**Evolutionary Divide and Conquer (I): a novel genetic approach to the TSP.**

by

Christine L. Valenzuela and Antonia J. Jones

**Abstract.** Experiments with genetic algorithms using permutation operators applied to the Travelling Salesman Problem (TSP) tend to suggest that these algorithms fail in two respects when applied to very large problems: they scale rather poorly as the number of cities $n$ increases, and the solution quality degrades rapidly. We propose an alternative approach for genetic algorithms applied to hard combinatoric search which we call *Evolutionary Divide and Conquer* (*EDAC*). This method has potential for any search problem in which knowledge of good solutions for subproblems can be exploited to improve the solution of the problem itself. The idea is to use the genetic algorithm to explore the space of *problem subdivisions* rather than the space of solutions themselves. We give some preliminary results of this method applied to the geometric TSP.

**Keywords:** Evolutionary algorithms, Geometric TSP, Divide and Conquer, Karp's algorithm.

DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

UNIVERSITY OF LONDON

180 Queen's Gate, London, SW7 2BZ

Telephone: 071-589 5111 Ext: 5017/5096
Telefax: 071-581 8024
E-Mail: ajj@doc.ic.ac.uk

Date/version: 12 July 2001
Evolutionary Computation **1**(4), 313-333, 1994.

**Evolutionary Divide and Conquer (I): a novel genetic approach to the TSP.**

*CONTENTS*



*List of figures*

# Evolutionary Divide and Conquer (I): a novel genetic approach to the TSP.

*Christine L. Valenzuela and Antonia J. Jones*

## 1. Introduction.

Our experience with genetic algorithms using permutation operators applied to the Geometric Travelling Salesman Problem (TSP) suggests that these algorithms fail in two respects when applied to very large problems: they scale rather poorly as the number of cities $n$ increases, and the solution quality degrades rapidly. We shall present detailed results to illustrate these observations in a more comprehensive discussion. However, our goal here is to describe a new approach which we are developing that is designed to overcome these problems.

We call our alternative method, for genetic algorithms applied to hard combinatoric search, *Evolutionary Divide and Conquer* (*EDAC*). This approach has potential for any search problem in which knowledge of good solutions for subproblems can be exploited to improve the solution of the problem itself. The idea is to use the genetic algorithm to explore the space of *problem subdivisions* rather than the space of solutions themselves. We give some preliminary results of this method applied to the geometric TSP. Essentially we are suggesting that intrinsic parallelism is no substitute for divide and conquer in hard combinatoric search and we aim to have both.

Our goal has been to develop a genetic algorithm capable of producing reasonable quality solutions for problems of several thousand cities, and one which will scale well as the problem size $n$ increases. 'Scaling well' in this context almost inevitably means a time complexity of $O(n)$ or at worst $O(n\log n)$. This is a fairly severe constraint, for example given a list of $n$ city co-ordinates the simple act of computing all possible edge lengths, a $O(n^2)$ operation is excluded. Such an operation may be tolerable for $n = 5000$ but becomes intolerable for $n = 100{,}000$.

Given the self-imposed scaling constraint the other two important axes of comparison are the quality of solutions and the actual run time. To provide some basis for comparison we contrast our approach with the standard *2-Opt*.

*TSP algorithms.* The best *exact solution* methods for the travelling salesman problem are capable of solving problems of several hundred cities [Grötschel 1991], but unfortunately excessive amounts of computer time are used in the process and, as $n$ increases, any exact solution method rapidly becomes impractical. For large problems we therefore have no way of knowing the exact solution, but in order to gauge the solution quality of any algorithm we need a reasonably accurate estimate of the minimal tour length. This is usually provided in one of two ways.

For a uniform distribution of cities the classic work by Beardwood, Halton and Hammersley (BHH) [Beardwood 1959] obtains an asymptotic best possible upper bound for the minimum tour length for large $n$. Let $\{X_i\}$, $1 \le i < \infty$, be independent random variables uniformly distributed over the unit square, and let $L_n$ denote the shortest closed path which connects all the elements of $\{X_1,...,X_n\}$. In the case of the unit square they proved, for example, that there is a constant $c > 0$ such that, with probability 1,

$$\lim_{n \to \infty} L_n \, n^{-1/2} = c \tag{1}$$

where $c > 0$ is a constant. In general c depends on the geometry of the region considered.

One can use the estimate provided by the BHH theorem in the following form: the expected length $L_n^*$ of a minimal tour for an $n$-city problem, in which the cities are uniformly distributed in a square region of the Euclidean plane, is given by

$$L_n^* \approx 0.765\sqrt{nR} \tag{2}$$

where R is the area of the square and the constant 0.765 has been determined empirically [Stein 1977]. In all our experiments we fix the area $R$ so that $L_n^* = 100$ and the *percentage excess* of a tour length is the percentage excess *relative to this estimate*.

A second possibility would be to use a problem specific estimate of the minimal tour length which gives a very accurate estimate: the *Held-Karp lower bound* [Held 1970], [Held 1971]. Computing the *Held-Karp lower bound* is an iterative process involving the evaluation of Minimal Spanning Trees for $n$-1 cities of the TSP followed by *Lagrangean relaxations*. However, the typical percentage excess of the present version of our algorithm does not really require us to implement this estimate.

If one seeks *approximate solutions* then various algorithms based on simple rule based heuristics (e.g. nearest neighbour and greedy heuristics), or local search tour improvement heuristics (e.g. *2-Opt*, *3-Opt* and Lin-Kernighan), can produce good quality solutions much faster than exact methods. A *combinatorial local search* algorithm is built around a 'combinatoric neighbourhood search' procedure, which given a tour, examines all tours which are closely related to it and finds a shorter 'neighbouring' tour, if one exists. Algorithms of this type are discussed in [Papadimitriou 1982]. The definition of 'closely related' varies with the details of the particular local search heuristic.

The particularly successful combinatorial local search heuristic described by Lin and Kernighan [Lin 1973] defines 'neighbours' of a tour to be those tours which can be obtained from it by doing a limited number of interchanges of tour edges with non-tour edges. The slickest local heuristic algorithms[1], which on average tend to have complexity $O(n^\alpha)$, for $\alpha > 2$, can produce solutions with approximately 1-2% excess for 1000 cities in a few minutes. However, for 10,000 cities the time escalates rapidly and one might expect that the solution quality also degrades, see [Gorges-Schleuter 1990], p 101.

[1] The most impressive results in this direction are due to David Johnson at AT&T Bell Laboratories - mostly reported in unpublished Workshop presentations.

An *approximation scheme* A is an algorithm which given problem instance *I* and $\varepsilon > 0$ returns a solution of length A(I,ε) such that

$$\frac{|A(I,\varepsilon) - L_n(I)|}{L_n(I)} \leq \varepsilon \tag{3}$$

Such an approximation scheme is called a *fully polynomial time approximation scheme* if its run time is bounded by a function that is polynomial in both the instance size and 1/ε. Unfortunately the following theorem holds, see for example [Lawler 1985], p165-166.

**Theorem.** *If $\wp \neq N\wp$ then there can be no fully polynomial time approximation scheme for the TSP, even if instances are restricted to points in the plane under the Euclidean metric.*

Although the possibility of a fully polynomial time approximation scheme is effectively ruled out, there remains the possibility of an approximation scheme that although it is not polynomial in 1/ε, does have a running time which is polynomial in *n* for every fixed $\varepsilon > 0$. The Karp algorithms, based on cellular dissection, provide 'probabilistic' approximation schemes for the geometric TSP.

**Theorem** [Karp 1977]. *For every ε > 0 there is an algorithm A(ε) such that A(ε) runs in time C(ε)n+O(nlogn) and, with probability 1, A(ε) produces a tour of length not more than 1+ε times the length of a minimal tour.*

The Karp-Steele algorithms [Steele 1986] can *in principle* converge in probability to near optimal tours very rapidly. Cellular dissection is a form of divide and conquer. Karp's algorithms partition the region *R* into small subregions, each containing about *t* cities. An exact or heuristic method is then applied to each subproblem and the resulting subtours are finally patched together to yield a tour through all the cities.

To date the best *genetic algorithms* designed for TSP problems have used permutation crossovers for example [Davis 1985], [Goldberg 1985], [Smith 1985], or edge recombination operators [Whitley 1989], and required massive computing power to gain very good approximate solutions (often actually optimal) to problems with a few hundred cities [Gorges-Schleuter 1990]. Gorges-Schleuter cleverly exploited the architecture of a transputer bank to define a topology on the population and introduce local mating schemes which enabled her to delay the onset of premature convergence. However, this improvement to the genetic algorithm is independent of any limitations inherent in permutation crossovers.

*Genetic algorithms based on Karp's approach*. In practice a one-shot deterministic Karp algorithm yields rather poor solutions, typically 30% excess (with simple patching) when applied to 500 - 1000 city problems.  Nevertheless, we believe it is a good starting point for exploring *EDAC* applied to the TSP. Our reasons are two-fold. First, there is some probabilistic asymptotic guarantee of solution quality as the problem size increases. Second, the time complexity is about as good as one can hope for, namely O(*n*log*n*). The run time of a genetic algorithm based on exploring the space of 'Karp-like' solutions will be proportional to *n*log*n* multiplied by the number of times the Karp algorithm is run, i.e. the number of individuals tested.

Thus we have reasonable probabilistic guarantees for both the complexity and the solution quality. For large enough problems several thousand Karp runs (individuals tested) will be much faster than a combinatorial local search heuristic algorithm. The practical objection might very well be that 'large enough' turns out to be very large indeed but still this would seem to be an approach worthy of study.

## 2. Developing a Divide-and-Conquer approach.

*Bisection method 1.* Let rectangle *R* contain *m* cities. Let *y* be the y-coordinate of the [m/2]*th* closest city to the top edge of *R*. A horizontal cut through *y* subdivides *R* into two rectangles, a upper rectangle and a lower rectangle. The situation is illustrated in Figure 1. The effect is to place half the cities either side of the bisecting line with at least one city on the bisector. In a similar fashion, a vertical cut could be applied to bisect the cities through *x*, which is the x-coordinate of the [m/2]*th* closest city to the left edge of *R*. In Karp's first algorithm the direction of the cut is always parallel to the shorter side of the rectangle. Karp showed that by minimizing the lengths of the perimeters of the rectangles he was able to minimise the expected lengths of the tours. The preliminary results reported in section 4 used this method of bisection.

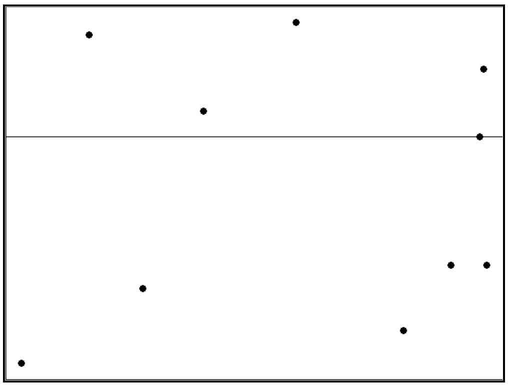

**Figure 1**. Horizontal bisection of a 10 city problem.

*Bisection method 2*. Karp's second algorithm partitions the problems by exactly bisecting the area of the rectangle parallel to the shorter side. This produces, however, a more complex situation for the patching algorithm as there is no shared city.

*Bisection method 3*. In order to keep the patching algorithm simple, the original bisection method 1 was replaced by the following bisection rule:

- Rectangles are bisected through the city nearest to the true area bisection line.

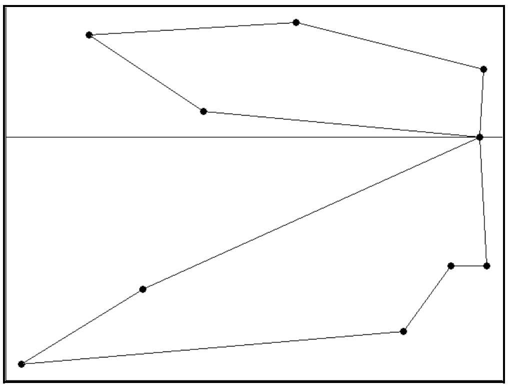

**Figure 2**. Subproblems solved.

In this way a shared city is maintained and to some degree the simplest features of the first and second method are combined. The main advantage of this modified bisection method is the fact that the cities in the region of bisection need not be sorted, they are simply partitioned into two sets either side of the bisection line, producing either a left-hand set and a right-hand set, or an upper set and a lower set, depending upon the direction of bisection. The complexity of a single application of this operation is $O(n)$ (instead of $O(n\log n)$) and the total cumulative effect is $O(n\log n)$.

*Solving the subproblems*. The subproblem size t is kept as small as possible, typically $5 \leq t \leq 8$. We tried various techniques for solving the subproblems, including exhaustive search.

However *2-Opt* was chosen as the main method for this preliminary work because of its speed and simplicity and the fact that it can be applied to larger subproblems, without large time penalties, if required.

*The simple patching algorithm*. Figure 1 shows the bisection technique resulting in a shared city occurring on the line separating each adjacent pair of subproblems. After the subproblems have been solved, as in Figure 2, the four incident edges to the shared city must be reduced to two. This is achieved by the removal of two of the incident edges, one from each subproblem, and the creation of a new edge between the two "stranded" cities. As there are only four possible ways this patching can be done, they are all tried and one that results in the shortest patched tour is selected. For later purposes the new edge can be added to an edge list L as a candidate for repair.

Figure 3 illustrates the best patching obtained in this way for the 10 city problem used in Figure 1 and Figure 2.

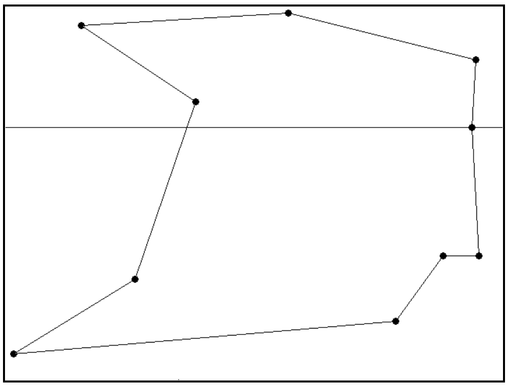

**Figure 3**. Patched solution.

*Recursive divide and conquer*. In Karp's algorithms the bisection technique is repeated recursively until the individual subproblem sizes are at or below some predetermined maximum value, this is illustrated in Figure 4. When the resulting subtours have been solved Karp then patches the solutions globally using two operations called *Loop* and *Pass*.

The final *EDAC* algorithm described here differs from Karp's in three important respects:

- A genetic algorithm determines the direction of bisection (horizontal or vertical) used at each stage.

- The patching technique described above is used to join the subproblem solutions recursively in pairs instead of patching globally as Karp does.

- Because simple patching turns out in practice to be a major source of error the new edges created by patching (on the list L) are reviewed for repair. The repair procedures ultimately used are called *Recursive-Fast-2-repair* and *Far-repair*. These will be described later.

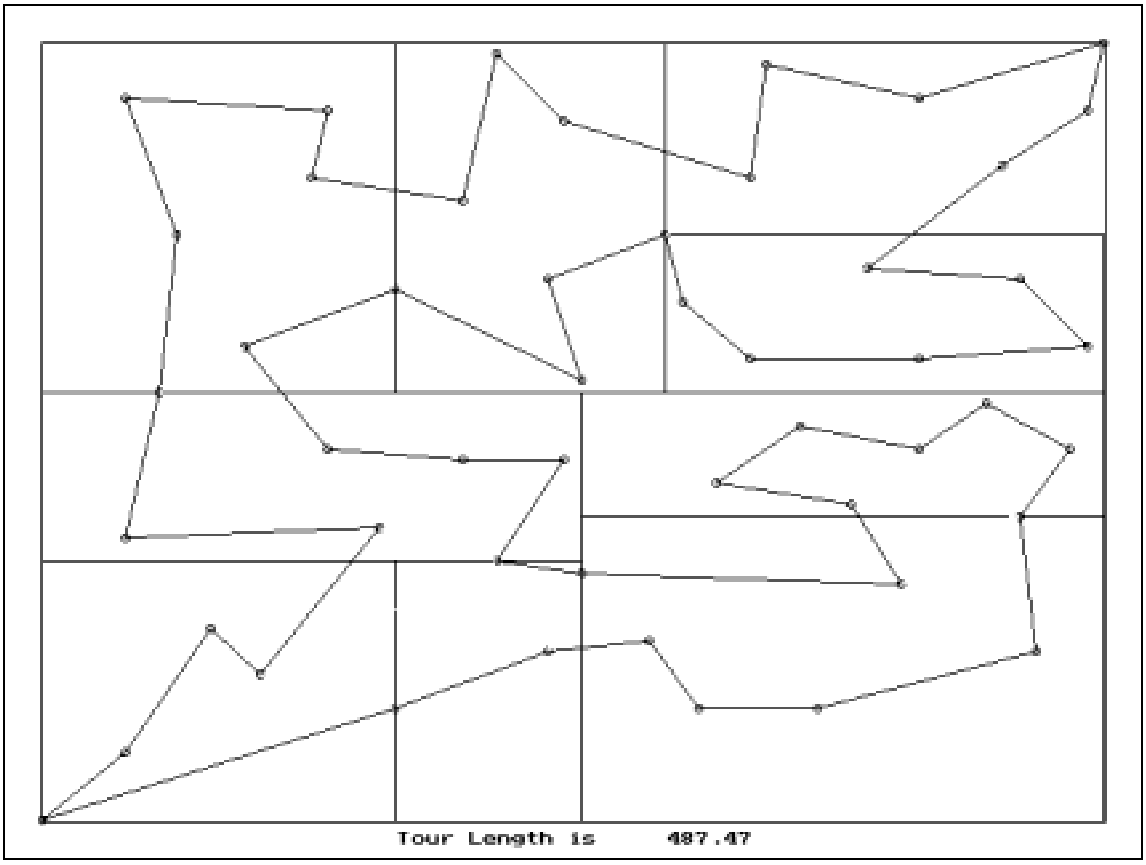


**Figure 4**. Solution to 50 City Problem using Karp's deterministic bisection method 1.

## 3. Implementation of a preliminary *EDAC* algorithm.

For this study we chose an extremely simple genetic algorithm based on Cavicchio's preselection paradigm, in which a child either replaces a weaker parent or dies (in the latter case we still count this evaluation as a trial). Cavicchio's technique in the form used has the virtue of extreme simplicity and low computational overhead while successfully maintaining diversity in the relatively small population of 100. Gorges-Schleuter, for example, reports excellent results for a closely related algorithm, in which the superior offspring replace one or other parent [Gorges-Schleuter 1990].

Our main initial objective is to demonstrate that genetic algorithms have potential in this area and we leave work on improving the genetic algorithm to a later date. The Genetic Algorithm used for the present is outlined in Algorithm 1. Here there is no need to tune such factors as the crossover rate or the relationship between tour length and fitness.

```
begin
        Generate N random structures {N is the population size}
        Evaluate tour length produced by each structure and store each one
        store best-so-far
        repeat
                select next (first) structure
                select a second structure stochastically from a uniform distribution
                apply crossover to produce offspring
                apply mutation to offspring
                evaluate tour length produced by offspring
                if offspring better than weaker parent then it replaces it in population
                if offspring better than best-so-far then it replaces best-so-far
        until stopping condition satisfied
        print best-so-far
end.
```

Algorithm 1. The Genetic Algorithm.

*The genotype representation and crossover*. The data structure for the genotype required some thought. Our initial view favoured a binary tree structure in which each node in the tree is labelled with either a 'vertical' or 'horizontal' cut instruction. This structure lends itself naturally to the recursive nature of both the bisection and the construction of the resulting tour. However, as the tree becomes deeper the link between a cut instruction at a node and the *geometrical region* to which that instruction applies becomes progressively more tenuous. Performing a crossover between two binary trees (by exchanging subtrees, for example) could easily produce a child where the subtrees were dissecting completely different geometrical regions for the child than they were for the parents.

The representation actually used was a $p$ by $p$ binary array which is correlated with the geometrical regions by imagining the array superimposed on the TSP square. Given some rectangle to be bisected, the partitioning algorithm selects the array component which most closely corresponds to the centre of the rectangle, and this component (1/0) determines the direction of the current cut (horizontal/vertical). This maintains a close correspondence between the chromosomes of the genotype and the geometrical locality of the centre of the rectangle. In the current study we used $40 \leq p \leq 80$.

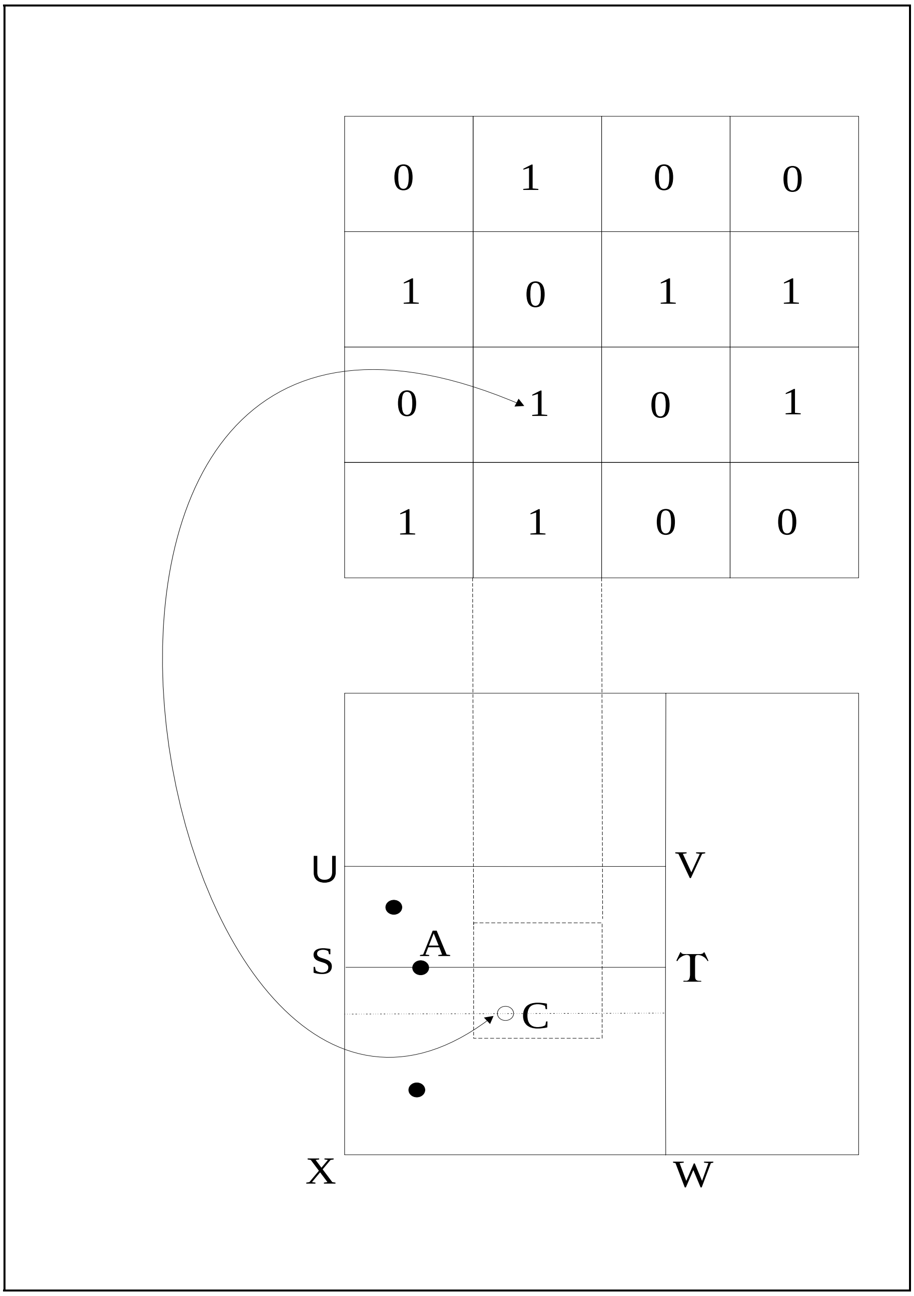


**Figure 5** Relationship between the genotype (top) and the direction of bisection ST.

Figure 5 illustrates the geometrical relationship between the genotype with p = 4 (top) and the direction of bisection of the rectangle UVWX (bottom). The centre of the rectangle UVWX is C which corresponds to the square indicated in the genotype. The genotype entry of ’1’ denotes a horizontal bisection of the area. The city nearest to the bisector through C is A, and the horizontal line ST through A is the bisector actually constructed by method 3 (section 2).

Given that the genotype is a binary array the crossover becomes relatively trivial, requiring only the swapping of binary elements between two parent arrays. In our current implementation we select the x or y axis with equal probability and then choose two cut points at random on the selected axis with the proviso that the distance between the two points must be more than a third and less than two thirds along this axes of the genotype. The reason for the 1/3, 2/3 restriction is to ensure that each offspring contains a reasonable proportion of genetic material from each parent thus attempting to avoid the early proliferation of a few superior individuals.

The first cut relates to the whole region and, as bisection progresses, the region corresponding to a single array element becomes geometrically smaller and the cities within the region less uniformly distributed. Since the genotype is a binary array one can envisage that a suitably modified schema theorem may possibly apply. Although a schema theorem by itself would be no guarantee of progress [Grefenstette 1989], it might be useful in the overall scheme of things if the optimal decision to cut horizontally or vertically near any rectangle centre is correlated with the distribution of cities. It seems likely that this is the case.

*The size of the genotype*. The array acts as a look-up table for the genetic algorithm with only a few points being accessed for each application of the partition algorithm. In this respect it is analogous to the DNA in natural chromosomes for which only a small part is active in each cell, the remainder being "switched off". Certainly $p$ must be at least $\sqrt{(n/t\text{-}1)}$, but for extreme distributions of cities a given value of $p$ may not provide sufficient resolution and a larger value may be required. Although suitable array sizes for TSP problems of different magnitudes is an obvious area for investigation, it is worth noting that whilst a population of large arrays would occupy much more memory than a population of small arrays it would not consume significantly more computing time. More copying would obviously be required to produce the offspring and more mutations to achieve a given mutation rate, but the number of times the genotype is accessed as a look-up table is dependent only upon the number of partitions required for a particular problem and is completely independent of the genotype size.

*Mutation*. A random point of the array is inverted such that a horizontal instruction becomes a vertical instruction and vice versa. For each genotype created by the genetic algorithm 0.1% of array components were mutated.

## 4. Random Karp-like solutions versus GA Karp-like solutions.

If we maintain the subproblem size, $t$, and increase the number of cities in the TSP, then a partition better than Karp’s becomes progressively harder to find by randomly choosing a horizontal or vertical bisection at each step. If the problem size is $n \sim 2^k t$, where $2^k$ is the number of subsquares, then the corresponding genotype requires at least $n/t$ - 1 bits. The size of the partition space is 2 to the power $p^2$, which for p = 80 (the value we used for n = 5000) is approximately exp(4436). For n = 5000 the size of permutation search space, roughly estimated using Stirling’s formula, is around exp(37586). Thus searching partition space is easier than searching permutation space but still the hard nature of the bisection problem provides sufficient motivation for exploring genetic algorithms as a possible adaptive search technique.

In Figure 6 we present the results of 1000 attempts at dissecting a 500 city problem "tossing a coin" at each stage to determine whether to perform a horizontal or vertical bisection and then using bisection method 1. Again the subproblem size is about 6. Not one of the thousand random trials produced a solution as good as the deterministic Karp bisection technique which gave 127.96.

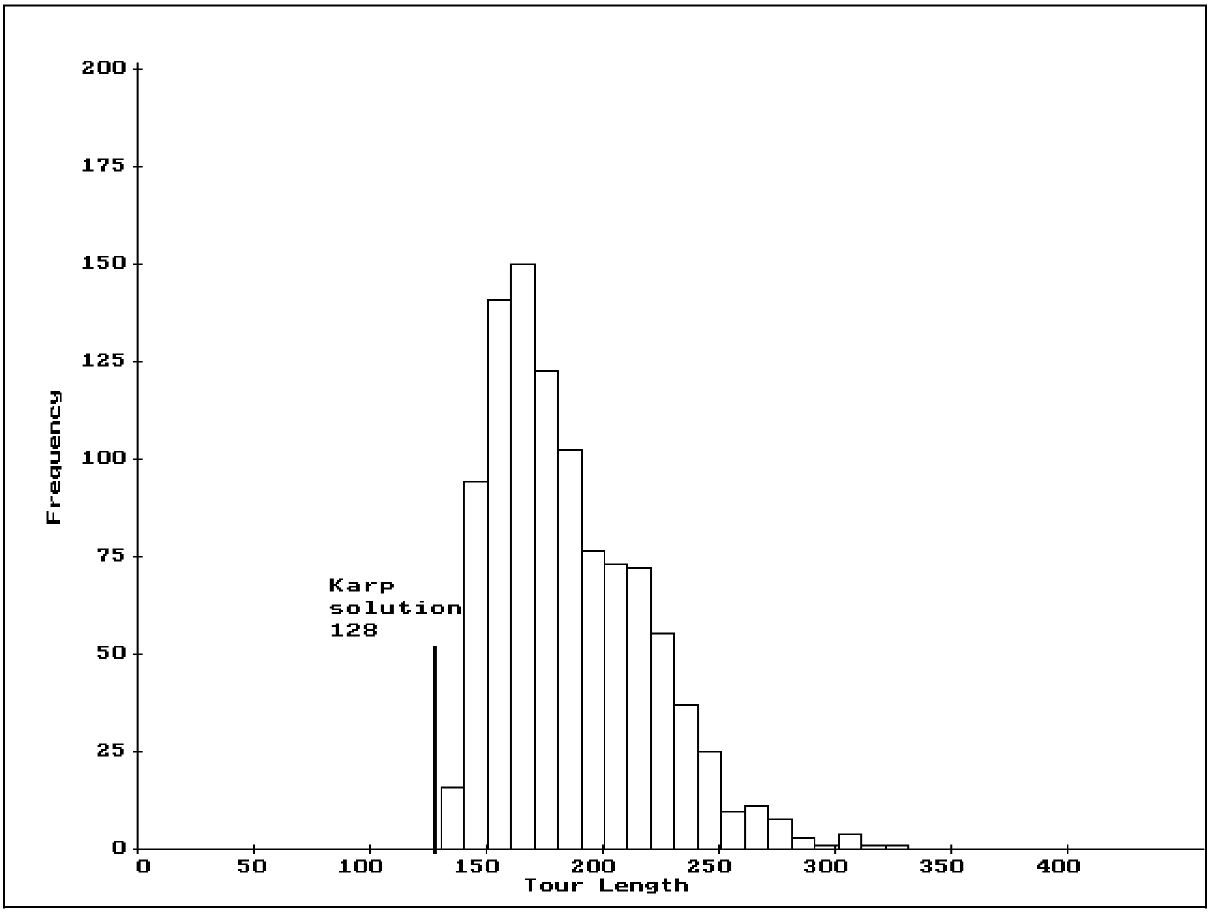


**Figure 6**. Results of 1000 Random Dissection Experiments on a 500 City Problem using simple patching.

A single run of *EDAC* using the same bisection method, for 100 generations with a population of 100 (10000 individuals examined), produced a solution of 122.58 thus at n = 500 the *EDAC* approach proved capable of improving upon the deterministic Karp algorithm. This was reassuring since it demonstrated that the method had some hope of success. Nevertheless, the solution quality was still unsatisfactory and this led us to search for ways to improve the quality of the Karp-like solutions produced by *EDAC*.

## 5. Improving the quality of Karp-like solutions: *Recursive-Fast-2-repair*.

Using bisection method 3 (see section 2) gives a overall improvement in run time without seeming to affect the solution quality much either way, and all subsequent results reported herein used this method. It became clear that in order to eliminate the more obvious defects introduced by patching it would be necessary to weaken the link between genotype and phenotype by using a repair mechanism on the Karp-like solutions generated by the genetic

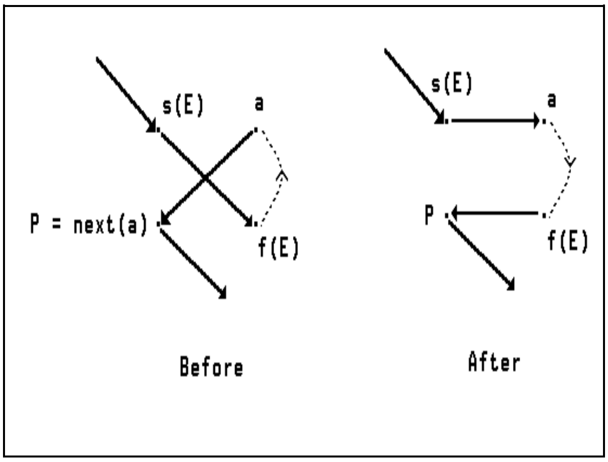


**Figure 7**. A *2-move* on edge E involving a neighbour *a*.

algorithm. We have not yet explored all the options in this direction, but in this study we initially opted for a method we called *Global-Fast-2-repair* which we subsequently modified to *Recursive-Fast-2-repair*.

The constraints on the repair technique are fairly obvious: it should address errors typical of Karp-like solutions, it should ideally be O($n$), and it should use information which can be acquired at low time cost.

The most basic of the combinatorial tour repair heuristics is *2-Opt* which proceeds by a series of pairwise edge exchanges called *2-moves*. Figure 7 illustrates a *2-move* for the intuitive edge-crossing case, but it is possible to effect a *2-move* improvement even in cases where the two replaced edges do not cross.

To find a *2-move* which decreases the tour length a simple *2-Opt* must consider all edge pairs for a possible exchange, which itself requires a O($n^2$) calculation. If a *2-move* leads to a decrease of the tour length the edge exchange is accepted and this requires inverting and rewriting part of the tour. Once accepted a single *2-move* therefore costs an amount of computation time $d(n)$, which depends on the length of the segment to be inverted, i.e. the quality of the current tour.[2] If the current tour is very bad, $d(n)$ is proportional to $n$. For good tours $d(n)$ can be much less, proportional to $n^\alpha$, where $\alpha < 1$. This leads to an overall time complexity of O($n^2 d(n)$) and it is easy to prove that the worst case analysis is O($n^3$) (see, [Lawler 1985], p 164).

First described in [Martin 1992], *Fast-2-Opt* is a modification of the standard *2-Opt* which restricts the number of *2-moves* considered. For the geometric TSP, when using *2-Opt* it is silly to consider pairs of edges which are far apart in the physical space of the problem. One way in which *Fast-2-Opt* makes this idea precise is by maintaining a list for each city of the edge lengths to (say) the 10 nearest neighbours, and restricting *2-moves* to these edges. Unfortunately constructing these lists is itself at least a O($n^2$) operation if one is not given all the edge lengths to begin with. Fortunately, as shortly described, in the context of an evolutionary search this problem is easily overcome.

To further encourage rapid termination Martin *et al* introduced the guard condition which depends on *Min*, the minimum edge length of all edges, and *Max*, the maximum edge length of the current tour. The guard condition is the essence of their *Fast-2-Opt* since it introduces an element of geometrical locality which restricts the number of cases to be considered. The original guard condition requires that one calculates the minimum possible edge length, a O($n^2$) calculation. We replaced this by an estimate based on the initial population, which

[2] It might appear at first sight that this cost is implementation dependent, and may possibly be avoided by skilful use of pointers. However, a number of experiments convinced us that the more tempting alternatives yielded longer run-times in practice.

means we may have missed some *2-move* improvements. This estimate could be updated as the evolutionary search progressed.

Since we already have a list L of potential edges for repair, which is generated by the simple patching process, our first attempt at a repair algorithm consisted of a modification of *Fast-2-Opt* which, in addition to the nearest neighbour lists, also used the list L. Algorithm 2 contains outline pseudocode for this procedure which we called *Global-Fast-2-repair*. A similar routine is also considered in [Gorges-Schleuter 1990].

The initial version of *Global-Fast-2-repair* did not require the $n^2$ nearest neighbour calculations. Instead the nearest neighbours to each city are estimated from the initial population of patched solutions, where each city's neighbours in the tour are candidates for insertion into the nearest neighbour lists found so far (the lists are sorted in increasing order of edge lengths). As the evolutionary search progresses and further neighbourhood information becomes available these lists could be progressively updated. However, comparisons between the initial lists and those generated by the full $O(n^2)$ calculation were quite favourable. Once the initial neighbour lists have been constructed and prior to the start of the genetic algorithm, the initial set of tours was itself subjected to *Global-Fast-2-repair*, and the tour lengths recorded. Subsequently each new tour generated by the genetic algorithm was subjected to *Global-Fast-2-repair* using the edge list L described in section 2.

Plainly Algorithm 2 terminates (each improvement decreases the tour length and there are only finitely many tour lengths), the important issue is how quickly. The initial length of the active list L is at most $n/t$ - 1, where $t$ is the number of cities in each subproblem, but L can sometimes get longer, since the edge (a, next(a)), or (prev(b),b), which is subtracted if present in L, may not (in fact) be in L. If we do not add the second edge then a much faster, but less accurate procedure, results.

However initial experiments showed that, whilst *Global-Fast-2-repair* was successful in lifting the quality of solution from 13% (using slightly more elaborate patching) to 4-5% excess, the scaling was poor. Up to $n$ = 5000 *Global-Fast-2-repair* was scaling at around $O(n^{1.7})$ and the exponent seemed to be increasing as $n$ got larger. Not adding the second edge improved the scaling to approximately $O(n^{1.3})$ but the solution quality was around 10% excess. The next step towards improving the situation was to attempt to get as much benefit from *2-moves* as possible whilst limiting the combinatoric growth of cases considered.

We modified *Global-Fast-2-repair* to become a local procedure, *Recursive-Fast-2-repair*, which is applied to repairing subsolutions rather than the whole tour. *Recursive-Fast-2-repair* succeeds each simple patching operation in the recursive construction of the global tour. Whilst the function of *Recursive-Fast-2-repair* is essentially the same as its global counterpart, its implementation is subtly different. *Recursive-Fast-2-repair* expends most of its efforts repairing small subproblems, where accepted *2-moves* require only short subtour inversions. In addition each call to *Recursive-Fast-2-repair* is initiated with an edge list L containing just one edge, the rogue edge produced by a single simple patching algorithm. *Global-Fast-2-repair*, on the other hand, is characterised by longer subtour inversions and is initiated with an edge list containing *all* the rogue edges resulting from *all* the simple patching operations.

Some results obtained from single runs of *EDAC* with *Recursive-Fast-2-repair* are presented in Table 2 (Appendix). A least-squares analysis reveals an empirical scaling of $O(n^{1.07})$, and a linear plot results from time vs $n\log n$. Thus the scaling properties meet our requirements. Unfortunately the quality of the solutions at 8-10% excess are considerably worse than those obtained using *Global-Fast-2-repair* at 4-5% excess.

```
Procedure Global-Fast-2-repair(T, L, Neighbourhood lists)
   {T is the current tour, L = L(T) is list of edges of T to be considered. Max is the
   maximum edge length of the current tour, Min is the estimated minimum edge
   length of all edges. l is the length of the neighbour lists (l = 10 in these
   experiments). s(E) is start city of edge E, f(E) is final city of edge E. next(a) and
   prev(a), for city a, are next city and previous city, respectively of current tour
   T}
begin

while L ≠ Ø do
   select edge E = (s(E), f(E)) ∈ L
   m := 1: improvement := false
      while m ≤ l and (improvement = false) do {check neighbours of s(E) & f(E)}
         a := neigh(s(E),m):           {mth neighbour of s(E) on list}
         b := neigh(f(E),m)            {mth neighbour of f(E) on list}
         if (d(s(E),a) + Min > d(s(E),f(E)) + Max) and
            (d(f(E),b) + Min > d(s(E),f(E)) + Max) then break inner while loop
         {check neighbour of s(E)}
         if d(s(E),a) + d(f(E),next(a)) < d(s(E),f(E)) + d(a,next(a)) then
            L := L - {E,(a,next(a))} + {(s(E),a),(f(E),next(a))}
            make 2-move on T           {see Figure 4}
            update Max
            improvement := true
         {check neighbour of f(E)}
         if d(s(E),prev(b)) + d(f(E),b) < d(s(E),f(E)) + d(prev(b),b) then
            L := L - {E,(prev(b),b)} + {(s(E),prev(b)),(f(E),b)}
            make 2-move on T
            update Max
            improvement := true
         m := m + 1                    {no 2-moves, check next neighbour}
      end while                        {take next edge in L}
   L = L - {E}                         {delete edge from the active list}
end while

end
```

Algorithm 2. *Global-Fast 2-repair*.

## 6. Improving the quality of Karp-like solutions: *Far-repair*.

With a view to further improving the solution quality we developed a low-cost tour improvement heuristic. In essence the scheme deletes cities from their positions in the current tour and inserts them in new positions whenever this move produces a reduction in the tour length. The algorithm, which we call *Far-repair*, is applied globally following the construction of the initial tour by simple patching and *Recursive-Fast-2-repair*.

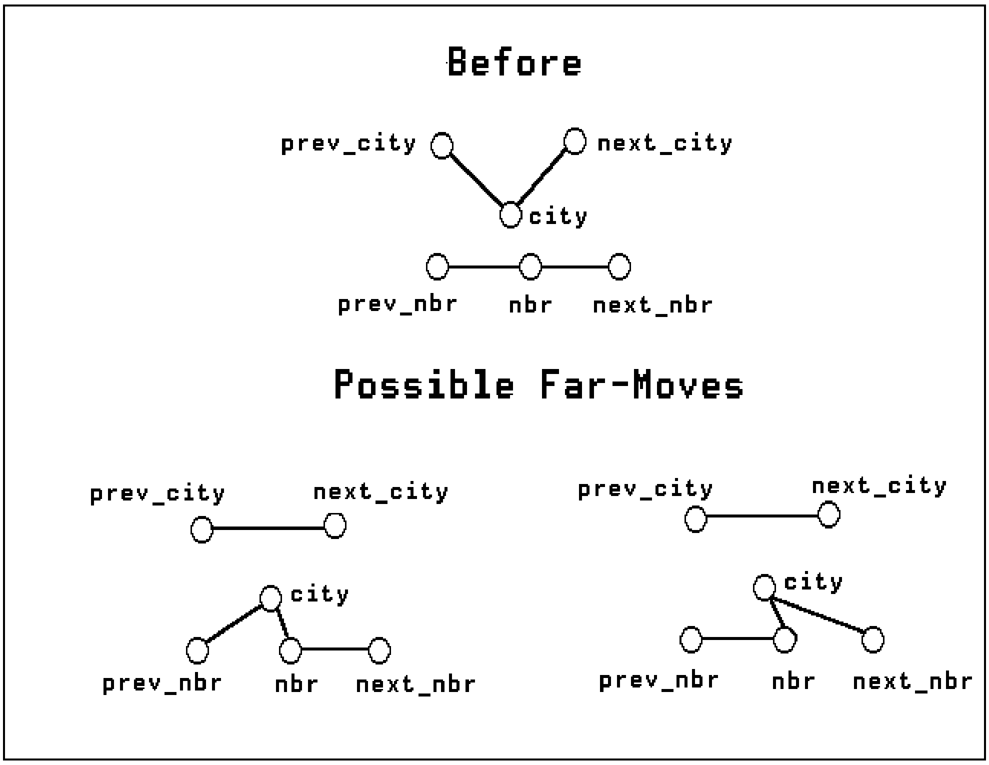


**Figure 8**. Potential *Far-moves*.

Algorithm 3 details *Far-repair* which obviously has time complexity O($n$). *Far-repair* involves exchanging three edges (a *3-move*) and so will repair defects which are beyond the scope of any *2-move*.

The lists of nearest neighbours accumulated for the *2-move* procedures are employed by *Far-repair* to ensure that the algorithm does not waste valuable time evaluating potential moves that have little chance of success. Figure 8 shows how a city is tested as a candidate for a *Far-move*. It is tried first one side of a near neighbour, then the other. The term ’far’ repair refers to the fact that individual cities can be moved to new positions in the current tour that are "far away" from their present positions in terms of where they are on the permutation list defining the tour.

```
Procedure Far-repair(Tour, Position, Which_Slot, Nbhd)

   {Attempts to move individual cities - see Figure 5.
   Position[ ] is an array of pointers to the location of the city in the tour.
   Which_Slot[ ] defines which city is in a given tour position.
   Nbhd[ ] specifies the l nearest neighbours of each city.}

begin

for each city in Problem do      {check neighbours of city}
   a := Position[city]           {pointer to city in tour}
   prev_city := Which_Slot[a-1]
   next_city := Which_Slot[a+1]
   i := 1
      while i ≤ l do             {each near neighbour}
         nbr = Nbhd[city][i]
         b = Position[nbr]       {pointer to neighbour in tour}
         prev_nbr = Which_Slot[b-1]
         next_nbr = Which_Slot[b+1]
         {investigate move}
         if d(prev_city,next_city) + d(prev_nbr,city) + d(city,nbr)
            < d(prev_city,city) + d(city,next_city) + d(prev_nbr,nbr) then
            delete city from current position
            insert it between prev_nbr and nbr
            break while loop
         else if d(prev_city,next_city) + d(nbr,city) + d(city,next_nbr)
               < d(prev_city,city) + d(city,next_city) + d(nbr,next_nbr) then
            delete city from current position
            insert it between nbr and next_nbr
            break while loop
         i := i + 1              {no far-moves, check next neighbour}
      end while                  {take next city in problem}
end for

end
```

Algorithm 3. *Far-repair*.

## 7. Some preliminary results.

In practice the simplest *2-Opt* has a time complexity of slightly more than O($n^2$), see Figure 9 - bottom trace. Here each data point represents the average for 100 random tours subjected to *2-Opt* for fixed problems of size of $n$ = 100, 200, 500, 1000, 2000, and 5000 respectively. The line represents the least squares fit and has slope 2.028. Accuracy for the simple *2-Opt* is around 8-9% excess.

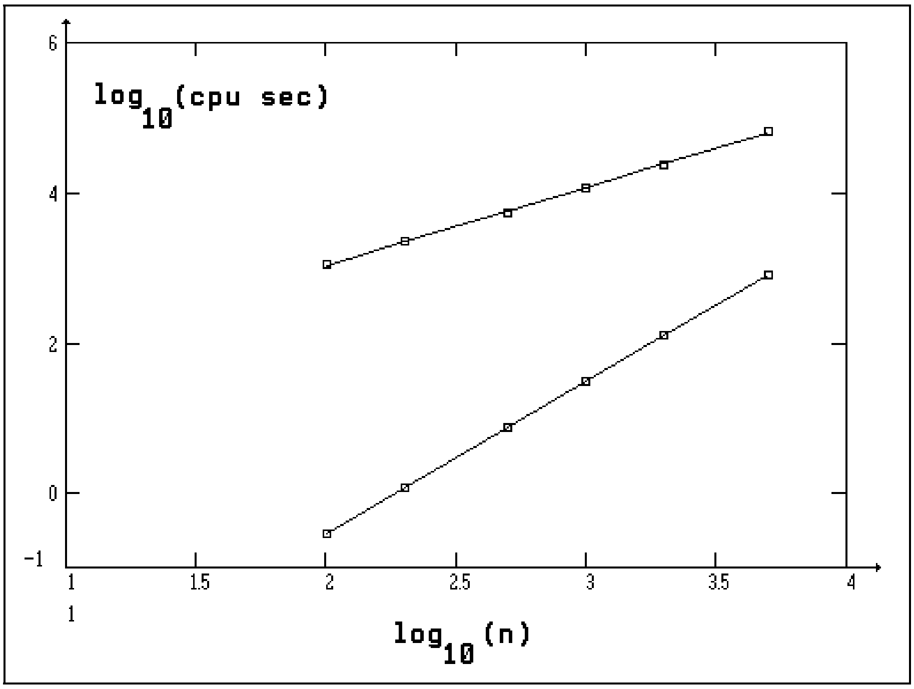


**Figure 9**. The *EDAC* (top) and simple *2-Opt* (bottom) time complexity (log scales).

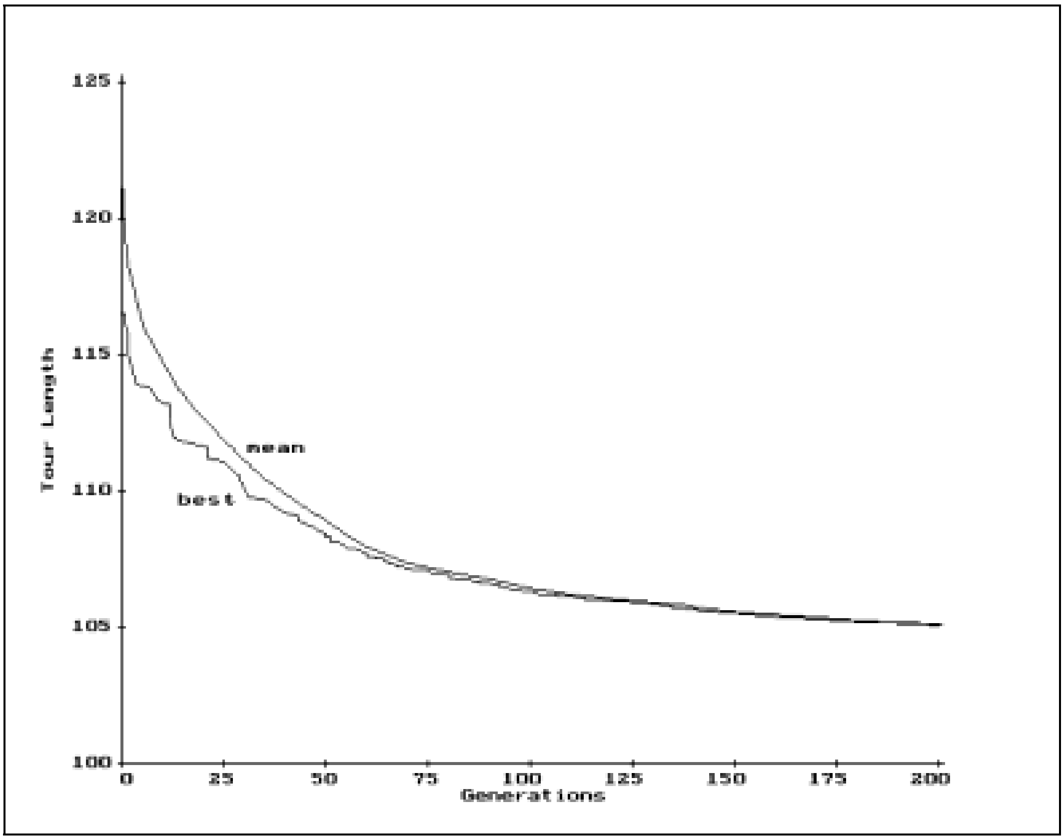


**Figure 10**. *EDAC* for 200 Generations on a 5000 City Problem.

Table 3 (Appendix) gives some results for *EDAC* with *Recursive-Fast-2-repair + Far Repair*. These results are plotted in the top trace of Figure 9. The least squares slope is about 1.04 and the accuracy is at worst 6%. It may seem counter-intuitive that the scaling exponent 1.04 (using both repair techniques) is less than 1.07 (when only one is used). Of course, for sufficiently large *n* the larger exponent must dominate. However, when *n* is small *Far-repair* takes up a high proportion of the total cpu time and as *n* gets larger this proportion decreases rapidly.

Although these are preliminary results it is quite clear that the general method of approach is viable. Our initial attempt succeeded in the goal of designing a genetic algorithm capable of reliably giving solutions with around 5-6% excess for geometric TSP problems involving several thousand cities within 100 generations (10,000 individuals tested).

In Figure 10 the *EDAC* algorithm has been allowed to run for 200 generations (as opposed to the normal 100) and although the tour quality is still improving it is clear that, without more effective repair heuristics, further tour quality improvement will be marginal.

This particular 200-generation run produced a tour having a 5% excess - see Figure 13. On the downside it is clear that, whilst viable, the method is probably not yet practical[3]. The natural comparison would perhaps be with iterated Lin-Kernighan. However, in reviewing the compute-time figures one should bear in mind that we were interested in scaling and made no attempt to optimise the *EDAC* code.

[3] For example, wildly extrapolating our figures gives the breakeven point with *2-Opt* at around n = 422,800 requiring some 74 cpu days! Of course, other things would collapse before then.

**8. What is the overall contribution of the genetic algorithm?**

In order to assess the contribution of the genetic algorithm over and above both random search and Karp's deterministic bisection method, we ran some control experiments.

Figure 11 shows a distribution of 10000 Karp-like random trials on a 500 city problem. *Recursive-Fast-2-repair* + *Far-repair* are used here, and other parameters are set to make the 10000 trials comparable to a single run of *EDAC* with these heuristics. (*EDAC* has a population size of 100, and runs for 100 generations).

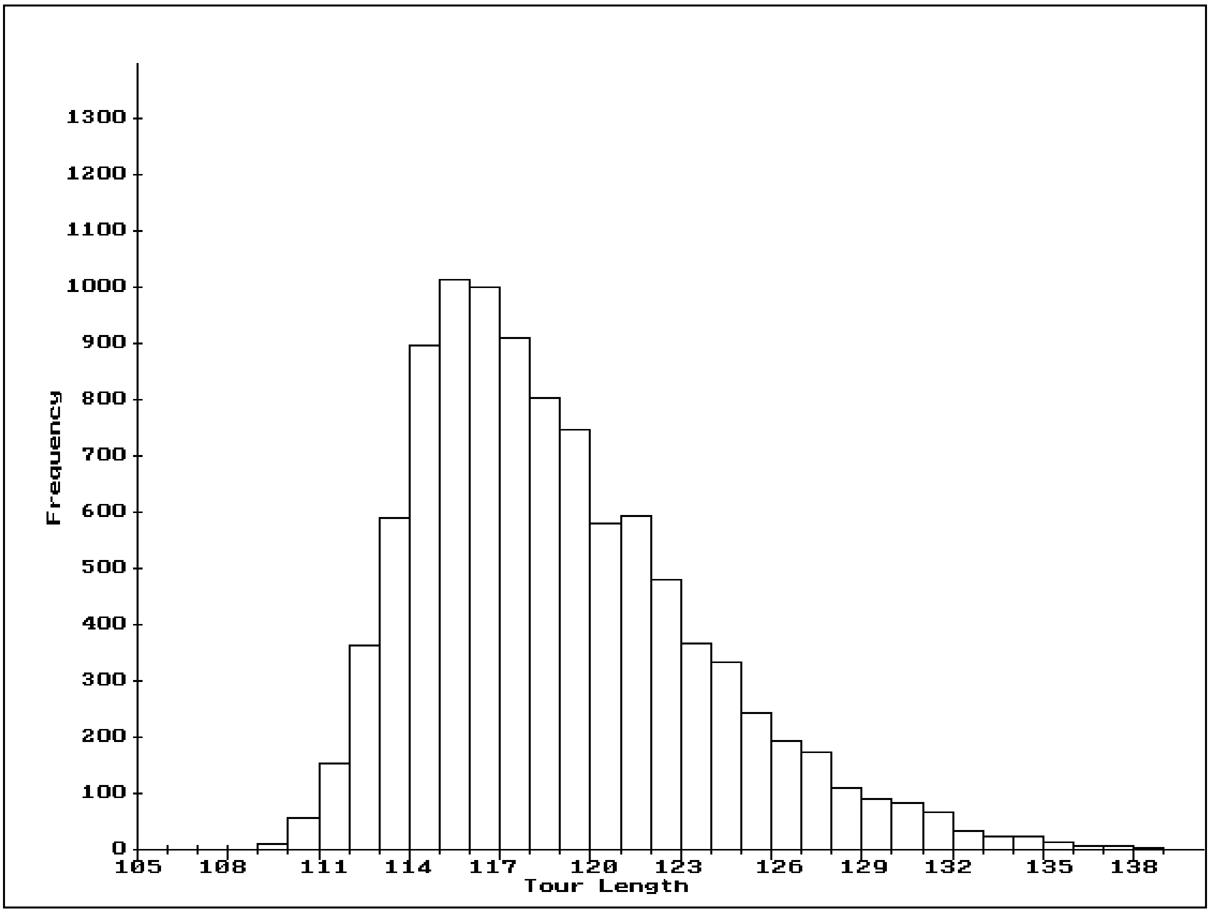


**Figure 11**. Random search + repair heuristics for a 500 city problem. The deterministic Karp + repair heuristics solution yields a tour length 112.33.

Table 1 summarizes random search experiments on 1000 and 2000 city problems as well as the 500 city results, and compares these with the results obtained by running *EDAC* and recording the best solution produced. Entries in the *EDAC* column represent the mean of 5 runs of *EDAC* for the 500 and 1000 city problems, and the mean of 2 runs for the 2000 city problem.

Examining the table, for n = 500 the mean of the distribution of randomly generated Karp-like solutions plus repair heuristics is 118.87. The best of 10000 such trials gives a solution 2.13 Standard Deviations (SD) better than the mean, whereas the same number of evaluations using the *EDAC* algorithm plus repair heuristics yields an improvement 2.90 SD better than the mean, a difference of 0.77 SD between the two. For 1000 cities this difference is 1.38 SD and for 2000 cities it is 1.82 SD. The *EDAC* algorithm is steadily improving its performance relative to random search as the problem size increases. Since the distribution is skewed the relative improvement using the genetic algorithm is actually better than these figures indicate.

It is interesting to note that the one-shot Karp deterministic algorithm plus repair heuristics yielded a solution of 112.33 on the 500 city problem. Random search plus repair heuristics did better than this with a best of 108.91. It would appear that Karp's deterministic rule for deciding the direction of bisection becomes less effective as more repair heuristics are added. Fortunately, the same does not seem to be true of the *EDAC* algorithm.

**Table 1** Comparing 10000 random *Karp-like solutions + repair heuristics* with *EDAC + repair heuristics*.

| *Problem size* | *Mean* | *Standard Deviation* | *Best* | *EDAC* |
|---|---|---|---|---|
| 500 | 118.87 | 4.66 | 108.91 | 105.34 |
| 1000 | 120.15 | 4.35 | 110.66 | 104.65 |
| 2000 | 120.12 | 4.16 | 112.20 | 104.66 |

## 9. Conclusions.

Evolutionary divide and conquer offers a new approach for genetic algorithms applied to hard combinatoric search. We have applied this idea to the geometrical TSP and shown it to be viable if not yet practical. The genotype represents a division of the original problem into subproblems and the process of constructing a phenotype (tour) from the genotype is analogous to the growth of an individual. To meet the goal of an algorithm with good scaling it is necessary that this growth process scales at $O(n)$ or, at worst, $O(n(\log n)^{\alpha})$ for some $\alpha > 0$. Since the standard combinatorial local repair heuristics scale at $O(n^2)$ or worse, to satisfy this requirement for an acceptable tour quality we have been obliged to develop *geometrically local* repair heuristics one of which, *Far-repair*, is presented here. We feel confident that the overall accuracy can be improved by a more sophisticated combination of geometrically local heuristics, and we have a number of promising approaches yet to be explored. In addition we expect modifications to our genetic algorithm, currently a very simple but non-standard form, will also yield some improvements.

Once the model is refined, an obvious direction for further work is to parallelise the *EDAC* algorithm. It is clear that the overall design lends itself to parallelisation at several levels and in a number of different ways depending upon the parallel architecture. We plan to explore these possibilities when algorithm refinement is complete.

*Added in revision.*

Subsequent experiments reveal that using repair combinations of *Recursive-Fast-2-repair* and an *Enhanced-Far-repair* the *EDACII* algorithm consistently produces solutions at the 1% level. For example for a 2000 city problem we obtained a solution of 100.00 and in other large problems solutions with a *negative* excess. For this variation of the algorithm the scaling is preserved (the least squares line has slope 1.014), see Figure 12 and Table 4 for preliminary results. However, the overall run times are approximately 5 times higher.

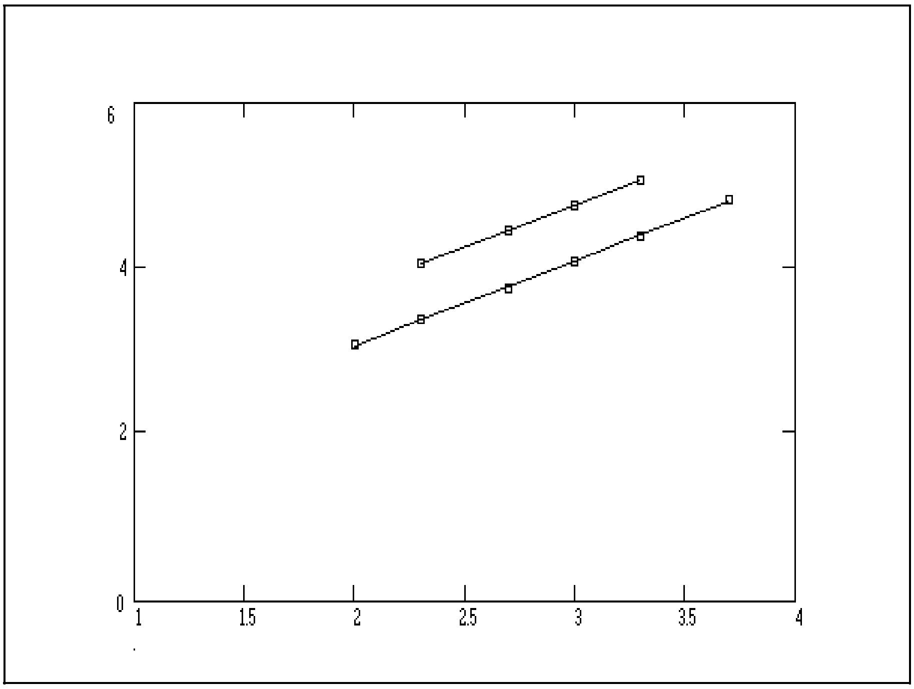


**Figure 12** Comparative scaling plots for *EDACII* (top) and the previous results (bottom). The horizontal axis is log cities, and the vertical axis log cpu secs.

We are thus moving into a phase where we need more accurate estimates of the optimum tour length and have implemented variations of the Held-Karp lower bound. Similarly, it would be extremely useful to have a more accurate estimate of Stein's constant. The difficulty in approaching this empirically is that so few exact solutions are known for very large problems of the right type (uniformly random distributions of cities).

The *Enhanced-Far-repair* heuristics in version *EDACII* attempt to gain improvements by moving very small groups, as well as individual cities. Despite their obvious success in buying an improved solution quality, we have recently come to consider our recipes of global *Far-repair* combinations to be the least elegant part of the implementation.

The idea behind *Recursive-Fast-2-repair* is to exploit the recursive structure of a Karp-like tour, and so limit the combinatorial growth of *2-moves* when *Fast-2-repair* is called. This seems to us more in keeping with the divide-and-conquer paradigm. Moreover, this idea is capable of generalisation in the sense that it can be applied to any combinatorial repair heuristic. With this in mind we are now considering more powerful recursive repair mechanisms. Hopefully, the mix-and-match combinations of various types of *2-repair* and *3-*

*repair* can then be discarded and the resulting algorithm will be more accurate and less time consuming.

**Acknowledgements.**

We should like to thank Heinz W. Muhlenbein of GMD for first stimulating our interest in parallel algorithms for very large TSP problems, and Martina Gorges-Schleuter for her continued encouragement and many illuminating discussions. We are also grateful for the helpful comments of the referees.

**Figure 13**. The 200-generation *EDAC* 5% excess solution for a 5000 city problem.

## References

[Beardwood 1959] J. Beardwood, J. H. Halton and J. M. Hammersley. *The shortest path through many points.* Proc. Cambridge Philos. Soc. **55**:299-327, 1959.

[Davis 1985] L. Davis. *Job shop scheduling with genetic algorithms*. Proceedings of an International Conference on Genetic Algorithms and Their Applications, pp136-140. Ed. J. Grefenstette, Lawrence Erlbaum Associates, Hillsdale, NJ, 1985.

[Goldberg 1985] D. E. Goldberg and R. Lingle. *Alleles, loci, and the traveling salesman problem.* Proceedings of an International Conference on Genetic Algorithms and Their Applications, pp154-159. Ed. J. Grefenstette, Lawrence Erlbaum Associates, Hillsdale, NJ, 1985.

[Gorges-Schleuter 1990] Martina Gorges-Schleuter. *Genetic Algorithms and Population Structures, A Massively Parallel Algorithm*. Ph.D. Thesis, Department of Computer Science, University of Dortmund, Germany. August 1990.

[Grefenstette 1989]. John J. Grefenstette and James E. Baker. *How Genetic Algorithms Work: A critical look at Implicit Parallelism.* Proceedings of the Third International Conference on Genetic Algorithms, pp 20-27, San Mateo, CA, Ed. J. D. Schaffer, Morgan Kaufmann, 1989.

[Grötschel 1991] M. Grötschel and O. Holland. *Solutions of large-scale symmetric traveling salesman problems*. Mathematical Programming **51**:141-202. North Holland, 1991.

[Held 1970] M. Held and R. M. Karp. *The travelling salesman problem and minimum spanning trees*. Oper. Res. **18**:1138-1162, 1970.

[Held 1971] M. Held and R. M. Karp. *The travelling salesman problem and minimum spanning trees: part II*. Math. Programming **1**:6-25, 1971.

[Karp 1977] R. M. Karp. *Probabilistic analysis of partitioning algorithm for the Travelling-Salesman Problem in the plane*. Mathematics of Operations Research, **2**(3):209-224, August 1977.

[Lawler 1985] Ed. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. *The travelling salesman problem - A guided tour of combinatoric optimisation*. John Wiley & Sons, 1985.

[Lin 1973] S. Lin and B. W. Kernighan. *An effective heuristic algorithm for the travelling salesman problem.* Oper. Res. **21**:498-516, 1973.

[Martin 1992] O. Martin, S. W. Otto, and E. W. Felten. *Large-Step Markov chains for the TSP in cooperating local search heuristics.* Operations Research Letters, **11**(4):219-224, 1992.

[Papadimitriou 1982] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimisation: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, N.J., 1982.

[Smith 1985] D. Smith. *Bin packing with adaptive search*. Proceedings of an International Conference on Genetic Algorithms and Their Applications, pp202-206. Ed. J. Grefenstette, Lawrence Erlbaum Associates, Hillsdale, NJ, 1985.

[Steele 1986] J. M. Steele. *Probabilistic algorithm for the directed Travelling Salesman Problem.* Mathematics of Operations Research, **11**(2):343-350, May 1986.

[Stein 1977] D. Stein. *Scheduling Dial-a-Ride transportation systems: an asymptotic approach.* Ph.D. thesis, Harvard University, Cambridge, MA. 1977.

[Whitley 1989] Darrell Whitley. *Scheduling Problems and Travelling Salesman: The Genetic Edge Recombination Operator*. Proceedings of the Third International Conference on Genetic Algorithms, pp 133-140, San Mateo, CA, Ed. J. D. Schaffer, Morgan Kaufmann, 1989.

## Appendix.

**Table 2** EDAC with *Recursive-Fast-2-repair* (Sparc 10) - single runs.

| n | log n | p | time (secs) | log time | best |
|---|---|---|---|---|---|
| 100 | 2 | 40 | 706.8 | 2.8493 | 101.07 |
| 200 | 2.3010 | 40 | 1504 | 3.1772 | 109.47 |
| 500 | 2.6990 | 40 | 3718.5 | 3.5704 | 108.39 |
| 1000 | 3 | 60 | 8992.9 | 3.9539 | 108.75 |
| 2000 | 3.3010 | 80 | 16908.4 | 4.2281 | 109.24 |
| 5000 | 3.6990 | 80 | 46825.2 | 4.6705 | 110.35 |

**Table 3** *EDAC* with *Recursive-Fast-2-repair + Far-repair* (Sparc 10) - average of 5 runs.

| n | log n | p | time (sec) | log time | best |
|---|---|---|---|---|---|
| 100 | 2 | 40 | 1085.298 | 3.0355 | 99.27 |
| 200 | 2.3010 | 40 | 2205.258 | 3.3435 | 105.41 |
| 500 | 2.6990 | 40 | 5473.458 | 3.7383 | 105.34 |
| 1000 | 3 | 60 | 11679 | 4.0674 | 104.65 |
| 2000* | 3.3010 | 80 | 23444.4 | 4.3700 | 104.66 |
| 5000* | 3.6990 | 80 | 65012.4 | 4.8129 | 105.90 |

* denotes average of 2 runs.

**Table 4** *EDACII* with *Recursive-Fast-2-repair* + *Enhanced-Far-repair* (Sparc 10) - average of 4 runs.

| *n* | *log n* | *p* | *time (sec)* | *log time* | *best* |
|---|---|---|---|---|---|
| 200 | 2.3010 | 40 | 11089.9 | 4.04493 | 101.81 |
| 500 | 2.6990 | 40 | 28013.2 | 4.4474 | 100.71 |
| 1000 | 3 | 80 | 57071.5 | 4.75642 | 99.62 |
| 2000* | 3.3010 | 80 | 114127.2 | 5.05739 | 100.00 |

* single run.