A Global Approach to the n-Dimensional Traveling Salesman Problem: Application to the Optimization of Crystallographic Data Collection

BY JEFFREY B. WEINRACH AND DENNIS W. BENNETT*
Department of Chemistry, University of Wisconsin-Milwaukee, Milwaukee, Wisconsin 53201, USA

(Received 13 April 1987; accepted 24 July 1987)

Abstract
An algorithm for the optimization of data collection time has been written and a subsequent computer program tested for diffractometer systems. The program, which utilizes a global statistical approach to the traveling salesman problem, yields reasonable solutions in a relatively short time. The algorithm has been successful in treating very large data sets (up to 4000 points) in three dimensions with subsequent time savings of ca. 30%.

Introduction
The programmed movement of four-circle diffractometers is an inherently inefficient process as performed on most diffractometer systems. Improvement in the procedure involves sorting reflections in such a manner that the time required to collect data is minimized. This is essentially a problem of finding the shortest path in \{2θ, Χ, φ\} space, a tangible representative of a class collectively known as 'the traveling salesman problem' (Flood, 1956).

The traveling salesman problem (TSP) is a classic example of a branch of mathematics dealing with combinatorial optimization; that is, minimization of a function over non-continuum conditions for which differential calculus cannot be applied. The basic premise of the TSP is to span an array of points in n-space with the shortest path possible. For most applications the system under study has been two-dimensional and the determined path has been a closed one (the end point along the path is also the starting point). Many approaches have been developed to solve such constrained problems, with moderate success. Recently a review of the TSP and the approaches taken to solve specific cases have been published in a comprehensive book by Lawler, Lenstra, Rinnooy Kan & Shmoys (1985). For the most part the problems addressed in this book consist of a few hundred points arranged in a two-dimensional array. Approaches to the solutions of these problems fall into two general categories, one involving decisions based on the immediate environment of points along the path, a 'local' approach, and one in which decisions are made on the set of points as a whole, a 'global' approach.

In the limit the local strategy becomes one of connecting each point to its closest neighbor, providing an obviously poor solution. Extensions beyond this limit approach a function of \(n!\) in difficulty, where \(n\) is the number of independent neighbors included in a given local search, and thus expand quickly to formidable proportions. In addition, the c.p.u. time required for local approaches increases approximately as a function of \(n(n-1)\), making the algorithms themselves unreasonably inefficient for large data sets. On the other hand, global approaches exhibit their own difficulties, principally in developing an algorithm which will treat a large number of randomly distributed points simultaneously. In general, most such strategies only begin globally, and are 'localized' as soon as possible in order to render the problem tractable (Grotschel & Padberg, 1978). Karp's 'divide and conquer' algorithm (Karp, 1986) provides an elegant example. It begins by partitioning the space into rectangular regions, then generates optimal local tours within each region which are subsequently collected into a final path. Unfortunately, the efficiency of such algorithms depends critically on the way in which the space is divided, since a large number of points in a subdivision simply creates a local traveling salesman problem, whereas too few points in each region yield an approach to the 'closest neighbor' solution. Thus, neither local strategies nor those which begin globally have proven to be both practical and general. Furthermore, none of the approaches to date extrapolates readily to three or more dimensions, a point of obvious concern to the practicing crystallographer.

In this paper we describe a new algorithm which obtains a short-path solution to the unconstrained traveling salesman problem, characterized by a random array of points in n-dimensional space with no...
restrictions on the beginning and end points of the path. The algorithm readily handles thousands of points, is completely general, and demands only very modest computational time and mass storage.

Theory

Of the two approaches to a general solution of the TSP, the global approach is in principle superior to the local approach, since the solution must depend on the complete distribution of points. Computers, however, make binary decisions, and lend themselves naturally to the local approach, which considers only one neighbor at a time. It follows from this that a workable algorithm should consider the ensemble of points collectively, but instead of building a path point by point, the algorithm should consider the ensemble as it makes individual binary decisions. This can be accomplished by sequential division and sorting of the ensemble into smaller and smaller subgroups, until the final set is reached which consists of an ordered set of groups each containing one point, a path. Since the procedure involves the sequential division and sorting of an array of points the algorithm can be designed to be recursive, that is, it can operate on itself. In this manner the procedure becomes increasingly more efficient as it progresses towards a solution. The description which follows is restricted to three dimensions, \( \theta \), \( \chi \) and \( \phi \), with the added constraint that only one of the three independent variables will determine a given decision. Thus the algorithm described herein is specifically designed for the optimization of crystallographic data collection in which all angles are traversed simultaneously and independently until the final angle is reached. Nevertheless, modifications to the procedure for other problems should be completely straightforward.

The array to be sorted consists of a complete set of angular coordinates, scaled if necessary to take into account any difference in relative motor speeds. The array is then partitioned into two subsets by sorting it into lower and higher groups with respect to a selected variable; for example, group I might contain the smaller \( 2\theta \)’s, group II the larger \( 2\theta \)’s. A simple insertion sort is used for the first half of the original array with all these points becoming members of the lower group. At this point the largest value in this group is the critical value. Any point with a value larger than the critical value becomes a member of the higher group. If an insertion into the lower group needs to be made, the element of the smaller group with the critical value becomes an element of the higher group and the next highest value then becomes the critical value. This saves considerable time, since many elements can be placed directly into the higher group, thus circumventing the insertion sort for these points. Each group contains exactly half of the data points unless the total number of elements is odd, in which case one group will contain an extra element. It is important to mention this since the algorithm involves the switching of array elements which do not match in the case of even/odd parity. This makes it necessary to carry the starting and ending indices of all subgroups in an auxiliary array throughout the sorting process. Each division defines two subsets, one of which will be spanned completely in the sort sequence before entrance into the other subset along the path. After the initial division (which requires only an arbitrary decision regarding the ordering of the two initial subsets), the entire decision-making process simply involves the choice of which of two subsets to span first.

Following the first division another variable is selected and group I is divided into two new groups (e.g. \( Ia \) and \( Ib \)) with respect to this variable. At this juncture the algorithm must determine the order of these two subgroups. It does this by computing the mean position of the points in groups \( Ia \), \( Ib \), and \( II \), then determining whether the mean position in group \( Ia \) or \( Ib \) is ‘closer’ to the mean position in group \( II \). In the specific application considered here ‘closer’ and ‘distance’ take on special meanings. Because only a single variable (\( 2\theta \), \( \chi \) or \( \phi \)) is rate limiting, the maximum absolute value of the difference between the mean \( 2\theta \)’s, the mean \( \chi \)’s and the mean \( \phi \)’s is the actual criterion for deciding which group is ‘closer’ to another. The group closer to group II becomes the second of the two to be spanned, since group II follows group I in sequence. Group II is then divided with respect to the same variable, with mean positions for subgroups \( Ila \) and \( IIb \) calculated and distances to the ‘closer’ subgroup from group I used to determine the order of groups \( Ila \) and \( IIb \). This process is continued, selecting a new variable each time the division process has covered the entire ensemble, until each subgroup contains one and only one point. The procedure becomes a bit more complex when considering non-terminal subgroups, i.e. all those not at the beginning or end of the array. In these cases the mean positions of the groups both preceding and following the divided group must be evaluated in the decision. Consider, for example, groups IV, \( Va \), \( Vb \), and VI. The IV, \( Va \), \( Vb \), VI and IV, \( Vb \), \( Va \), VI distances are computed with the shorter of the two paths determining the sort order. The algorithm thus contains sections to handle the first group, the middle groups, and the final group in the array for each cycle through the entire ensemble.

The algorithm

In order to provide as much generality as possible the algorithm outlined below is written in symbolic logic using symbols which are easily transformed into appropriate computer code.
List of symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NV</td>
<td>Number of independent variables (NV = 3 for 2θ, χ, φ)</td>
</tr>
<tr>
<td>V(I)</td>
<td>Variable I (I = 1, 2, 3 for 2θ, χ, φ)</td>
</tr>
<tr>
<td>N</td>
<td>Group index (in the limit N is the reflection sequence number)</td>
</tr>
<tr>
<td>G(N)</td>
<td>Group with sequence number N in array</td>
</tr>
<tr>
<td>RG</td>
<td>Renumber groups: (G(1), G(2), ..., G(N), G(N'), G(N + 1), ...) → (G(1), G(2), ..., G(N), G(N + 1), G(N + 2), ...)</td>
</tr>
<tr>
<td>G(N) ↔ G(N')</td>
<td>Switch contents of group N and group N' (effectively switches order of groups)</td>
</tr>
<tr>
<td>S(N) → V(I) → G(N) + G(N')</td>
<td>Sort group N into two new groups with respect to variable V(I), N &lt; N'</td>
</tr>
<tr>
<td>M(N)</td>
<td>Mean position of points in group N</td>
</tr>
<tr>
<td>#(N)</td>
<td>Number of points in group N†</td>
</tr>
<tr>
<td>N_i</td>
<td>Total number of points in n space (2θ, χ, φ) space</td>
</tr>
<tr>
<td>N_c</td>
<td>Current group index</td>
</tr>
<tr>
<td>N_t</td>
<td>Total number of current groups with different indices (number of indexed groups)</td>
</tr>
<tr>
<td>MAB</td>
<td>Maximum absolute value</td>
</tr>
<tr>
<td>TCP{V(1), V(2), V(3)}</td>
<td>Array containing diffractometer angles to be sorted</td>
</tr>
<tr>
<td></td>
<td>Comment</td>
</tr>
</tbody>
</table>

Traveling salesman algorithm

* Initial partitioning
* Set up array to be sorted (TCP)

* NV = 3
* N_i = 1
* I = 1
* S(1) → V(I) → G(1) + G(1')
* N_i = 2
* I = 2
* S(1) → V(I) → G(2) + G(2)
* IF MAB[M(2') - M(1')] > MAB[M(2) - M(1')] THEN G(2') ↔ G(2')
* G(1) = G(2)

* First group algorithm

1 I = I + 1
* IF I > NV THEN I = 1
* N_c = 1
* IF #(1) = 1 THEN GO TO 2
* S(1) → V(I) → G(1) + G(1')
* IF MAB[M(1') - M(2)] > MAB[M(1) - M(2)] THEN G(1) ↔ G(1')
* N_i = N_i + 1
* RG
* N_c = N_c + 1

* Middle group algorithm

2 N_c = N_c + 1
* IF N_c = N_t THEN GO TO 3
* IF #(N_c) = 1 THEN GO TO 2
* S(N_c) → V(I) → G(N_c) + G(N_c')
* IF MAB[M(N_c - 1) - M(N_c)] + MAB[M(N_c - 1) - M(N_c')]
  > MAB[M(N_c + 1) - M(N_c)] THEN G(N_c) ↔ G(N_c')
* N_i = N_i + 1
* N_c = N_c + 1
* RG
* Value of variables in Fig. 1 determined here
* GO TO 2

* Last group algorithm

3 IF N_c = N_t THEN END
* Each group contains one point
* IF #(N_c) = 1 THEN GO TO 1
* S(N_c) → V(I) → G(N_c) + G(N_c')
* IF MAB[M(N_c) - M(N_c)] > MAB[M(N_c - 1) - M(N_c')]
  THEN G(N_c) ↔ G(N_c')
* N_i = N_i + 1
* N_c = N_c + 1
* RG
* GO TO 1

Results and discussion

Fig. 1 illustrates a simple worked example in two dimensions to assist the programmer in developing this algorithm for his own particular system. In order to test the efficiency of the algorithm a number of similar two-dimensional paths were generated and compared with the actual shortest routes, determined
by calculating all possible paths. In general only very small sets of points can be treated in this manner, since the time required to obtain 'brute-force' solutions for \( n \) points increases as a function of \( n! \). In every case the path produced by the algorithm was one of the shortest \( 0.03\% \) of the paths, and always differed negligibly from the optimal solution, in spite of requiring only a minuscule fraction of the time necessary for the corresponding brute-force solutions. For example, a ten-point solution such as that shown in Fig. 1 typically requires less than 1 c.p.u. second compared with 1800 c.p.u. seconds needed to determine the optimal path by brute force.

Fig. 2 shows the results obtained from a test program designed to find a short-path solution for a general two-dimensional array of randomly distributed points. A study of this path reveals some of the characteristics of the solution obtained from the algorithm. In general, the algorithm tends to migrate from regions of low point density towards regions of high point density, thus creating a path in which minimal time is spent in regions where there are few points. The circled areas are places where crossover has occurred along the path, resulting from the arbitrary nature of the space-partitioning process in the algorithm. These areas are always small, both in size and number, but they limit the efficiency of the algorithm. This particular problem arises when a group division separates points that are close to one another. In order to keep the procedure simple and general we have made no attempts at optimization, but the crossover problem can conceivably be resolved by treating the points as 'vibrating' rigid bodies in \( n \)-space, thus allowing them to overlap. The space can then be partitioned such that overlapping points are placed in the same group. It is also probable that the procedure can be improved by taking advantage of the specific nature of the variables involved in formulating constraints for particular cases. For crystallography such information might include Laue-group/space-
Table 1. Efficiency of algorithm

<table>
<thead>
<tr>
<th>Number of reflections</th>
<th>Time before sort</th>
<th>Time after sort</th>
<th>Time saved ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1347</td>
<td>28 401.48</td>
<td>23 657.17</td>
<td>0.167</td>
</tr>
<tr>
<td>785</td>
<td>17 466.14</td>
<td>10 875.75</td>
<td>0.377</td>
</tr>
<tr>
<td>2565</td>
<td>24 550.33</td>
<td>18 794.41</td>
<td>0.234</td>
</tr>
<tr>
<td>2794</td>
<td>25 019.20</td>
<td>17 465.88</td>
<td>0.302</td>
</tr>
</tbody>
</table>

The time is measured in arbitrary units; time saved ratio = (time before − time after)/time before.

As illustrated in Table 1, the application of this algorithm to crystallographic data sets results in substantial savings in data collection time. The ratio of 'time saved' to total time before sorting indicates savings ranging from 15 to 37%. Since the unsorted data sets approximate 'closest-neighbor' pathways, the algorithm clearly provides significantly shorter tours. It should be noted that these comparisons are relative since they also depend on the path calculated prior to application of the algorithm. The results in the table were generated on an IBM-PC with a program compiled in Microsoft Quickbasic®. The selection of the sorting routine is critical to the computational efficiency of the procedure. The program described here makes use of an insertion sort routine, since the algorithm only requires that a minimum of half an array be sorted during any one operation. A bubble sort would be extremely inefficient in this application, while there may be other more sophisticated routines which are more efficient (Amsterdam, 1985). As with the path determination, we have made no attempt to optimize the program itself. On the IBM-PC, a 3000-point pathway typically requires about 5 c.p.u. h, while the savings in data collection can amount to numbers of days. This serves to establish something of an upper limit which can readily be decreased with faster computers and improvements in code.

DWB would like to acknowledge Dr Ward T. Robinson at the University of Canterbury for initial discussions in the area of data collection time optimization.

References