DIMACS TSP Implementation Challenge 2000 Submission

David M. Neto

September 30, 2000

Abstract

I describe experiments run as part of the DIMACS TSP Implementation Challenge 2000. I used Lin-Kernighan and iterated Lin-Kernighan, both modified to use the cluster compensation technique developed for my doctoral research. We give a brief description of the technique, followed by a description of the machine, the experiments run, and how the results are being submitted. Finally, a special experiment showed that reordering verticies of according to a space filling curve could improve locality for runs of Lin-Kernighan on very large geometric instances.

Contents

1 Description of the algorithm

My results are for a modified Lin-Kernighan heuristic [**?**]. The base Lin-Kernighan heuristic follows the description of the Johnson*etal.* implementation as described by Johnson and McGeoch [?]. The modification is called *cluster compensation*, and is described in detail in my PhD thesis [?].

1.1 Cluster compensation

Cluster compensation aims to reduce the amount of wasted search work performed by the Lin-Kernighan heuristic on sharply clustered inputs. It provides extra lookahead to the search, augmenting the cumulative gain criterion by a negative lookahead term. By applying a negative term to the cumulative gain criterion, the search has an opportunity to stop earlier. We also applying the extra term to the greedy selection criterion, possibly reordering alternatives for the next flip so that more globally promising steps are tried first. The magnitude of the negative term is the *cluster distance* between the between the two endpoints of the current Hamiltonian path, as defined below.

The details are as follows. Let me fix the notation for the Lin-Kernighan heuristic. The heuristic searches in the underlying graph G for a cycle consisting of $2k$ edges $e_1, \ldots, e_{2k}$ with the following properties:

i Edge e_j is the pair of cities (t_j, t_{j+1}), except that edge e_{2k} is the pair of cities (t_{2k}, t_1).

ii Odd-numbered edges are in the current tour; even-numbered edges are not. Removing the odd edges from the current tour and adding the even edges results in another tour. The $2k$ edges form an augmenting cycle in the graph.

iii The total weight of the odd edges exceeds the total weight of the even edges. That is,

$$\sum_{j \text{ odd}} cost(e_j) - \sum_{j \text{ even}} cost(e_j) > 0.$$

Improving cycles t are built incrementally from t_1, by first removing edge $e_1 = (t_1, t_2)$ to form a Hamiltonian path. From then on, city t_1 anchors the head of the path while we repeatedly flip the tail of the path. The cumulative gain criterion says the cumulative alternating sum along the t sequence must be greater than the gain given by the best tour improvement seen so far in this sequence of flips, denoted by *best_gain*. In symbols, the cumulative gain criterion is:

$$cum\_gain(2i) = \sum_{j \text{ odd}, j<2i} cost(e_j) - \sum_{j \text{ even}, j \leq 2i} cost(e_j) > best\_gain.$$

The cumulative gain criterion in the usual Lin-Kernighan algorithm deliberately ignores the cost of immediately closing up the Hamiltonian path to form a tour. That is, $cost(t_{2i}, t_1)$ is not removed from the cumulative gain because there might be a way to extend the sequence a bit farther and achieve a closing-up cost much lower than $cost(t_{2i}, t_1)$.

But there's a problem with this, especially on sharply clustered instances. If the tail t_{2i} somehow crosses into a cluster of the graph distant from the cluster containing t_1, then we end up with an overly-optimistic cumulative gain. This makes the search go on for much longer than it should. We may never gain enough "potential" to cross the great divide back into the territory surrounding the anchor city t_1.

Cluster compensation discounts the cumulative gain by the *cluster distance* between t_{2i} and t_1. Given a path P from u to v, the *bottleneck cost* of P is the cost of the most expensive edge in P. Then the cluster distance between u and v is the minimum bottleneck cost over all paths from u to v. We denote this value by $c_G(u, v)$.

The cumulative gain criterion applied when considering adding t_{2i-1} and t_{2i} becomes:

$$\sum_{j \text{ odd}, j<2i} cost(e_j) - \sum_{j \text{ even}, j \leq 2i} cost(e_j) - c_G(t_{2i}, t_1) > best\_gain.$$

Lin-Kernighan also uses a greedy selection criterion. Given a number of alternatives for which pair of cities t_{2i+1}, t_{2i+2} meeting the cumulative gain criterion, we choose the pair which will result in the maximum cumulative gain, *i.e.*maximize $cum\_gain(2i + 2)$. Cluster compensation modifies this, saying choose the pair that will result in the maximum cumulative gain, discounted by the cluster distance between t_1 and t_{2i+2}, *i.e.*maximize $(cum\_gain(2i + 2) - c_G(t_1, t_{2i+2})$.

Cluster distance function c_G may be computed very efficiently. It can be shown that if T is a minimum spanning tree for G, then $c_G = c_T$. Furthermore, given tree T, with preprocessing requiring only $O(n \log n)$ and using only linear extra space, online queries of function c_T may be done in constant time, assuming fixed machine word width. The key is to use a fast least common ancestor algorithm on a tree T' constructed from T. Fast least common ancestor algorithms are described by Schieber [?].

Although cluster compensation is aimed at reducing running times on clustered inputs, it usually does the same for more uniform inputs, although to a lesser degree.

1.2 Other algorithmic details

The full backtracking width was used, up to t_6, with full width to t_8 when t_1 through t_6 does not form a legal 3-change. It is the same scheme as recommended by Lin and Kernighan [?] and re-iterated more recently [?].

Search depths were bounded at 50 flips beyond the backtracking levels, so that the longest t sequence extended to t_{108} before being forcibly cut off.

Two-level trees were used to represent the tour [?].

The code is available at `http://www.cs.utoronto.ca/~neto/research/lk`.

2 Machine and compiler characteristics

```
Machine:

    allen.cs.utoronto.ca, described in the local documentation as:
        Dell PW410 (2xPIII-550)  Redhat Linux 6.0 (2.2.x kernel)
        512KB cache each processor, 1024MB RAM shared between the processors.
        Linux kernel version is reported as 2.2.16smp by uname -a

Compiler:

    egcs-2.91.66
    Options:
        -march=i686 (producing a binary usable only on Pentium Pro and above)
        -O6
```

The benchmark results are as follows, with three runs for each benchmark:

```
bench.E1k.0.0: User time (seconds): 23.62
bench.E1k.0.1: User time (seconds): 20.53
bench.E1k.0.2: User time (seconds): 20.51
bench.E3k.0.0: User time (seconds): 21.96
bench.E3k.0.1: User time (seconds): 22.27
bench.E3k.0.2: User time (seconds): 21.95
bench.E10k.0.0: User time (seconds): 25.21
bench.E10k.0.1: User time (seconds): 28.66
bench.E10k.0.2: User time (seconds): 25.75
bench.E31k.0.0: User time (seconds): 36.56
bench.E31k.0.1: User time (seconds): 40.03
bench.E31k.0.2: User time (seconds): 36.15
bench.E100k.0.0: User time (seconds): 43.73
bench.E100k.0.1: User time (seconds): 44.46
bench.E100k.0.2: User time (seconds): 44.07
bench.E316k.0.0: User time (seconds): 48.35
bench.E316k.0.1: User time (seconds): 48.69
bench.E316k.0.2: User time (seconds): 48.61
bench.E1M.0.0: User time (seconds): 64.30
bench.E1M.0.1: User time (seconds): 64.36
bench.E1M.0.2: User time (seconds): 64.16
```

These are what one would expect, comparing to the results for the 500Mhz Pentium III posted on the web site.

3 Experiments performed

I am reporting results on several different candidate sets and numbers of iterations, in an attempt to match those parameters to the some of those reported as early results on the challenge web page:

Results are stored in files accompanying this report and have names of the following form:

out.*candidate set*.*seed*.*iterations*.plain

out.*candidate set*.*seed*.*iterations*.html

(The plain version is plain text, suitable for automatic processing, and the html verison is HTML......)

Parameter sets included combinations of:

candidate sets For example, 20 nearest neighbours for each vertex.

- **cnq3** 12 nearest quadrant neighbours,
- **cnn20** 20 nearest neighbours (for matrix problems),
- **cnq5nn20** 20 nearest quadrant neighbours union 20 nearest neighbours,
- **cnq10nn40** 40 nearest quadrant neighbours union 40 nearest neighbours,

seed This lists the three different random number seeds used to test variation in results

- **1999** for run 1
- **2000** for run 2
- **822** for run 3

iterations Where n is the number of vertices, we have:

- **1** for 1 iteration
- **1000** for 1000 iterations
- **nover10** for $n/10$ iterations.
- **n** for n iterations

3.1 Missing runs

Not all combinations of the above parameters were performed. This section describes some of the missing runs to watch out for when analyzing my data.

First, only seeds 822 and 2000 were for run instance C100k.1 with cnq10nn40 candidates and 1 iteration.

Second, of the nover10 runs for cnq10nn40, only 23 instances were used for seed 822, whereas 75 instances were used for seeds 1999 and 2000.

Third, only 36 instances were run for cnq5nn20 at n iterations and seed 1999. I preferred to deliver other parameter sets in time for the deadline. If you like I can grind away the rest of them.

Of course, I ommitted some of the larger instances from the runs with larger candidate sets and more iterations.

4 Reported fields

The fields reported are the four required fields, plus several more describing the breakdown of running time into phases. All times are user times in seconds.

The fields in order are:

instance Name of the instance

length Tour length produced

user_time Total user time in seconds

memory(MB) Maximum resident set size (RSS), in megabytes. On the version of Red Hat Linux used, `getrusage` is supposed to report this figure, but reports 0 always. Instead, I extracted the number from the special `/proc` filesystem. For example, for process with id 1234, the maximum RSS page count and the page size are two fields in the output of `cat /proc/1234/stat`. See `man proc` on a Linux machine. These RSS figures computed in this way were verified visually with the display of the `top` program .

pct_over_HK Percentage excess over the Held-Karp lower bound.

read The time to read the instance. (When Moore's space filling curve is used to sort the points, see Section 5, then this `read` time also includes the time to do that sorting.)

2-d/data_struct For geometric instances, the time to construct the k-d tree. For matrix instances, the time to build initial data structures.

CC_MST The time to construct the minimum spanning tree T for cluster compensation

CC_MST_proc The time to construct the associated tree T' from T, and preprocess it for fast least common ancestor queries later.

candidates The time to construct nearest neighbour lists.

greedyTSP The time to construct an initial tour using the greedy algorithm (multi-fragment). I used the randomized variant, where the best candidate edge is chosen with probability 2/3, and the second-best candidate edge is chosen with probability 1/3.

LK The time to run the Lin-Kernighan optimization phase, employing cluster compensation modifications. This counts all iterations, whether 1 through N.

5 A special experiment regarding the memory hierarchy

In the 12 quadrant neighbour case, with one iteration, I was rather disturbed to see that on instance E3M.0 more than 2/3 of the time (5300s) was spent in building the candidate sets. My code in that phase probably needs tuning for large data sets.

However, I wondered whether memory hierarchy effects were making things worse than they could be. My implementation has an option to renumber the cities in a 2-d instance so that they respect the order on a bounding square as given by Moore's space-filling curve on that square. (Moore's curve is a 'closed' variant of Hilbert's curve.)

This renumbering changes the position of a city in any array in which the cities or their related data structures are stored. I have reason to believe that by sorting them in according to Moore's curve, that locality of reference will improve. I believe Lin-Kernighan has a tendency to spend cycles in confined geometric areas, and Moore's curve maps confined areas on the line into confined areas on the square. (And if you're not too careful, you can think of the inverse being true as well.)

So for E1M.0 and E3M.0 with 12 quadrant neighbours (cnq3) and one iteration, I re-ran the experiments for the three seed values, with the space-filling curve option on. The results are listed in files

```
out.cnq3.1999.1.sfc.{html|plain}
out.cnq3.2000.1.sfc.{html|plain}
out.cnq3.822.1.sfc.{html|plain}
```

These correspond to the E1M.0 and E3M.0 runs in

```
out.cnq3.1999.1.{html|plain}
out.cnq3.2000.1.{html|plain}
out.cnq3.822.1.{html|plain}
```

It was my pleasure to see that the running times were reduced by about one third, yet the difference in excess over Held-Karp was at most 0.01%. This is consistent across the seed values.

6 Acknowledgements

I'd like to thank Derek Corneil for supervising my thesis work, and for bringing this implementation challenge to my attention. Part of this research was supported by the Natural Sciences and Engineering Research Council of Canada, and by the Department of Computer Science at the Univeristy of Toronto. The experiments were run on a CSLab compute server at the Department of Computer Science, UofT. I'd like to thank Dennis Hui for interesting discussions on space-filling curves.

7 References