An examination of heuristic algorithms for the Travelling Salesman problem.

Barbara Katja Höck

Thesis submitted towards the degree of
Master of Science in Operations Research,
in the Department of Mathematical Statistics,
University of Cape Town, March 1988.

Supervisor: Prof. T. Stewart

Acknowledgements.
I would like to acknowledge the help and supervision of Prof. T. Stewart, the C.S.I.R. for funding one year of the research, T.W. Payn of Saasveld Forestry Research Centre for providing the information for the case study, Dr F.J. Kruger (Director of the SA Forestry Research Institute) for permission to use forest survey data, and G.M. Höck for help in the final preparation of the thesis.

ABSTRACT

The role of heuristics in combinatorial optimization is discussed. Published heuristics for the Travelling Salesman Problem (TSP) were reviewed and morphological boxes were used to develop new heuristics for the TSP. New and published heuristics were programmed for symmetric TSPs where the triangle inequality holds, and were tested on micro computer. The best of the quickest heuristics was the furthest insertion heuristic, finding tours 3 to 9% above the best known solutions (2 minutes for 100 nodes). Better results were found by longer running heuristics, e.g. the cheapest angle heuristic (CCAO), 0-6% above best (80 minutes for 100 nodes). The savings heuristic found the best results overall, but took more than 2 hours to complete. Of the new heuristics, the MST path algorithm at times improved on the results of the furthest insertion heuristic while taking the same time as the CCAO. The study indicated that there is little likelihood of improving on present methods unless a fundamental new approach is discovered. Finally a case study using TSP heuristics to aid the planning of grid surveys was described.

TABLES

FIGURES

INTRODUCTION

An aspect of modern Operations Research is the problem of incorporating combinatorial optimization models into decision support systems, particularly on a micro computer. In practice this almost inevitably has to be done by means of heuristic rather than exact procedures, to cope with size and speed constraints and requirements.

The "Travelling Salesman Problems" (TSP), a class of combinatorial problems, typifies in many ways the problems arising in combinatorial optimization. For this reason, it is a fairly well studied problem, though there is a lack of comprehensive empirical surveys. Reported computational studies have primarily been in the context of relatively powerful mainframe computers, and it is not clear as to the extent in which the published solution methods can be used in the micro computer environment. It is, in any case, necessary from time to time to update comparisons in the light of newer methodological developments.

An outline of the contents of the thesis follows.

Combinatorial optimization problems and their complexity are discussed. The class of NP-complete problems is introduced, which consists of some of the hardest of the combinatorial problems. Heuristics are defined as, among other definitions, methods which find good solutions to difficult problems. The role of heuristics in combinatorial optimization is described. Aspects of designing heuristics are discussed, including the use of morphological

boxes for discovering new algorithms.

An important class of combinatorial optimization problems is introduced: the Travelling Salesman Problem. Some special properties and variations of the standard problem are given, as are several applications of the TSP. Exact solution methods for the TSP exist and are briefly described, but they all show an exponential increase in computational effort with respect to the size of the problem. Heuristics are the alternative solution techniques.

In Chapter 3, the published heuristics for the Travelling Salesman Problem are reviewed. The algorithms are grouped according to whether they are tour construction or tour improvement heuristics. Tour construction techniques are further differentiated according to the type of construction approach used. The computational complexity of the heuristics and, where known, their worst possible result is included. Some published comparisons of empirical tests on the TSP heuristics are discussed.

A critical evaluation of the existing algorithms is made, in order to establish whether other potential, as yet untried approaches exist. Several features of the algorithms are listed, together with the possible variations of each. Each TSP heuristic is characterized by its particular features. The important features are selected and cross-classified, to form morphological boxes. Each heuristic is entered into its box. Several new algorithms are developed to fill the empty morphological boxes. Ways of evaluating heuristics are discussed, with the performance of

empirical tests being chosen for this study.
Several of the published heuristics and all of the new heuristics are selected for the tests. The results of the tests, the percentages that the heuristic solutions are above the best known solutions, are tabulated in order of increasing running times of the heuristics.

The TSP heuristics are run on micro computer, on several published problems, as well as on several randomly generated problems. The test results are discussed, and the best heuristics are selected of the quickest techniques, of the slightly longer running, better techniques, and of the long, intensive search methods. The results for the new heuristics are also discussed.

Finally, a case study of a real problem is described. TSP heuristics are used as aids when planning the execution of grid surveys of large, inaccessible areas. To minimize effort, routes are chosen to minimize the amount of walking necessary and, to a lesser extent, the amount of driving required. Three programs are used; the first minimizes the walking distance, thereby determining the parking points. A TSP heuristic is used to sequence the parking points. The final program suggests the roads to be taken to visit the parking points in order, allowing interactive modifications to the suggested tour.

The emphasis of this study is not only on performing numerical experiments to compare existing heuristics, but also on evaluating these heuristics to see if there is scope for improving on them by discovering new heuristics. Our findings in this respect are discussed in the conclusion.

2.1 Combinatorial optimization

2.1.1 Combinatorial optimization problems

Where there is a problem, there is a quest for its solution.

> "The search for optimal solutions ... is a reflection of the philosophy of economic man whose aspirations are to derive the maximum utility in a given set of circumstances, by maximizing a measure of output for given inputs, or by minimizing the inputs required to attain a given output, or by optimizing some other criterion that relates outputs to inputs."
> (Eilon, 1977)

Even when a situation is not problematic, an improvement is perhaps possible. For example, an increase in the productivity of a factory may be possible, with or without this being an economic necessity.

One approach to problem solving is to take quantitative measurements and to use scientific, in particular mathematical techniques, as is done in Operations Research. For this approach it is necessary to formulate the situation in terms of mathematical notation and terminology. A formal model is proposed and then attempts are made to solve this representation of the original problem.

For many problems, the alternatives on which decisions must be made form a finite set of discrete elements. A "solution" to a problem with this property, is a combination, grouping or selection of elements from the set according to some criteria; the "decision" for each alternative is whether or not it is included in this combination. This may perhaps be best illustrated by a few examples.

(1) Set Covering.

A connection (i,j) between two elements i and j of a set of discrete objects is said to "cover" these two elements. Find the connections such that every element in the set is covered.

(2) Spanning Tree.

Find the connections such that every element in the set is covered (as for Set Covering), and in addition any two elements of the set are linked by a sequence of connections.

Problems such as these are known as "combinatorial" problems. Usually an arrangement or combination of elements which is best by some criterion, i.e. the optimal solution, is required. For the above examples, let each connection between two elements have an associated cost. Then the following questions may be asked.

(1) The Set Covering Problem:

What is the cheapest way of covering each element in the set?

(2) The Minimum Spanning Tree Problem:

What is the spanning tree of minimum cost?

One application of this problem is computer wiring, when each electric component must be connected to at least one other component. The cost of a connection is the length of wire needed between two components.

"Combinatorial optimization" refers to the finding of the solution which yields a minimum, as in the above examples, or a maximum value for the specified criterion. The challenge of combinatorial optimization is the development of fast and efficient solution methods to achieve this.

2.1.2 Computational complexity

An "algorithm" is a procedure, stated in mathematical or computer terms, which is used to solve a problem. Both the Set Covering Problem and the Minimum Spanning Tree Problem of the previous section have had algorithms for their solutions published. (For example in Lawler, 1976).

The time that an algorithm takes to solve a problem can be stated as a function of some measure of the "size" of the problem. For example, if a problem solution requires a specific combination of n discrete elements and there exists an algorithm

that solves this problem in time proportional to n^2, then the running time of the algorithm is said to be of

$$O(n^2).$$

Another algorithm for the same problem may have a running time of

$$O(2^n).$$

If the running time of an algorithm is a polynomial function of n, it is said to run in polynomial time. An algorithm whose running time is a non-polynomial function of n, such as the function n(log n), but which is bounded by some polynomial in n, is also called a polynomial time algorithm. If running times involve a term to the power of n, however, the algorithm is said to be an exponential time algorithm.

Algorithms that run in polynomial time terminate more quickly than those running in exponential time when problems are large enough. This is illustrated in Table 2.1: while the exponential time algorithms run more quickly than the polynomial time algorithm for n=5, the latter rapidly becomes very much more efficient.

	a polynomial function in n	functions exponential in n	
n	n^2	2^{n-2}	$n\ 2^n / 200$
5	25	8	4
10	100	256	512
20	400	262 144	2 097 152
100	10 000	3.17×10^{29}	6.34×10^{31}
1000	1 000 000	2.68×10^{300}	5.36×10^{304}

Table 2.1 Functions of the problem size n, describing the running times of algorithms.

Even when mainframe computers are available for solving problems of a large size, it would be preferable to have algorithms which run in polynomial or even linear time. For the examples of Table 2.1, if the times are in nanoseconds, then for n=100 the exponential time algorithms require times in excess of the age of the earth! For many combinatorial problems, though, no polynomial time algorithms are known. A few, such as the Minimum Spanning Tree problem, have however had polynomial time algorithms proposed for them.

A problem which can be solved by an algorithm in polynomial time is said to be in the class P. A wider class of problems is the NP ("nondeterministic polynomial") class. This class can be characterized in various ways. Perhaps the simplest description is that a problem is in NP if there exists a polynomial time algorithm to determine whether a particular hypothesized solution is feasible and whether its value lies above or below some prescribed level. It is evident then that P C NP.

Another subclass within NP is the class of NP-complete problems. A problem is said to be NP-complete when the following properties

hold:

(1) The problem is in NP.

(2) If a deterministic polynomial-time algorithm exists to solve the problem, then this algorithm can be used to find deterministic polynomial-time algorithms for every problem in NP.

There is strong evidence that such a deterministic polynomial time algorithm as described in (2) can never exist for NP-complete problems (Reingold et al, 1977). Thus this class of problems consists of the hardest problems in NP.

As all known algorithms (in fact, it is postulated, all algorithms) for solving NP-complete problems exactly run in exponential time, alternative solution methods have been studied. One approach is the use of heuristics. These methods are not guaranteed to find the optimum solution of a problem but, if correctly designed, have shorter running times than the algorithms which are guaranteed to find this optimum.

2.2 Heuristics

2.2.1 Definition and features of heuristics

The term "heuristics" is derived from the Greek "heuriskein" which means "to discover" (Zanakis and Evans, 1981) or "guiding discovery" (Groner, Groner and Bischof, 1983). From this an informal meaning has developed, which is the study of the methods and rules of discovery (Polya, 1945), and a more frequently used formal meaning which sees heuristics "as criteria for reducing the search process in a large space of alternatives" (Tikhomirov, 1983, also Pospelov, Pushkin and Sodovskii, 1972).

In the field of artificial intelligence, similarly to Operations Research, these criteria have been described as rules of thumb (Pearl, 1984) and as methods which are intelligently directed but still have an inherent uncertainty (Groner et al, 1983, Reiter and Sherman, 1965). In Operations Research itself heuristics are seen as methods which use common sense to find good, though not necessarily optimal solutions to difficult problems (Zanakis and Evans, 1981, Muller-Merbach, 1973), or as methods producing acceptable solutions within limited computing time (Lin, 1975).

Heuristics are thus unlike exact solution methods which guarantee to find, or to come arbitrarily close to finding, the solution of a problem. In fact, any problem solving technique could be termed a heuristic until it can be proved that this technique will always converge to the required solution, when it becomes an exact method.

It is possible to describe heuristics as used in Operations Research in terms of their algorithms. Firstly let us define a few terms, including the functioning of exact methods, and then let us show how heuristics compare to such methods.

An algorithm is said to be iterative if it, or a part of it, is repeated several times during the search for a solution to the problem. At each repetition, the algorithm starts from a partial solution (in which decisions have not been specified for every alternative) or a full solution which has not yet been accepted as a final solution; makes some evaluations; and then, based on the results of these evaluations, updates or modifies the solution in some way. Finally a test is made to see whether the result of the update or modification should become the starting point of the next iteration or not. If not, then the algorithm has either found a solution to the problem or indicated that it

has failed to find a solution. Combinatorial optimization problems are usually solved by iterative algorithms, and we will deal only with these types of solution methods.

For a combinatorial problem, let the finite number of discrete alternatives on which decisions must be made, be called "candidates". At each iteration of an algorithm, one or more candidates are added to the partial solution, or, when the starting solution of the iteration is a full solution, one or more candidates in the full solution are exchanged for candidates not in the solution. An algorithm can be characterized by:

(1) which candidates are chosen for evaluation (these will be called the "potential" candidates);
(2) what evaluations are made;
(3) how candidates are selected to be added to the partial solution (respectively included in the full solution): these will be called the "selected" candidates.

Exact algorithms form the set of potential and the set of selected candidates in such a way that no candidate which could lead to the optimal solution of the problem is excluded forever from further processing. In heuristic algorithms, however, it is possible for candidates which could be part of an optimal solution of the problem, not to be included in the set of selected candidates. A heuristic must therefore proceed carefully about the selections that it makes, both when forming the set of potential and the set of selected candidates. (Muller-Merbach, 1981)

Of the many possible solutions of a combinatorial problem, "good" solutions can be defined as all optimal solutions, plus other non-optimal solutions which are within a neighbourhood of the optimal solution by some criterion. A typical criterion may

be that the solution is within, say, 10% of the optimal solution. The term heuristic will be used for an algorithm which has been shown to lead typically to good solutions, but for which an optimal solution cannot be guaranteed. In some cases, by analyzing the worst results possible, it may even be possible to guarantee a good solution.

Certain characteristics are desirable in a heuristic. A good heuristic should, if possible, have such features as:

(1) Simplicity, that is simplicity of design and of approach, possible for the user to understand, preferably explainable in intuitive terms.

(2) Realistic storage and computing time requirements. Minimal computational growth, preferably low order polynomial or linear growth.

(3) Accuracy; close to optimum on average.

(4) Robustness, in the sense that the chance of a solution being far from optimality should be low.

(Foulds, 1983, Ignizio, 1980, Silver et al, 1980, Zanakis and Evans, 1981)

2.2.2 Reasons for using heuristics

When exact problem solving methods are available but are lengthy and cumbersome, requiring excessive amounts of computing time and storage for large sized problems, heuristics may be the only possible option.

There are several other reasons for using heuristics in operations research, and in particular in combinatorial optimization. (Fisher, 1980, Foulds, 1983, Muller-Merbach, 1981, Silver, Vidal and de Werra, 1980, and Zanakis and Evans, 1981)

(1) A problem may be of such a nature that reliable, exact methods are not available. This can arise when the problem has a very complicated structure, or is based on large quantities of differing information.

(2) In formulating a real world problem mathematically, simplifications and assumptions frequently have to be made. This is because such problems tend to be dynamic and unstructured, while their mathematical model assumes a static and predictable situation. In addition to aspects of a problem being ill-defined, problem data may be difficult or costly to collect, and so be prone to inaccuracies. Solving a problem to optimality in such situations may be of academic interest only, as a 'suboptimal' solution of a heuristic may in fact be seen as good enough by the decision maker.

(3) A manager or decision maker may be more favourably disposed to a method that is simpler to understand, thus increasing the chances that the solution to a problem by this method will be implemented.

(4) Many exact methods of combinatorial optimization themselves use heuristic techniques to speed up the search for the solution. Examples are the selection of pivots in linear programming, using a heuristic to find a good starting solution, or to provide 'not worse than' bounds in tree search techniques.

(5) Heuristics may be used on problems where the optimum solution could be found by standard techniques, for teaching purposes, similar to the use of simulation for gaining insight into the mechanisms of a problem.

Heuristics have not always been received favourably. Besides not guaranteeing to find the optimal solution to a problem, they frequently lack the mathematical elegance of sophisticated, exact methods (Muller-Merbach, 1974). Part of the reason for this 'lack of elegance' is that non exact methods have at times been not much more than computerized number crunching or computer aided guessing (Ignizio, 1980).

However, at times there are no options other than using a heuristic approach for some problem types.

2.2.3 Heuristics in combinatorial optimization

Heuristics have been published for many classes of combinatorial problems. A few surveys of heuristics for specific types of problems are given below.

Balas and Padberg (1976) published a comprehensive survey of heuristics for set partitioning problems such as the node covering, edge matching, node packing and set partitioning problems. Heuristics for the combinatorial problems of subset-sum, bin packing, maximum satisfiability, set covering, graph colouring and maximum clique have been reviewed by Johnson (1974). A more recent survey of the heuristics for one of these problems, the bin packing problem, is by Coffman, Garey and Johnson (1984).

Within the class of standard scheduling problems, heuristics for the flow shop scheduling problem have been published by Liesegang and Schirmer (1975), and Matthaus (1975) has surveyed several heuristics for the vehicle scheduling problem.

Other standard combinatorial problems include the assignment

problem, surveyed by Burkard (1979), and the assembly line balancing problem, reviewed by Kilbridge and Webster (1962). Surveys of heuristics for the travelling salesman class of problems are discussed in a later section (3.4).

Garey and Johnson (1976) published an extensive annotated bibliography of heuristics for combinatorial problems such as packing and storage allocation problems, scheduling problems, routing and placement problems, and graph problems.

2.2.4 The design of heuristics

The design of heuristics for problems requires an "ample amount of creativity and experience" (Muller-Merbach, 1974). As well as studying methods proposed in the literature, attempting own designs and experimenting with different approaches give the best insight into the design of heuristics.

When confronted with a problem for which a heuristic is to be formulated, it is sometimes possible to divide the problem into two or more subproblems, each of which is to be solved heuristically. The division is itself a heuristic approach. Choosing the method of decomposition must take into consideration the trade-off of effort between the two subproblems; making one of these too easy may adversely affect the computational requirements of the other. Approaching a problem situation from several different angles, either by the use of different heuristics or by starting one heuristic from different points, will increase the chance of coming close to the optimal solution (Hillier 1983).

Features that may be desirable to include in the design of heuristics for a problem are: making weaker demands on the

information of that problem; making the heuristics flexible; and allowing for the creative combination of methods comprising a heuristic (Fuller, 1978).

A heuristic can be designed to exploit the structure of a type of problem. Each heuristic may be most effective within a certain situation (Lenat 1983), although this is not necessarily desirable.

Once the task of a heuristic has been decided on, for example to find a tour in the Travelling Salesman Problem (see Section 2.3), further study can be made of the approach to be used to complete this task. Some guidelines to be used each time a new design is necessary have been proposed.

One can start by asking some of the questions below (adapted from methods to solve problems, Polya 1945, see also Newell, 1983):

- Has this problem been studied before? Perhaps in a different form?
- Is a related problem known?
- Could an auxiliary element help?
- Restate the problem.
- Solve a related problem:
  Was it possible to learn from the process? Was the related problem more accessible? general? special? analogous?
  Is it possible to solve a part of the problem? Or to keep only a part of a condition?
  Can the data be used somehow?
  Could changing the unknown help? Or changing the data, or both?
- Has all the data been used? Are all the conditions known? Are all the essential notions known?

Taking a slightly different approach to designing a heuristic, it is possible to work with simplified models of the original problem, where simplification occurs by (Pearl, 1984):

- removing constraints from the problem (relaxed model)
- adding constraints to the problem (overconstrained model)
- using prior knowledge of the most probable solution value to guide the search
- using analogical or metaphorical models, i.e. transforming the problem into another form for which expertisé already exists.

A more systematic approach than those suggested above is possible. Heuristic algorithms for a class of problems can generally be characterized in terms of a number of design features (ways in which different activities are carried out). One example of this is the selection of the three basic steps identified in section 2.2.1 as characterizing iterative algorithms, viz. choosing potential candidates, performing evaluations on them, and making the final selection of candidates.

In principle each feature can be chosen in many different ways, more or less independently of the others. To assist in the decision as to precisely what combination of possible features are to be included when designing a new algorithm, so called "morphological methods" can be used (Muller-Merbach, 1976 and 1981, Zwicky, 1968). These approaches have been used extensively in fields such as engineering design. The approach is to list the features on which design decisions have to be made. For each of these all possible variations are also listed. Then "morphological boxes" are constructed, giving a cross-tabulation of all possible combinations of the variations of each feature.

Table 2.2 shows a hypothetical example of such morphological boxes for a problem that requires finding connections between points on a graph, such that each point is connected in some way. One of the features of heuristics for this problem is the number of points included into the solution-in-process at each iteration, while another is the selection of which points to evaluate. For each of these features, there are at least two possible variations, resulting in the following cross-tabulation.

		NUMBER OF POINTS SELECTED	
		one only	several
WHICH POINTS ARE EVA-LUATED	all unconnected points		
	all unconnected points within distance d of the previously processed point		

Table 2.2 Example of morphological boxes

If a heuristic already exists for a problem type, it will of necessity fall into a box or boxes (depending on how the classification is done), and this can be entered into the relevant box or boxes. Design possibilities for new heuristics are indicated, at least implicitly, by empty morphological boxes.

2.3 The Travelling Salesman Problem

2.3.1 Definition of the problem

An important class of combinatorial optimization problem is termed the "Travelling Salesman Problem", or TSP. The name arises from the original casting of the problem in terms of a hypothetical "travelling salesman", who needs to visit all cities in his area once and only once, before returning to his home base. This he wishes to do with minimum cost (see Figure 2.1).

Although it is a problem that can be visualised easily, its solution can be difficult to find. It is this combination of 'simplicity of statement and difficulty of solution' that has led to many different problem solving approaches being attempted (Garfinkel, 1985).

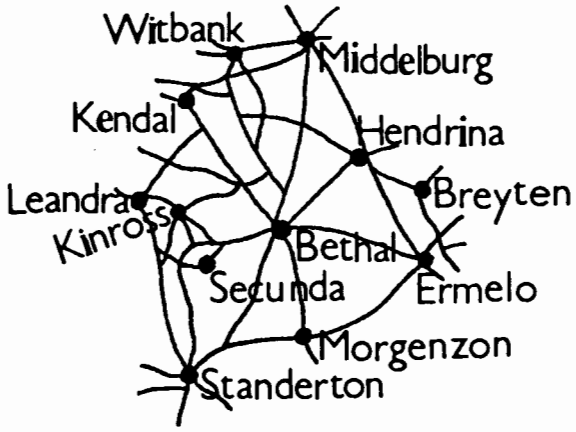


Fig. 2.1 Classic Travelling Salesman Problem with 11 towns to be visited.

Let us define the following.

node i : a compulsory destination, of which there are n, for example, the city that the travelling salesman must visit.

edge (i,j) : the connection between node i and node j, for example the road connecting the two cities.

cost c_{ij} : the cost incurred by including edge (i,j) into the solution of the problem. The cost may be the physical distance between two nodes (such as the road distance between two cities), the time taken to move from node i to node j, or any other cost incured in linking node i to node j. All connections are assumed possible. All costs are assumed known. Where no real cost exists, or where connections are not possible, dummy costs of infinity are used to fulfill this requirement.

The travelling salesman problem then requires finding a route (i.e. a sequence of edges), which:

- starts from a given node;
- visits all the other nodes, exactly once;
- returns to the starting node; and
- does this with minimum cost.

Such a route is called a tour. The TSP is thus one of finding a minimum cost tour for a given situation.

The structure of a Travelling Salesman Problem may satisfy certain special properties, which sometimes facilitate the solution:

(1) If $c_{ij} = c_{ji}$ for all i,j then the problem is symmetric.

Referring to the classical example, if the cost of travelling from one city to another is of importance, then travelling from a city at the bottom of a pass to one at the top of the pass may be more expensive than the trip in the other direction. This would then be an asymmetric problem. A less trivial example is given in the applications of the TSP (Section 2.3.2).

(2) The TSP satisfies the triangle-inequality if

$$c_{ik} \leq c_{ij} + c_{jk} \quad \text{for } i,j,k=1,\ldots,n$$

An example of the violation of this rule is the flight from Harare to London via Moscow with Aeroflot, which is cheaper than flying Harare-London direct with British Airways.

(3) The Euclidean TSP is a special case of a symmetric TSP for which the triangle-inequality holds. The position of a node in this problem is represented as coordinates in the 2-dimensional plane, and the cost of an edge between two nodes with coordinates (x_1, y_1) and (x_2, y_2) is the

Euclidean distance between them, i.e.

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Figure 2.2 shows some of the variations of the basic TSP structure. Note that if an algorithm solves one type of TSP, then it will also solve a more specific type of problem, where, for example, a symmetric TSP is more specific than an asymmetric TSP. This thesis will deal mainly with symmetric Travelling Salesman Problems, with or without the triangle-inequality.

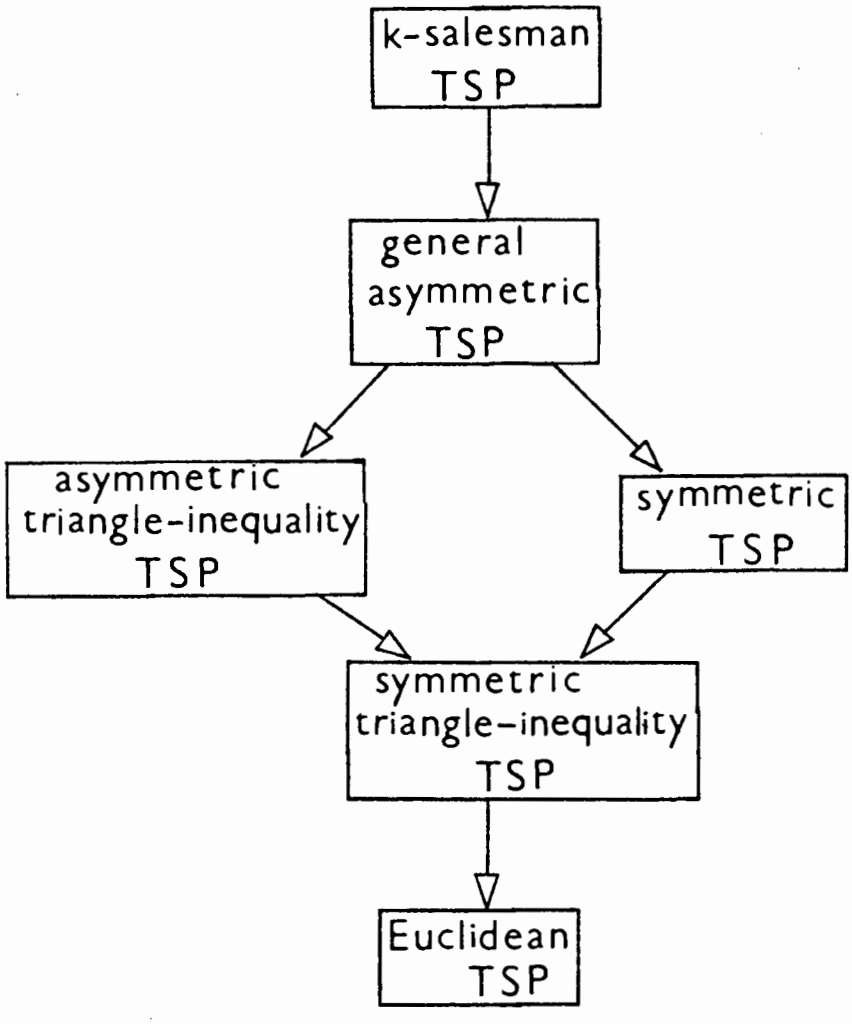


Fig. 2.2 Additional constraints made on the structure of the TSP

Other variations of the standard problem are possible. The problem of minimising the largest edge cost that is incurred in a tour (the bottleneck TSP) can be solved fairly easily (Hochbaum and Schmoys, 1986). A TSP may have the cost of going from city i to city j dependent on the time in which the travelling is done (Picard and Queyranne, 1978). The TSP may be subject to stochastic decision rules, where node j is visited after node i with probability p_{ij} (Derman and Klein, 1966). These last two problem types have no easily found solution.

Although there are many variations possible (see for example Lawler et al, 1985), the basic problem has itself many applications.

2.3.2 Some applications

Some of the earliest real applications of TSP algorithms include routing applications such as school-bus routing, and routing a job through assembly stations (Flood, 1956).

Since then, several seemingly unrelated problems have been formulated and solved as TSPs. The examples that follow are given by Lenstra and Rinnooy Kan (1975), Garfinkel (1985), and Telgen (1985).

An obvious example is the connecting of several points in such a way that the total distance covered is the minimum possible.

Computer wiring problems, which occur during the design of computer interfaces, are one of the many examples possible. In this problem, each interface consists of a number of modules and each module has a number of pins on it. Groups of pins must be connected, however each pin can have at most two wires attached to it. The aim is to minimize the total wire length necessary.

The wiring problem can be formulated as a symmetric TSP by letting nodes represent the pins on a module, and the cost between two nodes be the distance between them. If the wire is to follow a route which includes all pins, each at most once, but does not return to the starting pin, then by including a dummy pin which can be connected to any other at zero cost, this version of the problem reduces to the standard TSP form. In converting the TSP tour to a solution of the real problem, the two pins which are connected to the dummy pin form the start and end of the wire, and the dummy is discarded. End pins on one module can then be connected to those on other modules.

Several types of sequencing and scheduling problems have the same structure as a TSP. A straightforward sequencing problem requires the ordering of a number of jobs on a single machine, where the machine must be in a certain state for a specific job. Each change from job i to job j has an associated cost, such as the machine setup time. For example: each job requires a different colour to be painted onto an article. The setup time for changing from a dark paint to a light one may be longer than changing from light to dark.

We represent the beginning and ending state of the machine by a dummy node, each job by a node, and denote the setup time necessary when changing from job i to job j as the cost c_{ij}.

The costs may be symmetric or asymmetric. The problem can now be solved as a TSP.

There are many examples of simple sequencing problems. One such is the wallpaper cutting problem, where several sheets must be cut from a roll of patterned wallpaper. Each sheet starts and ends at a specific point in the pattern. Wastage is to be minimised. To formulate this as a TSP, let a node represent a sheet, and let the cost between node i and node j represent the amount of wallpaper wasted if sheet i is followed by sheet j.

Another problem is the scheduling of meetings between a manager and employees. The manager needs to discuss n projects where each project has a different group of people assigned to it, but wants to keep to a minimum the number of people entering and leaving the office between discussions. Let a node denote a project, and let the cost between nodes i and j denote the number of people going in or out of the office when changing from project i to project j. The problem can then be solved as a TSP.

A more complicated type of problem is the machine scheduling problem where little or no intermediate storage is allowed between processing on one machine and the next. This situation occurs for computer buffers, and for aluminium rolling where high temperatures must be kept throughout. TSP approaches have been used for these type of problems. Lenstra and Rinooy Kan (1975) show how this situation can be formulated as an asymmetric TSP.

Classical vehicle routing problems can be solved using the TSP approach. The vehicle routing problem is to determine for a fleet of vehicles, which customer should be served by which vehicle, and in what order each vehicle should visit its customers. One

approach is to construct one tour of all the customers, and then to partition this tour such that any additional constraints, such as maximum vehicle capacity and maximum travelling time, are not violated.

In another example, n cities are to be visited by m vehicles leaving from and returning to a depot, with known travelling time between cities and time spent in each city. Each vehicle must return within a certain time limit. The aim is to minimise the number of vehicles used and the travelling time associated with a given number of vehicles.

The example can be stated as a symmetric TSP. The depot is replaced by m artificial depots, with the distance v between two artificial depots set to find one of the following:

(1) the minimum total time for m vehicles.
    (v = very high)

(2) the minimum total time for any number of vehicles.
    (v = 0)

(3) the minimum total time for the minimum number of vehicles.
    (v = very low)

The time spent in each of any two cities can be allocated to the travelling time between these cities. Additional constraints can be incorporated in the formulation. (Lenstra and Rinooy Kan, 1975)

Clustering a data array is an approach useful for problem decomposition and data reorganization. Let the entries a_{ij} in a data array measure the strength of a relationship between elements i and j. A strong relationship between subsets of these elements can then be identified by rearranging the rows and columns of this array, that is permutations may reveal clusters of values.

For example, let i be a marketing technique and j a marketing application, and let

a_{ij} = 1 if technique i was successfully used for application j,

= 0 otherwise.

A cluster of ones would show which combination of techniques were succesful for which applications.

This problem can be converted into an optimization problem by defining an optimization criterion, called the "measure of effectiveness" or ME. The ME is the sum of all the products of horizontally and vertically adjacent elements. The problem is now one of finding the column and row permutation which maximises the ME. For example in Figure 2.3, no clustering is obvious for the 3x3 matrix with ME=0. Permutating the rows and columns of this matrix reveals a cluster of 1's when the ME=4.

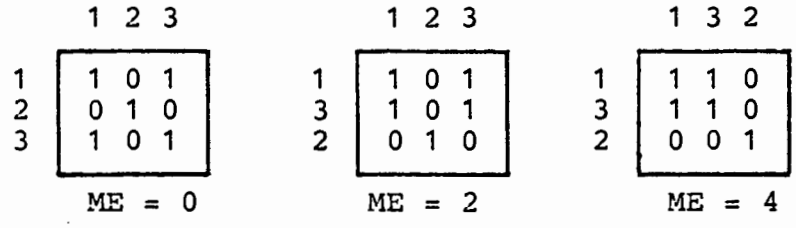


Fig. 2.3 ME for permutations of a 3x3 array

Maximising the ME of a matrix reduces to two separate but similar problems, one to maximise the ME of the columns and the other to maximise the ME of the rows. Each of these problems can be formulated as a symmetric TSP and solved by standard TSP solution techniques. (Lenstra and Rinooy Kan, 1975)

2.3.3 Exact solution methods

Exact solution methods exist for the Travelling Salesman

Problem, i.e. methods which can be proved (rigorously) to converge to the optimum solution of a problem.

One exact technique is that of integer programming, i.e. by optimising a linear function subject to a number of linear constraints, where all variables are constrained to be integers. The objective function for the Travelling Salesman Problem is the sum of the costs incurred if a specific tour is selected, and is to be minimised. The restrictions on the problem, for example that each node in a tour is connected to two other nodes also in the tour, can be formulated as linear constraints. The result is a 0-1 integer programming problem (Hu, 1969):

For a TSP with n nodes, artificially split the starting node into two, letting one of them be node 0 (starting node) and the other be node n (ending node). The problem is now one of finding an "open" tour from starting node 0 to end node n.

let x_{ij} = { 1 if the edge (i,j) forms part of the solution / tour
{ 0 otherwise
for $i=0,\ldots,n-1$ $j=1,\ldots,n$ and $i \neq j$

then the constraints of the problem are:

(1) one and only one node may be reached from node i (except from node n)

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \text{for } i=0,\ldots,n-1 \text{ and } i \neq j$$

(2) one and only one other node may initiate an edge to node j (except to node 0)

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \text{for } j=1,\ldots,n \text{ and } i \neq j$$

(3) the result must be a single tour, and not two or more subtours.

To achieve this, associate with each node i a real number

Y_i $(0 \leqslant Y_i \leqslant n)$

and include the constraints

$$Y_i - Y_j + nx_{ij} \leq n-1 \quad \text{for } i=0,..,n-1 \quad j=1,..,n \quad i \neq j$$

(3a) To show that these constraints will be satisfied by an open tour, let $Y_i = t$ if node i is the t-th node on the tour. Then

$$Y_i - Y_j \leq n-1 \quad \text{for all } i=0,..,n-1 \quad j=1,..,n \quad i \neq j$$

Thus the constraints hold for $x_{ij} = 0$, and for $x_{ij} = 1$ become

$$Y_i - Y_j + n.1 = t - (t+1) + n = n-1$$

(3b) Let there be a subtour of k edges in the solution. For each of these edges $x_{ij} = 1$ and there are k inequalities

$$Y_i - Y_j + n \leq n-1$$

Adding these inequalities we get, since the differences $Y_i - Y_j$ cancel, $nk \leq (n-1)k$

which is a contradiction. Thus the constraints of (3) prevent the formation of subtours.

The objective is to

$$\text{minimise} \quad \sum_{i=0}^{n-1} \sum_{j=1}^{n} c_{ij} x_{ij} \qquad \text{for } i \neq j.$$

The problem can now be solved by integer programming methods, such as the cutting plane / branch and bound technique of Crowder and Padberg (1980).

Another approach which has been used to solve the Travelling Salesman Problem is that of dynamic programming. This approach

requires a problem to be formulated as a sequence of decisions, with each decision made generating a return. The sequence of actions which minimises some function of the returns is then found. For the Travelling Salesman Problem, a decision is the selection of an edge, and a return is the cost incurred by including the edge into the solution tour. The function of the returns is the sum of the cost of the edges in the tour.

To solve the TSP as a dynamic programming problem, define stage k by the number of nodes not yet assigned to the tour. A state at stage k is defined by {S,i}, where S is the set of unassigned nodes, and i is the last node assigned to the tour, i.e. the (n-k)th node on the tour. Let the function of returns $f_k(S,i)$ be

the minimum cost from node i back to the origin, passing only through the nodes in S.

Initially, i.e. when k=n-1, S is the set of all nodes except the origin. Define $f_0(\{\},i) = C_{i,origin}$. The dynamic programming

algorithm for the TSP can then be given.

For k=1 to n-1:

For all possible combinations of {S,i}:

$$f_k(S,i) = \min_{j \in S} [\, C_{ij} + f_{k-1}(S-\{j\},j) \,]$$

Record j.

As an example, a 4-node TSP may be formulated as in Figure 2.4 (where node 1 is split into two, a start and an end node, with the routes connecting start and end node being the permutations of the nodes 2...n); the above dynamic programming algorithm can be used to solve this problem expressed in this form.

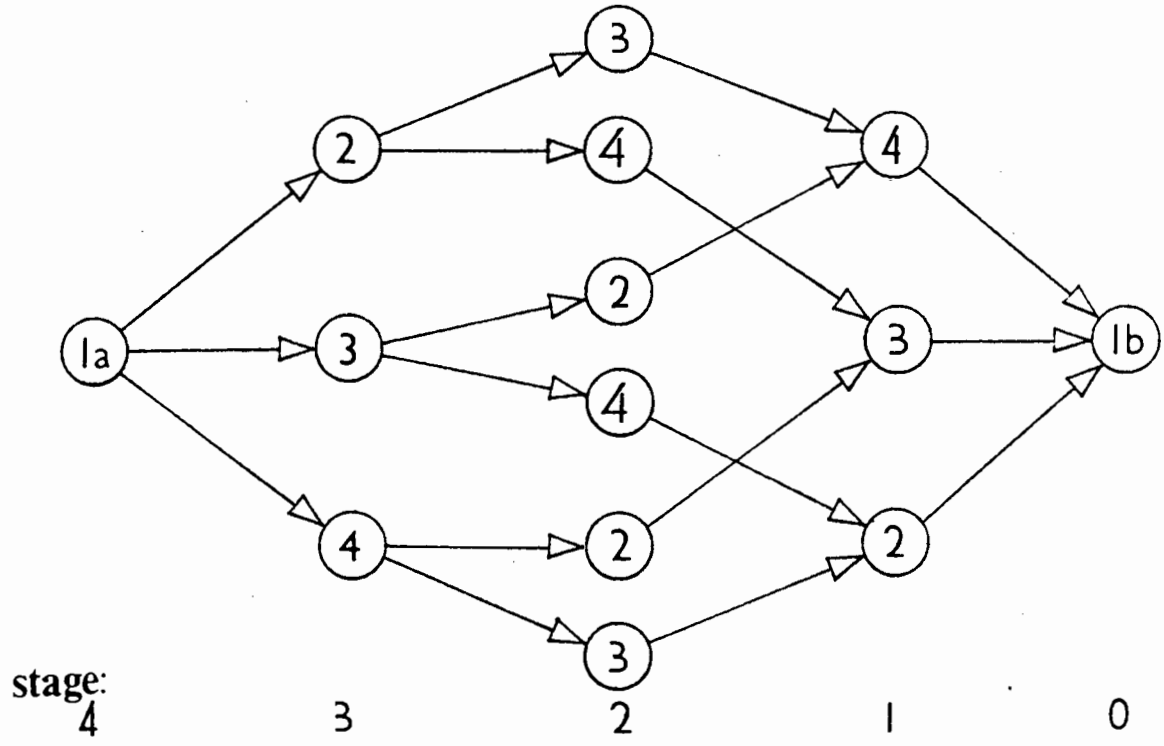


Fig. 2.4 Formulation of a 4-node TSP as a network problem, to be solved by dynamic programming techniques.

Dynamic programming requires large amounts of storage. It works best when the number of states are not too numerous. Thus for the TSP we require the number of nodes to be low.

2.3.4 Complexity of exact methods

Although, as shown in the previous section, it is known how to find the optimum solution of the Travelling Salesman Problem, difficulties can arise when using the above methods. All known exact methods show an exponential increase in computational effort with respect to the size of the problem, defined by the number of nodes, n, in the problem. In fact, the Travelling Salesman Problem is NP-complete (Papadimitriou, 1977).

For example, for the dynamic programming approach, the number of

combinations of {S,i}, i.e. the number of states at stage k, is

$(n-k-1)\binom{n-1}{k}$ for $k<n-1$, and 1 for $k=n-1$.

For each state, k additions and k-1 comparisons must be performed, giving a total of 2k-1 operations. Thus the total number of operations for a TSP of size n is

$$2n - 3 + \sum_{k=1}^{n-2} (2k-1)\,(n-k-1)\binom{n-1}{k}$$

which when expanded has the highest term in n in the form

$$n^2\,2^n.$$

Thus the dynamic programming approach has a running time of

$$O(n^2\,2^n)$$

to find the optimal solution of a TSP of size n.

Exact solution methods are therefore of limited use in solving large scale TSPs. Heuristics are the alternative.

3.1 Background

The process of finding the solution to the Travelling Salesman Problem by heuristics can be divided into two stages. First a feasible tour is found or constructed. As there is no guarantee that this tour is optimal, the second stage attempts to improve it using a tour improvement heuristic. The starting point of the improvement heuristic is the previously constructed solution. It is possible of course to use a tour construction heuristic only, without improving on its result; or to use a randomly generated starting solution for a tour improvement heuristic. If there is a limit on the time available to run both heuristics, fast construction methods can be combined with lengthier improvement ones, or vice versa.

A variety of heuristic tour construction methods are discussed in Section 3.2, and the tour improvement techniques in Section 3.3. As tour improvement techniques essentially only rearrange edges, there are relatively less heuristic variations possible than for the tour construction methods which deal with a much larger number of combinations, and consequently have many more possible approaches.

Where it is known, the "worst case behaviour" of a heuristic is included. This behaviour is the ratio of the worst possible heuristic solution to the optimal solution, i.e.

$$\frac{\text{worst heuristic solution}}{\text{optimal solution}}$$

In addition, the computational complexities of the heuristics (as functions of the problem size) are shown.

The types of TSPs that can be solved by specific construction and improvement heuristics vary from general asymmetric problems where the triangle inequality need not hold, to Euclidean problems. The discussion of each heuristic includes the TSP type the method is suitable for. All worst case behaviour and computational complexity results, however, assume that the TSP is symmetrical and that the triangle inequality holds.

In Section 3.4 some published comparisons of heuristics are discussed.

3.2 Tour construction heuristics

3.2.1 Introduction

Some definitions are necessary prior to discussing tour construction heuristics.

A "path" between two nodes a and b is a sequence of edges

$$(a,e_1), (e_1,e_2), (e_2,e_3) \ldots\ldots (e_k,b)$$

where $e_i \in \{1,\ldots,n\}$ but $e_i \neq a,b$ for $i=1,\ldots,k$, $k \leq n-2$

and $e_i \neq e_j$ for $i,j=1,\ldots,k$

The nodes a and b are termed the end nodes of the path. Note that if one edge is removed from a tour, the result is a path.

A "subtour" of a TSP is a tour of k nodes where $k<n$.

A tour construction heuristic starts with an empty "solution set" of edges. At each iteration, one or more edges are added to the solution until a full tour is constructed. The following are alternative ways of doing this:

(1) Ordered sequence methods:

At the start of the heuristic, all edges are ranked by some suitable criterion (as discussed in Section 3.2.2). At each iteration the best edge according to the criterion (the edge with the highest or lowest rank), is added to the solution. These rankings may or may not be updated at each iteration. Edges which violate any constraints of the problem are discarded. For example if the edge added to the solution in the previous iteration was the second edge from node i, then the ranks of all other edges into node i are excluded from further processing.

(2) Increasing path methods:

One node or edge is selected as the starting point of a path. At each iteration one edge is selected according to some criterion (see Section 3.2.3), and added to one of the ends of the path. A disadvantage of this approach is that it is possible for the ends of the path to be 'far' from one another when the heuristic terminates, i.e. the last edge needed to change the path to a tour may be an expensive edge.

(3) Subtour insertion methods:

An initial subtour is selected, for example one edge is selected. In each iteration, a node is selected and inserted into the subtour according to some criterion (as discussed in Section 3.2.4). To insert a node an edge in the subtour is replaced by two edges not in the subtour. After each iteration the number of nodes and edges in the subtour has increased.

(4) Merged multiple subtours:

Multiple subtours are constructed by some technique and are

then sequentially merged until one tour results. This approach will be discussed in Section 3.2.5.

A tour construction heuristic may be a combination of several subheuristics. The iterations of all but the final subheuristic lead to an intermediary solution, i.e. to a result that is not a tour. Further processing is necessary to convert this solution into a tour, the type of processing required depending on the type of intermediary solution found. In other words, the tour construction is completed by a combination of two or more heuristics, or of heuristics and exact methods, as discussed in Section 3.2.6.

The tour construction heuristics discussed are grouped according to the above categories. For categories (1) to (4) (Sections 3.2.2 to 3.2.5), a simple form of the particular construction approach is presented followed by more sophisticated approaches. Note that these sophistications increase the running time of a heuristic, and that while they usually result in a significantly improved solution being found, this need not always be true. In fact one must beware of too much local optimization (Johnson and Papadimitriou, 1985).

3.2.2 Ordered sequences

The three ordered sequence methods, the greedy, savings and loss heuristics, as discussed below, require the costs of the TSP to be symmetric. The savings and loss heuristics can be easily adapted to solve asymmetric TSPs (for example see Golden and Stewart, 1985).

Greedy heuristic.

Among the conceptually simplest techniques is the greedy heuristic (Fischer et al, 1978). Since the cheapest tour is required, a simple rule-of-thumb is to include the cheapest edges in the solution. The heuristic orders edges according to their cost, then selects the edge with the lowest cost from the list, adds it to the solution, and proceeds to the next lowest cost on the list. A cost whose edge would violate a constraint of the TSP if it were included is discarded. The process is continued until a tour is formed.

Algorithm.

Step 1. Find the cheapest edge. Include it in the solution.

Step 2. Repeat until a tour is found:

2.1 Find the cheapest edge (i,j) not yet included in the solution, subject to the restrictions that:
- nodes i and j are each of degree at most 1, and
- a subtour is not formed by including (i,j).

2.2 Include the edge in the solution of the TSP.

The heuristic runs in time of $O(n^2)$.

Savings heuristic.

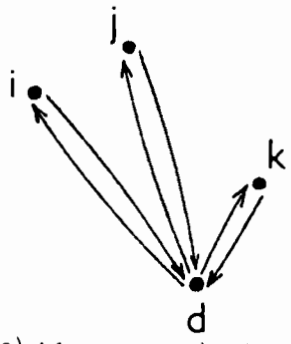
A heuristic which is a sophistication of the simple ordered sequence approach was first developed for vehicle routing problems by Clarke and Wright (1964) and has been adapted to the TSP (Golden, 1977).

An initial node is selected, and "savings" for all edges other than the edges into this node are calculated, as follows. Let the initial node be node d, and let the initial 'tour' be a visit to each other node, such that between every two visits node d is

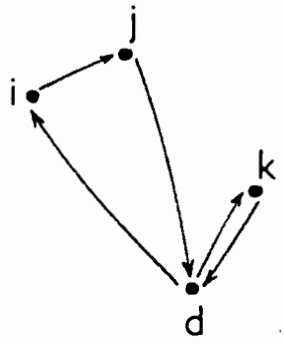
revisited, as in Figure 3.1(a). If two of these visits from node d, say those to nodes i and j, could be combined as is done in Figure 3.1(b), then there is a "saving" of

$$S_{ij} = C_{id} + C_{dj} - C_{ij}$$

If combining the visits to nodes i and j saves more than combining the visits to nodes i and k, say, i.e. $S_{ij} > S_{ik}$, then the former combination is preferred. A saving can be calculated for each edge not into the initial node.



(a) After a node is visited, node d is revisited.



(b) Both node i and node j are visited before returning to node d.

Fig. 3.1

Savings made by combining visits.

In order of largest to smallest saving S_{ij}, the appropriate nodes i and j are linked, until these links form a tour. A saving whose edge would violate a constraint of the TSP if it were included is discarded during processing. The heuristic is repeated for each node as initial node, or for as many initial nodes as possible.

Algorithm.

Repeat until all nodes have been selected as initial nodes:

Step 1. Select an initial node. Let this be node d.

Step 2. For all edges (i,j) where i,j $\in$ {1,...n} and i,j $\neq$d,

find the savings

$$S_{ij} = C_{id} + C_{dj} - C_{ij}$$

Sort the savings into descending order.

Step 3. Repeat until a tour is formed:

Include the edge with the largest saving in the solution. Delete that saving from the list. Also delete all savings S_{ij} where the inclusion of the edge (i,j)

would create a subtour.

The worst case behaviour of this heuristic has not been published, while the computation time is of

$$O(n^2 \lg n)$$

if a fast sort is used in step 2 and the heuristic uses each node in turn as the initial node. (Golden, Bodin, Doyle and Stewart, 1980)

Loss heuristic.

An ordered sequence approach which, unlike the savings heuristic, updates the order of an edge at each iteration has been proposed by Webb (1971).

Each node in a tour is always connected to two other nodes, by two distinct edges. Let i be a node which is not yet connected by any edges in the current solution. The preference would be to connect i through the two edges having cheapest costs

C_{i,i_1} and C_{i,i_2} say. If this cannot be achieved, the next

cheapest option would be C_{i,i_2} and C_{i,i_3} say, giving a "loss" (in

having to use edge (i,i_3) insted of (i,i_1)) of:

$$loss_i = C_{i,i_3} - C_{i,i_1}$$

If one edge connecting node i is already in the solution, then the preference would be to connect i through the edge with cheapest cost C_{i,i_1} say, with the next cheapest option as C_{i,i_2}

say, giving a loss for node i as: $loss_i = C_{i,i_2} - C_{i,i_1}$

In either case, a node with high loss can be taken as indication that not including the cheapest edge from this node in the solution, will lead to the sustaining of a large opportunity cost. Thus the heuristic at each iteration selects the edge which causes the highest of the losses.

One of several special cases for which the loss function has to be adapted, is when joining node i to the two closest nodes would result in the formation of a subtour. Webb (1971) supplies a Fortran algorithm to deal with all the special cases.

Algorithm.

Step 1. Calculate the loss per node.

Step 2. Repeat until a tour is formed:

2.1 For the node with the highest loss, say node i, include the edge causing this loss in the solution, say edge (i,j).

2.2 Update the losses for the nodes i and j.

The running time for this algorithm is $O(n^2)$. It is possible to update the losses only occasionally, resulting in a faster algorithm without the solution being much affected. Webb showed experimentally that this variation runs in time of $O(n^{1.47})$. The faster heuristic has been tested on large problems of up to 2500 nodes (Webb, 1971).

3.2.3 Paths

The first two of the path construction methods, the nearest neighbour and dynamic weighting heuristics, can be used to solve TSPs which are asymmetric. The third method, the spacefilling curve heuristic, is best suited to the Euclidean TSP.

Nearest neighbour heuristic.

The simplest technique which uses a path to construct a TSP solution, is the nearest neighbour heuristic (Rosenkrantz, Stearns and Lewis, 1977). It proceeds by forming an ever increasing path, always adding the node 'closest' to an end of the path, i.e. including in the solution edge (i,j) where node i is an end node of the path, node j is not in the path, and out of all the edges that satisfy these two conditions edge (i,j) is the cheapest. The algorithm below can be repeated using each node as starting node, then selecting the cheapest tour out of all the tours found.

Algorithm.

Repeat until all nodes have been used as starting nodes:

Step 1. Start with any node. This is the first node in the path.

Step 2. Repeat until all nodes are in the path:

For node i an end node of the path, find node j not on the path such that cost C_{ij} is the cheapest of all such nodes not on the path.

Step 3. Join the first and last nodes of the path.

The running time of the heuristic is of $O(n^2)$ and, if the triangle inequality holds, the worst case performance is

$$.5 \lceil \lg_2 n \rceil + .5$$

where $\lceil x \rceil$ is the smallest integer greater than or equal to x.

Dynamic weighting heuristic.

The nearest neighbour heuristic proceeds by selecting the node not yet selected which is closest to an end node of the path being constructed. Pohl (1977) suggests an additional selection criterion which will be called "dynamic weighting". In addition to considering the cost of adding one more node to the end of the path, an attempt is made to include the effect that choosing this node has on the solution.

Let L be a lower bound on the optimum solution of the TSP. Fast algorithms which find lower bounds are known (e.g. Pohl, 1973), although of course there is a trade-off between computational efficiency and quality of the bound. The effect of choosing a node can thus be approximated by finding the lower bound on the tour completion through the unselected nodes, and adding it to

the cost of the current path and the cost of getting from the path to the chosen node. The cost of the current path is the same for whichever node is under consideration during one iteration, and therefore need not be included in the evaluation.

For s and t, denoting end nodes of the path P, let i be any node not in path P. The values which influence the choice of i are the cost of the edge between node s and node i, C_{si}, and the lower bound L(i) of the path from node i to node t such that this path includes all nodes not in path P. The effect of choosing node i is thus approximated by

$$f(i) = W_1 C_{si} + W_2 L(i)$$

where the W's represent relative weights on each criterion. Pohl (1977) suggests weights such as:

(1) $W_1 = W_2 = \frac{1}{2}$

(2) $W_1 = 1, \qquad W_2 = 1 + \exp\left(1 - \frac{\text{depth of } i}{n}\right)$

where the depth of node i is the number of edges between i and the initial node of the path P. Here the weight on the lower bound increases as the algorithm progresses.

Algorithm.

Repeat for each node as the initial node:

Step 1. Select a node to be the initial node of the path.
(This node also represents the two end nodes of the path for the first iteration of the next step.)

Step 2. While there are nodes not in the path do:

2.1 For each node not in the path find the lower bound L(i), and compute f(i).

2.2 Select the node which minimizes f(i). Add it to the

end of the path. (The path now ends at this node.)

Step 3. Join the two end nodes of the path.

If a lower bound which runs in time of $O(n^2)$ is used, such as the in-out-estimator suggested by Pohl (1973), then the algorithm has a running time of $O(n^4)$.

Spacefilling curve heuristic.

A different approach was developed initially for a TSP that needed to be solved manually. This arose out of the routing problem of a 'meals-on-wheels' service to the elderly where the charity lacked the funds to purchase even the simplest computer configuration. As the delivery list changes constantly, a one-off computer generated tour was not acceptable either. The TSP formulation of this problem was solved by a heuristic based on the spacefilling curve. (Bartholdi, Platzman, Collins and Warden, 1983)

The nodes (delivery points) of the problem are assumed to lie in a flat plane, and for convenience we assume this plane is the unit square [0,1]X[0,1]. The spacefilling curve used is a continuous mapping F from the interval [0,1] onto the unit square, and is the limit of the sequence of curves in Figure 3.2. (Platzman and Bartholdi, 1984)

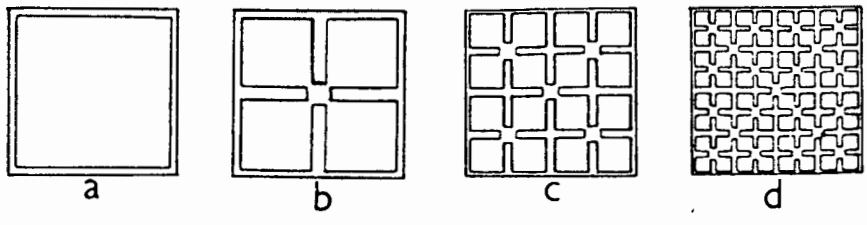


Fig. 3.2

Successions of the spacefilling curve

Each node is assigned a value representing its position along the spacefilling curve superimposed onto the unit square, i.e. for every node p, find $q \in [0,1]$ such that

$$p = F(q) \qquad \text{where } p=(x,y) \in [0,1]X[0,1].$$

Platzman and Bartholdi (1984) present a recursive algórithm to find q, first finding the quadrant of the unit square containing p and then finding the position of p along the part of the curve in this quadrant.

A tour is found by sequencing the nodes according to their respective q values, that is they are sequenced as they appear along a spacefilling curve. An outline of the algorithm for the heuristic is:

Algorithm.

Step 1. For each node p find $q \in [0,1]$ such that

$$p = F(q)$$

Step 2. Sort the nodes according to the corresponding q value.

When no computer is available, the calculation of the q-values can be replaced. The procedure is then as follows. A diagram of the nodes, for example a road map with delivery points marked, is overlaid by a grid. A set of (x,y)-coordinates is read off for each node. The coordinates of a node are used to look up the required q-value from a set of tables that need only be calculated once. The q-values are sorted, and a tour is found by visiting the nodes in the order of the corresponding q values. (Bartholdi et al, 1983)

Platzman and Bartholdi (1984) conjecture that the worst case bound for the heuristic is 4.7, but can only prove a bound of O(ln n). This worst case bound compares unfavourably with, say, the bound of 1.5 for the Christofides heuristic (Section 3.2.6). The running time of O(n ln n) for the spacefilling curve algorithm, however, compares favourably with the time of $O(n^3)$ for the Christofides algorithm.

3.2.4 Subtour insertions

Simple insertion heuristics.

The simplest subtour insertion approach is a heuristic which selects one edge as an initial subtour, at each iteration inserts one more node into the subtour, and continues until all the nodes have been inserted.

The general algorithm.

Step 1. Select an edge. This is the initial subtour.
(Selection criteria are discussed below.)

Step 2. While there are nodes not yet in a tour do:
Select one node k and insert it into the subtour. This implies deleting one edge from the subtour, say edge (i,j), and adding two new edges (i,k) and (k,j).
(Selection and insertion criteria are given below.)

Several selection and insertion rules have been proposed: (Rosenkrantz, Stearns and Lewis, 1977, Golden, Bodin, Doyle and Stewart, 1980, Johnson and Papadimitriou, 1985)

(1) The nearest addition heuristic
- let the initial edge be the cheapest edge.
- select node k' which minimizes

C_{ik} where i is any node in the subtour and k is any node not in the subtour.

- let the minimum occur for node i', then insert k' on either side of i', whichever of these two insertions is cheaper.

(2) The nearest insertion heuristic improves on the nearest addition heuristic by inserting the selected node in the cheapest possible way, that is by minimizing the cost of replacing an edge in the subtour by two new edges not in the tour.

- let the initial edge be the cheapest edge.
- select node k' which minimizes

C_{lk} where l is any node in the subtour and k is any node not in the subtour.

- find edge (i',j') which minimizes

$C_{ik'} + C_{k'j} - C_{ij}$ where (i,j) is any edge of the subtour,

and insert k' between i' and j'.

(3) The farthest insertion heuristic is a variation of the nearest insertion heuristic. Instead of selecting k' to minimize cost, choose k' "far" from the subtour, i.e. find the edge with one node in the tour and one not and which has the maximum cost. The insertion rule, however, is unchanged.

- let the initial edge be the most expensive edge. If the cost of an edge represents its length, this implies finding the longest edge.
- select node k' which maximizes

C_{lk} where l is any node in the subtour and k is any node not in the subtour.

- find edge (i',j') which minimizes

$C_{ik'} + C_{k'j} - C_{ij}$ where (i,j) is any edge of the subtour,

and insert k' between i' and j'.

(4) A final variation is the cheapest insertion heuristic which

chooses the node to be inserted into the subtour in such a way that the result is the cheapest possible.

- let the initial edge be the cheapest edge.
- select node k' and edge (i',j') which minimizes

  $C_{ik} + C_{kj} - C_{ij}$ where (i,j) is any edge of the subtour, and k is any node not in the subtour.
- insert k' between i' and j'.

In reported empirical studies (Rosenkrantz et al, 1977, Golden and Stewart, 1985), the heuristic using the farthest insertion rule has produced the best results out of these four variations for problems where the triangle inequality holds. An intuitive reason for this is given by Rosenkrantz et al (1977) - "the method establishes the general outline of the approximate tour at the outset and then fills in the details." Table 3.1 lists the published running times and worst case performance of some of the above heuristics.

heuristic	running time	worst case performance
nearest insertion	$O(n^2)$	2
farthest insertion	$O(n^2)$	$2 \ln n + 0.16$ but probably closer to 1.5
cheapest insertion	$O(n^2 \lg_2 n)$	2

Table 3.1 Performance of some simple insertion heuristics. (Rosenkrantz et al, 1977, Johnson et al, 1985)

The simple insertion heuristics as well as the difference heuristic, the next method to be discussed, can be used to solve asymmetirc TSPs.

Difference heuristic.

More complex insertion rules are possible. Raymond (1979) published an insertion rule which we shall call the "difference" rule. We first need some definitions.

Define the insertion cost of a node k between nodes i and j to be

$$C_{ik} + C_{kj} - C_{ij} \qquad \text{where } (i,j) \text{ is an edge in the subtour,}$$

i.e. the cost of the new edges from the subtour to node k less the cost of the edge which would be removed from the subtour if node k were inserted. Define M_k and N_k as the lowest and second lowest insertion costs of node k amongst all (i,j) in the subtour.

If the "difference" between M_k and N_k is large, we have reason to suspect that the cheapest insertion place of node k will lead to much better tours than any of the other possible insertion places. Furthermore, if the difference $N_k - M_k$ for node k is much larger than the difference $N_l - M_l$ for some other node l, then there is reason to believe that ensuring the insertion of node k in the cheapest way, rather than node l, is important in finding a good tour. Thus the difference rule is to insert node k' which maximizes $|N_k - M_k|$ for all nodes k not in the subtour.

Algorithm.

Step 1. Select two nodes to form the initial subtour.

For example, select the cheapest edge of the problem.

Step 2. Repeat until a tour is formed:

2.1 For each node k not on the subtour find the lowest insertion cost M_k with associated edge (i_k, j_k), and also the second lowest insertion cost N_k.

2.2 Find node k' which maximizes $|N_k - M_k|$ for all k.

2.3 Insert node k' into the subtour by including edges $(i_{k'}, k')$ and $(k', j_{k'})$ and removing edge $(i_{k'}, j_{k'})$.

The running time of the algorithm is of $O(n^3)$.

Raymond (1979) also suggests two modifications of this algorithm. One is to replace step 2.2 for the first few insertions with a maximization which resembles the one used in the farthest insertion heuristic, to exploit the advantages of that heuristic. The other is to check the subtour after every second insertion to determine whether an exchange of up to three edges in the subtour with edges not in the subtour will improve it. (This check does not increase the number of nodes in the subtour.) Raymond shows that the difference heuristic with the modifications is, in empirical tests, capable of producing better results than the simple insertion approaches.

Convex hull heuristics.

If we have a TSP that is Euclidean, that is each node is a point

in the x-y plane and the cost of an edge is the Euclidean distance between two nodes, we can use properties of the Euclidean space when solving the problem. The convex hull can be formed, which can be described as the shortest subtour with all nodes either on it or enclosed by it. Then we have the property (Or, 1976) that there is an optimal tour of the TSP in which the nodes on the convex hull are visited in the same order as in this subtour. Heuristics for the TSP can thus be based on first forming the convex hull of the nodes, using it as the first subtour.

An example of an algorithm used to find the convex hull is included below. It runs in time $O(n^2)$.

Algorithm.

Step 1. Form the convex hull.

(1) Select the node with the smallest y-coordinate and define it to be the start node. Call the node i.

(2) A dummy line is drawn parallel to the x-axis through node i. Find node j such that edge (i,j) forms the smallest angle with the line. This is the first edge of the convex hull. Let edge (i,j) be the "current" edge.

(3) Repeat until the hull is closed, that is until node j of the current edge (i,j) is the start node:

(3.1) For the current edge (i,j) draw a line through nodes i and j. Find node k such that the angle between edge (j,k) and the line through node j is the minimum.

(3.2) Include node k in the convex hull, and set the current edge to edge (j,k).

(See figure 3.3)

Step 2. While not all nodes are in the tour, select and insert one node according to some criterion, alternatives for which are outlined below.

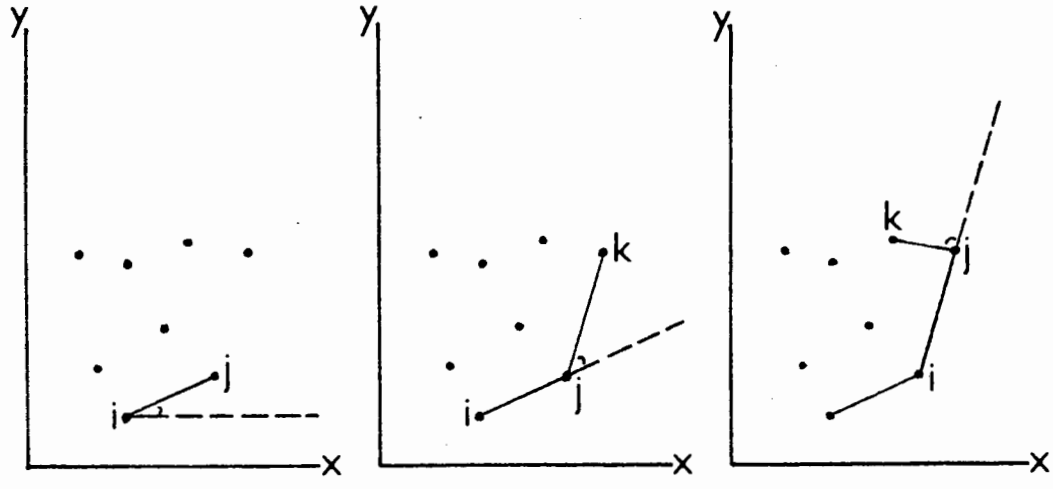


Fig. 3.3 First 3 iterations of the convex hull algorithm.

Several rules for the insertion of nodes into the convex hull have been proposed. Norback and Love (1977) suggest two rules, one of which is the greatest angle insertion rule. This algorithm includes node k not on the partial tour between nodes i and j in the partial tour if the angle formed by edges (i,k) and (k,j) is the maximum of all nodes not on the tour.

Another approach is the most eccentric ellipse rule. Instead of finding the largest angle, an ellipse is formed with two adjacent nodes i and j (on the tour) as the foci and a node k (not on the tour) on the ellipse. The node which determines the most eccentric (least circular) ellipse is included between the two nodes used as foci for this ellipse.

Norback and Love found that of these two methods, the most eccentric ellipse rule gave the better results.

Other insertion rules are (Or, 1976):

Insert node k such that:

(1) the perpendicular distance from node k to any edge (i,j) in the subtour is a minimum.

(2) the difference $C_{ik} + C_{kj} - C_{ij}$, as used in the simple insertion heuristics, is a minimum. I.e. the cost of forming the two new edges (i,k) and (k,j), less the cost of the edge (i,j) which will be replaced, is minimized.

(3) the ratio $(C_{ik} + C_{kj}) / C_{ij}$ is a minimum.

The cost of the new edges is small with respect to the edge which will be replaced.

(4) the ratio times the difference is a minimum.

The idea is to avoid the cases where either the difference or the ratio rule on their own do not select the node which by topology and common sense should be selected first. For example, in Figure 3.4 the difference rule cannot distinguish between case (a) and (b), but (b) would seem to be preferable. Similarly the ratio rule cannot distinguish between case (c) and (d), where (c) would seem to be preferable.

The ratio-times-difference approach has produced the best results in empirical studies out of these four rules (Or, 1976).

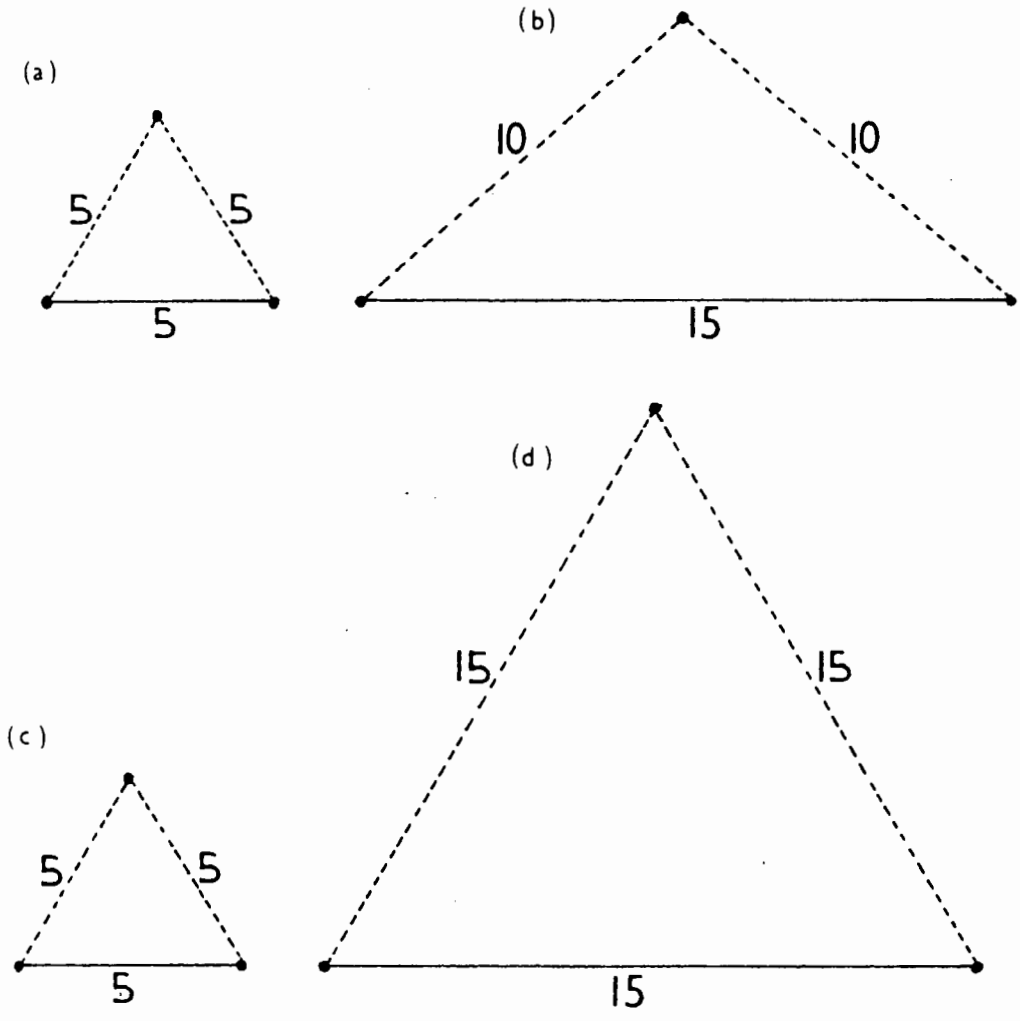


Fig. 3.4 Special cases for the ratio and difference rules. (after Or, 1976)

Golden and Stewart (1985) combine the cheapest insertion and greates angle ideas into a single method which we shall term the cheapest angle rule. For each node k not on the subtour find the edge (i_k, j_k) in the subtour, which minimizes

$$C_{i_k k} + C_{k j_k} - C_{i_k j_k}$$

Include the node k' into the subtour whose edges $(i_{k'}, k')$ and $(k', j_{k'})$ form the largest angle.

By combining two selection rules, here the cheapest insertion

and the greatest angle rules, an attempt is made to exploit the advantages of each rule while minimizing each ones disadvantages, as is attempted by the ratio-times-difference rule. In empirical studies (Golden and Stewart, 1985), the cheapest insertion angle selection heuristic performed the best out of all the simple insertion heuristics and all the convex hull heuristics.

All the insertion rules have a running time of $O(n^3)$. Thus the convex hull insertion heuristics run in a time of $O(n^3)$.

3.2.5 Merged multiple subtours

Both of the multiple subtour techniques discussed, the nearest merger and the patching or assignment heuristics, can be used to solve TSPs which are asymmetric.

Nearest merger heuristic.

The nearest merger heuristic starts with n subtours of a single node each, and at each iteration selects the two 'closest' tours and merges them. (Johnson and Papadimitriou, 1985)

Algorithm.

Step 1. Let each node represent a subtour.

Step 2. Repeat until one tour is formed:

(The next two steps are similar to those of the nearest insertion heuristic.)

2.1 Find the two subtours which minimize C_{ij} over all pairs i and j in different subtours.

2.2 For these two subtours, find edges (k',l') and

(g',h') which minimize

$$C_{kg} + C_{lh} - C_{kl} - C_{gh}$$

where (k,l) is an edge in one subtour and (g,h) is an edge in the other. Replace edges (k',l') and (g',h') by edges (k',g') and (l',h').

The running time of the heuristic is of $O(n^3)$.

The assignment problem and patching heuristic.

We return to the formulation of the TSP as an integer programming problem (Section 2.3.3). Dropping the third set of constraints which restrict the formation of subtours, results in an I.P. problem of the form:

$$\text{minimise} \quad \sum_{i=0}^{n-1} \sum_{j=1}^{n} c_{ij} x_{ij} \qquad \text{for } i \neq j.$$

subject to

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \text{for } i=0,\ldots,n-1 \text{ and } i \neq j,$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \text{for } j=1,\ldots,n \text{ and } i \neq j,$$

and $x_{ij} = 0$ or 1 for $i=0,\ldots,n-1$ $j=1,\ldots,n$ and $i \neq j$.

This problem is known as the "assignment problem", and can be

solved by special L.P. algorithms in time O (n^3). (See for example Balas and Toch, 1985).

The subtours, if any, that result from solving the assignment problem form of the TSP, need to be patched together into one tour. An example of a patching heuristic is given by Karp and Steele (1985).

Patching algorithm.

Repeat until no subtours remain:

Step 1. Find S_1 and S_2, the two longest subtours where "longest"

means having the largest number of edges.

Step 2. Find edge (i ', j ') in S_1 and edge (k ', l ') in S_2

which minimize

$$C_{ik} + C_{jl} - C_{ij} - C_{kl}$$

for all edges (i,j) in S_1 and edges (k,l) in S_2. Replace

edges (i',j') and (k',l') by edges (i',k') and (j',l').

Note that the nearest merger heuristic and the patching heuristic have the same approach to the selection of the edges to be replaced. The running time for the patching heuristic is of

O (n^3), and the worst case ratio is bounded by $\dfrac{v_A + v_I}{v_A}$

where v_A is the cost of the solution to the assignment problem,

and v_I is the cost of the insertions. For a TSP with n large, the probable error of the solution found by this heuristic decreases toward zero. (Karp and Steele, 1985)

Other techniques to convert the solution of an assignment problem to the solution of the related TSP are given by Balas and Toch (1985) and Diegel (1986).

3.2.6 Combination of approaches

Christofides heuristic.

Some definitions are necessary before the Christofides heuristic can be stated. The assumptions are made that the triangle inequality holds and that the costs are symmetric.

A tree is a set of edges such that each edge has at least one node in common with another edge (one could say that each edge is attached to at least one other edge), but no subtours are allowed. In other words, a tree is a set of nodes and edges such that a path exists between any pair of nodes in the set, without any loops existing. If the problem consists of n nodes, then a tree which includes all n nodes is called a spanning tree. We can now define the "minimum spanning tree" as the spanning tree which minimizes the sum of the costs of the edges in the tree. Fast exact solution algorithms are known for the minimum spanning tree (for example Foulds, 1984).

Let the "degree" of a node be the number of edges included in a solution which connect that node. Thus, for example, the TSP requires that each node is of degree 2.

A "matching" of a set of nodes links each node with one other node, i.e. all nodes are of degree 1. If the number of nodes in this set is odd then including a dummy node with dummy edge costs ensures that a matching can be found. A minimum matching has the minimum sum of the costs of the edges used. For the minimum matching the dummy costs are set higher than any other edge cost. Algorithms to find a minimum matching are given by, for example, Lawler (1976).

Finally we must define a "shortcut". Suppose that we have a tour in which one node is visited more than once. In particular, suppose that edges (j,i), (i,k), (g,i) and (i,h) are included in the tour. We can shortcut past i by replacing one pair of edges by a single edge in the minimum cost manner. For example if

$$C_{ji} + C_{ik} - C_{jk} \geq C_{gi} + C_{ih} - C_{gh}$$

(i.e. the savings in shortcutting past node i when proceeding from node j to k is greater than the savings in shortcutting from g to h), then replace edges (j,i) and (i,k) by edge (j,k).

The algorithm to solve the TSP can now be given. First a minimum spanning tree is found. This results in some of the nodes being of odd-degree and the others not. For the odd-degree nodes, of which there will be an even number, a minimum weight matching algorithm finds the minimum cost inclusion of exactly one more edge for each one of these nodes. Thus the edges of the minimum spanning tree plus those of the minimum matching result in each node having an even degree, that is each is of degree 2, 4, 6, or more. A travelling salesman tour is found by shortcutting past any duplicate nodes. (Christofides, 1976)

Algorithm.

Step 1. Find the minimum spanning tree.

Step 2. For the odd-degree nodes in the tree, find a minimum

matching.

Step 3. Shortcut past duplicate nodes.

The running time of the heuristic is of

$$O(n^3)$$

and the ratio of the heuristic solution to the optimal solution is at worst 3/2. (Johnson and Papadimitrou, 1985)

Partitioning heuristics.

Partitioning algorithms divide the region of the TSP into subregions, and optimal tours within each subregions are combined to yield a tour through all the nodes. For convenience this region is assumed to be a Euclidean plane. Karp (1977) presents two partitioning algorithms which find a spanning walk through the n nodes. A "spanning walk" is an improper tour in which a node may be visited more than once. The walk is then transformed into a tour.

Partitioning heuristic 1.

Let $d = \lceil \log_2 (n-1)/(t-1) \rceil$

where $\lceil x \rceil$ is the smallest integer $\geq x$

n is the number of nodes in the problem,

t is given by the user. There will be at most t nodes in each partition, so t must not exceed the capability of the method used to find an optimal tour in each partition.

Step 1. Partition the region into 2^d subregions, each of which containing most t nodes, such that there is a node on the border between each adjacent partitions. A node on a border is assumed to be contained in each of the two

adjacent partitions.

Step 2. Construct an optimum tour within each subregion for both the nodes within the partition and those on the border of the partition or subregion. For t not too large, an exact method can be used. The union of these tours is a spanning walk.

Step 3. Transform the spanning walk into a tour by shortcutting past the duplicate nodes in the walk in the same way as is done for the Christofides heuristic.

Partitioning heuristic 2.

Let $d = \lceil \log_2 n/t \rceil$

where n and t are as for heuristic 1.

Step 1. Partition the region into 2^d subregions, each of at most t nodes.

Step 2. Construct an optimum tour within each subregion. Link tours in adjacent subregions by an edge, to form a spanning walk. Each link added will occur twice in the walk, and is selected as cheaply as possible.

Step 3. Transform the spanning walk into a tour by shortcutting past duplicate nodes.

The heuristics run in time of $O(n \ln n)$. Karp (1976) proves that for every $\varepsilon > 0$, the second partitioning heuristic can solve the Euclidean TSP to within $1+\varepsilon$. This heuristic can also be used to solve symmetric TSPs where the triangle inequality holds (Karp and Steele, 1985).

Strip heuristic.

A simpler partitioning approach is the decomposition heuristic which partitions the region into strips, then traverses up one

strip, down the next strip, up the next strip, and so on. The nodes of the TSP are assumed to lie in the unit square [0,1]X[0,1]. (Supowit, Reingold and Plaisted, 1983)

Algorithm.

Step 1. Let

$$d = \left\lceil \; 2\sqrt{n} \;\Big/\; \sqrt{\log_z f(n)} \; \right\rceil$$

where $Z > 2$, real,

$f(n)$ is a nonnegative, unbounded, nondecreasing integer-valued function computable in $O(nf(n))$ time.

Step 2. Divide the unit square into d^2 squares, each square having side length of $1/d$.

Step 3. Find the optimum tour within each square. An exact method can be used.

Step 4. Sequence the squares by starting at, say, the top left corner of the region, moving down the left column of squares, then up the next column, down the column after, and so on. These columns are called strips.

Step 5. For all adjacent squares in the sequence, join the adjacent tours as follows. In each tour delete one edge; for example delete the most expensive (longest) edge. Each square now has a path. Join one end node of the path in square S to an end node of the path in the square adjacent to S. Join the other end node in S to an end node of the path in the other adjacent square. Repeat these joins for all adjacent squares, defining the first and last squares as being adjacent to one another. The result of this step is a tour.

The running time of the heuristic is of $O(n \ln n)$ and the

worst case ratio is O(n). (Supowit et al, 1983)

3.3 Tour improvement heuristics

3.3.1 Edge exchange

Improvements of an initial tour are attempted by searching for advantageous exchanges of edges in the tour with edges not in the tour. Each exchange made reduces the cost of the tour, or at least leaves the cost unchanged, and a heuristic terminates if no more advantageous edge exchanges can be found. The heuristics discussed below assume that the TSP costs are symmetric.

k-opt heuristic.

The number of edges exchanged in each iteration step of the heuristic is fixed at the value k. The tour found by the heuristic is said to be k-optimal if no additional k-exchanges can be made. Lin (1965) published the following heuristic for k=2,3.

Algorithm.

Step 1. Generate a tour, either randomly or by a tour construction heuristic.
Set k=2.

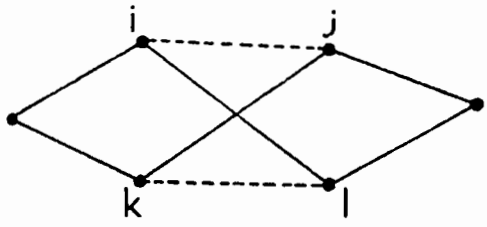
Step 2. Repeat until no more improvements can be found:
2.1 For each combination of k edges in the tour, compare the cost of these edges with the cost of replacing them by a set of k edges not in the tour. The edges not in the tour are selected such that if they replace the k edges in the tour, the result is still a tour. For example, in Figure 3.5 (a) exchange edges

(i,l) and (k,j) by edges (i,j) and (k,l).

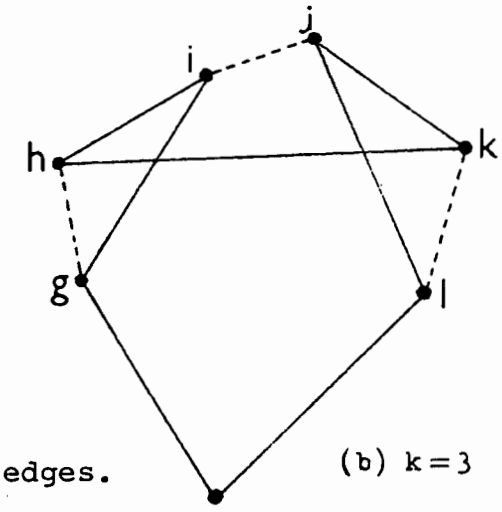
Step 3. Set k=3.

Step 4. Repeat step 2.

For an example of a 3-edge exchange see Figure 3.5 (b), where edges (g,h), (i,j) and (k,l) replace edges (g,i), (h,k) and (j,l).



(a) k=2



(b) k=3

Fig. 3.5 Exchange of k edges.

The running time of the k-opt heuristic is of $O(n^k)$ (Golden, Bodin, Doyle and Stewart, 1980), and the worst case performance for $n \geq 8$ and $k \leq n/4$ is $2(1 - 1/n)$ (Parker and Rardin, 1983).

Or-opt heuristic.

A heuristic similar to the k-opt heuristic with k=3, evaluates only some of all the possible combinations of edges (Or, 1976). The running time of this heuristic is substantially less than that of the 3-opt heuristic, while the results are comparable (Golden and Stewart, 1985). Instead of comparing each combination of three edges, two of the edges are selected so that they have 3, 2 or 1 adjacent nodes connecting them.

Algorithm.

Step 1. Generate a tour.

Step 2. Repeat until no more improvements can be found:

Consider every three adjacent nodes such as for example i_1, i_2 and i_3 between i_0 and i_4 in Figure 3.6 (a). If

an edge (j_1, j_2) can be found in the tour such that

inserting the three nodes between nodes j_1 and j_2

results in a cheaper tour, then replace edges

(i_0, i_1), (i_3, i_4) and (j_1, j_2) by edges (i_0, i_4), (j_1, i_1)

and (i_3, j_2) as in Figure 3.6 (b).

Step 3. Repeat step 2 for every two adjacent nodes.

Step 4. Repeat step 2 for each node.

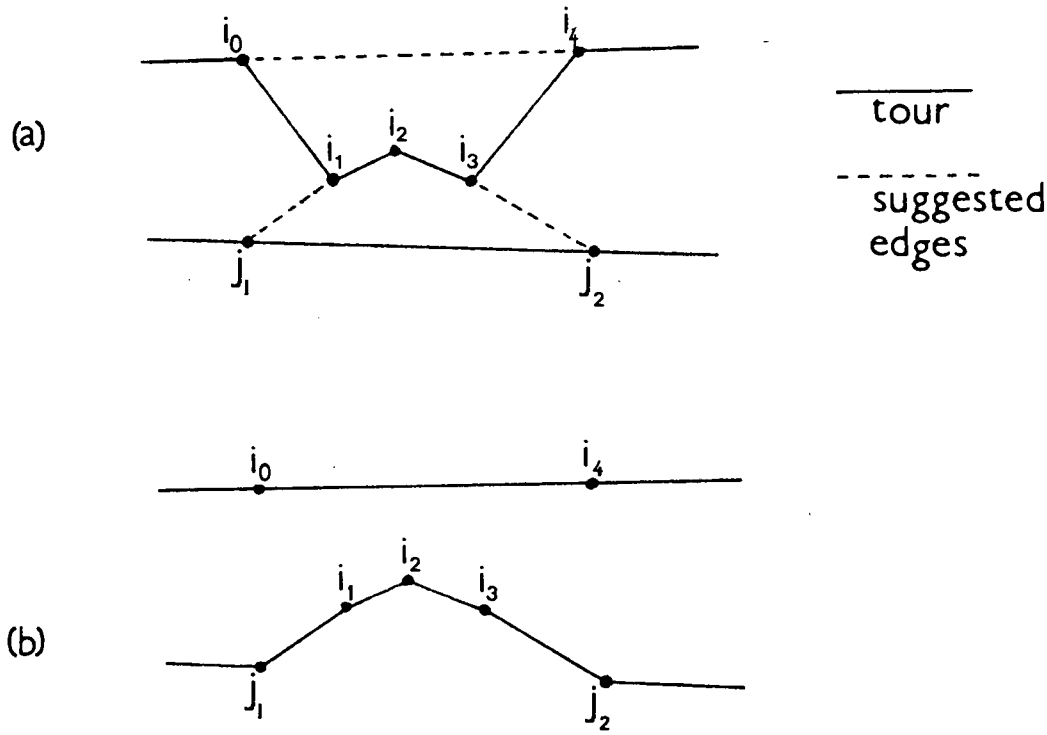


Fig. 3.6 Edge exchange for three adjacent nodes.

The heuristic runs in time $O(n^2)$.

Variable k-opt heuristic.

A variation of the k-opt heuristic determines the value of k during each iteration by letting the number of exchanges made depend on the edge currently being evaluated. Lin and Kernighan (1978) give a six step algorithm for this heuristic.

3.3.2 Simulated annealing

Optimization heuristics such as the edge exchange heuristics for the TSP, attempt to iteratively improve a given solution of a problem. At each iteration of an edge exchange heuristic such as the k-opt heuristic, a decrease of the tour cost is sought, and when no further decreases are possible the algorithm terminates. The final solution found may, however, be a local minimum. As only strictly decreasing solutions are possible, the algorithm is said to have got "stuck" at this minimum. For example, the tour in Figure 3.7(a) is k-optimal for k=2, but is not the shortest tour possible.

If a slight increase in tour cost were possible, it may be possible to overcome this tendency. Figure 3.7(b) shows a two edge exchange which increases total cost, but when the 2-opt heuristic is applied to the larger tour, a minimum tour is found (Figure 3.7(c)).

Variable k-opt heuristic.

A variation of the k-opt heuristic determines the value of k during each iteration by letting the number of exchanges made depend on the edge currently being evaluated. Lin and Kernighan (1978) give a six step algorithm for this heuristic.

3.3.2 Simulated annealing

Optimization heuristics such as the edge exchange heuristics for the TSP, attempt to iteratively improve a given solution of a problem. At each iteration of an edge exchange heuristic such as the k-opt heuristic, a decrease of the tour cost is sought, and when no further decreases are possible the algorithm terminates. The final solution found may, however, be a local minimum. As only strictly decreasing solutions are possible, the algorithm is said to have got "stuck" at this minimum. For example, the tour in Figure 3.7(a) is k-optimal for k=2, but is not the shortest tour possible.

If a slight increase in tour cost were possible, it may be possible to overcome this tendency. Figure 3.7(b) shows a two edge exchange which increases total cost, but when the 2-opt heuristic is applied to the larger tour, a minimum tour is found (Figure 3.7(c)).

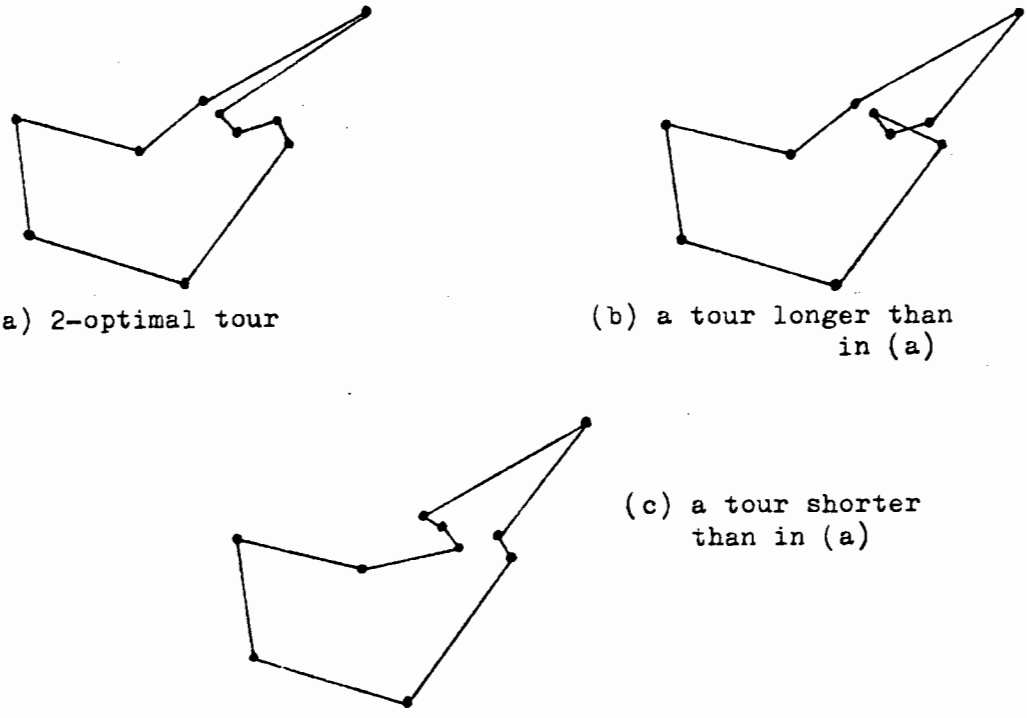


Fig. 3.7

Allowing an increase in tour cost.

(after Golden and Skiscim, 1986)

The approach of allowing slight increases during convergence towards the minimum solution has been suggested by analogy with the statistical mechanics model of the annealing process (Kirkpatrick, Gelatt and Vecchi, 1983). Annealing is the process whereby a material is melted and then slowly cooled, with much time being spent at temperatures close to freezing point. If the cooling proceeds too quickly, irregularities are frozen into the material, the lowest energy state is not achieved, and the material will lack strength.

The analogy between annealing and combinatorial optimization is achieved by defining a control parameter corresponding to

"temperature", which changes as the optimization proceeds. This parameter controls the degree of freedom allowed in letting the solution bounce around, perhaps to solutions with increased values. Reaching the temperature of freezing point is analogous to terminating the optimization process, i.e. freezing the solution at the current value.

If the annealing process is executed carefully (and the same is true for "simulated annealing" in combinatorial optimization), then the result is that the material reaches a nearly minimum energy state (or the combinatorial problem a nearly optimum solution). Too rapid cooling, resulting in defects in the frozen material, is analogous to a local search heuristic getting stuck at a local minimum. (Lundy and Mees, 1986 and Johnson, Aragon, McGeoch and Schevon, 1987) The above points are summarized in Table 3.2.

Table 3.2 Analogy between annealing and combinatorial optimization.

annealing process	optimization of combinatorial problems
temperature	(changing) control parameter
current state	feasible solution
energy of the state	cost of the solution
minimum energy state	optimal solution
careful annealing	simulated annealing
rapid cooling	local search

An algorithm for simulating the annealing process in solving the TSP is as follows (Johnson et al, 1987):

Simulated annealing heuristic.
(The algorithm is stated in general terms; for specific interpretations see below)

Step 1. Set S to an initial tour.

Set $T = T_0$ (initial temperature)

Step 2. While not "frozen" do:

2.1 While not "equilibrium reached at this temperature" do:

2.1.1 Generate a neighbour S' of S

2.1.2 Let Δ = cost (S') - cost (S)

2.1.3 If $\Delta \leq 0$ ("downhill move")

set S = S'

2.1.4 If $\Delta > 0$ ("uphill move")

set S = S' with probability

$p = \exp(-\Delta / T)$

i.e. generate a $\in U(0,1)$ and

if a < p then accept S'

2.2 Set $T = T_{next}$ (reduce temperature)

(The result is a tour S.)

where

- temperatures follow an "annealing schedule" such as:

  (Golden et al, 1986) The interval (0,T) is divided into 25 subintervals of equal length such that

  $0 < t_1 < t_2 < \ldots < t_{25} = T$ and

  T_{next} = next lowest temperature.

  (Johnson et al, 1987) $T_{next} = rT$ for a cooling ratio

  $r \in (0,1)$, e.g. $r=0.95$

  Rossier, Troyon and Liebling (1986) present two annealing schedules, one of which is a geometrically decreasing sequence.

- "frozen"

  (Golden et al, 1986) The temperature is zero, i.e. after repeating 25 times.

  (Johnson et al, 1987) There were no improvements for the last 5 temperatures and the number of moves accepted at this temperature are less than some percentage (e.g. 2%) of the size of the problem.

  (Rossier et al, 1986) After repeating k times, where k depends on the annealing schedule used.

- "equilibrium reached at this level"

  (Golden et al, 1986) The percentage difference in tour lengths between the current tour and any one of the previous tours at this temperature, must be less than some $\varepsilon>0$, e.g. $\varepsilon=10\%$.

  (Johnson et al, 1987) Repeat L times, e.g. 16 times.

- neighbour

The most frequently used is a randomly generated 2-edge exchange (Golden et al, 1986, Johnson et al, 1987, Lundy and Mees, 1986 and Rossier et al, 1986), which requires the TSP costs to be symmetric. Rossier et al in addition suggest several other neighbourhoods which are labelled as alternately horizontal and vertical strips, square regions, and balls.

Simulated annealing has been used successfully on complicated problems, but has the disadvantages of being sensitive to the choice of parameters, and of requiring long runs to get the best results (Johnson et al, 1986). It is not possible to guarantee convergence of the algorithm in less than exponential time, although it is possible to adjust the parameters heuristically to almost always achieve polynomial time convergence (Lundy and Mees, 1986).

For the TSP, simulated annealing is one out of many heuristics available. Although it finds better results than for example the 2-opt edge exchange heuristic on a randomly generated starting solution (Rossier et al, 1987), it is clearly outperformed by a careful combination of tour construction and improvement techniques, both in the time required and the solutions found (Golden et al, 1986).

Rossier et al (1986) suggest that the worst case bound for the simulated annealing heuristic is of $O(kN^2)$, where k is the number of temperature steps used, when solving a Euclidean TSP.

3.4 Published comparisons

Comparisons of the worst case performances, as given in the previous sections, have been published by numerous authors (Rosenkrantz, Stearns and Lewis, 1977, Golden, Bodin, Doyle and Stewart, 1980, Christofides, 1982, Parker and Rardin, 1983).

Few comparative empirical tests of TSP heuristics have been published. Tests are usually performed on previously published problems, and on randomly generated problems. A comparative study by Golden et al (1980) lists the percentages above optimality of the solutions found by the nearest neighbour, the savings, the nearest, farthest and cheapest insertion heuristics, a convex hull variation, the Christofides and the k-opt heuristics. Computational results of several combinations of the nearest neighbour, convex hull and insertion heuristics with the k-opt heuristic for either k=2 or 3 or both are also given.

One conclusion from these results is that it is fairly easy to solve a TSP to within 3% of the (best known) solution by using a combination of tour construction and improvement heuristics, but no one combination performs better than every other combination. It is also not possible to conclude which tour construction technique is best with which improvement technique.

Repeating the above tests on symmetric problems without the triangle inequality results in an even greater variation in the performance of the heuristics than when the triangle inequality holds. Finally, if all the edges with cost greater than

$$\min_{ij} C + 0.6\,(\max_{ij} C - \min_{ij} C)$$

are removed from a problem, that is 40% of the more expensive or

'longer' edges are removed, the possibility of finding an expensive tour is reduced. Heuristics perform very well for these modified problems.

Another empirical study (Golden and Stewart, 1985) compares the above test results for the nearest neighbour, convex hull,insertion and k-opt heuristics with the cheapest angle heuristic combined with the Or-opt heuristic (the combination is called CCAO). The CCAO heuristic is shown to outperform the other combinations, and also outperforms the TSP heuristic based on simulated annealing (Golden and Skiscim, 1986).

4.1 Introduction

The heuristics described in Chapter 3 can be classified in terms of a number of features. In section 4.2, these features are listed, together with the possible variations of each. Each TSP heuristic is listed according to its particular characterization in terms of each feature.

The most important features characterizing heuristics are selected, and cross-classified. A cross-classification of two features is formed by tabulating all variations of one feature against all variations of the other feature. The cells formed by the cross-classification matrix are called "morphological" boxes. For any pair of features, each heuristic is entered into its morphological box, i.e. into the appropriate row and column. (Section 4.3)

An empty morphological box shows that no heuristic with this combination of variations of two features has been reported, which suggests a new possibility for a heuristic. The effects of variations in one feature can be compared by implementing all variations in this feature while leaving all others unchanged.

The gaps identified in the cross-classification tables are given in section 4.3, and the heuristics developed to fill some of these gaps are described in section 4.4.

4.2 Classification of the heuristics

In this section, a classification of the tour construction and

improvement heuristics is attempted. The heuristics described in the previous chapter will be referred to by the abbreviations of Tables 4.1 and 4.2.

The features of the heuristics are not entirely distinct, mutually exclusive aspects of TSP solution processes. A certain amount of overlap between some features is unavoidable. A heuristic has been classified according to its main characteristic, though a clear decision on which variation of a feature is the most appropriate to the heuristic can be difficult. Occasionally, more than one variation may apply strongly to one heuristic, for example if two subheuristics of a heuristic are very different.

The features will be labelled by A,B,C,... while the variations in each feature are numbered. Unless stated otherwise, these features are common to both tour construction and tour improvement techniques. Each discussion of a feature includes a table of the variations, with each heuristic listed under the appropriate variation.

Table 4.1 Abbreviations for the tour construction heuristics

abbreviation	heuristic and section where discussed	main characteristic of the heuristic
	3.2.2	
GREEDY	greedy	Select the cheapest edge.
SAVINGS	savings	A saving is made by combining two 'visits' (see Section 3.2.2). Select the edge with the largest saving.
LOSSES	loss	A loss is the cost incurred if the cheapest edge into a node is not selected. Include the edge causing the largest loss.
	3.2.3	
NEIGHBOUR	nearest neighbour	From the current node, go to its nearest neighbour.
DYN.WGHT	dynamic weighting	Evaluate both the cost of going to the next node and the effect this has on the solution.
SPACEFILL	spacefilling curve	Sequence nodes according to their order of appearance on the spacefilling curve.
	3.2.4	
NEAREST	nearest insertion (*1)	Insert the node nearest to the subtour.
FURTHEST	furthest insertion (*1)	Insert the node furthest from the subtour.
CHEAPEST	cheapest insertion (*1)	Insert a node into the subtour in the cheapest way possible.
DIFFERENCE	difference	Evaluate the differences between best and second best insertion places. Find the maximum difference.
ANGLE	greatest angle (*2)	Find the convex hull. Insert the node which forms the greatest angle.
ELLIPSE	eccentric ellipse (*2)	Find the convex hull. Insert the node which forms the most eccentric ellipse.
RATIOxDIFF	ratio-times-difference (*2)	Find the convex hull. Insert the node which minimizes the ratio-times-difference function.
CHEAP.ANGLE	cheapest angle (*2)	Find the convex hull. Insert the node which, at its cheapest insertion place, forms the greatest angle.
	3.2.5	
TOURMERGE	nearest merger	Merge the two closest subtours into one.
ASSIGNMENT	assignment and patching	Solve as an assignment problem. Patch all subtours together.
	3.2.6	
CHRISTOF	Christofides	Find the minimum spanning tree. Find the minimum matching. Shortcut past duplicate nodes.
PARTITION1	partition 1 (*3)	Partition the region such that there are nodes on the partition borders. Solve the TSP within each partition. Transform into a tour.
PARTITION2	partition 2 (*3)	Partition region such that there is a limited number of nodes in each partition. Solve the TSP within each partition. Transform into a tour.
STRIPS	strips (*3)	Partition region into squares. Solve the TSP within each partition. Sequence columns of squares, and link together.

(*1) simple insertion heuristics
(*2) convex hull heuristics
(*3) partitioning heuristics

Table 4.2 Abbreviations for tour improvement heuristics

abbreviation	heuristic and section where discussed	main characteristic of the heuristic
	3.3.1	
K-OPT	k-opt (*4)	Exchange k edges.
OR-OPT	Or-opt (*4)	Exchange edges of adjacent nodes.
VAR-OPT	variable k-opt (*4)	The number of edges exchanged depends on the edge under consideration.
	3.3.2	
SIM.ANN	simulated annealing	Heuristic based on simulating the annealing process.

(*4) edge exchange heuristics

A. Type of tour construction approach

A feature of the tour construction heuristics is the type of construction approach used, as described in Section 3.2.1. If the iterations of these heuristics terminate when a full tour is constructed, four different construction approaches are possible i.e. the ordered sequence, the increasing path, the subtour insertion, and the merged multiple subtours methods. A brief description of these four approaches follows, for more detail refer to Section 3.2.1.

The ordered sequence methods allocate a rank to each edge in the first iteration, and then in all further iterations select an edge according to its rank. The increasing path methods at each iteration add an edge onto one end of a path. At each iteration of the subtour insertion methods, a node is inserted into a subtour by replacing an edge in the subtour with two edges not in it. In the fourth construction approach, multiple subtours are constructed, and merged two at a time .

Tour construction heuristics may be a combination of several subheuristics. The solution of the intermediary subheuristics are not a tour. The word 'subheuristic' may be misleading as an exact method on a simplification of the TSP may be used, giving an

falls into first category (terminate when goal is reached) and the tour improvement approach falls into the last category (terminate when no more improvements can be found).

Table 4.3 Feature A
(for tour construction heuristics only)

A. Type of tour construction approach used	
1. Iterate until a full tour is constructed	
1.1 ordered sequences	GREEDY, SAVINGS, LOSSES
1.2 path	NEIGHBOUR, DYN.WGHT, SPACEFILL
1.3 subtour insertions	NEAREST, FARTHEST, CHEAPEST, DIFFERENCE, ANGLE, ELLIPSE, RATIOxDIFF, CHEAP.ANGLE
1.4 multiple subtours	TOURMERGE, ASSIGNMENT
2. two or more subheuristics are used	CHRISTOF, PARTITION1, PARTITION2, STRIPS
A2. If a heuristic is a combination of two or more subheuristics, two further classifications are possible.	
1. Sequence of subheuristics.	
(a) path	CHRISTOF, PARTITION1, PARTITION2, STRIPS
(b) others, e.g. cycle, random sequence, hierarchy	-
2. Termination rules for subheuristic.	
(a) continue until completion of subheuristic	CHRISTOF, PARTITION1, PARTITION2, STRIPS
(b) continue until failure to complete,	-
(c) continue until no more improvements found.	-

There is no feature equivalent to feature A, the type of tour construction technique used by tour construction heuristics, for the tour improvement heuristics, as these essentially only rearrange edges, stopping when a locally optimal tour is found.

B. Modification of the TSP

The TSP can be modified to take advantage of existing knowledge for the 'new' problem created. The results obtained for the modified problem are then used to find a solution to the original

TSP. Several of the heuristics, however, attempt the problem as it exists. These are the greedy, the neighbour, the nearest and cheapest insertion heuristics, the tourmerge and edge exchange heuristics.

If one of the constraints on the structure of the TSP is relaxed, a simplified problem results. If an algorithm for the simplified problem exists, this can be used to find an intermediary solution, which then has to be processed further to find a solution which satisfies all the original constraints. For example, relaxing the constraint that the tour must be connected, reduces the TSP to the assignment problem, for which efficient solution methods exist. If more than one subtour results when solving the related assignment problem, additional processing is necessary. Another example is relaxing the constraint that each node must be of at most degree two, reducing the TSP to the problem of finding a minimum spanning tree. The Christofides heuristic converts a minimum spanning tree into a tour.

For small sized problems, that is when there are only a few nodes, an exact method can be used. Thus if only a portion of the total number of nodes need to be visited at one time, the best route for these nodes can be found. There are several techniques that partition the amount of nodes or the region that the nodes are in, so that within these partitions an exact method or a good heuristic can be used.

The next two variations of feature B discussed do not, strictly speaking, involve a modification of the TSP, yet they do take advantage of knowledge from other fields or knowledge about the structure of the TSP to assist in finding the solution of the problem. One could view them as being modifications of a 'blind' solution approach.

If the TSP can be compared to an analogue model, existing expertise for this model can be used. For example, a spacefilling curve can indicate a possible tour of an area. Simulated annealing is an analogue model from the field of statistical mechanics.

A priori knowledge about the type of problem or of the problem structure is often available, and can be used when constructing a heuristic. Knowledge about the problem structure may also manifest itself in the type of consideration given to the implications of each decision made in the solution process.

In the savings heuristic, the reason for including an edge into a tour is based on the size of the saving generated by the inclusion the edge. The loss heuristic selects an edge based on the fact that if this edge is not included, then another must be taken in its place. The dynamic weighting heuristic takes into consideration that, as there are a finite number of nodes to be visited, each decision made reduces the alternatives of the subsequent decisions.

The farthest node insertion heuristic exploits the fact that as all nodes must be visited, those that are the 'furthest' away will be included in the tour first while it is still possible to select the cheapest way of visiting them. The difference heuristic finds the insertion place for a node that we feel most sure is the best insertion place possible.

The convex hull heuristics use the fact that there is an optimal tour of the TSP which if it is followed, then the nodes on the convex hull are visited in the same order as if the convex hull was followed. Initially the farthest insertion heuristic generates a subtour closely resembling a convex hull (Golden and Stewart, 1985).

Table 4.4 Feature B

B. Modification of the TSP	
1. none	GREEDY, NEIGHBOUR, NEAREST, CHEAPEST, TOURMERGE, K-OPT, OR-OPT, VAR-OPT
2. relaxation of constraint:	
2.1 no subtours allowed	ASSIGNMENT
2.2 of degree two	CHRISTOF
2.3 visit all nodes	PARTITION1, PARTITION2, STRIPS
3. analogue model	SPACEFILL, SIM.ANN
4. a priori knowledge	SAVINGS, LOSSES, DYN.WGHT, FARTHEST, DIFFERENCE, ANGLE, ELLIPSE, RATIOxDIFF, CHEAP.ANGLE

C. Dividing the TSP into subproblems

The TSP problem can be divided into k subproblems, which are solved individually, or it may be processed as a whole (k=1). Several tour construction heuristics (see Table 4.5, variation C1), and all tour improvements heuristics treat the problem as a whole.

At some stage of its processing, a heuristic may work with only part of the problem (k=2), i.e. with a subset of nodes or edges. For the convex hull heuristics, initially only the nodes on the hull are selected. The minimum matching algorithm in the Christofides heuristic only processes the nodes of odd degrees. Finally, the number of subproblems created can depend on the problem (k=unknown), as happens in the partitioning heuristics.

Table 4.5 Feature C

C. Division of the TSP into k subproblems	
1. k=1	GREEDY, SAVINGS, LOSSES, NEIGHBOUR, DYN.WGHT, SPACEFILL, NEAREST, FARTHEST, CHEAPEST, TOURMERGE, DIFFERENCE, ASSIGNMENT K-OPT, OR-OPT, VAR-OPT, SIM.ANN
2. k=2	ANGLE, ELLIPSE, RATIOxDIFF, CHEAP.ANGLE, CHRISTOF
3. k=unknown, function of the problem	PARTITION1, PARTITION2, STRIPS

The next features discussed characterize the steps performed in each iteration of a heuristic in terms of the evaluation and selection of candidates, as defined in Section 2.2.1. The candidates are the nodes and edges, respectively only the edges, on which decisions must be made by the tour construction and tour improvement heuristics.

At each iteration, a heuristic defines a set of potential candidates, evaluates the candidates, and based on the evaluations forms a set of selected candidates which form the input to the next iteration. Several approaches are possible. (Pearl, 1984, Muller-Merbach, 1981, 1976, 1976a)

D. Selection of potential candidates

The edges (and nodes) to be evaluated, that is the potential candidates, are selected by one or other criteria.

Formal criteria are those which do not take advantage of structural properties of a problem, and rely on for example a fixed ordering scheme. No iterative evaluations are made. The spacefilling curve heuristic orders nodes according to their positions on the spacefilling curve. Adjoining partitions are linked in the partitioning heuristics, ignoring problem structures such as clustering.

An a priori priority criterion orders the nodes or edges according to some preference scheme before any iterations are made. The order is not updated during the iterations. The greedy heuristic selects edges from cheapest to most expensive. In the savings heuristic, edges are selected according to the order of their saving made with respect to one of the nodes. Initially in the convex hull heuristics, only the nodes on the hull are

considered.

The number of potential candidates may be a function of the results in the previous iteration. The nearest neighbour and dynamic weighting heuristic proceed by adding one more edge at each iteration on to the end of a path. The number of edges (potential candidates) to be considered depends on which two nodes are currently the end nodes of the path. In the tourmerge and assignment heuristics, the potential candidates depend on the subtours remaining. The Christofides heuristic has been fitted into this category as the number of nodes processed by the minimum matching routine depends on the result of the minimum spanning algorithm, as does the number of nodes processed by the shortcutting routine.

A heuristic may make no pre-selection of the elements in the set of potential candidates, including all remaining candidates into this set. In the loss, the nearest, farthest, cheapest and the difference heuristic, all the remaining nodes are considered. All tour improvement heuristics fall into this class as every edge not in the tour is considered. (The Or-opt heuristic limits only the combination of the edges considered.)

Table 4.6 Feature D

D. The potential candidates are selected by	
1. formal criteria	SPACEFILL, PARTITION1, PARTITION2, STRIPS
2. a priori priority criteria	GREEDY, SAVINGS, ANGLE, ELLIPSE, RATIOxDIFF, CHEAP.ANGLE
3. function of previous results	NEIGHBOUR, DYN.WGHT, TOURMERGE, CHRISTOF, ASSIGNMENT
4. no pre-selection of potential candidates; all remaining candidates are considered	LOSSES, NEAREST, FARTHEST, CHEAPEST, DIFFERENCE K-OPT, OR-OPT, VAR-OPT, SIM.ANN

E. Number of potential candidates

Let k be the number of potential candidates (i.e. edges or nodes) evaluated at one iteration.

Particularly when a heuristic uses a formal ordering scheme, there is only one potential candidate (k=1) as no choice is involved. If the edges are assumed to be ordered according to their cost, then the potential candidate of the greedy heuristic is the next cheapest edge in the order. In the savings heuristic the potential candidate is the edge with the next smallest saving. The candidate of the spacefilling curve heuristic is the next node on the spacefilling curve. The strips heuristic links the partitions in a fixed pattern or "strip traversal", moving down one column of partitions, up the next, down the column after, and so on. The potential candidate is the edge from the current partition to the next partition in the strip traversal.

The number of potential candidates may be fixed (k>1, constant). The number may vary from problem to problem, but will remain fixed within a problem. The number of nodes within the partitions of the partitioning heuristics is fixed for each problem. The tour improvement heuristics have this feature, as every edge not in the

tour is considered and there is a fixed number of these edges.

The number of potential candidates may decrease as the solution progresses. For the loss heuristic there are less and less edges to choose from. For the tourmerge and the assignment heuristics there are a decreasing number of tours left to merge. The Christofides heuristic fits into this category (loosely) as both the minimum spanning tree and minimum matching subheuristics have less options left as they progress. For the simple insertion heuristics and the convex hull heuristics there is a decreasing number of nodes to choose from.

The number of potential candidates may increase with solution progress. For the simple insertion heuristics and the convex hull heuristics there are an increasing number of places in the subtour where an insertion may occur.

The number of potential candidates may vary, depending on a previous result. For the nearest neighbour and dynamic weighting heuristics the number of potential candidates depends on the number of edges connected to the node currently being considered. If it is assumed that every node is connected to every other node, then the nearest neighbour and dynamic weighting heuristics fall in the category of decreasing number of potential candidates.

Table 4.7 Feature E

E. There are k potential candidates, where	
1. k=1	GREEDY, SAVINGS, SPACEFILL. STRIPS when linking partitions.
2. k>1, constant	PARTITION1, PARTITION2. STRIPS within partitions. K-OPT, OR-OPT, VAR-OPT, SIM.ANN
3. k decreases with solution progress	LOSSES, TOURMERGE, CHRISTOF, ASSIGNMENT. {NEAREST, CHEAPEST, FARTHEST, DIFFERENCE, ANGLE, ELLIPSE, RATIOxDIFF, CHEAP.ANGLE} as there are less nodes to choose from.
4. k increases with solution progress	{NEAREST, FARTHEST, CHEAPEST, DIFFERENCE, ANGLE, ELLIPSE, RATIOxDIFF, CHEAP.ANGLE} as there are more insertion places to choose from.
5. variable, depending on previous result	NEIGHBOUR, DYN.WGHT

F. Construction of a new intermediary solution

The intermediary solution may be constructed by one of the techniques below. The first two discussed are only applicable to the tour construction heuristics.

In the construction technique called "append", a selected candidate is added to the end of the current subsolution, forming an increasing sequence such as A, AD, ADB, ADBE, ... This category contains all of the heuristics which form a path i.e. the nearest neighbour, the dynamic weighting and the spacefilling curve heuristics. In the partitioning heuristics "append" is used when linking neighbouring partitions. The technique can be used to construct the convex hull in the convex hull heuristics.

The "insertion" technique breaks the sequence of nodes or edges in the subsolution, for example ABCA, ABDCA, ABDCEA, ... The simple insertion heuristics and the difference heuristic use this approach. Insertion techniques are used in the convex hull heuristics to find a tour from the convex hull.

A heuristic can "replace" candidates in a partial solution with others not in the solution, e.g. ABEDA, ACEDA, ACEBA, ... The tourmerge and the assignment heuristics replace edges in partial tours with edges not in these tours. The insertion techniques in fact also fit into this category. In a complete solution a heuristic can "exchange" candidates, e.g. ABCDEA, ACBDEA, AECBDA, ..., as is done by the tour improvement heuristics.

Other techniques are possible. The greedy, savings and loss heuristics select edges according to a specific order. The techniques of the minimum spanning tree and the minimum matching subheuristics of the Christofides heuristic depend on the algorithms used.

Table 4.8 Feature F

F. New solutions constructed by 1. append (tour construction heuristics only) NEIGHBOUR, DYN.WGHT, SPACEFILL, PARTITION1, PARTITION2, STRIPS. {ANGLE, ELLIPSE, RATIOxDIFF, CHEAP.ANGLE} in convex hull construction. 2. insert (tour construction heuristics only) NEAREST, FARTHEST, CHEAPEST, DIFFERENCE. {ANGLE, ELLIPSE, RATIOxDIFF, CHEAP.ANGLE} after convex hull found. 3. replace or exchange TOURMERGE, ASSIGNMENT K-OPT, OR-OPT, VAR-OPT, SIM.ANN 4. other (tour construction heuristics only) GREEDY, SAVINGS, LOSSES, CHRISTOF

G. Types of evaluations

Several types of evaluations of potential candidates are possible.

Comparisons can be made of simple priority criteria such as the cost of an edge, but not the full cost of the current subsolution. The greedy and the neighbour heuristics are examples of heuristics which evaluate the cost of a single edge. The loss, the nearest

node, the tourmerge and the difference heuristics evaluate functions of the costs of several edges. The convex hull heuristics use this approach to find a tour from a convex hull subtour. The tour improvement heuristics evaluate the cost of exchanging two or more edges.

An evaluation in a heuristic may be based on the value of the current subsolution, which includes the total of all costs so far plus the cost added by the inclusion of the next candidate. The cheapest heuristic inserts a node in the cheapest possible way. As the cost of a tour is the sum of the cost of the edges in the tour, evaluating for example the effect of exchanging two edges in the tour with edges not in the tour by subtracting the cost of the former from and adding the cost of the latter to the tour cost, is equivalent to only evaluating the sum of the new costs less the old costs. Thus some of the heuristics in the first category of feature G could also fit into this category.

A look ahead technique or 'not better than' criterion, i.e. a lower bound, can be used to estimate the solution. Dynamic weighting evaluates a lower bound at every iteration. Solving the assignment problem gives a lower bound of the TSP. In the partitioning heuristics let all the nodes in a partition be represented by a single node, for example the center of the partition. Connecting these single nodes into a tour gives a lower bound on the TSP tour.

A tour of the problem is 'not better than' the convex hull subtour of the problem found by the convex hull heuristics. The Christofides heuristic finds a minimum spanning tree technique, which is a lower bound of the TSP.

An estimate of 'not worse than' or upper bounds can for example

be used by another heuristic which improves on the bound, as in the Christofides heuristic where the minimum matching is improved on by the shortcutting subheuristic. The calculation of the savings for an initial node in the savings heuristic, can be seen as estimating the 'worst case scenario' when each edge must be visited and after each edge the initial node must be revisited. In the furthest heuristic, selecting the node which is furthest away from the subtour implies that the worst edge is selected.

If no computing facilities are available, a heuristic could be formulated so as to make no formal evaluations, such as is done in the spacefilling curve heuristic.

Table 4.9 Feature G

G. Evaluate potential candidates by 1. priority criteria GREEDY, LOSSES, NEIGHBOUR, NEAREST, TOURMERGE, DIFFERENCE. {ANGLE, ELLIPSE, RATIOxDIFF, CHEAP.ANGLE} after convex hull found. K-OPT, OR-OPT, VAR-OPT, SIM.ANN 2. value of objective function of solution in progress CHEAPEST 3. (look ahead criteria) 'not better than' criteria; lower bounds DYN.WGHT, ASSIGNMENT, PARTITION1, PARTITION2, STRIPS. {ANGLE, ELLIPSE, RATIOxDIFF, CHEAP.ANGLE} in convex hull construction. CHRISTOF when finding minimum spanning tree. 4. (look ahead criteria) 'not worse than' criteria; upper bounds SAVINGS, FARTHEST. CHRISTOF when finding minimum matching. 5. no formal evaluation SPACEFILL

H. Types of rules to determine selected candidates

Based on the results of the evaluations, candidates are selected as input for the next iteration. Two types of rules to determine the selected candidates are possible.

All tour construction heuristics and the edge exchange tour

improvement heuristics have deterministic rules. Only one heuristic has stochastic rules, the simulated annealing heuristic. This results in the TSP solutions having an element of probability to them (they cannot be replicated).

Table 4.10 Feature H

H. Type of rules to determine sets of selected candidates	
1. deterministic	all tour construction heuristics K-OPT, OR-OPT, VAR-OPT
2. stochastic	SIM.ANN

I. Decisions are or are not revisable

It may or may not be possible to revise a decision made about whether or not an edge is to be included in the solution.

The decision made may not be revisable, i.e. once an edge has been selected, it will be in the tour. The ordered sequence approaches (greedy, savings and loss heuristics) and the path approaches (neighbour, dynamic weighting and spacefilling curve heuristics) make irrevocable decisions.

The decisions made may be revisable. Although a node remains selected in all the insertion heuristics, the selected edges are updated in each iteration. For the heuristics which use more than one subheuristic, one of the subheuristic may proceed by making irrevocable decisions, but some edges may need to be exchanged by another subheuristic to convert the intermediary results into a tour. The Christofides heuristic exchanges edges in the shortcutting subheuristic. The assignment heuristic, similar to the tourmerge heuristic, replaces edges in the subtours by edges not in the subtours. The partitioning heuristics exchange edges when linking the tours in the partitions.

All the tour improvement heuristics exchange edges that have already been selected with edges not in the tour.

Table 4.11 Feature I

I. The decisions made about whether or not to include an edge is	
1. not revisable	GREEDY, SAVINGS, LOSSES, NEIGHBOUR, DYN.WGHT, SPACEFILL.
2. revisable	NEAREST, FARTHEST, CHEAPEST, TOURMERGE, DIFFERENCE, ANGLE, ELLIPSE, RATIOxDIFF, CHEAP.ANGLE, CHRISTOF, ASSIGNMENT, PARTITION1, PARTITION2, STRIPS K-OPT, OR-OPT, VAR-OPT, SIM.ANN

J. The number of selected candidates

Let k represent the number of selected candidates, i.e. k edges (or nodes) are added to the solution in progress at each iteration.

In some algorithms, only one node or edge may be selected at each iteration (k=1). Note that this category cannot apply to the tour improvement heuristics.

In the ordered sequence approaches (greedy, savings and loss heuristics) only one edge is selected. In the path approaches (neighbour, dynamic weight and spacefilling curve heuristics), one node and one edge is selected at each iteration. In all the insertion approaches (the simple insertion, the difference and the convex hull heuristics), one node is added to the subtour.

The subheuristics of the Christofides heuristic select one edge (and in certain minimum spanning tree algorithms one node) per iteration. The partitioning heuristics combine partitions by adding an edge between adjacent partitions.

The number of edges (or nodes) selected can be set to a constant

value greater than one. All the insertion approaches (the simple insertion, the difference and the convex hull heuristics) add two new edges to the solution at each iteration. The tourmerge and assignment heuristics link subtours by two (new) edges. Of the tour improvement heuristics, the k-opt and Or-opt heuristics select a fixed number of edges.

The number of selected nodes or edges may be a function of the number of times the heuristic has been repeated. For example, at the first iteration select k edges, at the second iteration select k-1 edges, at the third k-2, and so on.

The number of edges selected may depend on the results of the evaluations. In the ordered sequence approaches (greedy, savings and loss heuristics), both, one or neither of the nodes at the end of the selected edge may already be in the solution. The result of the evaluations determines whether two or more edges are exchanged in the variable k-opt tour improvement heuristic.

None of the above values of k may be applicable. The number of nodes selected at each iteration of the assignment problem solution technique depends on which algorithm is used. When partitioning, the number of nodes in a region are user specified. In the simulated annealing heuristic, the number of edges accepted depends on the 'neighbour' used (see the description of the algorithm in Section 3.3.2).

Table 4.12 Feature J

J. There are k selected candidates, where	
1. k=1	GREEDY(e), SAVINGS(e), LOSSES(e), NEIGHBOUR, DYN.WGHT SPACEFILL, NEAREST(n), FARTHEST(n), CHEAPEST(n), DIFFERENCE(n), ANGLE(n), ELLIPSE(n), RATIOxDIFF(n), CHEAP.ANGLE(n), CHRISTOF. {PARTITION1, PARTITION2, STRIPS} when combining partitions.
2. k>1, constant	NEAREST(e), FARTHEST(e), CHEAPEST(e), TOURMERGE, DIFFERENCE(e), ANGLE(e), ELLIPSE(e), RATIOxDIFF(e), CHEAP.ANGLE(e), ASSIGNMENT(e), K-OPT, OR-OPT
3. k=function of level of iteration	-
4. k=function of evaluation	GREEDY(n), SAVINGS(n), LOSSES(n), VAR-OPT
5. other	ASSIGNMENT(n) {PARTITION1, PARTITION2, STRIPS} when partitioning. SIM.ANN
where (n) = the number of nodes (e) = the number of edges	

4.3 Cross classifications

The previous section introduced several features of the TSP heuristics. The same notation as in that section will be used here. Three of these features are argued to be the most useful for the design of new heuristics. For each pairwise combination of the three features, morphological boxes are compiled. Finally the gaps in the boxes are discussed.

Muller-Merbach (1981) emphasized that, when designing a new heuristic, great care must be taken when deciding on:

- which candidates are chosen for evaluation, i.e. which are the potential candidates,
- the type of evaluations performed, and
- how candidates are selected to be included in the solution, i.e. how the selected candidates are chosen.

In terms of the notation of the previous section, choosing the potential candidates is described by features D, E and F, the types of evaluations by features G and H, and choosing the selected candidates by features J. This gave a shortlist of features to use in the morphological studies, and further investigation cut these down to one feature for each of Muller-Merbach's points.

Of the features D, E and F, the choice of a specific variation of feature F for a heuristic determines to a large extent which variation of features D and E will be apllicable, as the type of construction used to find a solution (F) determines the number of potential candidates (E) and how they are selected (D). We therefore selected F as being the most important feature for describing the choice of potential candidates. Of the features G and H, evaluations were selected to be deterministic (H) for this study, and feature G was included in the construction of morphological boxes.

Thus the three features which appear to be most important for the design of a heuristic are F, G and J, where:

F - New solutions are constructed by appending, inserting or exchanging candidates, or some other approach.

G - The evaluations of the potential candidates use priority criteria, the value of the solution in progress, or lower or upper bounds, or no formal evaluations are performed.

J - The number of selected candidates is fixed at one or more, or is a function of the level of iteration or evaluation, or is based on some other criterion.

The features F, G and J are used to form morphological boxes or cross-classifications. Tables 4.5 to 4.7 show the cross-classifications for the tour construction heuristics. They are

followed by a discussion of the uses of the tables for designing tour construction heuristics. Tables 4.8 to 4.10 show the same for the tour improvement heuristics.

Table 4.13 Cross-classification of features G and F for tour construction heuristics.

G \ F	1 append	2 insert	3 exchange	4 other
1 priority	NEIGHBOUR	NEAREST ANGLE(b) ELLIPSE(b) RATIOxDIFF(b) CHEAP.ANGLE(b)	TOURMERGE	GREEDY LOSSES
2 value of solution	*1	CHEAPEST	*2	-
3 lower bound	DYN.WGHT ANGLE(a) ELLIPSE(a) RATIOxDIFF(a) CHEAP.ANGLE(a) PARTITION1 PARTITION2 STRIPS	DIFFERENCE	ASSIGNMENT	CHRISTOF(
4 upper bound	*3	FARTHEST	*4	SAVINGS CHRISTOF(
5 no calculation	SPACEFILL	-	-	-

where for ANGLE, ELLIPSE, RATIOxDIFF, CHEAP.ANGLE:
(a) = when constructing convex hull
(b) = after hull has been found,
and for CHRISTOF: (a) = minimum spanning tree subheuristic
(b) = minimum matching subheuristic.
The *'s denote the gaps, and the dashes indicate gaps which are of secondary interest only (see text following).

Table 4.14 Cross-classification of features G and J for tour construction heuristics.

G \ J	1 k=1	2 k>1,const.	3 k= f(iteratn)	4 k=f(evaluatn)	5 other
1 priority	GREEDY(e) LOSSES(e) NEIGHBOUR NEAREST(n) DIFFERENCE(n) ANGLE(n) ELLIPSE(n) RATIOxDIFF(n) CHEAP.ANGLE(n)	NEAREST(e) TOURMERGE DIFFERENCE(e) ANGLE(e) ELLIPSE(e) RATIOxDIFF(e) CHEAP.ANGLE(e)	*1	GREEDY(n) LOSSES(n)	-
2 value of solution	CHEAPEST(n)	CHEAPEST(e)	*2	*3	-
3 lower bound	DYN.WGHT ANGLE(a) ELLIPSE(a) RATIOxDIFF(a) CHEAP.ANGLE(a) CHRISTOF(a) PARTITION1(e) PARTITION2(e) STRIPS(e)	ASSIGNMENT(e)	*4	*5	ASSIGNMENT(PARTITION1(PARTITION2(STRIPS(n)
4 upper bound	SAVINGS(e) FARTHEST(n) CHRISTOF(b)	FARTHEST(e)	*6	SAVINGS(n)	-
5 no calculatn	SPACEFILL	-	-	-	-

where (n) indicates the number of nodes
(e) indicates the number of edges
for any heuristic,
for ANGLE, ELLIPSE, RATIOxDIFF, CHEAP.ANGLE:
(a) = when constructing convex hull,
and for CHRISTOF: (a) = minimum spanning tree subheuristic
(b) = minimum matching subheuristic.
The *'s denote the gaps, and the dashes indicate gaps which are of secondary interest only (see text following).

Table 4.15 Cross-classification of features J and F for tour construction heuristics.

J \ F	1 append	2 insert	3 exchange	4 other
1 k=1	NEIGHBOUR DYN.WGHT SPACEFILL ANGLE(a) ELLIPSE(a) RATIOxDIFF(a) CHEAP.ANGLE(a) PARTITION1(e) PARTITION2(e) STRIPS(e)	NEAREST(n) FARTHEST(n) CHEAPEST(n) DIFFERENCE(n) ANGLE(n) ELLIPSE(n) RATIOxDIFF(n) CHEAP.ANGLE(n)	*1	GREEDY(e SAVINGS(e LOSSES(e CHRISTOF
2 k>1, constant	*2	NEAREST(e) FARTHEST(e) CHEAPEST(e) DIFFERENCE(e) ANGLE(e) ELLIPSE(e) RATIOxDIFF(e) CHEAP.ANGLE(e)	TOURMERGE ASSIGNMENT(e)	-
3 k=function of iteration	*3	*4	*5	-
4 k=function of evaluation	*6	*7	*8	GREEDY(r SAVINGS(r LOSSES(r
5 other	PARTITION1(n) PARTITION2(n) STRIPS(n)	-	ASSIGNMENT(n)	-

where (n) indicates the number of nodes
(e) indicates the number of edges
for any heuristic,
and for ANGLE, ELLIPSE, RATIOxDIFF, CHEAP.ANGLE:
(a) = when constructing convex hull.
The *'s denote the gaps, and the dashes indicate gaps which are of secondary interest only (see text following).

The gaps in the morphological boxes marked by numbered stars, indicate combinations of the most important features that do not occur in any of the previously published TSP heuristics. By incorporating the specific features of a gap into a heuristic, a new design is achieved. The gaps in the boxes marked by dashes, which involve non-specific characteristics or no calculations, are of limited use in the design of new heuristics because the characteristics are not specific enough to give a clear picture of a new heuristic.

For the tour construction heuristics, 5 starred gaps in the G\F cross-classification are marked, 6 in the G\J and 8 in the J\F

cross-classification. Four of these gaps are not very distinct, as for example choosing the cheapest remaining edge is equivalent to choosing the remaining edge which minimizes the sum of the costs of all selected edges added to the cost of this edge. The gaps affected are all those which involve feature G2, i.e. gaps *1 and *2 in G\F, and *2 and *3 in G\J.

Five new heuristics were eveantually selected out of these gaps in the cross-classifications. They are discussed in detail in the next section, but brief mention is made here of which gaps they were designed to fill.

A gap in G\F for which a new heuristic was developed is *3. Using an upper bound in the evaluations (G4) is combined with a solution construction which appends candidates (F1) i.e. which uses a path construction approach. The heuristic is named the "dynamic weighting variation".

The next new heuristic uses convex hulls to give a series of lower bounds when evaluating (G3). This feature is combined with the number of selected candidates being a function of evaluation (J4). The heuristic fits gap *5 in cross-classification G\J and is called "peeled convex hull".

The cross-classification J\F suggested a number of new heuristics. Combining the append or path construction approach (F1) with selecting a constant, greater than one number of candidates (J2) forms the basis of the "k-node look-ahead" heuristic. This heuristic fits gap *2 in the cross-classification. Two other new heuristics append (F1) and insert (F2) a number of candidates that is a function of evaluating the minimum spanning tree of the remaining nodes (J4). They fill the gaps *6 and *7, and are both described in section 4.4.4.

For the sake of completeness, the morphological boxes for the tour construction heuristics are given below.

Table 4.16 Cross-classification of features G and F for tour improvement heuristics.

G \ F	exchange	other
priority	K-OPT OR-OPT VAR-OPT SIM.ANN	-
obj. function	*1	-
'not better'	*2	-
'not worse'	*3	-
no evaluation	-	-

Table 4.17 Cross-classification of features G and J for tour improvement heuristics.

G \ J	k=1	k>1, const	k=f(iteratn)	k=f(evaluatn)	oth
priority	*1	K-OPT OR-OPT	*2	EXCH2	SIM.A
obj. funct.	*3	*4	*5	*6	-
not better	*7	*8	*9	*10	-
not worse	*11	*12	*13	*14	-
no evaluatn	-	-	-	-	-

Table 4.18 Cross-classification of features J and F for tour improvement heuristics.

J \ F	exchange	other
k>1, constant	K-OPT OR-OPT	-
k=f(iteration)	*1	-
k=f(evaluatn)	VAR-OPT	-
other	SIM.ANN	-

The above statement regarding the variation 2 of feature G also applies to the tour improvement heuristics. The gap *1 of cross-classification G\F, and the gaps *3 to *6 of cross-

classification G\J are therefore not very distinct gaps. No new tour improvement heuristics were thus attempted.

4.4 New heuristics

4.4.1 Dynamic weighting variation

The dynamic weighting variation heuristic is based on both the dynamic weighting and the farthest insertion heuristic. The algorithm is the same as the dynamic weighting one, but insted of calculating a lower bound on the expected solution given that a next node is selected, an upper bound is used. At each iteration the node with the lowest upper bound is added to the solution.

The classification of the heuristic is:

F1 - The solution is constructed by appending an edge to a path.

G4 - An upper bound is used in the evaluations.

I1 - One edge (and node) at a time is added to the solution.

Algorithm.

Repeat for each node as the initial node:

Step 1. Select a node to be the initial node of the path.
(This node also represents the two end nodes of the path for the first iteration of the next step.)

Step 2. While there are nodes not in the path do:

2.1 For each i node not in the path find an upper bound U(i) of the TSP solution which passes through node i and all other remaining nodes, and compute

$$f(i) = W_1 C_{si} + W_2 U(i)$$

where the W's are the weighting functions used in the dynamic weighting heuristic, and s is an end node of

the path.

2.2 Select the node which minimizes f(i). Add it to the end of the path. (The path now ends at this node.)

Step 3. Join the two end nodes of the path.

If a lower bound which runs in time $O(n^2)$ is used, the algorithm has a running time of $O(n^4)$.

4.4.2 Peeled convex hull

A convex hull gives a fast, although not very good, lower bound for the TSP. If after a convex hull is found, the nodes on this hull are deleted from the problem, then another convex hull can be found, its nodes deleted, and so on. The assigning of nodes to a series of hulls terminates when all nodes have been assigned. The hulls found form a series of subtours, one within the next. The effect is like peeling an onion; each onion skin has another within it. This gives the heuristic its name.

The classification of the heuristic is:

G3 - Each convex hull found is a lower bound.

J4 - The number of nodes (and edges) added to the solution by each convex hull is a function of the evaluations when finding the hull.

F3 - The series of convex hulls, or subtours, are joined by replacing edges in the subtours by edges not in the subtours.

Algorithm.

Step 1. Repeat the following until all have been assigned to a

convex hull:

1.1 Find the convex hull of the problem, by for example using the algorithm of the convex hull insertion heuristics.

1.2 Form a smaller subproblem by deleting the nodes on the hull from the problem.

Step 2. Repeat the following until all hulls or subtours have been joined:

(Each subtour is joined to the first subtour enclosing it and the first subtour it encloses.)

1.1 Replace edge (i',j') in one subtour and edge (k',l') in another subtour with edges (i',k') and (j',l') not in the subtours, where these edges are choosen to minimize

$$C_{ik} + C_{jl} - C_{ij} - C_{kl}$$

If the running time of the convex hull algorithm used does not exceed $O(n^2)$, then the algorithm of the peeled convex hull heuristic runs in time of $O(n^2)$.

4.4.3 k-node look-ahead

The k-node look-ahead heuristic is an extension of the nearest neighbour heuristic, the simplest of the path heuristic. Instead of selecting only one node at a time, two or more are choosen at each iteration. The look-ahead part of the heuristic involves selecting not just the k-nodes to give a minimum addition of costs to the solution, but also ensuring that the node chosen after the k-th node will also lead to a minimum path, i.e.

minimize the cost of adding k+1 nodes, but add only k nodes to the path.

The classification of the heuristic is:

F1 - The solution is constructed by appending an edge to a path.

J1 - A constant, greater than one number of candidates is appended.

G3 - Minimizing over k+1 candidates, while only k candidates are selected, gives a rough estimate of the effect of adding the k candidates to the solution.

Algorithm.

Repeat until all nodes have been used as starting nodes:

Step 1. Start with any node. This is the first node in the path.

Step 2. Repeat until all nodes are in the path:

2.1 For node i an end node of the path, find nodes

$$j_1, j_2, \ldots, j_k, j_{k+1}$$

not on the path such that

$$C_{ij_1} + C_{j_1 j_2} + \ldots + C_{k,k+1}$$

is minimized.

2.2 Add nodes $j_1, j_2, \ldots, j_k$ to the end of the path.

Step 3. Join the first and last nodes of the path.

The algorithm has a running time of $O(n^{k+2})$.

4.4.4 Using minimum spanning trees

The minimum spanning tree of a set of nodes and edges can be found by fast algorithms. In additon, it gives a lower bound for the minimum tour of the nodes. This technique has not been used much in solving the TSP: only the Christofides heuristic makes explicit use of it. Two heuristics using minimum spanning trees (MST's) are proposed, the MST path heuristic and the MST subtour heuristic.

Both methods find the minimum spanning tree of the nodes not yet in the solution. In the MST path heuristic, the end nodes of the path are also included in the minimum spanning tree. This heuristic adds to the solution all the nodes in the minimum spanning tree which connected the path end nodes to one another. The resulting subtour is reset to a path by deleting its longest (most expensive) edge. This procedure is repeated until all nodes have been included into a tour.

The MST subtour heuristic finds the path in the minimum spanning tree which contributes the 'most' for this iteration. The total extra cost of including a specific path is measued in proportion to the number of nodes on this path, as follows. Find the edge (i',j') in the subtour, and the nodes k' and l' of degree 1 in the minimum spanning tree (i.e. end nodes), such that

$$\frac{C_{ik} + \text{cost}(k,l) + C_{lj} - C_{ij}}{f(\text{number of nodes on path from k to l})} \qquad (1)$$

is minimized, where

cost (k,l) is the total cost of getting from node k to node l by the minimum spanning tree, and

f(number of nodes), letting M be the number of nodes on the path from k to 1, is set to

$$M \text{ and } M^2.$$

The classifications of the heuristics are:

MST path heuristic.

F1 - The edges selected in each iteration are appended onto the end of a path.

J4 - The number of nodes and edges to be added to the solution is a function of the evaluations performed on the minimum spanning tree found.

G3 - The minimum spanning tree is a lower bound of the minimum tour.

MST subtour heuristic.

F1 - The edges selected in each iteration are inserted into a subtour.

J4 and G3 are the same as for the MST path heuristic.

Algorithm of the MST path heuristic.

Step 1. Find the cheapest edge. This edge is the initial path.

Step 2. Repeat until a tour is found:

2.1 Find the minimum spanning tree of all the nodes not in the solution and the two end nodes of the path .

2.2 Add to the solution all the nodes in the minimum spanning tree which connected the end nodes to one another.

2.3 If the result of the step 1.2 is a subtour, i.e. not all nodes are in the solution, form a path by deleting edge (i',j') in this subtour which minimizes

$$C_{ik} + C_{jl} - C_{ij}$$

where k and l are any two nodes not in the subtour,

i.e. delete the edge with the lowest replacement cost.

Algorithm of the MST subtour heuristic.

Repeat for each node as the initial subtour:

Step 1. Let one node be the initial subtour.

Step 2. Repeat until a tour is found:

2.1 Find the minimum spanning tree of the nodes not in the subtour.

2.2 Find the path which minimizes function (1) above for some edge (i',j') in the subtour, and the nodes k' and l' in the minimum spanning tree.

2.3 Insert the path into the subtour by deleting edge (i',j') and including edges (i',k'), (j',l').

A minimum spanning tree of a set of n nodes can be found in time $O(n^2)$. The MST path heuristic runs in time $O(n^3)$, and the running time of the MST subtour heuristic is of $O(n^2)$.

5.1 Evaluating heuristics

A discussion of a range of heuristics is not complete without an assessment of the quality of the heuristics. There are various ways of doing this.

One way of evaluating a set of heuristics is to compare their worst case behaviour, i.e. the ratio

$$\frac{\text{worst heuristic solution}}{\text{optimal solution}}$$

The published worst case results were included in the discussion of the TSP heuristics (Chapter 3), and have been repeated in Table 5.1.

Table 5.1 Known worst case performances of TSP heuristics

heuristic	section	worst case behaviour
TOUR CONSTRUCTION: nearest neighbour	3.2.3	$.5 \lceil \lg n \rceil + .5$
spacefilling curve	3.2.3	$n \ln n$
nearest insertion	3.2.4	2
furthest insertion	3.2.4	$2 \ln n + .16$ but probably closer to 1.5
cheapest insertion	3.2.4	2
assignment and patching	3.2.5	$(v_A + v_I) / v_I$ where v_A is the cost of the assignment problem solution and v_I is the cost of the insertions.
Christofides	3.2.6	1.5
strips	3.2.6	n
TOUR IMPROVEMENT: k-opt	3.3.1	$2(1 - 1/n)$ when $n \geq 8$, $k \leq n/4$
simulated annealing	3.3.2	$O(kN^2)$ where k is the number of temperature steps used.

A second, less common way of evaluating heuristics, is to use probabilistic results, that is to attempt to establish

theoretical descriptions of the most probable behaviour of a heuristic. Such a description may, for example, state that for large n, the optimal and heuristic solutions are very close with high probability. For example, Karp and Steele (1985) showed that if T is the cost of the optimal tour of a problem in the unit square ([0,1]x[0,1]), and T' is the the cost of the tour found by the assignment and patching heuristic for this problem, then

$$E\left[\frac{T' - T}{T}\right] = O(n^{-.5})$$

Thus this heuristic gives near-optimal tours for very large n. Results of probabilistic analyses have also been published for partitioning heuristics (See Karp, 1985, for example).

The final type of comparison is to subject the heuristics to empirical tests. This involves running the heuristics on a number of problems selected to be representative of the types of problems for which the heuristics are intended. These problems may be real problems, or hypothesized randomly generated problems.

TSP problems may be generated by randomly selecting nodes in an area, which implies randomly selecting (x,y) or (x,y,z) coordinates where each x, y and z value is within some specified interval. For example, a TSP on the unit square has coordinates $\{(x,y): 0 \leq x \leq 1, 0 \leq y \leq 1\}$. The edge costs are calculated using a distance metric such as the Euclidean distance function (defined in Section 2.3.1). The metric used determines whether the problem is symmetric or not, and whether the triangle inequality holds. TSP problems may also be generated by randomly selecting edge costs from some interval. For these problems the triangle inequality does not hold. The costs may be selected such that the problem is symmetric or asymmetric.

There are disadvantages to testing heuristics on only randomly

generated problems or only real problems (Crowder, Dembo and Mulvey, 1978). The disadvantages of randomly generated problems is that these problems are not necessarily representative of real world situations. For example, the cities of a country, or the pins on the modules of computer wiring problems, may not be completely randomly distributed. A disadvantage of real problems is that often the population from which the problems are drawn is not known, and generalizations based on the sample cannot be made. A distribution such as the uniform distribution on the interval (a,b), can be selected when generating a problem, i.e. the coordinates of the nodes are generated uniformly on (a,b).

The results of the empirical tests may be summarized in terms of the time required and the quality of the solution found for a set of problems, where the former is the running time for a particular size of TSP, while the latter is usually expressed as the percentage by which a heuristic solution exceeds the best known solution of the problem (the best known solution may be the true optimum of the problem). Heuristics may be compared on the basis of results from many problems and, if none of the sets of results dominates all others, one of the following techiniques may be used to determine which performance, if any, is significantly better than all other performances:

- simple statistics,
- non-parametric statistical tests, and
- utility measures.

Simple statistics include the number of times a heuristic finds the best or the worst solution, out of all the heuristics run on the same problem. The mean and variance of the percentages above best known solutions for each heuristic run on a number of problems may be calculated. The solution found by a heuristic may be ranked relative to all other heuristic solutions for the same

problem, and for each heuristic an average of all its ranks may be calculated.

Non-parametric statistical tests may be used to indicate whether there are any significant differences in the performance of two or more heuristics, if the performance of a heuristic does not clearly dominate that of the other heuristics. For these tests, the measure of performance, i.e. the percentage above the best known solution, is replaced by ranks, and the difference in ranks is tested. To test for significant difference between the sets of ranks of two heuristics, the Wilcoxon signed rank test or the sign test can be used. To compare the sets of ranks of three or more heuristics, the Friedman test should be used. Golden and Stewart (1985) give a detailed description of how these tests are performed.

An alternative to statistical comparisons is the expected utility approach of Golden and Assad (1984). This approach is based on determining which heuristic performs well on average and very rarely performs badly. The expected utilities are found by the following steps:

Step 1. For each heuristic, fit a gamma distribution to the frequency histogram of the percentage deviations of the heuristic solutions from the best known solutions. Using the density function

$$x^{c-1} e^{-x/b} / (b^c T(c)) \quad \text{where } T(c) = \int_0^\infty e^{-u} u^{c-1} du$$

the parameters of the distribution are estimated as:

$$\hat{b} = s^2 / x \qquad \text{and} \qquad \hat{c} = (x / s)^2$$

where x is the mean and s the standard deviation of

the percentage deviations. The gamma distribution is chosen for its computational convenience.

Step 2. Select a decreasing utility function such as

$$u(x) = v - w e^{tx} \quad \text{where } v, w, t > 0.$$

The constants v and w are chosen arbitrarily (Golden and Assad use v=600 and w=100). The value selected for t gives a measure of the aversion to a poor performance of a heuristic (Golden and Assad use t=0.05).

Step 3. Calculate the expected utility for each heuristic.

$$v - w (1 - bt)^{-c}$$

The heuristic with the largest utility value is preferred.

Three ways of evaluating heuristics have been discussed: comparing the worst case performance, comparing probabilistic results, and performing empirical tests. The disadvantage of the first type of comparison is that in practice heuristics tend to behave better than is suggested by their worst case performance (Rinnooy Kan, 1984). Probabilistic analysis has the disadvantage that assumptions must be made about the probability distribution of the problems (Johnson and Papadimitriou, 1985). The empirical approach has been chosen for this study because it, if performed correctly, can reflect the behaviour of a heuristic across the spectrum of practical situations (Fisher ,1980).

5.2 Empirical testing

5.2.1 Scope and limitations of the tests

The objective of the empirical tests in this case was to find one or more heuristics which run well on a micro computer. The advantages of the micro computer are that problems can be solved cheaply, micro computers are generally available, even in small organizations, and are portable. These advantages are illustrated in the case study discussed in Section 5.3, concerning heuristically aided survey planning; such planning may be required in remote forest stations where access to extensive computer facilities is limited, while answers may be required rapidly.

Several published examples of TSPs are available. These have been augmented by randomly generated problems. Of the heuristics described in chapter 3, we decided to concentrate on tour construction heuristics because of the greater scope; there are many more tour construction than tour improvement techniques. In fact the general rule for the tour improvement heuristics seems to be (see for example Golden and Stewart, 1985): the better the initial tour for a tour improvement algorithm, the better the final tour found by this algorithm (best in, best out).

For the test of the tour construction heuristics, eight published heuristics were selected for testing. They were selected from Sections 3.2.2 (the ordered sequence type of construction approach), 3.2.3 (path approach) and 3.2.4 (subtour insertion approach). From Section 3.2.2, the greedy heuristic (a simple heuristic) and the savings heuristics (a sophistication of the simple ordered sequence approach) were chosen. The nearest neighbour and dynamic weighting heuristics were selected from

Sections 3.2.3, where again the former is a simple construction approach while the latter is more complex. From Section 3.2.4, two simple subtour insertion techniques were selected: the nearest and furthest insertion heuristics, the latter being selected because it has been shown as having the best performance of the simple heuristics (Adrabinski and Syslo, 1983). Of the more sophisticated subtour insertion methods in Section 3.2.4, the most eccentric ellipse heuristic (best of the variations described by Norback and Love, 1977), and cheapest angle heuristic (found by Golden, Bodin, Doyle and Stewart, 1980, to be the best of the convex hull techniques) were selected.

All five heuristics proposed in the cross-classification exercise of Chapter 4 were included in the tests. The new heuristics were the dynamic weighting variation heuristic (Section 4.4.1), the peeled convex hull heuristic (Section 4.4.2), the k-node look-ahead heuristic (Section 4.4.3), and the MST subtour and MST path heuristics (Section 4.4.4).

Computations were done on an IBM-compatible micro computer without a maths co-processor (a Mitac PC), running under MS-DOS at 8 Mhz. The programs were written in Pascal, and were run using the Turbo Pascal compiler. The programs were written as simply as possible; no special storage techniques were developed.

The results of the tests to be done in the next sections, will be expressed as the percentage above the best known solution, and will be tabulated in order of increasing running time of the programmed heuristic as in Table 5.2. This table shows the approximate running times of the heuristics for a 100-node problem, in order of increasing times, based on all test results. This type of ordering allows comparisons such as: what effort is required if a very good solution is required; what performance

can be expected if a quick running time is essential.

Table 5.2 Approximate running time for any TSP with n=100, on a micro computer (excluding input and output times)

category	heuristic	approximate time to solve fully any n=100 TSP
A	nearest insertion	1.5 minutes
	furthest insertion	1.5 minutes
	greedy	2 minutes
	peeled hull	2 minutes
	nearest neighbour	2.5 minutes
	MST subtour	10 minutes
B	eccentric ellipse	50 minutes
	cheapest angle	80 minutes
	MST path	80 minutes
C	savings	3 hours 45 minutes
	dynamic weighting	approximately 6 days
	variation of dynamic weighting	approximately 6 days
	2-node look-ahead	approximately 7 days

Category A represents the 'quick-and-dirty' approach. At the other end of the spectrum are the long running, intensive search heuristics of category C. These running times are only linked to the size of a problem and are thus always the same for a certain number of nodes.

Running the tests on a micro computer put several constraints on the test procedure, mainly due to lack of storage size and to slow processing speeds resulting in long running times. As can be seen from Table 5.2, letting most of the heuristics in category C run until completion on large sized problems is not feasible if the emphasis is on good results found cheaply and quickly. Running times were thus truncated at two hours; heuristics were then terminated at the first logical point after this time. All of the heuristics in category C have algorithms that are repeated for every node, i.e. each node serves in turn as initial node. Thus before another starting node is selected, a check is made that the two hour limit has not been exceeded. The initial nodes are assumed to be chosen randomly (without replacement), as it is

unknown which initial node will give the best solution.

Storage space also restricted the size of the problems to be tested. The TSP size was limited to just over 100 nodes by the Turbo Pascal compiler for micro computers. There are ways of getting around this, e.g. by compacting the cost matrix of symmetric TSPs, but they increase the running time. The size limit still allowed comparisons with published test results, which normally have most of their test problems with the number of nodes less than or equal to 100 (see for example Golden et al, 1980, and Adrabinski and Syslo, 1983). For the savings heuristic, limitations of storage space meant that the savings and the cost matrix could not be stored simultaneously. The algorithm thus required modification: insted of calculating and sorting the savings once, each saving was calculated when needed, increasing the running time slightly.

5.2.2 Tests on published problems

Eight published problems were tested. They include the 5-, 10-, 33-, and 57-node problems of Karg and Thompson (1964), and the 80- and 100-node problems of Norback and Love (1977). Note that these problems consist of cities in the U.S.A., but that Karg and Thompson give the intercity distances, while Norback and Love give the longitude and latitude of each city. A 42-node problem by Dantzig, Fulkerson and Johnson (1954) and a 30-node problem by Clarke and Wright (1964) were also tested.

The heuristics which include the construction of a convex hull, i.e. the most eccentric ellipse, the greatest angle and the peeled hull heuristics, require the coordinates of each node to be known and thus could not be applied when these data were

unavailable. Several of the published problems give only the edge costs of a TSP. Although the coordinates of, for example, the 57-node problem of Karg and Thompson (1964) are given by Norback and Love (1977), the comparison of the tours found from the coordinates with those found from the originally published costs is problematic, as several of the edge costs are very different to their Euclidean length. These differences are large enough to affect the optimum solution. (Norback and Love, 1977)

When the optimum solution of a TSP is known, a value of 0% shows that the heuristic has found this optimum. Where the optimum solution is not known, a value of 0% denotes the best result. Table 5.3 shows the results of the tests of the published problems.

Table 5.3 The percentages above best known solutions for the published problems.

\source: / (ii): / heuristic \ n=	(1) opt.known 5	(1) opt.known 10	(2) 30	(1) opt. known 33	(3) opt. known 42	(1) 57	(4) 80	(4) 100
A								
nearest insertion	3	6	10	9	19	13	16	13
furthest insertion	0 *	3	8 a	3 a	9 a	13	4 a	5 a
greedy	0 *	12	28	16	43	15	6	22
peeled hull	-	-	-	-	-	-	111	135
nearest neighbour	0 *	0 *	17	5	22	11 a	9	13
MST subtour	0 *	2	26	30	20	15	36	30
B								
eccentric ellipse	-	-	-	-	-	-	26	11
cheapest angle	-	-	-	-	-	-	1 b	0 *
MST path	0 *	3	6 b	5	9	6 b	13	27
C								
savings	0 *	0 *	0 *	1 *	.3	3 *	0 *	3
dynamic weighting (i)	0 *	2	4	1.3	0 *	14	21	13
variation of dynamic weighting (i)	0 *	2	26	30	29	65	75	62
2-node look-ahead (i)	0 *	10	21	10	20	33	44	37

(1) edge costs from Karg and Thompson, 1964
(2) edge costs from Clarke and Wright, 1964
(3) edge costs from Dantzig, Fulkerson and Johnson, 1954
(4) node coordinates from Norback and Love, 1977

* indicates the optimal or best solution found
a indicates the solution which is the best one found in 1.5 - 10 minutes, but not the best overall
b indicates the solution which is better than the one marked 'a', but which took between 10 minutes and 2 hours to find, and is not the best solution overall

(i) heuristic stopped at first logical point after 2 hours, if it has not yet terminated
(ii) the optimum solutions of these problems are known and have been used in the comparison

From Table 5.3 it can be seen that good results can be obtained by tour construction heuristics, even without the use of a tour improvement heuristic. Of the heuristics that do not require coordinates, the savings heuristic has the best overall performance. All other tour construction heuristics, in the test results (Table 5.3) and in other published results (Table 5.4), rarely outperform the savings heuristic. (Table 5.4 shows some published results of heuristics not tested in this thesis, run on the same problems as in Table 5.3.) Only the tour improvement heuristics, or a combination of tour construction and tour improvement heuristics, as shown in the lower part of Table 5.4, consistently produce better results than the savings heuristic.

Table 5.4 The percentages above best known solutions for the savings heuristic (the best of the heuristics tested in this study) and for some published heuristics not included in the test.

source of results	heuristic \ source:	(1) n= 33	(3) 42	(1) 57
	TOUR CONSTRUCTION:			
Table 5.3	savings	1	.3	3
(5)	loss	8	7	16
(5)	loss with refinements	3	4	.5
(6)	cheapest insertion	-	9	-
(7)	ratio times difference	.6	4	1
	TOUR CONSTRUCTION AND TOUR IMPROVEMENT:			
(7)	ratio times difference with Or-opt heuristic	0	1	1
(8)	k-optimal	0	0	0
(9)	variable k-optimal	0	0	0

where (1) Karg and Thompson, 1964
(3) Dantzig, Fulkerson and Johnson, 1954
(5) Webb, 1971
(6) Golden, Bodin, Doyle and Stewart, 1980
(7) Or, 1976
(8) Lin, 1965
(9) Lin and Kernighan, 1978

5.2.3 Tests on generated problems

Seven random problems were generated, five of which had 100 nodes, while the others had 50 and 75 nodes respectively. On the micro computer, the computer processing time alone (i.e. excluding input and output times) already took approximately 180 hours for all the tests, so testing many more problems was not practical. In addition, the tests gave clear and consistent results for these few problems.

Each node was generated as a set of three coordinates (x,y,z), where x and y were generated uniformly on (0,1) and z uniformly on (0, 0.1) using the Statgraphics package. The choice of the two intervals meant that the problems were 'close' to being Euclidean TSPs on the x,y-plane, but not actually so. Heuristics involving the construction of a convex hull used only (x,y) coordinates.

The edge costs were the distances between each pair of nodes:

$$C_{ij} = d(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

The problems are therefore symmetric and the triangle inequality holds.

Table 5.5 shows the percentages above the best known solutions for each of the heuristics tested on the generated problems.

Table 5.5 The percentages above best known solutions for the generated problems

heuristic	Number of nodes in the problem 50	75	100	100	100	100	100
A							
nearest insertion	21	17	16	17	13	11	17
furthest insertion	6 a	8 a	3 a	9	4 a	6 a	9
greedy	23	17	6	6 a	17	18	9
peeled hull	68	84	87	97	86	93	74
nearest neighbour	19	9	10	8	12	6.4	7 a
MST subtour	19	13	10	39	29	24	10
B							
eccentric ellipse	6	16	4	4	7	8	11
cheapest angle	0 *	3 b	1 b	1 b	6	.4 b	.05 b
MST path	16	5	7	9	7	15	8
C (i)							
savings	2	0 *	0 *	0 *	0 *	0 *	0 *
dynamic weighting	3	12	18	14	16	18	12
variation of dynamic weighting	39	29	43	55	38	41	67
2-node look-ahead	21	39	47	37	35	33	30

* indicates the best solution found

a indicates the solution which is the best one found in 1.5 - 10 minutes

b indicates the solution which is better than the one marked 'a', but which took between 10 minutes and 2 hours to find, and is not the best solution overall

(i) all the heuristics in this category were stopped at the first logical point after running for 2 hours, if they had not yet terminated

The best results overall were again found by the savings heuristic. The dominance of this heuristic in category C, of the cheapest angle heuristic in category B, and of the furthest insertion heuristic in category, meant that non-parametric statistical tests were unnecessary. The combined test results are discussed in more detail in the next section.

For those heuristics that were stopped after running for two hours, Table 5.6 shows the number of times each algorithm was repeated. The lower the number of initial nodes used, the smaller is the chance that the node which gives the best results for this method, has been used as an initial node.

Table 5.6 Number of times a category C heuristic was repeated, before it was stopped at the end of a repetition after running for two hours.

heuristic	Number of nodes 50	75	100
savings	-	-	54
dynamic weighting	22	5	2
variation of dynamic weighting	22	5	2
2-node look-ahead	17	4	2

- shows the heuristic completed within 2 hours

5.2.4 Comparison of the test results

The comparisons that follow refer to the tables of results, Tables 5.3 and 5.5, in the previous sections.

The best performance, on average, for the quickest heuristics (category A) is that of the furthest insertion heuristic. If a longer processing time is allowed, say up to 1.5 hours (category B), then the cheapest angle heuristic produces the best results. These results agree with the empirical tests of Adrabinski and Syslo (1983) and Golden and Stewart (1985). The cheapest angle heuristic, however, requires the problem to be Euclidean. For

problems where only the edge costs are known, the MST path heuristic at times does improve on the best tours found by the quickest heuristics.

The best overall performance of the tour construction techniques (categories A to C) is by the savings heuristic. This method consistently outperforms the furthest insertion and the cheapest angle heuristic, although taking longer to do so.

The performance of the dynamic weighting heuristic is fairly good for small problems, but as soon as the number of nodes becomes large it is repeated only a few times, and the quality of the solution decreases. On a 100-node problem, the heuristic is only run for two starting nodes before exceeding the 2 hour time limit, and the probability is small that one of these nodes is in fact the node which gives the best results for this method.

New heuristics.

The MST subtour heuristic involves a function of M, the number of nodes between two end nodes of each minimum spanning tree found at each iteration (Section 4.4.4). Two functions were proposed: $f(M)= M$ and $f(M) = M^2$. Using the latter function results in a quicker heuristic, but the tours found are approximately 30% longer than the best tours found (Table 5.7), whereas the tours found when using the function $f(M)= M$ are only approximately 20% longer than the best tours found.

Table 5.7 Percentages above the best known solution. for two functions in the MST subtour heuristic.

f(M)	(1)	(2)	(1)	(3)	(1)	(4)	(4)	approximate running time (n=100)
	published problems with n =					(*)		
	10	30	33	42	57	80	100	
M	2	26	30	20	15	36	30	10 mins
M^2	12	26	29	46	35	38	24	8 mins

(*) for references see Table 5.2

The function $f(M)= M^2$ of the MST subtour heuristic puts more emphasis on the number of nodes between two end nodes of each minimum spanning tree, than on the cost of the edges between these two end nodes. Comparing the results of Table 5.7 with those of the nearest and furthest insertion heuristics (Table 5.2), it seems that for the subtour insertion approaches, the shorter the cost of the insertion performed at each iteration, the better. Note that this does not hold for the path approaches: the MST path heuristic, which may add several edges onto the end of the path in each iteration, frequently improves on the results of the nearest neighbour heuristic, which adds on one edge at a time.

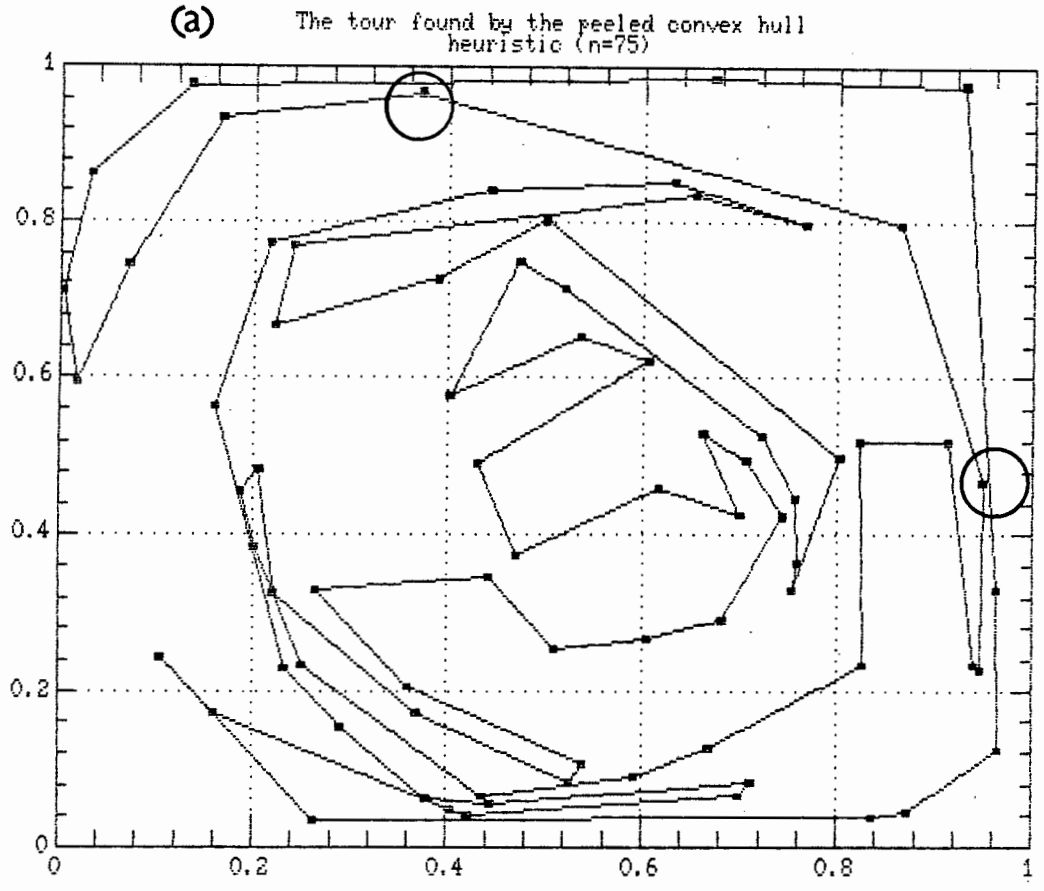
The k-node heuristic was tested for k=2 only, as for k>2 the running time becomes excessive. The results of this heuristic were worse than the nearest neighbour heuristic which uses a similar construction approach.

The variation of the dynamic weighting heuristic does not improve the results of the original heuristic. At each iteration, choosing the least of the upper bounds eventually leads to a worse tour than the one found by choosing the least of the lower bounds at each iteration.

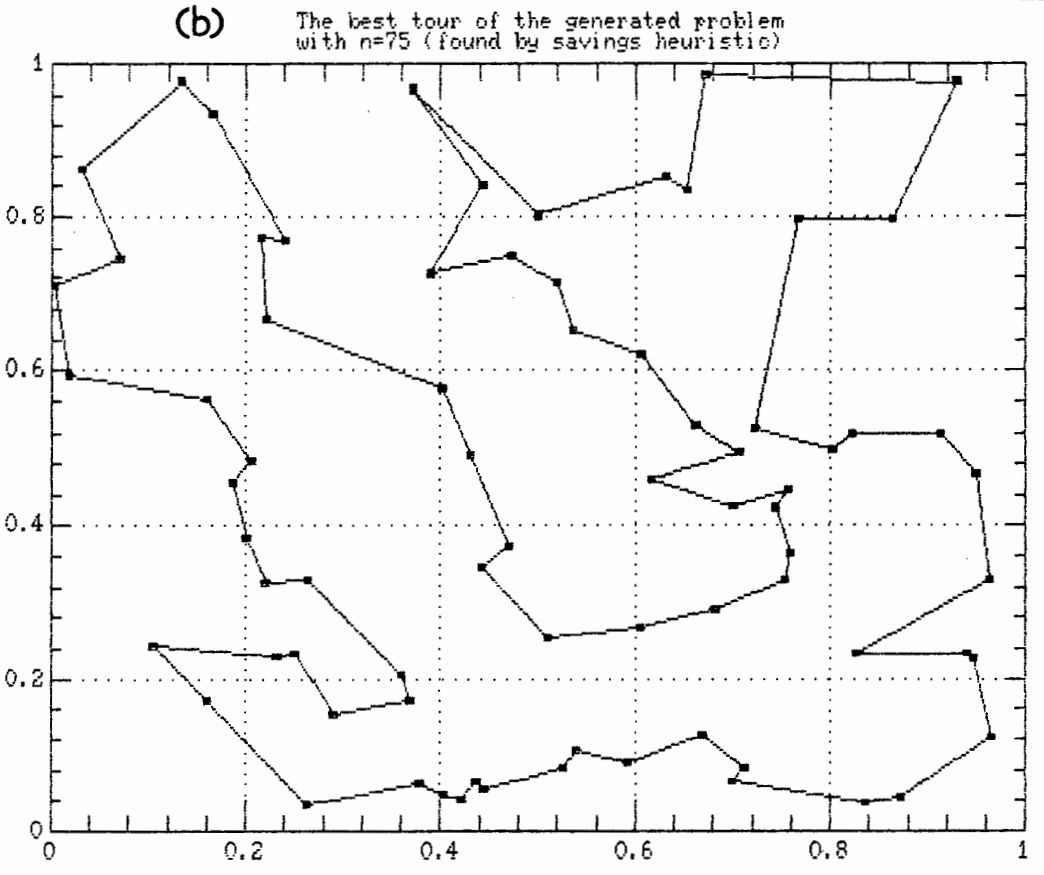
The worst performance of all the tour construction heuristics,

particularly for the published problems, is that of the peeled convex hull heuristic. The reason for this can be shown by the comparison of the tour of a generated problem as found by the peeled hull heuristic (Figure 5.1a), with the best tour found for the same problem (Figure 5.1b). When finding a convex hull, nodes 'close' to the edges of the hull are not included into the hull, instead they are included into the next hull constructed. For the problem in Figure 5.1a, nine convex hulls were constructed by the heuristic, whereas from the drawing of the best tour of the problem (Figure 5.1b), finding and merging two hull-like subtours (where one subtour is contained within the other) would lead to a better solution. Although it may be possible to improve the results of the peeled hull heuristic by inserting nodes 'close' to a hull into that hull, before finding the next hull, simpler techniques giving much better tours already exist.

Fig. 5.1 Two tours of the generated problem with n=75.



two examples of nodes on one hull, 'close' to another hull



In fact most of the new heuristics tested did not have a very good performance compared to previously published heuristics. The k-node look-ahead heuristic finds tours that are approximately 40% longer than the best tours found, the dynamic weighting variation heuristic finds tours approximately 30% longer and the MST subtour finds tours approximately 20% longer than the best tours found. The only exception is the MST path heuristic, which does fairly well on those problems where only edge costs are known, finding tours approximately 9% longer the best tour.

5.3 A case study

The case study arose out of the forest soil survey program of the Saasveld Forestry Research Institute, George. When surveying large areas intensively, the work involved can be great and can take much time. The effort can be reduced by efficient planning of the field survey. For large surveys, the best use of vehicular access routes to planned observation points can save expensive surveyor's legwork. In dense vegetation such as plantation or natural forest, vehicle access is confined to the defined road systems, and the surveyor may spend much time on foot.

The survey program has been concentrating on the use of geostatistics, which requires using a grid survey approach to the surveys (McBratney and Webster, 1982). As the observation points are known in advance, it is possible to plan the field work to minimize the effort in some way. The work involves surveying in often innaccessible areas - plantation and natural forest - where the surveyors must make use of the existing infrastructure of roads and tracks to reach the observation points.

One way of minimizing effort is to plan a route which minimises

the amount of walking necessary and, to a lesser extent, the amount of driving required. The program required had to be a planning aid rather than a program giving an optimum answer; it needed to be quick running and interactive to allow the planner to investigate a number of options to fit his operating criteria. It would have to output the cost of various options, the final decision would then rest with the surveyor.

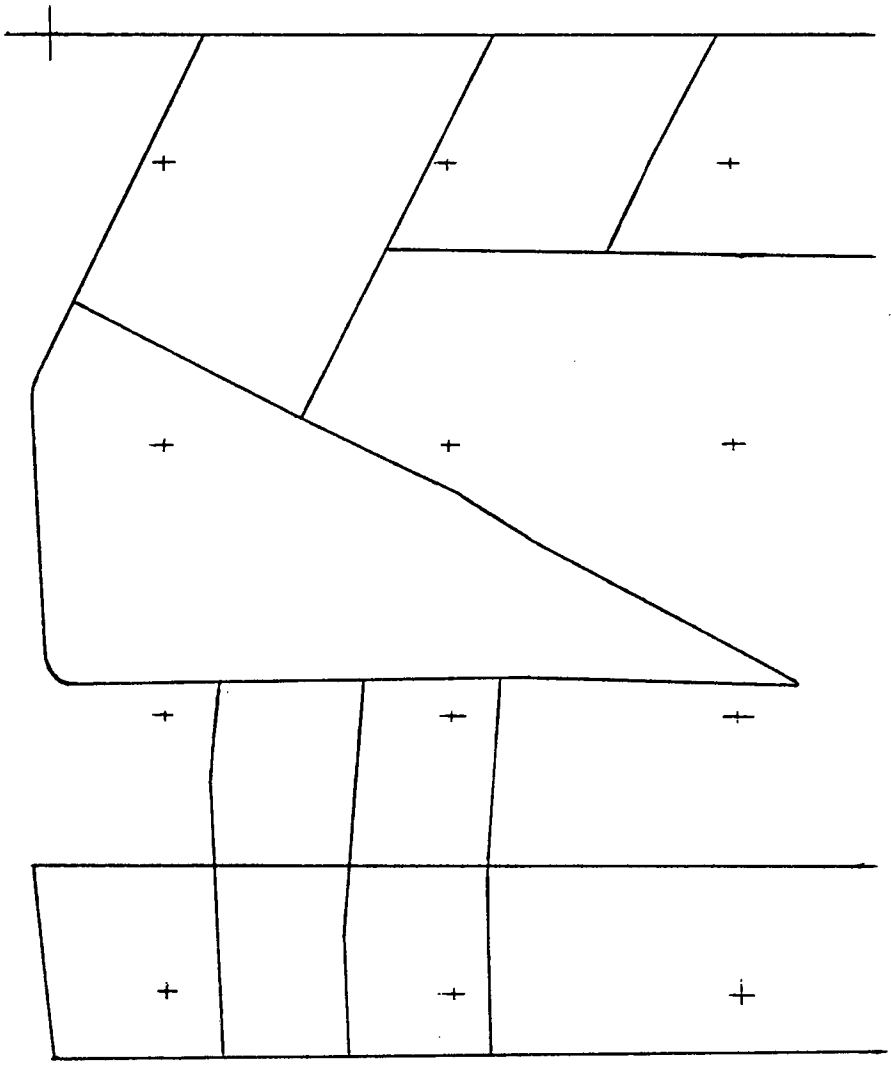
The problem was split into three parts. The first aim was to minimize the walking distance from a road or track, i.e. where the vehicle could be parked, then the parking points were sequenced using a TSP heuristic, and finally the distances to be driven were calculated. The necessary data were the coordinates of all road and track intersections in the area to be surveyed, and for each intersection all neighbouring intersections connected to it, and the coordinates of the survey observation points as output by the geostatistical programs used. All coordinates were given according to the SA national mapping program.

The parking points were calculated by finding the shortest distance to the nearest road, and were displayed. Any of the TSP tour construction heuristics could be used to sequence the parking points, with the recommendation made to use, in order of running time, the furthest insertion heuristic, the cheapest angle heuristic or the savings heuristic. To visit the parking points in the order as found by the TSP heuristics, the sequence of the intersections necessary is calculated and displayed. The planner is then given the options of changing the sequence of the parking points, or the combination of roads used.

This may perhaps be best explained by using an example of a set of roads and observation points, as shown in Figure 5.2 (a).

Fig. 5.2 Example of a surveying tour suggested.

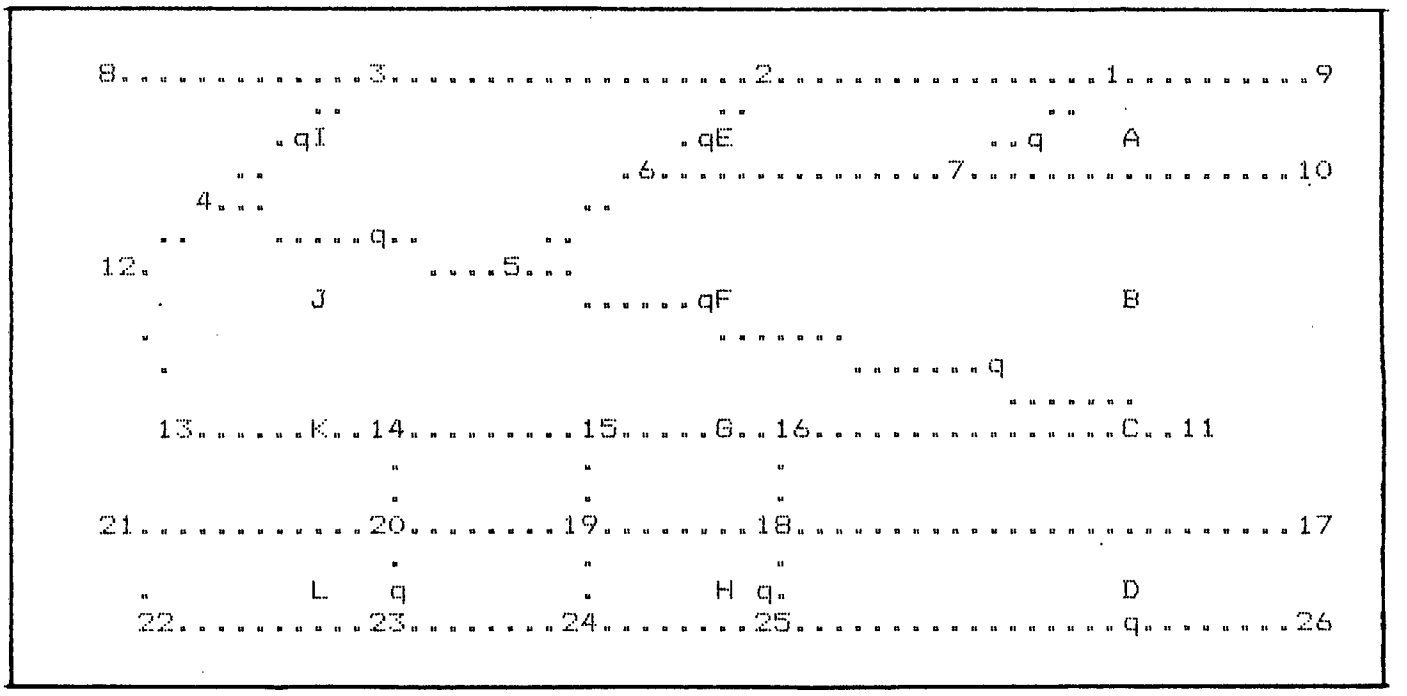
(a) Proposed survey area



survey observation points

roads and tracks

(b) Proposed parking points



1 - 26 intersection numbers

A - L survey observation points

q parking point

Parking points are:			Length:	Between intersections:
A	91.826	3760.800	0.918	1 and 7
B	92.133	3767.108	2.393	5 and 11
C	91.000	3768.200	0.500	11 and 16
D	91.000	3773.300	0.700	25 and 26
E	94.912	3761.139	0.128	5 and 6
F	95.077	3765.519	0.588	4 and 5
G	94.800	3768.200	0.500	14 and 15
H	94.476	3772.631	0.325	16 and 18
I	98.925	3761.096	0.248	3 and 4
J	98.113	3763.900	1.247	4 and 5
K	98.700	3768.200	0.500	13 and 14
L	98.035	3772.613	0.665	14 and 20

The sum of the minimum walking distances is : 8.712

(c) Proposed tour plus modification.

```
Where do you want to start the tour ?i

Visit the observation points in the sequence:
  I - E - A - F - G - B - C - D - H - L - K - J - I

The recommended tour is:
  I - 3 - 2 - E
  E - 6 - 7 - A
  A - 7 - 6 - 5 - F
  F - 11 - 16 - G
  G - 16 - 11 - B
  B - 11 - C
  C - 16 - 18 - 25 - D
  D - 25 - H
  H - 25 - 24 - 23 - L
  L - 20 - 14 - K
  K - 13 - 12 - 4 - J
  J - 4 - I
Total distance driven: 81.91

Do you want to change the sequence of:  O = observation points
                                        I = intersections between two poi
                                        N = no changes
                                      ?o
You are unhappy about point ?g
Between which two points do you think it should be ?c d
       *** the new tour is shorter ****

Visit the observation points in the sequence:
  I - E - A - F - B - C - G - D - H - L - K - J - I

The recommended tour is:
  I - 3 - 2 - E
  E - 6 - 7 - A
  A - 7 - 6 - 5 - F
  F - B
  B - 11 - C
  C - 16 - G
  G - 16 - 18 - 25 - D
  D - 25 - H
  H - 25 - 24 - 23 - L
  L - 20 - 14 - K
  K - 13 - 12 - 4 - J
  J - 4 - I
Total distance driven: 70.30

Do you want to change the sequence of:  O = observation points
                                        I = intersections between two poi
                                        N = no changes
                                      ?n
```

First the data relating to the area needs to be stored: the coordinates of all road and track intersections in the area, and for each intersection all neighbouring intersections connected to it. Secondly the coordinates of the survey observation points must be stored. The program which calculates the parking points is run; Figure 5.2 (b) shows its output, which is printed for later use. The program also creates a file with input data for any one of the TSP heuristics.

In this example, the parking points were sequenced by the furthest insertion heuristic, as it is the best of the quick heuristics. The output of the TSP heuristic is combined with the data of the intersections of the area into a tour by a final program. This program displays the suggested tour and allows interactive modifications, as shown in Figure 5.2 (c). The letters and numbers used for the recommended tour refer to the diagram output by the first program (Figure 5.2 (b)).

The easiest way of viewing the recommended tour is to follow it around on the print of the diagram, looking for possible improvements. An example of a section of the tour that could be improved is the part of the tour from point C to D via intersections 16, 18 and 25, which comes close to point G and passes by point H without stopping. These points have to be returned to later. The improvements are made by selecting to change the sequence of points, and specifying that point G is to be visited between points C and D (see the middle of Figure 5.2 (c)). The program checks the suggested improvement, and displays the shorter tour. If a suggestion lengthens the total distance of the tour, then the user is first asked if this is acceptable before the tour is displayed. The other improvement (inserting point H between G and D rather going to it after visiting D and before visiting L) is done similarly.

Another option is to change the sequence of the intersections between two points. The letters of two adjacent points on the tour are input, and then the numbers which represent the intersection suggested to connect the points. The program checks the suggestion and displays the tour if it is shorter, otherwise the user is first asked if this is acceptable before the program displays the tour. When no more changes are made by the user, the final tour is stored.

This set of programs has received favourable interest from the people concerned with soil surveying at Saasveld. Combined with geostatistics, it was considered to be a very useful tool in survey planning. Results can be produced quickly and interactively.

CONCLUSION

Using classification techniques such as the morphological boxes, gave an overview of the many and diverse TSP heuristics, and was found useful in discovering new heuristics for the TSP.

Even with examining the additional algorithms implied by the morphological boxes, however, no substantial improvement over the previously published algorithms were found. This implies that there may not be much more scope for improvement on TSP heuristics, unless some fundamentally new idea comes up. Even then, success is not guaranteed: for example, the process of simulated annealing signified a new approach to solving combinatorial problems such as the TSP heuristically, but Golden and Skiscim (1986) found it was outperformed by a combination of tour construction and tour improvement heuristics, both in the time required to find a solution and in the quality of the solution found.

From the empirical tests performed in this study, recommendations can be made for which heuristic to use on symmetric TSPs where the triangle inequality holds. If a quick solution is necessary, the furthest insertion heuristic will find tours about 3 to 9% above the best tours possible. Better tours can be achieved by heuristics which take somewhat longer (approximately 80 minutes for 100 node problem): if the coordinates of the nodes are available, the cheapest angle heuristic may be used (0 to 6% above the best tour), otherwise the new MST path heuristic does at times improve on the results of the furthest heuristic (4 improvements out of 15 problems, with 4 out of the other results tying with the results of the furthest heuristic). The longer the processing time, the better the result: the savings heuristic, even though stopped if it had

not completed in two hours, usually found the best tours, and was at worst only 3% above the best known solution.

Choosing a heuristic for a specific TSP depends not only on the time available and the accuracy required, but it also depends on the structure of the problem. The cheapest angle heuristic has been shown (in this thesis and by Golden and Stewart, 1985) to perform very well, certainly better than the quick heuristics, on the test problems that were only 'close' to being Euclidean on a plane. However, the heuristic has the disadvantages that when no coordinates are available, it cannot be used, and when the coordinates are available but the costs of edges are very different to their Euclidean distance, the quality of the heuristic's solution decreases. The savings heuristic, the best of the tour construction heuristics, does not have these disadvantages, and can also be used on asymmetric problems, but it takes longer to complete.

Prospects for future research.

Empirical tests could be performed on the use of the TSP heuristics reviewed in this study for solving the more general TSPs where, for example, the triangle inequality does not hold, or for solving the more restricted problems such as the directed TSPs. Another possibility is the study of why a non-dominant heuristic at times manages to find a better result than the dominant heuristic.

B I B L I O G R A P H Y

Adrabinski, A. & Syslo, M.M., 1983, Computational Experiments with Some Approximation Algorithms for the Travelling Salesman Problem, Zastosowania Matematyki, 18:1, 91-95

Balas, E. & Padberg, M.W., 1976, Set Partitioning: A Survey, SIAM Revue, 18:4, 710-760

Balas, E. & Toch, ., 1985, , in: The Travelling Salesman Problem, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan & D.B. Shmoys (eds.), Wiley, New York

Ball, M. & Magazine, M., 1981, The Design and Analysis of Heuristics, Networks, 11:2, 215-219

Bartholdi, J.J., Platzman, L.K., Collins, R.L., Warden, W.H., 1983, A Minimal Technology Routing System for Meals on Wheels, Interfaces, 13, 1-8

Baum, R., Carlson, R., 1979, On Solutions that are Better than Most, Omega (International Journal Of Management Science), 7, 249-255

Bellmore, M. & Malone, J.C., 1971, Pathology of Travelling Salesman Subtour-elimination Algorithms, Operations Research, 19, 278-307

Bellmore, M. & Nemhauser, G.L., 1968, The Travelling Salesman Problem: A Survey, Opn. Res., 16, 538-558

Burkard, R.E., 1979, Travelling Salesman and Assignment Problems: A Survey, Annals of Discrete Mathematics, 4, 193-215

Carpaneto, G., Martello, S. & Toth, P., 1984, An Algorithm for the Bottleneck Travelling Salesman Problem, Opn. Res., 32:2

Christofides, N., 1972, Bounds for the Travelling Salesman Problem, Opns. Res., 20, 1044-1055

Christofides, N., 1982, The Travelling Salesman Problem - A Survey, in: Applications of Combinations, R.J. Wilson, Shira Publ. Ltd, Cheshire

Christofides, N. & Eilon, S., 1976, Algorithms for Large-scale Travelling Salesman Problems, Operations Research Quarterly, 23:4, 511-518

Clarke, G. & Wright, J.W., 1964, Scheduling Vehicles from a Central Depot to a Number of Delivery Points, Operations Research, 12:4, 569-581

Coffman, E.G., Garey, M.R. & Johnson, D.S., 1984, Approximation Algorithms for Bin Packing - An Updated Survey, in: Algorithm Design for Computer Systems Design, G. Ausiello, M. Lucertini & P. Sarafini (eds), Springer, New York, 49-106

Crowder, H.P., Demdo, R.S. & Mulvey, J.M., 1978, Reporting Computational Experiments in Mathematical Programming, Mathematical Programming, 15:3, 316-329

Crowder, H.P. & Padberg, M.W., 1980, Solving Large-scale Symmetric Travelling Salesman Problems to Optimality, Management Science, 26:5 , 495-509

D'Atri, G., 1980, A Note on Heuristics for the Travelling Salesman Problem, Mathematical Programming, 19:1, 111-114

Dantzig, G., Fulkerson, R. & Johnson, S., 1954, Solution of a Large-scale Travelling Salesman Problem, Operations Research, 2, 393-410

Dembo, R.S. and Mulvey, J.M., 1978, Reporting Computational Experiments in Mathematical Programming, Mathematical Programming, 15:3, 316-329

Derman, C. & Klein, M., 1966, Surveillance of Multi-component Systems: A Stochastic Travelling Salesman's Problem, Naval Res. Logistics Quarterly, 13, 103-111

Diegel, A., 1986, From Simultaneous Equations to Linear and Integer Programming, Manuscript for Private Circulation, 180-182

Eilon, S., 1977, More Against Optimization, OMEGA (Int. J. of Management Science), 5, 627-633

Fisher, M.L., 1980, Worst-case Analysis of Heuristic Algorithms, Management Science, 26:1, 1-16

Fisher, M.L., Nemhauser, G.L. & Wolsey, L.A., 1978, An Analysis of Approximations for Maximizing Submodular Set Functions - II, Mathematical Programming, 8, 73-87

Fleischmann, B., 1985, A Cutting Plane Procedure for the Travelling Salesman Problem on Road Networks, European J. of Operational Res., 21, 307-317

Flood, M.M., 1956, The Travelling Salesman Problem, Operations Res., 4:1, 61-75

Foulds, L.R., 1983, The Heuristic Problem-solving Approach, J. Opl. Res. Soc., 34:10, 927-934

Foulds, L.R., 1984, Combinational Optimization for Undergraduates, Springer-Verlag, New York

Fuller, J.A., 1978, Optimal Solutions Versus 'Good' Solutions, Omega (Int. J. of Management Science), 6:6, 479-484

Gallus, G., 1976, Heuristische Verfahren zur Losung Allgemeiner Ganzzahliger Linearer Optimierungsprobleme, Zeitschrift Fur OR, 20, 89-104

Garey, M.R. & Johnson, D.G., 1976, Approximation Algorithms for Combinational Problems : An Annotated Bibliography, in: Algorithms and Complexity, J.F. Traub, Academic Press, New York

Garfinkel, R.S., 1985, Motivation and Modeling, in: The Travelling Salesman Problem, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys (ed.), Wiley, New York

Glover, F., 1986, Future Paths for Integer Programming and Links to Artificial Intelligence, Computers & Operations Res., 13:5, 533-549

Golden, B.L., 1977, Statistical Approach to the Travelling Salesman Problem, Networks, 7, 209-225

Golden, B.L., & Assad, A.A., 1984, A Decision-theoretic Framework for Comparing Heuristics, European J. of Operational Res., 18, 167-171

Golden, B.L., Assad, A.A., Wasil, E.A. & Baker, E., 1986, Experimentation in Optimization, European J. of Operational Res., 27, 1-16

Golden, B.L., Bodin, L., Doyle, T., Stewart, W.R., 1980, Approximate Travelling Salesman Algorithms, Operations Research, 28:3, 694-711

Golden, B.L. & Skiscim, C.C., 1986, Using Simulated Annealing to Solve Routing and Location Problems, Naval Res. Logistics Quarterly, 33, 261-279

Golden, B.L. & Stewart, W.R., 1985, Empirical Analysis of Heuristics, in The Travelling Salesman Problem, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys (ed.), Wiley, New York

Groner, R. & M., Bischof, W.F. (eds.), 1983, Methods of Heuristics, Lawrence Erlbaum Ass., London

Hansen, K.h., & Krarup, J., 1974, Improvements of the Held-Karp Algorithm for the Symmetric Travelling Salesman Problem, Mathematical Programming, 7, 87-96

Held, M. & Karp, R.M., 1970, The Travelling Salesman Problem and Minimum Spanning Trees: Part 1, Opns. Res., 18, 1138-1162

Hillier, F.S., 1983, Heuristics: A Gambler's Role, Interfaces, 13:3, 9-12

Hochbaum, D.S. & Shmoys, D.B., 1986, Best Possible Heuristics for the Bottleneck Wandering Salesperson and Bottleneck vehicle routing problem, European J. of Operational Res., 26:3, 380-384

Hu, T.C., 1969, Integer Programming and Network Flows, Addison-Wesley, Massachusets

Ignizio, J.P., 1980, Solving Large-scale Problems: A Venture Into a New Dimension, J. Opl. Res. Soc., 31, 217-225

Isaac, A.M. & Turban, E., 1969, Some Comments on Travelling Salesman Problems, Operations Res., 17, 543-546

Jeromin, B. & Korner, F., 1985, On the Refinement of Bounds of Heuristic Algorithms for the Travelling Salesman Problem, Mathematical Programming, 32:1, 114-117

Johnson, D.S., 1974, Approximate algorithms for combinatorial Problems, J. of Computer and System Sciences, 9, 256-278

Johnson, D.S. & Papadimitriou, C.H., 1985, Performance Guarantees for Heuristics, in The Travelling Salesman Problem, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys (ed.), Wiley , New York

Johnson,D.S, Aragon,C.R, McGeoch,L.A, & Schevon, C., 1987, Draft of Paper, AT & T Laboratories, August 6, 1987

Jongens, K. & Volgenaut, T., 1985, The Symmetric Clustered Travelling Salesman Problem, European J. of Operational Res., 19, 68-75

Jonker, R., De Leve, G., Van Der Velde, J.A. & Volgenaut, A., 1980, Rounding Symmetric Travelling Salesman Problems with an Asymmetric Assignment Problem, Opns. Res., 28:3:I

Kahneman, D., Slovic, P. & Tverskey, A., 1982, Judgement Under Uncertainty: Heuristics and Biases, Cambridge Univ. Press

Karg, R.L. & Thompson, G.L., 1964, A Heuristic Approach to Solving Travelling Salesman Problems, Management Science, 10:2 225-248

Karp, R.M., 1975, On the Computational Complexity of Combinational Problems, Networks, 5, 45-68

Karp, R.M., 1976, Probabilistic Analysis of Some Combinational Search Algorithms, in: Algorithms and Complexity, Academic Press

Karp, R.M., 1977, Probabilistic Analysis of Partitioning Algorithms for the Travelling Salesman Problem in the Plane, Math. of Op. Res., 2:3, 209-224

Karp, R.M. & Steele, J.M., 1985, Probabilistic Analysis of Heuristics, in The Travelling Salesman Problem, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys (ed.), Wiley, New York

Kilbridge, M.D., Webster, L., 1962, A Review of Analytical Systems of Line Balancing, Operations Research, 10, 626-638

Kirkpatrick, S., Gelatt, C.D. & Vecchi, M.P., 1983, Optimization by Simulated Annealing, Science, 220:4598

Kleinmutz, D.N., 1985, Dynamic Decision Environment: Heuristics and Feedback, Management Science, 31:6, 680-702

Lawler, E.L., 1976, Combinational Optimization : Networks and Matroids, Holt, Riehart & Winston, New York

Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G. and Shmoys, D.B. (eds.), 1985, The Travelling Salesman Problem, Wiley, New York

Lenat, D.B., 1983, Toward a Theory of Heuristics, in: Methods of Heuristics, R. & M. Groner & W.F. Bischof (eds.), Lawrence Erlbaum Ass., London

Lenstra, J.K., 1974, Clustering a Data Array and the Travelling Salesman Problem, Operations Research, 22, 413-414

Lenstra, J.K. & Rinnooy Kan, A.H.G., 1975, Some Simple Applications of The Travelling Salesman Problem, Opl. Res. Quarterly, 26:4, 717-733

Liesegang, G. & Schirmer, A., 1975, Heuristische Verfahren zur Maschinenbelegungsplannung bei Reihenfertigung, Z. Operations Res., 19, 195-211

Lin, B.W. & Rardin, R.L., 1980, Controlled Experimental Design for Statistical Comparisons of Integer Programming Algorithms, Management Science, 25:12, 1258-1271

Lin, S., 1965, Computer Solutions of the Travelling Salesman Problem, Bell Sytstem Technical J., 44, 2245-2269

Lin, S., 1975, Heuristic Programming as an Aid to Network Design, Networks, 5, 33-43

Lin, S. & Kernigham, B.W., 1978, An Effective Heuristic Algorithm for the Travelling Salesman Problem, Operations Research, 21, 498-516

Little, J.D.C., Murty, K.G., Sweeney, D.W., Karel, C., 1963, An Algorithm for the Travelling Salesman Problem, Operations Research, 11, 972-989

Lundy, M. & Mees, A., 1986, Convergence of an Annealing Algorithm, Mathematical Programming, 34, 111-124

Matthaus, F.W., 1975, Heuristische Losungsverfahren fur Lieferplanprobleme, Z. Operations Res., 19, 163-181

McBratney, . & Webster, R., 1981, The Design of Optimal Sampling Schemes for Local Estimation and Mapping of Regionalized variables - II, Computers + Geosciences, 7, 335 - 365

Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H. & E., 1953, Equation of State Calculations by Fast Computing Machines, J. Chemical Physics, 21:6

Moore, J.M., 1968, An Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs, Management Science, 15, 102-109

Muller-Merbach, H., 1969, Das Verfahren der "Vorsichtigen Annaherungen", Elektronische Datenverarbeitung, 12/69, 564-566

Muller-Merbach, H., 1973, Heuristische Verfahren, Handwortenbuch Der Math. Wirtschafts Wissenshaften, 63-69

Muller-Merbach, H., 1974, Heuristic Methods: Structures, Applications, Computational Experience, in: Optimization Methods for Resource Allocation, R. Cottle & J. Krarup, English Univ. Press, London

Muller-Merbach, H., 1976, Morphologie Heuristischer Verfahren, Zeitschrift fur OR, 20, 69-87

Muller-Merbach, H., 1976a, Ansatze zu Entwurfsmethodologie fur Algorithmen der Kombinatorischen Opt., 655-667

Muller-Merbach, H., 1981, Heuristics and Their Design: A Survey, European J. of Operational Res., 8, 1-23

Muller-Merbach, H., 1984, A 5-Facet Frame for the Design of Heuristics, European J. of Operational Res., 17:3, 313-316

Newell, A., 1983, Model of Human Problem Solving, in: Methods of Heuristics, R. & M. Groner & W.F. Bischof (eds.), Lawrence Erlbaum Ass., London

Norback, J.P. & Love, R.F., 1977, Geometric Approaches to Solving the Travelling Salesman Problem, Management Science, 23:11, 1208-1223

Or, I., 1976, Travelling Salesman-Type Combinational Problems and Their Relation to the Logistics of Regional Blood Banking, PhD-thesis, Northwestern Univ. Evanston, IL.

Padberg, M.W., & Hong, S., 1980, On the Symmetric Travelling Salesman Problem: A Computational Study, Math. Progr. Study, 12, 78-107

Papadimitriou, C.H., 1977, The Euclidean Travelling Salesman Problem is NP-complete, Theoret. Comput. Science, 4, 237-244

Parker, R.G., Rardin, R.L., 1983, The Travelling Salesman Problem: An Update of Research, Naval Res. Logistics Quarterly, 30, 69-96

Pearl, J., 1984, Heuristics Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, N.Y.

Picard, J.C. & Queyranne, M., 1978, The Time-dependant TSP and Its Application to the Tardiness Problem in One-machine Sceduling, Operations Res., 26:1, 86-110

Platzman, L.K. & Bartholdi, J.J., 1984, Spacefilling Curves and the Planar Travelling Salesman Problem, Submitted To JACM

Pohl, I., 1973, The Avoidance of (Relative) Catastrophe Heuristic Competence, Genuine Dynamic Weighting and Computationel Issues in Heuristic Problem Solving, Proc. I.J.C.A.I. (Int. Joint Conf. on AI) 3, Stanford, 20-23

Pohl, I., 1977, Practical and Theoretical Considerations in Heuristic Search Algorithms, in: Machine Intelligence 8, E.W. Elcock & D. Michie (eds.), Ellis Horwood Ltd, 55-72

Polya, G., How To Solve It, Princeton Univ. Press, Princeton, N.J.

Pospelov, D.A., Pushkin, V.N., Sadovski, V.N., 1972, Toward a Definition of Heuristics, in: Problems of Heuristics, V.N. Puskin (ed.), Keter Press, Jerusalem

Pursglove, W. & Boffey, T.B., 1980, Heuristic Improvement Methods, Mathematical Programming Studies, 13, 135-142

Raymond, T.C., 1969, Heuristic Algorithm for the Travelling Salesman Problem, IBM J. of Res. and Development, 13, 400-407

Reingold, E.M., Nievergelt, J., Deo, N., 1977, Combinational Algorithms, Prentice-Hall, N.Y.

Reiter, S. & Sherman, G., 1965, Discrete Optimizing, SIAM (Soc. for Ind. and App. Math.) J. on Applied Mathematics, 13, 864-889

Rinnooy Kan, A.H.G., 1984, An Introduction to Approximation Algorithms, Report 8420/0, Erasmus Univ., Rotterdam

Rinnooy Kan, A.H.G., 1985, Probabilistic Analysis of Algorithms, Report 8538/A

Rosenkrantz, Stearns & Lewis, 1977, An Analysis of Several Heuristics for the Travelling Salesman Problem, Soc. for Industrial and Applied Mechanics (SIAM) J. on Computing, 2, 563-581

Rossier, Y., Troyon, M. & Liebling, T.M., 1986, Probabilistic Exchange Algorithms and the Euclidean Travelling Salesman Problem, OR Spectrum, 8, 151-164

Silver, Vidal, De Werra, 1980, Introduction To Heuristic Methods, European Journal Of Operational Research, 5, 153-162

Supowit, K.J., Reingold, E.M., Plaisted, D.A., 1983, The Travelling Salesman Problem and Minimum Matching in the Unit Square, SIAM J. On Computing, 12:1, 144-155

Telgen, J., 1985, How to Schedule Meetings with the Travelling Salesman Q&D, and Why We Didn't, Interfaces, 15, 89-93

Tikhomirov, O.K., 1983, Informal Heuristic Principles of Motivation and Evolution in Human Problem Solving, in: Methods of Heuristics, R. & M. Groner & W.F. Bischof (eds.), Lawrence Erlbaum Ass., London

Volgenant, A., & Jonker, R., 1985, Improving Christofides' Lower Bound for the Travelling Salesman Problem, Mathematische Operationsforschung und Statistik, Series Optimization, 16:5, 691-704

Webb, M.H.J., 1971, Some Methods of Producing Approximate Solutions to the Travelling Salesman Problem with Hundreds or Thousands of Cities, Opl. Res. Quarterly, 22, 49-66

Welsh, D.J.A., Problems in Computational Complexity, in: Applications of Combinations, C.F. Christofides

Wolsey, L.A., 1980, Heuristic Analysis, Linear Programming and Branch and Bound, Mathematical Programming Study, 13, 121-134

Zanakis, S.H., 1977, Heuristic 0-1 LP: An Experimental Comparison of 3 Methods, Management Science, 24:1, 91-104

Zanakis, S.H. & Evans, J.R., 1981, Heuristic "Optimization": Why, When and How to Use It, Interfaces, 11:5, 84-91

Zwicky, F., 1968, Discovery, Invention, Research, MacMillan, New York

APPENDIX

```
program NodeCheapestInsertion;
{$R-}
{
    method:      progressively include the nearest remaining node into
                 the current subtour.
                 (Approach 2: insert node in cheapest way).
    conditions:  non-Euclidean
                 symmetric
    input:       - how many nodes
                 - input by distances:
                   for edge (i,j) connecting nodes i and j
                   input the cost/distance i,j.
                 - input by coordinates:
                   for node i input its coordinates.
}

const
      maxN       = 101;
      Nless1     = 100;
      LineLength = 250;

type
      NameType   = string [12];
      LineType   = string [LineLength];
      OneNless1  = 1..Nless1;
      ZeroN      = 0..maxN;
      OneN       = 1..maxN;
      TwoN       = 2..maxN;
      CostMatrx  = array [OneN, OneN] of real;
      CoordArray= array [OneN] of real;
      TourArray  = array [OneN, 0..1] of ZeroN;

var
  OutFile      : text;
  Cost         : CostMatrx;
  N, NodeCount: OneN;
  Subtour      : TourArray;



Function ReadNumber (var LL: integer;
                     var B: LineType) : real;
var Int, OK : integer;
    PointFlag : boolean;
    R, RD, Decimal : real;
begin                                   { read one number }
  while (B [LL] = ' ') and (LL <= LineLength) do
    LL:= LL+1;                          { skip all spaces }
  Decimal:= 0.1;
  PointFlag:= false;
  R:=  0;
  RD:= 0;
  while (B [LL] <> ' ') and (LL <= LineLength) do
    begin                               { find the number }
      if (B [LL]= '.') then
        PointFlag:=true
      else
        begin
          val (B [LL], Int, OK);
```



```
          if PointFlag = false then
            R:= R*10 + Int
          else
            begin
              RD:= RD + (Int * Decimal);
              Decimal:= Decimal * 0.1;
            end;
        end;
      LL:= LL+1;
    end;                                { one value found }
  ReadNumber:= R + RD;
end;          { ReadNumber }


Procedure ReadData (var DataFile: text;
                    var CDFlag  : char;
                    var X, Y, Z : CoordArray;
                    var ZFlag   : boolean);
var L: integer;
    NumLines, NL, NumPerLine, NPL, I, J: OneN;
    A: LineType;
begin
  readln (DataFile, A);
  L:= 4;
  N:= trunc (ReadNumber (L, A));
  ZFlag:= false;
  if A [1] = 'C' then
    begin
      CDFlag:= 'C';
      if A[2] = 'Z' then  ZFlag:= true;
      if frac (N/3) > 0 then  NumLines:= trunc (N/3) +1
      else                    NumLines:= trunc (N/3);
      NumPerLine:= 3;
    end
  else
    begin
      CDFlag:= 'D';
      NumLines:= N-1;
      NumPerLine:= N-1;
    end;
  readln (DataFile, A);                                 { comment line }
  I:= 1;
  J:= 2;
  for NL:=1 to NumLines do                              { process data lines }
    begin
      for L:=1 to LineLength do  A[L]:= ' ';
      readln (DataFile, A);
      L:= 1;
      for NPL:=1 to NumPerLine do
        if (CDFlag = 'C') and (I <= N) then
          begin
            X [I]:= ReadNumber (L, A);
            Y [I]:= ReadNumber (L, A);
            Z [I]:= 0;
            if ZFlag then  Z [I]:= ReadNumber (L, A);
            I:= I+1;
          end
        else
          if CDFlag = 'D' then
```

```
          begin
            Cost [I,J]:= ReadNumber (L, A);
            if (L >= LineLength) and (Cost [I,J] = 0) then
              begin
                readln (DataFile, A);
                L:= 1;
                Cost [I,J]:= ReadNumber (L, A);
              end;
            Cost [J,I]:= Cost [I,J];
            J:= J+1;
            if J = I then  J:= J+1;
            if J > N then
              begin
                I:= I+1;
                J:= I+1;
              end;
          end;
      if CDFlag = 'D' then
        NumPerLine:= NumPerLine -1;
    end;
end;          { ReadData }


procedure Timer;
type RegPack = record
                 AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: integer;
               end;
var  Regs     : RegPack;
     Hr, Mn, Sc, Fr: integer;
begin
  with Regs do
    begin
      AX:= $2C00;
      msdos (Regs);
      Hr:= hi (CX);
      Mn:= lo (CX);
      Sc:= hi (DX);
      Fr:= lo (DX);
    end;
  writeln (OutFile, ' time = ',Hr,':',Mn,':',Sc,'.',Fr:2);
end;    { Timer }


procedure SetupProblem;
var
    DataFile: text;
    DataName: NameType;
    I, MinI : OneNless1;
    J, MinJ : TwoN;
    NodeI   : OneN;
    MinCost, DX, DY, DZ: real;
    ZFlag              : boolean;
    X, Y, Z            : array [OneN] of real;
    Answer, Metric     : char;
begin
  repeat
    write ('Metric to be used (1, 2 or 0) ?');
    readln (Metric);
```



```
  until Metric in ['1', '2', '0'];
  write ('Result to be output to ?');
  readln (DataName);
  assign (OutFile, DataName);
  repeat
    writeln ('Data to be input by:');
    writeln ('                    C = Coordinates of nodes');
    writeln ('                    D = Distance between nodes');
    writeln ('                    F = stored in a File');
    write   ('                  ?');
    readln (Answer);
    Answer:= upcase (Answer);
  until Answer in ['C', 'D', 'F'];
  MinCost:= 9999;
  if Answer = 'F' then
    begin
      write    ('Data file name ?');                    { input file }
      readln   (DataName);
      assign   (DataFile, DataName);
      reset    (DataFile);
      ReadData (DataFile, Answer, X, Y, Z, ZFlag);
      close    (DataFile);
    end
  else
    begin
      repeat
        write('How many nodes are there ? (Maximum ',maxN,') ');
        readln (N);
      until (N > 3) and (N <= maxN);
      writeln;
      if Answer = 'D' then
        begin
          writeln('Please input costs / distances between nodes I and J');
          for I:=1 to N-1 do
            for J:=I+1 to N do
              begin
                write('   cost / distance between nodes ',I,' ',J,' ');
                readln (Cost [I,J]);
                Cost [J,I]:= Cost [I,J];              { costs are symmetric }
              end;
        end
      else
        begin
          ZFlag:= false;
          writeln ('z-coordinate to be entered ?');
          readln (Answer);
          if upcase (Answer) = 'Y' then  ZFlag:= true;
          Answer:= 'C';
          writeln ('Please input x, y (and z) coordinates of each node');
          for NodeI:=1 to N do
            if ZFlag then
              begin
                write('      x  y  z  coordinates of node ',NodeI,' :');
                readln (X [NodeI], Y [NodeI], Z [NodeI]);
              end
            else
              begin
                write('      x  y  coordinates of node ',NodeI,' :');
                readln (X [NodeI], Y [NodeI]);
                Z [NodeI]:= 0;
```

```
            end;
        end;
    end;
  MinCost:= 999999.0;
  for I:=1 to N-1 do
    begin
      Subtour [I,0]:= 0;
      Subtour [I,1]:= 0;
      for J:=I+1 to N do
        begin
          if Answer = 'C' then
            begin
              case Metric of
                '1': Cost[I,J]:=      abs(X[I]-X[J]) + abs(Y[I]-Y[J])
                                                     + abs(Z[I]-Z[J]);
                '2': Cost[I,J]:=sqrt (sqr(X[I]-X[J]) + sqr(Y[I]-Y[J])
                                                     + sqr(Z[I]-Z[J]));
                '0': begin
                       DX:= abs (X [I] - X [J]);
                       DY:= abs (Y [I] - Y [J]);
                       DZ:= abs (Z [I] - Z [J]);
                       if (DX >= DY) and (DX >= DZ) then  Cost[I,J]:= DX
                       else              if (DY >= DZ) then  Cost[I,J]:= DY
                                         else                Cost[I,J]:= DZ;
                     end;
              end;
              Cost [J,I]:= Cost [I,J];
            end;
          if Cost [I,J] < MinCost then
            begin
              MinCost:= Cost [I, J];
              MinI:= I;
              MinJ:= J;
            end;
        end;
    end;
  writeln;
  writeln ('Change floppy now if required.        Press any key to continue.');
  repeat until keypressed;
  rewrite (OutFile);
  writeln (OutFile, DataName);
  Timer;
  Subtour [N, 0]:= 0;
  Subtour [N, 1]:= 1;
  Subtour [MinI, 0]:= MinJ;
  Subtour [MinI, 1]:= MinJ;
  Subtour [MinJ, 0]:= MinI;
  Subtour [MinJ, 1]:= MinI;
end;  { SetupProblem }


procedure InsertNearestNode;
var
  MinCost, CompareCost: real;
  I, MinI: OneNless1;
  J, MinJ: TwoN;
  NewNode, OldNode, OtherOldNode: OneN;
begin                    { find the cheapest cost going 'away' from the subtour }
  MinCost:= 999999.0;
```



```
  for I:=1 to N-1 do
    for J:=I+1 to N do
    { for all edges with one node on the subtour and the other
      not, find which has the least cost. }
      begin
        if ((Subtour [I,0] > 0) and (Subtour [J,0] = 0)) or
           ((Subtour [I,0] = 0) and (Subtour [J,0] > 0)) then
          if Cost [I,J] < MinCost then
            begin
              MinCost:= Cost [I,J];
              MinI:= I;
              MinJ:= J;
            end;
      end;
                  { the node 'closest' to the current subtour has been found }
                  { insert the node into the subtour }
  if Subtour [MinI,0] = 0 then        { this is the new node }
    NewNode:= MinI
  else
    NewNode:= MinJ;
              { find nodes of the edge on the subtour which is to be replaced }
  MinCost:= 999999.0;
  for I:=1 to N-1 do
    for J:=I+1 to N do
    { for all edges on the subtour, find which one would cost
      the least if it were replaced. }
      if (Subtour [I,0] = J) or (SubTour [I,1] = J) then
        begin
          CompareCost:= Cost [I, NewNode] + Cost [J, NewNode]
                      - Cost [I, J];
          if CompareCost < MinCost then
            begin
              MinCost:= CompareCost;
              OldNode:= I;
              OtherOldNode:= J;
            end;
        end;
                                                { store the new subtour }
  Subtour [NewNode, 0] := OldNode;
  Subtour [NewNode, 1] := OtherOldNode;
  if Subtour [OldNode, 0] = OtherOldNode then
     Subtour [OldNode, 0] := NewNode
  else
     Subtour [OldNode, 1] := NewNode;
  if Subtour [OtherOldNode, 0] = OldNode then
     Subtour [OtherOldNode, 0] := NewNode
  else
     Subtour [OtherOldNode, 1] := NewNode;

end;    { InsertNearestNode }


procedure WriteBestTour;
var
  NodeI, Node, PreviousNode: OneN;
  Out4     : ZeroN;
  TourCost: real;
begin
```

```
  Out4:= 0;
  TourCost:= 0;
  PreviousNode:= 1;
  Node:= Subtour [1, 0];
  for NodeI:= 2 to N+1 do
    begin
      write (OutFile,'     ',PreviousNode:4,'  ',Node:4);
      Out4:= Out4 +1;
      if Out4 = 4 then
        begin
          Out4:= 0;
          writeln (OutFile);
        end;
      if Subtour [Node, 0] = PreviousNode then
        begin
          PreviousNode:= Node;
          TourCost:= TourCost + Cost [Node, Subtour [Node, 1]];
          Node:= Subtour [Node, 1];
        end
      else
        begin
          PreviousNode:= Node;
          TourCost:= TourCost + Cost [Node, Subtour [Node, 0]];
          Node:= Subtour [Node, 0];
        end;
    end;
  writeln (OutFile);
  writeln (OutFile, ' Cost of tour is ', TourCost:10:4);
  close (OutFile);
  writeln ('                    *** end of program ***');
end;    { WriteBestTour }


begin
  SetupProblem;
  for NodeCount:=3 to N do
        InsertNearestNode;
  Timer;
  WriteBestTour;
end.
```



```
program FarthestInsertion;
{$R-}
{
    method:     progressively include the farthest remaining node into
                the current subtour.
    conditions: non-Euclidean
                symmetric
                no missing edges
    input:      - how many nodes
                - for edge (i,j) connecting nodes i and j
                  input the cost/distance i,j
}

const
      maxN       = 101;
      Nless1     = 100;
      LineLength = 255;


type
      NameType  = string [12];
      LineType  = string [LineLength];
      OneNless1 = 1..Nless1;
      ZeroN     = 0..maxN;
      OneN      = 1..maxN;
      TwoN      = 2..maxN;
      CostMatrx = array [OneN, OneN] of real;
      TourArray = array [OneN, 0..1] of ZeroN;
      CoordArray= array [OneN] of real;

var
  OutFile     : text;
  Cost        : CostMatrx;
  N, NodeCount: OneN;
  Subtour     : TourArray;



Function ReadNumber (var LL: integer;
                     var B: LineType) : real;
var Int, OK : integer;
    PointFlag : boolean;
    R, RD, Decimal : real;
begin                                       { read one number }
  while (B[LL] = ' ') and (LL <= LineLength) do
    LL:= LL+1;                              { skip all spaces }
  Decimal:= 0.1;
  PointFlag:= false;
  R:=  0;
  RD:= 0;
  while (B [LL] <> ' ') and (LL <= LineLength) do
    begin                                   { find the number }
      if (B [LL]= '.') then
        PointFlag:=true
      else
        begin
          val (B [LL], Int, OK);
          if PointFlag = false then
            R:= R*10 + Int
          else
```

```
          begin
            RD:= RD + (Int * Decimal);
            Decimal:= Decimal * 0.1;
          end;
      end;
    LL:= LL+1;
  end;                                  { one value found }
ReadNumber:= R + RD;
end;          { ReadNumber }


Procedure ReadData (var DataFile: text;
                    var CDFlag  : char;
                    var X, Y, Z : CoordArray;
                    var ZFlag   : boolean);
var L: integer;
    NumLines, NL, NPL, I, J: OneN;
    NumPerLine: ZeroN;
    A: LineType;
begin
  readln (DataFile, A);
  L:= 4;
  N:= trunc (ReadNumber (L, A));
  ZFlag:= false;
  if A [1] = 'C' then
    begin
      CDFlag:= 'C';
      if A[2] = 'Z' then  ZFlag:= true;
      if frac (N/3) > 0 then  NumLines:= trunc (N/3) + 1
      else                    NumLines:= trunc (N/3);
      NumPerLine:= 3;
    end
  else
    begin
      CDFlag:= 'D';
      NumLines:= N-1;
      NumPerLine:= N-1;
    end;
  readln (DataFile, A);                                 { comment line }
  I:= 1;
  J:= 2;
  for NL:=1 to NumLines do                              { process data lines }
    begin
      for L:=1 to LineLength do  A[L]:= ' ';
      readln (DataFile, A);
      L:= 1;
      for NPL:=1 to NumPerLine do
        if (CDFlag = 'C') and (I <= N) then
          begin
            X [I]:= ReadNumber (L, A);
            Y [I]:= ReadNumber (L, A);
            Z [I]:= 0;
            if ZFlag then  Z [I]:= ReadNumber (L, A);
            I:= I+1;
          end
        else
          if CDFlag = 'D' then
            begin
              Cost [I,J]:= ReadNumber (L, A);
```



```
              if (L >= LineLength) and (Cost [I,J] = 0) then
                begin
                  readln (DataFile, A);
                  L:= 1;
                  Cost [I,J]:= ReadNumber (L, A);
                end;
              Cost [J,I]:= Cost [I,J];
              J:= J+1;
              if J = I then J:= J+1;
              if J > N then
                begin
                  I:= I+1;
                  J:= I+1;
                end;
            end;
      if CDFlag = 'D' then  NumPerLine:= NumPerLine -1;
    end;
end;          { ReadData }


procedure Timer;
type RegPack = record
                 AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: integer;
               end;
var  Regs    : RegPack;
     Hr, Mn, Sc, Fr: integer;
begin
  with Regs do
    begin
      AX:= $2C00;
      msdos (Regs);
      Hr:= hi (CX);
      Mn:= lo (CX);
      Sc:= hi (DX);
      Fr:= lo (DX);
    end;
  writeln (OutFile,' time = ',Hr,':',Mn,':',Sc,'.',Fr:2);
end;     { Timer }


procedure SetupProblem;
var
    I, MinI        : OneNless1;
    J, MinJ        : TwoN;
    NodeI          : OneN;
    MinCost, DX, DY, DZ: real;
    ZFlag          : boolean;
    X, Y, Z        : CoordArray;
    Answer, Metric: Char;
    DataFile       : text;
    DataName       : NameType;
begin
  repeat
    write ('Metric to be used (1, 2 or 0) ?');
    readln (Metric);
  until Metric in ['1', '2', '0'];
  write ('Result to be output to ?');
  readln (DataName);
```

```
assign (OutFile, DataName);
writeln;
repeat
  writeln ('Data to be input by:');
  writeln ('                    C = Coordinates of nodes');
  writeln ('                    D = Distance between nodes');
  writeln ('                    F = stored in a File');
  write   ('                   ?');
  readln (Answer);
  Answer:= upcase (Answer);
until Answer in ['C', 'D', 'F'];
if Answer = 'F' then
  begin
    write    ('Data file name ?');                    { input file }
    readln   (DataName);
    assign   (DataFile, DataName);
    reset    (DataFile);
    ReadData (DataFile, Answer, X, Y, Z, ZFlag);
    close    (DataFile);
  end
else
  begin
    repeat
      write('How many nodes are there ? (Maximum ',maxN,') ');
      readln (N);
    until (N > 3) and (N <= maxN);
    writeln;
    case Answer of
    'D': begin
           writeln('Please input costs / distances between nodes I and J');
           for I:=1 to N-1 do
             for J:=I+1 to N do
               begin
                 write('   cost / distance between nodes ',I,' ',J,' ');
                 readln (Cost [I,J]);
                 Cost [J,I]:= Cost [I,J];              { costs are symmetric }
               end;
         end;
    'C': begin
           ZFlag:= false;
           write ('z-coordinate to be entered ?');
           readln (Answer);
           if upcase (Answer) = 'Y' then  ZFlag:= true;
           Answer:= 'C';
           writeln('Please input x, y (and z) coordinates of each node');
           for NodeI:=1 to N do
             if ZFlag then
               begin
                 write ('      x  y  z  coordinates of node ',NodeI,' :');
                 readln (X [NodeI], Y [NodeI], Z [NodeI]);
               end
             else
               begin
                 write ('      x  y  coordinates of node ',NodeI,' :');
                 readln (X [NodeI], Y [NodeI]);
                 Z [NodeI]:= 0;
               end;
         end;
    end;
  end;
```



```
MinCost:= 9999;                              { find starting edge }
for I:=1 to N-1 do
  begin
    Subtour [I,0]:= 0;
    Subtour [I,1]:= 0;
    for J:=I+1 to N do
      begin
        if Answer = 'C' then
          begin
            DX:= abs (X[I] - X[J]);
            DY:= abs (Y[I] - Y[J]);
            DZ:= abs (Z[I] - Z[J]);
            case Metric of
            '1': Cost[I,J]:=      DX    + DY    + DZ;
            '2': Cost[I,J]:=sqrt (DX*DX + DY*DY + DZ*DZ);
            '0': begin
                   if (DX >= DY) and (DX >= DZ) then  Cost[I,J]:= DX
                   else                               Cost[I,J]:= DY
                                                      Cost[I,J]:= DZ;
                 end;
            end;
            Cost [J,I]:= Cost [I,J];
          end;
        if Cost[I,J] < MinCost then
          begin
            MinCost:= Cost [I,J];
            MinI:= I;
            MinJ:= J;
          end;
      end;
  end;
writeln;
writeln ('Change floppy now if required.        Press any key to continue.');
repeat until keypressed;
rewrite (OutFile);
writeln (OutFile, DataName);
Timer;
Subtour [N,0]:= 0;
Subtour [N,1]:= 0;
Subtour [MinI, 0]:= MinJ;
Subtour [MinI, 1]:= MinJ;
Subtour [MinJ, 0]:= MinI;
Subtour [MinJ, 1]:= MinI;
end;  { SetupProblem }


{*****************************************************************************}

procedure InsertFarthestNode;
var
  MaxMinCost, TestCost: real;
  I, MaxMinI: OneNless1;
  J, MaxMinJ: TwoN;
  NewNode, OldNode, OtherOldNode: OneN;
begin                    { find the cheapest cost going 'away' from the subtour }
  MaxMinCost:= -9999;
  for I:=1 to N-1 do
    for J:=I+1 to N do
    { for all edges with one node on the subtour and the other
```

```
    not, find which has the most cost. }
    begin
      if ((Subtour [I,0] > 0) and (Subtour [J,0] = 0)) or
         ((Subtour [I,0] = 0) and (Subtour [J,0] > 0)) then
        if Cost [I,J] > MaxMinCost then
          begin
            MaxMinCost:= Cost [I,J];
            MaxMinI:= I;
            MaxMinJ:= J;
          end;
    end;
              { the node 'farthest' from the current subtour has been found }
              { insert the node into the subtour }
  if Subtour [MaxMinI,0] = 0 then                  { this is the new node }
    NewNode:= MaxMinI
  else
    NewNode:= MaxMinJ;
            { find nodes of the edge on the subtour which is to be replaced }
  MaxMinCost:= 99999.0;
  for I:=1 to N do
    if (I <> NewNode) and (Subtour [I,0] > 0) then
    { for all edges with one node on the subtour and the other     }
    { being the new node, find which has the least insertion cost. }
      begin
        TestCost:= Cost [I,NewNode] + Cost [Subtour [I,0], NewNode]
                                    - Cost [Subtour [I,0], I];
             { TestCost:= Cost [I,NewNode] + Cost [Subtour [I,0], NewNode]; }
             { TestCost:= (TestCost  - Cost [Subtour [I,0], I]) }
             {           * TestCost  / Cost [Subtour [I,0], I]; }
             { result worse if use Difference * Ratio Test.     }
        if TestCost < MaxMinCost then
          begin
            MaxMinCost:= TestCost;
            OldNode:=I;
            OtherOldNode:= Subtour [I,0];
          end;
        TestCost:= Cost [I,NewNode] + Cost [Subtour [I,1], NewNode]
                                    - Cost [Subtour [I,1], I];
             { TestCost:= Cost [I,NewNode] + Cost [Subtour [I,1], NewNode]; }
             { TestCost:= (TestCost  - Cost [Subtour [I,1], I]) }
             {           * TestCost  / Cost [Subtour [I,1], I]; }
        if TestCost < MaxMinCost then
          begin
            MaxMinCost:= TestCost;
            OldNode:=I;
            OtherOldNode:= Subtour [I,1];
          end;
      end;
                                          { store the new subtour }
  Subtour [NewNode, 0] := OldNode;
  Subtour [NewNode, 1] := OtherOldNode;
  if Subtour [OldNode, 0] = OtherOldNode then
     Subtour [OldNode, 0] := NewNode
  else
     Subtour [OldNode, 1] := NewNode;
  if Subtour [OtherOldNode, 0] = OldNode then
     Subtour [OtherOldNode, 0] := NewNode
  else
     Subtour [OtherOldNode, 1] := NewNode;
end;     { InsertFarthestNode }
```



```
procedure WriteBestSubtour;
var
  NodeI, Node, PreviousNode: OneN;
  Out4     : ZeroN;
  TourCost: real;
begin
  Timer;
  Out4:= 0;
  TourCost:= 0;
  PreviousNode:= 1;
  Node:= Subtour [1, 0];
  for NodeI:= 2 to N+1 do
    begin
      write (OutFile,'     ',PreviousNode:4,'   ',Node:4);
      Out4:= Out4 + 1;
      if Out4 = 4 then
        begin
          Out4:= 0;
          writeln (OutFile);
        end;
      TourCost:= TourCost + Cost [PreviousNode, Node];
      if Subtour [Node, 0] = PreviousNode then
        begin
          PreviousNode:= Node;
          Node:= Subtour [Node, 1];
        end
      else
        begin
          PreviousNode:= Node;
          Node:= Subtour [Node, 0];
        end;
    end;
  writeln (OutFile);
  writeln (OutFile, ' Cost of tour is ', TourCost:11:4);
  close (OutFile);
  writeln('                *** end of program ***');
end;     { WriteBestSubtour }



begin
  SetupProblem;
  for NodeCount:=3 to N do
        InsertFarthestNode;
  WriteBestSubtour;
end.
```

```
program Greedy;
{$R-}
{
    method:      add lowest costing edge, such that no cycle are formed
                 except when the last edge is added.
    conditions:  non-Euclidean
                 symmetric
                 may have missing edges
                 (may have negative costs)
    input:       - how many nodes
                 - input by distance:
                   for edge (i,j) connecting nodes i and j
                   input either the cost/distance i,j or
                   (if edge does not exist) value 9999.
                 - input by coordinates:
                   for each node i give it's coordinates.


const
      maxN       = 101;
      maxNminus1 = 100;
      LineLength = 250;

type
      NameType  = string [12];
      LineType  = string [LineLength];
      ZeroN     = 0..maxN;
      OneN      = 1..maxN;
      TwoN      = 2..maxN;
      CoordArray= array [OneN] of real;
      CostMatrx = array [OneN, OneN] of real;
      TourArray = array [0..1, OneN] of ZeroN;

var
  OutFile  : text;
  PathFlag : boolean;
  Totalcost: real;

  N        : ZeroN;
  Cost     : CostMatrx;
  V        : TourArray;



Function ReadNumber (var LL: integer;
                     var B: LineType) : real;
var Int, OK : integer;
    PointFlag : boolean;
    R, RD, Decimal : real;
begin                                   { read one number }
  while (B [LL] = ' ') and (LL <= LineLength) do
    LL:= LL+1;                          { skip all spaces }
  Decimal:= 0.1;
  PointFlag:= false;
  R:=  0;
  RD:= 0;
  while (B [LL] <> ' ') and (LL <= LineLength) do
    begin                               { find the number }
      if (B [LL]= '.') then
```



```
        PointFlag:=true
      else
        begin
          val (B [LL], Int, OK);
          if PointFlag = false then
            R:= R*10 + Int
          else
            begin
              RD:= RD + (Int * Decimal);
              Decimal:= Decimal * 0.1;
            end;
        end;
      LL:= LL+1;
    end;                                { one value found }
  ReadNumber:= R + RD;
end;          { ReadNumber }


Procedure ReadData (var DataFile: text;
                    var CDFlag  : char;
                    var X, Y, Z : CoordArray;
                    var ZFlag   : boolean);
var L: integer;
    NumLines, NL, NPL, I, J: OneN;
    NumPerLine: ZeroN;
    A: LineType;
begin
  readln (DataFile, A);
  L:= 4;
  N:= trunc (ReadNumber (L, A));
  ZFlag:= false;
  if A [1] = 'C' then
    begin
      CDFlag:= 'C';
      if A [2] = 'Z' then     ZFlag:= true;
      if frac (N/3) > 0 then  NumLines:= trunc (N/3) + 1
      else                    NumLines:= trunc (N/3);
      NumPerLine:= 3;
    end
  else
    begin
      CDFlag:= 'D';
      NumLines:= N-1;
      NumPerLine:= N-1;
    end;
  readln (DataFile, A);                                 { comment line }
  I:= 1;
  J:= 2;
  for NL:=1 to NumLines do                              { process data lines }
    begin
      for L:=1 to LineLength do  A[L]:= ' ';
      readln (DataFile, A);
      L:= 1;
      for NPL:=1 to NumPerLine do
        if (CDFlag = 'C') and (I <= N) then
          begin
            X [I]:= ReadNumber (L, A);
            Y [I]:= ReadNumber (L, A);
            Z [I]:= 0;
```

```
          if ZFlag then Z [I]:= ReadNumber (L, A);
          I:= I+1;
        end
      else
        if CDFlag = 'D' then
          begin
            Cost [I,J]:= ReadNumber (L, A);
            if (L = LineLength) and (Cost [I,J] = 0) then
              begin
                readln (DataFile, A);
                L:= 1;
                Cost [I,J]:= ReadNumber (L, A);
              end;
            J:= J+1;
            if J = 1 then  J:= J+1;
            if J > N then
              begin
                I:= I+1;
                J:= I+1
              end;
          end;
    if CDFlag = 'D' then
      NumPerLine:= NumPerLine -1;
  end;
end;          { ReadData }



procedure SetupProblem;
var
    DataFile       : text;
    DataName       : NameType;
    I, MinI, NodeI: OneN;
    J, MinJ        : TwoN;
    DX, DY, DZ     : real;
    ZFlag          : boolean;
    X, Y, Z        : array [OneN] of real;
    Answer, Metric: char;
begin
  repeat
    write ('Metric to be used (1, 2 or 0) ?');
    readln (Metric);
  until Metric in ['1','2','0'];
  write ('Result to be output to ?');
  readln (DataName);
  assign (OutFile, DataName);
  writeln;
  repeat
    writeln ('Data to be input by:');
    writeln ('                    C = Coordinates of nodes');
    writeln ('                    D = Distance between nodes');
    writeln ('                    F = stored in a File');
    write   ('                   ?');
    readln (Answer);
    Answer:= upcase (Answer);
  until Answer in ['C', 'D', 'F'];
  if Answer = 'F' then
    begin
```



```
      write    ('Data file name ?');                    { input file }
      readln   (DataName);
      assign   (DataFile, DataName);
      reset    (DataFile);
      ReadData (DataFile, Answer, X, Y, Z, ZFlag);
      close    (DataFile);
    end
  else
    begin
      repeat
        write('How many nodes are there ? (Maximum ',maxN,') ');
        readln (N);
      until (N > 3) and (N <= maxN);
      writeln;
      if Answer = 'D' then
        begin
          writeln('Please input costs / distances between nodes I and J');
          writeln('If a cost / distance does not exist, input 9999');
          for I:=1 to N-1 do
            for J:=I+1 to N do
              begin
                write('   cost / distance between nodes ',I,' ',J,' ');
                readln (Cost [I,J]);
              end;
        end
      else
        begin
          ZFlag:= false;
          write  ('z-coordinate to be entered ?');
          readln (Answer);
          if upcase (Answer) = 'Y' then ZFlag:= true;
          Answer:= 'C';
          writeln('Please input x, y (and z) coordinates of each node');
          for NodeI:=1 to N do
            if ZFlag then
              begin
                write('      x  y  z  coordinates of node ',NodeI,' :');
                readln (X [NodeI], Y [NodeI], Z [NodeI]);
              end
            else
              begin
                write('      x  y  coordinates of node ',NodeI,' :');
                readln (X [NodeI], Y [NodeI]);
                Z [NodeI]:= 0;
              end;
        end;
    end;
  if Answer = 'C' then
   for I:=1 to N-1 do
    for J:=I+1 to N do
     begin
       DX:= abs (X[I] - X[J]);
       DY:= abs (Y[I] - Y[J]);
       DZ:= abs (Z[I] - Z[J]);
       case Metric of
       '1': Cost[I,J]:=      DX    + DY    + DZ;
       '2': Cost[I,J]:=sqrt (DX*DX + DY*DY + DZ*DZ);
       '0': begin
              if (DX >= DY) and (DX >= DZ) then  Cost[I,J]:= DX
              else                if (DY >= DZ) then  Cost[I,J]:= DY
```

```
                        else                Cost[I,J]:= DZ;
          end;
       end;
       Cost [J,I]:= Cost [I, J];
     end;
  writeln;
  writeln('Change floppy now if required.          Press any key to continue.');
  repeat until keypressed;
  rewrite (OutFile);,
  writeln (OutFile, DataName);
end;  { SetupProblem }



procedure Timer;
type  RegPack = record
                  AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: integer;
                end;
var   Regs    : RegPack;
      Hr, Mn, Sc, Fr: integer;
begin
  with Regs do
    begin
      AX:= $2C00;
      msdos (Regs);
      Hr:= hi (CX);
      Mn:= lo (CX);
      Sc:= hi (DX);
      Fr:= lo (DX);
    end;
  writeln (OutFile, ' time = ',Hr,':',Mn,':',Sc,'.',Fr:2);
end;    { Timer }



procedure CheckForSubtour (Left, Right: OneN;
                           var NoCycleFlag: boolean);
var NextNode: OneN;
begin                          { find last node on path from 'left' node }
  NextNode:= V [0, Left];
  while V [1, NextNode] > 0 do
    if V [0, NextNode] = Left then
      begin
        Left:= NextNode;
        NextNode:= V [1, Left];
      end
    else
      begin
        Left:= NextNode;
        NextNode:= V [0, Left];
      end;
  if NextNode = Right then          { check if last node is the 'right' node }
    NoCycleFlag:= false;
end;     { CheckForSubtour }



procedure FindSpanningPath (var OKflag: boolean);
var
  NoSubtour             : boolean;
```



```
  LowestCost            : real;
  I, J, LowestI, LowestJ: OneN;
  NodesOnPath, Out4     : ZeroN;
begin
  Out4:= 0;
  Timer;
  for I:=1 to N do        { initially there are no edges }
    begin
      V [0, I]:= 0;
      V [1, I]:= 0;
    end;
  TotalCost:=0;
  NodesOnPath:=0;
  OKflag:=true;
  while (OKflag = true) and (NodesOnPath < N-2) do
  { while there are valid edges left, connect up all nodes
  into a path, with two nodes left as the start and end
  nodes of the path                                        }
  begin
    LowestCost:=9999;
    for I:=1 to N-1 do
      for J:=I+1 to N do
       begin
        if (V [1, I] = 0) and
           (V [1, J] = 0) and                          { 'open' edges remain }
           (Cost[I,J] < LowestCost) then                       { check cost }
          begin
            NoSubtour:= true;
            if (V [0, I] > 0) and
               (V [0, J] > 0) then
               CheckForSubtour (I, J, NoSubtour);
            if NoSubtour then
              begin
                LowestCost:= Cost[I,J];
                LowestI:= I;
                LowestJ:= J;
              end;
          end;
       end;
    if LowestCost < 9999 then                  {include edge in spanning path}
      begin
        write (OutFile, '     ',LowestI:4,'   ',LowestJ:4);
        Out4:= Out4 +1;
        if Out4 = 4 then
          begin
            Out4:= 0;
            writeln (outFile);
          end;
        if V [0, LowestI] = 0 then  V [0, LowestI]:= LowestJ
        else
          begin
            V [1, LowestI]:= LowestJ;
            NodesOnPath:= NodesOnPath + 1;
          end;
        if V [0, LowestJ] = 0 then  V [0, LowestJ]:= LowestI
        else
          begin
            V [1, LowestJ]:= LowestI;
            NodesOnPath:= NodesOnPath + 1;
          end;
```

```
      TotalCost:= TotalCost + Cost [LowestI,LowestJ];
      Cost [LowestI,LowestJ]:= 9999;
    end
  else                                        {no more valid edges are left}
    OKflag:= false;
end;
nd;    { FindSpanningPath }


rocedure FindTour (OKflag: boolean);
ar
 I, J, II, Mismatch: ZeroN;
egin
 if OKflag = true then
 { find the final edge which will connect the only two remaining nodes
 that have less than two edges entering them.                        }
   begin
     II:=1;
     I:= 0;
     J:= 0;
     Mismatch:=0;
     while (II <= N) and ((I=0) or (J=0)) do
       begin                    {search for nodes with number of edges < 2}
         if V [1, II] = 0 then
           begin
             if I=0 then I:=II else J:=II;
             if (I>0) and (J>0) then
               begin
                 (* to check Cost, need to first check that
                    I and J are in the correct order *)
                 if Cost [I,J] = 9999 then
                   begin
                   { this edge is not valid }
                     Mismatch:=I;
                     J:=0;
                   end;
               end;
           end;
         II:= II+1;
         if (II = N+1) and (Mismatch > 0) and (J=0) then
           begin
             { check if any other edges have been missed }
             II:= Mismatch+1;
             I:=0;
           end;
       end;
     if (I > 0) and (J > 0) then
       begin
         { the final edge changing the path into a tour has been found }
         writeln (OutFile, '     ',I:4,'  ',J:4);  writeln (OutFile);
         TotalCost:= TotalCost + Cost [I,J];
         writeln (OutFile, 'Total cost of this tour is ',TotalCost:12:4);
       end
     else
       begin
         writeln (OutFile);
         writeln (OutFile, 'Run out of valid edges before tour could be complet
');
         writeln (OutFile, 'Total cost of above path is ',TotalCost:10:4);
       end;
```



```
     end
   else
     begin
       writeln (OutFile);
       writeln (OutFile, 'Run out of valid edges before tour could be completed.'
);
       writeln (OutFile, 'Total cost of above path is ',TotalCost:10:4);
     end;
   Timer;
   close (OutFile);
   writeln ('        *** end of program ***');
end;    { FindTour }


begin
  SetupProblem;
  FindSpanningPath (PathFlag);
  FindTour (PathFlag);
end.
```

```
program ConvergingHulls;
{$R-}
{
    method:      - find progressively decreasing convex hulls until
                   no nodes remain which are not on a hull.
                 - join the hulls in some minimum way.
    conditions: Euclidean
    input:       - how many nodes
                 - for each node input its x- and y-coordinates.
}

const
      maxN       = 101;
      LineLength = 180;

type
      NameType   = string [12];
      LineType   = string [LineLength];
      ZeroN      = 0..maxN;
      OneN       = 1..maxN;
      CoordArray = array [OneN] of real;
      TourMatrx  = array [0..1, OneN] of ZeroN;


var   OutFile                : text;
      N, NodeCount, HullNum: ZeroN;
      TimeStart              : real;
      NodesRemain, CFlag     : boolean;
      Metric                 : char;
      X, Y, Z                : CoordArray;
      V                      : TourMatrx;
      H                      : array [OneN] of OneN;


Function ReadNumber (var LL: integer;
                     var B: LineType) : real;
var Int, OK : integer;
    PointFlag : boolean;
    R, RD, Decimal : real;
begin                                   { read one number }
  while (B [LL] = ' ') and (LL <= LineLength) do
    LL:= LL+1;                          { skip all spaces }
  Decimal:= 0.1;
  PointFlag:= false;
  R:= 0;
  RD:= 0;
  while (B [LL] <> ' ') and (LL <= LineLength) do
    begin                               { find the number }
      if (B [LL]= '.') then
        PointFlag:=true
      else
        begin
          val (B [LL], Int, OK);
          if PointFlag = false then
            R:= R*10 + Int
          else
            begin
              RD:= RD + (Int * Decimal);
              Decimal:= Decimal * 0.1;
```



```
            end;
        end;
      LL:= LL+1;
    end;                                { one value found }
  ReadNumber:= R + RD;
end;          { ReadNumber }


Procedure ReadData (var DataFile: text;
                    var ZFlag   : boolean);
var L: integer;
    NumLines, NL, NPL, I: OneN;
    NumPerLine: ZeroN;
    A: LineType;
begin
  readln (DataFile, A);
  if A[1] <> 'C' then
    begin
      writeln; writeln;
      writeln ('****** Program requires coordinates of nodes !      ******');
      writeln ('****** The given data is not stored as coordinates. ******');
      CFlag:= false;
    end
  else
    begin
      L:= 4;
      N:= trunc (ReadNumber (L, A));
      ZFlag:= false;
      if A [2] = 'Z' then ZFlag:= true;
      if frac (N/3) > 0 then  NumLines:= trunc (N/3) + 1
      else                    NumLines:= trunc (N/3);
      NumPerLine:= 3;
      readln (DataFile, A);                                  { comment line }
      I:= 1;
      for NL:=1 to NumLines do                               { process data lines }
        begin
          for L:=1 to LineLength do  A[L]:= ' ';
          readln (DataFile, A);
          L:= 1;
          for NPL:=1 to NumPerLine do
            if I <= N then
              begin
                X [I]:= ReadNumber (L, A);
                Y [I]:= ReadNumber (L, A);
                Z [I]:= 0;
                if ZFlag then Z [I]:= ReadNumber (L, A);
                I:= I+1;
              end;
        end;
      for I:=1 to N do
        begin
          V [0,I]:= 0;
          V [1,I]:= 0;
        end;
    end;
end;          { ReadData }
```

```
procedure SetupProblem;
var
    DataFile: text;
    DataName: NameType;
    I       : OneN;
    ZFlag   : boolean;
    Answer  : char;
begin
  repeat
    write ('Metric to be used (1, 2, 0) ?');
    readln (Metric);
  until Metric in ['1', '2', '0'];
  write ('Results to be output to ?');
  readln (DataName);
  assign (OutFile, DataName);
  writeln;
  repeat
    writeln ('Data to be input by:');
    writeln ('                    C = Coordinates of nodes');
    writeln ('                    F = stored in a File');
    write   ('                   ?');
    readln (Answer);
    Answer:= upcase (Answer);
  until Answer in ['C', 'F'];
  CFlag:= true;
  if Answer = 'F' then
    begin
      write    ('Data file name ?');                    { input file }
      readln   (DataName);
      assign   (DataFile, DataName);
      reset    (DataFile);
      ReadData (DataFile, ZFlag);
      close    (DataFile);
    end
  else
    begin
      repeat
        write('How many nodes are there ? (Maximum ',maxN,') ');
        readln (N);
      until (N > 3) and (N <= maxN);
      writeln;
      ZFlag:= false;
      write ('z-coordinate to be entered ?');
      readln (Answer);
      if upcase (Answer) = 'Y' then  ZFlag:= true;
      writeln ('Please input x-, y- (and z-) coordinates of each point');
      for I:=1 to N do
        begin
          if ZFlag then
            begin
              write('   x  y  z  coordinates of point ',I,':');
              readln (X [I], Y [I], Z [I]);
            end
          else
            begin
              write('   x  y coordinates of point ',I,':');
              readln (X [I], Y [I]);
              Z [I]:= 0;
            end;
```



```
          V [0,I]:= 0;
          V [1,I]:= 0;
        end;
    end;
  if CFlag then
    begin
      NodeCount:= N;
      NodesRemain:= true;
      HullNum:= 0;
      TimeStart:= 0.0;
      writeln;
      writeln ('Change floppy now if required.        Press any key to continue
');
      repeat until keypressed;
      rewrite (OutFile);
      writeln (OutFile, DataName);
    end;
end;  { SetupProblem }


procedure Timer;
type RegPack = record
                 AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : integer;
               end;
var  Regs     : RegPack;
     TimeNow : real;
     Hr, Mn, Sc, Fr: integer;
begin
  with Regs do
    begin
      AX:= $2C00;
      msdos (Regs);
      Hr:= hi (CX);
      Mn:= lo (CX);
      Sc:= hi (DX);
      Fr:= lo (DX);
    end;
  if TimeStart = 0 then
    begin
      Cflag:= true;
      TimeStart:= Mn/60 + Hr;
      writeln (OutFile, ' time = ',Hr,':',Mn,':',Sc,'.',Fr);
    end
  else
    begin
      TimeNow:= Mn/60 + Hr;
      if (TimeNow - TimeStart) >= 2 then
        begin
          CFlag:= false;
          NodesRemain:= false;
          writeln (OutFile, ' ***** ran out of time *****');
        end;
    end;
end;        { Timer }



Function Dist (var A, B: OneN)  : real;
var  DX, DY, DZ: real;
begin
  DX:= abs (X[A] - X[B]);
```

```
  DY:= abs (Y[A] - Y[B]);
  DZ:= abs (Z[A] - Z[B]);
  case Metric of
  '1': Dist:=      DX    + DY    + DZ;
  '2': Dist:=sqrt (DX*DX + DY*DY + DZ*DZ);
  '0': begin
         if (DX >= DY) and (DX >= DZ) then  Dist:= DX
         else           if (DY >= DZ) then  Dist:= DY
                        else               Dist:= DZ;
       end;
  end;
end;    { Dist }


Procedure ConvexHull (var Continue: boolean);
var  I, J, Node1, Node2, Node3, StartNode: OneN;
     X1, Y1, X2, Y2, Min, Max, CosAngle: real;
           { Find the convex hull of a set of points.
             Input: x- and y-coordinates of the points.
             Method: For each remaining node not on the hull,
                     find the angle between it and an edge on the hull.
                     The largest angle formed indicates the next
                     node on the hull.                              }
begin
HullNum:=HullNum +1;
  Min:= 99990.0;                      { find the first node on the hull }
  for I:=1 to N do
    if (V [0,I] = 0) and (X [I] < Min) then
      begin
        Min:= X [I];
        Node2:= I;
      end;
  if NodeCount > 2 then
    begin
      Min:=  2;                       { find first two edges of the hull }
      Max:= -2;
      for I:=1 to N do
        if (V [0,I] = 0) and (I <> Node2) then
          begin
            X1:= X [I] - X [Node2];
            Y1:= Y [I] - Y [Node2];
            Y2:= Y [Node2];
            CosAngle:= (- Y1 * Y2) / (sqrt (X1*X1 + Y1*Y1) *  Y2);
            if CosAngle < Min then
              begin
                Node1:= I;
                Min:= CosAngle;
              end
            else
              if CosAngle = Min then
                begin
                  X1:= Dist (Node2, I);
                  X2:= Dist (Node2, Node1);
                  if Dist (Node2, I) < Dist (Node2, Node1) then
                    Node1:= I;
                end;
            if CosAngle > Max then
              begin
                Node3:= I;
```



```
                Max:= CosAngle;
              end
            else
              if CosAngle = Max then
                begin
                  X1:= Dist (Node2, I);
                  X2:= Dist (Node2, Node3);
                  if X1 < X2 then
                    Node3:= J;
                end;
          end;
      V [0, Node1]:= Node2;
      V [0, Node2]:= Node1;
      V [1, Node2]:= Node3;
      V [0, Node3]:= Node2;
      H [Node1]:= HullNum;
      H [Node2]:= HullNum;
      H [Node3]:= HullNum;
      StartNode:= Node1;
      Node1:= Node2;
      Node2:= Node3;
      repeat                                { check all remaining angles but one }
          Min:= 2;
          X1:= X [Node1] - X [Node2];
          Y1:= Y [Node1] - Y [Node2];
          for J:=1 to N do
            if (V [1,J] = 0) and (J <> Node1) and (J <> Node2) then
              begin
                X2:= X [J] - X [Node2];
                Y2:= Y [J] - Y [Node2];
                CosAngle:= ((X1 * X2)  +  (Y1 * Y2)) /
                           (sqrt (X1*X1 + Y1*Y1) * sqrt (X2*X2 + Y2*Y2));
                if CosAngle < Min then
                  begin
                    Node3:= J;
                    Min:= CosAngle;
                  end
                else
                  if CosAngle = Min then        { check distance }
                    begin
                      X2:= Dist (Node2, J);
                      Y2:= Dist (Node2, Node3);
                      if X2 < Y2 then
                        Node3:= J;
                    end;
              end;
          V [1, Node2]:= Node3;
          if V [0, Node3] = 0 then
             V [0, Node3]:= Node2
          else
             V [1, Node3]:= Node2;
          H [Node3]:= HullNum;
          Node1:= Node2;
          Node2:= Node3;
      until Node3 = StartNode;
      NodeCount:= 0;                        { check if any nodes left inside the hu
1 }
      for I:=1 to N do
       if V [0,I] = 0 then NodeCount:= NodeCount + 1;
      if NodeCount = 0 then Continue:= false;
end
```

```
  else                          { two or one node left }
    begin
      Continue:= false;
      H [Node2]:= HullNum;
      if NodeCount = 1 then
        begin
          V [0, Node2]:= Node2;
          V [1, Node2]:= Node2;
        end
      else
        begin
          Node1:=1;
          while (V [0,Node1] <> 0) or (Node1 = Node2) do
            Node1:= Node1 +1;
          H [Node2]:= HullNum;
          V [0, Node1]:= Node2;
          V [1, Node1]:= Node2;
          V [0, Node2]:= Node1;
          V [1, Node2]:= Node1;
        end;
    end;
  if Continue then Timer;
end;    { Convex Hull }



procedure JoinHulls;
var
  Min: array [1..4] of OneN;
  StartOuter, StartInner, I,
  NodeO1, NodeO2, NodeI1, NodeI2: OneN;
  NodeOPrev, NodeIPrev: ZeroN;
  { O1    = a node on outer hull,
    O2    = adjacent node on outer hull,
    OPrev = previous node on outer hull,
    I1    = a node on inner hull,
    I2    = adjacent node on inner hull,
    IPrev = previous node on inner hull. }
  DistOO, DistOI, Cost, MinCost: real;
  { OO= for two nodes on outer hull,
    OI= for two nodes on inner hull + OO. }
begin                           { combine hulls by finding the 'cheapest cost' join }
  StartOuter:= 1;
  while H [StartOuter] <> 1 do
    StartOuter:= StartOuter +1;
  NodeO1     := StartOuter;
  I:= 2;
  while (I <= HullNum) and (CFlag) do
    begin
      NodeOPrev := 0;
      StartInner:= 1;
      while H [StartInner] <> I do
        StartInner:= StartInner +1;
      NodeI1:= StartInner;
      MinCost := 9999;
      repeat                                { all possibilities on outer hull }
          if    (V [0, NodeO1] <> NodeOPrev) and
             (H [V [0, NodeO1]] = I-1)  then
              NodeO2:= V [0, NodeO1]
          else
              NodeO2:= V [1, NodeO1];
          DistOO:= Dist (NodeO1, NodeO2);
          NodeIPrev:= 0;
          repeat                                { all possibilities on inner hull }
              if NodeIPrev = V [0, NodeI1] then
                 NodeI2:= V [1, NodeI1]
              else
                 NodeI2:= V [0, NodeI1];
              DistOI:= DistOO + Dist (NodeI1, NodeI2);
              Cost:= Dist (NodeO1, NodeI1) + Dist (NodeO2, NodeI2)
                     - DistOI;
              if Cost < MinCost then
                begin
                  MinCost:= Cost;
                  Min [1]:= NodeO1;
                  Min [2]:= NodeO2;
                  Min [3]:= NodeI1;
                  Min [4]:= NodeI2;
                end;
              Cost:= Dist (NodeO1, NodeI2) + Dist (NodeO2, NodeI1)
                     - DistOI;
              if Cost < MinCost then
                begin
                  MinCost:= Cost;
                  Min [1]:= NodeO1;
                  Min [2]:= NodeO2;
                  Min [3]:= NodeI2;
                  Min [4]:= NodeI1;
                end;
              NodeIPrev:= NodeI1;
              NodeI1:= NodeI2;
          until NodeI1 = StartInner;
          NodeOPrev:= NodeO1;
          NodeO1:= NodeO2;
      until NodeO1 = StartOuter;
                                        { connect the outer and inner hull }
      if V [0, Min [1]] = Min [2] then
         V [0, Min [1]]:= Min [3]
      else
         V [1, Min [1]]:= Min [3];
      if V [0, Min [2]] = Min [1] then
         V [0, Min [2]]:= Min [4]
      else
         V [1, Min [2]]:= Min [4];
      if V [0, Min [3]] = Min [4] then
         V [0, Min [3]]:= Min [1]
      else
         V [1, Min [3]]:= Min [1];
      if V [0, Min [4]] = Min [3] then
         V [0, Min [4]]:= Min [2]
      else
         V [1, Min [4]]:= Min [2];
      NodeO1:= Min [3];
      StartOuter:= Min [4];
      Timer;
      I:= I +1;
    end;
  TimeStart:= 0.0;
end;    { JoinHulls }
```

```
procedure WriteBestTour;
var
  Node, PrevNode: OneN;
  Out4: ZeroN;
  TourCost: real;
begin
  if CFlag then
    begin                    { a tour was found }
      TourCost:= 0;
      Out4:= 0;
      PrevNode:= 1;
      Node:= V [1, 1];
      repeat
          write(OutFile, '   ', PrevNode:4, Node:4);
          Out4:= Out4 +1;
          if Out4 = 4 then
            begin
              Out4:= 0;
              writeln (OutFile);
            end;
          TourCost:= TourCost + Dist (PrevNode, Node);
          if V [0, Node] = PrevNode then
            begin
              PrevNode:= Node;
              Node:= V [1, Node];
            end
          else
            begin
              PrevNode:= Node;
              Node:= V [0, Node];
            end;
      until PrevNode = 1;
      writeln (OutFile);
      writeln (OutFile, ' Cost of tour is ', TourCost:11:4);
    end
  else
    begin                  { ran out of time }
      for Node:=1 to N do
        begin
          write (OutFile, 'V [',Node,'] = ');
          if V [0, Node] = 0 then  write (OutFile, '      ')
          else                     write (OutFile, V [0, Node]:5);
          if V [1, Node] <> 0 then write (OutFile, V [1, Node]:5);
          writeln (OutFile);
        end;
      writeln (OutFile, 'Number of hulls found : ', HullNum);
    end;
  close (OutFile);
  writeln ('                  *** end of program ***');
end;     { WriteBestTour }


begin
  SetupProblem;
  if CFlag then
    begin
      Timer;
      while NodesRemain do
          ConvexHull (NodesRemain);
      JoinHulls;
      Timer;
      WriteBestTour;
    end;
end.
```

```
program NearestNghbr;
{$R-}
{
    method:      starting from one node, go to the next 'closest' node
                 until all nodes have been visited.
    conditions:  non-Euclidean
                 symmetric or non-symmetric
                 may have missing edges
                 (may have negative costs)
    input:       - wether problem non-symmetric or symmetric
                 - how many nodes
                 - input by distances:
                   for edge (i,j) connecting nodes i and j
                   input either the cost/distance i,j or
                   (if edge does not exist) value 9999.
                 - input by coordinates:
                   for each node i give the coordinates.
}

const
      maxN       = 101;
      LineLength = 255;


type
      NameType   = string [12];
      LineType   = string [LineLength];
      ZeroN      = 0..maxN;
      OneN       = 1..maxN;
      CostArray  = array [OneN] of real;
      CostMatrx  = array [OneN, OneN] of real;
      TourMatrx  = array [0..1, OneN] of ZeroN;

var
  OutFile              : text;
  Symmetric, PathFlag: boolean;
  Cost                 : CostMatrx;
  N, NodeCount         : ZeroN;
  Tours                : TourMatrx;
  TourCost, MinTour    : real;



Function ReadNumber (var LL: integer;
                     var B: LineType) : real;
var Int, OK          : integer;
    PointFlag        : boolean;
    R, RD, Decimal : real;
begin                                    { read one number }
  while (B [LL] = ' ') and (LL <= LineLength) do
    LL:= LL+1;                           { skip all spaces }
  Decimal:= 0.1;
  PointFlag:= false;
  R:=  0;
  RD:= 0;
  while (B [LL] <> ' ') and (LL <= LineLength) do
    begin                                { find the number }
      if (B [LL]= '.') then
        PointFlag:=true
      else
```



```
        begin
          val (B [LL], Int, OK);
          if PointFlag = false then
            R:= R*10 + Int
          else
            begin
              RD:= RD + (Int * Decimal);
              Decimal:= Decimal * 0.1;
            end;
        end;
      LL:= LL+1;
    end;                                 { one value found }
  ReadNumber:= R + RD;
end;          { ReadNumber }


Procedure ReadData (var DataFile: text;
                    var CDFlag  : char;
                    var X, Y, Z : CostArray;
                    var ZFlag   : boolean);
var L          : integer;
    NumLines, NL, NPL, I, J: OneN;
    NumPerLine: ZeroN;
    SFlag      : boolean;
    { S: true = symmetric data,   false = non-symmetric data. }
    A          : LineType;
begin
  readln (DataFile, A);
  L:= 4;
  N:= trunc (ReadNumber (L, A));
  ZFlag:= false;
  if A [1] = 'C' then
    begin
      CDFlag:= 'C';
      SFlag := true;
      if A[2] = 'Z' then ZFlag:= true;
      if frac (N/3) > 0 then  NumLines:= trunc (N/3) + 1
      else                    NumLines:= trunc (N/3);
      NumPerLine:= 3;
    end
  else
    begin
      CDFlag  := 'D';
      if A [2] = 'N' then
        begin
          SFlag     := false;
          NumLines  := N;
          NumPerLine:= N-1;
        end
      else
        begin
          SFlag     := true;
          NumLines  := N-1;
          NumPerLine:= N-1;
        end;
    end;
  readln (DataFile, A);                                  { comment line }
  I:= 1;
  J:= 2;
```

```
  for NL:=1 to NumLines do                              { process data lines }
    begin
      for L:=1 to LineLength do  A[L]:= ' ';
      readln (DataFile, A);
      L:= 1;
      for NPL:=1 to NumPerLine do
        if (CDFlag = 'C') and (I <= N) then
          begin
            X [I]:= ReadNumber (L, A);
            Y [I]:= ReadNumber (L, A);
            Z [I]:= 0;
            if ZFlag then Z[I]:= ReadNumber (L, A);
            I    := I+1;
          end
        else
          if CDFlag = 'D' then
            begin
              Cost [I,J]:= ReadNumber (L, A);
              if (L >= LineLength) and (Cost [I,J] = 0) then
                begin
                  readln (DataFile, A);
                  L:= 1;
                  Cost [I,J]:= readNumber (L, A);
                end;
              J         := J+1;
              if J = I then  J:= J+1;
              if J > N then
                begin
                  I:= I+1;
                  if SFlag = true then
                    J:= I+1
                  else
                    J:=1;
                end;
            end;
      if (CDFlag = 'D') and (SFlag = true) then
        NumPerLine:= NumPerLine -1;
    end;
end;          { ReadData }



procedure SetupProblem;
var
    DataFile       : text;
    DataName       : NameType;
    Answer, Metric: char;
    ZFlag          : boolean;
    I, J           : ZeroN;
    X, Y, Z        : CostArray;
    DX, DY, DZ     : real;
begin
  repeat
    write ('Metric to be used (1, 2 or 0) ?');
    readln (Metric);
  until Metric in ['1', '2', '0'];
  write ('Result to be output to ?');
  readln (DataName);
```



```
assign (OutFile, DataName);
writeln;
repeat
  writeln ('Data to be input by:');
  writeln ('                    C = Coordinates of nodes');
  writeln ('                    D = Distance between nodes');
  writeln ('                    F = stored in a File');
  write   ('                   ?');
  readln (Answer);
  Answer:= upcase (Answer);
until Answer in ['C', 'D', 'F'];
Symmetric:= true;
if Answer = 'F' then
  begin
    write    ('Data file name ?');                          { input file }
    readln   (DataName);
    assign   (DataFile, DataName);
    reset    (DataFile);
    ReadData (DataFile, Answer, X, Y, Z, ZFlag);
    close    (DataFile);
  end
else
  begin
    repeat
      write('How many nodes are there ? (Maximum ',maxN,') ');
      readln (N);
    until (N > 3) and (N <= maxN);
    writeln;
    if Answer = 'D' then
      begin
        write('Is this problem symmetric? (Y/N) ');
        readln (Answer);
        if upcase (Answer) = 'N' then  Symmetric:= false;
        writeln('Please input costs / distances between nodes I and J');
        writeln('If there is no cost / distance, input 9999 (i.e. four 9''s');
        for I:=1 to N-1 do
          begin
            if Symmetric then J:=I+1 else J:=1;
            for J:=J to N do
              if I<>J then
                begin
                 write('   cost / distance between nodes ',I,' ',J,' ');
                 readln (Cost [I,J]);
                end;
          end;
      end
    else           { Answer was 'C' }
      begin
        ZFlag:= false;
        write ('z-coordinate to be entered ?');
        readln (Answer);
        if upcase (Answer) = 'Y' then ZFlag:= true;
        Answer:= 'C';
        writeln('Please input x, y (and z) coordinate of each node');
        for I:=1 to N do
          if ZFlag then
            begin
              write('      x  y  z  coordinates of node ',I,' :');
              readln (X [I], Y [I], Z[I]);
            end
```

```
          else
            begin
              write('      x  y  coordinates of node ',I,' :');
              readln (X [I], Y [I]);
              Z [I]:= 0;
            end;
        end;
    end;
  if Answer = 'C' then
   for I:=1 to N-1 do
    for J:=I+1 to N do
     begin
       DX:= abs (X[I] - X[J]);
       DY:= abs (Y[I] - Y[J]);
       DZ:= abs (Z[I] - Z[J]);
       case Metric of
       '1': Cost[I,J]:=      DX    + DY    + DZ;
       '2': Cost[I,J]:=sqrt (DX*DX + DY*DY + DZ*DZ);
       '0': begin
              if (DX >= DY) and (DX >= DZ) then  Cost[I,J]:= DX
              else                               Cost[I,J]:= DY
                                                 Cost[I,J]:= DZ;
            end;
       end;
       Cost [J,I]:= Cost [I,J];
     end;
  MinTour:= 999999.0;
  writeln;
  writeln ('Change floppy now if required.        Press any key to continue.');
  repeat until keypressed;
  rewrite (OutFile);
  writeln (OutFile, DataName);
end;  { SetupProblem }



procedure Timer;
type  RegPack = record
                  AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: integer;
                end;
var   Regs    : RegPack;
      Hr, Mn, Sc, Fr: integer;
begin
  with Regs do
    begin
      AX:= $2C00;
      msdos (Regs);
      Hr:= hi (CX);
      Mn:= lo (CX);
      Sc:= hi (DX);
      Fr:= lo (DX);
    end;
  writeln (OutFile, ' time = ',Hr,':',Mn,':',Sc,'.',Fr:2);
end;     { Timer }



procedure FindSpanningPath (K: ZeroN;
                            var OKflag: boolean);
var
```



```
  LowestCost: real;
  I, J, LowestI, LowestJ, CurrentNode: ZeroN;
begin                  { find the cheapest path starting at node K }
  CurrentNode:= K;
  for I:=1 to N do        { initially no nodes are connected }
    Tours [0,I]:= 0;
  TourCost:= 0;
  OKflag:= true;
  J:= 1;
  while (OKflag = true) and (J < N) do
  { while there are valid edges, and nodes, left,
  join the cheapest edge from the current node to the path,
  leaving the first and last nodes on the path unconnected. }
  begin
    LowestCost:=9999;
    for I:=1 to (CurrentNode - 1) do
      if (Tours[0,I] = 0) then                              { no cycle formed }
        begin
          if Symmetric then
           begin
            if Cost [I,CurrentNode] < LowestCost then     { valid edge found }
              begin
                LowestCost:= Cost [I,CurrentNode];
                LowestI:= I;
              end;
           end
          else
            if Cost [CurrentNode,I] < LowestCost then     { valid edge found }
              begin
                LowestCost:= Cost [CurrentNode,I];
                LowestI:= I;
              end;
        end;
    for I:=(CurrentNode + 1) to N do
      if (Tours[0,I] = 0) and                             { no cycle formed }
         (Cost [CurrentNode,I] < LowestCost) then         { valid edge found }
        begin
          LowestCost:= Cost [CurrentNode,I];
          LowestI:= I;
        end;
    if LowestCost < 9999 then           { include edge in spanning path }
      begin
        Tours [0,CurrentNode]:= LowestI;
        TourCost:= TourCost + LowestCost;
        CurrentNode:= LowestI;
      end
    else                                              {no more valid edges are left}
      OKflag:= false;
    J:= J+1;
  end;
end;     { FindSpanningPath }


procedure FindTour (K: ZeroN;
                    OKflag: boolean);
var
  I, OpenNodes, LastNode: ZeroN;
begin
  if OKflag = true then
  { find the final edge which will connect the two remaining nodes
```

```
 that are the start and end nodes of the path                    }
   begin
     OpenNodes:= 0;
     LastNode := 0;
     for I:=1 to N do                    { search for node with no 'next node' }
       if Tours [0,I] = 0 then
         begin
           OpenNodes:= OpenNodes +1;
           if Symmetric and (I < K) then
            begin
             if (Cost [I,K] < 9999) then      { valid edge found }
               LastNode:= I;
            end
           else
             if (Cost [K,I] < 9999) then      { valid edge found }
               LastNode:= I;
         end;
     if OpenNodes = 1 then
       begin
         Tours [0,LastNode] := K;                    { close the tour }
         if Symmetric and (LastNode < K) then
           TourCost:= TourCost + Cost [LastNode, K]
         else
           TourCost:= TourCost + Cost [K, LastNode];
         if TourCost < MinTour then        { this is the cheapest tour so far }
           begin
             MinTour:= TourCost;
             for I:=1 to N do  Tours [1, I]:= Tours [0, I];
           end
       end
     else
       OKFlag:= false;
   end;
end;    { FindTour }


procedure WriteBestTour;
var  K, Out4: ZeroN;
begin
  if MinTour < 999999.0 then                        { write one tour }
    begin
      Out4:= 0;
      K:= 1;
      repeat
        write(OutFile,'     ',K:4,'   ',Tours [1,K]:4);
        Out4:= Out4 +1;
        if Out4 = 4 then
          begin
            Out4:= 0;
            writeln (OutFile);
          end;
        K:= Tours [1,K];
      until K = 1;
      writeln(OutFile);
      writeln(OutFile, 'Cost of tour is    ', MinTour:11:4);
    end
  else
    begin
      writeln(OutFile, 'No tours could be found');
    end;
```



```
  close (OutFile);
  writeln ('                *** end of program ***');
end;    { WriteBestTour }


begin
  SetupProblem;
  Timer;
  for NodeCount:=1 to N do
    begin
      FindSpanningPath (NodeCount,PathFlag);
      FindTour (NodeCount,PathFlag);
    end;
  Timer;
  WriteBestTour;
end.
```

```
program MSTSubtour;
{$R-}
{
    method:      find minimum spanning tree of nodes not in subtour.
                 Add a portion of the minimum spanning tree to the
                 subtour to form a larger subtour.
                 Continue until no nodes remain.
    conditions:  non-Euclidean
                 symmetric
}

const
      maxN        = 101;
      maxNminus1  = 100;
      LineLength  = 250;

type
      NameType    = string [12];
      LineType    = string [LineLength];
      ZeroN       = 0..maxN;
      OneNm1      = 0..maxNminus1;
      OneN        = 1..maxN;
      TwoN        = 2..maxN;
      CoordArray= array [OneN]          of real;
      CostMatrx = array [OneN, OneN]    of real;
      TourMatrx = array [OneN, 0..1]    of ZeroN;
      TreeMatrx = array [OneN, OneNm1] of ZeroN;

var
      OutFile     : text;
      N, Start, NodesRemain: ZeroN;
      Cost        : CostMatrx;
      V           : TourMatrx;
      NodeOnTour: array [OneN] of boolean;



Function ReadNumber (var LL: integer;
                     var B: LineType) : real;
var Int, OK : integer;
    PointFlag : boolean;
    R, RD, Decimal : real;
begin                                   { read one number }
  while (B [LL] = ' ') and (LL <= LineLength) do
    LL:= LL+1;                          { skip all spaces }
  Decimal:= 0.1;
  PointFlag:= false;
  R:=  0;
  RD:= 0;
  while (B [LL] <> ' ') and (LL <= LineLength) do
    begin                               { find the number }
      if (B [LL]= '.') then
        PointFlag:=true
      else
        begin
          val (B [LL], Int, OK);
          if PointFlag = false then
            R:= R*10 + Int
          else
            begin
```



```
              RD:= RD + (Int * Decimal);
              Decimal:= Decimal * 0.1;
            end;
        end;
      LL:= LL+1;
    end;                                { one value found }
  ReadNumber:= R + RD;
end;          { ReadNumber }


Procedure ReadData (var DataFile: text;
                    var CDFlag  : char;
                    var X, Y, Z : CoordArray;
                    var ZFlag   : boolean);
var L: integer;
    NumLines, NL, NPL, I, J: OneN;
    NumPerLine: ZeroN;
    A: LineType;
begin
  readln (DataFile, A);
  L:= 4;
  N:= trunc (ReadNumber (L, A));
  ZFlag:= false;
  if A [1] = 'C' then
    begin
      CDFlag:= 'C';
      if A [2] = 'Z' then  ZFlag:= true;
      if frac (N/3) > 0 then NumLines:= trunc (N/3) + 1
      else                   NumLines:= trunc (N/3);
      NumPerLine:= 3;
    end
  else
    begin
      CDFlag:= 'D';
      NumLines:= N-1;
      NumPerLine:= N-1;
    end;
  readln (DataFile, A);                                 { comment line }
  I:= 1;
  J:= 2;
  for NL:=1 to NumLines do                              { process data lines }
    begin
      for L:=1 to LineLength do  A[L]:= ' ';
      readln (DataFile, A);
      L:= 1;
      for NPL:=1 to NumPerLine do
        if (CDFlag = 'C') and (I <= N) then
          begin
            X [I]:= ReadNumber (L, A);
            Y [I]:= ReadNumber (L, A);
            Z [I]:= 0;
            if ZFlag then  Z [I]:= ReadNumber (L, A);
            I:= I+1;
          end
        else
          begin
            Cost [I,J]:= ReadNumber (L, A);
            if (L >= LineLength) and (Cost [I,J] = 0) then
              begin
```

```
              readln (dataFile, A);
              L:= 1;
              Cost [I,J]:= ReadNumber (L, A);
            end;
          Cost [J,I]:= Cost [I,J];
          J:= J+1;
          if J = I then  J:= J+1;
          if J > N then
            begin
              I:= I+1;
              J:= I+1
            end;
        end;
      if CDFlag = 'D' then
        NumPerLine:= NumPerLine -1;
    end;
end;          { ReadData }



procedure SetupProblem;
var
    DataFile      : text;
    DataName      : NameType;
    I, MinI, NodeI: OneN;
    J, MinJ       : TwoN;
    MinCost, DX, DY, DZ: real;
    ZFlag         : boolean;
    X, Y, Z       : array [OneN] of real;
    Answer, Metric: char;
begin
  repeat
    write ('Metric to be used (1, 2 or 0) ?');
    readln (Metric);
  until Metric in ['1', '2', '0'];
  write ('Result to be output to ?');
  readln (DataName);
  assign (OutFile, DataName);
  writeln;
  repeat
    writeln ('Data to be input by:');
    writeln ('                    C = Coordinates of nodes');
    writeln ('                    D = Distance between nodes');
    writeln ('                    F = stored in a File');
    write   ('                   ?');
    readln (Answer);
    Answer:= upcase (Answer);
  until Answer in ['C', 'D', 'F'];
  MinCost:= 9999;
  if Answer = 'F' then
    begin
      write    ('Data file name ?');                    { input file }
      readln   (DataName);
      assign   (DataFile, DataName);
      reset    (DataFile);
      ReadData (DataFile, Answer, X, Y, Z, ZFlag);
      close    (DataFile);
    end
```



```
else
  begin
    repeat
      write('How many nodes are there ? (Maximum ',maxN,') ');
      readln (N);
    until (N > 3) and (N <= maxN);
    writeln;
    if Answer = 'D' then
      begin
        writeln('Please input costs / distances between nodes I and J');
        for I:=1 to N-1 do
          for J:=I+1 to N do
            begin
              write('   cost / distance between nodes ',I,' ',J,' ');
              readln (Cost [I,J]);
              Cost [J,I]:= Cost [I,J];
            end;
      end
    else
      begin
        ZFlag:= false;
        write ('z-coordinate to be entered ?');
        readln (Answer);
        if upcase (Answer) = 'Y' then  ZFlag:= true;
        Answer:= 'C';
        writeln('Please input x-, y- and z-coordinates of each node');
        for NodeI:=1 to N do
          if ZFlag then
            begin
              write('      x  y  z  coordinates of node ',NodeI,' :');
              readln (X [NodeI], Y [NodeI], Z [NodeI]);
            end
          else
            begin
              write('      x  y  coordinates of node ',NodeI,' :');
              readln (X [NodeI], Y [NodeI]);
              Z [NodeI]:= 0;
            end;
      end;
  end;
if Answer = 'C' then
  for I:=1 to N-1 do
    for J:=I+1 to N do
      begin
        DX:= abs (X[I] - X[J]);
        DY:= abs (Y[I] - Y[J]);
        DZ:= abs (Z[I] - Z[J]);
        case Metric of
        '1': Cost[I,J]:=      DX    + DY    + DZ;
        '2': Cost[I,J]:=sqrt (DX*DX + DY*DY + DZ*DZ);
        '0': begin
               if (DX >= DY) and (DX >= DZ) then  Cost[I,J]:= DX
               else                               Cost[I,J]:= DY
                                                  Cost[I,J]:= DZ;
             end;
        end;
        Cost [J, I]:= Cost [I, J];
      end;
Start:= 1;                                    { set first 'subtour' }
NodeOnTour [1]:= true;
```

```
  V [1, 0]:= 1;
  V [1, 1]:= 1;
  for I:=2 to N do
    begin
      NodeOnTour [I]:= false;
      V [I, 0]:= 0;
      V [I, 1]:= 0;
    end;
  writeln;
  writeln ('Change floppy now if required.        Press any key to continue.');
  repeat until keypressed;
  rewrite (OutFile);
  writeln (OutFile, DataName);
end;  { SetupProblem }



procedure Timer;
type RegPack = record
                 AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : integer;
               end;
var  Regs     : RegPack;
     Hr, Mn, Sc, Fr: integer;
begin
  with Regs do
    begin
      AX:= $2C00;
      msdos (Regs);
      Hr:= hi (CX);
      Mn:= lo (CX);
      Sc:= hi (DX);
      Fr:= lo (DX);
    end;
  writeln (OutFile, ' time = ',Hr,':',Mn,':',Sc,'.',Fr:2);
end;        { Timer }


procedure CheckForSubtour (var T           : TreeMatrx;
                           From, Node      : ZeroN;
                           var Compare     : ZeroN;
                           var NoCycleFlag: boolean);
var Next, L: ZeroN;
begin                      { check all branches of Node against Compare node }
  L:=1;
  while (T [Node, L] > 0) and (NoCycleFlag = true) and (L < N) do
    begin
      Next:= T [Node, L];
      if (Next = From) and (T [Node, L+1] <> 0) then
        begin
          L:= L+1;
          Next:= T [Node, L];
        end;
      if Next <> From then
        begin
          if Next = Compare then        { cycle would be formed }
            NoCycleFlag:= false
          else                          { check branch of Node }
            CheckForSubtour (T, Node, Next, Compare, NoCycleFlag);
        end;
      L:= L+1;
```



```
    end;
end;      { CheckForSubtour }



procedure FindSpanningTree (var OpenNodes: ZeroN;
                            var T: TreeMatrx);
var
  NoSubtour  : boolean;
  LowestCost : real;
  I, J, LowestI, LowestJ, EdgesOnTree: ZeroN;
  Edge       : array [OneNm1, TwoN] of boolean;
begin
  EdgesOnTree:=0;                   { initially there are no edges }
  for I:=1 to N-1 do
    begin
      T [1, I]:= 0;
      for J:=2 to N do
        begin
          T [J, I]:= 0;
          T [J, I]:= 0;
          Edge [I, J]:= false;
        end;
    end;
  while (EdgesOnTree < OpenNodes-1) do
  { while there are valid edges left, connect up all nodes
  into a minimum spanning tree.                              }
   begin
    LowestCost:=9999;
    for I:=1 to N-1 do
      for J:=I+1 to N do
       begin
        if (Edge [I, J] = false) and                  { 'open' edge remains }
           (NodeOnTour [I] = false) and (NodeOnTour [J] = false) and
           (Cost[I,J] < LowestCost) then                        { check cost }
          begin
            NoSubtour:= true;
            if (T [I, 1] > 0) and  (T [J, 1] > 0) then
               CheckForSubtour (T, 0, I, J, NoSubtour);
            if NoSubtour then
              begin
                LowestCost:= Cost[I,J];
                LowestI:= I;
                LowestJ:= J;
              end;
          end;
       end;
    if LowestCost < 9999 then                {include edge in spanning path }
      begin
        I:= 1;
        while (I < N) and (T [LowestI, I] > 0) do  I:= I+1;
        T [LowestI, I]:= LowestJ;
        J:= 1;
        while (J < N) and (T [LowestJ, J] > 0) do  J:= J+1;
        T [LowestJ, J]:= LowestI;
        Edge [LowestI, LowestJ]:= true;
        EdgesOnTree:= EdgesOnTree + 1;
      end
    else                                        {no more valid edges are left}
      writeln('Run out of valid edges before tour could be completed.');
```

```
  end;
 OpenNodes:= 0;
end;    { FindSpanningTree }



procedure CompareMin (var TStart, TEnd, NumNodes,
                          MinS1, MinS2, MinT1, MinT2: ZeroN;
                          { S for subtour, T for m.s. tree }
                      var Distance, MinCost: real);
var  SNode, SNext: ZeroN;
     CompareCost: real;
begin                              { check cost of using this branch end node }
  SNext:= Start;
  repeat
    if V [SNext, 0] = SNode then
      begin
        SNode:= SNext;
        SNext:= V [SNext, 1];
      end
    else
      begin
        SNode:= SNext;
        SNext:= V [SNext, 0];
      end;
    CompareCost:= (Cost [SNode, TStart] + Cost [SNext, TEnd]
                 - Cost [SNode, SNext]  + Distance)
                 / (NumNodes * NumNodes);
    if CompareCost < MinCost then
      begin
        MinCost:= CompareCost;
        MinS1:= SNode;
        MinT1:= TStart;
        MinS2:= SNext;
        MinT2:= TEnd;
      end;
    CompareCost:= (Cost [SNode, TEnd] + Cost [SNext, TStart]
                 - Cost [SNode, SNext]  + Distance)
                 / (NumNodes * NumNodes);
    if CompareCost < MinCost then
      begin
        MinCost:= CompareCost;
        MinS1:= SNode;
        MinT1:= TEnd;
        MinS2:= SNext;
        MinT2:= TStart;
      end;
  until SNext = Start;
end;     { CompareMin }



procedure AddNext (From, Node, NumNodes      : ZeroN;
                   Dist                      : real;
                   var TStart, S1, S2, T1, T2: ZeroN;
                   { S for subtour, T for m.s. tree }
                   var MinCost               : real;
                   var T                     : TreeMatrx);
var  L, Next: ZeroN;
begin                { search for end nodes, adding costs and number of nodes }
```



```
 L:= 1;
 NumNodes:= NumNodes + 1;
 if From > 0 then
   Dist:= Dist + Cost [From, Node];
 while (T [Node, L] > 0) and (L < N) do
   begin
     Next:= T [Node, L];
     if (Next = From) and (T [Node, L+1] <> 0) then
       begin                                      { not an end node }
         L:= L+1;
         Next:= T [Node, L];
       end;
     if (Next = From) and (L = 1) then            { an end node found }
       CompareMin (TStart, Node, NumNodes, S1, S2, T1, T2, Dist, MinCost)
     else
       if Next <> From then                       { continue looking }
        AddNext(Node, Next, NumNodes, Dist, TStart, S1,S2, T1,T2, MinCost, T);
     L:= L+1;
   end;
end;      { AddNext }



procedure SetMinPath (From, Node: ZeroN;
                      var TEnd  : ZeroN;
                      var Path  : boolean;
                      var T     : Treematrx);
var  L, Next: ZeroN;
begin           { search for min end node and set all nodes along path to it }
  L:= 1;
  while (T [Node, L] > 0) and (L < N) and (Path = false) do
    begin
      Next:= T [Node, L];
      if (Next = From) and (T [Node, L+1] <> 0) then
        begin                                     { not an end node }
          L:= L+1;
          Next:= T [Node, L];
        end;
      if (Next = From) and (L = 1) then           { an end node found }
        begin
          if Node = TEnd then
            Path:= true;
        end
      else
        if Next <> From then
          SetMinPath (Node, Next, TEnd, Path, T);
      L:= L+1;
    end;
  if Path then                                    { set path through this node }
    begin
      if From > 0 then    V [Node, 0]:= From;
      V [Node, 1]:= Next;
      NodeOnTour [Node]:= true;
    end;
end;      { SetMinPath }



procedure FindSubtour (var OpenNodes: ZeroN;
                       var T        : TreeMatrx);
```

```
var
    S1, S2, T1, T2, Closest: ZeroN;
  { S for subtour, T for m.s. tree }
    MinCost, D: real;
    PathFlag  : boolean;
begin                          { find branch of MST which adds the least cost }
  MinCost:= 9999;                    { for the number of nodes in the branch }
  MinCost:= MinCost * 1000;
  S1:= 0;
  for T1:=1 to N do                { find MST branch end closest to the subtour }
    if (NodeOnTour [T1] = false) and
       (T [T1, 2] = 0) then          { MST branch end }
      begin
        S2:= Start;
        repeat
          if V [S2, 0] = S1 then
            begin
              S1:= S2;
              S2:= V [S2, 1];
            end
          else
            begin
              S1:= S2;
              S2:= V [S2, 0];
            end;
          if Cost [T1, S1] < MinCost then
            begin
              MinCost:= Cost [T1, S1];
              Closest:=T1;
            end;
        until S2 = Start;
      end;
  MinCost:= 9999;                  { find MST branch end giving min new subtour }
  MinCost:= MinCost * 1000;
  AddNext (0, Closest, 0, 0, Closest, S1, S2, T1, T2, MinCost, T);
  PathFlag:= false;
  if T1 = Closest then  SetMinPath (0, Closest, T2, PathFlag, T)
  else                  SetMinPath (0, Closest, T1, PathFlag, T);
  if V [S1, 0] = S2 then  V [S1, 0]:= T1              { close subtour }
  else                    V [S1, 1]:= T1;
  if V [S2, 0] = S1 then  V [S2, 0]:= T2
  else                    V [S2, 1]:= T2;
  if T1 = Closest then
    begin
      V [T1, 0]:= S1;
      V [T2, 1]:= S2;
    end
  else
    begin
      V [T1, 1]:= S1;
      V [T2, 0]:= S2;
    end;
  for T1:=1 to N do                        { reset nodes not on subtour }
    if NodeOnTour [T1] = false then
      begin
        OpenNodes:= OpenNodes + 1;
        Closest:= T1;
      end;
  if NodesRemain = 1 then
    begin                      { include last remaining node in cheapest place }
```



```
      T1:= 1;
      D := 0;
      MinCost:= 9999;
      CompareMin (Closest, Closest, T1, S1, S2, T2, T2, D, MinCost);
      if V [S1, 0] = S2 then  V [S1, 0]:= Closest        { close subtour }
      else                    V [S1, 1]:= Closest;
      if V [S2, 0] = S1 then  V [S2, 0]:= Closest
      else                    V [S2, 1]:= Closest;
      V [Closest, 0]:= S1;
      V [Closest, 1]:= S2;
      NodesRemain:= 0;
    end;
end;     { FindSubtour }


procedure ExpandSubtour (var NodesLeft: ZeroN);
var T: TreeMatrx;
begin
  FindSpanningTree (NodesLeft, T);
  FindSubtour      (NodesLeft, T);
end;



procedure WriteTour;
var  Node, Next, Out4: ZeroN;
     TourCost        : real;
begin
  TourCost:= 0;
  Out4:= 0;
  Next:= Start;
  repeat
    if V [Next, 0] = Node then
      begin
        Node:= Next;
        Next:= V [Next, 1];
      end
    else
      begin
        Node:= Next;
        Next:= V [Next, 0];
      end;
    write (OutFile, '  ', Node:4, ' ', Next:4);
    Out4:= Out4 + 1;
    if Out4 = 4 then
      begin
        Out4:= 0;
        writeln (OutFile);
      end;
    TourCost:= TourCost + Cost [Node, Next];
  until Next = Start;
  writeln (OutFile);
  writeln (OutFile, 'Cost of tour is ', TourCost:11:4);
  close (OutFile);
  writeln ('                    *** end of program ***');
end;    { WriteTour }


begin
```

```
  SetupProblem;
  Timer;
  NodesRemain:= N-1;
  while (NodesRemain > 0) do
          ExpandSubtour  (NodesRemain);
  Timer;
  WriteTour;
nd.
```



```
program Ellipse;
{$R-}
{
    method:      - find the convex hull of the nodes.
                 - progressively include the node forming the most
                   eccentric ellipse, into the current subtour.
    conditions: - Euclidean
    input:       - how many nodes
                 - for each node input its x- and y-coordinates.
}

const
      maxN       = 101;
      LineLength = 250;

type
      NameType   = string [12];
      LineType   = string [LineLength];
      ZeroN      = 0..maxN;
      OneN       = 1..maxN;
      CoordArray = array [OneN] of real;
      TourArray  = array [0..1, OneN] of ZeroN;


var
  OutFile     : text;
  Metric      : char;
  Hour        : real;
  N, StartNode, Counter: ZeroN;
  NodesRemain, CFlag   : boolean;
  X, Y, Z     : CoordArray;
  V           : TourArray;



Function ReadNumber (var LL: integer;
                     var B: LineType) : real;
var Int, OK : integer;
    PointFlag : boolean;
    R, RD, Decimal : real;
begin                                        { read one number }
  while (B [LL] = ' ') and (LL <= LineLength) do
    LL:= LL+1;                               { skip all spaces }
  Decimal:= 0.1;
  PointFlag:= false;
  R:=  0;
  RD:= 0;
  while (B [LL] <> ' ') and (LL <= LineLength) do
    begin                                    { find the number }
      if (B [LL]= '.') then
        PointFlag:=true
      else
        begin
          val (B [LL], Int, OK);
          if PointFlag = false then
            R:= R*10 + Int
          else
            begin
              RD:= RD + (Int * Decimal);
```

```
            Decimal:= Decimal * 0.1;
          end;
      end;
    LL:= LL+1;
  end;                                  { one value found }
ReadNumber:= R + RD;
end;          { ReadNumber }


Procedure ReadData (var DataFile: text;
                    var MinX    : real;
                    var ZFlag   : boolean);
var L: integer;
    NumLines, NL, NumPerLine, NPL, I: OneN;
    A: LineType;
begin
  readln (DataFile, A);
  if A[1] <> 'C' then
    begin
      writeln; writeln;
      writeln ('******  Program requires coordinates of nodes !      ******');
      writeln ('******  The given data is not stored as coordinates.  ******');
      CFlag:= false;
    end
  else
    begin
      L:= 4;
      N:= trunc (ReadNumber (L, A));
      ZFlag:= false;
      if A[2] = 'Z' then ZFlag:= true;
      if frac (N/3) > 0 then  NumLines:= trunc (N/3) + 1
      else                    NumLines:= trunc (N/3);
      NumPerLine:= 3;
      readln (DataFile, A);                              { comment line }
      I:= 1;
      for NL:=1 to NumLines do                           { process data lines }
        begin
          for L:=1 to LineLength do  A[L]:= ' ';
          readln (DataFile, A);
          L:= 1;
          for NPL:=1 to NumPerLine do
            if I <= N then
              begin
                X [I]:= ReadNumber (L, A);
                Y [I]:= ReadNumber (L, A);
                Z [I]:= 0;
                if ZFlag then   Z [I]:= ReadNumber (L, A);
                if X [I] < MinX then
                  begin
                    MinX:= X [I];
                    StartNode:= I;          { set the first node on the hull }
                  end;
                I:= I+1;
              end;
        end;
    end;
end;          { ReadData }
```



```
procedure SetupProblem;
var
    DataFile: text;
    DataName: NameType;
    I       : OneN;
    MinX    : real;
    ZFlag   : boolean;
    Answer  : char;
begin
  repeat
    write ('Metric to be used (1, 2 or 0) ?');
    readln (Metric);
  until Metric in ['1', '2', '0'];
  write ('Result to be output to ?');
  readln (DataName);
  assign (OutFile, DataName);
  writeln;
  repeat
    writeln ('Data to be input by:');
    writeln ('                    C = Coordinates of nodes');
    writeln ('                    F = stored in a File');
    write   ('                  ?');
    readln (Answer);
    Answer:= upcase (Answer);
  until Answer in ['C', 'F'];
  CFlag:= true;
  if Answer = 'F' then
    begin
      write    ('Data file name ?');                    { input file }
      readln   (DataName);
      assign   (DataFile, DataName);
      reset    (DataFile);
      ReadData (DataFile, MinX, ZFlag);
      close    (DataFile);
    end
  else
    begin
      repeat
        write('How many nodes are there ? (Maximum ',maxN,') ');
        readln (N);
      until (N > 3) and (N <= maxN);
      writeln;
      ZFlag:= false;
      write ('z-coordinate to be entered ?');
      readln (Answer);
      if upcase (Answer) = 'Y' then  ZFlag:= true;
      writeln('Please input x, y (and z) coordinates of each node');
      for I:=1 to N do
        begin
          if ZFlag then
            begin
              write('   x  y  z  coordinates of node ',I,':');
              readln (X [I], Y [I], Z [I]);
            end
          else
            begin
              write('   x  y coordinates of node ',I,':');
              readln (X [I], Y [I]);
              Z [I]:= 0;
```

```
          end;
        end;
    end;
  if CFlag then
    begin
      MinX:= 99999.0;
      for I:=1 to N do
        begin
          if X [I] < MinX then
            begin
              MinX:= X [I];
              StartNode:= I;
            end;
          V [0,I]:= 0;
          V [1,I]:= 0;
        end;
      writeln;
      writeln ('Change floppy now if required.        Press any key to continue.
');
      repeat until keypressed;
      rewrite (OutFile);
      writeln (OutFile, DataName);
      NodesRemain:= true;
    end;
end;  { SetupProblem }


procedure Timer;
type RegPack = record
                 AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : integer;
               end;
var  Regs     : RegPack;
     TimeTest: real;
     Hr, Mn, Sc, Fr: integer;
begin
  with Regs do
    begin
      AX:= $2C00;
      msdos (Regs);
      Hr:= hi (CX);
      Mn:= lo (CX);
      Sc:= hi (DX);
      Fr:= lo (DX);
    end;
  if Counter = 0 then
    begin
      writeln (OutFile, ' time = ',Hr,':',Mn,':',Sc,'.',Fr:2);
      Hour:= Hr + (Mn /60);
    end
  else
    begin
      TimeTest:= Hr + (Mn /60) - Hour;
      WRITELN('TIME ELAPSED ',TIMETEST:6:2);
      if TimeTest > 2 then            { stop run after 2 hours }
        begin
          NodesRemain:= false;
          CFlag:= false;
        end
    end;
end;        { Timer }
```



```
Function Dist (var A, B: OneN)  : real;
var  DX, DY, DZ: real;
begin
  DX:= abs (X[B]-X[A]);
  DY:= abs (Y[B]-Y[A]);
  DZ:= abs (Z[B]-Z[A]);
  case Metric of
  '1': Dist:=       DX    + DY    + DZ;
  '2': Dist:= sqrt (DX*DX + DY*DY + DZ*DZ);
  '0': begin
         if (DX >= DY) and (DX >= DZ) then  Dist:= DX
         else           if (DY >= DZ) then  Dist:= DY
                        else                Dist:= DZ;
       end;
  end;
end;    { Dist }


Function Line (var Point1, Point2: OneN;
               var XVal: real): real;
var  A: real;                                  { compute using straight line }
begin
  if X [Point1] = X [Point2] then
    begin
      XVal:= X [Point1];
      if Y [Point1] < Y [Point2] then   Line:= Y [Point2] + 10
      else                              Line:= Y [Point2] - 10;
    end
  else
    begin
      A:= (Y [Point1] - Y [Point2]) / (X [Point1] - X [Point2]);
      Line:= A*XVal  +  Y[Point1] - (A*X[Point1]);            { Y = A*X + B }
    end;
end;    { Line }


Procedure ConvexHull (var Continue: boolean);
var
     I, NumNodes, OldNode1, Node1, LineNode1, NewNode1,
                  OldNode2, Node2, LineNode2, NewNode2: OneN;
     X1, Y1, X2, Y2, XTest, YTest, XLine1, YLine1, XLine2, YLine2,
                  MinCos, MaxCos1, MaxCos2, CosAngle: real;
     Search1, Search2: boolean;
                                  { Find the convex hull of a set of points.
                                    Input: x- and y-coordinates of the points.
                                    Method: For each remaining node not on the
                                            hull, find the angle between it and
                                            an edge on the hull. The largest
                                            angle formed indicates the next
                                            node on the hull.   }

begin
  OldNode1:= StartNode;
  MinCos :=  2;                          { find first two edges of the hull }
  MaxCos1:= -2;
  for I:=1 to N do
    if I <> OldNode1 then
```

```
    begin
      X1:= X [I] - X [OldNode1];
      Y1:= Y [I] - Y [OldNode1];
      CosAngle:= (- Y1 * Y [OldNode1]) /
                 (sqrt (X1*X1 + Y1*Y1) * Y [OldNode1]);
      if CosAngle < MinCos then
        begin
          Node2:= I;
          MinCos:= CosAngle;
        end
      else
        if CosAngle = MinCos then
          begin
            if dist (OldNode1, I) < dist (OldNode1, Node2) then
              Node2:= I;
          end;
      if CosAngle > MaxCos1 then
        begin
          Node1:= I;
          MaxCos1:= CosAngle;
        end
      else
        if CosAngle = MaxCos1 then
          if dist (OldNode1, I) < dist (OldNode1, Node1) then
            Node1:= I;
    end;
V [0, Node2]    := OldNode1;
V [0, OldNode1]:= Node2;
V [1, OldNode1]:= Node1;
V [0, Node1]    := OldNode1;
NumNodes:= 3;
OldNode2:= OldNode1;
Search1:= true;
Search2:= true;
while search1 or search2 do              { check all remaining angles }
  begin
    XLine1:= X [Node1] + 5;
    YLine1:= Line (OldNode1, Node1, XLine1);
    MaxCos1:= 2;
    XLine2:= X [Node2] + 5;
    YLine2:= Line (OldNode2, Node2, XLine2);
    MaxCos2:= 2;
    X1:= X [Node1] - XLine1;        { = 5 }
    Y1:= Y [Node1] - YLine1;
    X2:= X [Node2] - XLine2;        { = 5 }
    Y2:= Y [Node2] - YLine2;
    for I:=1 to N do
      if V [1,I] = 0 then
       begin
        if (I <> Node2) and (Search2) then
          begin                  { check for next node to be added (path 2) }
            XTest:= X [I] - X [Node2];
            YTest:= Y [I] - Y [Node2];
            CosAngle:= ((XTest * X2)  +  (YTest * Y2)) /
                       (sqrt (XTest*XTest + YTest*YTest) *
                        sqrt (X2*X2 + Y2*Y2));
            if CosAngle < MaxCos2 then
              begin
                NewNode2:= I;
                MaxCos2:= CosAngle;
```



```
              end
            else
              if CosAngle = MaxCos2 then
                if dist (Node2, I) < dist (Node2, NewNode2) then
                  NewNode2:= I;
          end;
        if (I <> Node1) and (Search1) then
          begin                  { check for next node to be added (path 1) }
            XTest:= X [I] - X [Node1];
            YTest:= Y [I] - Y [Node1];
            CosAngle:= ((X1 * XTest)  +  (Y1 * YTest)) /
                       (sqrt (X1*X1 + Y1*Y1) *
                        sqrt (XTest*XTest + YTest*YTest));
            if CosAngle < MaxCos1 then
              begin
                NewNode1:= I;
                MaxCos1:= CosAngle;
              end
            else
              if CosAngle = MaxCos1 then
                if dist (Node1, I) < dist (Node1, NewNode1) then
                  NewNode1:= I;
          end;
       end;
      if Search1 then
        begin
          NumNodes:= NumNodes + 1;
          V [1, Node1]:= NewNode1;
          if V [0, NewNode1] = 0 then  V [0, NewNode1]:= Node1
          else                         V [1, NewNode1]:= Node1;
          OldNode1:= Node1;
          Node1:= NewNode1;
          if X [Node1] < X [OldNode1] then Search1:= false;
          if NewNode1 = Node2 then         Search1:= false;
        end;
      if Search2 then
        begin
          NumNodes:= NumNodes + 1;
          V [1, Node2]:= NewNode2;
          if V [0, NewNode2] = 0 then  V [0, NewNode2]:= Node2
          else                         V [1, NewNode2]:= Node2;
          OldNode2:= Node2;
          Node2:= NewNode2;
          if X [Node2] < X [OldNode2] then Search2:= false;
          if NewNode2 = OldNode1 then      Search2:= false;
          if NewNode1 = NewNode2 then
            begin
              Search1:= false;
              Search2:= false;
            end;
        end;
    end;
  Continue:= false;                  { set for any nodes left inside the hull }
  if NumNodes < N then Continue:= true;
end;   { Convex Hull }


Procedure InsertNode (var Continue: boolean);
var NodeCount: ZeroN;
```

```
    I, HullNode1, HullNode2, Node1, Node2, NewNode: OneN;
    Eccentric, Ratio, Dist1, Dist2: real;
begin                                { insert one node into the current subtour }
  NodeCount:= 0;
  Eccentric:= -9999;                        { find the most eccentric ellipse }
  for I:=1 to N do
    if V [0, I] = 0 then                    { for each node not on subtour }
      begin
        NodeCount:= NodeCount +1;
        HullNode1:= StartNode;
        HullNode2:= V [1, StartNode];
        Dist1:= Dist (I, HullNode1);
        repeat                              { for each node on 'hull' (subtour) }
            Dist2:= Dist (I, HullNode2);
            Ratio:= Dist (HullNode1, HullNode2) / (Dist1 + Dist2);
            if Ratio > Eccentric then
              begin
                Eccentric:= Ratio;
                NewNode:= I;
                Node1:= HullNode1;
                Node2:= HullNode2;
              end;
            if V [0, HullNode2] = HullNode1 then
              begin
                HullNode1:= HullNode2;
                HullNode2:= V [1, HullNode2];
              end
            else
              begin
                HullNode1:= HullNode2;
                HullNode2:= V [0, HullNode2];
              end;
            Dist1:= Dist2;
        until HullNode1 = StartNode;
      end;
  V [0, NewNode]:= Node1;                   { insert node into subtour }
  V [1, NewNode]:= Node2;
  if V [0, Node1] = Node2 then
     V [0, Node1]:= NewNode
  else
     V [1, Node1]:= NewNode;
  if V [0, Node2] = Node1 then
     V [0, Node2]:= NewNode
  else
     V [1, Node2]:= NewNode;
  if NodeCount = 1 then                     { the last node was inserted }
    Continue:= false;
end;     { InsertNode }



procedure WriteBestTour;
var
  Node, PrevNode: OneN;
  Out4     : ZeroN;
  TourCost: real;
begin
  if Cflag then
    begin
      TourCost:= 0;
```



```
      Out4:= 0;
      PrevNode:= 1;
      Node:= V [1, 1];
      repeat
          write (OutFile, '   ', PrevNode:4, Node:4);
          Out4:= Out4 +1;
          if Out4 = 4 then
            begin
              Out4:= 0;
              writeln (OutFile);
            end;
          TourCost:= TourCost + Dist (PrevNode, Node);
          if V [0, Node] = PrevNode then
            begin
              PrevNode:= Node;
              Node:= V [1, Node];
            end
          else
            begin
              PrevNode:= Node;
              Node:= V [0, Node];
            end;
      until PrevNode = 1;
      writeln (OutFile);
      writeln (OutFile, ' Cost of tour is ', TourCost:11:4);
    end
  else
    begin
      writeln (OutFile, 'Best tour is:');
      for Node:=1 to N do  begin
        write (OutFile, 'V [',Node:2,'] = ');
        if V[0,Node]>0 then write (OutFile, V[0,Node]:2,'   ')
        else  write('     ');
        if V[1,Node]>0 then write (OutFile, V[1,Node]:2,'   '); writeln; end;
    end;
  close (OutFile);
  writeln ('                    *** end of program ***');
end;     { WriteBestTour }



begin
  SetupProblem;
  if CFlag then
    begin
       Counter:= 0;  Timer;
       ConvexHull (NodesRemain);
       while NodesRemain do
           begin
               InsertNode (NodesRemain);
               Counter:= Counter +1;
               if Counter = 20 then begin  Timer; Counter:=0;  end;
           end;
       Counter:= 0;  Timer;
       WriteBestTour;
    end;
end.
```

```
program CheapestAngle;
{$R-}
{
    method:      - find the convex hull of the nodes.
                 - for each node, find edge where cheapest insertion
                   would occur. (i,j) for node k.
                 - find which of these (i,k), (k,j) forms the largest
                   angle. Insert that node k between those i and j.
    conditions: - Euclidean
}

const
      maxN       = 101;
      LineLength = 250;

type
      NameType   = string [12];
      LineType   = string [LineLength];
      ZeroN      = 0..maxN;
      OneN       = 1..maxN;
      CoordArray = array [OneN] of real;
      TourArray  = array [0..1, OneN] of ZeroN;


var
  OutFile      : text;
  Metric       : char;
  Hour         : real;
  N, StartNode, Counter: ZeroN;
  NodesRemain, CFlag   : boolean;
  X, Y, Z      : CoordArray;
  V            : TourArray;



function ReadNumber (var LL: integer;
                     var B: LineType) : real;
var Int, OK : integer;
    PointFlag : boolean;
    R, RD, Decimal : real;
begin                                   { read one number }
  while (B [LL] = ' ') and (LL <= LineLength) do
    LL:= LL+1;                          { skip all spaces }
  Decimal:= 0.1;
  PointFlag:= false;
  R:=  0;
  RD:= 0;
  while (B [LL] <> ' ') and (LL <= LineLength) do
    begin                               { find the number }
      if (B [LL]= '.') then
        PointFlag:=true
      else
        begin
          val (B [LL], Int, OK);
          if PointFlag = false then
            R:= R*10 + Int
          else
            begin
              RD:= RD + (Int * Decimal);
```



```
              Decimal:= Decimal * 0.1;
            end;
        end;
      LL:= LL+1;
    end;                                { one value found }
  ReadNumber:= R + RD;
end;          { ReadNumber }


Procedure ReadData (var DataFile: text;
                    var MinX    : real;
                    var ZFlag   : boolean);
var L: integer;
    NumLines, NL, NumPerLine, NPL, I: OneN;
    A: LineType;
begin
  readln (DataFile, A);
  if A[1] <> 'C' then
    begin
      writeln; writeln;
      writeln ('******  Program requires coordinates of nodes !      ******');
      writeln ('******  The given data is not stored as coordinates. ******');
      CFlag:= false;
    end
  else
    begin
      L:= 4;
      N:= trunc (ReadNumber (L, A));
      ZFlag:= false;
      if A[2] = 'Z' then ZFlag:= true;
      if frac (N/3) > 0 then  NumLines:= trunc (N/3) + 1
      else                    NumLines:= trunc (N/3);
      NumPerLine:= 3;
      readln (DataFile, A);                                  { comment line }
      I:= 1;
      for NL:=1 to NumLines do                               { process data lines }
        begin
          for L:=1 to LineLength do  A[L]:= ' ';
          readln (DataFile, A);
          L:= 1;
          for NPL:=1 to NumPerLine do
            if I <= N then
              begin
                X [I]:= ReadNumber (L, A);
                Y [I]:= ReadNumber (L, A);
                Z [I]:= 0;
                if ZFlag then   Z [I]:= ReadNumber (L, A);
                if X [I] < MinX then
                  begin
                    MinX:= X [I];
                    StartNode:= I;          { set the first node on the hull }
                  end;
                I:= I+1;
              end;
        end;
    end;
end;          { ReadData }
```

```
procedure SetupProblem;
var
    DataFile: text;
    DataName: NameType;
    I       : OneN;
    MinX    : real;
    ZFlag   : boolean;
    Answer  : char;
begin
  repeat
    write ('Metric to be used (1, 2 or 0) ?');
    readln (Metric);
  until Metric in ['1', '2', '0'];
  write ('Result to be output to ?');
  readln (DataName);
  assign (OutFile, DataName);
  writeln;
  repeat
    writeln ('Data to be input by:');
    writeln ('                    C = Coordinates of nodes');
    writeln ('                    F = stored in a File');
    write   ('                   ?');
    readln (Answer);
    Answer:= upcase (Answer);
  until Answer in ['C', 'F'];
  CFlag:= true;
  if Answer = 'F' then
    begin
      write    ('Data file name ?');                    { input file }
      readln   (DataName);
      assign   (DataFile, DataName);
      reset    (DataFile);
      ReadData (DataFile, MinX. ZFlag);
      close    (DataFile);
    end
  else
    begin
      repeat
        write('How many nodes are there ? (Maximum ',maxN,') ');
        readln (N);
      until (N > 3) and (N <= maxN);
      writeln;
      ZFlag:= false;
      write ('z-coordinate to be entered ?');
      readln (Answer);
      if upcase (Answer) = 'Y' then  ZFlag:= true;
      writeln('Please input x, y (and z) coordinates of each node');
      for I:=1 to N do
        begin
          if ZFlag then
            begin
              write('   x  y  z  coordinates of node ',1,':');
              readln (X [I], Y [I], Z [I]);
            end
          else
            begin
              write('   x  y coordinates of node ',I,':');
              readln (X [I], Y [I]);
              Z [I]:= 0;
```



```
            end;
        end;
    end;
  if CFlag then
    begin
      MinX:= 99999.0;
      for I:=1 to N do
        begin
          if X [I] < MinX then
            begin
              MinX:= X [I];
              StartNode:= I;
            end;
          V [0,I]:= 0;
          V [1,I]:= 0;
        end;
      writeln;
      writeln ('Change floppy now if required.        Press any key to continue.
');
      repeat until keypressed;
      rewrite (OutFile);
      writeln (OutFile, DataName);
      NodesRemain:= true;
    end;
end;  { SetupProblem }


procedure Timer;
type RegPack = record
                 AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : integer;
               end;
var  Regs     : RegPack;
     TimeTest: real;
     Hr, Mn, Sc, Fr: integer;
begin
  with Regs do
    begin
      AX:= $2C00;
      msdos (Regs);
      Hr:= hi (CX);
      Mn:= lo (CX);
      Sc:= hi (DX);
      Fr:= lo (DX);
    end;
  if Counter = 0 then
    begin
      writeln (OutFile, ' time = ',Hr,':',Mn,':',Sc,'.',Fr:2);
      Hour:= Hr + (Mn /60);
    end
  else
    begin
      TimeTest:= Hr + (Mn /60) - Hour;
      WRITELN('TIME ELAPSED ',TIMETEST:6:2);
      if TimeTest > 2 then              { stop run after 2 hours }
        begin
          NodesRemain:= false;
          CFlag:= false;
        end
    end;
end;        { Timer }
```

```
Function Dist (var A, B: OneN)  : real;
var  DX, DY, DZ: real;
begin
  DX:= abs (X[B]-X[A]);
  DY:= abs (Y[B]-Y[A]);
  DZ:= abs (Z[B]-Z[A]);
  case Metric of
    '1': Dist:=       DX    + DY    + DZ;
    '2': Dist:= sqrt (DX*DX + DY*DY + DZ*DZ);
    '0': begin
           if (DX >= DY) and (DX >= DZ) then  Dist:= DX
           else           if (DY >= DZ) then  Dist:= DY
                          else                Dist:= DZ;
         end;
  end;
end;    { Dist }



function Line (var Point1, Point2: OneN;
               var XVal: real): real;
var  A: real;                                { compute using straight line }
begin
  if X [Point1] = X [Point2] then
    begin
      XVal:= X [Point1];
      if Y [Point1] < Y [Point2] then   Line:= Y [Point2] + 10
      else                              Line:= Y [Point2] - 10;
    end
  else
    begin
      A:= (Y [Point1] - Y [Point2]) / (X [Point1] - X [Point2]);
      Line:= A*XVal  +  Y[Point1] - (A*X[Point1]);              { Y = A*X + B }
    end;
end;    { Line }



Procedure ConvexHull (var Continue: boolean);
var
    I, NumNodes, OldNode1, Node1, LineNode1, NewNode1,
                 OldNode2, Node2, LineNode2, NewNode2: OneN;
    X1, Y1, X2, Y2, XTest, YTest, XLine1, YLine1, XLine2, YLine2,
                 MinCos, MaxCos1, MaxCos2, CosAngle: real;
    Search1, Search2: boolean;
                              { Find the convex hull of a set of points.
                                Input: x- and y-coordinates of the points.
                                Method: For each remaining node not on the
                                        hull, find the angle between it and
                                        an edge on the hull. The largest
                                        angle formed indicates the next
                                        node on the hull.    }


begin
  OldNode1:= StartNode;
  MinCos :=  2;
  MaxCos1:= -2;
  for I:=1 to N do
    if I <> OldNode1 then
```



```
      begin
        X1:= X [I] - X [OldNode1];
        Y1:= Y [I] - Y [OldNode1];
        CosAngle:= (- Y1 * Y [OldNode1]) /
                   (sqrt (X1*X1 + Y1*Y1) * Y [OldNode1]);
        if CosAngle < MinCos then
          begin
            Node2:= I;
            MinCos:= CosAngle;
          end
        else
          if CosAngle = MinCos then
            begin
              if dist (OldNode1, I) < dist (OldNode1, Node2) then
                Node2:= I;
            end;
        if CosAngle > MaxCos1 then
          begin
            Node1:= 1;
            MaxCos1:= CosAngle;
          end
        else
          if CosAngle = MaxCos1 then
            if dist (OldNode1, I) < dist (OldNode1, Node1) then
              Node1:= I;
      end;
  V [0, Node2]    := OldNode1;
  V [0, OldNode1]:= Node2;
  V [1, OldNode1]:= Node1;
  V [0, Node1]    := OldNode1;
  NumNodes:= 3;
  OldNode2:= OldNode1;
  Search1:= true;
  Search2:= true;
  while search1 or search2 do              { check all remaining angles }
    begin
      XLine1:= X [Node1] + 5;
      YLine1:= Line (OldNode1, Node1, XLine1);
      MaxCos1:= 2;
      XLine2:= X [Node2] + 5;
      YLine2:= Line (OldNode2, Node2, XLine2);
      MaxCos2:= 2;
      X1:= X [Node1] - XLine1;        { = 5 }
      Y1:= Y [Node1] - YLine1;
      X2:= X [Node2] - XLine2;        { = 5 }
      Y2:= Y [Node2] - YLine2;
      for I:=1 to N do
        if V [1,I] = 0 then
          begin
            if (I <> Node2) and (Search2) then
              begin                   { check for next node to be added (path 2) }
                XTest:= X [I] - X [Node2];
                YTest:= Y [I] - Y [Node2];
                CosAngle:= ((XTest * X2)  +  (YTest * Y2)) /
                           (sqrt (XTest*XTest + YTest*YTest) *
                            sqrt (X2*X2 + Y2*Y2));
                if CosAngle < MaxCos2 then
                  begin
                    NewNode2:= I;
                    MaxCos2:= CosAngle;
```

```
            end
          else
            if CosAngle = MaxCos2 then
              if dist (Node2, I) < dist (Node2, NewNode2) then
                NewNode2:= I;
        end;
      if (I <> Node1) and (Search1) then
        begin                    { check for next node to be added (path 1) }
          XTest:= X [I] - X [Node1];
          YTest:= Y [I] - Y [Node1];
          CosAngle:= ((X1 * XTest)  +  (Y1 * YTest)) /
                     (sqrt (X1*X1 + Y1*Y1) *
                      sqrt (XTest*XTest + YTest*YTest));
          if CosAngle < MaxCos1 then
            begin
              NewNode1:= I;
              MaxCos1:= CosAngle;
            end
          else
            if CosAngle = MaxCos1 then
              if dist (Node1, I) < dist (Node1, NewNode1) then
                NewNode1:= I;
        end;
    end;
  if Search1 then
    begin
      NumNodes:= NumNodes + 1;
      V [1, Node1]:= NewNode1;
      if V [0, NewNode1] = 0 then  V [0, NewNode1]:= Node1
      else                         V [1, NewNode1]:= Node1;
      OldNode1:= Node1;
      Node1:= NewNode1;
      if X [Node1] < X [OldNode1] then Search1:= false;
      if NewNode1 = Node2 then         Search1:= false;
    end;
  if Search2 then
    begin
      NumNodes:= NumNodes + 1;
      V [1, Node2]:= NewNode2;
      if V [0, NewNode2] = 0 then  V [0, NewNode2]:= Node2
      else                         V [1, NewNode2]:= Node2;
      OldNode2:= Node2;
      Node2:= NewNode2;
      if X [Node2] < X [OldNode2] then Search2:= false;
      if NewNode2 = OldNode1 then      Search2:= false;
      if NewNode1 = NewNode2 then
        begin
          Search1:= false;
          Search2:= false;
        end;
    end;
  end;
Continue:= false;                  { set for any nodes left inside the hull }
if NumNodes < N then Continue:= true;
end;   { Convex Hull }



procedure InsertNode (var Continue: boolean);
var NodeCount: ZeroN;
```



```
    I, HullNode1, HullNode2, Betw1, Betw2, NewNode, NBetw1, NBetw2: OneN;
    TestCost, MinCost, DX1, DX2, DY1, DY2, TestCos, MinCos: real;
begin                              { insert one node into the current subtour }
  NodeCount:= 0;
  MinCost:= 999900.0;
  MinCos:= 2.1;
  for I:=1 to N do
    if V [0, I] = 0 then                  { for each node not on subtour }
      begin
        NodeCount:= NodeCount +1;
        HullNode1:= StartNode;
        HullNode2:= V [1, StartNode];
        repeat                            { for each node on 'hull' (subtour) }
            TestCost:= Dist (HullNode1, I) + Dist (I,HullNode2) -
                       Dist (HullNode1, HullNode2);
            if TestCost < MinCost then
              begin
                MinCost:= TestCost;
                Betw1:= HullNode1;
                Betw2:= HullNode2;
              end;
            if V [0, HullNode2] = HullNode1 then
              begin
                HullNode1:= HullNode2;
                HullNode2:= V [1, HullNode2];
              end
            else
              begin
                HullNode1:= HullNode2;
                HullNode2:= V [0, HullNode2];
              end;
        until HullNode1 = StartNode;
        DX1:= X [I] - X [Betw1];
        DX2:= X [I] - X [Betw2];
        DY1:= Y [I] - Y [Betw1];
        DY2:= Y [I] - Y [Betw2];
        TestCos:= ((DX1 * DX2)  +  (DY1 * DY2)) /
                  (sqrt (DX1*DX1 + DY1*DY1) * sqrt (DX2*DX2 + DY2*DY2));
        if TestCos < MinCos then
          begin
            MinCos:= TestCos;
            NewNode:= I;
            NBetw1:= Betw1;
            NBetw2:= Betw2;
          end;
      end;
  V [0, NewNode]:= NBetw1;                { insert node into subtour }
  V [1, NewNode]:= NBetw2;
  if V [0, NBetw1] = NBetw2 then
     V [0, NBetw1]:= NewNode
  else
     V [1, NBetw1]:= NewNode;
  if V [0, NBetw2] = NBetw1 then
     V [0, NBetw2]:= NewNode
  else
     V [1, NBetw2]:= NewNode;
  if NodeCount = 1 then                   { the last node was inserted }
    Continue:= false;
end;    { InsertNode }
```

```
procedure WriteBestTour;
var
  Node, PrevNode: OneN;
  Out4    : ZeroN;
  TourCost: real;
begin
  if Cflag then
    begin
      TourCost:= 0;
      Out4:= 0;
      PrevNode:= 1;
      Node:= V [1, 1];
      repeat
          write (OutFile, '   ', PrevNode:4, Node:4);
          Out4:= Out4 +1;
          if Out4 = 4 then
            begin
              Out4:= 0;
              writeln (OutFile);
            end;
          TourCost:= TourCost + Dist (PrevNode, Node);
          if V [0, Node] = PrevNode then
            begin
              PrevNode:= Node;
              Node:= V [1, Node];
            end
          else
            begin
              PrevNode:= Node;
              Node:= V [0, Node];
            end;
      until PrevNode = 1;
      writeln (OutFile);
      writeln (OutFile, ' Cost of tour is ', TourCost:11:4);
    end
  else
    begin
      writeln (OutFile, 'Best tour is:');
      for Node:=1 to N do  begin
        write (OutFile, 'V [',Node:2,'] = ');
        if V[0,Node]>0 then write (OutFile, V[0,Node]:2,'  ')
        else  write('    ');
        if V[1,Node]>0 then write (OutFile, V[1,Node]:2,'  '); writeln; end;
    end;
  close (OutFile);
  writeln ('                  *** end of program ***');
end;    { WriteBestTour }


begin
  SetupProblem;
  if CFlag then
    begin
       Counter:= 0;  Timer;
       ConvexHull (NodesRemain);
       while NodesRemain do
           begin
```



```
                InsertNode (NodesRemain);
                Counter:= Counter +1;
                if Counter = 20 then begin  Timer; Counter:=0;  end;
            end;
        Counter:= 0;  Timer;
        WriteBestTour;
    end;
end.
```

```
program MSTPath;
{$R-}
{
    method:      find minimum spanning tree of nodes not in subpath.
                 Add a portion of the minimum spanning tree to the
                 current path to form a larger path.
                 Continue until no nodes remain.
    conditions:  non-Euclidean
                 symmetric


const
      maxN       = 101;
      maxNminus1 = 100;
      LineLength = 250;

type
      NameType   = string [12];
      LineType   = string [LineLength];
      ZeroN      = 0..maxN;
      OneNm1     = 1..maxNminus1;
      OneN       = 1..maxN;
      TwoN       = 2..maxN;
      CoordArray= array [OneN] of real;
      CostMatrx  = array [OneN, OneN] of real;
      PathMatrx  = array [OneN, 0..1] of ZeroN;
      TreeMatrx  = array [OneN, OneNm1] of ZeroN;

var
  N, End1, End2, NodesRemain: ZeroN;
  Cost       : CostMatrx;
  V          : PathMatrx;
  NodeInPath: array [OneN] of boolean;
  OutFile    : text;


Function ReadNumber (var LL: integer;
                     var B: LineType) : real;
var Int, OK : integer;
    PointFlag : boolean;
    R, RD, Decimal : real;
begin                                   { read one number }
  while (B [LL] = ' ') and (LL <= LineLength) do
    LL:= LL+1;                          { skip all spaces }
  Decimal:= 0.1;
  PointFlag:= false;
  R:=  0;
  RD:= 0;
  while (B [LL] <> ' ') and (LL <= LineLength) do
    begin                               { find the number }
      if (B [LL]= '.') then
        PointFlag:=true
      else
        begin
          val (B [LL], Int, OK);
          if PointFlag = false then
            R:= R*10 + Int
          else
            begin
```



```
              RD:= RD + (Int * Decimal);
              Decimal:= Decimal * 0.1;
            end;
        end;
      LL:= LL+1;
    end;                                { one value found }
  ReadNumber:= R + RD;
end;        { ReadNumber }


Procedure ReadData (var DataFile: text;
                    var CDFlag  : char;
                    var X, Y, Z : CoordArray;
                    var ZFlag   : boolean);
var L          : integer;
    NumLines, NL, NPL, I, J: OneN;
    NumPerLine: ZeroN;
    A          : LineType;
begin
  readln (DataFile, A);
  L:= 4;
  N:= trunc (ReadNumber (L, A));
  ZFlag:= false;
  if A [1] = 'C' then
    begin
      CDFlag:= 'C';
      if A [2] = 'Z' then  ZFlag:= true;
      if frac (N/3) > 0 then  NumLines:= trunc (N/3) + 1
      else                    NumLines:= trunc (N/3);
      NumPerLine:= 3;
    end
  else
    begin
      CDFlag:= 'D';
      NumLines:= N-1;
      NumPerLine:= N-1;
    end;
  readln (DataFile, A);                                 { comment line }
  I:= 1;
  J:= 2;
  for NL:=1 to NumLines do                              { process data lines }
    begin
      for L:=1 to LineLength do  A[L]:= ' ';
      readln (DataFile, A);
      L:= 1;
      for NPL:=1 to NumPerLine do
        if (CDFlag = 'C') and (I <= N) then
          begin
            X [I]:= ReadNumber (L, A);
            Y [I]:= ReadNumber (L, A);
            Z [I]:= 0;
            if ZFlag then Z [I]:= ReadNumber (L, A);
            I:= I+1;
          end
        else
          if CDFlag = 'D' then
            begin
              Cost [I,J]:= ReadNumber (L, A);
              if (L >= LineLength) and (Cost [I,J] = 0) then
```

```
            begin
              readln (DataFile, A);
              L:= 1;
              Cost [I,J]:= ReadNumber (L, A);
            end;
          Cost [J,I]:= Cost [I,J];
          J:= J+1;
          if J = I then  J:= J+1;
          if J > N then
            begin
              I:= I+1;
              J:= I+1
            end;
        end;
    if CDFlag = 'D' then
      NumPerLine:= NumPerLine -1;
  end;
end;          { ReadData }


procedure SetupProblem;
var
    DataFile      : text;
    DataName      : NameType;
    I, NodeI      : OneN;
    J             : TwoN;
    MinCost, DX, DY, DZ: real;
    ZFlag         : boolean;
    X, Y, Z       : array [OneN] of real;
    Answer, Metric: char;
begin
  repeat
    write ('Metric to be used (1, 2 or 0) ?');
    readln (Metric);
  until Metric in ['1', '2', '0'];
  write ('Results to be output to ?');
  readln (DataName);
  assign (OutFile, DataName);
  writeln;
  repeat
    writeln ('Data to be input by:');
    writeln ('                    C = Coordinates of nodes');
    writeln ('                    D = Distance between nodes');
    writeln ('                    F = stored in a File');
    write   ('                   ?');
    readln (Answer);
    Answer:= upcase (Answer);
  until Answer in ['C', 'D', 'F'];
  if Answer = 'F' then
    begin
      write    ('Data file name ?');                    { input file }
      readln   (DataName);
      assign   (DataFile, DataName);
      reset    (DataFile);
      ReadData (DataFile, Answer, X, Y, Z, ZFlag);
      close    (DataFile);
    end
  else
    begin
```



```
    repeat
      write('How many nodes are there ? (Maximum ',maxN,') ');
      readln (N);
    until (N > 3) and (N <= maxN);
    writeln;
    if Answer = 'D' then
      begin
        writeln('Please input costs / distances between nodes I and J');
        writeln('If a cost / distance does not exist, input 9999');
        for I:=1 to N-1 do
          for J:=I+1 to N do
            begin
              write('   cost / distance between nodes ',I,' ',J,' ');
              readln (Cost [I,J]);
              Cost [J,I]:= Cost [I,J];
            end;
      end
    else
      begin
        ZFlag:= false;
        write ('z-coordinate to be entered ?');
        readln (Answer);
        if upcase (Answer) = 'Y' then  ZFlag:= true;
        Answer:= 'C';
        writeln('Please input x-, y- (and z-) coordinates of each node');
        for NodeI:=1 to N do
          if ZFlag then
            begin
              write('      x  y  z  coordinates of node ',NodeI,' :');
              readln (X [NodeI], Y [NodeI], Z [NodeI]);
            end
          else
            begin
              write('      x  y  coordinates of node ',NodeI,' :');
              readln (X [NodeI], Y [NodeI]);
              Z [NodeI]:= 0;
            end;
      end;
  end;
MinCost:= 9999;
for I:=1 to N-1 do                                  { set first path }
  begin
    NodeInPath [I]:= false;
    V [I, 0]:= 0;
    V [I, 1]:= 0;
    for J:=I+1 to N do
      begin
        if Answer = 'C' then
        begin
          DX:= abs (X[I] - X[J]);
          DY:= abs (Y[I] - Y[J]);
          DZ:= abs (Z[I] - Z[J]);
          case Metric of
          '1':Cost[I,J]:=     DX    + DY    + DZ;
          '2':Cost[I,J]:=sqrt(DX*DX + DY*DY + DZ*DZ);
          '0':begin
                if (DX >= DY) and (DX >= DZ) then  Cost[I,J]:= DX
                else                               Cost[I,J]:= DY
                                                   else
                                                   Cost[I,J]:= DZ;
              end;
```

```
          end;
          Cost [J, I]:= Cost [I, J];
        end;
        if Cost [I,J] < MinCost then
          begin
            MinCost:= Cost [I,J];
            End1:= I;
            End2:= J;
          end;
      end;
  end;
  NodeInPath [N]:= false;
  V [N, 0]:= 0;
  V [N, 1]:= 0;
  V [End1, 0]:= End2;
  V [End2, 0]:= End1;
  writeln;
  writeln ('Change floppy now if required.        Press any key to continue.');
  repeat until keypressed;
  rewrite (OutFile);
  writeln (OutFile, DataName);
end;  { SetupProblem }



procedure Timer;
type RegPack = record
               , AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : integer;
               end;
var  Regs     : RegPack;
     Hr, Mn, Sc, Fr: integer;
begin
  with Regs do
    begin
      AX:= $2C00;
      msdos (Regs);
      Hr:= hi (CX);
      Mn:= lo (CX);
      Sc:= hi (DX);
      Fr:= lo (DX);
    end;
  writeln (OutFile, ' time = ',Hr,':',Mn,':',Sc,'.',Fr:2);
end;        { Timer }



procedure CheckForCycle (var T          : TreeMatrx;
                         From, Node     : ZeroN;
                         var Compare    : ZeroN;
                         var NoCycleFlag: boolean);
var Next, L: ZeroN;
begin                       { check all branches of Node against Compare node }
  L:=1;
  while (T [Node, L] > 0) and (NoCycleFlag = true) and (L < N) do
    begin
      Next:= T [Node, L];
      if (Next = From) and (T [Node, L+1] <> 0) then
        begin
          L:= L+1;
          Next:= T [Node, L];
```



```
        end;
      if Next <> From then
        begin
          if Next = Compare then        { cycle would be formed }
            NoCycleFlag:= false
          else                          { check branch of Node }
            CheckForCycle (T, Node, Next, Compare, NoCycleFlag);
        end;
      L:= L+1;
    end;
end;     { CheckForCycle }



procedure FindSpanningTree (var OpenNodes: ZeroN;
                            var T        : TreeMatrx);
var
  NoCycle   : boolean;
  LowestCost : real;
  I, J, LowestI, LowestJ, EdgesOnTree: ZeroN;
  Edge       : array [OneNm1, TwoN] of boolean;
begin
  EdgesOnTree:=0;                   { initially there are no edges }
  for I:=1 to N-1 do
    begin
      T [1, I]:= 0;
      for J:=2 to N do
        begin
          T [J, I]:= 0;
          Edge [I, J]:= false;
        end;
    end;
  if End1 < End2 then  Edge [End1, End2]:= true
  else                 Edge [End2, End1]:= true;
  while (EdgesOnTree <= OpenNodes) do
  { while there are valid edges left, connect up all nodes
  including the path ends into a minimum spanning tree.   }
   begin
    LowestCost:=9999;
    for I:=1 to N-1 do
     if NodeInPath [I] = false then
      for J:=I+1 to N do
       begin
        if (Edge [I, J] = false) and                  { 'open' edge remains }
           (NodeInPath [J] = false) and
           (Cost[I,J] < LowestCost) then                      { check cost }
          begin
            NoCycle:= true;
            if (T [I, 1] > 0) and  (T [J, 1] > 0) then
               CheckForCycle (T, 0, I, J, NoCycle);
            if NoCycle then
              begin
                LowestCost:= Cost[I,J];
                LowestI:= I;
                LowestJ:= J;
              end;
          end;
       end;
    if LowestCost < 9999 then                {include edge in spanning path }
      begin
```

```
        I:= 1;
        while (I < N) and (T [LowestI, 1] > 0) do  I:= I+1;
        T [LowestI, I]:= LowestJ;
        J:= 1;
        while (J < N) and (T [LowestJ, J] > 0) do  J:= J+1;
        T [LowestJ, J]:= LowestI;
        Edge [LowestI, LowestJ]:= true;
        EdgesOnTree:= EdgesOnTree + 1;
      end
    else                                    {no more valid edges are left}
      writeln('Run out of valid edges before tour could be completed.');
  end;
  OpenNodes:= 0;
end;     { FindSpanningTree }



procedure SetMinPath (var T     : TreeMatrx;
                      From, Node: ZeroN;
                      var TEnd  : ZeroN;
                      var Path  : boolean);
var  L, Next: ZeroN;
begin          { search for other end node and set all nodes along path to it }
  L:=1;
  while (T [Node, L] > 0) and (L < N) and (Path = false) do
    begin
      Next:= T [Node, L];
      if (Next = From) and (T [Node, L+1] <> 0) then
        begin                                          { not an end node }
          L:= L+1;
          Next:= T [Node, L];
        end;
      if Node = TEnd then
        begin
          Path:= True;
          Next:= V [Node, 0];
        end
      else
        if Next <> From then                          { continue looking }
          SetMinPath (T, Node, Next, TEnd, Path);
      L:= L+1;
    end;
  if Path then                                       { set path through this node }
    begin
      if From > 0 then    V [Node, 0]:= From;
      V [Node, 1]:= Next;
      NodeInPath [Node]:= true;
    end;
end;       { SetMinPath }



procedure FindPath (var OpenNodes: ZeroN;
                    var T        : TreeMatrx);
var
    Node, Next, T1, T2, Min1, Min2, Min3: ZeroN;
    MinCost, CompareCost: real;
    PathFlag: boolean;
begin                  { find the path along the MST which connects }
```



```
PathFlag:= false;           { the end nodes of the current path }
SetMinPath (T, 0, End1, End2, PathFlag);
NodeInPath [End1]:= true;
NodeInPath [End2]:= true;
for T1:=1 to N do                          { reset nodes not on path }
  if NodeInPath [T1] = false then
      OpenNodes:= OpenNodes + 1;
if OpenNodes > 0 then
  begin
    MinCost:= 9999;                        { find new path end nodes }
    Next:= End1;
    repeat
      if V [Next, 0] = Node then
        begin
          Node:= Next;
          Next:= V [Next, 1];
        end
      else
        begin
          Node:= Next;
          Next:= V [Next, 0];
        end;
      for T1:=1 to N-1 do
        for T2:= T1+1 to N do
          if (NodeInPath [T1] = false) and (NodeInPath [T2] = false) then
            begin
              CompareCost:= Cost [Node, T1] + Cost [Next, T2]
                            - Cost [Node, Next];
              if CompareCost < MinCost then
                begin
                  MinCost:= CompareCost;
                  Min1:= Node;
                  Min2:= Next;
                end;
              CompareCost:= Cost [Node, T2] + Cost [Next, T1]
                            - Cost [Node, Next];
              if CompareCost < MinCost then
                begin
                  MinCost:= CompareCost;
                  Min1:= Node;
                  Min2:= Next;
                end;
            end
          else   { if only one node remains, find closest insertion point }
            if OpenNodes = 1 then
              begin
                PathFlag:= false;
                if (NodeInPath [T1] = true) and
                   (NodeInPath [T2] = false) then
                  begin
                    PathFlag:= true;
                    Min3:= T2;
                  end;
                if (NodeInPath [T1] = false) and
                   (NodeInPath [T2] = true) then
                  begin
                    PathFlag:= true;
                    Min3:= T1;
                  end;
                if PathFlag then
```

```
                begin
                  CompareCost:= Cost [Node, Min3] + Cost [Next, Min3]
                                - Cost [Node, Next];
                  if CompareCost < MinCost then
                    begin
                      MinCost:= CompareCost;
                      Min1:= Node;
                      Min2:= Next;
                    end;
                end;
            end;
    until Next = End1;
    if OpenNodes = 1 then
      begin                                  { include last node into subtour }
        if V [Min1, 0] = Min2 then  V [Min1, 0]:= Min3
        else                        V [Min1, 1]:= Min3;
        if V [Min2, 0] = Min1 then  V [Min2, 0]:= Min3
        else                        V [Min2, 1]:= Min3;
        V [Min3, 0]:= Min1;
        V [Min3, 1]:= Min2;
        OpenNodes:= 0;
      end
    else
      begin                                  { change new subtour into a path }
        End1:= Min1;
        End2:= Min2;
        if  V [End1, 0] = End2  then
          begin
            V [End1, 0]:= V [End1, 1];
            V [End1, 1]:= 0;
          end
        else
            V [End1, 1]:= 0;
        if  V [End2, 0] = End1  then
          begin
            V [End2, 0]:= V [End2, 1];
            V [End2, 1]:= 0;
          end
        else
            V [End2, 1]:= 0;
        NodeInPath [End1]:= false;
        NodeInPath [End2]:= false;
      end;
  end;
end;    { FindPath }


procedure AddToPath (var NodesLeft: ZeroN);
var T: TreeMatrx;
begin
  FindSpanningTree (NodesLeft, T);
  FindPath         (NodesLeft, T);
end;   { AddToPath }



procedure WriteTour;
var  Node, Next, Out4: ZeroN;
     TourCost        : real;
begin
  TourCost:= 0;                                  { output the tour }
  Out4:= 0;
  Next:= End1;
  repeat
    if V [Next, 0] = Node then
      begin
        Node:= Next;
        Next:= V [Next, 1];
      end
    else
      begin
        Node:= Next;
        Next:= V [Next, 0];
      end;
    write (OutFile, Node:4, Next:4, '   ');
    Out4:= Out4 +1;
    if Out4 = 4 then
      begin
        Out4:= 0;
        writeln (OutFile);
      end;
    TourCost:= TourCost + Cost [Node, Next];
  until Next = End1;
  writeln (OutFile);
  writeln (OutFile, 'Cost of tour is ', TourCost:11:4);
  close (OutFile);
  writeln ('                    *** end of program ***');
end;   { WriteTour }



begin
  SetupProblem;
  Timer;
  NodesRemain:= N-2;
  while (NodesRemain > 0) do
          AddToPath (NodesRemain);
  Timer;
  WriteTour;
end.
```

```
      else
        if CDFlag = 'D' the
          begin
            Cost [I,J]:= Re
            if (L >= LineLe
              begin
                readln (Dat
                L:= 1;
                Cost [I,J]:
              end;
            Cost [J,I]:= Co
            J:= J+1;
            if J = I then
            if J > N then
              begin
                I:= I+1;
                J:= I+1
              end;
          end;
      if CDFlag = 'D' then
        NumPerLine:= NumPerLi
    end;
end;          { ReadData }



procedure SetupProblem;
var
    DataFile: text;
    DataName: NameType;
    I, J    : OneN;
    ZFlag   : boolean;
    X, Y, Z : array [OneN] of
    DX, DY, DZ    : real;
    Answer, Metric: char;
begin
  repeat
    write ('Metric to be used
    readln (Metric);
  until Metric in ['1', '2',
  write ('Result to be output
  readln (DataName);
  assign (OutFile, DataName);
  writeln;
  repeat
    writeln ('Data to be inpu
    writeln ('
    writeln ('
    writeln ('
    write   ('
    readln (Answer);
    Answer:= upcase (Answer);
  until Answer in ['C', 'D',
  if Answer = 'F' then
    begin
      write    ('Data file na
      readln   (DataName);
      assign   (DataFile, Dat
      reset    (DataFile);
```

```
program Savings;
{$R-}
{
    method:      for each node k
                 1. compute and order savings   S    =  C    +  C    -  C
                                                 ij      ki      kj      ij
                 2. for the largest remaining saving S    include edge (i,j).
                                                     ij
    conditions:  non-Euclidean
                 symmetric
    input:       - how many nodes
                 - input by distance:
                   for edge (i,j) connecting nodes i and j
                   input the cost/distance i.j.
                 - input by coordinates:
                   for each node i give its coordinates.
}

const
      maxN       = 101;
      Nless1     = 100;
      LineLength = 250;

type
      NameType  = string [12];
      LineType  = string [LineLength];
      ZeroN     = 0..maxN;
      OneN      = 1..maxN;
      CostMatrx  = array [OneN, OneN] of real;
      CoordArray = array [OneN] of real;
      TourMatrx  = array [OneN, 0..1] of ZeroN;
      TourArray  = array [OneN] of ZeroN;

var
  OutFile           : text;
  Cost              : CostMatrx;
  MinTour           : TourArray;
  MinCost, StartTime: real;
  N, NodeCount      : OneN;



Function ReadNumber (var LL: integer;
                     var B: LineType) : real;
var Int, OK : integer;
    PointFlag : boolean;
    R, RD, Decimal : real;
begin                                   { read one number }
  while (B [LL] = ' ') and (LL <= LineLength) do
    LL:= LL+1;                          { skip all spaces }
  Decimal:= 0.1;
  PointFlag:= false;
  R:=  0;
  RD:= 0;
  while (B [LL] <> ' ') and (LL <= LineLength) do
    begin                               { find the number }
      if (B [LL]= '.') then
        PointFlag:=true
      else
        begin
```



```
          val (B [LL], Int, OK);
          if PointFlag = false then
            R:= R*10 + Int
          else
            begin
              RD:= RD + (Int * Decimal);
              Decimal:= Decimal * 0.1;
            end;
        end;
      LL:= LL+1;
    end;                                { one value found }
  ReadNumber:= R + RD;
end;          { ReadNumber }


Procedure ReadData (var DataFile: text;
                    var CDFlag  : char;
                    var X, Y, Z : CoordArray;
                    var ZFlag   : boolean);
var L: integer;
    NumLines, NL, NPL, I, J: OneN;
    NumPerLine: ZeroN;
    A: LineType;
begin
  readln (DataFile, A);
  L:= 4;
  N:= trunc (ReadNumber (L, A));
  ZFlag:= false;
  if A [1] = 'C' then
    begin
      CDFlag:= 'C';
      if A[2] = 'Z' then  ZFlag:= true;
      if frac (N/3) > 0 then  NumLines:= trunc (N/3) + 1
      else                    NumLines:= trunc (N/3);
      NumPerLine:= 3;
    end
  else
    begin
      CDFlag:= 'D';
      NumLines:= N-1;
      NumPerLine:= N-1;
    end;
  readln (DataFile, A);                             { comment line }
  I:= 1;
  J:= 2;
  for NL:=1 to NumLines do                          { process data lines }
    begin
      for L:=1 to LineLength do  A[L]:= ' ';
      readln (DataFile, A);
      L:= 1;
      for NPL:=1 to NumPerLine do
        if (CDFlag = 'C') and (I <= N) then
          begin
            X [I]:= ReadNumber (L, A);
            Y [I]:= ReadNumber (L, A);
            Z [I]:= 0;
            if ZFlag then Z [I]:= ReadNumber (L, A);
            I:= I+1;
          end
```

```
  MinCost:= 999999.0;
  writeln;
  writeln ('Change floppy now if required.        Press any key to continue.');
  repeat until keypressed;
  rewrite (OutFile);
  writeln (OutFile, DataName);
  NodeCount:= N +1;
nd;  { SetupProblem }


rocedure Timer;
ype RegPack = record
                AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : integer;
              end;
ar  Regs     : RegPack;
    TimeNow : real;
    Hr, Mn, Sc, Fr: integer;
egin
  with Regs do
    begin
      AX:= $2C00;
      msdos (Regs);
      Hr:= hi (CX);
      Mn:= lo (CX);
      Sc:= hi (DX);
      Fr:= lo (DX);
    end;
  if NodeCount > N then
    begin
      writeln (OutFile, ' time = ',Hr,':',Mn,':',Sc,'.',Fr:2);
      StartTime:= Mn/60 + Hr;
    end
  else
    begin
      TimeNow:= Mn/60 + Hr;
      if (TimeNow - StartTime) > 2 then
        begin                                  { ran out of time }
          writeln (OutFile, ' Best out of ',NodeCount,' attempts.');
          NodeCount:= N +1;
        end;
    end;
nd;        { Timer }


rocedure CheckForSubtour (var Flag: boolean;
                          var Left, Right: ZeroN;
                          var Tour: TourMatrx);
ar FromNode, NextNode: OneN;
egin
  FromNode:= Left;                  { find last node on path from 'left' node }
  NextNode:= Tour [FromNode, 0];
  while Tour [NextNode, 1] > 0 do
    if Tour [NextNode, 0] = FromNode then
      begin
        FromNode:= NextNode;
        NextNode:= Tour [FromNode, 1];
      end
    else
```



```
      begin
        FromNode:= NextNode;
        NextNode:= Tour [FromNode, 0];
      end;
  if NextNode = Right then        { check if the last node = the 'right' node }
    Flag:= true;
end;      { CheckForSubtour }


Procedure StoreEdge (var Left, Right: ZeroN;
                     var TourCost    : real;
                     var Tour        : TourMatrx);
begin
  if Tour [Left, 0] = 0 then
    Tour [Left, 0]:= Right
  else
    Tour [Left, 1]:= Right;
  if Tour [Right, 0] = 0 then
    Tour [Right, 0]:= Left
  else
    Tour [Right, 1]:= Left;
  TourCost:= TourCost + Cost [Left, Right];
end;      { StoreEdge }


Procedure FindTour (K: OneN);
var  EdgeCount, Node1, Node2, I, J: ZeroN;
     TourCost, MaxSave, Save      : real;
     Tour: TourMatrx;
     Flag: boolean;
                              { for largest remaining saving S  ,  link I and J }
begin                         {                               IJ                 }
  TourCost:= 0;
  for I:=1 to N do
    begin
      Tour [I,1]:= 0;
      Tour [I,0]:= 0;
    end;
  for EdgeCount:=1 to N-2 do
    begin                              { search for the next edge to be included }
      MaxSave:= -9999;
      for I:=1 to N-1 do
       if (I <> K) and (Tour [I,1] = 0) then
        begin
         for J:=I+1 to N do
          if (J <> K) and (Tour [J, 1] = 0) then
            begin                                  { check saving for this edge }
              Flag:= false;
              if (Tour [I, 0] > 0) and (Tour [J, 0] > 0) then
                 CheckForSubtour (Flag, I, J, Tour);
              if Flag = false then
                begin
                  Save:= Cost [I,K] + Cost [J,K] - Cost [I,J];
                  if Save > MaxSave then
                    begin
                      Node1:= I;
                      Node2:= J;
                      MaxSave:= Save;
```

```
                end;
              end;
            end;
        end;
      StoreEdge (Node1, Node2, TourCost, Tour);
    end;
  Node1:= 0;                              { find last two connecting edges }
  Node2:= 0;
  EdgeCount:= 0;
  Flag := false;
  while Flag = false do
    begin
      EdgeCount:= EdgeCount + 1;
      if (EdgeCount <> K) and (Tour [EdgeCount, 1] = 0) then
        if Node1 = 0 then
          Node1:= EdgeCount
        else
          begin
            Node2:= EdgeCount;
            Flag:= true;
          end;
    end;
  EdgeCount:= K;
  StoreEdge (EdgeCount, Node1, TourCost, Tour);
  StoreEdge (EdgeCount, Node2, TourCost, Tour);
  if TourCost < MinCost then
    begin
      MinCost:= TourCost;
      Node1:= 1;                                          { store result }
      Node2:= Tour [1,0];
      repeat
        MinTour [Node1]:= Node2;
        if Tour [Node2, 0] = Node1 then
          begin
            Node1:= Node2;
            Node2:= Tour [Node1, 1];
          end
        else
          begin
            Node1:= Node2;
            Node2:= Tour [Node1, 0];
          end;
      until Node1 = 1;
    end;
  if NodeCount < N then  Timer;
end;      { FindTour }


procedure WriteBestTour;
var
  CurrentNode, NextNode: OneN;
  Out4                 : ZeroN;
begin
  Out4:= 0;
  CurrentNode:= 1;
  NextNode:= MinTour [1];
  repeat
    write(OutFile,'     ',CurrentNode:4,'    ',NextNode:4);
    Out4:= Out4 +1;
```



```
    if Out4 = 4 then
      begin
        Out4:= 0;
        writeln (OutFile);
      end;
    CurrentNode:= NextNode;
    NextNode:= MinTour [NextNode];
  until CurrentNode = 1;
  writeln(OutFile, ' Cost of tour is ', MinCost:11:4);
  close (OutFile);
  writeln ('                *** end of program ***');
end;     { WriteBestTour }


begin
  SetupProblem;
  Timer;
  NodeCount:= 1;
  while NodeCount <= N do
    begin
      FindTour (NodeCount);
      NodeCount:= NodeCount +1;
    end;
  Timer;
  WriteBestTour;
end.
```

```
program DynamicWeighting;
{$R-}
{
    method:      starting from one node, go to the next 'closest' node
                 until all nodes have been visited.
                 'Closeness' depends on weighting algorithm.
    conditions:  non-Euclidean
                 symmetric
    input:       - how many nodes
                 - for arc (i,j) connecting nodes i and j
                   input the cost/distance i,j
}

const
      maxN       = 101;
      maxNminus1 = 100;
      LineLength = 250;


type
      NameType   = string [12];
      LineType   = string [LineLength];
      ZeroN      = 0..maxN;
      OneN       = 1..maxN;
      CostArray  = array [OneN] of real;
      CostMatrx  = array [OneN, OneN] of real;
      TourArray  = array [OneN, 0..1] of ZeroN;

var
      OutFile           : text;
      Cost              : CostMatrx;
      N, NodeCount      : ZeroN;
      Tour              : TourArray;
      TourCost, MinTour, TimeStart: real;



Function ReadNumber (var LL: integer;
                     var B: LineType) : real;
var Int, OK         : integer;
    PointFlag       : boolean;
    R, RD, Decimal : real;
begin                                       { read one number }
  while (B [LL] = ' ') and (LL <= LineLength) do
    LL:= LL+1;                              { skip all spaces }
  Decimal:= 0.1;
  PointFlag:= false;
  R:=  0;
  RD:= 0;
  while (B [LL] <> ' ') and (LL <= LineLength) do
    begin                                   { find the number }
      if (B [LL]= '.') then
        PointFlag:=true
      else
        begin
          val (B [LL], Int, OK);
          if PointFlag = false then
            R:= R*10 + Int
          else
```



```
            begin
              RD:= RD + (Int * Decimal);
              Decimal:= Decimal * 0.1;
            end;
        end;
      LL:= LL+1;
    end;                                    { one value found }
  ReadNumber:= R + RD;
end;          { ReadNumber }


Procedure ReadData (var DataFile: text;
                    var CDFlag  : char;
                    var X, Y, Z : CostArray;
                    var ZFlag   : boolean);
var L: integer;
    NumLines, NL, NPL, I, J: OneN;
    NumPerLine: ZeroN;
    A: LineType;
begin
  readln (DataFile, A);
  L:= 4;
  N:= trunc (ReadNumber (L, A));
  ZFlag:= false;
  if A [1] = 'C' then
    begin
      CDFlag:= 'C';
      if A[2] = 'Z' then  ZFlag:= true;
      if frac (N/3) > 0 then  NumLines:= trunc (N/3) + 1
      else                    NumLines:= trunc (N/3);
      NumPerLine:= 3;
    end
  else
    begin
      CDFlag:= 'D';
      NumLines:= N;
      NumPerLine:= N-1;
    end;
  readln (DataFile, A);                                  { comment line }
  I:= 1;
  J:= 2;
  for NL:=1 to NumLines do                               { process data lines }
    begin
      for L:=1 to LineLength do  A[L]:= ' ';
      readln (DataFile, A);
      L:= 1;
      for NPL:=1 to NumPerLine do
        if (CDFlag = 'C') and (I <= N) then
          begin
            X [I]:= ReadNumber (L, A);
            Y [I]:= ReadNumber (L, A);
            Z [I]:= 0;
            if ZFlag then  Z [I]:= ReadNumber (L, A);
            I:= I+1;
          end
        else
          if CDFlag = 'D' then
            begin
              Cost [I,J]:= ReadNumber (L, A);
```

```
            if (L >= LineLength) and (Cost [I,J] = 0) then
              begin
                readln (DataFile, A);
                L:= 1;
                Cost [I,J]:= ReadNumber (L, A);
              end;
            Cost [J,I]:= Cost [I,J];
            J:= J+1;
            if J = I then  J:= J+1;
            if J > N then
              begin
                I:= I+1;
                J:= I+1;
              end;
          end;
      if CDFlag = 'D' then
        NumPerLine:= NumPerLine -1;
    end;
end;          { ReadData }



procedure SetupProblem;
var
    DataFile       : text;
    DataName       : NameType;
    Answer, Metric: char;
    I, J           : ZeroN;
    ZFlag          : boolean;
    X, Y, Z        : CostArray;
    DX, DY, DZ     : real;
begin
  repeat
    write ('Metric to be used (1, 2 or 0) ?');
    readln (Metric);
  until Metric in ['1', '2', '0'];
  write ('Result to be output to ?');
  readln (DataName);
  assign (OutFile, DataName);
  writeln;
  repeat
    writeln ('Data to be input by:');
    writeln ('                    C = Coordinates of nodes');
    writeln ('                    D = Distance between nodes');
    writeln ('                    F = stored in a File');
    write   ('                   ?');
    readln (Answer);
    Answer:= upcase (Answer);
  until Answer in ['C', 'D', 'F'];
  if Answer = 'F' then
    begin
      write    ('Data file name ?');                    { input file }
      readln   (DataName);
      assign   (DataFile, DataName);
      reset    (DataFile);
      ReadData (DataFile, Answer, X, Y, Z, ZFlag);
      close    (DataFile);
    end
```



```
else
  begin
    repeat
      write('How many nodes are there ? (Maximum ',maxN,') ');
      readln (N);
    until (N > 3) and (N <= maxN);
    writeln;
    if Answer = 'D' then
      begin
        writeln('Please input costs / distances between nodes I and J');
        for I:=1 to N-1 do
          for J:=I+1 to N do
            begin
              write('   cost / distance between nodes ',I,' ',J,' ');
              readln (Cost [I,J]);
              Cost [J,I]:= Cost [I,J];
            end;
      end
    else          { Answer was 'C' }
      begin
        ZFlag:= false;
        write ('z-coordinate to be entered ?');
        readln (Answer);
        if upcase (Answer) = 'Y' then  ZFlag:= true;
        Answer:= 'C';
        writeln('Please input x, y (and z) coordinate of each node');
        for I:=1 to N do
          if ZFlag then
            begin
              write('      x  y  z  coordinates of node ',I,' :');
              readln (X [I], Y [I], Z [I]);
            end
          else
            begin
              write('      x  y  coordinates of node ',I,' :');
              readln (X [I], Y [I]);
              Z [I]:= 0;
            end;
      end;
  end;
if Answer = 'C' then
  for I:=1 to N-1 do
    for J:=I+1 to N do
      begin
        DX:= abs (X[I] - X[J]);
        DY:= abs (Y[I] - Y[J]);
        DZ:= abs (Z[I] - Z[J]);
        case Metric of
        '1': Cost[I,J]:=      DX    + DY    + DZ;
        '2': Cost[I,J]:=sqrt (DX*DX + DY*DY + DZ*DZ);
        '0': begin
               if (DX >= DY) and (DX >= DZ) then  Cost[I,J]:= DX
               else           if (DY >= DZ) then  Cost[I,J]:= DY
                              else                Cost[I,J]:= DZ;
             end;
        end;
        Cost [J,I]:= Cost [I,J];
        COST [I,I]:= 0;  COST [J,J]:= 0;
      end;
writeln;
```

```
  writeln ('Change floppy now if required.        Press any key to continue.');
  repeat until keypressed;
  rewrite (OutFile);
  writeln (OutFile, DataName);
  MinTour:= 99999.0;
  TimeStart:= 0.0;
end;  { SetupProblem }



procedure Timer;
type RegPack = record
                 AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : integer;
               end;
var  Regs     : RegPack;
     TimeNow : real;
     Hr, Mn, Sc, Fr: integer;
begin
  with Regs do
    begin
      AX:= $2C00;
      msdos (Regs);
      Hr:= hi (CX);
      Mn:= lo (CX);
      Sc:= hi (DX);
      Fr:= lo (DX);
    end;
  if TimeStart = 0.0 then
    begin
      TimeStart:= Mn/60 + Hr;
      writeln (OutFile, ' time = ',Hr,':',Mn,':',Sc,'.',Fr:2);
    end
  else
    begin
      TimeNow:= Mn/60 + Hr;
      if (TimeNow - TimeStart) >= 2 then
        begin                  { assume program is started before 22h00 }
          writeln (OutFile, ' Best out of ', NodeCount, ' attempts');
          NodeCount:= N;
          TimeStart:= 0;
        end;
    end;
end;        { Timer }



function ApproxCost (StartNode, ViaNode, EndNode: ZeroN) : real;
var Sum, MinS, MinE, Min1, Min2: real;
    I, J: ZeroN;
begin                              { in-out estimator }
  MinS:= 999900.0;
  MinE:= 999900.0;
  Sum:= 0;
  for I:=1 to N do
    if (Tour [I,0] = 0) and
       (I <> StartNode) and (I <> ViaNode) and (I <> EndNode) then
      begin            { sum the minimum edges out of the start and end nodes }
        if Cost [StartNode, I] < MinS then   MinS:= Cost [StartNode, I];
        if Cost [EndNode, I]   < MinE then   MinE:= Cost [EndNode, I];
        Min1:= 999900.0;
```



```
        Min2:= 999900.0;
        for J:=1 to N do                    { for each node I not on the path, }
                                            { add it's two shortest edges }
          if (J <> I) and (Tour [J,0] = 0) and (J <> ViaNode) then
            begin
              if Cost [I, J] < Min1 then
                begin
                  Min2:= Min1;
                  Min1:= Cost [I, J];
                end
              else
                if Cost [I, J] < Min2 then
                  Min2:= Cost [I, J];
            end;
        Sum:= Sum + Min1 + Min2;
      end;
  Sum:= Sum + MinS + MinE;
  ApproxCost:= Sum / 2;
end;   { ApproxCost }



procedure FindSpanningPath (K: ZeroN);
var
  LowestCost, CompareCost, Weight: real;
  I, J, LowestI, LowestJ, CurrentNode: ZeroN;
begin                          { find the cheapest path starting at node K }
  CurrentNode:= K;
  TourCost:= 0;
  for I:=1 to N do                { initially no nodes are connected }
    Tour [I,0]:= 0;
  J:= 1;
  while (J < N) do
  { while there are nodes left,
  join the 'cheapest' arc from the current node to the path,
  leaving the first and last nodes on the path unconnected.
  'Cheapest' is determined by dynamic weighting algorithm. }
  begin
    J:= J + 1;
    LowestCost:=9999;
    LowestCost:=LowestCost*1000;
    for I:=1 to N do
      if (Tour [I,0] = 0) and (I <> CurrentNode) then
        begin                                                { no cycle formed }
          if J < (N-2) then                          { at least 3 nodes left }
            begin
              Weight:= 1 + exp(1) - (exp(1) * (J-1) / N);
(*            Weight:= 0.5;                                                  *)
              CompareCost:= Weight * ApproxCost (K, CurrentNode, I)
                            + Cost [CurrentNode,I];
(*                          + Weight * Cost [CurrentNode,I];                 *)
            end
          else
            CompareCost:= Cost [CurrentNode,I];
          if (CompareCost < LowestCost) then              { valid arc found }
            begin
              LowestCost:= CompareCost;
              LowestI:= I;
            end;
        end;
```

```
    Tour [CurrentNode,0]:= LowestI;          { include arc into spanning path }
    TourCost:= TourCost + Cost [CurrentNode, LowestI];
    CurrentNode:= LowestI;
  end;
end;    { FindSpanningPath }


procedure FindTour (K: ZeroN);
var
  I, LastNode: ZeroN;
begin
  { find the final arc which will connect the two remaining nodes
  that are the start and end nodes of the path                    }
  LastNode := 0;
  I:=0;
  while LastNode = 0 do              { search for node with no 'next node' }
    begin
      I:= I+1;
      if Tour [I,0] = 0 then
        LastNode:= I;
    end;
  Tour [LastNode, 0] := K;                    { close the tour }
  TourCost:= TourCost + Cost [LastNode, K];
  if TourCost < MinTour then                  { store cheapest tour }
    begin
      MinTour:= TourCost;
      for I:=1 to N do  Tour [I,1]:= Tour [I,0];
    end;
  Timer;
  if NodeCount = N then  TimeStart:= 0.0;
end;    { FindTour }


procedure WriteBestTour;
var
  I, Out4: ZeroN;
begin
  Out4:= 0;
  I:= 1;
  repeat
    write (OutFile,'     ',I:4,'  ',Tour [I,1]:4);
    Out4:= Out4 + 1;
    if Out4 = 4 then
      begin
        Out4:= 0;
        writeln (OutFile);
      end;
    I:= Tour [I,1];
  until I = 1;
  writeln (OutFile);
  writeln (OutFile,'    Cost is   ',MinTour:11:4);
  close (OutFile);
  writeln ('          *** end of program ***');
end;    { WriteBestTour }


begin
  SetupProblem;
  Timer;
  NodeCount:= 1;
```



```
  while NodeCount <= N do
    begin
      FindSpanningPath (NodeCount);
      FindTour (NodeCount);
      NodeCount:= NodeCount + 1;
    end;
  Timer;
  WriteBestTour;
end.
```

```
program VarioOfDynWeight;
{$R-}
{
    method:      starting from one node, go to the next 'closest' node
                 until all nodes have been visited.
                 'Closeness' depends on a variation of the algorithm
                 used in the dynamic weighting program.
    conditions: non-Euclidean
                 symmetric
                 no missing arcs
}

const
      maxN       = 101;
      maxNminus1 = 100;
      LineLength = 250;


type
      NameType   = string [12];
      LineType   = string [LineLength];
      ZeroN      = 0..maxN;
      OneN       = 1..maxN;
      CostArray  = array [OneN] of real;
      CostMatrx  = array [OneN, OneN] of real;
      TourArray  = array [OneN, 0..1] of ZeroN;

var
      OutFile             : text;
      Cost                : CostMatrx;
      N, NodeCount        : ZeroN;
      Tour                : TourArray;
      TourCost, MinTour, TimeStart: real;



Function ReadNumber (var LL: integer;
                     var B: LineType) : real;
var Int, OK         : integer;
    PointFlag       : boolean;
    R, RD, Decimal : real;
begin                                   { read one number }
  while (B [LL] = ' ') and (LL <= LineLength) do
    LL:= LL+1;                          { skip all spaces }
  Decimal:= 0.1;
  PointFlag:= false;
  R:=  0;
  RD:= 0;
  while (B [LL] <> ' ') and (LL <= LineLength) do
    begin                               { find the number }
      if (B [LL]= '.') then
        PointFlag:=true
      else
        begin
          val (B [LL], Int, OK);
          if PointFlag = false then
            R:= R*10 + Int
          else
            begin
```



```
              RD:= RD + (Int * Decimal);
              Decimal:= Decimal * 0.1;
            end;
        end;
      LL:= LL+1;
    end;                                { one value found }
  ReadNumber:= R + RD;
end;          { ReadNumber }



Procedure ReadData (var DataFile: text;
                    var CDFlag  : char;
                    var X, Y, Z : CostArray;
                    var ZFlag   : boolean);
var L: integer;
    NumLines, NL, NPL, I, J: OneN;
    NumPerLine: ZeroN;
    A: LineType;
begin
  readln (DataFile, A);
  L:= 4;
  N:= trunc (ReadNumber (L, A));
  ZFlag:= false;
  if A [1] = 'C' then
    begin
      CDFlag:= 'C';
      if A[2] = 'Z' then  ZFlag:= true;
      if frac (N/3) > 0 then  NumLines:= trunc (N/3) + 1
      else                    NumLines:= trunc (N/3);
      NumPerLine:= 3;
    end
  else
    begin
      CDFlag:= 'D';
      NumLines:= N;
      NumPerLine:= N-1;
    end;
  readln (DataFile, A);                                 { comment line }
  I:= 1;
  J:= 2;
  for NL:=1 to NumLines do                              { process data lines }
    begin
      for L:=1 to LineLength do  A[L]:= ' ';
      readln (DataFile, A);
      L:= 1;
      for NPL:=1 to NumPerLine do
        if (CDFlag = 'C') and (I <= N) then
          begin
            X [I]:= ReadNumber (L, A);
            Y [I]:= ReadNumber (L, A);
            Z [I]:= 0;
            if ZFlag then  Z [I]:= ReadNumber (L, A);
            I:= I+1;
          end
        else
          if CDFlag = 'D' then
            begin
              Cost [I,J]:= ReadNumber (L, A);
              if (L >= LineLength) and (Cost [I,J] = 0) then
```

```
              begin
                readln (DataFile, A);
                L:= 1;
                Cost [I,J]:= ReadNumber (L, A);
              end;
            Cost [J,I]:= Cost [I,J];
            J:= J+1;
            if J = I then  J:= J+1;
            if J > N then
              begin
                I:= I+1;
                J:= I+1;
              end;
          end;
      if CDFlag = 'D' then
        NumPerLine:= NumPerLine -1;
    end;
end;          { ReadData }



procedure SetupProblem;
var
    DataFile: text;
    DataName: NameType;
    Answer, Metric: char;
    I, J          : ZeroN;
    ZFlag         : boolean;
    X, Y, Z       : CostArray;
    DX, DY, DZ    : real;
begin
  repeat
    write ('Metric to be used (1, 2 or 0) ?');
    readln (Metric);
  until Metric in ['1', '2', '0'];
  write ('Result to be output to ?');
  readln (DataName);
  assign (OutFile, DataName);
  writeln;
  repeat
    writeln ('Data to be input by:');
    writeln ('                    C = Coordinates of nodes');
    writeln ('                    D = Distance between nodes');
    writeln ('                    F = stored in a File');
    write   ('                   ?');
    readln (Answer);
    Answer:= upcase (Answer);
  until Answer in ['C', 'D', 'F'];
  if Answer = 'F' then
    begin
      write    ('Data file name ?');                    { input file }
      readln   (DataName);
      assign   (DataFile, DataName);
      reset    (DataFile);
      ReadData (DataFile, Answer, X, Y, Z, ZFlag);
      close    (DataFile);
    end
  else
```



```
  begin
    repeat
      write('How many nodes are there ? (Maximum ',maxN,') ');
      readln (N);
    until (N > 3) and (N <= maxN);
    writeln;
    if Answer = 'D' then
      begin
        writeln('Please input costs / distances between nodes I and J');
        for I:=1 to N-1 do
          for J:=I+1 to N do
            begin
             write('   cost / distance between nodes ',I,' ',J,' ');
             readln (Cost [I,J]);
             Cost [J,I]:= Cost [I,J];
            end;
      end
    else          { Answer was 'C' }
      begin
        ZFlag:= false;
        write ('z-coordinate to be entered ?');
        readln (Answer);
        if upcase (Answer) = 'Y' then  ZFlag:= true;
        Answer:= 'C';
        writeln('Please input x, y (and z) coordinate of each node');
        for I:=1 to N do
          if ZFlag then
            begin
              write('      x  y  z  coordinates of node ',I,' :');
              readln (X [I], Y [I], Z [I]);
            end
          else
            begin
              write('      x  y  coordinates of node ',I,' :');
              readln (X [I], Y [I]);
              Z [I]:= 0;
            end;
      end;
  end;
if Answer = 'C' then
  for I:=1 to N-1 do
    for J:=I+1 to N do
      begin
        DX:= abs (X[I] - X[J]);
        DY:= abs (Y[I] - Y[J]);
        DZ:= abs (Z[I] - Z[J]);
        case Metric of
        '1': Cost[I,J]:=      DX    + DY    + DZ;
        '2': Cost[I,J]:=sqrt (DX*DX + DY*DY + DZ*DZ);
        '0': begin
               if (DX >= DY) and (DX >= DZ) then  Cost[I,J]:= DX
               else            if (DY >= DZ) then  Cost[I,J]:= DY
                               else                Cost[I,J]:= DZ;
             end;
        end;
        Cost [J,I]:= Cost [I,J];
      end;
writeln;
writeln ('Change floppy now if required.       Press any key to continue.');
repeat until keypressed;
```

```
  rewrite (OutFile);
  writeln (OutFile, DataName);
  MinTour:= 99999.0;
  TimeStart:= 0.0;
end;  { SetupProblem }



procedure Timer;
type RegPack = record
                 AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : integer;
               end;
var  Regs   : RegPack;
     TimeNow: real;
     Hr, Mn, Sc, Fr: integer;
begin
  with Regs do
    begin
      AX:= $2C00;
      msdos (Regs);
      Hr:= hi (CX);
      Mn:= lo (CX);
      Sc:= hi (DX);
      Fr:= lo (DX);
    end;
  if TimeStart = 0 then
    begin
      TimeStart:= Mn/60 + Hr;
      writeln (OutFile, ' time = ',Hr,':',Mn,':',Sc,'.',Fr:2);
    end
  else
    begin
      TimeNow:= Mn/60 + Hr;
      if (TimeNow - TimeStart) >= 2 then
        begin
          writeln (OutFile, '  Best out of ',NodeCount,' attempts.');
          NodeCount:= N;
          TimeStart:= 0;
        end;
    end;
end;        { Timer }



function ApproxCost (StartNode, ViaNode, EndNode: ZeroN) : real;
var Sum, MaxS, MaxE, Max1, Max2: real;
    I, J: ZeroN;
begin                                { in-out estimator }
  MaxS:= -9999;
  MaxE:= -9999;
  Sum:= 0;
  for I:=1 to N do
    if (Tour [I,0] = 0) and
       (I <> StartNode) and (I <> ViaNode) and (I <> EndNode) then
      begin             { sum the maximum edges out of the start and end nodes }
        if Cost [StartNode, I] > MaxS then   MaxS:= Cost [StartNode, I];
        if Cost [EndNode, I]   > MaxE then   MaxE:= Cost [EndNode, I];
        Max1:= -9999;
        Max2:= -9999;
        for J:=I+1 to N do                   { for each node not on the path, }
```



```
                                                  { add it's two longest edges }
          if (Tour [J,0] = 0) and (J <> ViaNode) then
            begin
              if Cost [I, J] > Max1 then
                begin
                  Max2:= Max1;
                  Max1:= Cost [I, J];
                end
              else
                if Cost [I, J] > Max2 then
                  Max2:= Cost [I, J];
            end;
        Sum:= Sum + Max1 + Max2;
      end;
  Sum:= Sum + MaxS + MaxE;
  ApproxCost:= Sum / 2;
end;   { ApproxCost }



procedure FindSpanningPath (K: ZeroN);
var
  LowestCost, CompareCost, Weight: real;
  I, J, LowestI, LowestJ, CurrentNode: ZeroN;
begin                          { find the cheapest path starting at node K }
  CurrentNode:= K;
  TourCost:= 0;
  for I:=1 to N do                { initially no nodes are connected }
    Tour [I,0]:= 0;
  J:= 1;
  while (J < N) do
  { while there are nodes left,
  join the 'cheapest' arc from the current node to the path,
  leaving the first and last nodes on the path unconnected.
  'Cheapest' is determined by dynamic weighting algorithm. }
  begin
    J:= J + 1;
    LowestCost:=9999;
    LowestCost:=LowestCost*1000;
    for I:=1 to N do
      if (Tour [I,0] = 0) and (I <> CurrentNode) then
        begin                                          { no cycle formed }
          Weight:= 1 + exp(1) - (exp(1) * (J-1) / N);
          CompareCost:= Weight * ApproxCost (K, CurrentNode, I)
                        + Cost [CurrentNode,I];
          if (CompareCost < LowestCost) then           { valid arc found }
            begin
              LowestCost:= CompareCost;
              LowestI:= I;
            end;
        end;
    Tour [CurrentNode,0]:= LowestI;          { include arc into spanning path }
    TourCost:= TourCost + Cost [CurrentNode, LowestI];
    CurrentNode:= LowestI;
  end;
end;    { FindSpanningPath }


procedure FindTour (K: ZeroN);
var
```

```
  I, LastNode: ZeroN;
begin
  { find the final arc which will connect the two remaining nodes
  that are the start and end nodes of the path                    }
  LastNode := 0;
  I:=0;
  while LastNode = 0 do              { search for node with no 'next node' }
    begin
      I:= I+1;
      if Tour [I,0] = 0 then
        LastNode:= I;
    end;
  Tour [LastNode, 0] := K;                        { close the tour }
  TourCost:= TourCost + Cost [LastNode, K];
  if TourCost < MinTour then                      { store cheapest tour }
    begin
      MinTour:= TourCost;
      for I:=1 to N do  Tour [I,1]:= Tour [I,0];
    end;
  Timer;
  if NodeCount = N then TimeStart:= 0.0
end;     { FindTour }


procedure WriteBestTour;
var
  I, Out4: ZeroN;
begin
  Out4:= 0;
  I:= 1;
  repeat
    write (OutFile,'     ',I:4,'   ',Tour [I,1]:4);
    Out4:= Out4 + 1;
    if Out4 = 4 then
      begin
        Out4:= 0;
        writeln (OutFile);
      end;
    I:= Tour [I,1];
  until I = 1;
  writeln (OutFile);
  writeln (OutFile,'     Cost is    ',MinTour:11:4);
  close (OutFile);
  writeln ('          *** end of program ***');
end;     { WriteBestTour }


begin
  SetupProblem;
  Timer;
  NodeCount:= 1;
  while NodeCount <= N do
    begin
      FindSpanningPath (NodeCount);
      FindTour (NodeCount);
      NodeCount:= NodeCount +1;
    end;
  Timer;
  WriteBestTour;
end.
```



```
program KLookAhead;
{$R-}
{
    method:      starting with a single node as a subtour, check the next
                 k nodes, then add k-1 edges to the subtour based on the
                 k nodes.  Repeat until all nodes have been visited.
    conditions: non-Euclidean
                 symmetric
}

const
      maxN       = 101;
      maxNminus1 = 100;
      LineLength = 250;


type
      NameType   = string [12];
      LineType   = string [LineLength];
      ZeroN      = 0..maxN;
      OneN       = 1..maxN;
      CoordArray= array [OneN] of real;
      CostMatrx  = array [OneN, OneN] of real;
      TourArray  = array [OneN] of ZeroN;

var
  OutFile         : text;
  Cost            : CostMatrx;
  N, K, NodeCount: ZeroN;
  Tour            : TourArray;
  TourCost, Hour : real;



Function ReadNumber (var LL: integer;
                     var B: LineType) : real;
var Int, OK         : integer;
    PointFlag       : boolean;
    R, RD, Decimal : real;
begin                                       { read one number }
  while (B [LL] = ' ') and (LL < LineLength) do
    LL:= LL+1;                              { skip all spaces }
  Decimal:= 0.1;
  PointFlag:= false;
  R:=  0;
  RD:= 0;
  while (B [LL] <> ' ') and (LL < LineLength) do
    begin                                   { find the number }
      if (B [LL]= '.') then
        PointFlag:=true
      else
        begin
          val (B [LL], Int, OK);
          if PointFlag = false then
            R:= R*10 + Int
          else
            begin
              RD:= RD + (Int * Decimal);
              Decimal:= Decimal * 0.1;
```

```
          end;
        end;
      LL:= LL+1;
    end;                                    { one value found }
  ReadNumber:= R + RD;
end;          { ReadNumber }


Procedure ReadData (var DataFile: text;
                    var CDFlag  : char;
                    var X, Y, Z : CoordArray;
                    var ZFlag   : boolean);
var L: integer;
    NumLines, NL, NPL, I, J: OneN;
    NumPerLine: ZeroN;
    A: LineType;
begin
  readln (DataFile, A);
  L:= 4;
  N:= trunc (ReadNumber (L, A));
  ZFlag:= false;
  if A [1] = 'C' then
    begin
      CDFlag:= 'C';
      if A[2] = 'Z' then  ZFlag:= true;
      if frac (N/3) > 0 then  NumLines:= trunc (N/3) + 1
      else                    NumLines:= trunc (N/3);
      NumPerLine:= 3;
    end
  else
    begin
      CDFlag:= 'D';
      NumLines:= N-1;
      NumPerLine:= N-1;
    end;
  readln (DataFile, A);                             { comment line }
  I:= 1;
  J:= 2;
  for NL:=1 to NumLines do                          { process data lines }
    begin
      for L:=1 to LineLength do  A[L]:= ' ';
      readln (DataFile, A);
      L:= 1;
      for NPL:=1 to NumPerLine do
        if (CDFlag = 'C') and (I <= N) then
          begin
            X [I]:= ReadNumber (L, A);
            Y [I]:= ReadNumber (L, A);
            Z [I]:= 0;
            if ZFlag then  Z [I]:= ReadNumber (L, A);
            I:= I+1;
          end
        else
          if CDFlag = 'D' then
            begin
              Cost [I,J]:= ReadNumber (L, A);
              if (L >= LineLength) and (Cost [I,J] = 0) then
                begin
                  readln (DataFile, A);
```



```
                  L:= 1;
                  Cost [I,J]:= ReadNumber (L, A);
                end;
              Cost [J,I]:= Cost [I,J];
              J:= J+1;
              if J = I then  J:= J+1;
              if J > N then
                begin
                  I:= I+1;
                  J:= I+1
                end;
            end;
      if CDFlag = 'D' then  NumPerLine:= NumPerLine -1;
    end;
end;          { ReadData }


procedure SetupProblem;
var
    DataFile: text;
    DataName: NameType;
    Answer, Metric: char;
    I, J          : ZeroN;
    ZFlag         : boolean;
    X, Y, Z       : CoordArray;
    DX, DY, DZ    : real;
begin
  repeat
    write ('Metric to be used (1, 2, 0) ?');
    readln (Metric);
  until Metric in ['1', '2', '0'];
  repeat
    write ('K-Node look ahead with K =?');
    readln (K);
  until (K>2) and (K<6);
  write ('Results to be output to ?');
  readln (DataName);
  assign (OutFile, DataName);
  writeln;
  repeat
    writeln ('Data to be input by:');
    writeln ('                    C = Coordinates of nodes');
    writeln ('                    D = Distance between nodes');
    writeln ('                    F = stored in a File');
    write   ('                   ?');
    readln (Answer);
    Answer:= upcase (Answer);
  until Answer in ['C', 'D', 'F'];
  if Answer = 'F' then
    begin
      write    ('Data file name ?');                    { input file }
      readln   (DataName);
      assign   (DataFile, DataName);
      reset    (DataFile);
      ReadData (DataFile, Answer, X, Y, Z, ZFlag);
      close    (DataFile);
    end
  else
    begin
```

```
    repeat
      write('How many nodes are there ? (Maximum ',maxN,') ');
      readln (N);
    until (N > 3) and (N <= maxN);
    writeln;
    case Answer of
    'D':  begin
            writeln('Please input costs / distances between nodes I and J');
            for I:=1 to N-1 do
              for J:=I+1 to N do
                begin
                  write('   cost / distance between nodes ',I,' ',J,' ');
                  readln (Cost [I,J]);
                  Cost [J,I]:= Cost [I,J];
                end;
          end;
     'C':  begin
            ZFlag:= false;
            write ('z-coordinate to be entered ?');
            readln (Answer);
            if upcase (Answer) = 'Y' then ZFlag:= true;
            Answer:= 'C';
            writeln('Please input x-, y- (and z-) coordinate of each node');
            for I:=1 to N do
              if ZFlag then
                begin
                  write('      x  y  z  coordinates of node ',I,' :');
                  readln (X [I], Y [I], Z [I]);
                end
              else
                begin
                  write('      x  y  coordinates of node ',I,' :');
                  readln (X [I], Y [I]);
                  Z [I]:= 0;
                end;
          end;
     end;
   end;
if Answer = 'C' then
 for I:=1 to N-1 do
  for J:=I+1 to N do
   begin
     DX:= abs (X[I] - X[J]);
     DY:= abs (Y[I] - Y[J]);
     DZ:= abs (Z[I] - Z[J]);
     case Metric of
     '1': Cost[I,J]:=      DX    + DY    + DZ;
     '2': Cost[I,J]:=sqrt (DX*DX + DY*DY + DZ*DZ);
     '0': begin
            if (DX >= DY) and (DX >= DZ) then  Cost[I,J]:= DX
            else               if (DY >= DZ) then  Cost[I,J]:= DY
                               else                Cost[I,J]:= DZ;
          end;
     end;
     Cost [J,I]:= Cost [I,J];
   end;
TourCost:= 999000000.0;
writeln;
writeln ('Change floppy now if required.       Press any key to conntinue.');
repeat until keypressed;
```



```
  rewrite (OutFile);
  writeln (OutFile, DataName);
  NodeCount:= 0;
  Hour:= 0;
end;  { SetupProblem }



procedure Timer;
type RegPack = record
                 AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : integer;
               end;
var  Regs     : RegPack;
     TimeTest: real;
     Hr, Mn, Sc, Fr: integer;
begin
  with Regs do
    begin
      AX:= $2C00;
      msdos (Regs);
      Hr:= hi (CX);
      Mn:= lo (CX);
      Sc:= hi (DX);
      Fr:= lo (DX);
    end;
  if (NodeCount = 0) or (NodeCount >= N) or (Hour = 0) then
    begin
      writeln (OutFile, ' time = ',Hr,':',Mn,':',Sc,'.',Fr:2);
      Hour:= Hr + (Mn /60);
    end
  else
    begin                                  { stop after 2 hours }
      TimeTest:= Hr + (Mn /60) - Hour;
      if TimeTest >= 2 then
        begin
          writeln (OutFile, 'This is the best out of ',NodeCount,' attempts.');
          NodeCount:= N +1;
          Hour:= 0;
          K:= 0;
        end;
    end;
end;        { Timer }



procedure FindSpanningPath (var TNum, Node: ZeroN;
                            var NewCost   : real;
                            var NewTour   : TourArray);
var
  LowestCost, CostC1, CostC2, CostC3, CostC4, CostC5: real;
  LowI                 : array [1..4] of ZeroN;
  Temp                 : ZeroN;
  I1, I2, I3, I4, I5, J: integer;
begin                { starting from node TNum, find the cheapest path of
                       length K-1, based on the path of K nodes long }
  LowI [1]:= 0;
  LowI [2]:= 0;
  LowI [3]:= 0;
  LowI [4]:= 0;
  Temp:= 0;
```

```
J:= 1;
while (J < N-1) do
{ while there are 2 or more nodes left,
join the 'cheapest' K-1 edges from the current node to the path,
leaving the first and last nodes on the path unconnected.
'Cheapest' is determined by the minimum cost of the next K edges. }
begin
  LowestCost:= 9999;
  LowestCost:= LowestCost * 1000;
  for I1:=1 to N do                    { pass over data }
   if (I1 <> Node) and (NewTour [I1] = 0) then
    begin
      CostC1:= Cost [Node, I1];
      for I2:=1 to N do
       if (I2 <> Node) and (I2 <> I1)
       and (NewTour [I2] = 0) then
        begin
         CostC2:= CostC1 + Cost [I1, I2];
         Temp:= NewTour [TNum];
         if J = N-2 then  NewTour [TNum]:= 0;
         for I3:=1 to N do
           if (I3 <> Node) and (I3 <> I1) and (I3 <> I2)
           and (NewTour [I3] = 0) then
            begin
              CostC3:= CostC2 + Cost [I2, I3];
              if (K>3) and (J <= N-3) then        { 3 or more nodes left }
               begin
                Temp:= NewTour [TNum];
                if J = N-3 then  NewTour [TNum]:= 0;
                for I4:=1 to N do
                 if (I4 <> Node) and (I4 <> I1)
                 and (I4 <> I2) and (I4 <> I3)
                 and (NewTour [I4] = 0) then
                  begin
                    CostC4:= CostC3 + Cost [I3, I4];
                    if (K=5) and (J <= N-4) then    { 4 or more nodes left }
                     begin
                      Temp:= NewTour [TNum];
                      if J = N-4 then  NewTour [TNum]:= 0;
                      for I5:=1 to N do
                       if (I5 <> Node) and (I5 <> I1)
                       and (I5 <> I2) and (I5 <> I3) and (I5 <> I4)
                       and (NewTour [I5] = 0) then
                        begin
                          CostC5:= CostC4 + Cost [I4, I5];
                          if CostC5 < LowestCost then
                            begin
                              LowestCost:= CostC5;
                              LowI [1]:= I1;
                              LowI [2]:= I2;
                              LowI [3]:= I3;
                              LowI [4]:= I4;
                            end;
                        end;  { I5 loop }
                      NewTour [TNum]:= Temp;
                     end
                    else
                      if CostC4 < LowestCost then
                        begin
                          LowestCost:= CostC4;
```



```
                          LowI [1]:= I1;
                          LowI [2]:= I2;
                          LowI [3]:= I3;
                        end;
                  end;  { I4 loop }
                NewTour [TNum]:= Temp;
               end
              else
                if CostC3 < LowestCost then
                  begin
                    LowestCost:= CostC3;
                    LowI [1]:= I1;
                    LowI [2]:= I2;
                  end;
            end;  { I3 loop }
          NewTour [TNum]:= Temp;
         end;  { I2 loop }
      end;  { I1 loop }
    NewTour [Node]    := LowI [1];            { include K-1 edges in path }
    NewTour [LowI [1]]:= LowI [2];
    NewCost:= NewCost + Cost [Node, LowI [1]] + Cost [LowI [1], LowI [2]];
    Node:= LowI [2];
    if (K>3) and (J <= N-3) then
      begin
        NewTour [LowI [2]]:= LowI [3];
        NewCost:= NewCost + Cost [LowI [2], LowI [3]];
        Node:= LowI [3];
        if (K=5) and (J <= N-4) then
          begin
            NewTour [LowI [3]]:= LowI [4];
            NewCost:= NewCost + Cost [LowI [3], LowI [4]];
            Node:= LowI [4];
          end;
      end;
    J:= J+ K-1;
  end;
end;    { FindSpanningPath }


procedure FindTour (var TNum, Node1: ZeroN;
                    var NewCost    : real;
                    var NewTour    : TourArray);
var
  I, Node2: ZeroN;
begin
{ find the final edge(s), if any, which will connect a remaining node,
  the start node and the end node of the path, if not connected.   }
  if Node1 <> TNum then
    begin
      Node2:= 0;
      for I:=1 to N do                  { search for node with no 'next' node }
        if (I <> Node1) and (NewTour [I] = 0) then  Node2:= I;
      if Node2 > 0 then                   { close the tour }
        begin
          NewTour [Node1]:= Node2;
          NewTour [Node2]:= TNum;
          NewCost:= NewCost + Cost [Node1, Node2] + Cost [Node2, TNum];
        end
      else
        begin
```

```
          NewTour [Node1]:= TNum;
          NewCost:= NewCost + Cost [Node1, TNum];
        end;
    end;
end;    { FindTour }


procedure FindNextTour (var StartNode: ZeroN);
var EndNode: ZeroN;
    NewCost: real;
    NewTour: TourArray;
begin
  NewCost:= 0;
  for EndNode:=1 to N do  NewTour [EndNode]:= 0;
  EndNode:= StartNode;
  FindSpanningPath (StartNode, EndNode, NewCost, NewTour);
  FindTour (StartNode, EndNode, NewCost, NewTour);
  if NewCost < TourCost then
    begin
      TourCost:= NewCost;
      for EndNode:=1 to N do  Tour [EndNode]:= NewTour [EndNode];
    end;
  Timer;
end;            { FindNextTour }


procedure WriteBestTour;
var
  Out4: ZeroN;
begin
  if K > 0 then   NodeCount:= 0;
  Out4:= 0;
  K:= 1;
  repeat
    write(OutFile, '   ', K:4, Tour [K]:4);
    Out4:= Out4 + 1;
    if Out4 = 4 then
      begin
        Out4:= 0;
        writeln (OutFile);
      end;
    K:= Tour [K];
  until K = 1;
  writeln (OutFile);
  writeln (OutFile, '     cost of tour is ',TourCost:11:4);
  close (OutFile);
  writeln ('              *** end of program ***');
end;    { WriteBestTour }


begin
  SetupProblem;
  Timer;
  NodeCount:= 1;
  while NodeCount <= N do
      begin
          FindNextTour (NodeCount);
          NodeCount:= NodeCount + 1;
```



```
      end;
  Timer;
  WriteBestTour;
nd.
```