The Internal Revenue Service Duty Location Modeling System - Programmer's Manual for Pascal Solver

Paul D. Domich, Richard H. F. Jackson, Marjorie A. McClain
David M. Tate

July 1986
Issued February 1987
THE INTERNAL REVENUE SERVICE
POST-OF-DUTY LOCATION MODELING
SYSTEM - PROGRAMMER'S MANUAL FOR
PASCAL SOLVER

Paul D. Domich, Richard H. F. Jackson, Marjorie A. McClain
David M. Tate

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
National Engineering Laboratory
Center for Applied Mathematics
Gaithersburg, MD 20899

July 1986
Issued February 1987

Sponsored by:
The Research Division
Internal Revenue Service
Washington, DC 20224
ABSTRACT

This report is a programmer's manual for a microcomputer system designed at the National Bureau of Standards for selecting optimal locations of IRS Posts-of-Duty. The mathematical model is the uncapacitated, fixed charge, facility location model which minimizes travel and facility costs. The package consists of two sections of code, one in FORTRAN and the other in PASCAL. The FORTRAN driver handles graphics displays and controls input and output for the solution procedure. This report discusses the mathematical techniques used to solve the mathematical model developed and includes a Greedy procedure, an Interchange procedure, and a Lagrangian approach to the related linear program. A description of these PASCAL routines and definitions of key data structures and variables are provided.

Key words: Uncapacitated fixed charge facility location problem, Greedy heuristic, Interchange heuristic, Lagrangian Relaxation.
# TABLE OF CONTENTS

Section I. INTRODUCTION ............................................. 4

Section II. METHODS..................................................... 7
1. Greedy Heuristic.................................................. 7
2. Teitz-Bart Interchange................................. 8
3. Graph Coloring............................................... 9
4. Lower Bounds and Lagrangian Relaxation............. 10

Section III. USER'S GUIDE.............................................. 11
1. System Requirements, Hardware..................... 11
2. Using the Package.......................................... 12

Section IV. THE CODE.................................................. 13
1. General Outline............................................. 13
2. List of Functions and Procedures.................. 14
3. Key Data Structures..................................... 16
4. Definition of Key Variables......................... 16
5. Input/Output Processing.................................. 19

BIBLIOGRAPHY.......................................................... 21

APPENDIX: PROGRAM LISTING........................................ 23
I. INTRODUCTION

The Internal Revenue Service Post-of-Duty Location System is a microcomputer package designed to assist IRS district planners in selecting locations for Post-of-Duty (POD’s) that will minimize total costs. This paper is part of a series of reports documenting the POD location system. The reports in the series are as follows.

1) **The Internal Revenue Service Post-of-Duty Location Modeling System: Final Report.**

   This report describes the post-of-duty location problem and its mathematical model. This report discusses the data used in calculating costs, describes the solution procedures, and provides a brief introduction to the computer implementation of the model (NBS Contact: Richard H. F. Jackson).

2) **The Internal Revenue Service Post-of-Duty Location Modeling System: User’s Manual.**

   This report is a user’s guide for the post-of-duty location computer system. This report gives the hardware and software requirements, the instructions for installing the system, a description of data files, and detailed instructions for operating the system (NBS Contact: Marjorie A. McClain).

3) **The Internal Revenue Service Post-of-Duty Location Modeling System: Programmer’s Manual for FORTRAN Driver.**

   This report describes the FORTRAN driver which handles graphics displays and controls input and output for the solution procedure. An alphabetical list of the FORTRAN routines includes a description of purpose, a list of variables, and the calling sequence (NBS Contact: Marjorie A. McClain).

4) **The Internal Revenue Service Post-of-Duty Location Modeling System: Programmer’s Manual for PASCAL Solver.**

   This report is a programmer’s manual for the PASCAL solver and describes the mathematical techniques used to solve the facility location problem. Included are a Greedy procedure, an Interchange procedure, and a Lagrangian approach to the related linear program. A description of these PASCAL routines and definitions of key data structures and variables are provided (NBS Contact: Paul D. Domich).
For the Internal Revenue Service (IRS), the facility location problem involves the placement of Posts-of-Duty (POD's) for a given tax district, according to the following model: locate k POD's so as to minimize the total "cost" of the allocation. This cost is the sum of the fixed costs incurred by opening or closing POD's, the operating costs for open POD sites, and the travel costs incurred by taxpayers and IRS personnel. The interested reader may also refer to any introductory textbook in Integer Programming (for example, Garfinkel and Nemhauser (1972), Hu (1969)) or one of the many papers on this subject (for example, Cornuejols, Fisher, and Nemhauser (1977), Erlenkotter (1978)) for a general mathematical description of the facility location model.

In the model, data is aggregated to a 5-digit zip code level. The travel cost of serving a given zip code is a function of the Euclidean distance to the nearest POD in the solution, the workload for that zip code in the period of interest (for example, one year), and the difficulty of travel between that particular zip code and the zip code in which the POD is located.

The disaggregated data for the problem comes in a variety of forms. The first is map data which includes co-ordinates used as the centroid to the zip code area, along with a list of zip code boundary points and of boundary segments of adjacent zip code areas. This data is provided by contract to IRS from Geographic Data Technology Inc. The centroid co-ordinates are used in calculating distances between zip code areas and for displaying the map of the state. Boundary points are used to draw the state map, while the list of adjacent zip code areas can be used to display the POD service regions for a given solution to the POD location problem.

A second source of data is workload data from the IRS Individual and Business Master Files and includes Examination, Collection, Taxpayer Service and Criminal Investigation workload data. Opening costs for new "potential" POD sites or closing costs for currently "existing" POD sites, and the cost of operating a POD facility in a particular zip code area are costs determined by the individual IRS District Offices. A more complete description of these costs follows.

For each zipcode-POD pair, workload is combined with the distance and travel difficulty factors between the two locations to produce a single factor which represents the cost of servicing the zip code by that POD site.
(see Report 1 for more information). The distance from zip code to POD is calculated using centroid co-ordinates from the geographic data mentioned above, while difficulty factors are user-specified.

The fixed costs represent the cost of opening a potential site or closing an existing site while operating costs are associated only with POD sites determined by the solver routine to be open. These costs are included directly in the zipcode-pair cost factors and are implicitly handled by the SOLVER routine. The model correctly accommodates the interactive changes made by the user without affecting the actual opening or closing costs for the POD sites; i.e., a currently existing POD site changed to a potential POD site by the user will not incur an opening cost, nor will a closing cost exist for a potential POD site designated by the user as currently existing. Thus, the initial POD file should reflect the current POD configuration and accurately specify opening and closing costs. Opening costs for potential POD sites not contained in the POD data file are interactively set by the user.

The operating cost for a POD site is computed in part from the zip code areas it services. For each zip code area the number of tax returns received is translated into a floor space requirement at a particular POD site. The cost of the floor space being different for each POD site requires that this cost be included with the travel cost associated with that zipcode-POD pair. Other related costs, for example overnight travel costs and parking costs, may also be added to this factor. This cost data can be used to determine the objective function coefficients for the facility location problem.

Finally, a list of zip code areas designated as potential or currently existing POD sites and a list of those zip code areas required to contain a POD site in any solution are required. The maximum distance allowed between a POD site and the zip codes it serves determines which zipcode-POD pairs are considered by the SOLVER routines. This distance represents the maximum distance either an agent or taxpayer is expected to travel and varies from region to region. This data along with the number of POD sites desired in the final solution define the constraints to the facility location problem.

Note: Tradenames and products mentioned in this report are not endorsed by the National Bureau of Standards nor does reference in this report imply any such endorsement.
II. METHODS

The method for finding a "good" solution to the IRS POD location problem is based on two well-known and dependable heuristic procedures. The first is the Greedy heuristic (see, for example, Cornuejols, et al., (1977)) and the other is the Teitz-Bart Interchange heuristic (Teitz and Bart (1968)). Also used by the procedure is a graph coloring algorithm, called the Sequential Least-first Interchange Algorithm (see Matula et al., (1972)), to display the final solution graphically. Each procedure is discussed below.

1. The Greedy Heuristic

In its simplest form, the Greedy heuristic for adding a POD to the current configuration proceeds as follows (see procedure GreedyADD in the Appendix).

1) Choose the "cheapest" POD site and assign all workload to that site.
2) Choose k, the final number of open POD sites desired in the optimal solution.
3) Among all allowable POD locations not currently in use, select that site S which would most diminish the total assignment cost for the problem, were it added to the current solution.
4) If this improvement is positive, and fewer than k sites are currently active, add site S to the active POD set, let k=k+1, and go to 3).
5) Else, stop.

The GreedySUB routine for removing a POD from the current configuration operates in a similar fashion.

The above procedure has been modified to accommodate the presence of feasibility restrictions for the IRS model. Specifically, because of the limit on the maximum travel distance from POD site to zip code area, an initial feasible solution must be provided by the user as input to the solver routines. Without these travel distance restrictions, step 1 would yield a feasible solution, though possibly at a large cost. Since the Greedy heuristic restricts itself to feasible assignments, it assumes that a feasible solution exists prior to altering POD sites.
The current POD configuration is an adequate initial solution provided
the distance limit is properly specified. Unfortunately, the pre-specified
distance limit may be less than the actual distance traveled between a zip
code and a POD site in the current configuration. Feasibility can be
attained by increasing the distance limit to the maximum actual distance
traveled. Note that as a result of altering the distance limit, the number
of feasible zipcode/POD pairs changes, which consequently, affects the
complexity of the problem.

The second modification is that the target number of facilities, $k$,
supercedes objective function tests; the Greedy heuristic adds or subtracts
facilities from the current set as long as feasibility is maintained.
It is possible that an increase in cost may result after adding a facility.
This may be a result of a large fixed cost associated with a particular POD
site or a temporary aberration in the current assignment which will be
adjusted later in the algorithm. The procedure will add the site regardless
of the effect on the objective value. This provides the user with control
over the number of open POD sites in the final solution, against the chance
that the number of facilities desired may be influenced by factors not
incorporated in the mathematical model.

In the event of such objective value degeneration, a warning message will
be printed to the user. Note that such worsening does not necessarily
imply that fewer facilities will yield an eventual solution which is better
than that yielded by a larger number of facilities. Rather, the Greedy
heuristic has exhausted all other advantageous POD sites given its initial
allocation. The final application of the interchange heuristic will attempt
to correct this objective function value deterioration.

Should the Greedy heuristic fail to find a feasible solution at some
iteration, the program will advise the user and continue with the last known
feasible number of facilities as the target number in all subsequent calcul-
ations.

2. Teitz-Bart Interchange.

Once the target number of facilities has been allocated by the Greedy
heuristic, the solution procedure tries to determine a better solution with
the same specified number of open POD sites. The procedure iteratively
locates pairs of POD sites, one which is presently selected and one not, such that if the two are interchanged in the current configuration, the overall cost is reduced. When no such pair exists, the routine terminates with the last configuration. The following heuristic, which is a modified version of that of Teitz and Bart (1968) is used:

1) Partition the set of allowable sites into two sets, A and B, where A is the set of currently assigned sites and B is all other potential POD sites.
2) Look for a pair of sites, a in A and b in B such that
   (i) \( \text{cost}(A - \{a\} + \{b\}) \) is less than \( \text{cost}(A) \),
   (ii) a is not required to be a POD site, and
   (iii) \( A - \{a\} + \{b\} \) is feasible.
3) For all pairs satisfying 2, select that pair which produces the largest improvement and exchange a for b in the set of active sites. Go to 2.
4) If no such pair exists, stop.

The modification of step 2 parts (ii) and (iii) are excluded in the original reference which did not have the initial feasibility restrictions. Because of the travel distance limit previously described, an initial feasible solution is required. The combination of the Greedy heuristic followed by the Interchange heuristic is well known to produce very good solutions to the facility location problem (see, for example, Cornuejols, Fisher, and Nemhauser (1977))

3. Graph Coloring.

To display in color the final assignment of zip code areas to POD locations, it is necessary to ensure that no two adjacent POD service areas, i.e., two areas sharing a common border, are colored with the same color. This is a map coloring problem, where the regions involved are groups of customers aggregated by their assigned POD facility. The problem is to choose colors \( C_j \) for the regions \( V_j \) of a graph \( G \), such that \( C_i \) is not equal to \( C_j \) if \( V_i \) and \( V_j \) are adjacent regions, and in such a fashion that a "small" number of colors are used. Since all of the zip code maps can be represented as planar graphs (i.e., graphs that can be drawn on a sheet of paper so that no two edges cross), theoretically all can be colored using only four colors. In practice, to find a four-coloring is a very difficult problem, so a five- or six-coloring is used. For a description of the
coloring algorithm, see Matula, et al, "Graph Coloring Algorithms", (1972). The procedure used is called the Sequential Least-first Interchange heuristic (SLI) and is presented in the Appendix.

4. Lower Bounds and Lagrangian Relaxation.

As previously stated, the Greedy heuristic and the Interchange heuristic described above are well-known to produce good solutions to the facility location problem. One drawback with these procedures involves determining when the generated solution is in fact the optimal integral solution to the described problem. One way to demonstrate the optimality of a solution involves generating lower bounds to the optimal objective function value. One bound can be obtained by solving the linear programming (LP) relaxation of the original problem, i.e., the original problem without the integrality constraints.

In general the LP formulation of the facility location problem has a large number of constraints in the problem description and it, too, can be difficult to solve. One Lagrangian relaxation of this LP problem removes the requirement that a zip code is serviced by exactly one POD and adds a penalty to the objective value for any violation of these constraints. This relaxation can produce the optimal LP objective function value in an iterative manner, and provide at each step a lower bound to the optimal solution to the original facility location problem. Further, by rounding the possibly fractional real-valued solution produced by this method, an improved integral solution may be found as a by-product. The interested reader may refer to the many articles in this subject (e.g., Cornuejols, et al. (1977), Fisher(1982)).

The Lagrangian solution procedure will relax the constraints requiring that a zip code area can be serviced by exactly one POD site while penalizing the objective function for any violation in these constraints. A feasible solution to the relaxed problem is found and the penalty factors are modified in a manner which forces the relaxed constraints to be satisfied. This iterative procedure generates a series of objective values which are lower bounds to the optimal integral objective value to the original problem. Further, by rounding the real-valued objective value
produced by this method, an improved integral objective value may be found as a by-product of the procedure.

Often for the facility location problem, the optimal LP objective value is equal to the optimal integral objective value (see, for example, Morris (1978)) and therefore the optimality of the heuristic integral solution can be demonstrated using the real-valued objective value. Otherwise, either there exists a "gap" between the optimal LP objective value and the optimal integral objective value, or the integral solution is nonoptimal. In the latter case, the bound provides an estimate on the "goodness" of the integral solution value.

III. USER'S GUIDE

1. System Requirements

The SOLVER package, and the graphics environment in which it runs, are written specifically for systems running MS/DOS on an IBM PC compatible (Intel 8088-based) microcomputer with a math co-processor and a 10mb fixed disk. When executing the FORTRAN driver routines, it is essential to have the math co-processor to ensure correct type-matching in the input data files produced by driver routines, as well as desirable speed of execution. The graphics capability is provided via a number of different hardware and software functions. Included are the following:

Graphics Display Monitor,
Graphics Expansion Card,
IBM Graphical Kernel System.

The SOLVER routines are written in TURBO PASCAL (Borland International Inc., (1985)). There are several reasons for choosing Pascal as the language for the SOLVER and these are summarized below.

1. Pascal has a dynamic storage capability, permitting a more efficient use of core memory than is possible in static-allocation languages like FORTRAN, BASIC, and APL. This is essential to solve large problems.

2. TURBO PASCAL compiles about one order of magnitude faster than other available Pascal compilers, and several orders of magnitude faster than available FORTRAN. As an example, the 1950 lines of code in the SOLVER program compile and link in under 30 seconds, to a file of only 18K
3. Pascal supports pointer variables and structured data-types (user-defined records), making for much more legible, structured, and easily altered code.

4. TURBO PASCAL is about one-fifth the price of most other Pascal or FORTRAN packages, and includes a number of graphics and utility programs in this price. It runs its own developmental operating system, and traps and locates run-time errors automatically, thus greatly enhancing program development.

The flexibility provided by the Pascal programming language allows development of a well-structured program which is easily understood. The only limitation of the language in this application involved data transfers. This problem was resolved using a FORTRAN unformatted write statement in the preprocessor graphics routines which create the data files used by the SOLVER routines. I/O issues are discussed in Section 4.

2. Using the Package

The SOLVER package is used as a subprogram to the IRS POD Location Modeling System which performs all preprocessed of input data and graphically displays workload data and SOLVER's final solution. As input data the SOLVER routine requires a single file (called STATE_.DBL) that is automatically generated by the graphics package (see the IRS Post-Of-Duty Location Modeling System: User's Manual). This file defines the facility location problem and contains information about the individual zip code areas and also specifies assignment costs from zip code areas to the feasible POD sites. For computational efficiency, this file is written in binary format. The exact commands needed to call SOLVER from the main program are discussed in detail in the report mentioned above.

Once the driver routine generates the input files for the solver routines, the user is provided with a summary of the problem characteristics, followed by a query to the user for additional information on the number of POD's desired in the final solution. Once the current problem is fully described, control is passed to the SOLVER routines and the following steps occur. The following text illustrates this phase of the program:
Total number of zip codes is \( nnn \).
Number of possible POD's is \( mmm \).
Number of existing POD's is \( lll \).

Enter the desired number of POD's in the final solution: \( kkk \)

where \( nnn, mmm, lll, \) and \( kkk \) are integer values. After the last prompt has been answered, the solver proceeds to solve the POD location problem. An in-depth examination of the solver routine is given in the next section.

IV. THE CODE

1. General Outline

The structure of the solver routines involves four basic program units. The first performs the input of the facility location problem as defined in the pre-processor graphics package (see the IRS Post-Of-Duty Location Modeling System: User's Manual). The problem file is read and entered into the data structures and, from the existing configuration of POD's, an initial interchange is performed so as to locate the best possible solution given the original number of POD facilities. Next, the number of POD's is altered by adding or deleting POD's as required via the Greedy heuristic. Upon termination of the Greedy heuristic, a final interchange is performed which seeks the best possible solution of the given size. Finally, a graph coloring is performed so as to display the POD service areas in the final solution.

The code for the solver portion of the package is found in the files:

```
SOLVER.PAS,
INIT.PAS,
DSTRUCT.PAS,
GREEDY.PAS,
INTCHG.PAS,
FTVCLR.PAS,
PODCLR.PAS,
LGRN.PAS.
```
The SOLVER file contains the driver program as well as routines to compute the cost of an allocation (i.e., assign customers to their nearest facility) and to output the current solution. The routine INIT performs array, set, and pointer initializations. The sparse-matrix data structures determined from the input data are set up by DSTRUCT. The procedure GREEDY performs the Greedy heuristic calculations, while INTCHG is the interchange heuristic algorithm. General utility routines, denoted as InsertPOD and DeletePOD, add and delete POD’s from the data structure, re-establishing the data structure for the new set of POD’s. The LGRN routine determines a lower bound on the optimal integer-valued solution to the problem and can be used to verify the solution found by the Greedy and interchange heuristics. The FIVCLR and PODCLR routines are used to determine a coloring of the final solution map for displaying the POD service regions.

2. List of Functions and Procedures:

The following is a list of procedures and functions, and their purposes:

**FUNCTIONS:**

- **SwapVal(old,new)**
  Returns the change in objective function value associated with an exchange of facility "new" for facility "old" in the current set of facilities.

- **Exist(filename)**
  Boolean function returning true if the string "filename" is the name of a current disk file, false otherwise.

**PROCEDURES:**

- **Match**
  Associates with each customer area (zip code) the nearest facility in the current set of facilities. Values are set for the arrays BestPOD[], NextBestPOD[], CurrentCost[], NextCost[].

- **ComputeCost**
  Adds all assignment costs to find the current objective function value.

- **ListCurrent**
  Sends a list of current POD assignments to the default list device.

- **Initialize**
  Zeroes arrays, empties sets, and NIL's pointers prior to program execution.
CreateDataStructures: Reads zip code data from the special file STATE_.DBL then establishes the sparse array data structure which has cost and feasibility data for specific POD allocations. Rows of the array are pointed to by the vector of pointers Map[], and columns are pointed to by the pointer fields of the vector of records CanBe[]. As each record of data is read for a feasible zipcode/POD pair, an entry in the sparse array is created, specifying the zip code index and the zip code index of the POD site involved, the cost, and a pointer to the next zip code entry for that POD site and a pointer to the next POD site for that zip code. This record is inserted in the data structure ordered by increasing cost. Rows correspond to all POD sites which may feasibly serve a given zip; columns correspond to all zips which may be feasibly served by a given POD site.

Greedy: Performs the Greedy heuristic as described in Section II.1.

GreedyADD: Increases the number of POD's by one, according to the Greedy heuristic.

GreedySUB: Decreases number of POD's by one, according to the Greedy heuristic (if feasible).

Interchange: Performs interchange heuristic on problem, as described above.

InsertPOD: Performs the insertion of a POD to the current set and updates the BestPOD[], NextBestPOD[], CurrentCost[], NextCost[] arrays.

RemovePOD: Performs the removal of a POD from the current set and updates the BestPOD[], NextBestPOD[], CurrentCost[], NextCost[] arrays.

GraphColor: Performs the sequential least-first interchange coloring algorithm on the graph of the final solution, coloring POD "spheres of influence" to avoid having identical adjacent colors.

Lagrangian_dual: Computes a lower bound on the best possible solution to the problem. Can be used to verify the optimality of the heuristic solutions.

Quick_Sort: Performs a sort of a vector of real numbers.
3. Key Data Structures:

The key data structure in the solver program is a doubly linked-list for maintaining the zipcode/POD pair data. The basic element of this structure is a five-field record, defined as follows:

(1) node,
(2) target,
(3) cost,
(4) nextZip, and
(5) nextPOD.

The "node" field is the index of the zip code for this record. The "target" field is the index of the potential POD site to which this node refers. "Cost" is the cost of assigning zip "node" to POD "target" (if node=target, then this also includes the fixed operating cost of having a POD at target). The entry "nextZip" is a pointer to the next record which refers to POD site "target", and "nextPOD" is a pointer to the next record which refers to zip code index "node".

Map[1..MaxZips] is an array whose entries for any given zip, are pointers to the linked records by POD, and CanBe[1..MaxPossible] is an array whose entries are records, one field of which is, for any given allowable POD site, a pointer to the linked records by zip code index. Thus, starting with Map[27] and following the "nextPOD" links results in a linked list of records corresponding to all possible POD's which can serve zip code index #27 with their associated costs. This linked list of potential POD sites is sorted in order of increasing cost.

Similarly, starting with CanBe[11].next (the pointer field of the 27th entry of array CanBe) and following the NextZip links produces a linked list of records corresponding to all zip code indices which can be served from the 11th allowable POD site. Both of these data structures are static, in the sense that once they are created (by procedure CreateDataStructures), they will never change.

4. Definition of Key Variables:

There are certain global variables in the program that the programmer should be familiar with before attempting to modify the code. This section
will list the most important variables and their definitions and structures (if any). First the various Pascal constants and variable types are introduced.

CONSTANTS:

MaxZips = 1000; 
Maximum number of zip code areas allowed. This constant may be changed.

MaxPossible = 85; 
Maximum number of possible POD sites allowed. This constant may be changed but can not exceed 256.

These two constants determine the size of the various storage arrays used in the SOLVER routines. Consequently, limiting the size of these constants will lower the storage requirements for the system.

VARIABLE TYPES:

Zcode = 0..MaxZips; 
Integer type in the range [0,Maxzips]

ZipSet = set of 1..Maxpossible; 
NOTE: The "set" data type is an implementation dependent type. TURBO PASCAL allows set types up to 255 distinct possible elements. This means that in no case can MaxPossible be set to a value of more than 256.

Link = 'Neighbor; 
A Pascal pointer type for record of type Neighbor.

Neighbor
Site 
zip code index of type Zcode,
Target 
zip code index of POD's of type Zcode,
Cost 
cost of Site-Target assignment,
NextZip 
pointer to the next zip code,
NextPOD 
pointer to the next POD zip code.

PODsite
Where 
the zip code index of type Zcode,
Must 
boolean flag for a required POD site,
Next 
a Pascal link to the first of its neighbors.

SingleZip
zip code 
actual zip code number,
SType 
site type;
FixCost

SType=0 => never a POD site,
SType=1 => can be a POD site,
SType=2 => must be a POD site,
Opening/Closing cost for a POD site.

PairOfZips

<table>
<thead>
<tr>
<th>Data record type which includes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>zip code index of zip code area,</td>
</tr>
<tr>
<td>zip code index of POD site,</td>
</tr>
<tr>
<td>cost of area to site assignment.</td>
</tr>
</tbody>
</table>

ColumnPointArray

Array type of length MaxPossible of PODsite.

RowPointArray

Array type of length MaxZips of Link.

IndexArray

Array type of length MaxZips of Zcode.

ValueArray

Array type of length MaxZips of real.

FileString

Character string of length 15.5.

VARIABLE DEFINITIONS:

CurrentPODs

The set of POD’s in the current assignment.

PossiblePODs

The set of all possible POD sites.

BestPOD[1..MaxZips]

The POD index which is the nearest POD in the current solution.

NextBestPOD[1..MaxZips]

The POD index which is the second best POD in the current solution.

CurrentCost[1..MaxZips]

Value of the cost of the BestPOD.

NextCost[1..MaxZips]

Value of the cost of the NextbestPOD.

CanBe[1..MaxPossible]

This is an array of ColumnPoint Array storing the index of the POD site and a pointer to the linked list of zip code areas reached from that POD site.

Index[1..MaxZips]

Pointer for all possible POD sites to records in array CanBe.

Map[1..MaxZips]

This is an array of RowPointArray pointing to the start of the linked list of feasible POD sites for a zip code area.

ZCreal[1..MaxZips]

The actual zip code number.
### 6. Input/Output Processing:

Input to the SOLVER routines comes from the `STATE___.DBL` file where `__` refers to the index of the tax district (1 to 76). The DBL file is written by the driver routines using an unformatted FORTRAN write statement. This file consists of sets of records, each set preceded by, and followed by, a two byte word indicating the total number of bytes used in that set (see the IBM Professional FORTRAN Reference Guide). The following is a representation of one such set:

$$
\text{Word}1, i, \text{ZIP}_i, j_1, c_{ij_1}, j_2, c_{ij_2}, \ldots, j_k, c_{ij_k}, \text{type}_i, \text{Word}1.
$$
The first parameter, Word1, is used by the CreateDataStructures routine to determine the number of elements in the set. The set involves index \( i \) having zip code \( \text{ZIP}_i \) which is of type \( \text{type}_i \) and has feasible POD assignments to \( j_1, j_2, \ldots, j_k \), at a cost of \( C_{i j_1}, C_{i j_2}, \ldots, C_{i j_k} \), respectively. The costs are in decreasing sorted order except possibly for the last record which, if the index is also a POD site, contains the operating cost for that site.

This type of data transfer is very efficient. Alternative methods of transferring large amounts of data from a FORTRAN to a Pascal program consumed nearly twice as much time. Further, all of the problem information for the SOLVER routines is contained in a single file. This includes travel costs, floor space rental costs, operating costs and fixed opening and closing costs. The latter two costs are included into the \( C_{ij} \) factors above before the data transfer is performed.

The Pascal input is performed in a pairwise form. Each pair consists of a two byte integer followed by an eight byte real number. The CreateDataStructures procedure reads 116 pairs at a time and processes the vector of information sequentially. The length of the vector is arbitrary. To ensure proper sequencing of the Pascal read statements with the DBL file, additional zero entries are inserted during the FORTRAN write statement.

Output from the SOLVER routines is stored in the STATE__.SOL file. Included in this output is the index of the zip code, its assigned POD, and a number indicating the color determined by the graph coloring algorithm for this zip code area. The STATE__.SOL file is used by the driver package to display the final solution.
BIBLIOGRAPHY


Geographic Data Technology, Inc., 13 Dartmouth College Highway, Lyme, NH.


APPENDIX: Program Listing

Program Solver;
{
This is the main driver program for the package that finds good
heuristic solutions to IRS Post-of-Duty (POD) location problem.
The program takes data from specially formatted data files, which
have been created by a separate pre-processing package. The final
solution is colored, and the result may be saved for graphic dis-
play on a map of the region in question.)

Const MaxZips = 800;
MaxPossible = 40;
{
Maximum number of zip-code areas allowed and the maximum number of
possible POD sites allowed. These numbers are somewhat flexible,
although 1000 may not be large enough for some districts and 75 is
probably more than we need for any district }

Type
Zcode = 0..MaxZips;
ZipSet = set of 1..Maxpossible;
{
!!NOTE!! : The "set" data type is an implementation-dependent type.
TURBO Pascal allows set types up to 255 distinct possible elements.
This means that in no case can MaxPossible be set to a value of more
than 255.)

Link = ^Neighbor; ( a pointer to a record of type Neighbor )

Neighbor = record
    Site, Target : Zcode;
    Cost : Real;
    NextZip, NextPOD : Link;
end;

(Each "Neighbor" record is one entry in the sparse matrix of
information relating zip codes to POD sites. The field SITE indicates
which zip-code area the information in the record applies to. The
field TARGET tells with reference to which POD site. COST gives the
cost of travel between SITE and TARGET (which will always be less than
the user-supplied upper limit on travel distance for any customer.
NEXTZIP is a pointer to the record which holds the next-nearest zip
code to TARGET (after SITE), if there is one. NEXTPOD is a pointer to
the record which holds the POD possible location which is the next-
nearest (after TARGET) to SITE, if one exists.)

PODsite = record
    Where : Zcode; ( Which site is this? )
    Must : boolean; ( Is it a required site? )
    Next : Link; ( a pointer to the first of its neighbors)
end;

SingleZip = record
ZipCode : real;
SType : integer;
end;

(actual zipcode number, a real var because integers in TURBO are not large enough. Site type:
SType=1 => not a POD site, ever.
SType=2 => can be a POD site.
SType=3 => is now a POD site.
SType=4 => must be a POD site.)

PairOfZips = record
    Number, PODnum: integer;          (site, target POD, edge)
    Cij: real;                         (type: Cij weights)
end;

ColumnPointArray = array[1..MaxPossible] of PODsite;
RowPointArray = array[1..MaxZips] of Link;
IndexArray = array[1..MaxZips] of Zcode;
ValueArray = array[1..MaxZips] of real;
FileString = string[14];

var

CurrentPODs, PossiblePODs : Zipset;

(CurrentPODs is the set of all POD's assigned in the current solution.
PossiblePODs is the set of all possible POD sites.)

BestPOD, NextBestPOD : IndexArray;

(BestPOD holds the zip which is the nearest POD in the current solution. NextBestPOD holds the second best current POD for each zip.)

CurrentCost, NextCost : ValueArray;

(CurrentCost[zip] holds the cost from zip to BestPOD[zip] in the current solution. Similarly, NextCost[zip] is the cost from zip to NextBestPOD[zip].)

CanBe : ColumnPointArray;

(CanBe is an array which allows us to find all the pertinent data concerning the Jth potential POD site: which site it is, which zips can be served from it, and how much that would cost. Its field NEXT points to a column of Neighbor records, along the NEXTZIP links.)

Index : Array[1..MaxZips] of Zcode;

(Index[i] tells which entry in CanBe refers to POD i)

Map : RowPointArray;

(Map is an array which lets us find, for any zip area, which POD sites
can serve it and how much that costs. Each entry of Map is a pointer to a row of Neighbor records, along the NEXTPOD links.)

ZCreal : ValueArray;

{Actual zip code number in real format}

RawDat : SingleZip;

{Will hold individual zip area data for the construction of the data structures.}

CurrentNumber, Switch, Nzips, Nposs, EndNumber : integer;

{CurrentNumber is the number of POD sites assigned in the current solution. Switch is 0 if graph coloring is to be used, 1 otherwise. Nzips is the number of zip code areas (<= MaxZips). NPOSS is the number of possible POD sites (<= MaxPossible). EndNumber is the number of POD sites the user has requested be in the final solution.}

TotalCost, Limit : real;

{TotalCost is the current objective function value. Limit is the user-supplied upper bound on travel distance.}

Error, Changes, Stuck : boolean;

{ERROR is a flag to warn that the solver has run into a situation where the user's wishes cannot be done; e.g., no feasible solution exists using only EndNumber POD sites. Changes indicates whether any swapping has been done in the interchange heuristic.}

ErrLoc : Zcode; {Site of ERROR if true}
MinCode : real; {Smallest zip-code in state}
StateNumber : string[2]; {Two-digit state code number}
StateNameFile : text;

-----------------------------------------------

function exist(fn:FileString):boolean;

{TURBO PASCAL FUNCTION returns true if file fn already exists}

var fil:file;

begin
assign(fil,fn);
($I-)$
reset(fil);
($I+)$
exist := (IOresult = 0);
end;

-----------------------------------------------
procedure Match;

( Given the contents of CurrentPODs and the arrays of neighbor data, this
procedure determines the nearest and next-nearest currently assigned POD for
each individual zip-code area, and the associated costs.)

var
    base : link;
    zip, pod : zcode;
    empty, done : boolean;
    ipod, izip : integer;

begin
    TotalCost := 0.0;
    error := false;
    for zip := 1 to Nzips do
      { find the first current POD in zip's list of possible POD's, and assign
        zip to it. }
      begin
        done := false;
        base := map[zip];
        if base=nil then
          done := true;
        while not done do
          if base=nil then { no POD is close enough, so this is illegal: }
            begin
              done := true;
              error := true;
              writeln('feasibility error at ',zip:5);
            end
          else
            begin
              pod := base^.target;
              ipod := Index[pod];
              if ipod in CurrentPODs then { pod is the best choice: }
                begin
                  done := true;
                  BestPOD[zip] := pod;
                  CurrentCost[zip] := base^.cost;
                  NextBestPOD[zip] := 0;
                  base := base^.nextpod;
                  empty := false;
                while not empty do { see if there's a next-best POD: }
                  if base=nil then { there isn't a next-best: }
                    empty := true;
                  else if Index[base^.target] in CurrentPODs then
                    begin
                      (this is next best)
                      NextBestPOD[zip] := base^.target;
                      NextCost[zip] := base^.cost;
                      empty := true;
                    end
                  else
                    base := base^.nextpod; { keep looking for a next-best }
                end
              end
            end
          end { if POD in CurrentPODs... }
end
else
  writeln(' pod not in CurrentPODS ',pod:5);
  base := base^.nextpod; { keep looking for a best POD }
end; (while not done...)
end; ( for zip := 1 to ...)
end; { Procedure Match }

{-----------------------------------------------}

procedure ComputeCost;
{ Just add up all CurrentCost values to find the new total cost. Recall that fixed costs are included in the i,j assignment costs }

var
  zip : Zcode;

begin
  TotalCost := 0.0;
  for zip := 1 to Nzips do
    TotalCost := TotalCost + CurrentCost[zip];
end;

{-----------------------------------------------}

procedure ListCurrent;
{ For larger problems, modify this to only print out POD sites }

var i:Zcode;

begin
  writeln(' Current zip-code assignments:');
  for i:= 1 to NZips do
    writeln(i:5,' at ',BestPOD[i],' : cost = ',CurrentCost[i]:3:2);

  ComputeCost;
  writeln([LST,]' Total cost of this allocation is ',TotalCost:12:2);
end;

{-----------------------------------------------}

procedure dumpstruct;
{ This is a diagnostic procedure which prints out the contents of the sparse matrix structure set up in procedure CreateDataStructures }

var zip,i : zcode;
  ptr : link;

begin


```python
for zip := 1 to Nzips do
begin
    ptr := Map[zip];
    while ptr<>nil do
    begin
        write(LST,ptr^.target:5);
        ptr := ptr^.nextpod;
    end;
    writeln(LST);
    writeln(LST);
end;

for zip := 1 to Nposs do
begin
    ptr := CanBe[zip].next;
    write(LST,CanBe[zip].where:5);
    while ptr<>nil do
    begin
        write(LST,ptr^.site:5);
        ptr := ptr^.nextzip;
    end;
    writeln(LST);
    writeln(LST);
end;
}
```

(* *************** MAIN PROGRAM *************** *)

```plaintext
($i init.pas )  ( Include array initializations )
($i dstruct.pas) ( Include data-structure initialization package )
($i greedy.pas ) ( Include greedy heuristic routines )
($i intchg.pas ) ( Include interchange routines )
($i fivclr.pas ) ( Include graph-coloring algorithm )
($i podclr.pas ) ( Include POD-coloring algorithm )
($i lgrn.pas )  ( Include Lagrangian Dual algorithm )
```

( Read the number of the state under consideration )

begin
    Assign(StateNameFile,'NAMES');
    reset(StateNameFile);
    read(StateNameFile,StateNumber);
    close(StateNameFile);

    Initialize;
    CreateDataStructures;
    writeln;
    write(' Enter the desired number of POD's in the final solution:');
    readln(EndNumber);
    Match;
    ClrScr;
    if error then
        writeln([lst],'Initial allocation is not feasible--program aborted.');
    else
        begin
```
writeln(' *************** INITIAL ASSIGNMENT *************** ');
ComputeCost;
ListCurrent;
writeln;
writeln(' *************** INITIAL INTERCHANGE *************** ');
writeln;
Interchange;
if not changes then
  writeln((LST,)’ No interchanges were necessary.’);
ComputeCost;
ListCurrent;

if EndNumber <> CurrentNumber then
begin
  writeln;
  writeln(' *************** GREEDY HEURISTIC *************** ');
  writeln;
  Greedy;
  if changes then
  begin
    if error then
      writeln(‘ Greedy heuristic solution is not feasible’)
    else
      begin
        writeln;
        writeln(' *************** FINAL INTERCHANGE *************** ');
        writeln;
        Interchange;
        if not changes then
          writeln((LST,)’ No interchanges were necessary.’);
        end;
      end;
  end;
end;
Match;
ComputeCost;
ListCurrent;

{perform the optional routine to find a lower bound on the optimal solution}
{ writeln(' start lagrangian dual ');
  lagrangian_dual((NextCost,NextBestPOD))(
    writeln(' end lagrangian dual ');
  )
}

if Switch=0 then
begin
  writeln;
  writeln(‘Calculating colors for solution map - Please wait’);
  GraphColor
end
else
  PODColor;
end;
end.
{---------------------------------------------------------------}
procedure initialize;

    (This procedure initializes various data arrays, pointers and sets
     used by the solver package.)

var
    i,j : integer;
begin
    for i:=1 to MaxZips do
        begin
            BestPOD[i] := 0;
            CurrentCost[i] := 0.0;
            NextBestPOD[i] := 0.0;
            NextCost[i] := 0.0;
            Index[i] := 0;
            Map[i] := nil;
        end;
    for i:=1 to MaxPossible do
        CanBe[i].next := nil;
    PossiblePODs := [];
    CurrentPODs := [];
end;

{---------------------------------------------}

procedure CreateDataStructures;

{ This procedure creates the sparse matrix structure which holds the
  information concerning which zip code area can be served from which POD
  sites, and at what cost. The data structure is a cross-linked array, with
  row links joining all PODs which can serve a given zip, and column links
  joining all zips which can be served by a given POD site. The entries are
  ordered along both row and column lists in order of increasing cost. }

type
    pair = record
        item1: integer;
        item2: real;
    end;
pair_vec = array[1..116] of pair;
pair_rec = record
        pair_vector: pair_vec;
    end;

var
    pairs                  : pair_rec;
    i,j,k,pointer         : integer;
pair_file              : file of pair_rec;
hold                   : Array[1..Maxpossible] of integer;
C_ij                   : Array[1..Maxpossible] of real;
filename               : FileString;
TempPair               : PairOfZips;
DoubleFile             : file of PairOfZips;
count,POD,t            : integer;
begin  ( M A I N  P R O C E D U R E )

count := 0;
Nposs := 0;
Nzips := 0;
mincode := 99999.0;
CurrentNumber := 0;
filename := 'STATE'+Statenumber+'.ADJ';
if exist(filename) then
  Switch := 0
else
  Switch := 1;
Assign(Pair_file, 'STATE'+StateNumber+'.DBL');
reset(pair_file);
read (pair_file,pairs);
pointer := 1;
while pairs.pair_vector[pointer].item1 <> 0 do
  begin
    k := pairs.pair_vector[pointer].item1;
    k := k div 10 - 2;
    for j := 1 to k do
      begin
        pointer := pointer+1;
        if pointer < 117 then
          begin
            hold[j] := pairs.pair_vector[pointer].item1;
            C_ij[j] := pairs.pair_vector[pointer].item2;
            end
        else
          begin
            pointer := 1;
            read( pair_file, pairs);
            hold[j] := pairs.pair_vector[pointer].item1;
            C_ij[j] := pairs.pair_vector[pointer].item2;
            end;
      end;
  end;
end.
end
else
begin
  \textit{T} := \texttt{pairs.pair\_vector[pointer+1].iteml};
  pointer := pointer+2;
end;

\begin{verbatim}
( start entering the i,j data )
( i,j entries are assumed to be in sorted order, except for i,i )

number:= \texttt{hold[1]};
count := \texttt{succ(count)};
ZCreal[number] := C_{ij}[l];
if C_{ij}[l] < mincode then mincode := C_{ij}[l];
if Nzips < number then Nzips := number;
if \texttt{T > 1} then \{ this is a possible POD site \}
begin
  if \texttt{Index[number]} = 0 then
    begin
      Nposs := \texttt{succ(Nposs)};
      Index[number] := Nposs;
      POD := Nposs
    end
  else
    POD := Index[number];

  CanBe[POD].where := number;
  if \texttt{T=4} then
    CanBe[POD].must := true
  else
    CanBe[POD].must := false;
  PossiblePODs := PossiblePODs + [POD];
case \texttt{T of}
  3,4 : begin
    CurrentPODs := CurrentPODs + [POD];
    CurrentNumber := \texttt{succ(CurrentNumber)};
  end;
end;

for j:=2 to k do
begin
  new(pt);
  with pt^ do
    begin
      site := number;
      target := \texttt{hold[j]};
      cost := C_{ij}[j];
      nextzip:= \texttt{nil};
      nextpod:= \texttt{nil};
    end;
  pl := Map[number];
  if pl = \texttt{nil} then
    Map[number] := pt
\end{verbatim}

32
else
    begin
    ( If pl^\text{\textasciicircum}.\text{cost} < C_{ij}[j] then
        writeln('scanning ', pl^\text{\textasciicircum}.\text{cost}, C_{ij}[j], ', index[number]);
        Map[number] := pt;
        pt^\text{\textasciicircum}.nextpod := pl;
    end; ( if pl=nil...else...)

if Index[hold[j]] = 0 then
    begin
        Nposs := succ(Nposs);
        Index[hold[j]] := Nposs;
        POD := Nposs
    end
else
    POD := Index[hold[j]];

pl := CanBe[POD].next;
if pl = nil then
    CanBe[POD].next:=pt
else
    begin
        p2:=pl^\text{\textasciicircum}.nextzip;
        if pl^\text{\textasciicircum}.site <> pl^\text{\textasciicircum}.target then
            begin
                CanBe[POD].next:=pt;
                pt^\text{\textasciicircum}.nextzip:=pl;
            end
        else
            begin
                pt^\text{\textasciicircum}.nextzip:=p2;
                pl^\text{\textasciicircum}.nextzip:=pt;
            end;
    end;
end;
close(pair_file);

( LST sends output to default list device (printer) )
( Remove braces to change from console to printer )

writeln((LST,)' ');
writeln((LST,)' Total number of zipcodes is ',count:5,'.'');
writeln((LST,)' Number of possible POD's is ',Nposs:5,'.'');
writeln((LST,)' Number of existing POD's is ',CurrentNumber:5,'.'');

( dumpstruct; ) ( diagnostic only--prints out entire data structure )
end;

-------------------------------------------------------------------------------

procedure InsertPOD ( new_POD: Zcode );

( this procedure performs the actual addition of pod site 'new_POD'
and updates all pertinent data in arrays such as CurrentCost[], BestPOD[], NextBestPOD[], NextCost[], and the set CurrentPODs.)

var
base                      : link;
opt, old_pod, zip         : Zcode;
new_cost                   : real;

begin
opt := Index[new.Pod];
base := CanBe[opt].next;
new_cost := base^.cost;      // include operating cost
CurrentPODs := CurrentPODs + [opt];

base := base^.nextzip;
while base <> nil do
    begin
        with base^ do
            begin
                zip := site;
                old.Pod := BestPOD[zip];
                if (cost < CurrentCost[zip]) and (old.Pod <> zip) then
                    begin
                        NextCost[zip] := CurrentCost[zip];
                        CurrentCost[zip] := cost;
                        NextBestPOD[zip] := old.Pod;
                        BestPOD[zip] := new.Pod;
                    end;
                else if (cost < NextCost[zip]) then
                    begin
                        NextCost[zip] := cost;
                        NextBestPOD[zip] := new.Pod;
                    end;
                end;  // with base^ do...
        base := base^.nextzip;
    end;
NextBestPOD[new.Pod] := BestPOD[new.Pod];
end;  // procedure InsertPOD

{---------------------------------------------------------------------------}

procedure RemovePOD( old.Pod: integer);
{ this procedure performs the actual removal of pod site 'old'
  and updates all pertinent data in arrays such as CurrentCost[],
  BestPOD[], NextBestPOD[], NextCost[], and the set CurrentPODs.)

var
base, base2 : link;
id, idold   : Zcode;
looking : boolean;

Begin
idold := Index[old_POD];
CurrentPODs := CurrentPODs - [idold];
base := CanBe[idold].next;
while base<>nil do
  begin
    with base^ do
      begin
        if NextBestPOD[site] = old_POD then  { NextBestPOD corrected here }
          begin
            base2 := map[site];
            looking := true;
            while looking do
              begin
                id := base2^.target;
                if (Index[id] in CurrentPODs) and (id <> BestPOD[site]) then
                  begin
                    looking := false;
                    NextBestPOD[site] := base2^.target;
                    NextCost[site] := base2^.cost;
                  end
                else if base2=nil then
                  begin
                    looking := false;
                    NextBestPOD[site] := 0;
                  end
                else
                  base2 := base2^.nextpod;
              end;
          end;
        if BestPOD[site] = old_POD then
          begin
            BestPOD[site] := NextBestPOD[site];
            CurrentCost[site] := NextCost[site];
            base2 := map[site];
            looking := true;
            while looking do
              begin
                id := base2^.target;
                if(Index[id] in CurrentPODs) and (id <> NextBestPOD[site]) then
                  begin
                    looking := false;
                    NextBestPOD[site] := base2^.target;
                    NextCost[site] := base2^.cost;
                  end
                else if base2=nil then
                  begin
                    looking := false;
                    NextBestPOD[site] := 0;
                  end
                else
                  base2 := base2^.nextpod;
              end;
          end;
      end;
  end;
procedure GreedyADD;

( Locates one new POD by greedy heuristic )

var
  Zip_Code : string[5];
  opt, zip, ind,
  new_POD, i, old_POD, z : Zcode;
  impr, addval : real;
  base : link;
  done : boolean;

begin
  ( Procedure GreedyADD )
  opt := 1;
  impr := 1E+38;
  for ind := opt to Nposs do  { find the index of the best choice }
    begin
      zip := CanBe[ind].where;
      if BestPOD[zip] <> zip then { zip isn’t a POD site at the moment }
        begin
          base := CanBe[ind].next;
          addval := base^.cost - CurrentCost[zip];  { Operating Cost }
          base := base^.nextzip;
          while base <> nil do
            begin
              z := site;
              if (cost < CurrentCost[z]) and (z <> BestPOD[z]) then
                addval := addval - currentcost[z] + cost;
              base := nextzip;
              end;
            end;
        end;
      if addval < impr then { this is the best choice so far }
        begin
          impr := addval;
          opt := zip;
        end;  { if BestPOD... }
    end;  { for ind ... }
  if impr < 1E+38 then  { add the new POD to the current set }
    begin
      InsertPOD(opt);
      str(ZCreal[opt]:5:0, Zip_Code);
      writeln((LST,' Adding ',Zip_Code,' to active set of PODs.'));
    end;  { procedure RemovePOD() }
else
begin
  stuck := true;
  writeln(' No POD can be added to the current allocation.' );
end;
end;  { procedure GreedyADD }

---------------------------------------------------------------------

procedure GreedyDEL;
{
  This procedure subtracts a POD from the currently assigned set
  according to the greedy heuristic.)
}

var
  Zip_Code : String[5];
  base : Link;
  tset : ZipSet;
  minval,change : Real;
  i, opt, ipod : Zcode;

begin
  opt := 0;
  minval:= 1E+32;
  for i := 1 to Nposs do
    begin
      if ( i in CurrentPODs ) and not ( Canbe[i].must ) then
        begin
          {ith site is a candidate for deletion }
          base := Canbe[i].next;
          ipod := base^.site;
          change := 0;
          while base <> nil do
            with base^ do
              if BestPOD[site] = ipod then
                if NextBestPOD[site] <> 0 then
                  begin
                    change := change + NextCost[site] - CurrentCost[site];
                    base := base^.nextzip;
                  end
                else
                  begin
                    base := nil;
                    change := 1E+32;
                  end
              else
                base := base^.nextzip;
          if ( change < minval ) then
            begin
              opt := ipod;
              minval := change;
            end;
        end;  { if i in CurrentPODs ... } 
    end;  { for .... } 
  if opt <> 0 then { we're not stuck: delete opt from CurrentPODs }
begin
  str(ZCreal[\text{opt}]:5:0,Zip_Code);
  writeln(' Deleting ',Zip_Code,' from active set of PODs');
  RemovePOD(opt);
end
else
  begin
    stuck := true;
    writeln(' No POD can legally be deleted from current allocation.' );
  end;
end;  \text{(procedure GreedyDEL)}

\begin{verbatim}
{procedure greedy;}

\begin{verbatim}
begin \text{( MAIN PROCEDURE )}
  writeln(' Entering Greedy heuristic...');
  stuck := false;
  changes := false;
  while (CurrentNumber $\not<$ EndNumber) and not stuck do
    if (CurrentNumber $\not<$ EndNumber) then
      begin
        GreedyADD;
        ComputeCost;
        ListCurrent;
        changes := true;
        if not stuck then
          CurrentNumber := succ( CurrentNumber );
      end
    else
      begin
        GreedyDEL;
        ComputeCost;
        ListCurrent;
        changes := true;
        if not stuck then
          CurrentNumber := pred( CurrentNumber );
      end;
    if not changes then
      writeln(' No changes in greedy heuristic. ');
  end; \text{( Procedure Greedy )}
\end{verbatim}

\begin{verbatim}
procedure interchange;

\begin{verbatim}
(This procedure performs the Teitz/Bart interchange heuristic for the
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}
Simple Plant Location Problem. Given an initial allocation of customers to service sites, the heuristic checks to see if it would be advantageous to exchange one currently assigned service site for one potential service site not currently assigned. The best such exchange is performed, and the heuristic repeats until no advantageous exchanges exist.

```pascal
var
  done : boolean;
  POD, TestOut, SwapIn, SwapOut, i : integer;
  mincost, val : real;
  Zip_Code_out, Zip_Code_in : string[5];

function SwapVal(old_POD, new_POD: integer): real;
  (This function computes the value of a potential site-exchange of site 'new' for site 'old'.)
const
  failure = 100.0;
var
  contr : real;
  illegal, looking : boolean;
  idold, idnew : Zcode;
  base, base2 : link;
begin
  contr := 0.0;
  illegal := false;
  idold := Index[old_POD];
  idnew := Index[new_POD];
  if CanBe[idold].must then
    illegal := true
  else
    begin
      base := canbe[idnew].next;
      contr := base^.cost - CurrentCost[new_POD];  // Operating Cost for new_POD
      base := base^.nextZip;
      while base <> nil do
        begin
          with base do
            if (cost < currentCost[site]) then
              begin
                if (site <> BestPOD[site]) and (old_POD <> BestPOD[site]) then
                  contr := contr + cost - CurrentCost[site];
                base := base^.nextZip;
              end;
            base := CanBe[idold].next;
          while base <> nil do
            begin
              with base do
                if BestPOD[site] = old_POD then
```
begin
  base2 := Map[site];
  if site = new_pod then
    looking := false
  else
    looking := true;
  while looking do
    begin
      looking := false;
      if base2=nil then
        illegal := true
      else if (Index[base2^.target] in CurrentPODs ) and
        (base2^.target <> old_POD) then
        contr := contr + base2^.cost - CurrentCost[site]
      else if (base2^.target = new_POD) then
        contr := contr + base2^.cost - CurrentCost[site]
      else
        begin
          looking := true;
          base2 := base2^.nextpod;
        end;
    end;
  end; (else clause from top)
  if illegal then
    Swapval := failure
  else
    Swapval := contr;
end; { function swapval() }
{

{ MAIN PROEDURE }
begin
  writeln(′ Entering interchange heuristic...′);
  changes := false;
  repeat
    done := true;
    for POD := 1 to nposs do
      begin
        if not (POD in CurrentPODs) then { POD is a candidate: }
          begin
            SwapIn := CanBe[POD].where;
            SwapOut := 0;
            mincost := 0.0;
            for i:= 1 to Nposs do
              begin
                if i in CurrentPODs then
                  begin
                    TestOut := canbe[i].where;
                  end;
                end;
              end;
            end;
          end;
        end;
    end;
end;

val := swapval(TestOut,SwapIn);
if val<mincost then ( this is the best swap so far )
begin
  mincost := val;
  SwapOut := TestOut;
end;
end;
end;
if mincost<0.0 then ( go ahead and make the best swap: )
begin
  str( ZCreal[SwapOut]:5:0, Zip_Code_out);
  str( ZCreal[SwapIn ]:5:0, Zip_Code_in );
  writeln((LST,)’ Swapping ’,Zip_Code_out,’ out’,
          Zip_Code_in,’ in.’);
  InsertPOD(SwapIn);
  RemovePOD(SwapOut);
  changes := true;
  done :=false;
end;
end;
end; until done;
{ writeln(’ Leaving interchange heuristic:’);  }
end; ( interchange heuristic procedure )

procedure PODColor;
( This procedure creates a solution file which does not use the
graph-coloring algorithm. )

var
  zip, clr, pod : integer;
  WriteFile : text;
  base : link;
begin
  Assign (WriteFile,’STATE’+StateNumber+’.SOL’);
  Rewrite(WriteFile);
  write (Writefile,totalcost:12:2);
  writeln(WriteFile);
  clr:=0;
  for pod:=-1 to Nzips do
    if Index[pod] <> 0 then
      begin
        base:=CanBe[Index[pod]].next;
        while base<>nil do
          begin
            zip:= base^.site;
            if BestPOD[zip]=pod then
              begin
                write (WriteFile,zip:5,ZcReal[zip]:6:0,clr:3,pod:5);
                writeln(WriteFile);
              end;
          end;
      end;
end;
procedure GraphColor;
{
This procedure computes the adjacency of POD service-regions in
the current solution to the POD location problem, and colors the
zips in these regions such that no two adjacent regions use the
same color. At most six (five?) colors will be used. For a good
description of the coloring algorithm, see David W. Matula et al,
"Graph Coloring Algorithms", in Ronald C. Read, "Graph Theory and
is based on the 'two-color chain' proof of the five-color theorem.
The solution may be saved to a file, if desired.
}

type
  ptr_type = ^adj_list_el;
  adj_list_el = record
    v : integer;
    next : ptr_type;
  end;
  graph_type = array [1..MaxPossible] of adj_list_el;
  Node_array = array [1..MaxPossible] of integer;
  set_type = set of 1..Maxpossible;

var
  graph : graph_type;
  { contains adjacency list representation of the graph }
  Node_num, color, ordering : Node_array;
  { color[vi] is the color assigned to vertex vi,
    ordering[] stores the order in which vertices should be
    colored. Node_num[i] tells which vertex in Graph corresponds to
    zip site number i }
  last_color, num_nodes, count, pod, spot : integer;
  { total # of colors used, number of nodes in graph
    count,pod,spot are temporary variables }
  answer : string[3];
  filename : FileString;
  WriteFile : text;
  GettingName : boolean;
  base : link;

(..........................................................)
procedure init_graph;
var
  nodes, count, w : integer;
begin
  for count := 1 to CurrentNumber do
    graph[count].next := NIL;
  nodes := 0;
  for count := 1 to Nposs do
    if count in CurrentPODs then
      begin
        nodes := succ(nodes);
        w := CanBe[count].where;
        graph[nodes].v := w;
        Node_num[w] := nodes;
      end;
end;  { init_graph }

procedure write_out_graph (var graph : graph_type; num_nodes : integer);
var
  count : integer;
  temp : ptr_type;
begin
  writeln; writeln;
  for count := 1 to num_nodes do begin
    write('adjaceny list for node ',count,' is : ');
    temp := graph[count].next;
    while(temp <> nil) do begin
      if temp <> nil then write(temp>v);
      temp := temp.next;
      if temp <> nil then write(',');
    end;  { while }
    writeln;
  end;  { for }
end;  { write_out_graph }

procedure read_in_graph;
var
  adjnum, zipnum, neighbor, count : integer;
  adj_file : text;

procedure Add_to_list(z,n:integer);
var
  pod1, pod2 : integer;
  ptr, p : ptr_type;
begin
  pod1 := Node_num[BestPOD[z]];
  pod2 := Node_num[BestPOD[n]];
  ptr := nil;
new(ptr);
ptr^.v := pod1;
ptr^.next := graph[pod2].next;
graph[pod2].next := ptr;
ptr := nil;
new(ptr);
ptr^.v := pod2;
ptr^.next := graph[pod1].next;
graph[pod1].next := ptr;

end; { Add_to_list }

begin
assign(adj_file,'STATE'+StateNumber+'.ADJ');
reset(adj_file);
while not EOF(adj_file) do
begin
read(adj_file,zipnum,adjnum);
readln(adj_file);
for count := 1 to adjnum do
begin
read(adj_file,neighbor);
if BestPOD[zipnum]=BestPOD[neighbor] then
Add_to_list(zipnum,neighbor);
end; { for }
readln(adj_file);
end; { while }
end; { read_in_graph }

procedure delete_node ( node : ptr_type; var list_ptr : ptr_type);
var
temp : ptr_type;
begin
temp := list_ptr;
if (node = list_ptr) then begin
 writeln(' error');
 list_ptr := node^.next;
 temp := node;
 dispose(temp);
end (if)
else begin
while (temp^.next <> node) do
 temp := temp^.next;
 temp^.next := node^.next;
 temp := node;
 dispose(temp);
end; { else }
end; { delete node }

--------------------------------------------------------------------------------
procedure clean_up (var graph : graph_type; num_nodes : integer);

{ eliminates duplications from the adjacency list of each vertex }

var
    node, temp_node : ptr_type;
    index, current : integer;
    adjacent : set_type;
begin
    for index := 1 to num_nodes do begin
        adjacent := [];
        node := graph[index].next;
        while (node <> nil) do begin
            current := node^.v;
            if (current IN adjacent) then begin
                temp_node := node;
                node := node^.next;
                delete_node(temp_node, graph[index].next)
            end; { if }
            else begin
                adjacent := adjacent + [current];
                node := node^.next;
            end; { else }
        end; { while }
    end; { for }
end; { clean_up }

-------------------------------------------------------------------

procedure find_min_degree(var vertex : integer; var graph : graph_type;
num_nodes : integer; var deleted : set_type);

var
    v_count, degree, min_degree : integer;
    temp : ptr_type;
begin
    min_degree := MaxPossible;
    for v_count := 1 to num_nodes do begin
        if not (v_count IN deleted) then begin
            temp := graph[v_count].next;
            degree := 0;
            while (temp <> NIL) do begin
                if not (temp^.v IN deleted) then begin
                    degree := degree + 1;
                    temp := temp^.next;
                end; { while }
                if (degree < min_degree) then begin
                    vertex := v_count;
                    min_degree := degree;
                end; { if }
            end; { if }
        end; { if }
    end; { find min degree }

-------------------------------------------------------------------
procedure order_graph (var graph : graph_type; num_nodes : integer;
    var ordering : Node_array);

var
deleled : set_type;
count,vertex : integer;
begin
    deleted := [];
    for count := num_nodes downto 1 do begin
        find_min_degree(vertex,graph,num_nodes,deleted);
        ordering[count] := vertex;
        deleted := deleted + [vertex];
    end;
end; { order_graph }

{---------------------------------------------------------------------}

procedure write_colors (var color : Node_array; last_color,num_nodes :
    integer);

var
count : integer;
begin
    writeln; writeln;
    for count := 1 to num_nodes do
        writeln(' 'POD',graph[count].v,' is colored in color '
            ',color[count]);
    writeln(' This coloring used ',last_color,' colors.');
end; { write_out_graph }

{---------------------------------------------------------------------}

procedure find_available( var color : Node_array; v_point : ptr_type;
    vertex : integer; var first_not_used : integer);

var
temp : ptr_type;
begin
    temp := v_point;
    first_not_used := 1;
    while (temp <> NIL) do
        if (color[temp^.v] = first_not_used)and((temp^.v <> vertex) then
            begin
                first_not_used := first_not_used + 1;
                temp := v_point;
            end { if } else temp := temp^.next;
        end; { find_available }

{---------------------------------------------------------------------}

procedure determine_colors_used(point : ptr_type;
    last_color : integer; var color,used : Node_array);

var
temp : ptr_type;
current : Integer;
begin
    for current := 1 to last_color do

used[current] := 0;
temp := point;
while (temp <> NIL) do begin
    current := color[temp^.v];
    if (used[current] = 0) then
        used[current] := temp^.v
    else if (used[current] > 0) then
        used[current] := -1;
    temp := temp^.next;
end; ( while )
{ writeln( 'determining the colors used out of ',last_color);
    for current := 1 to last_color do
        writeln(current,': ',used[current]);
} end; ( { determine_colors_used } )

procedure change_colors( var mark,color : Node_array;
    color1,color2,num_marked : integer);
var
    index,count1 : integer;
begin
    for count1 := 1 to num_marked-1 do begin
        index := mark[count1];
        if (color[index] = color1) then
            color[index] := color2
        else if (color[index] = color2) then
            color[index] := color1
        else writeln( 'ERROR in change_colors, ignore the results');
    end;
end; ( change_colors )

procedure try_swap (a,b : integer;
    var graph : graph_type;
    var color : Node_array;
    var success : boolean);
var
    visited,sub_colors : set_type;
    mark : Node_array;
    num_marked,current,color1,color2 : integer;

procedure find_component(var current : integer);
var
    temp : ptr_type;
begin
    { writeln( 'visited vertex ',current, ',',success); } 
    if (current = b) then success := false
    else begin
        visited := visited + [current];
        mark[num_marked] := current;
        num_marked := num_marked + 1;
        temp := graph[current].next;
while (temp <> NIL) and (success) do begin
    current := temp^.v;
    { writeln(' should we visit ',current,' ?'); }
    if (not (current IN visited)) and
    (color[current] IN sub_colors) then begin
        find_component(current);
        { writeln(' popped up'); }
    end;
    temp := temp^.next;
end; { while }
end; { else }
end; { find_component }

{---------------------------------------------------}

begin { try_swap }
    success := true;
    num_marked := 1;
    current := a;
    visited := [];
    color1 := color[a];
    color2 := color[b];
    sub_colors := [color1]+[color2];
    find_component(current);
    if (success) then change_colors(mark,color,color1,color2,num_marked);
end; { try_swap }

{---------------------------------------------------}

procedure try_interchange (var graph : graph_type; vertex : integer;
    var color : Node_array; var first,last_color : integer);
var
    used : Node_array;
    count1,count2,trial_color : integer;
    change_successful : boolean;
begin
    { writeln('trying interchange for #',vertex,' ..'); }
    determine_colors_used(graph[vertex].next,last_color,color,used);
    change_successful := false;
    count1 := 1;
    while (count1 <= last_color) and (not change_successful) do begin
        if (used[count1] > 0) then begin
            count2 := count1 + 1;
            while (count2 <= last_color) and (not change_successful) do begin
                if (used[count2] > 0) then begin
                    { writeln('considering ',count1,' ,',count2); }
                    trial_color := color[count1];
                    try_swap(used[count1],used[count2],graph,color,
                        change_successful);
                    if (change_successful) then first := trial_color;
                end; { if .. }
                count2 := count2 + 1;
            end; { while count2 ... }
        end; { if .. }
    end; { while count1 ... }

48
end;  { if .. count1 }
count1 := count1 + 1;
end;  { while }
end;  { try_interchange }

procedure color_graph (var graph : graph_type; num_nodes : integer;
                        var color : Node_array; var last_color : integer);
var
    vertex, first_not_used, counter : integer;
begin
    for counter := 1 to num_nodes do
        color[counter] := 0;
    last_color := 1;
    for counter := 1 to num_nodes do begin
        vertex := ordering[counter];
        writeln( ' now coloring vertex number ',vertex);
        find_available(color,graph[vertex].next,vertex,first_not_used);
        if (first_not_used > last_color) then
            try_interchange(graph,vertex,color,first_not_used,last_color);
        color[vertex] := first_not_used;
        if (first_not_used > last_color) then
            last_color := last_color + 1;
    end;  { for }
end;  { color_graph }

--- MAIN PROGRAM ---

begin { main }
  init_graph;
  num_nodes := CurrentNumber;  { CurrentNumber is a global variable
telling how many PODs are assigned }
  read_in_graph;
  clean_up(graph,num_nodes);
  { write_out_graph(graph,num_nodes);  }  { diagnostic only }
  order_graph(graph,num_nodes,ordering);
  color_graph(graph,num_nodes,color,last_color);
  { write_colors(color,last_color,num_nodes); }
  writeln( ' Do you wish to save this solution and coloring on the disk' );
  writeln( ' for later graphic output ? (Yes or No)' );
  readln(answer);  
  answer[1] := 'y';
  if (answer[1] = 'y') or (answer[1] = 'Y') then begin
    writeln( ' Enter the filename under which you wish to save the data:' );
    readln(filename);  
    filename := 'STATE'+StateNumber+' .SOL';
    GettingName := exist(filename);  
    while GettingName do
      begin
        writeln( ' NOTE: file ',filename, ' already exists:' );
        writeln( ' Write over this file ? ' );
        readln(answer);
    end;  { while }
end;  { if (answer[1] = 'y') or (answer[1] = 'Y') }

}
if (answer[1]<>'y') and (answer[1]<>'Y') then begin
  writeln(' Enter new filename:');
  readln(filename);
  GettingName := exist(filename);
end else
  GettingName := false;
end; ( while )
Assign (WriteFile,filename);
Rewrite(WriteFile);
write (Writefile,totalcost:12:2);
writeln(Writefile);
for pod:=1 to Nzips do
  if Index[pod] <> 0 then begin
    base:=CanBe[Index[pod]].next;
    while base=nil do begin
      count:= base^.site;
      if BestPOD[count]=pod then begin
        spot := Node_num[pod];
      end;
      write(Writefile,count:5,ZcReal[count]:6:0,Color[spot]:3,pod:5);
      writeln(WriteFile);
      end;
      base:=base^.nextzip;
    end;
  end;
Close(Writefile);
{ writeln(' Data have been saved in file ',filename);} end;
end; { GraphColor }

Procedure Lagrangian_dual;((var Dual_var:ValueArray;mono_Xij:Indexarray));
{ This procedure attempts to locate a lower bound on the optimal allocation. }
Type
  RealArray = array[1..Maxpossible] of real;
  IntegerArray = array[1..Maxpossible] of integer;
Const
  w_eps = 0.001;
var
  Hold_POD_set : zipset;
  Running_Average : Array[1..5] of real;
  red_cost,work : RealArray;
  pod_id : IntegerArray;
  Dual_var : ValueArray;
Sum_Xij : ValueArray; (Indexarray;
mono_Xij : Indexarray;
multiplier, mean, delta_s, delta_u, scale_factor, norm_factor, min_dif,
w_target, w_previous, w_new, w, s : real;
Sum_Xj, mono, monotone, nits, iter,
pod, ipod, zip, i, tick : integer;
error, Existing : boolean;
base : link;
dual_file : file of real;
filename : FileString;
Out_file : Text;

procedure Assignment;
{Given the contents of CurrentPODs and the arrays of neighbor data,
this procedure determines the nearest currently assigned POD for
each individual zip-code area, and the associated costs. }

var
base : link;
zip, pod : zcode;
empty, done : boolean;
ipod, zip : integer;

begin
error := false;
TotalCost := 0.0;
for zip := 1 to Nzips do { find the first current POD in zip's list of
possible POD's, and assign zip to it. }

begin
done := false;
base := map[zip];
if base-nil then
  done := true;
while not done do
  if base-nil then { this zipcode will be skipped }
  begin
    done := true;
    error := true;
    writeln(' Feasibility error at ', zip:5);
  end
else
  begin
    pod := base^target;
ipod := Index[pod];
    if ipod in CurrentPODs then { pod is the best choice: }
    begin
      done := true;
      BestPOD[zip] := pod;
      CurrentCost[zip] := base^.cost;
    end { if POD in CurrentPODs...}
else
  base := base^.nextpod; { keep looking for a best POD }

end
}
Procedure QuickSort(Var value: RealArray; Var index: IntegerArray; N: Integer);

Procedure Exchange(I, J: Integer);
    { Change records I and J }

var
temp: real;
indx: integer;

Begin
    temp := value[I];
    indx := index[I];
    value[I] := value[J];
    index[I] := index[J];
    value[J] := temp;
    index[J] := indx;
End;

Const
    MaxStack = 20; { Log2(N) = MaxStack, i. e. for MaxStack = 20 it is possible to sort 1 million records }

Var
    { The stacks }
    LStack: Array[1..MaxStack] Of Integer; { Stack of left index }
    RStack: Array[1..MaxStack] Of Integer; { Stack of right index }
    Sp: Integer; { Stack SortPointer }
    M, L, R, I, J: Integer;
    X: Real;

Begin
    { The quicksort algorithm }
    If N > 0 Then
        Begin
            LStack[1] := 1;
            RStack[1] := N;
            Sp := 1
        End
    Else
        Sp := 0;
    While Sp > 0 do
        Begin
            { Pop(L,R) }
            L := LStack[Sp];
            R := RStack[Sp];
            Sp := Sp - 1;
            Repeat
                I := L; J := R;
                M := (I + J) shr 1;
                X := Value[M];
                (writeln('l r m x ', I:5, r:5, m:5, x));
            End
        End
End;
Repeat
  while (I<=J) and (Value[I] < x) do
    I:=I+1;
  while (I<=J) and (Value[J] > x) do
    J:=J-1;
  If I<=J Then
    begin
    (writeln('i j v[i] v[j] ',i:5,j:5,value[i],value[j]);)
      If i>j then Exchange(I,J);
      i:=i+1;
      j:=j-1;
    end
  Until I>J;
( Push longest interval on stack )
  If J-L < R-I Then
  Begin
    If I<R Then
      Begin
        ( Push(I,R) )
        Sp:=Sp+1;
        LStack[Sp]:=I;
        RStack[Sp]:=R;
        writeln('sp i r ',sp:5,i:5,r:5);
      End;
      R:=I-1
    End
  Else
    Begin
      If L<J Then
        Begin
          ( Push(L,J) )
          Sp:=Sp+1;
          LStack[Sp]:=L;
          RStack[Sp]:=J;
          writeln('sp l j ',sp:5,l:5,j:5);
        End;
        L:=J+1
      End;
      Until I>R
    End;
  End ( QuickSort );

(***********************************************************************)

begin
  ( Initialize Lagrangian Solution )
    tick := 0;
    nits := 50;
    Error := false;
    scale_factor := 1;
    w_previous := 0;
    w_new := 2*w_eps;
    w_target := totalCost+10;
    hold_POD_set := CurrentPODs;
Compute a dual value for the number of open POD sites

\[ s = \text{totalcost}; \]
GreedyADD;
ComputeCost;
\[ s = \text{totalcost} - s; \]
CurrentPODs := hold_POD_set;
match;
\[ s = -1035; \]

Perform File Initialization

\[
\text{assign}(\text{dual\_file}, \text{dual}.\text{var});
\text{reset}\ (\text{dual\_file});
\text{assign}(\text{out\_file}, \text{out}.\text{fil});
\text{rewrite}(\text{out\_file});
\text{writeln}(\text{Out\_file}, \text{scale\_factor}, s);
\]

Initialize dual variables using best and nextbest costs

\[
\text{for } i = 1 \text{ to } \text{nzips} \text{ do }
\text{if } \text{map}[i] \text{nil then begin }
\text{determine an interval for the dual variable}
\text{if nextbestPOD}[i] = 0 \text{ then }
\text{nextcost}[i] := 2 \times \text{currentcost}[i]
\text{else if } \text{currentcost}[i] > \text{nextcost}[i] \text{ then }
\text{currentcost}[i] := \text{nextcost}[i]/2;
\text{Estimate an initial dual value}
\text{w} := \text{CurrentCost}[i]/\text{nextcost}[i];
\text{nextcost}[i] := \text{nextcost}[i] - \text{currentcost}[i];
\text{if } \text{nextcost}[i] > \text{abs}(s) \text{ then } \text{nextcost}[i] := \text{abs}(s);
\text{dual\_var}[i] := \text{CurrentCost}[i] + w \times \text{NextCost}[i];
\text{mono\_Xij}[i] := 0;
\text{end}
\text{else }
\text{dual\_var}[i] := 0.0;
\]

\[
\text{read}(\text{dual\_file}, \text{scale\_factor}, s);\]

\[
\text{for } i = 1 \text{ to } \text{nzips} \text{ do }
\text{begin }
\text{(read}(\text{dual\_file}, \text{dual\_var}[i]));
\text{(if } \text{index}[i] < 0 \text{ then)}
\text{writeln}(\text{Out\_file}, ', \text{dual\_var}[i]:7.1', ', \text{NextCost}[i]:7.1', ', \text{CurrentCost}[i]:7.1', ', \text{index}[i]:3', ', i:4, \text{BestPOD}[i]:5);\text{end;}
\text{for } i = 1 \text{ to } 5 \text{ do }
\text{Running\_average}[i] := 1.0;
\]

\[
\text{Begin the main loop}
\]

while abs(\text{w\_previous} - \text{w\_new}) > \text{w\_eps} \text{ do }
\text{begin }
\text{clrscr;}
\text{monotone} := 0;
\text{for iter := 1 to } \text{nits} \text{ do }
\text{end;}
\]

54
begin

{ Compute Reduced Costs }

tick := tick + 1;
if iter = 1 then
for i:=1 to Nposs do
begin
{ Compute the reduced costs from scratch }
    base := CanBe[i].next;
ipod := base^.site;
work[i] := base^.cost-dual_var[ipod]-s;
base := base^.nextzip;
while base <> nil do
    with base do
        begin
            zip := site;
            if cost-dual_var[zip] < 0 then
                work[i] := work[i] + cost-dual_var[zip];
            base := base^.nextzip;
        end;
end;
else
{ Compute the same thing only faster }
begin
    for i:=1 to nposs do
        work[i] := work[i] + delta_s*(Sum_Xj[j] - Endnumber);
    for i:=1 to nzips do
        if Sum_Xij[i] <> 0 then
            begin
                dual_var[i] := dual_var[i] + Sum_Xij[i];
                base := map[i];
                while base <> nil do
                    with base do
                        begin
                            ipod := Index[target];
                            if site = target then
                                work[ipod] := work[ipod] - Sum_Xij[i]
                            else if cost-dual_var[i] < 0 then
                                begin
                                    if cost-dual_var[i] + Sum_Xij[i] < 0 then
                                        work[ipod] := work[ipod] - Sum_Xij[i]
                                    else
                                        work[ipod] := work[ipod] + cost -
                                            dual_var[i];
                                end
                            else if cost-dual_var[i] + Sum_Xij[i] < 0 then
                                work[ipod] := work[ipod] - cost + dual_var[i]
                                           - Sum_Xij[i];
                            base := nextpod;
                        end;
            end;
end;

{ find the best k POD set }

for i:=1 to Nposs do
begin
pod_id[i]:=i;
red_cost[i]:=work[i];
end;
Quicksort(red_cost,pod_id,nposs);

find a feasible solution
Sum_Xj[j]:=0;
w_previous := w_new;
w_new := s*EndNumber;
CurrentPODs:=[i];
for i:= 1 to nposs do
  if (red_cost[i] < 0) then
    begin
      Sum_Xj[j]:=Sum_Xj[j] + 1;
      w_new:=w_new+red_cost[i];
      CurrentPODs:=CurrentPODs+:pod_id[i];
    end;

Compute the Xij's and Objective Value
for i:=1 to Nzips do
  if map[i]=nil then
    Sum_X[i] := 0
  else
    Sum_X[i] := -1;

for i:=1 to Nposs do
  begin
    base := CanBe[i].next;
    pod := base^.site;
    ipod := index[pod];
    if ipod in currentPODs then
      while base <> nil do
        with base^ do
          begin
            if (site=pod) or (cost-dual_var[site]<0) then
              Sum_X[i][site] := Sum_X[i][site]+1;
              base:=nextzip;
            end;
      end;
  end;
norm_factor:=0.0;
for i:=1 to nzips do
  begin
    w_new:=w_new+dual_var[i];
    norm_factor:=norm_factor+Sum_X[i][i]*Sum_X[i][i];
    norm_factor:=norm_factor+abs(Sum_X[i][i]);
  end;

if norm_factor = 0 then norm_factor:=0.9;
Running_average[iter mod 3 + 1] := norm_factor;

56
mean :=

( compute a new scale factor and compute new dual variables )

if \( w_{\text{new}} - w_{\text{previous}} < -w_{\text{eps}} \) then
    monotone:=0
else
    monotone:=monotone+1;
if (monotone >= 5) and (scale_factor < 0.5) then
    begin
        scale_factor := 2.0*scale_factor;
        writeln(LST,'2*scale_factor',scale_factor:7:6);
        monotone:=0;
    end;
multiplier:= scale_factor*(\( w_{\text{target}} - w_{\text{new}} \))/mean + w_{\text{eps}};
{
    compute the min_dif needed to change Sum_Xij by 1
}

min_dif:=1E+38;
for i:=1 to Nzips do
    begin
        if Sum_Xij[i] < 0 then
            begin
                if mono_Xij[i] > 0 then
                    mono_Xij[i]:=0
                else
                    mono_Xij[i]:=mono_Xij[i]-1;
                base := map[i];
                while base <> nil do
                    with base^ do
                    begin
                        pod:= target;
                        if Index[pod] in currentPODs then
                            if(cost-dual_var[i] > 0) then
                                if (cost-dual_var[i]<min_dif) then
                                    min_dif:=cost-dual_var[i];
                                base:=nextpod;
                            end;
                        end
                else if Sum_Xij[i] > 0 then
                    begin
                        if mono_Xij[i] < 0 then
                            mono_Xij[i]:=0
                        else
                            mono_Xij[i]:=mono_Xij[i]+1;
                        base := map[i];
                        while base <> nil do
                            with base^ do
                            begin
                                pod:= target;
                                if Index[pod] in currentPODs then
                                    if(cost-dual_var[i] < 0) then

57
if (dual_var[i] - cost < min_dif) then
    min_dif := dual_var[i] - cost;
    base := nextpod;
end;
else
    mono_Xi[j][i] := 0;
end;

multiplier := scale_factor * (w_target - w_new) / mean + w_eps;
if (min_dif < 1E+38) and (min_dif > multiplier) then
    begin
        multiplier := min_dif;
        writeln(LST, ' min_dif applied ', min_dif:10:5);
    end;
mean := w_new / w_target;
if (tick mod 5 = 0) then
    writeln(LST, tick:3, ', w_target:12, ', w_new:12, ', norm_factor:9,
        ', multiplier:9, ', mean:12, ', s:6:1, ', Sum_Xj:3);
GotoXY(1,1);
writeln(' Upgraded lower bound on the optimal solution value is',
        ' w_new:7:0,', ', iter:3);
writeln(' iter w_target w_new norm_fact multiplier',
        ' min_dif s Sum_Xj ');
writeln(tick:3, ', w_target:12, ', w_new:12, ', norm_factor:9,
        ', multiplier:9, ', min_dif:9, ', s:6:1, ', Sum_Xj:3);

for i:=1 to nzips do
    begin
        if abs(mono_Xi[j][i]) >= 5 then
            begin
                Sum_Xi[j][i] := Sum_Xi[j][i] * 2;
                writeln(' 2*Sum_Xi on ', i:4);
            end;
        if min_dif = multiplier then
            Sum_Xi[j][i] := multiplier * Sum_Xi[j][i]
        elseif multiplier < nextcost[i] then
            Sum_Xi[j][i] := multiplier * Sum_Xi[j][i]
        else
            Sum_Xi[j][i] := Sum_Xi[j][i] * scale_factor * Nextcost[i];
    end;

if multiplier > 10 then
    delta_s := scale_factor
else if (Sum_Xj[EndNumber] > 0) and (multiplier < red_cost[Sum_Xj])
    then
        delta_s := abs(red_cost[Sum_Xj])
else if (Sum_Xj[EndNumber] < 0) and (multiplier < red_cost[Sum_Xj+1])
    then
        delta_s := abs(red_cost[Sum_Xj+1])
else
    delta_s := multiplier;
s := s - delta_s * (Sum_Xj[EndNumber]);
writeln('i pod sumXij red_cost dual_var ');
for i:=1 to npos do
begin
    ipod:=pod_id[i];
    pod :=canbe[ipod].where;
    writeln(i:5,' ',ipod:5,' ',sum_Xij[pod]:5,' ',
    red_cost[i]:5:0,' ',dual_var[ipod]:5:0);
end;
Assignment;
if error then
begin
    writeln(' current lagrangian solution not feasible ');
end
else
begin
    listcurrent;
    if (totalCost < w_target) and (Sum_Xjj = EndNumber) then
begin
    mean:=(w_target-w_new)/(totalCost-w_new);
    if mean > 2 then
begin
    writeln(lst,' scale_factor adjusted',
    scale_factor:5:4);
    scale_factor:=scale_factor*mean;
    if scale_factor > 1 then scale_factor:=1;
end;
    w_target:=totalCost;
    hold_POD_set:=current?ODs;
    writeln('upgraded w_target',w_target);
end;
iter:=nits shr 1;
nits:=iter+10;
scale_factor:=scale_factor/2.0+w_eps;
{ writeln(' enter nits scalefactor',nits:6, ',scale_factor:8);
  readln(nits,scale_factor);
  writeln('w_new ',w_new,' norm_factor ',norm_factor);
  writeln(' nits ',nits, ' scale_factor ',scale_factor);
  reset(dual_file);
  write(dual_file,scale_factor,s);
  for i=-1 to nzips do
    write(dual_file,dual_var[i]);
  end;
  close(dual_file);
  close(out_file);
end;
1. **Publication or Report No.**
   
   NBSIR 86-3472

2. **Performing Organ. Report No.**

3. **Publication Date**
   
   FEBRUARY 1987

4. **Title and Subtitle**
   
   The Internal Revenue Service Post-of-Duty Location Modeling System - Programmer's Manual for Pascal SOLVER

5. **Author(s)**
   
   Paul D. Domich, Richard H. F. Jackson, Marjorie A. McClain, David M. Tate

6. **Performing Organization (If joint or other than NBS, see instructions)**
   
   NATIONAL BUREAU OF STANDARDS
   DEPARTMENT OF COMMERCE
   WASHINGTON, D.C. 20234

7. **Contract/Grant No.**

8. **Type of Report & Period Covered**

9. **Sponsoring Organization Name and Complete Address (Street, City, State, ZIP)**
   
   The Research Division
   U. S. Internal Revenue Service
   1201 E Street
   Washington, DC 20224

10. **Supplementary Notes**

   - Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. **Abstract**

    This report documents a project undertaken by the National Bureau of Standards to develop a mathematical model which identifies optimal locations of Internal Revenue Service Posts-of-Duty. The mathematical model used for this problem is the uncapacitated, fixed charge, facility location model which minimizes travel and facility costs, given a specified level of activity. This report discusses the mathematical techniques used to solve the mathematical model developed and includes a Greedy procedure, an Interchange procedure, and a Lagrangian Relaxation of the related linear program. A description of the Pascal routines, definitions of key data structures and variables is provided. Data sources identified and used are also described.

12. **Key Words**

    uncapacitated fixed charge facility location problem, Greedy heuristic, Interchange heuristic, Lagrangian Relaxation

13. **Availability**

    - Unlimited
    - For Official Distribution. Do Not Release to NTIS
    - Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. **No. of Printed Pages**

    62

15. **Price**

    $13.95