A Multilevel Approach to the Travelling Salesman Problem

Chris Walshaw*

Computing and Mathematical Sciences, University of Greenwich, Park Row, Greenwich, London, SE10 9LS, UK.

May 8, 2001

Abstract

We motivate, derive and implement a multilevel approach to the travelling salesman problem. The resulting algorithm progressively coarsens the problem, initialises a tour and then employs either the Lin-Kernighan (LK) or the Chained Lin-Kernighan (CLK) algorithm to refine the solution on each of the coarsened problems in reverse order. In experiments on a well established test suite of 79 problem instances we found multilevel configurations that either improved the tour quality by over 25% as compared to the standard CLK algorithm using the same amount of execution time, or that achieved approximately the same tour quality over 7 times more rapidly. Moreover the multilevel variants seem to optimise far better the more clustered instances with which the LK & CLK algorithms have the most difficulties.

Keywords: Multilevel Refinement; Travelling Salesman; Combinatorial Optimisation.

1 Introduction

In this paper we address the Travelling Salesman Problem (TSP) which can be simply stated as follows: given a collection of 'cities', find the shortest tour which visits all of them and returns to the starting point. Typically the cities are given coordinates in the 2D plane and then the tour length is measured by the sum of Euclidean distances between each pair on the tour. However, in the more general form, the problem description simply requires a metric which specifies the distance between every pair of cities.

In particular here we consider the problem of finding low cost tours in reasonable time rather than solving the problem to optimality. We also focus on the Euclidean version of distance and, by default therefore, the symmetric TSP. In other words, if $d(c_1, c_2)$ is the Euclidean distance between cities c_1 & c_2 then $d(c_1, c_2) = d(c_2, c_1)$ and the tour can be executed in either direction for the same cost. However in §5.1 we discuss how our approach might easily be extended to a more general distance metric.

The TSP, a combinatorial optimisation problem, has been shown to be NP-hard, [13], but has a number of features which make it stand out amongst such problems. Firstly, and perhaps because of the fact that the problem is so intuitive and easy to state, it has almost certainly been more widely studied than any other NP-hard combinatorial optimisation problem. For example Johnson & McGeoch, [20], survey a wide range of approaches which run the gamut from local search, through simulated annealing, tabu search & genetic algorithms to neural nets. Remarkably, and despite all this interest, the local search algorithm proposed by Lin & Kernighan in 1973, [25], still remains at the heart of the most successful approaches. In fact Johnson & McGeoch describe the Lin-Kernighan (LK) algorithm as the world champion heuristic for the TSP from 1973 to 1989. Further, this was only conclusively superseded by chained or iterated versions of LK (see §3.3 for clarification) originally proposed by Martin, Otto & Felten, [26, 27], in 1991.

Even today, in spite of all the work on exotic and complex combinatorial optimisation techniques, Johnson & McGeoch, [20], conclude that an iterated Lin-Kernighan (ILK) scheme provides the highest quality tours for a reasonable cost. This conclusion has been backed up very recently by Applegate, Cook & Rohe, [2], who also illustrate the scalability of the algorithm by applying it to random examples containing up to

*Email: `C.Walshaw@gre.ac.uk`; URL: `www.gre.ac.uk/~c.walshaw`

25,000,000 cities. In fact it *is* usually possible to improve on the quality of (suboptimal) chained/iterated LK tours, for example by sophisticated tour merging techniques similar to genetic algorithm crossovers (see e.g. [1]), but Johnson & McGeoch suggest that 'the ILK variant . . . , is the most cost effective way to improve on Lin-Kernighan, at least until one reaches stratospheric running times'.

Another unusual feature of the TSP is that, for problems which have not yet been solved to optimality (typically with 10,000 or more cities), an extremely good lower bound can be found for the optimal tour length. This bound, known as the Held-Karp Lower Bound (HKLB), was developed in 1970 by Held & Karp, [16, 17], and usually comes extremely close to known optimal tour lengths (often within 1% – see the results in §4.1). Thus to measure the quality of an algorithm for a given set of problem instances (especially if some or all of them do not have known optimal tours), we can simply calculate the average percentage excess of tours produced by the algorithm over the HKLB for each instance.

To illustrate this for the instances of the TSP tested in this paper, LK produces tours about 3.86% in excess of the HKLB on average, whilst the chained LK algorithm brings this down to about a 1.50% excess although on average requires nearly 40 times as long to achieve this. Again, this would appear to be another unusual feature of the TSP, that what are basically local search algorithms can get so close to optimality (and recall that the HKLB is a *lower* bound so the results will be even closer to optimality than this). For example, no such heuristic (and no such lower bound) is known to exist for the graph partitioning problem.

1.1 Overview & notation

In this paper we describe the motivation, implementation and testing of a multilevel approach to finding high quality TSP tours. Below we discuss the merits & features of the multilevel paradigm, based on previous multilevel algorithms for the graph partitioning and graph drawing problems, which led us to investigate a similar approach to the TSP. In Section 3 we then give the details of the resulting procedure that was devised and outline the chained LK algorithm which is used as a basic building block of the scheme. In Section 4 we test the algorithm on a large suite of problems and attempt to analyse its behaviour. Finally in Section 5 we summarise the paper and present some suggestions for further work.

Although we do not require a great deal of notation for this paper it is worth remarking that we sometimes use graph based terminology and refer to cities as vertices and inter-city distances as edge lengths. We sometimes refer to tours as cycles and we shall also use the terms objective function & cost function to denote the tour length, the quantity we are trying to minimise.

2 Multilevel Optimisation

Before describing the strategy we shall first attempt to motivate it. We shall do so by describing the process of ideas which led us to conclude that a multilevel strategy might be beneficial for the TSP. Although such process driven research does not often form a part of the literature, we feel that in this case it is instructive. A broader survey of the multilevel paradigm and the combinatorial optimisation problems to which it has been applied can be found in [35].

2.1 Background

Our interest in the TSP, and in fact behind our approach to the problem, arises from our work in the field of graph partitioning, [36], and subsequently graph drawing, [33]. Typically a P-way graph partitioning algorithm aims to divide a graph into P disjoint subdomains of equal size and minimise the number of cut edges, another NP-hard combinatorial optimisation problem, [14]. In recent years it has been recognised that an effective way of both accelerating graph partitioning algorithms and, more importantly, giving them a 'global' perspective, is to use multilevel techniques. The idea is to match pairs of vertices to form *clusters*, use the clusters to define a new graph and recursively iterate this procedure until the graph size falls below some threshold. The coarsest graph is then partitioned (often with a crude algorithm) and the partition is successively refined on all the graphs starting with the coarsest and ending with the original. This sequence of contraction followed by repeated refinement loops is known as multilevel partitioning and has been successfully developed as a strategy for overcoming the localised nature of the Kernighan-Lin (KL), [22], and other partition optimisation algorithms. The multilevel partitioning paradigm was first proposed by

Barnard & Simon, [3], as a method of speeding up spectral bisection and improved by both Hendrickson & Leland, [18], and Bui & Jones, [7], who generalised it to encompass local refinement algorithms. Several enhancements for carrying out the matching of vertices have been devised by Karypis & Kumar, [21].The multilevel partitioning strategy is widely used and forms the basis of at least 4 public domain partitioning packages, CHACO [18], JOSTLE [36], METIS [21], and SCOTCH [29].

In another recent development, multilevel strategies have also been applied to the graph drawing problem (and in particular force directed placement). Given a graph with no coordinate information, the aim of a graph drawing algorithm is to infer a 'nice' layout of the vertices based on the adjacency structure (e.g. a layout with relatively few edge crossings and where edges all have approximately the same length). Force directed placement (FDP) algorithms achieve this by assuming repulsive forces between every pair of vertices and the modelling the edges as springs which attempt to maintain their natural length (i.e. neither stretched nor compressed). Strictly speaking this is not a combinatorial optimisation problem but it does share many of the features.

Up until recently most FDP algorithms were at least $O(N^3)$ in complexity and were unable to deal with large graphs (e.g. in comprehensive survey of graph drawing from 1998 one FDP algorithm was singled out as exceptional in that it could handle graphs with over 1,000 vertices, [10]). However Harel & Koren, [15], have used multiscale ideas in combination with an FDP algorithm and are able to easily handle graphs of 6,000 vertices (although their algorithm still contains an $O(N^2)$ component). Meanwhile, Walshaw has independently applied multilevel techniques to the same problem (although using a different FDP algorithm) and presents examples with up to 225,000 vertices, [33].

Most recently a multilevel graph colouring approach has been tested, [34]. The graph colouring problem is to colour the vertices of a graph so that adjacent vertices have different colours and so that a minimum number of colours are used. It is known as one of the most challenging NP-hard combinatorial optimisation problems but has a wide range of applications including scheduling and timetabling. Although the multilevel approach was not universally successful for colouring, it did manage to enhance the performance of the two local search heuristics under investigation, tabu search and the iterated greedy algorithm, for graphs with low edge density (less than about 30% density), a subgroup into which many real-life scheduling applications fall, e.g. [23]. This augments existing evidence that, although the multilevel framework cannot be considered as a panacea for combinatorial optimisation problems, it can provide a useful addition to the combinatorial optimisation toolkit.

2.2 The generic multilevel paradigm

The important questions about these approaches are – why do multilevel approaches appear to work, and, is there an abstraction of the paradigm that can be applied to other combinatorial optimisation problems (such as the TSP)?

Considered from the point of view of the multilevel procedure, a series of increasingly smaller & coarser versions of the original problem are being constructed. It is hoped that each problem P_l retains the important features of its parent P_{l-1} but the (usually) randomised and irregular nature of the coarsening precludes any rigorous analysis of this process.

On the other hand, viewing the multilevel process from the point of view of the optimisation problem and, in particular, the objective function is considerably more enlightening. For a given problem instance the *solution space*, $\mathcal{X}$, is the set of all possible solutions for that instance. The *objective function* or *cost function*, $f : \mathcal{X} \to \mathbb{R}$, assigns a cost to each solution in $\mathcal{X}$(e.g. in the case of the TSP $f(x)$ expresses the length of the tour x). Typically the aim of the problem is to find a state $x \in \mathcal{X}$ at a minimum (or maximum) of the objective function. Iterative refinement algorithms usually attempt to do this by moving stepwise through the solution space (which is hence also known as a *search space*) but often can become trapped in local minima of f.

Suppose then for the partitioning problem that two vertices $u, v \in G_{l-1}$ are matched and coalesced into a single vertex $v' \in G_l$. When a partition refinement algorithm is subsequently used on G_l and whenever v' is assigned to a subdomain, both u & v are also both being assigned to that subdomain. In this way the partitioning of G_l is being restricted to consider only those configurations in the solution space in which u & v lie in the ***same*** subdomain. Since many vertex pairs are generally coalesced from all parts of G_{l-1} to form G_l this set of restrictions is in some way a sampling of the solution space and hence the surface of the objective function.

This is an important point (see also [32]). Previously authors have made a case for multilevel partitioning on the basis that the coarsening successively ***approximates*** the problem. In fact it is somewhat better than this; the coarsening ***samples*** the solution space by placing restrictions on which states the refinement algorithm can visit. Furthermore, this methodology is not confined to multilevel partitioning but can be applied to other combinatorial optimisation problems.

We can then hypothesise that, if the coarsening manages to sample the solution space so as to gradually ***smooth*** the objective function, the multilevel representation of the problem combined with an iterative refinement algorithm should work well as an optimisation ***meta-heuristic***. In other words, by coarsening and smoothing the problem, the multilevel component allows the refinement algorithm to find regions of the solution space where the objective function has a low average value (e.g. broad valleys). This does rely on a certain amount of 'continuity' in the objective function but it is not unusual for these sort of problems that changing one or two components of the solution tends not to change the cost very much.

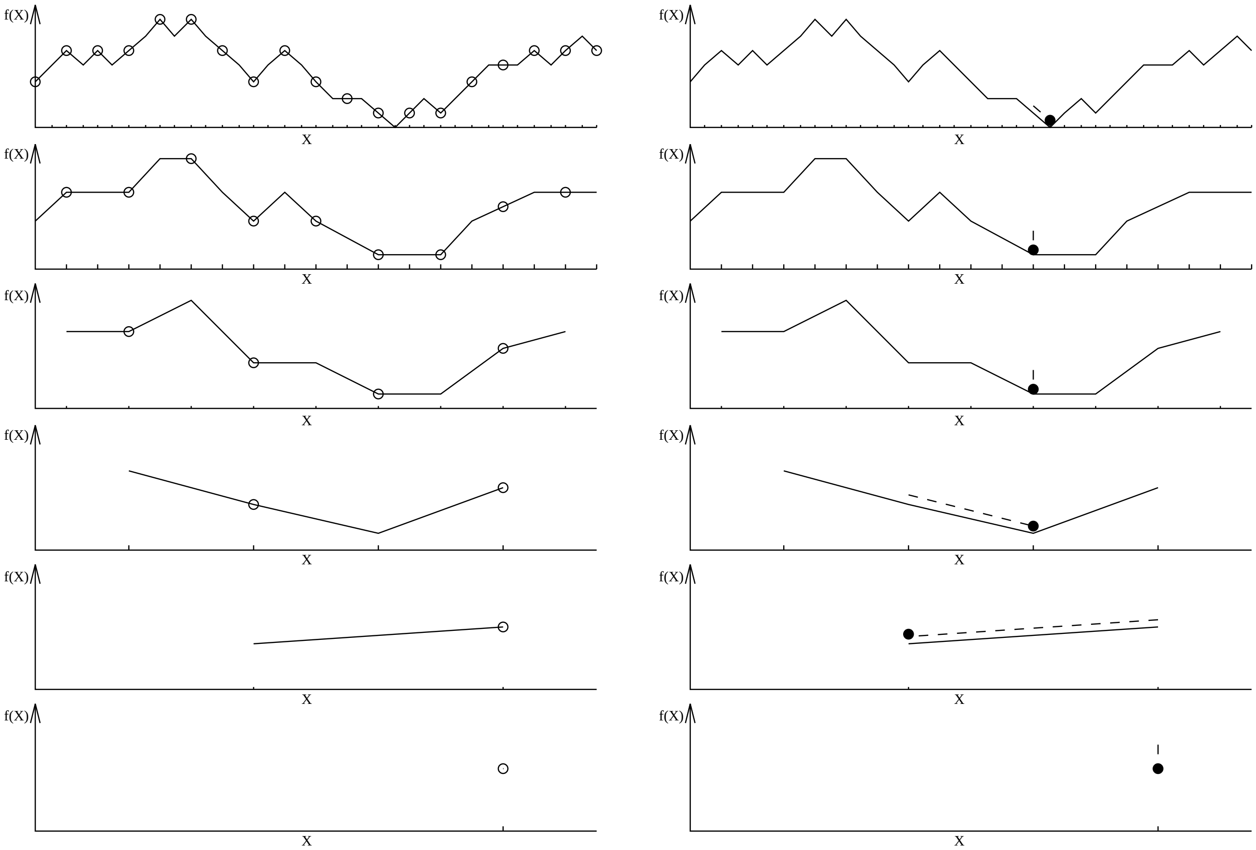


Figure 1: The multilevel scheme in terms of a simple objective function

Figure 1 shows an example of how this might work for a search space $\mathcal{X}$ and objective function $f(\mathcal{X})$ which we aim to minimise. On the left hand side the objective function is gradually sampled and smoothed (the sampled points are circled and all intermediate values removed to give the next coarsest representation). The initial solution for the final coarsened space (shown as a black dot in the bottom right hand figure) is then trivial (because there is only one possible state) although the resulting configuration is not an optimal solution to the overall problem. However this state is used as an initial configuration for the next level up and a *steepest descent* refinement policy finds the nearest local minimum (steepest descent refinement will only move to a neighbouring configuration if the value of the objective function is lower there). Recursing this process keeps the best found solution (indicated by the black dot) in the same region of the solution space. Finally this gives a good initial configuration for the original problem and (in this case) the optimal solution can be found. Note that it is possible to pick a different set of sampling points for this example for which the steepest descent policy will fail to find the global minimum, but this only indicates, as might be expected, that the multilevel procedure is somewhat sensitive to the coarsening strategy.

Of course, this motivational example might be considered trivial or unrealistic (in particular an objective function cannot normally be pictured in 2D). However, consider other meta-heuristics such as repeated random starts combined with steepest descent local search, or even simulated annealing, applied to this same objective function; without lucky initial guesses either might require many iterations to solve this simple problem.

It should be stressed that this hypothesis is nothing more than speculation and we cannot prove that this process underlies the multilevel paradigm. However experimental evidence, here and elsewhere (see e.g. [35]), suggests that the multilevel approach does indeed enhance local search strategies and we suspect that the sampling/smoothing of the objective function contributes to this.

multilevel refinement(`input` *problem instance* P_0 , `output` *solution* $C_0\{P_0\}$)
begin
  $l := 1$
  while *(coarsening)*
    $P_{l+1} =$ **coarsen**(P_l)
    $l := l{+}1$
  end
  $C_l\{P_l\} =$ **initialise**(P_l)
  while $l > 0$
    $l := l{-}1$
    $C_l^0\{P_l\} =$ **extend**($C_{l+1}\{P_{l+1}\}, P_l$)
    $C_l\{P_l\} =$ **refine**($C_l^0\{P_l\}, P_l$)
  end
end

Figure 2: The multilevel optimisation algorithm

To summarise the paradigm then, multilevel optimisation combines a coarsening strategy together with a refinement algorithm (employed at each level in reverse order) to provide an optimisation meta-heuristic. Figure 2 contains a schematic of this process in pseudo-code. Here P_l refers to the coarsened problem after l coarsening steps, $C_l\{P_l\}$ is a solution of this problem and $C_l^0\{P_l\}$ denotes the initial solution.

2.3 Algorithmic requirements

Assuming that the above analysis does contain some elements of truth, how can we implement a multilevel strategy to test it on a given combinatorial optimisation problem?

First of all let us assume that we know of a refinement algorithm for the problem, which refines in the sense it can reuse an existing solution and (attempt to) improve it. Typically the refinement algorithm will be a local search strategy which can only explore small regions of the solution space neighbouring to the current solution, however there is no reason (other than execution time) why it should not be a more sophisticated scheme such as simulated annealing. The refinement algorithm must also be able to cope with any additional restrictions placed on it by using a coarsened problem (e.g. in graph partitioning the coarser graphs are always weighted whether or not the original is). If such a refinement algorithm does not exist (e.g. if the only known heuristics for the problem are based on construction rather than refinement) it is not clear that the multilevel paradigm can be applied.

To implement a multilevel algorithm, given a problem and a refinement strategy for it, we then require three additional basic components: a coarsening algorithm, an initialisation algorithm and an extension algorithm (which takes the solution on one problem and extends it to the parent problem). It is difficult to talk in general terms about these requirements, but the existing examples suggest that the extension is a simple and obvious reversal of the coarsening step which preserves the same cost (e.g. a pair of parent vertices are assigned to the same solution state as their child). The initialisation is also generally a simple canonical mapping (e.g. for partitioning assign P vertices to P subdomains). By canonical we mean that a (non-unique) solution is 'obvious' and that the refinement algorithm cannot possibly improve on the initial solution at the coarsest level (because there are no degrees of freedom).

This just leaves the coarsening algorithm which is then perhaps the key component of a multilevel optimisation implementation. From existing examples two principles seem to hold:

(C1) Any solution in any of the coarsened spaces should induce a legitimate solution on the original space. Thus at any stage after initialisation the current solution could simply be extended through all the problem levels to achieve a solution of the original problem. Furthermore both solutions (in the coarse space and the original space) should have the same cost with respect to the objective function. This requirement ensures that the coarsening is sampling the solution space (rather than approximating and/or distorting it).

(C2) The number of levels need not be determined *a priori* but coarsening should cease when any further coarsening would render the initialisation degenerate.

This still does not tell us *how* to coarsen a given problem. So far most solutions for the partitioning problem have employed a gradual and fairly uniform reduction and it has been shown that it is usually more profitable for the coarsening to respect the objective function in some sense (see e.g. the heavy edge matching strategy in [21]). In this respect it seems likely that the most difficult aspect of finding an effective multilevel algorithm for a given problem and given refinement scheme is the (problem dependent) task of devising the coarsening strategy.

3 A Multilevel Algorithm for the Travelling Salesman Problem

Having motivated the approach the next question is: how can the multilevel paradigm be applied to the TSP? Clearly the LK or CLK/ILK algorithms will make a good refinement method although in principle any iterative refinement procedure including the well known 2-opt, [9], and 3-opt, [24], algorithms could be used. However, with no graph as such, how can the problem be coarsened?

In fact it seems that the crucial point in devising a coarsening algorithm is the above requirement (C1) – that the solution to each coarsened problem must contain a solution of the original problem (even if it is a poor solution). One way of achieving this is for the coarsening to successively fix edges into the tour. For example, given a TSP instance P of size N, if we fix an edge between cities c_a and c_b then we create a smaller problem P' of size $N-1$ (because there are $N-1$ edges to be found) where we insist that the final tour of P' must somewhere contain the fixed edge (c_a, c_b). Having found a tour T' for P' we can then return to P and look for better tours using T' as the initial tour. This process is once again equivalent to restricting the solution space (to all tours which contain the edge (c_a, c_b)) and in fact by fixing many distinct edges in one coarsening step we are again sampling the solution space.

Figure 3 shows an example of this. The top row demonstrates the coarsening process where dotted lines represent matchings of vertices (and hence new fixed edges) which are being made in the current coarsening step whilst solid lines represent fixed edges that have been created in previous coarsening steps. Notice in particular that from the third step onwards chains of fixed edges are reduced down to a single edge with a vertex at either end and any vertices internal to such a chain are removed. The coarsening terminates when the problem is reduced to one fixed edge & two vertices and at this point the tour is initialised. The initialisation is trivial and merely consists of completing the cycle by adding an edge between the two remaining vertices. At this point we could just expand all the fixed edges and get a legitimate tour (and in this sense the coarsening takes the place of an initial tour construction algorithm). However now the procedure commences the extend/refine loop (as shown in the second & third rows of Figure 3). Again solid lines represent fixed edges whilst dotted lines represent free edges which may be changed by the refinement. The extension itself is trivial; we simply expand all fixed edges created in the corresponding coarsening step and add the free edges to give an initial tour for the refinement process. The refinement algorithm then attempts to improve on the tour (without changing any of the fixed edges) although notice that for the first two refinement steps in the Figure no improvement is possible. The final tour is shown at the bottom left of the Figure; note in particular that fixing any edge during coarsening does not force it to be in the final tour since for the final refinement step all edges are free to be changed. However, fixing an edge early on in the coarsening does give it less possibilities for being flipped.

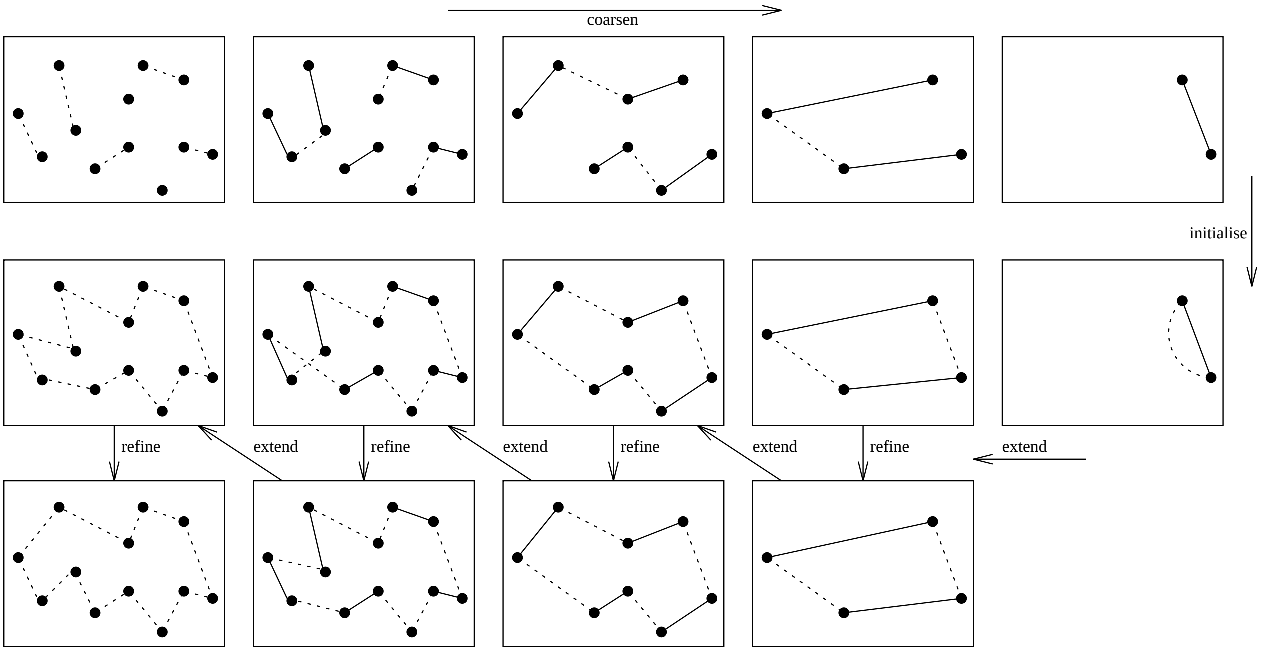


Figure 3: An example of a multilevel TSP algorithm at work

3.1 Matching and coarsening

3.1.1 Data structures

To implement the above process, although we are matching vertices and then fixing edges between matched pairs, it is more convenient within the code to regard this as the matching of fixed edges. For example, having matched vertex v_1 with vertex v_2 in Figure 4(a) we do not want vertices w_1 & w_2 (the vertices already fixed to v_1 & v_2) to match with any other vertices. Although there is no intrinsic reason why they should not, we feel that this might coarsen the problem too rapidly (by building long multi-edge fragments in a single step) and thus miss out on the benefits of the multilevel strategy. This policy also avoids the need to check that a given matching will not create a subcycle (a tour through a subset of the vertices) by matching a series of fixed edges together in a loop.

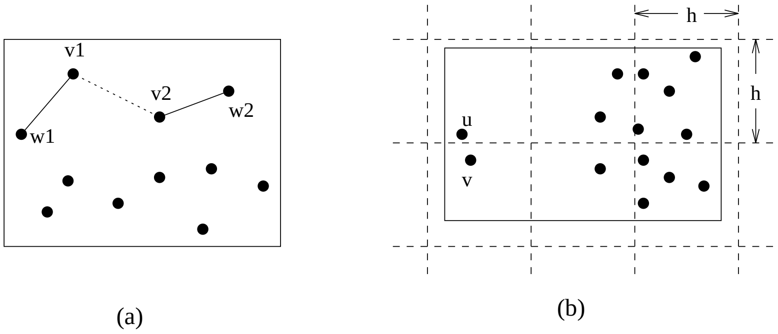


Figure 4: Matching examples

Our data structure for handling the coarsening thus consists of edge objects. Initially each edge is of zero length and has the same vertex at either end however after the first coarsening most edges will have different vertices at either end. Once a given vertex v_1 at one end of edge (v_1, w_1) is matched with another vertex v_2 from edge (v_2, w_2) then all involved vertices v_1, v_2, w_1, w_2 (although some of these may be the same) are prevented from matching again during the same coarsening step.

3.1.2 Matching

The aim during the matching process should be to fix those edges that are most likely to appear in a high quality tour thus allowing the refinement to concentrate on the others. For example, consider Figure 4(b); it is difficult to imagine an optimal tour which does not include the edge (u, v) and so ideally the matching should fix it early on in the process. Indeed, if by some good fortune, the matching *only* selected optimal edges then the optimal tour would be found by the end of the matching process and the refinement would have no possible improvements. However, in the absence of any other information about the optimal tour, we have chosen to match vertices with their nearest neighbours.

The computation of matchings of this type is known as the minimum-weight perfect matching problem and polynomial algorithms which can compute optimal matchings are available, e.g. [8]. However they are too expensive for our application (e.g. 3 minutes for a 100,000 vertex problem) and so we use an algorithm which is fast (e.g. around 4 seconds for 100,000 vertices) but which finds a matching that has only approximately minimal weight.

We have implemented this strategy with a simple data structure similar to that used previously by Bonomi & Lutton, [6], for excluding long distance edges from random perturbations in a simulated annealing TSP implementation and by Fruchterman & Reingold, [12], for graph drawing. Specifically for each coarsening level we choose a maximum matching distance h and only allow vertices to be matched with neighbours closer than this. To accomplish this we find the smallest rectangle containing all the vertices and overlay it with a square grid of spacing h, e.g. Figure 4(b). For each grid cell we place all vertices lying within that cell into a linked list in random order. While there are unmatched vertices we then randomly pick a cell containing unmatched vertices and find the first unmatched vertex, v, in the linked list for that cell. Next we locate the two closest neighbouring vertices, w_1 & w_2 within distance h of v by visiting all vertices in the cell containing v together with all the vertices in the 8 adjacent cells. We then match v with the closest unmatched vertex from w_1 & w_2 (provided of course that v is not already fixed to the chosen vertex). If there is no vertex within distance h of v, or if both w_1 & w_2 are already matched, then v is matched with itself and prevented from matching with any other vertex in that coarsening step.

It might appear that such a process (which only allows a vertex to match with its two nearest neighbours) would be destined to terminate prematurely since the two nearest neighbours of any given vertex, v, might already be contained within the interior of a tour-fragment of fixed edges. However, recall from above that any such vertices not at either end of a tour fragment are removed from the problem representation thus giving v a chance to match with vertices which are not its nearest neighbours in the original problem representation.

3.1.3 The choice of h

Virtually the only variable parameter within this matching process is h, the grid spacing. In fact it is easier to work with the average number of vertices per grid cell, n, and to calculate h on this basis. If A is the area of the smallest rectangle containing all the vertices then, in order for the average number of vertices in each cell to be n, the area of each grid cell should be An/N giving $h = \sqrt{An/N}$.

Notice that N here refers to the number of vertices in the current coarsening step. Since this value decreases at every step the grid spacing gets larger and larger until eventually, when $N < n$, the entire problem is contained in one cell. This prevents the coarsening from coming to a premature end as would happen if h were fixed and there were vertices further apart than h. It also allows matchings to take place at increasingly longer range as the coarsening proceeds.

We have experimented (using the test suite described in §4.1) with n set to 5, 10, 15 & 20. In fact there was very little difference between any of them although as n increases the matching becomes slower (as the code must check for the closest vertices in 9 cells). On the other hand, as n decreases the coarsening rate becomes slower (because matching takes place later on in the sparser areas of the problem) and so the code must refine more levels. In the end we chose to use $n = 10$ and this is the value which applies in all the experiments in Section 4.

Finally note that if n is too small one can imagine pathological cases where no vertices are within distance h of each other. However, there must be at least one cell with n vertices in it and it is easy to see that provided $n \geq 4$, even if a cell contains 4 vertices, one at each corner of the cell and with two pairs of fixed edges between them, then at least one match is possible, coarsening can take place and on the next level h will increase. This is probably true for smaller n but proof is complicated by the fixed edges (vertices at either end of a fixed edge cannot be matched with each other).

3.2 Initialisation

As suggested in §2.3, principle (C2), the coarsening ceases when further coarsening would cause a degenerate problem, in this case when there remain only two vertices with a fixed edge between them. This is guaranteed to occur because each coarsening level will match at least one pair of vertices and so the problem size will shrink. Initialisation is then trivial and consists of adding an edge between the two vertices to complete the tour (the other edge of the tour being the fixed one).

3.3 Refinement: the (chained) Lin-Kernighan algorithm

We use the chained Lin-Kernighan algorithm for the refinement step of our multilevel procedure. As stated in the introduction, the chained or iterated Lin-Kernighan algorithm is the most successful local search technique for iteratively optimising a TSP tour. It is usually combined with a tour construction heuristic which builds an legitimate initial tour. One such construction technique is Bentley's greedy algorithm, [4, 5], which proceeds by sorting all the inter-city distances by length and repeatedly adding in the shortest edge which is not already in the tour and which will not create a subcycle (a tour with fewer than N edges). In this way it progressively builds a series of tour fragments and in many ways resembles the matching & coarsening process described above (although the coarsening tends, at least initially, to create fragments with a uniform number of edges whereas the greedy algorithm has no such restriction).

Once a tour is constructed, optimisation can take place by 'flipping' edges. For example, if the tour contains the edges (v_1, w_1) & (w_2, v_2) in that order, then these two edges can always be flipped to create (v_1, w_2) & (w_1, v_2). This sort of step forms the basis of the 2-opt algorithm due to Croes, [9], which is a steepest descent approach, repeatedly flipping pairs of edges if they improve the tour quality until it reaches a local minimum of the objective function and no more such flips exist. In a similar vein, the 3-opt algorithm of Lin, [24], exchanges 3 edges at a time. The Lin-Kernighan (LK) algorithm, [25], also referred to as variable-opt, however incorporates a limited amount of hill-climbing by searching for a sequence of exchanges, some of which may individually increase the tour length, but which combine to form a shorter tour. A vast amount has been written about the LK algorithm, including much on its efficient implementation together with some additional ideas to improve its quality, and we shall not repeat it here. For an excellent overview of techniques see the survey of Johnson & McGeoch, [20], and for more details of the implementation used here see Applegate, Bixby, Chvátal & Cook, [1], and Applegate, Cook & Rohe, [2].

The basic LK algorithm employs a good deal of randomisation and for many years the accepted method of finding the shortest tours was simply to run it repeatedly with different random seed values and pick the best (a technique which also had the advantage that it could be run in parallel on more than one machine at once). Martin, Otto & Felten's important contribution to the field, [26, 27], came with the observation that, instead of restarting the procedure from scratch every time, it was more efficient to perturb the final tour of one LK search and use this as the starting point for the next. In their original approach, Martin *et al.* referred to their algorithm as chained local optimisation and used it as a form of accelerated simulated annealing. Thus they would perturb or 'kick' a tour and use LK to find a nearby local minimum. If the new tour was not as good as the champion tour at that point, the algorithm would decide whether or not to keep it as a starting point for the next perturbation by using a simulated annealing cooling schedule. Subsequent implementations however generally discard any new tour which does not improve on the current champion and always perturb the champion, [1, 2, 19, 20].

The method for perturbing the tour varies from implementation to implementation but generally involves a so-called 'double-bridge' move which exchanges four edges. This has the advantages of being simple, compact and the move cannot be undone by standard implementations of the LK algorithm. In [2], Applegate, Cook & Rohe test some different perturbation strategies and conclude that generally those which do not alter the cost too greatly are to be preferred over completely random kicks.

In this paper we used the chained Lin-Kernighan (CLK) implementation of Applegate, Bixby, Chvátal & Cook, [1], because a research version was easily accessible. However, as stated previously the multilevel procedure can, in principle, be used with any iterative refinement scheme and had a version of Johnson & McGeoch's 'production mode iterated LK' algorithm, [20], been available we would have tried that too.

3.3.1 Fixed edges

The only change we needed to make to the implementation of the CLK algorithm was to ensure that none of the fixed edges were exchanged. We enforced this by altering the subroutine which calculated edge lengths between a given pair of cities to return a large negative value whenever it was asked to evaluate the length of a fixed edge. The reasoning behind this was that such a value would make any fixed edge so unattractive for being exchanged that the code would never flip it. It should also mean that such edges are kept well away from searches looking for candidate long edges to replace. In practice we found that in all our experiments no such edges were ever flipped, however it is possible that, with a good knowledge of the LK code, a more efficient implementation might be found by simply blocking (at a high level) any fixed edges from ever being considered.

4 Experimental Results

We have tested the multilevel strategy with a summary and somewhat inefficient implementation of the matching and coarsening techniques built around a well engineered, highly optimised and very efficient research implementation of the chained Lin-Kernighan algorithm (which also handles input of the TSP instance and output of the final tour). The LK software is contained in an optimisation package written by Applegate, Bixby, Chvátal & Cook, [1], and known as `concorde`. The version that we have been using is `co991215.tar.gz`[1] and we very gratefully acknowledge its authors for making this code available, hence saving many months of work in the preparation of this paper.

The multilevel code wrapper that we have written around `concorde` is called `sierra`, both to reflect the nature of the multilevel paradigm which ascends and descends through a mountain like structure of problems, and also because it was the name of a car reputedly popular amongst sales representatives in the 1990's. To give an idea of the ease of implementation, `sierra` is written in C and contains less than 1,000 lines of code. Whilst we acknowledge that it is somewhat inefficient (although see §5.2 for possible improvements), since the vast majority of the execution time is generally spent in the execution of the CLK algorithm (and this is an inherent feature of this algorithm rather than a fault of `concorde`), we do not believe that a more efficient version would significantly improve the results.

The tests were carried out on a DEC Alpha machine with a 466 MHz CPU and 1 Gbyte of memory. For each instance and each code configuration we ran 3 tests with different random seed values.

4.1 Test suite

This paper has been written in part for the 8th DIMACS implementation challenge[2] which is aimed at characterising approaches to the TSP. As such we have used the test suite of TSP problem instances supplied there. These are in four groups:

(I) All 34 symmetric instances of 1,000 or more vertices from `TSPLIB`[3], a collection of sample TSP instances, including some from real-life applications, compiled by Reinelt, [30, 31].

(II) 26 randomly generated instances with uniformly distributed vertices. These range in size from 1,000 to 10,000,000 vertices, going up in size gradations of $\sqrt{10}$ and were constructed by Johnson, Bentley & McGeoch specifically to study asymptotic behaviour in tour finding heuristics.

(III) 22 randomly generated instances with randomly clustered vertices. These range in size from 1,000 to 100,000 and have the same origin and purpose as (II) although clustered examples such as these are generally considered to be more difficult to solve.

[1] available from `http://www.keck.caam.rice.edu/concorde/download.html`
[2] see `http://www.research.att.com/~dsj/chtsp/`
[3] available from `http://www.iwr.uni-heidelberg.de/iwr/comopt/software/TSPLIB95/`

(IV) 7 randomly generated instances with the distances specified by a matrix. These range in size from 1,000 to 10,000 and again have the same origin as (II).

Of these examples we have omitted the 8 instances, 1 from `TSPLIB` plus all 7 from category IV, which specify the problem as the upper triangular part of an $N \times N$ matrix of inter-city distances (rather than Euclidean distance). This is because our grid based matching algorithm is unable to handle instances which do not have an associated coordinate system. However, this is not an intrinsic problem of the multilevel paradigm and in §5.1 we suggest a possible alternative.

We have also omitted the two largest instances (with 3,162,278 and 10,000,000 vertices) from the uniformly distributed random category (II) as they were too large to run on our test platform.

In order for the reader to make their own analysis of the test data, in Table 4 we give the Held-Karp lower bound for each instance, the optimal tour length (if known) and the execution time, T_{LK}, for the Lin-Kernighan algorithm averaged over three runs. For each instance the name indicates the problem size, e.g. `fl1577` has 1,577 cities, `C10k` has 10 thousand cities and `E1M` has 1 million cities. The HKLB figures were downloaded from the DIMACS implementation challenge webpage and originally calculated using `concorde`, [1]. Optimal tour lengths were also downloaded from the same webpage and were either originally calculated using `concorde` (categories II & III) or, for the `TSPLIB` instances (category I), were obtained from `TSPLIB`, [30, 31].

4.2 A worked example

```
                  fixed     free        tour  cumulative
 level vertices   edges    edges      length     time
         13509             13509                 1.07
     1   13509     5494     8015                 1.51
     2    9005     4063     4942                 1.75
     3    5801     2740     3061                 1.88
     4    3659     1769     1890                 1.95
     5    2247     1100     1147                 1.99

    14      26       13       13                 2.04
    15      16        8        8                 2.04
    16      10        5        5                 2.04
    17       6        3        3                 2.04
    18       4        2        2                 2.04
    19       2        1        1   25498974      2.04
    18       4        2        2   25498974      2.04
    17       6        3        3   25303476      2.04
    16      10        5        5   25303476      2.04
    15      16        8        8   24912931      2.05
    14      26       13       13   24829781      2.06

     5    2247     1100     1147   22659916     10.41
     4    3659     1769     1890   22147276     18.17
     3    5801     2740     3061   21551965     33.58
     2    9005     4063     4942   20866993     65.23
     1   13509     5494     8015   20283439    133.19
         13509             13509   20025663    273.34
Tour Length: 20025663
Total Running Time: 273.39
```

Figure 5: An example of the `sierra` output

Before describing the large scale tests and analysing the general trends of the results we first demonstrate the nature of the multilevel CLK algorithm (MLCLK) by looking in detail at one particular example, the instance `usa13509` from `TSPLIB`. The example is illustrative but we have not examined the complete set of tests in enough detail to know whether it is truly representative. Figure 5 shows the output from `sierra` (with some intermediate lines removed) as the problem is coarsened from 13,509 vertices down to 2 and then back out to 13,509. As can be seen there are 20 levels (numbered from 0 to 19) and the coarsening rate is around 1.6, i.e. the problem size, which we define to be the number of free edges (which in turn is the number of vertices minus the number of fixed edges), shrinks by a factor of approximately 1.6 at every level.

The tour is initialised on the coarsest level and the optimisation commences on the next level down. The tour length figures shown are those at the end of a given level; the tour length at the beginning of the next level is the same as this figure. At each level the CLK algorithm is allowed N kicks or restarts where N is the problem size (the number of free edges) at that level. For example on level 3 the CLK algorithm is allowed 3061 kicks. In the following sections below we refer to this configuration as MLC^{N}LK.

In terms of runtime, notice that the problem input combined with the coarsening and initialisation only take a total of 2.04 seconds out of 273.39. In fact for this configuration over half the time is spent in the 13,509 CLK iterations on the final level.

With regards to the tour quality, it is very interesting to compare these results with the standard CLK algorithm run on the same instance and allowed $2N$ kicks (i.e. 27018). This configuration, C^{2N}LK, has almost the same runtime (in fact 276.77 seconds) and nearly the same final tour length. Most interestingly the MLC^{N}LK configuration has only achieved a tour length of 20,283,439 after 133.19 seconds when it starts on the final C^{N}LK refinement step. The C^{2N}LK configuration on the other hand surpasses this value after just 179 kicks and 4.07 seconds (reaching a tour of length 20,253,398). However, C^{2N}LK then spends a further 271 seconds to reach its final tour quality of 20,026,251 whilst MLC^{N}LK marginally surpasses this figure in just 140 seconds. We take this as evidence that backs up our speculation about the smoothing of the cost function so that the final refinement step of MLC^{N}LK is searching a more profitable region of the solution space. Thus, even though the tour quality is not exceptional at the start of the final CLK refinement, the final set of fixed edges which are released for optimisation are generally the shortest and typically the CLK algorithm finds these the easiest to optimise.

4.3 Parameter settings

As described in Section 3 the multilevel CLK algorithm has very few modifiable parameters. One is the average number of vertices, n, in each coarsening grid cell which in turn determines the grid spacing. After some initial testing, as mentioned in §3.1.3, we used $n = 10$ for all of the tests.

A second, more important parameter which perhaps deserves more thorough testing is the relative number of kicks or restarts that the CLK algorithm is allowed at each level (relative as compared to the number of kicks on other levels). In the experiments described below we set it to a user specified fraction of the problem size N (the number of free edges) at that level. Thus if the user picks $N/10$ then at each level it is allowed $1/10$th of the problem size for that level. However it could be argued that the algorithm should be allowed a greater proportion of kicks on the upper levels (especially since the problem sizes are so small and hence optimisation so fast) in order to better explore the solution space. On the other hand it could equally be argued that the lower levels should be favoured even more than they already are because they represent the original problem more closely. We have not properly investigated this issue save for some incomplete testing using the same strategy as above but redefining the problem size (and hence the number of kicks) as the number of vertices (which is typically around double the number of free edges). Experiments with this configuration provided marginally better results than those for MLC^{N}LK but, since both the penultimate and final refinement steps include all the vertices, took much longer to run and we concluded that it was not a worthwhile investment of time.

4.4 (Chained) Lin-Kernighan benchmarks

In Table 5, columns 2-4, we present example benchmark results from the `concorde` implementation of the CLK algorithm (which uses an algorithm known as *Quick-Boruvka* to generate the initial tour, [2]). For the chained variant it is important to realise that, in common with many optimisation algorithms, the more time that it is allowed, the better the solution it may be able to find (although with a rapid tail off as the algorithm starts to approach its quality limit). An important parameter, therefore, is the amount of optimisation allowed which can be specified as a time limit in seconds or, as we have used here, the number of kicks or restarts that the algorithm is given. We have expressed this as a factor of N, the problem size, and so for example we use $\mathrm{C}^{N/10}$LK to denote the configuration which allows $N/10$ kicks (and we can also then refer to LK as C^0LK). The figures in Table 5 are for C^{2N}LK configuration but more extensive results for LK, $\mathrm{C}^{N/10}$LK & C^NLK can be found in [32].

The results in Table 5 are laid out as follows. For the TSPLIB instances (category I) we report the results for each example averaged over 3 runs with different seed values. However for the randomly generated instances we average the results for each class so that asymptotic analysis is easier. For example the row labelled '`E31k (2)`' contains average values over the 3 runs for the 2 instances with 31 thousand vertices, `E31k.0` & `E31k.1` (i.e. this row is averaged over a total of 6 runs). For each different algorithmic configuration and each instance we then present the percentage excess over the HKLB and, in the next column, the percentage excess over the optimum tour length (if known). We also give the ratio of average runtime for the instance and configuration over the average runtime for the LK algorithm for the same instance (the T_{LK} figures in Table 4).

Finally, for each configuration, at the bottom of the Table we average all of the results; the HKLB and runtime results are averaged over all 3 runs and all 79 instances whilst the optimal excess figures are averaged over all 3 runs and those 58 instances for which an optimal tour is known.

4.5 A comparison of CLK and MLCLK

Denoting the multilevel versions of the LK & CLK code as MLLK & MLCLK, Table 5, columns 5-7, contains an example detailed listing of the results from the MLC^NLK configuration. Again, more extensive results (for MLLK, $\mathrm{MLC}^{N/10}$LK & $\mathrm{MLC}^{N/2}$LK) can be found in [32]. Recall from §4.2 that for each MLC^mLK variant, the number of kicks or restarts, m, refers to the problem size (the number of free edges) for the particular level and not the original problem size.

In order to compare the overall results, Table 1 contains a summary of different variants by just presenting the overall averages sorted in order of tour quality. Note again that here the average percentage excess above optimal is calculated over the 58 instances for which an optimal tour is known whilst the HKLB excess and runtime results are averaged over all 79 instances. Firstly then, we can immediately see from the Table that each MLC^mLK result is better than the corresponding C^mLK. This might not be too surprising since each multilevel variant takes longer than the C^mLK counterpart and includes a complete C^mLK run (albeit with different starting conditions). However notice that although the quality measures are sorted in order, the timings are not and impressively $\mathrm{MLC}^{N/10}$LK actually achieves higher quality results than C^NLK and is nearly 4 times faster.

Table 1: A summary of C^mLK and MLC^mLK results

configuration	Average % excess HKLB	Average % excess opt	T/T_{LK}
MLC^NLK	1.029	0.252	73.318
$\mathrm{MLC}^{N/10}$LK	1.389	0.625	9.947
C^NLK	1.497	0.763	38.585
$\mathrm{C}^{N/10}$LK	2.085	1.382	5.175
MLLK	2.536	1.751	2.542
LK	3.864	3.122	1.000

Looking at the figures in more detail there is actually a remarkable consistency. For each value of m ($= 0, N/10, N$), the MLC^mLK configuration cuts the percentage excess over the HKLB by about a third as compared to C^mLK. Furthermore, for those instances where an optimal tour is known, MLC^NLK cuts the percentage excess over the optimal tour length by two thirds as compared with C^NLK (in other words C^NLK is 3 times further from the optimum).

In order to achieve these improvements MLC^NLK and $\mathrm{MLC}^{N/10}$LK only require roughly twice the runtime of C^NLK and $\mathrm{C}^{N/10}$LK respectively. Of course any additional runtime is regrettable but this twofold increase compares very favourably with the near 40-fold average increase required (for these instances) to go from LK to C^NLK.

The additional time overhead for MLLK as compared to LK is also of a similar order, about 2.54. However MLLK has greater relative overheads than say MLC^{N}LK where most of the execution time is spent in CLK iterations. We therefore feel that this 2.54 figure is more susceptible to the improvements suggested in §5.2 and might well be considerably reduced by a more efficient implementation.

In fact it is not too difficult to give an approximate justification why the multilevel strategy should add this factor of two. Suppose first of all that the CLK algorithm were of $O(N)$ in execution time. In fact we know that it is greater than this but Johnson & McGeoch conclude that for instances of up to 1 million cities it is subquadratic, [20]. Now suppose that the multilevel coarsening manages to halve the problem size at every step. Again we know that this is an upper bound and in practice it is actually somewhat less than this (e.g. 1/1.6 in the worked example, §4.2) but experience indicates that typically this is not too far off. Let T_O be the time for CLK to run on a given instance of size N and T_M the time to coarsen and contract it. The assumption on the coarsening rate gives us a series of problems of size $N, N/2, \ldots, N/N$ whilst the assumption on CLK having linear runtime gives the total runtime for MLCLK as $T_M + T_O/N + \ldots + T_O/2 + T_O$. Again experience (and §4.2) indicate that $T_M \ll T_O$ and so we can neglect it giving a total runtime of $T_O/N + \ldots + T_O/2 + T_O = 2T_O$, i.e. MLCLK takes twice as long as CLK to run. Of course the fact that the CLK algorithm is actually superlinear and that the coarsening rate is less than 2 serve to neutralise each other in some way. Also the final CLK run of the MLCLK algorithm is likely to already have a very good starting tour which means that it should run even faster than when used as a standalone. Nonetheless this factor of two is a good 'rule of thumb'. Finally note that if the multilevel procedure were to be combined with an $O(N^2)$ or even $O(N^3)$ algorithm then this analysis comes out even better for the multilevel overhead as the final refinement step would require an even larger proportion of the total.

With this factor of two in mind we then decided to compare C^{m}LK against MLC$^{m/2}$LK using $m = 0, N/10, N/5, N, 2N$, reasoning that for each pair their runtimes should be approximately equivalent. Detailed figures are given in [32] but we summarise the comparison in Table 2. As can be seen the runtime assumptions were fairly good, particularly for larger m, and for such pairs, C^{m}LK & MLC$^{m/2}$LK, the average runtime figures are extremely close. Meanwhile the quality improvement imparted by the multilevel process is again fairly consistent with both multilevel variants cutting the percentage excess over the HKLB by over a quarter (actually 26-27%).

Table 2: A summary of results comparing the quality achieved for similar runtimes

configuration	Average % excess HKLB	opt	T/T_{LK}	configuration	Average % excess HKLB	opt	T/T_{LK}
LK	3.864	3.122	1.000	MLLK	2.536	1.751	2.542
C$^{N/10}$LK	2.085	1.382	5.175	MLC$^{N/20}$LK	1.610	0.862	6.234
C$^{N/5}$LK	1.843	1.134	8.967	MLC$^{N/10}$LK	1.389	0.625	9.947
C^{N}LK	1.497	0.763	38.585	MLC$^{N/2}$LK	1.099	0.326	38.407
C^{2N}LK	1.422	0.678	73.973	MLC^{N}LK	1.029	0.252	73.318

These tests also illustrate something further. It could be argued that the multilevel scheme MLC^{m}LK works better than C^{m}LK because for a given problem instance it is allowed more kicks in total (even though at the coarser levels the kicks are applied to a problem with large numbers of fixed edges). However assuming the coarsening rate of 2 again then MLC^{m}LK is allowed approximately $2m$ kicks. Hence the fact that MLC^{m}LK produces results 25% better than C^{2m}LK (at least for $m = N/2, N$) demonstrates that the multilevel procedure is adding some extra quality.

Figure 7(a) illustrates these results graphically by plotting the average percentage excess over the HKLB against average normalised runtime. The runtime factor of 2 is illustrated by the fact that each point of the MLC$^{m/2}$LK curve is almost directly below underneath that for the C^{m}LK curve. More importantly Figure 7(a) clearly shows the dramatic improvement in the asymptotic convergence of the solution quality.

Finally, by good fortune the average quality of MLC$^{N/10}$LK, an excess of 1.389 over the HKLB, is almost the same as (in fact marginally better than) that of C^{2N}LK, an excess of 1.422, allowing us to make a comparison based on how much time each variant requires to achieve the same quality. In fact for this set of benchmark instances MLC$^{N/10}$LK is a factor of 7.4 faster than C^{2N}LK on average, an impressive improvement.

4.5.1 Results on subclasses of the test suite

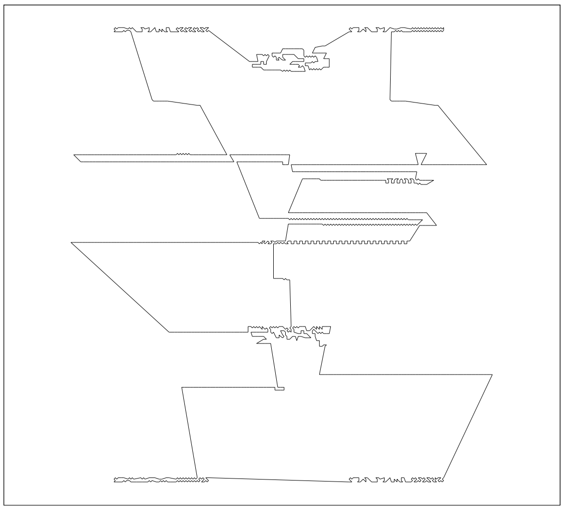


Figure 6: An optimal tour of the `fl1577` instance

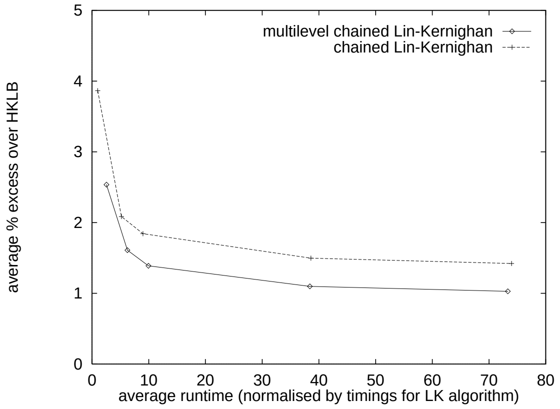
The above analysis is based on averaged results over all 79 test instances and has thrown up some interesting consistencies. However, looking in more detail at the individual instances in Table 5, the results are a little less clear. Firstly comparing C^{2N}LK and MLC^{N}LK for the uniformly distributed random instances `E`n`k` for $n = $ `1, 3, 10, 31, 100, 316,` & `E1M`, it is clear that the multilevel process only contributes a little to the quality. On the other hand, for the clustered random examples, `C`n`k` for $n = $ `1, 3, 10, 31, 100`, those which the CLK algorithm finds more difficult to optimise, the multilevel strategy significantly enhances the tour quality. This is even more striking for the real life instances `fl1400`, `fl1577` & `fl3795` from `TSPLIB`. Figure 6 illustrates an optimal tour for the `fl1577` instance which shows some of the dense clustering (this optimal tour was found by an MLC^{N}LK variant in around 16 seconds). CLK variants all have great difficulty with these examples (this is also remarked on in [1, 20] and indeed Neto has suggested improvements in the CLK algorithm to make it *cluster-aware*, [28]) and yet the multilevel variants find very good solutions (especially if one considers the percentage excess over the optimal solution which in these cases are further away from the HKLB than average).

Table 3: A summary of results on subsets of the test suite

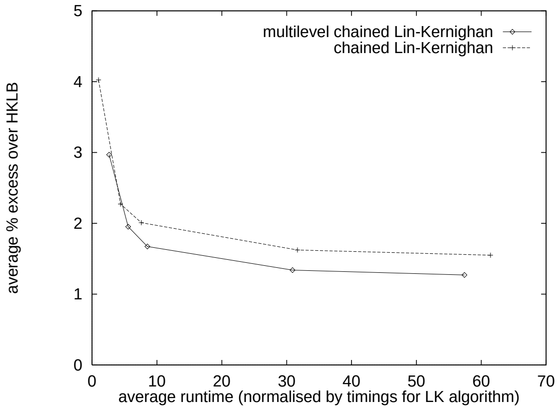
	`TSPLIB`			random uniform instances			random clustered instances		
	Average % excess			Average % excess			Average % excess		
configuration	HKLB	opt	T/T_{LK}	HKLB	opt	T/T_{LK}	HKLB	opt	T/T_{LK}
LK	4.024	3.285	1.000	2.535	1.762	1.000	5.075	4.178	1.000
$\mathrm{C}^{N/10}$LK	2.272	1.443	4.391	1.338	0.653	4.450	2.618	1.998	7.142
$\mathrm{C}^{N/5}$LK	2.009	1.167	7.596	1.162	0.473	7.356	2.338	1.732	12.781
C^{N}LK	1.623	0.747	31.661	0.953	0.255	30.860	1.903	1.302	57.397
C^{2N}LK	1.550	0.668	61.429	0.910	0.212	56.846	1.787	1.164	111.472
MLLK	2.969	2.077	2.635	2.248	1.456	2.460	2.199	1.436	2.493
$\mathrm{MLC}^{N/20}$LK	1.953	1.051	5.571	1.440	0.754	5.720	1.283	0.620	7.789
$\mathrm{MLC}^{N/10}$LK	1.673	0.752	8.530	1.239	0.542	8.844	1.128	0.471	13.275
$\mathrm{MLC}^{N/2}$LK	1.340	0.409	30.897	0.989	0.294	32.025	0.857	0.201	56.635
MLC^{N}LK	1.271	0.336	57.408	0.919	0.212	62.420	0.785	0.134	109.071

To explore this dependency on different classes of problem instance a little further we split the test suite into its three subclasses, `TSPLIB` instances (class I in §4.1), random instance with uniform distribution (class II) and random instances with clustered distribution (class III). The averaged results are shown in Table 3 (in the same format as Table 2) and illustrated graphically in Figures 7(b), 7(c) & 7(d).

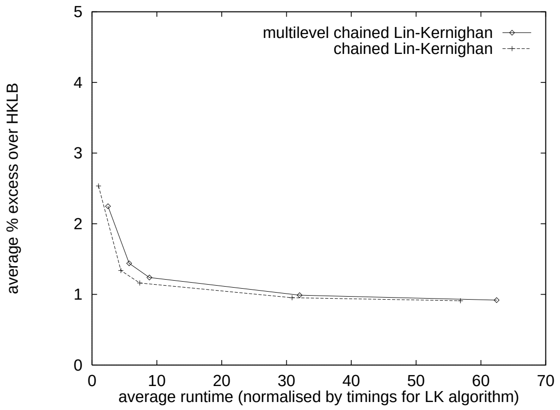
As can be seen this subdivision is highly illuminating. The `TSPLIB` instances with their wide range of examples, some from real-life applications, mirror fairly closely the results over the entire test suite, Figure 7(a), with the multilevel version achieving considerably better asymptotic convergence. For the uniformly distributed random instances however the picture is completely different. The multilevel performance is actually slightly worse initially, presumably because of the additional runtime overhead, although the asymptotic convergence looks to be about the same. Finally of the randomly clustered examples, Figure 7(d), the ability of the multilevel framework to aid the convergence is at its most dramatic.



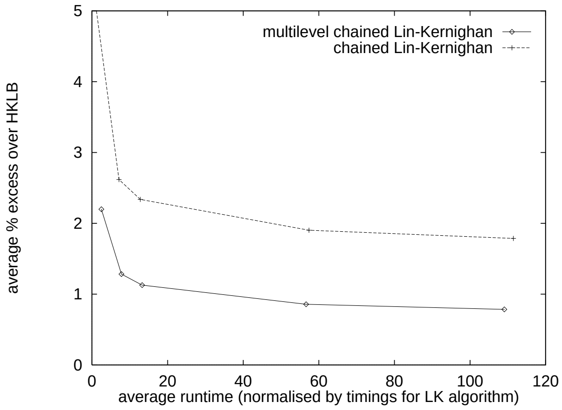
(a) complete test suite



(b) `TSPLIB` instances



(c) random uniform instances



(d) random clustered instances

Figure 7: Plot of convergence behaviour for different subsets of the test suite

It seems likely that this is because the multilevel algorithm is good at regarding clusters as a single entity, a mega-city as it were. In a high quality tour a cluster is typically only going to have one inbound edge and one outbound. The algorithm can thus concentrate on getting these longer edges correct when it has a much simpler coarse representation of the problem and then sort out the tour details within the cluster later on.

These results also give a salutary warning to practitioners about the dependency of the computational experiments on the test suite. If we had just considered random uniformly distributed instances (as some existing papers on the TSP and other combinatorial optimisation problems do) we could not have demonstrated that the multilevel framework offered any significant advantages. Clearly then algorithms need to be tested on as broad a range of examples as possible and preferably on a suite which includes real-life instances so that theoretical benefits are tested for, and hence can be realised in, practical applications.

5 Summary and Future Research

We have described and tested a multilevel approach to the Travelling Salesman Problem. The approach has been derived from first principles; by examing existing examples of the multilevel paradigm in action and extracting 'generic' techniques we have been able to apply it to a completely different problem. The resulting multilevel algorithm is shown to considerably enhance the quality of tours for both the Lin-Kernighan and Chained Lin-Kernighan algorithms, in combination the TSP champion heuristics for nearly 30 years. We speculate that this is because the multilevel process samples the solution space and smooths the objective function and is thus able to get closer to the global minimum. In this sense we regard the multilevel paradigm as a type of optimisation accelerator which here we have used in combination with the (C)LK algorithm rather than as a specific enhancement to (C)LK alone.

For the instances and code configurations tested here, the highlights of the results are:

- For 3 different C^{m}LK configurations (for $m = 0, N/10, N$) the multilevel procedure cuts the percentage excess over the HKLB by around a third in return for a modest twofold increase in runtime.
- For those instances where an optimal tour is known, MLC^{N}LK cuts the percentage excess over the optimal tour length by two thirds as compared with C^{N}LK (in other words C^{N}LK is 3 times further from the optimum).
- In two cases ($m = N/2, N$), given approximately the same amount of execution time, a multilevel configuration MLC^{m}LK cut the percentage excess over the HKLB by over a quarter as compared with a single level configuration C^{2m}LK.
- Alternatively, in order to achieve the same quality of tour, the C^{2N}LK configuration took over 7 times as long as MLC$^{N/10}$LK.
- The multilevel versions tend to do significantly better on the harder, clustered problems which the LK & CLK algorithms have the most difficultly with.

However the warning in §4.5.1 about the dependency of the results on the test suite applies and we certainly would not claim that they could necessarily be repeated for a completely different set of problem instances.

We conclude that the multilevel strategy can be a powerful tool in the solution of the TSP and that the multilevel paradigm can be successfully applied to yet another combinatorial optimisation problem. One major piece of work for further research therefore is to apply the multilevel paradigm to further combinatorial optimisation problems and examine the results (see also [35] for further thoughts on this project).

5.1 Variants and extensions

In terms of the multilevel CLK scheme, apart from optimisation of parameter settings (see §4.3), we suspect that the algorithm might benefit from further research into matching strategies. For example, one could build a Delaunay triangulation of the vertices (by constructing a Voronoi diagram as in [11]) or some other form of neighbour graph and only allow matching along its edges. Alternatively one could use any tour construction heuristic initially (such as the greedy algorithm) and force the coarsening to match only those pairs of vertices which are adjacent in this tour.

This tour construction suggestion might also be a good way to address instances where the inter-city distances are specified by a matrix rather than by Euclidean distance (e.g. see §4.1). It would certainly be more efficient than a naïve but straightforward matching procedure such as picking vertices at random, searching all $N - 1$ edge lengths to find the closest pair of neighbours and matching with one of them.

5.2 Efficiency

As mentioned in Section 4, the `sierra` implementation of the multilevel techniques is not as efficient as it could be and it is certainly possible that further work could give improvements in execution time. In particular, during the coarsening inter-city distances are calculated 'on the fly' as required and it is likely that the use of caching techniques such as those used by `concorde`, [1], would improve efficiency. Perhaps a more important enhancement though would be closer integration of `sierra` and `concorde`.

In particular concorde is called as a self-contained subroutine which creates and then deletes all of its internal data structures every call. A better strategy would be for sierra to maintain them itself and to allow concorde to reuse them. Finally, as mentioned in §3.3.1, a more efficient method for the blocking of fixed edges might prove beneficial.

It should be stressed here that these improvement suggestions all fall on the sierra implementation rather than on the concorde package which could not anticipate them. As to how much difference they might make it it impossible to say. However, as mentioned above, since the vast majority of the code execution time is spend in CLK iterations, it is doubtful that, apart from the MLLK configuration, such efficiency enhancements would significantly improve runtimes further.

Acknowledgements. The author very gratefully acknowledges the concorde team, David Applegate, Robert Bixby, Vasek Chvátal & William Cook, for making their software available, Gerd Reinelt, David Johnson, Lyle McGeoch & others for providing the test cases and the anonymous reviewers for their suggestions to improve this paper.

References

[1] D. Applegate, R. Bixby, V. Chvátal, and W. J. Cook. Finding Tours in the TSP. Tech. Rep. TR99-05, Dept. Comput. Appl. Math., Rice Univ., Houston, TX 77005, 1999.

[2] D. Applegate, W. J. Cook, and A. Rohe. Chained Lin-Kernighan for large traveling salesman problems. Tech. Rep., Dept. Comput. Appl. Math., Rice Univ., Houston, TX 77005, July 2000.

[3] S. T. Barnard and H. D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Concurrency: Practice & Experience*, 6(2):101–117, 1994.

[4] J. L. Bentley. Experiments on Traveling Salesman Heuristics. In D. S. Johnson, editor, *Proc. 1st Annual ACM-SIAM Symp. Discrete Alg. (SODA '90)*, pages 91–99. SIAM, Philadelphia, 1990.

[5] J. L. Bentley. Fast Algorithms for Geometric Traveling Salesman Problems. *ORSA J. Comput.*, 4(4):387–411, 1992.

[6] E. Bonomi and J.-L. Lutton. The N-City Travelling Salesman Problem: Statistical Mechanics and the Metropolis Algorithm. *SIAM Rev.*, 26(4):551–568, 1984.

[7] T. N. Bui and C. Jones. A Heuristic for Reducing Fill-In in Sparse Matrix Factorization. In R. F. Sincovec *et al.*, editor, *Parallel Processing for Scientific Computing*, pages 445–452. SIAM, Philadelphia, 1993.

[8] W. J. Cook and A. Rohe. Computing minimum-weight perfect matchings. *INFORMS J. Comput.*, 11(2):138–148, 1999.

[9] G. A. Croes. A method for solving traveling salesman problems. *Oper. Res.*, 6:791–812, 1958.

[10] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, Englewood Cliffs, NJ, 1998.

[11] S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[12] T. M. J. Fruchterman and E. M. Reingold. Graph Drawing by Force-Directed Placement. *Software — Practice & Experience*, 21(11):1129–1164, 1991.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[14] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoret. Comput. Sci.*, 1:237–267, 1976.

[15] D. Harel and Y. Koren. A Fast Multi-scale Algorithm for Drawing Large Graphs. In J. Marks, editor, *Graph Drawing, 8th Intl. Symp. GD 2000*, volume 1984 of *LNCS*, pages 183–196. Springer, Berlin, 2001.

[16] M. Held and R. M. Karp. The Traveling Salesman Problem and Minimum Spanning Trees. *Oper. Res.*, 18:1138–1162, 1970.

[17] M. Held and R. M. Karp. The Travelling Salesman Problem and Minimum Spanning Trees: Part II. *Math. Program.*, 1(1):6–25, 1971.

[18] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In S. Karin, editor, *Proc. Supercomputing '95, San Diego*. ACM Press, New York, NY 10036, 1995.

[19] D. S. Johnson. Local Optimization and the Traveling Salesman Problem. In M. S. Paterson, editor, *Proc. 17th Colloq. on Automata, Languages and Programming*, volume 443 of *LNCS*, pages 446–461. Springer, Berlin, 1990.

[20] D. S. Johnson and L. A. McGeoch. The travelling salesman problem: a case study. In E. Aarts and J. K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley & Sons, Chichester, 1997.

[21] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[22] B. W. Kernighan and S. Lin. An Efficient Heuristic for Partitioning Graphs. *Bell Syst. Tech. J.*, 49:291–308, 1970.

[23] F. T. Leighton. A Graph Colouring Algorithm for Large Scheduling Problems. *J. Res. National Bureau Standards*, 84:489–503, 1979.

[24] S. Lin. Computer solutions of the traveling salesman problem. *Bell Syst. Tech. J.*, 44:2245–2269, 1965.

[25] S. Lin and B. W. Kernighan. An effective heuristic for the traveling salesman problem. *Oper. Res.*, 21(2):498–516, 1973.

[26] O. C. Martin, S. W. Otto, and E. W. Felten. Large-Step Markov Chains for the Traveling Salesman Problem. *Complex Systems*, 5(3):299–326, 1991.

[27] O. C. Martin, S. W. Otto, and E. W. Felten. Large-step Markov chains for the TSP incorporating local search heuristics. *Oper. Res. Lett.*, 11(4):219–224, 1992.

[28] D. M. Neto. *Efficient Cluster Compensation for Lin-Kernighan Heuristics*. PhD thesis, Dept. Comp. Sci., Univ. Toronto, Canada, 1999.

[29] F. Pellegrini and J. Roman. SCOTCH : A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In H. Liddell *et al.*, editor, *High-Performance Computing & Networking, Proc. HPCN'96, Brussels*, volume 1067 of *LNCS*, pages 493–498. Springer, Berlin, 1996.

[30] G. Reinelt. TSPLIB— A Traveling Salesman Problem Library. *ORSA J. Comput.*, 3(4):376–384, 1991.

[31] G. Reinelt. TSPLIB95. Tech. Rep., Inst. Angewandte Math., Univ. Heidelberg, 1995.

[32] C. Walshaw. A Multilevel Approach to the Travelling Salesman Problem. Tech. Rep. 00/IM/63, Comp. Math. Sci., Univ. Greenwich, London SE10 9LS, UK, August 2000.

[33] C. Walshaw. A Multilevel Algorithm for Force-Directed Graph Drawing. In J. Marks, editor, *Graph Drawing, 8th Intl. Symp. GD 2000*, volume 1984 of *LNCS*, pages 171–182. Springer, Berlin, 2001.

[34] C. Walshaw. A Multilevel Approach to the Graph Colouring Problem. Tech. Rep. 01/IM/69, Comp. Math. Sci., Univ. Greenwich, London SE10 9LS, UK, May 2001.

[35] C. Walshaw. Multilevel Combinatorial Optimisation. To appear in Proc. Metaheuristics Intl. Conf. (MIC 2001), Porto, Portugal, 2001.

[36] C. Walshaw and M. Cross. Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm. *SIAM J. Sci. Comput.*, 22(1):63–80, 2000. (originally published as Univ. Greenwich Tech. Rep. 98/IM/35).

Table 4: Baseline results: HKLB, optimal tour lengths and T_{LK}

TSPLIB				Random instances			
instance	Held-Karp lower bound	optimal tour length	T_{LK}	instance	Held-Karp lower bound	optimal tour length	T_{LK}
dsj1000	18546976.916667	18659688	0.24	E1k.0	23183212.500000	23360648	0.17
pr1002	256765.916667	259045	0.16	E1k.1	22839567.526190	22985695	0.17
u1060	222650.875000	224094	0.18	E1k.2	22858725.666667	23023351	0.17
vm1084	236162.416667	239297	0.15	E1k.3	23002034.500000	23143748	0.18
pcb1173	56351.000000	56892	0.17	E1k.4	22542848.833333	22698717	0.15
d1291	50208.578571	50801	0.17	E1k.5	23057465.166667	23192391	0.18
rl1304	249093.833333	252948	0.19	E1k.6	23166620.333333	23349803	0.17
rl1323	265814.500000	270199	0.17	E1k.7	22666814.125000	22879091	0.17
nrw1379	56396.208333	56638	0.21	E1k.8	22795477.333333	23025754	0.16
fl1400	19783.000000	20127	0.44	E1k.9	23215285.062500	23356256	0.15
u1432	152535.000000	152970	0.20	E3k.0	40348236.083333	40634081	0.63
fl1577	21886.000000	22249	0.26	E3k.1	40046054.229167	40315287	0.58
d1655	61549.250000	62128	0.22	E3k.2	40006528.375000	40303394	0.61
vm1748	332060.722222	336556	0.27	E3k.3	40318840.516667	40589659	0.60
u1817	56688.250000	57201	0.21	E3k.4	40462881.041667	40757209	0.61
rl1889	311704.500000	316536	0.28	E10k.0	71362276.444048		2.32
d2103	79307.000000	80450	0.23	E10k.1	71565485.443519		2.36
u2152	63858.062500	64253	0.25	E10k.2	71351794.654167		2.31
u2319	234215.000000	234256	0.55	E31k.0	126474847.479514		9.21
pr2392	373489.666667	378032	0.33	E31k.1	126647285.326389		9.13
pcb3038	136587.500000	137694	0.44	E100k.0	224330692.072818		40.78
fl3795	28477.250000	28772	0.69	E100k.1	224241788.835713		39.95
fnl4461	181568.833333	182566	0.71	E316k.0	398582616.458995		159.61
rl5915	556848.833333	565530	0.85	E1M.0	708703512.913668		617.50
rl5934	548470.550000	556045	0.83	C1k.0	11325839.750000	11387430	0.28
pla7397	23126463.916667	23260728	1.31	C1k.1	11330835.500000	11376735	0.28
rl11849	913980.291667	923288	2.23	C1k.2	10809149.125000	10855033	0.28
usa13509	19851463.750000	19982859	3.33	C1k.3	11823905.791667	11886457	0.33
brd14051	467127.622222		2.69	C1k.4	11433764.375000	11499958	0.32
d15112	1564880.029167		3.75	C1k.5	11328719.175000	11394911	0.30
d18512	642116.826389		3.78	C1k.6	10092637.145833	10166701	0.28
pla33810	65705438.125000		6.45	C1k.7	10602996.291667	10664660	0.32
pla85900	141806385.000000		20.04	C1k.8	11566101.750000	11605723	0.39
				C1k.9	10835951.222222	10906997	0.37
				C3k.0	19080350.708333	19198258	1.09
				C3k.1	18901572.291667	19017805	1.05
				C3k.2	19410947.104167	19547551	1.09
				C3k.3	19001115.625000	19108508	1.08
				C3k.4	18757584.625000	18864046	1.05
				C10k.0	32782155.221284		4.01
				C10k.1	32958945.515201		3.92
				C10k.2	32926889.150397		4.00
				C31k.0	59169192.695278		15.90
				C31k.1	58840096.446393		16.53
				C100k.0	103916253.916802		67.35
				C100k.1	104663040.340307		69.78

Table 5: C^{2N}LK & MLC^{N}LK results

instance	C^{2N}LK			MLC^{N}LK		
	Average % excess			Average % excess		
	HKLB	opt	T/T_{LK}	HKLB	opt	T/T_{LK}
dsj1000	0.960	0.350	65.292	0.888	0.279	67.625
pr1002	1.295	0.404	37.812	1.375	0.484	38.938
u1060	0.866	0.217	44.667	0.859	0.209	43.222
vm1084	1.598	0.267	55.333	1.433	0.105	51.000
pcb1173	1.303	0.340	32.765	1.420	0.456	36.118
d1291	2.548	1.352	38.765	1.513	0.329	34.529
rl1304	2.361	0.802	45.737	1.643	0.095	33.842
rl1323	2.282	0.622	44.176	1.876	0.223	42.765
nrw1379	0.595	0.166	35.286	0.610	0.181	40.190
fl1400	2.411	0.661	102.727	1.867	0.126	83.091
u1432	0.661	0.375	44.150	0.709	0.423	43.950
fl1577	6.139	4.408	90.577	1.768	0.108	60.538
d1655	1.359	0.415	35.864	1.460	0.515	37.182
vm1748	1.483	0.128	55.741	1.619	0.262	54.296
u1817	1.792	0.880	33.952	1.544	0.633	32.286
rl1889	2.179	0.620	58.643	1.958	0.402	50.036
d2103	2.428	0.973	63.870	2.300	0.847	52.783
u2152	1.278	0.656	33.960	1.266	0.643	33.920
u2319	0.165	0.147	90.909	0.211	0.194	68.273
pr2392	1.739	0.516	41.121	1.744	0.521	45.303
pcb3038	1.124	0.311	41.818	1.109	0.296	47.227
fl3795	2.928	1.873	100.304	1.576	0.535	67.870
fnl4461	0.731	0.181	49.620	0.753	0.202	57.141
rl5915	2.197	0.629	67.353	1.789	0.227	51.471
rl5934	2.004	0.615	67.205	1.746	0.360	67.313
pla7397	0.842	0.260	74.809	0.919	0.337	77.420
rl11849	1.356	0.334	79.816	1.243	0.223	71.422
usa13509	0.878	0.215	83.724	0.850	0.187	86.535
brd14051	0.716		73.524	0.670		81.030
d15112	0.685		76.771	0.721		80.160
d18512	0.646		70.013	0.667		77.008
pla33810	0.971		92.936	1.082		88.819
pla85900	0.634		97.923	0.746		91.151
E1k (10)	0.935	0.197	40.579	0.957	0.219	46.652
E3k (5)	0.948	0.242	52.230	0.905	0.199	57.218
E10k (3)	0.859		75.292	0.897		79.273
E31k (2)	0.841		78.074	0.851		83.810
E100k (2)	0.868		73.495	0.878		80.353
E316k (1)	0.905		75.740	0.930		83.708
E1M (1)	0.862		92.599	0.879		95.617
C1k (10)	1.407	0.865	86.376	0.645	0.107	88.941
C3k (5)	2.385	1.761	106.280	0.804	0.189	104.770
C10k (3)	1.924		154.550	0.919		145.012
C31k (2)	1.897		154.926	1.028		138.254
C100k (2)	1.875		141.854	0.989		137.376
Average	1.422	0.678	73.973	1.029	0.252	73.318