1-1989

Euclidean Traveling Salesman Heuristics

Ron Greenberg
Rgreen@luc.edu

Cindy Phillips

Joel Wein

Recommended Citation

Ron Greenberg, Cindy Phillips, and Joel Wein. Euclidean traveling salesman heuristics. In Connection Machine Projects Technical Report MIT/LCS/RSS 4, Laboratory for Computer Science, Massachusetts Institute of Technology, January 1989.

6.849J/ 18.436J Project Report

Ron Greenberg, Cindy Phillips, and Joel Wein

April 13, 1987

1 Introduction

We implemented a series of Euclidian traveling salesman heuristics. We assume for the sake of brevity that the traveling salesman problem (TSP) is sufficiently well known that it needs no description. Throughout this discussion, the variable n refers to the number of cities.

The general strategy we adopted was to generate an initial tour of varying quality, and then "tweak" it to two-optimality as described in sections 2 and 3. The cities were generated randomly and locally, using the *lisp command random!! to generate x and y coordinates between 1 and n, the number of cities. This yields a probability of $1/n$ that two cities will have identical coordinates. The only serious data structure issue was how to represent the tour. Throughout most of our code we represent the tour by two pvars (tour-x!! and tour-y!!) containing the x and y coordinates of the city represented by that processor. The tour is formed by passing through the cities in processor order. Then we refer to the "edge" in a given processor, we mean the edge of the tour that originates at the city stored in that processor. The space cost of representing the edges explicitly was prohibitive; we assume a complete graph on the n generated cities. Distance between the cities is calculated by the standard Euclidean metric.

2 Building the Initial Tour

2.1 Background

Once one assumes a triangle inequality on the set of distances between the cities, there are several known heuristics that are guaranteed to produce a tour of length at worst some small constant factor longer than the optimal tour length [2,3,6]. One standard approach is based on the construction of a minimum spanning tree (MST). Given the depth first traversal of such a tree one can "shortcut" around the repeated nodes to construct a tour of all the cities. This tour is no more than twice as long as the optimal tour.

Christofides improved this result by constructing a minimum spanning tree and then a minimum weight matching on the odd-degree nodes. He then constructs an Euler tour of this graph and shortcuts it to a tour. The resulting tour length is no worse than 3/2 the optimal length.

These techniques seem well suited for parallelization, since building mininium spanning trees and min-weight matchings is possible in polylogarithmic time. The minimum spanning tree methods prove not to be useful for us, however, since they assume one processor per edge. Working only with a representation of the city positions, we saw no way to develop a poly-logarithmic time algorithm for the minimum spanning tree. Essentially the troublesome step reduces to "Given n processors each representing a point in space, find the closest point to each point." We can, of course, gain from parallelism a factor of n.

Given this situation, one can either use $O(n)$-time heuristics, or look for other poly-logarithmic techniques. We pursued a small subset of each.

The simplest TSP heuristics are "greedy" heuristics, that at each step choose the city that is "best" in some sense — closest city, lowest cost inserted city, etc. [2,6]. In the nearest neighbor heuristic, for example, at each step the city closest to the last city on the tour is added. This heuristic is easily implemented in $O(n)$ parallel ti Rosenkrantz, Stearns, and Lewis proved that in the worst case the ratio of the leng of nearest-neighbor to optimal tours can grow as $O(\lg n)$ and that in fact this rate be achieved [6].

However, the experimental results of Bentley and Saxe [2] indicate that quite often the results of the Nearest Neighbor Algorithm are comparable to or better than those of the MST heuristic. Given the ease of implementation of greedy techniques and their comparable running times and experimental performances, it seemed worthwhile to investigate these greedy heuristics.

2.2 Greedy Heuristics

All of the greedy heuristics can be described within the following general framework:

0. Start the tour at an arbitrary city.
1. Choose a city to add to the tour.
2. Choose a position in the tour at which to add the city.
3. Add the city, and return to step 1.

For this reason, most of the greedy heuristics are performed by a single *lisp function which calls subroutines to perform functions involved in choosing the cities and positions and making the insertions. The exception is the nearest neighbor-heuristic, which was coded separately since it requires significantly less work to perform the relevant subfunctions.

The simplest of the greedy heuristics is the nearest-neighbor heuristic. This heuristic involves starting the tour at an arbitrary city and performing $n-1$ iterations of adding to the end of the tour the city nearest the last one visited. This is very straightforward to implement both sequentially and in parallel. The sequential running time is $O(n^2)$; in parallel, it is $O(n)$. Furthermore, the constant in the running time is smaller than for the other heuristics to be discussed. It will also be seen that although the nearest-neighbor heuristic does not generally provide as short a tour as some of the other greedy heuristics, it responds well to tweaking.

The other heuristics can be classified as insertion or addition heuristics. Most are discussed in Rosenkrantz, Stearns, and Lewis [6]; two are obvious extensions. All of these methods involve choosing a city for insertion which is best in some sense relative to the tour constructed so far. In nearest-insertion and nearest-addition, we choose the city which is closest in distance to the tour constructed so far (where distance to the tour is the minimum of all the distances to points on the tour). In farthest-insertion and farthest-addition, we choose the city which is farthest in distance from the tour. In cheapest insertion we choose the city for which we incur the least cost by adding it to the tour in its best place. In random insertion, we just pick the next random city not in the tour. There are two ways of deciding where to insert the selected city into the tour, which is what creates the distinction between insertion and addition heuristics. In insertion heuristics, the chosen city is inserted in the position in the tour which incurs the least cost, while in addition heuristics the city is added at the position which was responsible for making it best. For example, in nearest-addition, we add the closest city to the tour at the position of the tour which is closest to it (even though adding in another position could be less costly).

All of the insertion and addition heuristics can be implemented in $O(n^2)$ time sequentially and $O(n)$ time in parallel. This is easy to see for the addition methods. For insertion methods, we just use a technique mentioned in Rosenkrantz, Stearns, and Lewis with regard to nearest-insertion. Namely, instead of recomputing the distance of all cities to the tour after each insertion, we have each city keep track of its distance to the tour and just perform a constant time update of each city's distance by checking its old distance against its distance to the most recently inserted city. Rosenkrantz, Stearns, and Lewis recognized only an $O(n^2 \lg n)$ (sequential) algorithm for cheapest-insertion, but, in fact, the same technique is applicable to cheapest-insertion.

There are only a couple of implementation details of the insertion and addition heuristics which are at all worth mentioning, since the translation from the descriptions above to *lisp is quite straightforward. One detail which might be useful is that the basic index-sort type of technique was used. That is, instead of shifting the city data down the sequence of processors in order to accomodate the insertion of a city, each processor (city) simply keeps track of what position it belongs in as the tour is constructed. Then at the end, a single routing operation is used to put the cities in the desired tour order. Another technique which we have found to be generally useful in *lisp programs is to avoid multiple references to data of some other processor if that data can be stored locally for several future references. With these techniques, we avoid costly communication operations as much as possible.

There are a few interesting observations about the general performance of the various greedy heuristics. Nearest-neighbor is fastest, as would be expected. The addition methods take about twice as long as nearest-neighbor, and the insertion methods take about 6 or 7 times as long. Since the addition methods generally yield the longest tours, they seem to offer no advantage over use of the nearest-neighbor heuristic. Only if they were generally more amenable to tweaking, would they be useful. We did encounter one example of this behavior among our test cases, but this is probably not true in general. The greedy heuristic which yields the shortest tour is almost always farthest-insertion (surprisingly, better even than cheapest-insertion). The tours constructed by farthest-insertion are, however, generally less amenable to improvement through tweaking than are the other tours, so it is not necessarily best when tweaking techniques are included in the set of techniques.

2.3 Partitioning Heuristics

In situations where the data points are uniformly distributed in the plane, more local techniques for initial-tour building can provide expected polylogarithmic performance.

A technique that can be executed extremely quickly on the Connection Machine is building a *strip tour*. We consider the points distributed in an $n \times n$ square. We partition the square into $\sqrt{n}$ vertical strips, either letting each strip contain $\sqrt{n}$ points, or by letting each strip be of physical width $\sqrt{n}$. We then sort the points in each of these strips by y coordinates. The strip tour is the result of starting at the bottom left hand corner, constructing the tour that goes up the first strip then to and down the second, up the third, etc. The last point is joined to the first by one long edge. Building this tour is quite simple on the CM. We start with an initial random tour in two pvars, one for x coordinates, one for y and proceed as follows:

1. Sort the tour by x coordinates. (Make sure to take the y coordinates to the right places)
2. Set up a boolean segment pvar to mark the boundaries of the strips.
3. Now do a segmented sort , *i.e.* sort in each strip by y-coordinates. (An easy way to accomplish this is to add $n * i$ to each value, where i is the segment number, and then do a regular sort.)
4. "Flip" the odd numbered segments – to set up the tour order so that we are traversing ***down*** those segments.

The result is the strip tour. It requires two *sort operations and one additional reasonably local routing operation. The time of execution is accordingly extremely fast. On a 16K machine we constructed a strip tour for a 65,536 city world in .99 seconds.

Although the length of the strip tour is bounded above by $2n\sqrt{n}$ for an $n \times n$ square, it performs significantly worse than the greedy heuristics. Thus the time gained in the initial tour construction is lost in tweaking this worse initial tour to two-optimality.

We did not have time to implement what ultimately would be the most promising method of initial tour generation: Karp's partitioning algorithm [4]. This algorithm can roughly be understood as follows:

1. Divide the original area into 2^k subrectangles.
2. Construct an optimal tour in each subrectangle.
3. Combine these local tours through shortcutting.

This algorithm can be implemented in $O(\lg N*(\text{time of constructing each local tour}))$. If we were to partion in such a way as to expect $O(\lg N)$ cities per square, constructing the exact local tour in each square would take us back to expected polynomial time. If we were again to use a heuristic to construct the local tours, (such as nearest neighbor), we would have a polylogarithmic heuristic that would be expected to be more accurate than the strip tour.

3 Tweaking

In this section we discuss the implementation of tweaking heuristics. We first define the tweaking operation. We then discuss three implementations of tweaking. The first implementation finds tweaks in parallel but performs them sequentially. The other two implementations perform tweaks in parallel.

After building an initial tour, we perform *tweaking* based on the *quadrangle inequality*. More specifically, suppose we are given some directed tour of the cities such as the tour illustrated in figure 1. In this tour, it is better to replace the long pair of crossing edges (a, b) and (d, c) with the edges (c, b) and (d, a). The direction of the directed path T_2 must be reversed. We call such an edge replacement with path reversal a *tweak*. A tweak is *advantageous* if by performing the tweak we reduce the length of the tour. If no pair of edges can be tweaked advantageously, we say the tour is *2-optimal*. We repeatedly perform advantageous tweaks until a termination criterion is satisfied. We implemented tweaking with two termination criteria. If the user furnishes an optional parameter indicating the desired percentage improvement p, then tweaking stops when the tour is p percent shorter than the initial tour or when the tour is 2-optimal. Otherwise, if no percent-improvement goal is specified the tour is tweaked until it is 2-optimal.

One might at first believe that by considering the tour undirected, allowing traversal in either order, path reversal in tweaks is unnecessary. If the tour in figure 1 were undirected, however, then we could not distinguish between the desired edge pair (c, b), (d, a) and the edge pair (b, d), (a, c) which shortens the tour by chopping it into two smaller disjoint tours. In a directed tour, to tweak edges (head1, tail1) and (head2, tail2), we merely tie head1 to head2 and tail1 to tail2 and reverse the path from tail2 to head1.

The tour is represented as a pvar of cities, or more precisely two global pvars tour-x!! and tour-y!! containing the x and y coordinates respectively of the cities in the plane. Consecutive cities in the pvar represent consecutive cities on the tour with wraparound. The tweaking routine operates as a procedure (as opposed to a function). It interpretes the cities in tour-x!! and tour-y!! as an initial tour and modifies them during the tweaking.

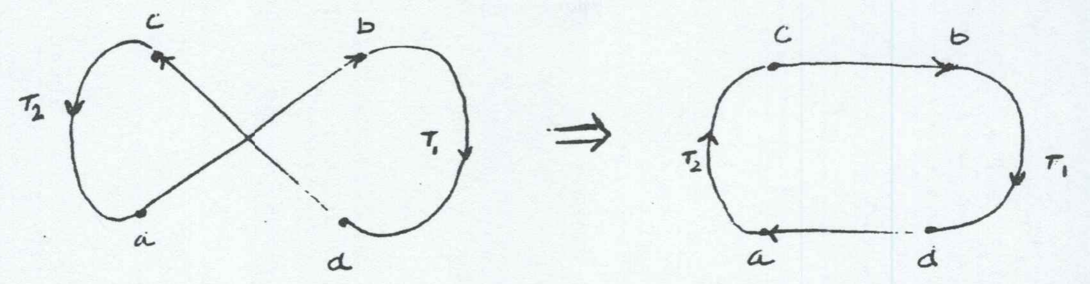

Figure 1: We tweak a pair of edges by replacing them by two edges, one that joins the heads and one that joins the tails. Half the tour is then reversed to yield a new directed tour. A tweak is advantageous if by performing the tweak we reduce the length of the tour.

In our tour representation, a tweak consists of flipping a set of consecutive entries in the city pvar. For example, to perform the tweak illustrated in figure 1, we take the piece of the tour pvar tour-x!! (similarly tour-y!!) from processor c to processor a inclusive and reverse it. Alternatively, we can reverse from processor b to processor d inclusive to get the directed tour going in the opposite direction. Thus we can always tweak by flipping a consecutive set of cities in the pvar representation without worry of wraparound. To actually perform the reversal, we broadcast the addresses of the processors at the ends of the path to be reversed. All cities then determine if they must move and if so what processor they must move to. Then all cities ship themselves off if necessary with a *pset.

The choice of edge pair to tweak is also heuristic. Initially at least it seems that tweaking the longest edges will maximize our gain. Using this assumption, in our simplest, "global" tweaking routine we chose an edge pair as follows:

1. make all edges in the tour active. If edge (x, y) is in the tour, then the processor containing city x controls edge (x, y).
2. Find the edge (a, b) of maximum length quickly using the wired OR (*max).
3. Broadcast (a, b).
4. Each city c which is followed by city d in the tour calculates the incremental change to the tour caused by tweaking (a, b) and (c, d).
5. Choose the most advantageous tweak using the *max function. If the best tweak is advantageous, perform that tweak and go back to step 1. If no tweak is advantageous, go back to step 2 and inactivate edge (a, b).

The tour is 2-optimal if no active edges remain. The only way we can see to deterministically find the tweak with the maximum gain in constant time is to use $\Omega(n^2)$ space and check each pair of edges in parallel. Unfortunately the space cost is prohibitive. We could determine the best tweak in $\Theta(n)$ time but we felt it was better to do $\Theta(n)$ good tweaks in that time than one great tweak.

We implemented two versions of the tweaking heuristic that allow limited parallization of tweaking. We discussed in our proposal some of the difficulties of parallization, for example, calculation of the tour resulting from parallel overlapping reversals and efficiently choosing in parallel a set of good tweaks. We chose the simplest form of parallelization, namely we divided the tour into disjoint sets of consecutive cities and performed the tweaking procedure in parallel on each set of cities. For each group of cities, we find the maximum-length active edge and perform the most advantageous tweak with that edge, if any. For this simple division, tweaks do not interfere. In a *parallel tweak*, each group performs the best tweak that it found in the last iteration of the tweak-finding procedure provided that the tweak is advantageous. We only perform parallel tweaks when at least one group has found an advantageous tweak. By dividing into groups, we can determine and perform multiple advantageous tweaks at the same time, but each iteration runs more slowly. Each global wired-OR operation (such as *max) and each broadcast is replaced by a segmented scan. We estimated that a parallel iteration runs about 32 time slower than a global iteration so we never perform parallel tweaking on less than 32 groups.

Our first attempt at parallel tweaking is a one-pass version. We divide the cities into groups of 4, and perform tweaks in parallel on each group until all groups are 2-optimal. We then double the group size and repeat the process until there are less than 32 groups. We then do a straight global 2-optimization. As with the global tweaking routine, all parallelized versions will accept an optional argument specifying a satisfactory percentage improvement and the routine will terminate if the tour achieves the desired percentage improvement.

We also implemented a parallel tweaking procedure that operates in multiple passes. For the multiple-pass version, we start by performing some tweaks at the top level. Then we divide the cities into 32 groups of size $n/32$ and perform some number of parallel tweaks at that level. Then we halve the group size, and so on until the groups are of size 2. We then go back to the global level and continue to smaller groups, with the groups now shifted. If the groups are of size g, then we tweak until all the groups are 2-optimal or until we perform $g/4$ parallel tweaks, or until we have achieved the desired percent improvement if any. If we are trying to make the tour 2-optimal, the procedure must end at the top level (*ie.* the global tweaking). In the best case, we perform $n/4$ tweaks at each level so we achieve $\Theta(n \lg n)$ tweaks in $\Theta(n)$ time. That is, the overall time is dominated by the time spent at the global level since the time spent on each level is roughly geometrically decreasing, and yet we can possibly perform the same number of tweaks on each level. We chose to go from global tweaks to tweaks at smaller levels because we envisioned that the globally best tweaks would be performed first, bringing cities closer to where they would ultimately be, and then local tweaks would act as fine-tuning.

The two parallel versions of tweaking have large pieces of code in common. We chose to write the two procedures as separate functions rather than using macros because macros can be very difficult to test and can waste a lot of time compiling during debugging. Similarly, we chose to make the *lisp termination statements more complicated rather than use macros to distinguish versions that do straight 2-opted vs. versions that also look for a goal percentage. The cost in lisp time for the extra tests is probably minimal. We did not do any "speed hacking" using Paris code, etc. We aimed for readable, easily debugged code that had no obvious inefficiencies (using, for example, *pset rather than *pref).

Tweaking a tour to 2-optimality required much less time if the tour was already near optimal. Tweaking a random tour required at least twice as much time as tweaking a tour built by one of the initial tour heuristics. Of course we expected much poorer behavior on inferior initial tours because many more tweaks are required to reach 2-optimality. If we are actually tweaking to full 2-optimality (no goal percentage), we spend $O(n)$ time just to verify the termination criterion. By the final stages when we are performing mostly local tweaking, we spend $O(n)$ time finding a pair to tweak, since most of the longest-length edges have stabilized. In the worst case, it seems that 2-opting can require more than $O(n^3)$ time since finding a tweak can be $O(n)$-time in the worst case and we can perform more than $O(n^2)$ tweaks. That is, pairs of edges may be tweaked more than once, although not consecutively. In practice, we observed about $O(n^2)$ behavior.

Had we had more time, we would have implemented the Lin, Kernighan generalization of tweaking called k-opting [5]. Their algorithm exchanges k edges of the tour, determining both k and the edge-sets to exchange on the fly. We anticipate that the time spent determining a k-tweak will increase substantially over the time to find a 2-tweak, but the number of tweaks may also diminish substantially.

4 Results and Conclusions

In this section we attempt to draw some conclusions about the best strategies for achieving near-optimal solutions to the Euclidian TSP on the Connection Machine. We begin by presenting the results of our limited testing in tabular form. We then suggest possible explanations for the results we achieved and speculate about which methods might be best asymptotically in terms of both time and final tour length. All of our conclusions about performance of initial tour methods, etc. are implicitly prefaced by the statement "for a uniformly distributed set of points ..." We did not have time to implement any other distributions of initial tours.

Our results appear in a table at the end of this report. The first part of the table presents the results of all of our basic methods on test cases of 128 and 256 cities. More specifically, we apply every initial tour method and then tweak them to 2-optimality using the global tweaking (tweaks performed sequentially). The results are presented in increasing order of final tour length. Next we selected the most promising initial tour methods for testing on larger problems with more sophisticated tweaking approaches. We list the results of one-pass and multiple-pass 2-optimal tweaking as well as parallel tweaking to a given goal percentage improvement. We conclude with the results of strip tours performed on worlds of 16k and 64k cities.

The features of the Connection Machines that proved most valuable were the fast max, min, and broadcast and the scanning facility. Due to our dense problem representation, chosen to allow us to attack large problems, the advantage of the Connection Machine often consisted primarily of a factor of n improvement resulting from the fast max, min, and broadcast operations. In the construction of strip tours and the attempts at parallel tweaking, the novel facilities of the Connection Machine were used more fully.

We encountered trade-offs in the choice of initial-tour construction method. Farthest insertion seemed to produce tours that were nearly 2-optimal and quite good even compared to 2-opted versions of the other methods. As the problem size grew, however, we were able to beat farthest-insertion in both time and tour length using parallel tweaking with a suitable goal percentage starting from a faster initial tour construction method.

As seen from the relatively small running times of tweaking with percentage goals, It appears that most of the gain from tweaking occurs early in the procedure. Unfortunately, it is quite difficult to decide upon a reasonable goal percentage. The percentage improvement depends upon the quality of the initial tour. Also, different initial tours of similar quality can lead to different percent improvement depending upon which local minimum the tweaking routine arrives at. If the user is too pessimistic, the routine will run very fast, but the improvement will not be anywhere near as good as it could be. If the user is too optimistic, the routine may end up doing straight 2-opting (*ie.* tweaking until the tour is 2-optimal). The situation is particularly delicate since, according to our limited testing, the fall-off in gain vs. time is very steep near the ultimate percent gain, but the cost of additional percentage gain is very small even up to, say, 2 percent off the ultimate gain. Of course the user does not want to 2-opt in order to determine a good goal percentage, so we conclude that it might have been better to monitor the gain of the system and stop when the average gain over a "reasonable" period of time is "small", or until the search for an advantageous tweak requires, say, $O(n)$ time.

As the size of the problem grew, the multiple-pass tweaking seemed to perform the best of the three tweaking routines. We feel that the multiple-pass tweaking could be made more efficient by modifying the termination criteria as described above, and perhaps by reducing the time spent on each level. For the relatively small problems that we tested, the routine did not perform many passes. That is, the routine only went from top-level down to small groups two or three times. We would have to experiment to determine a good trade-off between number of passes and time spent in overhead for the transitions between computing phases. With so few passes, we pay much more for "just missing" 2-optimality at the top level. We must complete the whole pass before we recognize that we are done, and with so few passes, the relative time wasted is large.

It seems that for medium-sized problems of a few hundred to a few thousand cities the most promising of the approaches we tried is the nearest-neighbor initial tour followed by multiple-pass parallel tweaking. The blazing speed of the strip tour leads us to believe that for large problems we will benefit from a similar expected log-time heuristic such as the Karp partitioning method. Hopefully, such a method not only would be very fast but also would produce much better tours than the strip method and be amenable to parallel tweaking.

References

[1] J. L. Bentley, "A case study in applied algorithm design: The traveling salesman problem," unpublished manuscript, April 1983.

[2] J. L. Bentley and J. B. Saxe, "An analysis of two heuristics for the Euclidean traveling salesman problem," *Proceedings of the Eighteenth Annual Allerton Conference on Communications, Control, and Computing*, Monticello, IL, October 1980, pp. 41–49.

[3] D. S. Johnson and C. H. Papadimitriou, "Performance guarantees for heuristics," in *The Traveling Salesman Problem* (Lawler, Lenstra, Kan, Shmoys, Editors), 1985, pp. 145–180.

[4] R. M Karp, "Probabilistic analysis or partitioning algorithms for the traveling-salesman problem in the plane," *Mathematics of Operations Research*, Vol. 2, No. 3, August 1977, pp. 209–224.

[5] S. Lin and B. W. Kernighan, "An effective heuristic for the traveling-salesman problem," *Operations Research*, Vol. 21, pp. 498–516.

[6] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "An analysis of several heuristics for the traveling salesman problem," *SIAM Journal of Computing*, Vol. 6, No. 3, September 1977.

NO. OF CITIES: 128

Tour Type	Tour Length	Initial Tour Time Total (CM%)	Additional Time Total (CM%)	Total Time (sec.) Total (CM%)
RANDOM	8302.784			
2-OPT FARTHEST-ADDITION	1177.5277	3.372300 (33.22%)	29.077866 (35.41%)	32.450165
2-OPT NEAREST-ADDITION	1183.0123	3.344416 (33.26%)	12.581933 (35.25%)	15.926349 (34.83%)
2-OPT	1217.7987		56.505714 (35.24%)	56.505714 (35.24%)
2-OPT RANDOM-INSERTION	1222.7161	6.377315 (47.48%)	13.126816 (35.20%)	19.504131
2-OPT FARTHEST-INSERTION	1222.8916	7.715856 (44.83%)	11.225888 (35.91%)	18.941744
2-OPT NEAREST-NEIGHBOR	1224.2448	1.136359 (35.67%)	12.336342 (35.24%)	13.472701
2-OPT CHEAPEST-INSERTION	1237.9708	7.436869 (45.08%)	14.180634 (35.93%)	21.617502
FARTHEST-INSERTION	1242.6844	7.715856 (44.83%)		7.715856 (44.83%)
RANDOM-INSERTION	1263.5787	6.377315 (47.48%)		6.377315 (47.48%)
2-OPT NEAREST-INSERTION	1278.4491	7.743375 (44.64%)	12.763078 (35.50%)	20.506453
CHEAPEST-INSERTION	1326.713	7.436869 (45.08%)		7.436869 (45.08%)
NEAREST-INSERTION	1379.6909	7.743375 (44.64%)		7.743375 (44.64%)
NEAREST-NEIGHBOR	1483.7495	1.136359 (35.67%)		1.136359 (35.67%)
NEAREST-ADDITION	1592.5822	3.344416 (33.26%)		3.344416 (33.26%)
FARTHEST-ADDITION	2255.458	3.372300 (33.22%)		3.372300 (33.22%)

NO. OF CITIES: 256

Tour Type	Tour Length	Initial Tour Time Total (CM%)	Additional Time Total (CM%)	Total Time (sec.) Total (CM%)
RANDOM	34790.285			
2-OPT NEAREST-NEIGHBOR	3174.9563	2.394484 (36.33%)	47.698112 (34.74%)	50.092598
2-OPT RANDOM-INSERTION	3271.2417	12.842165 (47.46%)	48.390713 (35.16%)	61.23288
2-OPT NEAREST-ADDITION	3282.564	6.700523 (33.82%)	61.337593 (34.76%)	68.03812
2-OPT FARTHEST-INSERTION	3298.053	15.379892 (44.94%)	27.496212 (34.84%)	42.876106
FARTHEST-INSERTION	3312.3005	15.379892 (44.94%)		15.379892 (44.94%)
2-OPT FARTHEST-ADDITION	3335.6665	6.692418 (33.40%)	144.704160 (34.66%)	151.39658
2-OPT CHEAPEST-INSERTION	3347.3225	14.763211 (45.35%)	81.837746 (34.70%)	96.60096
2-OPT STRIP	3349.41	0.200846 (47.61%)	94.274208 (35.14%)	94.47505
2-OPT	3364.3945		152.761860 (34.78%)	152.761860 (34.78%)
RANDOM-INSERTION	3383.9958	12.842165 (47.46%)		12.842165 (47.46%)
2-OPT NEAREST-INSERTION	3428.0579	15.382431 (45.08%)	79.219681 (34.87%)	94.60211
CHEAPEST-INSERTION	3731.0981	14.763211 (45.35%)		14.763211 (45.35%)
NEAREST-INSERTION	3886.2168	15.382431 (45.08%)		15.382431 (45.08%)
NEAREST-NEIGHBOR	3931.2803	2.394484 (36.33%)		2.394484 (36.33%)
NEAREST-ADDITION	4312.006	6.700523 (33.82%)		6.700523 (33.82%)
STRIP	4657.516	0.200846 (47.61%)		0.200846 (47.61%)
FARTHEST-ADDITION	6138.9346	6.692418 (33.40%)		6.692418 (33.40%)
1-P-P NEAREST-NEIGHBOR	3173.7473	2.394484 (36.33%)	43.773930 (35.08%)	46.168415
M-P-P NEAREST-NEIGHBOR	3174.9563	2.394484 (36.33%)	47.618198 (35.00%)	50.012684
1-P-P 15% NEAREST-NEIGHBOR	3330.959	2.394484 (36.33%)	4.509444 (35.50%)	6.9039283
1-P-P 18% NEAREST-NEIGHBOR	3220.9531	2.394484 (36.33%)	9.030979 (35.29%)	11.425463
1-P-P 19% NEAREST-NEIGHBOR	3179.172	2.394484 (36.33%)	16.936022 (34.87%)	19.330505
M-P-P 18% NEAREST-NEIGHBOR	3214.6233	2.394484 (36.33%)	5.941272 (34.23%)	8.335756
M-P-P 19% NEAREST-NEIGHBOR	3181.9436	2.394484 (36.33%)	18.201437 (34.41%)	20.59592
1-P-P STRIP	3365.4734	0.200846 (47.16%)	65.557373 (35.25%)	65.75822
M-P-P STRIP	3349.41	0.200846 (47.16%)	56.640156 (34.94%)	56.841003
M-P-P 25% STRIP	3474.752	0.200846 (47.16%)	13.774291 (33.80%)	13.975137
M-P-P 27% STRIP	3390.747	0.200846 (47.16%)	18.017279 (34.27%)	18.218124
1-P-P 27% STRIP	3391.9812	0.200846 (47.16%)	31.094189 (35.12%)	31.295034
M-P-P NEAREST-ADDITION	3282.926	6.700523 (33.82%)	40.594517 (34.54%)	47.29504
1-P-P NEAREST-ADDITION	3317.6902	6.700523 (33.82%)	49.852062 (32.92%)	56.552586
M-P-P 20% NEAREST-ADDITION	3448.6506	6.700523 (33.82%)	11.005413 (33.67%)	17.705936
M-P-P 22% NEAREST-ADDITION	3357.1597	6.700523 (33.82%)	21.463181 (34.45%)	28.163704

NO. OF CITIES: 2048

Tour Type	Tour Length	Initial Tour Time	Additional Time	Total Time (sec.)
		Total (CM%)	Total (CM%)	Total (CM%)
RANDOM	2158810.3			
FARTHEST-INSERTION	74873.89	123.861069 (45.11%)		123.861069 (45.11%)
NEAREST-NEIGHBOR	84210.66	19.261005 (36.07%)		19.261005 (36.07%)
M-P-P 10% NEAREST-NEIGHBOR	75782.14	19.261005 (36.07%)	10.292409 (33.18%)	29.553413
1-P-P 10% NEAREST-NEIGHBOR	75757.61	19.261005 (36.07%)	39.122208 (37.59%)	58.383213
M-P-P 12% NEAREST-NEIGHBOR		19.261005 (36.07%)	24.036905 (33.75%)	43.297913
M-P-P 13% NEAREST-NEIGHBOR	73232.75	19.261005 (36.07%)	51.730186 (34.11%)	70.991196
STRIP	106459.57	0.231038 (52.87%)		0.231038 (52.87%)
M-P-P 25% STRIP	79819.62	0.231038 (52.87%)	296.028990 (34.66%)	296.26004
M-P-P 20% STRIP	85159.73	0.231038 (52.87%)	61.908379 (34.02%)	62.139416

NO. OF CITIES: 16K

Tour Type	Tour Length	Initial Tour Time	Additional Time	Total Time (sec.)
		Total (CM%)	Total (CM%)	Total (CM%)
RANDOM	1.3980667e8			
STRIP	2368123.0	0.283137 (54.80%)		
STRIP (EQUIPOPULATED STRIPS)	2369464.0	0.235359 (60.47%)		

NO. OF CITIES: 64K

Tour Type	Tour Length	Initial Tour Time	Additional Time	Total Time (sec.)
		Total (CM%)	Total (CM%)	Total (CM%)
RANDOM	2.2420385e9			
STRIP	1.8902352e7	0.999		

NO. OF CITIES: 64K

Tour Type	Tour Length	Initial Tour Time	Additional Time	Total Time (sec.)
		Total (CM%)	Total (CM%)	Total (CM%)
RANDOM	2.237172e9			
STRIP	1.8913808e7	0.992828 (89.42%)		

```
;;;; -*- Package: *lisp ; Syntax: Common-Lisp; Mode: LISP -*-

;;;; Lisp Machine Conveniences

;; These don't seem to stop package from getting changed when debugger entered
;(zl:setq-standard-value zl:package (find-package '*lisp))
;(zl:setq-standard-value si:package (find-package '*lisp))

(zl:setq-standard-value si:*prinarray* t)

;;;; Potentially required *lisp junk

(proclaim '(*defun dist-sq distance tour-length t-l-rand t-l-tour show-rand show-tour
                   make-cities near-neighbor-tour insert-pos-cost!! find-best-position
                   find-insert-city find-insert-pos insert insert-city-cost!!
                   init-tour-perigee-and-dist update-tour-perigee-and-dist
                   insertion-tour sort-tour-by-x sort-tour-by-y strip-tour-num 2-opt
                   one-pass-par-2-opt tweak par-tweak 1-p-p n-n-t f-i-t n-i-t r-i-t
                   c-i-t n-a-t f-a-t s-t 2-opt-times par-2-opt mult-pass-par-2-opt))

;;;; Generally Useful *Lisp stuff

;; Can be changed to achieve different compilation for the simulator and the actual CM.
(defconstant simulator-flag t)

;; Makes usages of cm:time refer to regular time function when using simulator.
(unless (find-package 'cm) (export 'time (make-package 'cm)))

;; Simulator doesn't recognize :no-collisions in *pset, so will give :default then.
(defmacro *pset-no-coll (&rest args)
  '(*pset ,(if simulator-flag ':default ':no-collisions) . ,args))

;;; Debugging Aids.  Prints values of a variable, a pvar, or pvar restricted to
;;; active processors, along with a label.

(defmacro show-var (var) '(format *standard-output* "~%~s: ~s" ',var ,var))

(defmacro show-pvar (pvar)
  '(progn (format *standard-output* "~%~s:" ',pvar) (ppp ,pvar)))

(defmacro show-pvar-css (pvar)
  '(progn (format *standard-output* "~%~s:" ',pvar) (ppp-css ,pvar)))

;;; Macro to find first processor realizing min or max of a pvar (and optionally the value)
;;; The processor where the extreme is realized is returned by the function.  The value
;;; of the extreme is placed in the variable named by the optional argument if it is present.

(defmacro find-extreme-proc (*min-or-*max pvar-name &optional (value-var-name nil))
  (if value-var-name
      '(*when (=!! (!! (setq ,value-var-name (,*min-or-*max ,pvar-name))) ,pvar-name)
         (*min (self-address!!)))
      '(*when (=!! (!! (,*min-or-*max ,pvar-name)) ,pvar-name)
         (*min (self-address!!)))))

;;; Subroutines to compute square of distance and distance between two points in the plane

(*defun dist-sq!! (x1!! y1!! x2!! y2!!)
  (*let ((diff-x!! (-!! x1!! x2!!))
         (diff-y!! (-!! y1!! y2!!)))
    (declare (type (signed-pvar 17) diff-x!! diff-y!!))
    (+!! (*!! diff-x!! diff-x!!) (*!! diff-y!! diff-y!!))))

(*defun distance!! (x1!! y1!! x2!! y2!!)
  (*let ((diff-x!! (-!! x1!! x2!!))
         (diff-y!! (-!! y1!! y2!!)))
    (declare (type (signed-pvar 17) diff-x!! diff-y!!))
    (sqrt!! (+!! (*!! diff-x!! diff-x!!) (*!! diff-y!! diff-y!!)))))
```

```
;;;; Global Variables Generally Useful for Traveling Salesman

(proclaim '(type (signed-pvar 19) world-size!! rand-x!! rand-y!!
                 next-x!! next-y!! perigee!!))

(proclaim '(type float-pvar dist-to-next!! dist-to-tour!!))

(defvar n)                         ;number of cities
(*defvar world-size!!)             ;upper bound on x and y coordinates
(*defvar rand-x!!)                 ;random x coordinates
(*defvar rand-y!!)                 ;random y coordinates
(*defvar tour-x!!)                 ;x coordinates of tour
(*defvar tour-y!!)                 ;y coordinates of tour
(*defvar save-tour-x!!)            ;used to save a tour
(*defvar save-tour-y!!)            ; for later use

;;; Global variables used in constructing insertion tours

(*defvar pos-in-tour!!)            ;position in tour of this city (processor)
(*defvar next-x!!)                 ;x coordinate of next city in the tour
(*defvar next-y!!)                 ;y coordinate of next city in the tour
(*defvar dist-to-next!!)           ;distance to next city in the tour
(*defvar dist-to-tour!!)           ;distance (maybe not literally) to tour
                                   ; a metric for choice of insert-city
(*defvar perigee!!)                ;nearest point of tour to this city
(*defvar insert-city-x!!)          ;x coordinate of city to be inserted
(*defvar insert-city-y!!)          ;y coordinate of city to be inserted
(defvar insert-city)               ;city to be inserted into tour (processor number)
(defvar precede-insert-city)       ;city in tour to precede inserted city (processor number)

;;;; Functions and Macros of General Usefulness for Traveling Salesman

;;; Function to generate an example with num-cities cities.
;;; Generates random x and y coordinates having values in [0,num-cities), and
;;; Takes initial ordering of cities in processors as initial tour.

(*defun make-cities (num-cities) ;May want to sort to get rid of duplicates, but we didn't
  (setq n num-cities)
  (*all
    (*when (<!! (self-address!!) (!! num-cities))
      (*set world-size!! (!! num-cities))
      (*set rand-x!! (random!! world-size!!))
      (*set rand-y!! (random!! world-size!!))
      (*set tour-x!! rand-x!!)
      (*set tour-y!! rand-y!!))))

;;; Functions print random coordinates and tours in reasonably readable fashion.

(*defun show-rand (&aux (x (make-array n)) (y (make-array n)))
  (pvar-to-array rand-x!! x :cube-address-end n)
  (pvar-to-array rand-y!! y :cube-address-end n)
  (pprint (map 'array #'cons x y)))

(*defun show-tour (&aux (x (make-array n)) (y (make-array n)))
  (pvar-to-array tour-x!! x :cube-address-end n)
  (pvar-to-array tour-y!! y :cube-address-end n)
  (pprint (map 'array #'cons x y)))

;;; Functions to save and restore tours.

(*defun save-tour ()
  (*all
    (*set save-tour-x!! tour-x!!)
    (*set save-tour-y!! tour-y!!)))

(*defun restore-tour ()
  (*all  (*set tour-x!! save-tour-x!!)
         (*set tour-y!! save-tour-y!!)))
```

```
;;; Functions to compute the length of a tour in processor order.

(*defun tour-length (x-coords!! y-coords!!)
  (*all
    (*when (<!! (self-address!!) (!! n))
      (*let ((prev-neighbor!! (mod!! (1-!! (self-address!!)) (!! n)))
             (next-x!!)
             (next-y!!)
             (edge-to-next!!))
        (*pset-no-coll x-coords!! next-x!! prev-neighbor!!)
        (*pset-no-coll y-coords!! next-y!! prev-neighbor!!)
        (*set edge-to-next!! (distance!! x-coords!! y-coords!! next-x!! next-y!!))
        (*sum edge-to-next!!)))))

(*defun t-l-rand () (tour-length rand-x!! rand-y!!))

(*defun t-l-tour () (tour-length tour-x!! tour-y!!))

;;;; Initial Tour Construction -- Approach A: Nearest Neighbor

;;; Function to build a nearest-neighbor initial tour

(*defun near-neighbor-tour (&aux nearest)
  (*all
    (*let ((not-in-tour!! t!!)  ;; t for processors not yet put in improved initial tour
           (dist-sq!!))         ;; square of distance to last city on nearest neighbor tour
      (declare (type float-pvar dist-sq!!))
      (declare (type boolean-pvar not-in-tour!!))
      (setf (pref tour-x!! 0) (pref rand-x!! 0))       ;First city in tour will be
      (setf (pref tour-y!! 0) (pref rand-y!! 0))       ; the first random one.
      (setf (pref not-in-tour!! 0) nil)               ;Check off that city.
      (*when (<!! (self-address!!) (!! n))
        (dotimes (i (1- n))
          (*when not-in-tour!!
            (*set dist-sq!!   ;; Compute distance of each active city to last city on nn-tour.
                  (dist-sq!! rand-x!! rand-y!! (!! (pref tour-x!! i)) (!! (pref tour-y!! i))))
            (setq nearest (find-extreme-proc *min dist-sq!!)))  ;Find proc realizing min dist.
          (setf (pref tour-x!! (1+ i)) (pref rand-x!! nearest)) ;Make him next in the tour.
          (setf (pref tour-y!! (1+ i)) (pref rand-y!! nearest))
          (setf (pref not-in-tour!! nearest) nil)))))) ;Check him off.
```

```
;;;; Initial Tour Construction -- Approach B: Insertion and Addition Methods

;;; Often all of these methods will be referred to as insertion methods
;;; for convenience.  There is one general function which performs any
;;; of a number of such methods and which appears at the bottom of the
;;; next page.  Preceding it here are various subroutines which it uses.
;;; Finds cost of each choice of insert city for the city to precede given
;;; by argument.

(*defun insert-city-cost!! (pos)
  (-!! (+!!
        (distance!! rand-x!! rand-y!! (!! (pref rand-x!! pos)) (!! (pref rand-y!! pos)))
        (distance!! rand-x!! rand-y!! (!! (pref next-x!! pos)) (!! (pref next-y!! pos))))
       (!! (pref dist-to-next!! pos))))

;;; Finds the distance of each city to the tour using appropriate metric for the
;;; appropriate mode (e.g. "distance" could be min. cost of inserting into tour).
;;; Then finds the point on the tour nearest to the city (refered to as "perigee").

(*defun find-tour-perigee-and-dist (pass mode)
  (case mode
    ((nearest farthest nearest-addition farthest-addition) ;; distance is actual dist. to tour
     (if (zerop pass)                          ;initialize
         (*set dist-to-tour!! (dist-sq!! rand-x!! rand-y!! (!! (pref rand-x!! 0))
                                          (!! (pref rand-y!! 0))))
         (*when (not!! pos-in-tour!!)          ;; update by checking dist. to last city inserted
           (*let ((dist-to-new!! (dist-sq!! rand-x!! rand-y!!
                                             (!! (pref rand-x!! insert-city))
                                             (!! (pref rand-y!! insert-city)))))
             (declare (type float-pvar dist-to-new!!))
             (*when (>!! dist-to-tour!! dist-to-new!!)
               (*set dist-to-tour!! dist-to-new!!)
               (*set perigee!! (!! insert-city)))))))
    (cheapest                                  ;; distance is cost of inserting city into tour
      (if (zerop pass)                         ;initialize
          (*set dist-to-tour!! (insert-city-cost!! 0))
          (*when (not!! pos-in-tour!!) ;; update: check insertion before/after last inserted
            (*let ((dist-to-new!! (insert-city-cost!! insert-city))
                   (dist-to-precede-new!! (insert-city-cost!! precede-insert-city)))
              (declare (type float-pvar dist-to-new!! dist-to-precede-new))
              (*when (>!! dist-to-tour!! dist-to-precede-new!!)
                (*set dist-to-tour!! dist-to-precede-new!!)
                (*set perigee!! (!! precede-insert-city)))
              (*when (>!! dist-to-tour!! dist-to-new!!)
                (*set dist-to-tour!! dist-to-new!!)
                (*set perigee!! (!! insert-city)))))))
    (random nil)
    (otherwise
      (format *error-output*
              "find-tour-perigee-and-dist:  mode not implemented-- ~s" mode)))
  (if (zerop pass) (*set perigee!! (!! 0))))

;;; Finds cost of each choice of insert position for insert-city

(*defun insert-pos-cost!! ()
  (-!! (+!!
        (distance!! rand-x!! rand-y!! insert-city-x!! insert-city-y!!)
        (distance!! next-x!! next-y!! insert-city-x!! insert-city-y!!))
       dist-to-next!!))

;;; Finds the best place to insert insert-city.

(*defun find-best-position ()
  (*when pos-in-tour!! (find-extreme-proc *min  (insert-pos-cost!!))))
```

```
;;; The following two subroutines find the city to insert next and the city
;;; in the tour which should precede it according to the desired mode of
;;; insertion.  Results go into insert-city and precede-insert-city.

(*defun find-insert-city (tour-size mode)
  (case mode
    ((farthest farthest-addition)              ;insert city farthest from tour
     (setq insert-city
           (*when (not!! pos-in-tour!!) (find-extreme-proc *max dist-to-tour!!))))
    ((nearest nearest-addition cheapest)       ;insert city nearest to tour
     (setq insert-city
           (*when (not!! pos-in-tour!!) (find-extreme-proc *min dist-to-tour!!))))
    (random
      (setq insert-city tour-size))            ;insert next random city
    (otherwise (format *error-output*
                       "find-insert-city:  mode not implemented-- ~s" mode))))

(*defun find-insert-pos (mode)
  (case mode
    ((farthest nearest random)
     (setq precede-insert-city (find-best-position))) ;; insert into least costly position
    ((nearest-addition farthest-addition cheapest)
     (setq precede-insert-city (pref perigee!! insert-city))) ;; insert after nearest tour pos
    (otherwise (format *error-output*
                       "find-insert-pos:  mode not implemented-- ~s" mode))))

;;; Inserts city after precede-by in the partial tour.

(*defun insert (city precede-by)
  (let ((position (1+ (pref pos-in-tour!! precede-by)))) ;; pos. to be taken by insert-city
    (*when pos-in-tour!! (*when (>=!! pos-in-tour!! (!! position))     ;; push up all cities
                         (*set pos-in-tour!! (1+!! pos-in-tour!!)))) ;; from that pos. on
    (setf (pref pos-in-tour!! city) position)
    (setf (pref next-x!! city)        (pref next-x!! precede-by)
          (pref next-y!! city)        (pref next-y!! precede-by)
          (pref next-x!! precede-by)  (pref rand-x!! city)
          (pref next-y!! precede-by)  (pref rand-y!! city))
    (*set dist-to-next!! (distance!! rand-x!! rand-y!! next-x!! next-y!!))))

;;; Function to build an initial tour by an insertion (or addition) method.
;;; Possibilities are farthest-insertion, nearest-insertion, random-insertion,
;;; cheapest-insertion, nearest-addition, and farthest-addition.
;;; Uses the standard index-sort hack of letting everyone keep track of
;;; his position in the tour and only moving things around at the end.

(*defun insertion-tour (mode)
  (*all
    (*when (<!! (self-address!!) (!! n))
      (*set pos-in-tour!! nil!!
            next-x!! rand-x!!
            next-y!! rand-y!!
            dist-to-next!! (!! 0))
      (setf (pref pos-in-tour!! 0) 0)           ;Initial tour is self-loop on
                                                ; first random city.
      (dotimes (i (1- n))
        (find-tour-perigee-and-dist i mode)     ;Find dist. to tour, perigee.
        (find-insert-city (1+ i) mode)          ;Sets insert-city.
        (*set insert-city-x!! (!! (pref rand-x!! insert-city))) ;Inform everybody of
        (*set insert-city-y!! (!! (pref rand-y!! insert-city))) ; insert-city coords.
        (find-insert-pos mode)                  ;Sets precede-insert-city.
        (insert insert-city precede-insert-city))     ;Do the insertion.
      (*pset-no-coll rand-x!! tour-x!! pos-in-tour!!)  ;Route everyone to positions
      (*pset-no-coll rand-y!! tour-y!! pos-in-tour!!))))      ; determined for tour.
```

```
;;;; Initial Tour Construction -- Approach C: Strips Tours

;;;    Assume that we receive tour in form rand-x!! rand-y!!

;This is a function to sort the random tour by x-coordinates and arrange the y's correctly
;We add 1/(self-address 2) to use as a pointer to the spot from which it came

(*defun sort-tour-by-x ( &aux num-of-bits)
  (*all
    ;Have to add self-address to them so we can pull the y's along.
    (*when (<!! (self-address!!) (!! n))
    (*let
      ((new-tour-x!!)
       (new-tour-y!! tour-y!!)
       (y-pointers!!)
       (y-pointers-aux!!))

       (setq num-of-bits (ceiling (log (*max tour-x!!) 2)))

      (*set new-tour-x!!
       (deposit-byte!!
         (*!! (!! (expt 2 num-of-bits)) tour-x!!)
         (!! 0 )
         (!! num-of-bits)
         (self-address!!) ))

       (*set tour-x!!  (sort!! new-tour-x!! '<=!!))

       ;Recover pointers from low order bits that were appended

       (*set y-pointers!! (load-byte!! tour-x!! (!! 0) (!! num-of-bits)))

       (*set y-pointers-aux!! y-pointers!!)
       (*set tour-x!! (load-byte!! tour-x!! (!! num-of-bits) (!! num-of-bits)))

       (*pset-no-coll (self-address!!) y-pointers!! y-pointers-aux!!)
       (*pset-no-coll new-tour-y!! tour-y!! y-pointers!!)))))

(*defun sort-tour-by-y ()
  (*all
  (*when (<!! (self-address!!) (!! n))
  (*let
    ( (buffer!!))
    (*set buffer!! tour-x!!)
    (*set tour-x!! tour-y!!)
    (*set tour-y!! buffer!!)
    (sort-tour-by-x)
    (*set buffer!! tour-x!!)
    (*set tour-x!! tour-y!!)
    (*set tour-y!! buffer!!)))))
  JOEL:  CAN YOU TIDY THIS UP AND CUT OUT ENOUGH WHITE SPACE TO MAKE HERE TO NEXT PAGE
BOUNDARY ACTUALLY FIT ON ONE PAGE?
; Function for computing the tour by making strips and then going long strips
(*defun strip-tour (switch &aux sqrtn)
  (*all
    (*when (<!! (self-address!!) (!! n))
    (*set tour-x!! rand-x!!)
    (*set tour-y!! rand-y!!)
    (sort-tour-by-x)

    (setq sqrtn (floor (sqrt n)))

    ;MARK WHICH BLOCK EVERYONE IS IN
    ; if switch is 1 go by num of cities, 2 by length
    ;We want to add to each block

    (*let ( (sqrt-mark!!)
            (seg-length!!)
            (min-addr!!)
            (max-addr!!)
            (seg!!)
            (back-seg!!))

            (if (eql switch 1)
            (*set sqrt-mark!!    (truncate!! (/!! (self-address!!) (!! sqrtn))))
            (*set sqrt-mark!!    (truncate!! (/!!  tour-x!!        (!! sqrtn)))))


      (*set tour-y!! (+!! tour-y!!
                          (*!! (!! n)
                               sqrt-mark!!)))
      (sort-tour-by-y)
      (*set tour-y!! (mod!! tour-y!! (!! n)))

;THIS IS EXPERIMENTAL
      ;Now tour is exactly as we want it except we want to flip every other segment
      ;That corresponds to going up and down the columns
      ;THIS IS OK IF switch = 1
      ; Have to do segmneted max and min scans subtract and use that fdor sqrtn here

      ;FIND THE SEGMENT PVARto use here -
      (when (eql switch 2)
       (*when (>!! (self-address!!) (!! 0))
        (*if (/=!! (pref!! sqrt-mark!! (self-address!!))
                   (pref!! sqrt-mark!! (-!! (self-address!!) (!! 1))))
             (*pset-no-coll
               (!! t)
               seg!!
               (self-address!!))
             (*pset-no-coll
               (!! nil)
               seg!!
               (self-address!!))))

        (setf (pref seg!! 0) t)
        ;MAKE BACWARDS SEeGMENT

        (*when (>!! (self-address!!) (!! 0))
          (*pset-no-coll seg!! back-seg!! (-!! (self-address!!) (!! 1))))
        (setf (pref back-seg!! (- n 1)) t)


      ;Find min length -- ie endpoints of each segment
      (*set min-addr!! (scan!! (self-address!!) 'min!!
                               :segment-pvar seg!!))
      ; Find maximum address
      (*set max-addr!! (scan!! (self-address!!) 'max!!
                               :segment-pvar back-seg!!
                               :direction :backward))

      ;First build the lengths of each segment


      (*set seg-length!! (+!!
                          (!! 1)
                          (-!!
                            max-addr!!
```

```
                                          min-addr!!)))


      (*if (=!! (!! 1) (mod!! sqrt-mark!! (!! 2)))
           (*pset-no-coll
             tour-y!!
             tour-y!!
             (+!! (self-address!!)
                  seg-length!!
                  (-!! (1+!! (*!! (!! 2)
                                  (-!!
                                    (self-address!!)
                                    min-addr!!))))))))
;THIS IS WHAT WORKED BEFORE
(when (eql switch 1)
      (*if (=!! (!! 1) (mod!! sqrt-mark!! (!! 2)))
           (*pset-no-coll
             tour-y!!
             tour-y!!
             (+!! (self-address!!)
                  (!! sqrtn)
                  (-!! (1+!! (*!! (!! 2)
                                  (mod!! (self-address!!) (!! sqrtn))))))))))))
(*defun test-1-2 (&aux length)
  (print n)
  (print "Random Toour length")
  (setq length (tour-length rand-x!! rand-y!!))
  (print length)
  (strip-tour 1)
  (print "options length and distance")
  (setq length (tour-length tour-x!! tour-y!!))
  (print length)
  (strip-tour 2)
  (setq length (tour-length tour-x!! tour-y!!))
  (print length)
  (print "near-nbr-tour")
  (n-n-t)
  (setq length (tour-length tour-x!! tour-y!!))
  (print length))
```

```
;;;; Heuristics for tour improvement based on
;;;; 2-optimality/quadrangle-defun

;This procedure tweaks the tour in tour-x!! and tour-y!! until it is 2-optimal or, if the
;optional parameter is used, until the desired percentage gain is achieved.

(*defun 2-opt (&optional (percent-goal nil)
                         &aux max-gain max-tail-addr max-gain-addr tour-goal)
  (*all
    (*when (<!! (self-address!!) (!! n))
      (*let ((active!! t!!)  ;; edges that are still eligible for tweaking
             (max-head-x!!)  ;; x-coordinate of the head of the longest active edge
             (max-head-y!!)  ;; y-coordinate of the head of the longest active edge
             (max-tail-x!!)  ;; x-coordinate of the tail of the longest active edge
             (max-tail-y!!)  ;; y-coordinate of the tail of the longest active edge
             (max-length!!)  ;; length of the longest active edge
             (edge-to-next!!) ;Distance of edge to next city
             (gain!!) ;reduction in tour length if the edge starting at this city is
                      ;tweaked with the maximum-length edge
             (prev-neighbor!! (mod!! (1-!! (self-address!!)) (!! n)))
             (next-x!!) ;; x-coordinate of the city after this one in the tour
             (next-y!!)) ;; y-coordinate    "        "       "
        (if percent-goal (setq tour-goal (* (- 1 percent-goal) (t-l-tour))))
;; repeat until goal met or no good tweaks (ie. only one edge left active)
        (do () ((or (= (*sum (if!! active!! (!! 1) (!! 0))) 1)
                       (and percent-goal (< (t-l-tour) tour-goal))) nil)
          (*set active!! t!!)
          (setf max-gain -1)
          (*pset-no-coll tour-x!! next-x!! prev-neighbor!!)  ;get coords of next city
          (*pset-no-coll tour-y!! next-y!! prev-neighbor!!)
          (*set edge-to-next!! (distance!! tour-x!! tour-y!! next-x!! next-y!!))
          (do ()    ;until we find a good tweak or no good tweaks left in tour
              ((or (> max-gain 0)   (= (*sum (if!! active!! (!! 1) (!! 0))) 1) ) nil)
           (*when active!!
             (*set max-length!! (!! (*max edge-to-next!!)))  ;find max length edge
             (*when (=!! max-length!! edge-to-next!!)  ;; break ties to find processor
               (setf max-tail-addr (*min (self-address!!)))) ;; with max length edge
             (*set max-head-x!! (!! (pref tour-x!! (mod (1+ max-tail-addr) n))))
             (*set max-head-y!! (!! (pref tour-y!! (mod (1+ max-tail-addr) n))))
             (*set max-tail-y!! (!! (pref tour-y!! max-tail-addr)))
             (*set max-tail-x!! (!! (pref tour-x!! max-tail-addr)))
             (*set gain!!
                   (-!! (+!! max-length!! edge-to-next!!)
                        (+!! (distance!! max-head-x!! max-head-y!! next-x!! next-y!!)
                             (distance!! max-tail-x!! max-tail-y!! tour-x!! tour-y!!))))
             (*when (not!! (=!! (self-address!!) (!! max-tail-addr)))
               (setf max-gain (*max gain!!)))  ;find best edge to tweak with max edge
             (*when (=!! (!! max-gain) gain!!)
               (setf max-gain-addr (*min (self-address!!))))
             (if (<= max-gain 0) (setf (pref active!! max-tail-addr) nil))))
          (if (> max-gain 0) (tweak max-tail-addr max-gain-addr)))))))
```

```
;This procedure is identical to the above procedure except that it performs at most
;n/4 tweaks on the tour (or until a termination criterion is met).  It procedure returns
;t if it performed all n/4 tweaks and it returns nil if the procedure terminated early.

(*defun 2-opt-times (&optional (percent-goal nil)
                     &aux max-gain max-tail-addr max-gain-addr tour-goal)
  (*all
    (*when (<!! (self-address!!) (!! n))
      (*let ((active!! t!!)
             (max-head-x!!)
             (max-head-y!!)
             (max-tail-x!!)
             (max-tail-y!!)
             (max-length!!)
             (edge-to-next!!)                        ;Distance of edge to next node
             (gain!!)                                ;
             (prev-neighbor!! (mod!! (1-!! (self-address!!)) (!! n)))
             (next-x!!)
             (next-y!!))
        (if percent-goal (setq tour-goal (* (- 1 percent-goal) (t-l-tour))))
        (dotimes (j (/ n 4) nil)
          (if (or (= (*sum (if!! active!! (!! 1) (!! 0))) 1)
                  (and percent-goal (< (t-l-tour) tour-goal))) (return t))
          (*set active!! t!!)
          (setf max-gain -1)
          (*pset-no-coll tour-x!! next-x!! prev-neighbor!!)
          (*pset-no-coll tour-y!! next-y!! prev-neighbor!!)
          (*set edge-to-next!! (distance!! tour-x!! tour-y!! next-x!! next-y!!))
          (do ()
              ((or (> max-gain 0)   (= (*sum (if!! active!! (!! 1) (!! 0))) 1) ) nil);;3 helps
            (*when active!!
              (*set max-length!! (!! (*max edge-to-next!!)))
              (*when (=!! max-length!! edge-to-next!!)
                (setf max-tail-addr (*min (self-address!!))))
              (*set max-head-x!! (!! (pref tour-x!! (mod (1+ max-tail-addr) n))))
              (*set max-head-y!! (!! (pref tour-y!! (mod (1+ max-tail-addr) n))))
              (*set max-tail-y!! (!! (pref tour-y!! max-tail-addr)))
              (*set max-tail-x!! (!! (pref tour-x!! max-tail-addr)))
              (*set gain!!
                    (-!! (+!! max-length!! edge-to-next!!)
                         (+!! (distance!! max-head-x!! max-head-y!! next-x!! next-y!!)
                              (distance!! max-tail-x!! max-tail-y!! tour-x!! tour-y!!))))
              (*when (not!! (=!! (self-address!!) (!! max-tail-addr)))
                (setf max-gain (*max gain!!)))
              (*when (=!! (!! max-gain) gain!!)
                (setf max-gain-addr (*min (self-address!!))))
              (if (<= max-gain 0) (setf (pref active!! max-tail-addr) nil))))
          (if (> max-gain 0) (tweak max-tail-addr max-gain-addr)))))))

;;This procedure accepts the processor addresses of the tails of two edges to tweak.  It
;;then performs the tweak by reversing the tour between the head of the edge with lower
;;processor address and the tail of the edge with higher processor address inclusive

(*defun tweak (tail1-addr tail2-addr)
  (*let ((min-tail!! (!! (min tail1-addr tail2-addr))) ;; tail of edge with higher proc addr.
         (max-tail!! (!! (max tail1-addr tail2-addr))) ;; tail of edge with lower proc addr.
         (where-to!!))
    (*when (and!! (>!! (self-address!!) min-tail!!) (<=!! (self-address!!) max-tail!!))
      (*set where-to!! (-!! (+!! min-tail!! (!! 1) max-tail!!) (self-address!!)))
      (*pset-no-coll tour-x!! tour-x!! where-to!!)
      (*pset-no-coll tour-y!! tour-y!! where-to!!))))
```

```
;This procedure performs one-pass parallel 2-optimization.  It divides the cities into
;disjoint groups of size 4 and performs a parallelized version of 2-opt on each city
;using scans instead of *max and broadcast.  It tweaks until each group is 2-optimal
;(or a goal percentage achieved), then doubles the size of the group and continues until
;there are fewer than 32 groups.  It then finishes up with a final call to the global
;2-optimization procedure

(*defun one-pass-par-2-opt (&optional (goal-percent nil)
                            &aux group-size num-groups max-gain tour-goal)

  (*all
    (*when (<!! (self-address!!) (!! n))
      (*let ((active!! t!!) ;; variables with same names have same function as in 2-opt
             (max-head-x!!)
             (max-head-y!!)
             (max-tail-x!!)
             (max-tail-y!!)
             (max-length!!)
             (edge-to-next!!)
             (gain!!)
             (max-gain!!)
             (prev-neighbor!! (mod!! (1-!! (self-address!!)) (!! n)))
             (next-x!!)
             (next-y!!)
             (max-tail-addr!!)
             (max-gain-addr!!)
             (city-groups!!)  ;; segment pvar
             (rev-city-groups!!)) ;; segment pvar for reverse scanning
         (if goal-percent (setq tour-goal (* (- 1 goal-percent) (t-l-tour))))
         (setq group-size 4)
         (setq num-groups (/ n group-size))
         (do () ((or (< num-groups 32)
                     (and goal-percent (< (t-l-tour) tour-goal))) nil)
           (*set city-groups!! (=!! (mod!! (self-address!!) (!! group-size)) (!! 0)))
           (*if (plusp!! (self-address!!))
               (*pset-no-coll city-groups!! rev-city-groups!! (1-!! (self-address!!))))
           (setf (pref rev-city-groups!! (1- n)) t)
           (*set active!! t!!)
           (do () ((or (not (*or active!!))  ;; do until no good tweaks or goal reached
                       (and goal-percent (< (t-l-tour) tour-goal))) nil)
             (setf max-gain -1)
             (*pset-no-coll tour-x!! next-x!! prev-neighbor!!)
             (*pset-no-coll tour-y!! next-y!! prev-neighbor!!)
             (*set edge-to-next!! (distance!! tour-x!! tour-y!! next-x!! next-y!!))
             (do () ;until find good tweak or no good tweaks exist, search for tweak
                 ((or (> max-gain 0) (not (*or active!!))) nil)
  ;;next two lines find max length of an edge
                  (*set max-length!! (scan!! (if!! active!! edge-to-next!! (!! -1)) 'max!!
                                             :direction :backward
                                             :segment-pvar rev-city-groups!!))
                  (*set max-length!! (scan!! max-length!! 'copy!! :segment-pvar city-groups!!))
  ;; find processor number of tail of max-length edge (breaking ties with max)
                  (*set max-tail-addr!! (scan!! (if!! (and!! active!!
                                                             (=!! max-length!! edge-to-next!!))
                                                       (self-address!!) (!! -1)) 'max!!
                                                :direction :backward
                                                :segment-pvar rev-city-groups!!))
                  (*set max-tail-addr!! (scan!! max-tail-addr!! 'copy!!
                                                :segment-pvar city-groups!!))
                  (*set max-head-x!! (scan!! (if!! (=!! (self-address!!) max-tail-addr!!)
                                                   (pref!! tour-x!! (mod!!(1+!! max-tail-addr!!)
                                                                          (!! n)))
                                                   (!! -1)) 'max!! :direction :backward
                                                   :segment-pvar rev-city-groups!!))
   ;segmented version of pref'ing value and broadcasting
                   (*set max-head-x!! (scan!! max-head-x!! 'copy!! :segment-pvar city-groups!!))
                   (*set max-head-y!! (scan!! (if!! (=!! (self-address!!) max-tail-addr!!)
                                                    (pref!! tour-y!! (mod!!(1+!! max-tail-addr!!)
                                                                           (!! n)))
                                                    (!! -1)) 'max!! :direction :backward
                                                    :segment-pvar rev-city-groups!!))
                   (*set max-head-y!! (scan!! max-head-y!! 'copy!! :segment-pvar city-groups!!))
                   (*set max-tail-y!! (scan!! (if!! (=!! (self-address!!) max-tail-addr!!)
                                                    tour-y!! (!! -1)) 'max!! :direction :backward
                                                    :segment-pvar rev-city-groups!!))
                   (*set max-tail-y!! (scan!! max-tail-y!! 'copy!! :segment-pvar city-groups!!))
                  (*set max-tail-x!! (scan!! (if!! (=!! (self-address!!) max-tail-addr!!)
                                                   tour-x!! (!! -1)) 'max!! :direction :backward
                                                   :segment-pvar rev-city-groups!!))
                  (*set max-tail-x!! (scan!! max-tail-x!! 'copy!! :segment-pvar city-groups!!))
                  (*set gain!!    ;;; find reduction from tweak with max-length edge
                    (-!! (+!! max-length!! edge-to-next!!)
                         (+!! (distance!! max-head-x!! max-head-y!! next-x!! next-y!!)
                              (distance!! max-tail-x!! max-tail-y!! tour-x!! tour-y!!))))
                  (*set max-gain!! (scan!! (if!! (and!!
                                                   (not!! (=!! (self-address!!) max-tail-addr!!))
                                                   active!!) gain!! (!! -1)) 'max!!
                                           :direction :backward
                                           :segment-pvar rev-city-groups!!))
                  (*set max-gain!! (scan!! max-gain!! 'copy!! :segment-pvar city-groups!!))
                  (setf max-gain (*max max-gain!!))
                  (*if (and!! (<=!! max-gain!! (!! 0))(=!! (self-address!!) max-tail-addr!!))
                    (*set active!! nil!!))
                  (when (> max-gain 0)  ;; if we're going to tweak, find out where
                    (*set max-gain-addr!! (scan!! (if!! (=!! gain!! max-gain!!)
                                                        (self-address!!) (!! -1))
                                                 'max!! :direction :backward
                                                 :segment-pvar rev-city-groups!!))
                    (*set max-gain-addr!! (scan!! max-gain-addr!! 'copy!!
                                                 :segment-pvar city-groups!!))
;; only groups that perform a tweak have all cities reactivated, others have some inactive
                  (*if (>!! max-gain!! (!! 0)) (*set active!! t!!))
                  (par-tweak max-tail-addr!! max-gain-addr!! max-gain!!))))
          (setq group-size (* group-size 2))
          (setq num-groups (/ num-groups 2))))
;; account for progress made so far when calling 2-opt with a goal-percentage
      (if (not goal-percent) (2-opt)
          (if (> (t-l-tour) tour-goal) (2-opt (- 1 (/ tour-goal (t-l-tour)))))))))

;; simply a parallelized version of tweak.  Only tweak where gain was positive.  Now the
;; tail addresses are pvars instead of lisp variables

(*defun par-tweak (tail1-addr!! tail2-addr!! max-gain!!)
  (*when (>!! max-gain!! (!! 0))
    (*let ((min-tail!! (min!! tail1-addr!! tail2-addr!!))
           (max-tail!! (max!! tail1-addr!! tail2-addr!!))
           (where-to!!))
      (*when (and!! (>!! (self-address!!) min-tail!!) (<=!! (self-address!!) max-tail!!))
        (*set where-to!! (-!! (+!! min-tail!! (!! 1) max-tail!!) (self-address!!)))
        (*pset-no-coll tour-x!! tour-x!! where-to!!)
        (*pset-no-coll tour-y!! tour-y!! where-to!!)))))

;; if there are less than 32 groups, the slowdown due to segmentation overrides the benefits
;; of parallelization.

(*defun par-2-opt (&optional (goal-percent nil))
  (if (< n 128) (2-opt goal-percent)
    (mult-pass-par-2-opt goal-percent)))

;; Almost the same as one-pass parallel 2-optimization except now we go from global tweaks
;; down to progressively smaller groups rather than build up from small groups.  At each
;; level we go till 2-optimal or goal reached or group-size/4 iterations.  Through each
;; pass (top to small groups) the starting point for group division shifts (eg 0-centered
;; at the start, then 1-centered, and so on) so that different cities can be grouped together
;; if they didn't move during the last pass.  The basic loop is identical to the one-pass
;; version; only the control structure changed

(*defun mult-pass-par-2-opt (&optional (goal-percent nil)
                                      &aux group-size num-groups max-gain tour-goal)
  (*all
    (*when (<!! (self-address!!) (!! n))
      (*let ((active!! t!!)
             (max-head-x!!)
             (max-head-y!!)
             (max-tail-x!!)
             (max-tail-y!!)
             (max-length!!)
             (edge-to-next!!)
             (gain!!)
             (max-gain!!)
             (prev-neighbor!! (mod!! (1-!! (self-address!!)) (!! n)))
             (next-x!!)
             (next-y!!)
```

```
            (max-tail-addr!!)
            (max-gain-addr!!)
            (city-groups!!)
            (rev-city-groups!!))
        (if goal-percent (setq tour-goal (* (- 1 goal-percent) (t-l-tour))))
;; repeat passes until goal reached (if any) or the tour is 2-optimal, which means 2-opt
;; at the global level (signaled back from the 2-opt routine]
        (do ((num-iters 0 (1+ num-iters))) ((or (and goal-percent (< (t-l-tour) goal-percent))
                                (2-opt-times goal-percent)) nil)
          (setq group-size (/ n 32))
          (setq num-groups 32)
          (do () ((or (< group-size 4)
                      (and goal-percent (< (t-l-tour) tour-goal))) nil)
            (*set city-groups!! (=!! (mod!! (self-address!!) (!! group-size))
                                    (!! (mod num-iters group-size))))
            (*if (plusp!! (self-address!!))
                  (*pset-no-coll city-groups!! rev-city-groups!! (1-!! (self-address!!))))
            (setf (pref rev-city-groups!! (1- n)) t)
            (*set active!! t!!)
;; do until all groups 2-opt (no active edges) or goal reached or done enough par tweaks
            (dotimes (k (max (/ group-size 4) 4))
              (if (or (not (*or active!!))
                      (and goal-percent (< (t-l-tour) tour-goal))) (return t))
              (setf max-gain -1)
              (*pset-no-coll tour-x!! next-x!! prev-neighbor!!)
              (*pset-no-coll tour-y!! next-y!! prev-neighbor!!)
              (*set edge-to-next!! (distance!! tour-x!! tour-y!! next-x!! next-y!!))
              (do ()  ;; until find good tweak or no good tweak, search for tweak
                  ((or (> max-gain 0) (not (*or active!!))) nil)
                  (*set max-length!! (scan!! (if!! active!! edge-to-next!! (!! -1)) 'max!!
                                              :direction :backward
                                              :segment-pvar rev-city-groups!!))
                  (*set max-length!! (scan!! max-length!! 'copy!! :segment-pvar city-groups!!))
                  (*set max-tail-addr!! (scan!! (if!! (and!! active!!
                                                            (=!! max-length!! edge-to-next!!))
                                                       (self-address!!) (!! -1)) 'max!!
                                                :direction :backward
                                                :segment-pvar rev-city-groups!!))
                  (*set max-tail-addr!! (scan!! max-tail-addr!! 'copy!!
                                                :segment-pvar city-groups!!))
                  (*set max-head-x!! (scan!! (if!! (=!! (self-address!!) max-tail-addr!!)
                                                 (pref!! tour-x!! (mod!!(1+!! max-tail-addr!!)
                                                                        (!! n)))
                                                 (!! -1)) 'max!! :direction :backward
                                                 :segment-pvar rev-city-groups!!))
                  (*set max-head-x!! (scan!! max-head-x!! 'copy!! :segment-pvar city-groups!!))
                  (*set max-head-y!! (scan!! (if!! (=!! (self-address!!) max-tail-addr!!)
                                                   (pref!! tour-y!! (mod!!(1+!! max-tail-addr!!)
                                                                          (!! n)))
                                                   (!! -1)) 'max!! :direction :backward
                                                   :segment-pvar rev-city-groups!!))
                  (*set max-head-y!! (scan!! max-head-y!! 'copy!! :segment-pvar city-groups!!))
                  (*set max-tail-y!! (scan!! (if!! (=!! (self-address!!) max-tail-addr!!)
                                                   tour-y!! (!! -1)) 'max!! :direction :backward
                                                   :segment-pvar rev-city-groups!!))
                  (*set max-tail-y!! (scan!! max-tail-y!! 'ccpy!! :segment-pvar city-groups!!))
                  (*set max-tail-x!! (scan!! (if!! (=!! (self-address!!) max-tail-addr!!)
                                                   tour-x!! (!! -1)) 'max!! :direction :backward
                                                   :segment-pvar rev-city-groups!!))
                  (*set max-tail-x!! (scan!! max-tail-x!! 'copy!! :segment-pvar city-groups!!))
                  (*set gain!!
                    (-!! (+!! max-length!! edge-to-next!!)
                          (+!! (distance!! max-head-x!! max-head-y!! next-x!! next-y!!)
                               (distance!! max-tail-x!! max-tail-y!! tour-x!! tour-y!!))))
                  (*set max-gain!! (scan!! (if!! (and!!
                                                    (not!! (=!! (self-address!!) max-tail-addr!!))
                                                    active!!) gain!! (!! -1)) 'max!!
                                            :direction :backward
                                            :segment-pvar rev-city-groups!!))
                  (*set max-gain!! (scan!! max-gain!! 'copy!! :segment-pvar city-groups!!))
                  (setf max-gain (*max max-gain!!))
                  (*if (and!! (<=!! max-gain!! (!! 0))(=!! (self-address!!) max-tail-addr!!))
                       (*set active!! nil!!))
                  (when (> max-gain 0)
                    (*set max-gain-addr!! (scan!! (if!! (=!! gain!! max-gain!!)
                                                         (self-address!!) (!! -1))
                                                   'max!! :direction :backward
                                                 :segment-pvar rev-city-groups!!))
          (*set max-gain-addr!! (scan!! max-gain-addr!! 'copy!!
                                                 :segment-pvar city-groups!!))
          (*if (>!! max-gain!! (!! 0)) (*set active!! t!!))
          (par-tweak max-tail-addr!! max-gain-addr!! max-gain!!))))
    (setq group-size (/ group-size 2))
    (setq num-groups (* num-groups 2)))
  (if goal-percent  ;; adjust goal percentage gain to reflect progess
      (setq goal-percent (- 1 (/ tour-goal (t-l-tour))))))))))
```

```
;;; Functions with abbreviated names to perfrom heuristics

(*defun n-n-t () (near-neighbor-tour))

(*defun f-i-t () (insertion-tour 'farthest))

(*defun n-i-t () (insertion-tour 'nearest))

(*defun r-i-t () (insertion-tour 'random))

(*defun c-i-t () (insertion-tour 'cheapest))

(*defun n-a-t () (insertion-tour 'nearest-addition))

(*defun f-a-t () (insertion-tour 'farthest-addition))

(*defun s-t (type) (strip-tour type))

(*defun 1-p-p (&optional (goal-percent nil))
  (one-pass-par-2-opt goal-percent))

(*defun m-p-p (&optional (goal-percent nil))
  (mult-pass-par-2-opt goal-percent))
```

```
;;; Function to make a random problem, try and time many different
;;; heuristics applied to it, and rank results by tour length.

(*defun try-problem (num-cities &optional (time nil))
  (make-cities num-cities)
  (setq lengths nil)
  (format *standard-output* "~%      RANDOM tour length ~s~%" (t-l-tour))
  (if (not time) (2-opt) (terpri) (cm:time (2-opt)))
  (format *standard-output* "~%      2-OPT tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) 2-opt) lengths)
  (if (not time) (n-n-t) (terpri) (cm:time (n-n-t)))
  (format *standard-output* "~%      NEAREST-NEIGHBOR tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) n-n-t) lengths)
  (if (not time) (2-opt) (terpri) (cm:time (2-opt)))
  (format *standard-output* "~%      2-OPT NEAREST-NEIGHBOR tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) 2-opt n-n-t) lengths)
  (if (not time) (n-i-t) (terpri) (cm:time (n-i-t)))
  (format *standard-output* "~%      NEAREST-INSERTION tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) n-i-t) lengths)
  (if (not time) (2-opt) (terpri) (cm:time (2-opt)))
  (format *standard-output* "~%      2-OPT NEAREST-INSERTION tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) 2-opt n-i-t) lengths)
  (if (not time) (r-i-t) (terpri) (cm:time (r-i-t)))
  (format *standard-output* "~%      RANDOM-INSERTION tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) r-i-t) lengths)
  (if (not time) (2-opt) (terpri) (cm:time (2-opt)))
  (format *standard-output* "~%      2-OPT RANDOM-INSERTION tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) 2-opt r-i-t) lengths)
  (if (not time) (f-i-t) (terpri) (cm:time (f-i-t)))
  (format *standard-output* "~%      FARTHEST-INSERTION tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) f-i-t) lengths)
  (if (not time) (2-opt) (terpri) (cm:time (2-opt)))
  (format *standard-output* "~%      2-OPT FARTHEST-INSERTION tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) 2-opt f-i-t) lengths)
  (if (not time) (c-i-t) (terpri) (cm:time (c-i-t)))
  (format *standard-output* "~%      CHEAPEST-INSERTION tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) c-i-t) lengths)
  (if (not time) (2-opt) (terpri) (cm:time (2-opt)))
  (format *standard-output* "~%      2-OPT CHEAPEST-INSERTION tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) 2-opt c-i-t) lengths)
  (if (not time) (n-a-t) (terpri) (cm:time (n-a-t)))
  (format *standard-output* "~%      NEAREST-ADDITION tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) n-a-t) lengths)
  (if (not time) (2-opt) (terpri) (cm:time (2-opt)))
  (format *standard-output* "~%      2-OPT NEAREST-ADDITION tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) 2-opt n-a-t) lengths)
  (if (not time) (f-a-t) (terpri) (cm:time (f-a-t)))
  (format *standard-output* "~%      FARTHEST-ADDITION tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) f-a-t) lengths)
  (if (not time) (2-opt) (terpri) (cm:time (2-opt)))
  (format *standard-output* "~%      2-OPT FARTHEST-ADDITION tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) 2-opt f-a-t) lengths)
  (if (not time) (s-t 2) (terpri) (cm:time (s-t 2)))
  (format *standard-output* "~%      STRIP tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) s-t) lengths)
  (if (not time) (2-opt) (terpri) (cm:time (2-opt)))
  (format *standard-output* "~%      2-OPT STRIP tour length ~s~%" (t-l-tour))
  (push '(,(t-l-tour) 2-opt s-t) lengths)
  (print (sort lengths #'< :key #'car))
  )
```

```
(*defun compare-times ()                             ;JOEL: CAN WE CHUCK FROM HERE ON?
  (print "Length of Random Tour:")
  (print (t-l-rand))
  (print "NEAREST NEIGHBOR")
  (cm:time (n-n-t))
  (print (t-l-tour))
  (cm:time (2-opt))
  (print (t-l-tour))
  (print " farthest INSERTION")
  (cm:time (f-i-t))
  (print (t-l-tour))
  (print "2opt")
  (cm:time (2-opt))
  (print (t-l-tour))
  (print "STRIPS")
   (cm:time (strip-tour 2))
  (print (t-l-tour))
  (print "2opt")
  (cm:time (2-opt))
  (print (t-l-tour)))
```