). It provides a reasonably good tour within reasonable time bounds. The Greedy algorithm operates much like Kruskal's algorithm, but adapted to building tours. That is, begin with no edges, then consider the edges of the graph in non-decreasing weight order. Add an edge if it doesn't create a vertex of degree more than two, and if doesn't create a cycle of less than $|V|$ edges. As with JBMR, we allow for variation by randomizing the edge list slightly. At each step we pick the smallest edge next with probability 2/3, and then the next smallest edge next with probability 1/3.

It is important to use good data structures. For instances of 1000 cities or less, we use an array representation of the tour, and for more than 1000 cities we use a two-level tree [3]. For nearest neighbour queries on 2-D instances, we use Bentley's 2-D trees for semi-dynamic point sets [2] [1]. We use an ordinary array to store the tabu lists (the lists of edges we are forbidden from deleting). We found that fancier data structures did not pay for their complexity when the search sequences were bounded at t_{108}.

5.2 Computing platform

The experiments were run on an ordinary workstation with a 1GHz Pentium III processor and 2GB of RAM. Appendix B provides the speed calibration data required by the DIMACS Challenge methodology.

The operating system was Windows NT version 4 Service Pack 6. We used the MinGW edition of version 2.95 of the GCC compiler. (When we tried the Cygwin `glibc` C library implementation, allocations of 64MB or more would always fail. The MinGW adaptation of GCC allows GCC-compiled code to interface directly with the native Microsoft C language runtime, `msvcrt.dll`. In particular, when copmiling with MinGW, our code had no problems allocating memory.)

5.3 Results for DIMACS Challenge instances

The DIMACS Challenge instances fall into two main categories: geometric instances, and random distance matrices.

A geometric instance of the TSP is one in which each vertex has a position in some space, and the distance between two vertices is just the Euclidean distance between them. Other norms may be considered, but are not treated here. We present the results for both kinds of candidate sets: 20 nearest neighbours, and 20 nearest neighbours with 5 quadrant neighbours in each direction.

A random distance matrix instance consists of a symmetric matrix of random distances. Although probably not of practical value, these are useful for exhibiting extereme behaviour in heuristics.

Table 1 summarizes the geometric means from experiments where the candidate set is the 20 nearest neighbours. Each row displays the results for one class of inputs: TSPLIB, Uniform Euclidean, Clustered Euclidean, or Random Distance Matrix. Columns two through four show tour quality, and columns five through seven show running times. Columns two and three summarize the tour quality produced by the base heuristic ("LK base") and by the heuristic with cluster compensation ("LK cc"). The units are given in percent excess over the Held-Karp bound. Column four is the "LK cc" percentage number divided by the "LK base" percentage figure. A number greater than 1 means the cluster compensation variant produced a worse tour, and a number lower than 1 means cluster compensation produced a better tour. Columns five and six show the geometric mean of running times in seconds on the experimental platform. Column seven is the ratio of the running time of the cluster compensation variant to the running time of the base heuristic. A value of greater than 1 means cluster compensation spent more time than the base heuristic. Conversely, a value of less than one means cluster compensation took less time than the base heuristic.

Instance class	LK base % over HK	LK cc % over HK	LK cc % / LK base %	LK base seconds	LK cc seconds	LK cc / LK base
TSPLIB	2.58	2.63	1.02	4.26	2.56	0.60
Uniform Euclidean	1.88	1.92	1.02	2.41	1.72	0.71
Clustered Euclidean	5.23	5.37	1.03	51.25	20.78	0.41
Random Distance Matrix	3.67	3.62	0.98	0.44	0.50	1.14

Table 1: Summary of geometric means from experiments using the candidate set consisting of the 20 nearest neighbours. "LK base" is the base heuristic, whereas "LK cc" is the variant using cluster compensation.

Instance class	LK base % over HK	LK cc % over HK	LK cc % / LK base %	LK base seconds	LK cc seconds	LK cc / LK base
TSPLIB	2.29	2.33	1.01	4.98	2.59	0.52
Uniform Euclidean	1.88	1.94	1.03	3.55	2.73	0.77
Clustered Euclidean	3.09	3.14	1.02	68.05	21.48	0.32

Table 2: Summary of geometric means from experiments using the candidate set consisting of the 20 nearest neighbours together with the 5 quadrant neighbours in each direction.

Table 2 summarizes the geometric means from experiments where the candidate set is the 20 nearest neighbours together with the 5 quadrant neighbours in each direction. The column definitions are the same as for Table 1. There is no row for the Random Distance Matrix class because quadrant neighbours are defined only for geometric instances. This candidate set does not apply to that class of instances.

The detailed results for each instance class are given in Appendix A. Each row of those tables presents the results for a single instance. The values are the arithmetic average over five runs, where the random number generator seeds are drawn from $(2002, 2003, 2004, 2005, 2006)$. Tables 3 through 6 present the results for TSPLIB instances. Tables 7 through 11 present the results for Uniform Euclidean instances. Tables 12 through 15 present the results for Clustered Euclidean instances. Finally, Tables 16 through 17 present the results for Random Distance Matrix instances. Each detailed results table presents the geometric average for the values in its columns. Summary tables 1 and 2 were constructed from those geometric means.

For very large Euclidean instances, the overhead of computing the candidate sets dominates the rest of the running time. Obviously, that part of our implementation is not well-tuned. We therefore present Table 11, in which running times include only the "interesting" parts. That is, for the base case the running time includes only the LK local search phase, and for the cluster compensation case, it includes the LK optimization phase and the time to build the MST and the topology tree as required by cluster compensation.

Typical memory use with the 20 nearest neighbour candidate for runs on the largest instances were as follows. For the E3M.0 instance, the base heuristic allocated 643MB, and the cluster compensation variant allocated 808MB. For the C100k.0 instance, the base heuristic allocated 21MB, and the cluster compensation variant allocated 26MB. For the M10k.0 instance, the base heuristic allocated 407MB, and the cluster compensation variant also allocated 407MB.

5.4 Discussion of summary results

The summary tables show that the quality of the tours produced by both variants are very similar. The ratio of the percentages over Held-Karp are very close to 1.

As for running times, the results differ between the geometric instances and the general matrix instances.

Let us consider geometric intances first. For those classes of instances, cluster compensation reduces overall running times between 23% and 68%. Generally, TSPLIB instances are more sharply clustered than Uniform Euclidean instances, and Clustered Instances are more sharply clustered than TSPLIB instances. The ratios of running times show that as instances get more clustered, the running time advantage due to cluster compensation increases.

For random distance matrix instances, the story is different. Cluster compensation increases running time in those cases. First, running times are short because the the greedy algorithm finds a very good initial tour, so the optimization phase has little work to do. Cluster compensation is trying to cut from the optimization phase, which in these cases is a smaller portion of the overall runtime. Second, for distance matrix instances the overhead for cluster compensation is proportionally more expensive, in particular the task of finding a minimum spanning tree. Computing a MST in a general distance matrix takes time proportional to the number of edges in the graph, which is quadratic in the number of cities. This is a much greater proportional overhead than for Euclidean instances where k-d trees provide fast proximity lookups. Table 18 presents only the "interesting" parts of running time, *i.e.* only the LK optimization phase for the base case, or the LK optimization phase and the cluster compensation preprocessing overhead for the cluster compensation case. The trend shown by that table is that cluster compensation overhead grows as the size of the Random Distance Matrix instance grows. (For example, in one run of the `M10k.0` instance, the time to compute the MST was 3.2 seconds, and the time for the LK optimization phase itself was only 6.2 seconds.) Taking both factors together, *i.e.* the high quality of the initial tour and the increasing costs of computing minimum spanning trees, cluster compensation does not pay for its own overhead on these instances.

6 Sample probe depths and move depths

The previous section discussed measurements of running time and tour quality. This section considers the question of whether the measured differences in those statistics are due to cluster compensation forcing the

Lin-Kernighan heuristic to behave as intended, or whether they are due to some other artifact of the algorithm. Cluster compensation is supposed to prune poor search sequences while still finding good ones. We provide some direct anecdotal evidence that cluster compensation indeed does have the intended pruning effect, and that it correlates with running time improvements. To measure the pruning effect, we define two statistics for every t_i search sequence, the probe depth and the move depth.

The first statistic is the *probe depth*, which is the depth to which the t sequence grows. For example, a sequence that reaches t_{108} has a probe depth of 108. (Note that 108 is the maximum probe depth allowed under our normal parameters: a 4-change followed by up to 50 more edge swaps.)

If the search along a t sequence of cities results in a successful k-change to which the heuristic commits, then we say that that t sequence has a *move depth* of $2k$. The vertices involved in the k-change are t_1 through t_{2k}. If the search along a t sequences does not find an improvement, then the move depth for that sequence is 0.

The probe depth is always an upper bound on the move depth. The probe depth is an indication of the total amount of work expended, whereas the move depth indicates the total amount of *useful* work expended. In any search sequence that actually finds an improving k-change, any probing beyond t_{2k} is wasted effort, although the heuristic does not know it until it exhausts the possibilities.

(For both probe and move depth statistics, we count each new sequence beyond the initial 2-, 3-, or 4-change as a new sequence. That appears to be the only fair way to account for the full backtracking performed over the set of initial 2-, 3-, or 4-changes.)

Figure 1 shows histograms for both move depths and probe depths for one run of each variant of the heuristic on instance `E100k.0`. The top half of the figure shows that cluster compensation reduces the total probing work: its histogram stays below the histogram for the base heuristic. Note that the biggest difference is at the smaller depths, and that neither variant spends much effort doing very deep probes. At the same time, the useful work as measured by the move depth histograms, is roughly the same between the two variants. That tells us that cluster compensation really is pruning bad searches and not cutting off good ones. These histograms correlate well with the fact that for this instance, both variants of the heuristic produces similar quality tours, and that cluster compensation reduces running time by about 20%, which is at the lower range of its benefit for geometric instances.

Figure 2 shows probe and move depth histograms for the `dsj1000` instance. This instance was specifically constructed as a clustered instance, and was cited as known to be a greater challenge for the Lin-Kernighan heuristic. There is a striking difference between the probe depth histograms for the two variants of the heuristic. The base heuristic spends a lot of effort probing very deeply. Indeed the spike at 106 and 108 is caused by the large number of probes that are forcibly stopped at 50 flips beyond the initial 3- or 4-changes over which we are backtracking. In contrast, when cluster compensation is used, very few probes explore beyond t_40. Meanwhile, both variants have similar move depth histograms for this instance. Cluster compensation has managed to prune away large and

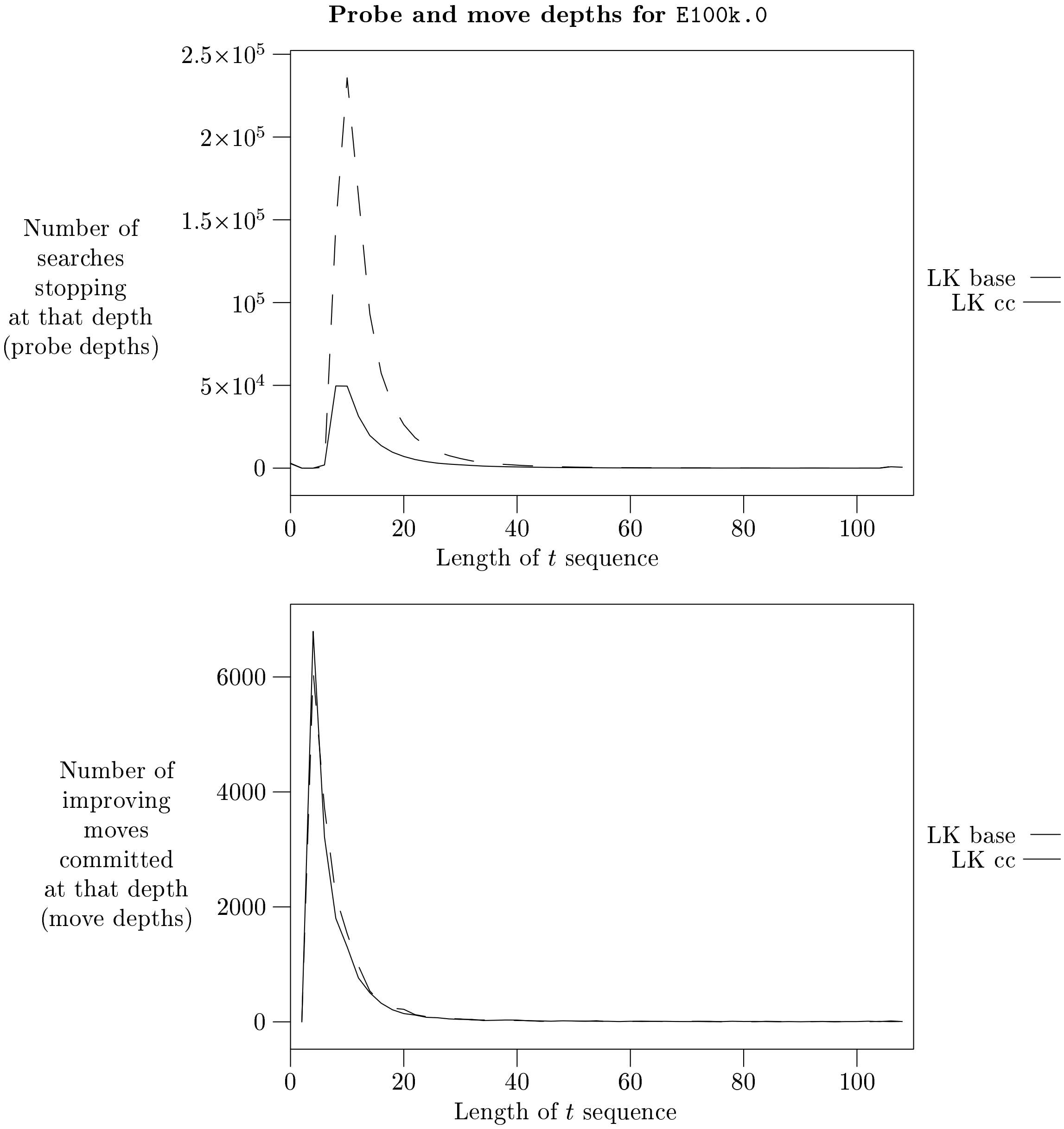

Figure 1: Comparison of probe depths and move depths for instance E100k.0, using candidate set of 20 nearest neighbours and 5 quadrant neighbours in each direction, and using seed 2002.

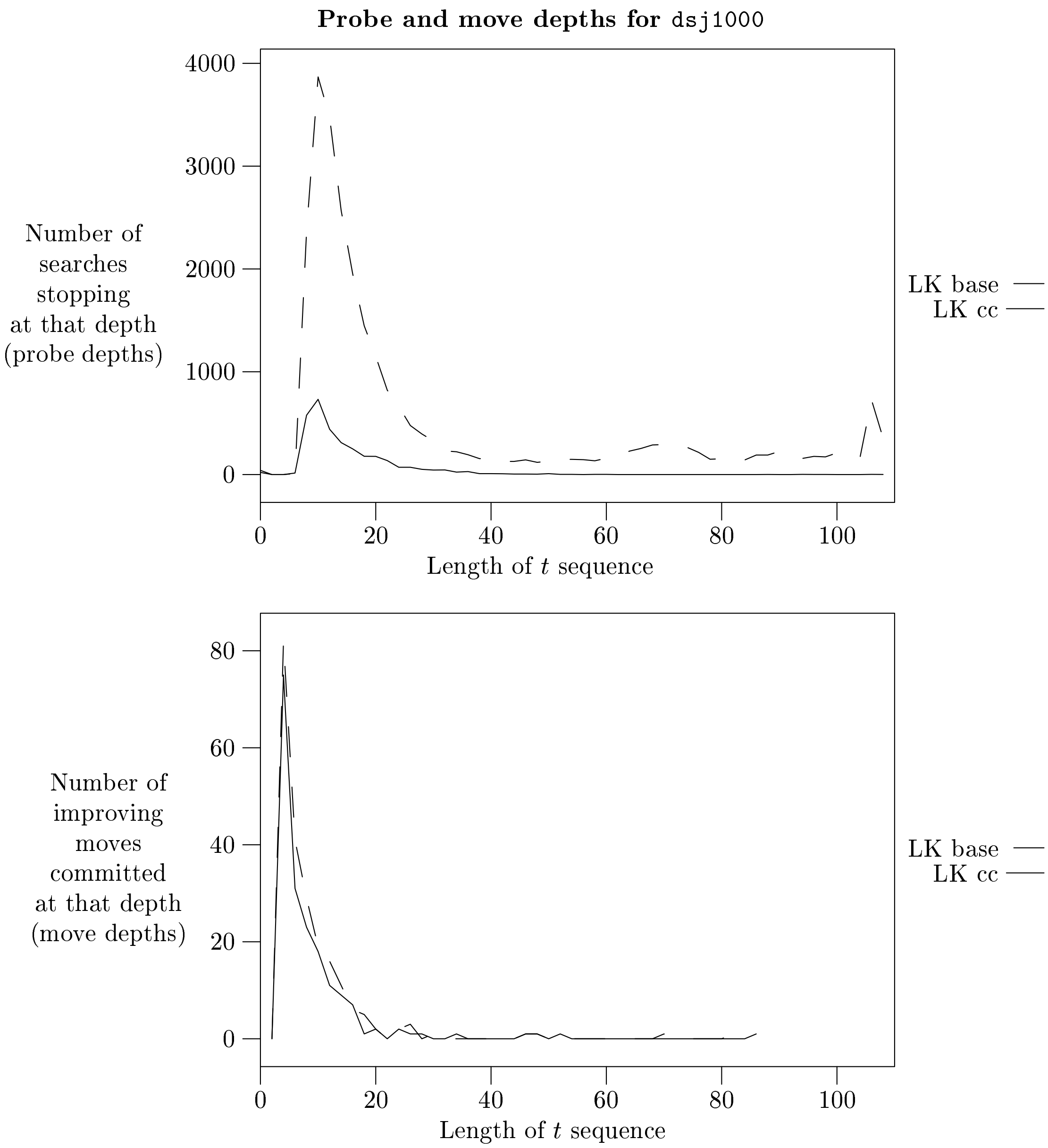

Figure 2: Comparison of probe depths and move depths for instance dsj1000, using candidate set of 20 nearest neighbours and 5 quadrant neighbours in each direction, and using seed 2002.

unfruitful portions of the search space while still finding a set of improving moves with a similar distribution of complexity as measured by move depths. These histograms again correlate well with the measured results for this instance. Both variants produce similar quality tours, but cluster compensation reduces running time by 87%, which is a particularly good case for it.

When comparing the histograms and relative results for the two instances, we see that when cluster compensation provides a larger running time advantage (as in the case of `dsj1000`), it is due to its greater ability to prune unsucessful probes.

7 Other results

This section briefly describes other topics. They are treated in full in [9]. The first topic is the extension of the results to Iterated Lin-Kernighan. The second topic considers a technical result on the expressivity of the Lin-Kernighan heuristic itself, and whether the result still holds when cluster compensation is applied.

7.1 Iterated Lin-Kernighan

Iterated Lin-Kernighan is a variant of Lin-Kernighan in which one trades extra time for better results. See [4] for a full description. The idea is to wrap Lin-Kernighan in an outer loop as follows. After a run of the Lin-Kernighan optimization phase terminates, perturb the result and run another Lin-Kernighan optimization phase. One saves considerable time because one can reuse preprocessed data (*e.g.* candidate sets), by one can restrict the choices of t_1 to the list of cities touched by the by the perturbation, and because all but the first Lin-Kernighan iteration begins with a nearly locally-optimal tour.

We showed in [9] that the results for cluster compensation extend to the Iterated Lin-Kernighan case. That is, tour quality improves with more iterations, and the runtime reduction due to cluster compensation often improves with more iterations. Cluster compensation has an even greater opportunity to amortize its preprocessing costs in the Iterated Lin-Kernighan case because the optimziation phase takes longer. Also, the time-saving features mentioned in the previous paragraph apply equally well to the cluster compensation case as they do to the base case.

7.2 PLS-completeness

Papadimitriou [10] showed a hardness result for the optimization problem defined by the Lin-Kernighan heuristic itself. Set *PLS* captures the class of problems solvable by reasonable local search heuristics, *i.e.* where the number of improving steps is unbounded, but where each improving step must take at most polynomial time. Papadimitriou showed that a variant of the Lin-Kernighan heuristic for the TSP solves a problem that is complete for PLS. In other words, there are reasonably efficient transformations of any problem in PLS to and from the TSP such that the local optimum found by Lin-Kernighan for the transformed TSP instance may be transformed into a local optimium for the original problem.

Given that cluster compensation makes the Lin-Kernighan heuristic more conservative at each step, one might wonder if the theoretical expressive power remains the same. The answer is that it does. One needs only a small technical adjustment to Papadimitriou's argument to shows that his variant of the Lin-Kernighan heuristic with cluster compensation solves a problem that is complete for PLS [9].

8 Conclusion

This paper describes the essential features of the Lin-Kernighan heuristic, and argues that those features combine to force the heuristic to waste effort when presented with a clustered input. We introduce cluster compensation as a way to effectively prune the search space explored by the heuristic. Cluster compensation provides a lookahead term for use in the Lin-Kernighan gain function. The lookahead term estimates the future closing-up costs for search sequences. With an initial investment of finding a minimum spanning tree for the graph together with a tree rewrite, online computations of the lookahead term can be computed in constant time.

We provide experimental results showing that when finding a minimum spanning tree is fast, as is the case for geometric instances, then cluster compensation more than pays for itself. It can reduce overall running times by up to two thirds, with greater advantage given on more sharply clustered instances. When finding a minimum spanning tree is relatively expensive, as is the case for instances represented by full distance matrices, cluster compensation does not pay for itself in running time.

An attractive feature of cluster compensation is that it is orthogonal to many other techniques, because it merely augments the gain functions with an extra quickly-computed term. That is, it can be easily combined with other techniques for improving or adapting the Lin-Kernighan heuristic.

For example, elsewhere we have shown that cluster compensation can be effective when used with a Lin-Kernighan style heuristic for the Minimum Weight Perfect Matching Problem [9]. In that context we have similar results as for the TSP heuristic. However, in the matching case cluster compensation has more difficulty paying for itself because the optimization phase for weighted perfect matching typically runs in far less time than in the TSP case. That is, the preprocessing overheads are always a larger factor in overall runtime.

We hope other researchers will carry out further investigations with cluster compensation or other topologically-inspired techniques. Perhaps there are other uses for cluster compensation's key ability to quickly compute cluster distances.

9 Acknowledgements

This research was conducted as part of the author's doctoral studies at the Department of Computer Science at the University of Toronto. I gratefully acknowledge the academic and financial support of my thesis supervisor, Derek Corneil, and the support of my doctoral committee. The research was funded in part by the Department, and by the Natural Sciences and Engineering Research Council of Canada Post-Graduate Scholarhip Programme. The author's current employer, Altera Corporation, provided the workstation on which the experiments for this paper were run.

References

[1] 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.

[2] 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.

[3] M. L. Fredman, D. S. Johnson, L. A. McGeoch, and G. Ostheimer. Data structures for traveling salesmen. In *Proceedings of the Fourth ACM-SIAM Symposium on Discrete Algorithms*, pages 145–154, 1993.

[4] 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.

[5] D. S. Johnson. 8th DIMACS Implementation Challenge: The Traveling Salesman Problem. See the web site `http://www.research.att.com/~dsj/chtsp/index.html`, 2000–2002.

[6] 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.

[7] D. S. Johnson and L. A. McGeoch. Experimental analysis of heuristics for the STSP. Draft of 21 November 2001, available from `http://www.research.att.com/~dsj/chtsp/index.html`. This is a chapter to appear in the book "The Traveling Salesman Problem and its Variations", Gutin and Punnen (editors), Kluwer Academic Publishers, 2002, 2001.

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

[9] D. M. Neto. *Efficient Cluster Compensation for Lin-Kernighan Heuristics*. PhD thesis, University of Toronto, 1999.

[10] 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.

[11] B. Schieber. Parallel lowest common ancestor computation. In John H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 6, pages 259–273. Morgan Kaufmann, 1993.

A Detailed experimental results

This section provides the details of the experimental results. The values in summary Tables 1 and 2 are taken from the summary lines of these tables.

A.1 Tour quality trends

The quality of the tours produced by both variants of the heuristic are very similar. Within each instance class, there are few strong trends in the difference between the two heuristics as instance sizes grow.

For the larger Uniform Euclidean instances, one could argue that cluster compensation consistently produces tours that are 0.04% longer than those produced by the base heuristic.

For the larger Random Distance Matrix instances, both variants of the heuristic appear to be producing worse tours as compared to the Held-Karp lower bound. In 1000-city range, both heuristics produce tours about 3 to 3.3% longer than the Held-Karp bound. By the 10000 city instance range, both heuristics fail to produce tours within 5% the Held-Karp bound. However, at that range, cluster compensation produces tours that are roughly 0.2 to 0.3% shorter than those produced by the base heuristic.

A.2 Running time trends

As for running times, there is of course a general increase in absolute running times as instances grow larger.

On top of that trend, the runtimes on TSPLIB instances can vary a great deal, depending on how sharply clustered they are. Cluster compensation increases running times for only two TSPLIB instances.

For Uniform Euclidean instances, cluster compensation tends to provide less of a proportional advantage as instances grow very large. We believe this is due to the fact that the fixed overhead of computing the candidate set tends to dominate overal running time. This becomes very clear when comparing Table 10 which has total running times with Table 11 in which the fixed overheads are removed from the running time. Clearly our implementation of computing candidate sets needs to be better tuned.

For Clustered instances, the running time advantage of cluster compensation appears to be converging to a savings of about 65%, despite the increase in the proportion of running time due the fixed overheads.

For Random Distance Matrix instances, the overheads specific to cluster compensation are growing proportionally worse as instance sizes grow.

As discussed earlier, this is due to the fact that the time to compute the minimum spanning tree in such instances grows quadratically with the number of cities, while the running time for the optimization phase tends to be subquadratic. Thus, cluster compensation has no hope of making up the lost ground, unless its initial overheads are amortized over a very large number of iterations of Iterated Lin-Kernighan.

B Calibration of the computing platform

Table 19 provides the DIMACS TSP Challenge calibration data for the experimental platform. They are the running times of a greedy tour construction algorithm.

Instance	LK base % over HK	LK cc % over HK	LK cc % / LK base %
dsj1000.tsp	3.05	3.23	1.06
pr1002.tsp	2.52	2.71	1.07
u1060.tsp	1.87	1.90	1.02
vm1084.tsp	2.39	2.81	1.18
pcb1173.tsp	2.19	2.33	1.06
d1291.tsp	4.30	5.15	1.20
rl1304.tsp	4.02	3.35	0.83
rl1323.tsp	2.83	2.65	0.94
nrw1379.tsp	1.22	1.39	1.14
fl1400.tsp	6.89	6.22	0.90
u1432.tsp	1.56	1.43	0.92
fl1577.tsp	9.25	8.05	0.87
d1655.tsp	2.85	3.92	1.38
vm1748.tsp	2.48	2.55	1.03
u1817.tsp	2.75	2.96	1.08
rl1889.tsp	3.56	3.96	1.11
d2103.tsp	3.50	3.59	1.02
u2152.tsp	2.46	2.65	1.07
u2319.tsp	0.43	0.42	0.97
pr2392.tsp	2.87	3.15	1.10
pcb3038.tsp	2.15	2.21	1.03
fl3795.tsp	8.75	8.37	0.96
fnl4461.tsp	1.58	1.60	1.01
rl5915.tsp	2.98	2.71	0.91
rl5934.tsp	3.91	3.69	0.94
pla7397.tsp	2.59	2.08	0.80
rl11849.tsp	2.44	2.74	1.12
usa13509.tsp	1.99	1.98	1.00
brd14051.tsp	1.78	1.84	1.03
d15112.tsp	1.62	1.69	1.04
d18512.tsp	1.60	1.60	1.00
pla33810.tsp	2.83	2.69	0.95
pla85900.tsp	2.10	2.22	1.06
Geometric average	2.58	2.63	1.02

Table 3: Comparison of tour lengths for TSPLIB instances, using the candidate set consisting of the 20 nearest neighbours.

Instance	LK base seconds	LK cc seconds	LK cc / LK base
dsj1000.tsp	4.86	1.80	0.37
pr1002.tsp	1.49	0.33	0.22
u1060.tsp	4.03	1.49	0.37
vm1084.tsp	0.88	0.46	0.52
pcb1173.tsp	0.35	0.23	0.65
d1291.tsp	4.90	2.88	0.59
rl1304.tsp	2.33	1.53	0.65
rl1323.tsp	1.65	0.79	0.48
nrw1379.tsp	0.28	0.34	1.21
fl1400.tsp	29.68	8.06	0.27
u1432.tsp	0.35	0.37	1.07
fl1577.tsp	20.62	10.28	0.50
d1655.tsp	7.33	2.66	0.36
vm1748.tsp	1.92	1.47	0.77
u1817.tsp	2.23	0.81	0.37
rl1889.tsp	5.33	4.19	0.79
d2103.tsp	8.23	2.76	0.33
u2152.tsp	1.13	0.70	0.62
u2319.tsp	0.44	0.51	1.16
pr2392.tsp	1.01	0.74	0.73
pcb3038.tsp	0.92	0.61	0.66
fl3795.tsp	38.83	16.14	0.42
fnl4461.tsp	0.94	0.91	0.96
rl5915.tsp	7.08	5.26	0.74
rl5934.tsp	12.87	9.88	0.77
pla7397.tsp	17.74	11.92	0.67
rl11849.tsp	11.68	10.65	0.91
usa13509.tsp	13.04	6.14	0.47
brd14051.tsp	10.32	6.80	0.66
d15112.tsp	7.23	4.97	0.69
d18512.tsp	6.76	4.53	0.67
pla33810.tsp	108.61	72.97	0.67
pla85900.tsp	180.16	230.63	1.28
Geometric average	4.26	2.56	0.60

Table 4: Comparison of total runtime for TSPLIB instances, using the candidate set consisting of the 20 nearest neighbours.

Instance	LK base % over HK	LK cc % over HK	LK cc % / LK base %
dsj1000.tsp	2.32	2.07	0.89
pr1002.tsp	2.59	2.51	0.97
u1060.tsp	1.61	1.65	1.02
vm1084.tsp	2.42	2.61	1.08
pcb1173.tsp	2.39	2.25	0.94
d1291.tsp	4.75	4.78	1.01
rl1304.tsp	3.59	3.17	0.88
rl1323.tsp	2.87	2.85	0.99
nrw1379.tsp	1.50	1.39	0.93
fl1400.tsp	2.90	2.66	0.92
u1432.tsp	1.70	1.62	0.95
fl1577.tsp	3.98	5.16	1.30
d1655.tsp	2.50	2.96	1.18
vm1748.tsp	2.40	2.53	1.06
u1817.tsp	2.61	2.65	1.01
rl1889.tsp	3.30	3.45	1.04
d2103.tsp	3.36	3.11	0.92
u2152.tsp	2.51	2.80	1.12
u2319.tsp	0.46	0.43	0.94
pr2392.tsp	3.06	2.75	0.90
pcb3038.tsp	2.18	2.25	1.03
fl3795.tsp	3.72	4.74	1.27
fnl4461.tsp	1.65	1.70	1.03
rl5915.tsp	2.85	2.71	0.95
rl5934.tsp	3.18	3.14	0.99
pla7397.tsp	1.90	1.81	0.95
rl11849.tsp	2.45	2.48	1.01
usa13509.tsp	1.91	1.94	1.01
brd14051.tsp	1.70	1.76	1.04
d15112.tsp	1.60	1.65	1.03
d18512.tsp	1.56	1.63	1.04
pla33810.tsp	2.00	2.08	1.04
pla85900.tsp	1.73	1.99	1.15
Geometric average	2.29	2.33	1.01

Table 5: Comparison of tour lengths for TSPLIB instances, using the candidate set consisting of 20 nearest neighbours together with the 5 nearest neighbours in each quadrant direction.

Instance	LK base seconds	LK cc seconds	LK cc / LK base
dsj1000.tsp	6.15	0.82	0.13
pr1002.tsp	1.23	0.50	0.41
u1060.tsp	3.83	1.38	0.36
vm1084.tsp	1.05	0.40	0.38
pcb1173.tsp	0.43	0.28	0.66
d1291.tsp	7.71	2.57	0.33
rl1304.tsp	2.43	1.17	0.48
rl1323.tsp	1.61	0.91	0.57
nrw1379.tsp	0.61	0.39	0.64
fl1400.tsp	34.22	6.40	0.19
u1432.tsp	0.48	0.46	0.96
fl1577.tsp	34.13	13.36	0.39
d1655.tsp	7.04	1.09	0.15
vm1748.tsp	1.85	1.24	0.67
u1817.tsp	1.56	0.73	0.47
rl1889.tsp	3.72	2.93	0.79
d2103.tsp	7.23	1.54	0.21
u2152.tsp	1.19	0.68	0.57
u2319.tsp	0.61	0.65	1.07
pr2392.tsp	1.16	0.73	0.63
pcb3038.tsp	1.17	0.87	0.74
fl3795.tsp	60.52	19.30	0.32
fnl4461.tsp	1.76	1.36	0.77
rl5915.tsp	6.08	5.28	0.87
rl5934.tsp	12.23	9.62	0.79
pla7397.tsp	21.94	13.31	0.61
rl11849.tsp	14.38	13.07	0.91
usa13509.tsp	14.28	8.79	0.62
brd14051.tsp	16.38	9.36	0.57
d15112.tsp	9.39	7.41	0.79
d18512.tsp	10.24	8.06	0.79
pla33810.tsp	127.03	48.74	0.38
pla85900.tsp	202.00	224.35	1.11
Geometric average	4.98	2.59	0.52

Table 6: Comparison of total runtime for TSPLIB instances, using the candidate set consisting of 20 nearest neighbours together with the 5 nearest neighbours in each quadrant direction.

Instance	LK base % over HK	LK cc % over HK	LK cc / LK base
E1k.7	1.97	2.17	1.10
E1k.8	1.95	2.14	1.10
E1k.9	1.92	1.97	1.02
E1k.0	2.05	1.96	0.96
E1k.1	1.78	1.72	0.97
E1k.2	1.71	1.99	1.16
E1k.3	1.66	1.67	1.00
E1k.4	1.98	1.77	0.89
E1k.5	1.92	2.02	1.05
E1k.6	1.90	1.89	1.00
E3k.1	1.87	1.83	0.98
E3k.2	1.97	2.00	1.02
E3k.3	1.80	1.87	1.04
E3k.4	1.94	1.87	0.97
E3k.0	1.86	1.91	1.03
E10k.1	1.95	1.93	0.99
E10k.2	1.82	1.85	1.02
E10k.0	1.92	1.99	1.04
E31k.0	1.86	1.92	1.03
E31k.1	1.81	1.87	1.03
E100k.1	1.89	1.92	1.02
E100k.0	1.91	1.97	1.03
E316k.0	1.92	1.97	1.03
E1M.0	1.87	1.93	1.03
E3M.0	1.83	1.88	1.03
Geometric average	1.88	1.92	1.02

Table 7: Comparison of tour lengths for Uniform Euclidean instances, using the candidate set consisting of the 20 nearest neighbours.

Instance	LK base seconds	LK cc seconds	LK cc / LK base
E1k.7	0.32	0.25	0.78
E1k.8	0.25	0.19	0.74
E1k.9	0.29	0.20	0.67
E1k.0	0.30	0.18	0.61
E1k.1	0.36	0.23	0.65
E1k.2	0.33	0.18	0.56
E1k.3	0.28	0.18	0.64
E1k.4	0.29	0.18	0.63
E1k.5	0.32	0.19	0.59
E1k.6	0.30	0.19	0.64
E3k.1	0.93	0.64	0.69
E3k.2	0.96	0.66	0.69
E3k.3	0.93	0.65	0.70
E3k.4	0.91	0.61	0.66
E3k.0	0.93	0.65	0.70
E10k.1	3.34	2.52	0.75
E10k.2	3.24	2.37	0.73
E10k.0	3.34	2.44	0.73
E31k.0	11.88	9.25	0.78
E31k.1	11.54	9.11	0.79
E100k.1	39.42	31.40	0.80
E100k.0	40.50	32.13	0.79
E316k.0	133.99	108.76	0.81
E1M.0	455.70	386.86	0.85
E3M.0	1701.11	1613.48	0.95
Geometric average	2.41	1.72	0.71

Table 8: Comparison of total runtime for Uniform Euclidean instances, using the candidate set consisting of the 20 nearest neighbours.

Instance	LK base % over HK	LK cc % over HK	LK cc % / LK base %
E1k.7	1.91	2.17	1.13
E1k.8	1.96	2.20	1.12
E1k.9	2.02	2.09	1.03
E1k.0	1.99	2.02	1.02
E1k.1	1.86	1.70	0.91
E1k.2	1.77	1.89	1.07
E1k.3	1.55	1.77	1.14
E1k.4	1.99	1.96	0.99
E1k.5	1.81	1.92	1.06
E1k.6	2.09	1.99	0.95
E3k.1	1.84	1.78	0.97
E3k.2	2.00	1.95	0.98
E3k.3	1.82	1.95	1.07
E3k.4	1.90	1.91	1.01
E3k.0	1.80	1.99	1.10
E10k.1	1.89	1.91	1.01
E10k.2	1.85	1.87	1.01
E10k.0	1.92	1.98	1.03
E31k.0	1.86	1.92	1.03
E31k.1	1.82	1.87	1.03
E100k.1	1.89	1.94	1.03
E100k.0	1.91	1.98	1.04
E316k.0	1.92	1.99	1.03
E1M.0	1.87	1.92	1.03
E3M.0	1.83	1.87	1.03
Geometric average	1.88	1.94	1.03

Table 9: Comparison of tour lengths for Uniform Euclidean instances, using the candidate set consisting of 20 nearest neighbours together with the 5 nearest neighbours in each quadrant direction.

Instance	LK base seconds	LK cc seconds	LK cc / LK base
E1k.7	0.37	0.27	0.74
E1k.8	0.29	0.23	0.78
E1k.9	0.32	0.24	0.76
E1k.0	0.31	0.22	0.71
E1k.1	0.38	0.25	0.67
E1k.2	0.37	0.23	0.62
E1k.3	0.29	0.22	0.75
E1k.4	0.33	0.22	0.68
E1k.5	0.35	0.23	0.66
E1k.6	0.34	0.23	0.67
E3k.1	1.10	0.79	0.72
E3k.2	1.11	0.78	0.70
E3k.3	1.11	0.79	0.71
E3k.4	1.08	0.78	0.72
E3k.0	1.12	0.79	0.71
E10k.1	4.59	3.66	0.80
E10k.2	4.33	3.40	0.79
E10k.0	4.43	3.45	0.78
E31k.0	20.30	17.66	0.87
E31k.1	19.38	17.01	0.88
E100k.1	88.98	81.03	0.91
E100k.0	88.94	80.95	0.91
E316k.0	398.18	373.19	0.94
E1M.0	1974.47	1904.53	0.96
E3M.0	9632.17	9540.04	0.99
Geometric average	3.55	2.73	0.77

Table 10: Comparison of total runtime for Uniform Euclidean instances, using the candidate set consisting of 20 nearest neighbours together with the 5 nearest neighbours in each quadrant direction. As instances become very large, the computation of the candidate set ends up dominating the runtime. We provide the next table to remove the distorting effect of that common computation on the comparison of running times.

Instance	LK base seconds	LK cc seconds	LK cc / LK base
E1k.7	0.27	0.16	0.60
E1k.8	0.20	0.12	0.59
E1k.9	0.24	0.15	0.62
E1k.0	0.23	0.18	0.76
E1k.1	0.30	0.16	0.53
E1k.2	0.27	0.12	0.44
E1k.3	0.19	0.16	0.81
E1k.4	0.23	0.11	0.48
E1k.5	0.26	0.13	0.51
E1k.6	0.25	0.12	0.49
E3k.1	0.79	0.52	0.66
E3k.2	0.81	0.42	0.52
E3k.3	0.81	0.48	0.59
E3k.4	0.78	0.47	0.60
E3k.0	0.80	0.48	0.60
E10k.1	2.64	1.73	0.66
E10k.2	2.48	1.58	0.64
E10k.0	2.55	1.59	0.62
E31k.0	8.21	5.62	0.68
E31k.1	7.93	5.56	0.70
E100k.1	26.44	18.74	0.71
E100k.0	26.91	19.08	0.71
E316k.0	88.65	64.07	0.72
E1M.0	300.93	231.91	0.77
E3M.0	1158.04	1073.62	0.93
Geometric average	1.86	1.17	0.63

Table 11: Comparison of only the "interesting" parts of runtime on the Uniform Euclidean class of instances. For the base case, we include only the time for the LK optimziation phase. For the cluster compensation case, we include the time for the LK optimization phase and also the preprocessing overhead for cluster compensation (which is dominated by the time to compute a minimum spanning tree of the graph). That is, we exclude other common setup time, *e.g.* building the candidate sets. The candidate set consists of 20 nearest neighbours together with the 5 nearest neighbours in each quadrant direction.

Instance	LK base % over HK	LK cc % over HK	LK cc / % LK base %
C1k.0	4.04	4.19	1.04
C1k.1	2.18	2.23	1.02
C1k.2	3.00	3.35	1.12
C1k.3	2.27	2.38	1.05
C1k.4	4.31	4.12	0.96
C1k.5	6.30	6.39	1.01
C1k.6	6.84	6.96	1.02
C1k.7	4.48	4.94	1.10
C1k.8	2.85	3.05	1.07
C1k.9	4.52	4.68	1.03
C3k.0	5.35	5.97	1.12
C3k.1	7.71	7.48	0.97
C3k.2	6.06	6.08	1.00
C3k.3	6.77	6.93	1.02
C3k.4	6.48	6.65	1.03
C10k.0	6.85	6.77	0.99
C10k.1	6.96	7.08	1.02
C10k.2	8.10	8.06	0.99
C31k.0	7.25	7.26	1.00
C31k.1	6.72	6.88	1.02
C100k.0	6.75	6.85	1.02
C100k.1	6.03	6.08	1.01
C316k.0	6.09	6.36	1.04
Geometric average	5.23	5.37	1.03

Table 12: Comparison of tour lengths for Clustered Euclidean instances, using the candidate set consisting of the 20 nearest neighbours.

Instance	LK base seconds	LK cc seconds	LK cc / LK base
C1k.0	13.28	7.45	0.56
C1k.1	13.64	3.69	0.27
C1k.2	13.73	7.92	0.58
C1k.3	12.57	4.90	0.39
C1k.4	15.40	6.56	0.43
C1k.5	15.29	6.99	0.46
C1k.6	8.82	7.51	0.85
C1k.7	12.17	5.14	0.42
C1k.8	13.83	5.73	0.41
C1k.9	10.22	4.25	0.42
C3k.0	34.78	13.52	0.39
C3k.1	36.78	16.80	0.46
C3k.2	35.57	12.45	0.35
C3k.3	31.74	12.98	0.41
C3k.4	37.45	12.07	0.32
C10k.0	104.09	31.69	0.30
C10k.1	100.87	41.10	0.41
C10k.2	105.23	40.68	0.39
C31k.0	315.17	114.54	0.36
C31k.1	314.70	118.29	0.38
C100k.0	997.92	355.94	0.36
C100k.1	1003.65	349.38	0.35
C316k.0	3180.84	1114.59	0.35
Geometric average	51.25	20.78	0.41

Table 13: Comparison of total runtime for Clustered Euclidean instances, using the candidate set consisting of the 20 nearest neighbours.

Instance	LK base % over HK	LK cc % over HK	LK cc / % LK base %
C1k.0	1.96	1.78	0.91
C1k.1	1.56	2.07	1.33
C1k.2	2.45	2.41	0.98
C1k.3	1.48	1.61	1.09
C1k.4	2.53	1.98	0.78
C1k.5	5.08	5.62	1.11
C1k.6	1.85	1.83	0.99
C1k.7	2.68	2.27	0.85
C1k.8	1.74	1.85	1.06
C1k.9	1.90	1.80	0.95
C3k.0	2.51	2.53	1.01
C3k.1	3.84	4.04	1.05
C3k.2	3.26	3.68	1.13
C3k.3	5.61	5.79	1.03
C3k.4	3.57	3.54	0.99
C10k.0	4.44	4.36	0.98
C10k.1	4.83	5.21	1.08
C10k.2	4.59	5.03	1.09
C31k.0	4.52	4.58	1.01
C31k.1	4.63	4.45	0.96
C100k.0	4.38	4.40	1.01
C100k.1	3.78	3.83	1.01
C316k.0	3.88	4.09	1.05
Geometric average	3.09	3.14	1.02

Table 14: Comparison of tour lengths for Clustered Euclidean instances, using the candidate set consisting of 20 nearest neighbours together with the 5 nearest neighbours in each quadrant direction.

Instance	LK base seconds	LK cc seconds	LK cc / LK base
C1k.0	17.46	5.63	0.32
C1k.1	16.43	4.63	0.28
C1k.2	16.10	7.04	0.44
C1k.3	16.96	3.20	0.19
C1k.4	19.99	4.92	0.25
C1k.5	15.46	6.29	0.41
C1k.6	11.47	3.20	0.28
C1k.7	16.85	5.32	0.32
C1k.8	16.47	4.79	0.29
C1k.9	13.45	8.48	0.63
C3k.0	43.21	11.30	0.26
C3k.1	55.60	22.51	0.40
C3k.2	52.45	13.43	0.26
C3k.3	36.81	14.79	0.40
C3k.4	49.44	10.03	0.20
C10k.0	156.15	41.39	0.27
C10k.1	140.20	52.23	0.37
C10k.2	144.10	49.18	0.34
C31k.0	453.40	146.67	0.32
C31k.1	445.64	134.30	0.30
C100k.0	1393.49	450.08	0.32
C100k.1	1450.09	450.90	0.31
C316k.0	4666.95	1602.44	0.34
Geometric average	68.05	21.48	0.32

Table 15: Comparison of total runtime for Clustered Euclidean instances, using the candidate set consisting of 20 nearest neighbours together with the 5 nearest neighbours in each quadrant direction.

Instance	LK base % over HK	LK cc % over HK	LK cc / % LK base %
M1k.0	3.00	3.23	1.08
M1k.1	2.86	3.15	1.10
M1k.2	2.92	3.01	1.03
M1k.3	3.47	2.98	0.86
M3k.0	4.22	4.06	0.96
M3k.1	4.39	4.16	0.95
M10k.0	5.57	5.24	0.94
Geometric average	3.67	3.62	0.98

Table 16: Comparison of tour lengths for Random Distance Matrix instances, using the candidate set consisting of the 20 nearest neighbours.

Instance	LK base seconds	LK cc seconds	LK cc / LK base
M1k.0	0.50	0.52	1.03
M1k.1	0.52	0.52	0.99
M1k.2	0.53	0.53	1.01
M1k.3	0.54	0.52	0.97
M3k.0	4.03	4.26	1.05
M3k.1	4.05	4.33	1.07
M10k.0	36.13	39.05	1.08
Geometric average	1.72	1.77	1.03

Table 17: Comparison of total runtime for Random Distance Matrix instances, using the candidate set consisting of the 20 nearest neighbours.

Instance	LK base seconds	LK cc seconds	LK cc / LK base
M1k.0	0.14	0.16	1.10
M1k.1	0.16	0.14	0.86
M1k.2	0.17	0.19	1.08
M1k.3	0.18	0.17	0.95
M3k.0	0.84	1.08	1.28
M3k.1	0.85	1.12	1.32
M10k.0	5.90	8.92	1.51
Geometric average	0.44	0.50	1.14

Table 18: Comparison of only the "interesting" parts of runtime on the Random Distance Matrix class of instances. For the base case, we include only the time for the LK optimziation phase. For the cluster compensation case, we include the time for the LK optimization phase and also the preprocessing overhead for cluster compensation (which is dominated by the time to compute a minimum spanning tree of the graph).

Instance	Iterations	Run 1 (sec)	Run 2 (sec)	Run 3 (sec)
E1k.0	1000	9.68	9.65	9.62
E3k.0	316	10.11	10.14	10.17
E10k.0	100	16.50	16.54	16.56
E31k.0	32	25.96	26.00	25.98
E100k.0	10	30.45	30.48	30.46
E316k.0	3	33.61	33.42	33.40
E1M.0	1	46.78	45.11	45.09
E3M.0	1	169.21	168.31	168.34
E10M.0	1	632.50	630.62	630.51

Table 19: Running times for the calibration application on the experimental platform.