NISTIR 86-3472-1

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 Institute of Standards and Technology
(Formerly National Bureau of Standards)
Center for Computing and Applied Mathematics
Gaithersburg, MD 20899

February 1989

A Report to:
The Research Division
Internal Revenue Service
Washington, DC 20224

NISTIR 86-3472-1

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 Institute of Standards and Technology
(Formerly National Bureau of Standards)
Center for Computing and Applied Mathematics
Gaithersburg, MD 20899

February 1989

National Bureau of Standards became the National Institute of Standards and Technology on August 23, 1988, when the Omnibus Trade and Competitiveness Act was signed. NIST retains all NBS functions. Its new programs will encourage improved use of technology by U.S. industry.

A Report to:
The Research Division
Internal Revenue Service
Washington, DC 20224

U.S. DEPARTMENT OF COMMERCE
C. William Verity, Secretary
Ernest Ambler, Acting Under Secretary for Technology
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
Raymond G. Kammer, Acting Director

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) <u>The Internal Revenue Service Post-of-Duty Location Modeling System: Final Report.</u>

   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: Paul D. Domich).

2) <u>The Internal Revenue Service Post-of-Duty Location Modeling System: User's Manual.</u>

   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) <u>The Internal Revenue Service Post-of-Duty Location Modeling System: Programmer's Manual for FORTRAN Driver.</u>

   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) <u>The Internal Revenue Service Post-of-Duty Location Modeling System: Programmer's Manual for PASCAL Solver.</u>

   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. The initial POD file should reflect the current POD configuration and accurately specify opening and closing costs. Opening or closing costs for POD sites not specified in the POD data file can be 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 calculations.

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) cost(A - {a} + {b}) is less than 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 Version 5.0 (Borland International Inc., (1988)). 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 2721 lines of code in the SOLVER program compile and link in under 9 seconds, to a file of only 45K

bytes. A similar FORTRAN code requires over 6 minutes to compile and link and has a much larger storage requirement.

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 preprocessing of input data and graphically displays workload data and SOLVER's final solution. As input data the SOLVER routine requires a single file (called TRANSER.xx) 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 possible POD's is nnn.
Number of current and fixed POD's is mmm.

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

where nnn, mmm, and kkk are integer values. After the last prompt has been answered, the solver proceeds to solve the POD location problem. A summary of the problem characteristics is provided with the initial and improved solution values. 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,
FIVCLR.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 and contains general utility routines, denoted as InsertPOD and DeletePOD, that add and delete POD's from the data structure, re-establishing the data structure for the new set of POD's. INTCHG is the interchange heuristic algorithm. 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:

<u>FUNCTIONS:</u>

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.

<u>PROCEDURES:</u>

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 TRANSFER.xx 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 = 2000;	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.
MaxRead = 64;	Maximum number of records read from the TRANSFER.xx file during the BLOCK READ.

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		Record type which includes:
	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		Record type which includes:
	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		Data record type which includes:
	zip code	actual zip code number,
	SType	site type; SType=0 => never a POD site, SType=1 => can be a POD site, SType=2 => must be a POD site,
	FixCost	Opening/Closing cost for a POD site.
PairOfZips		Data record type which includes:
	Number	zip code index of zip code area,
	PODnum	zip code index of POD site,
	Cij	cost of area to site assignment.
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 ColumnpointArray 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.
CurrentNumber	The current number of POD sites.
EndNumber	The desired number of POD sites in the final solution.
Nzips	The total number of zip code areas.
Nposs	The total number of possible POD sites.
Switch	0 if graph-coloring is used, and 1 otherwise.
TotalCost	The current objective function value.
Error	A flag to warn that the solver has run into a situation where the user's wishes cannot be satisfied; e.g. no feasible solution exists using only EndNumber POD sites.
Change	Flag indicating whether any swapping was performed by the interchange heuristic.
ErrLoc	The site of ERROR if true.
MinCode	The smallest zip code number in the state.
StateNumber	The two-digit state code number.
StateNameFile	The name of the state.

6. Input/Output Processing:

Input to the SOLVER routines comes from the TRANSFER.xx file where xx refers the index of the tax district (1 to 76). The TRANSFER 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:

$$\text{Word1}, i, ZIP_i, j_1, C_{ij_1}, j_2, C_{ij_2}, \ldots, j_k, C_{ij_k}, type_i, \text{Word1}.$$

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 ZIP_i which is of type $type_i$ and has feasible POD assignments to $j_1, j_2, \ldots, j_k$, at a cost of $C_{ij_1}, C_{ij_2}, \ldots, C_{ij_k}$, 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 transfering 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 MaxRead 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 TRANSFER file, additional zero entries are inserted during the FORTRAN write statement.

Output from the SOLVER routines is stored in the SOLUTION.xx 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 SOLUTION.xx file is used by the driver package to display the final solution.

BIBLIOGRAPHY

Borland International Inc., TurboPascal version 3.0, Reference Manual, Scotts Valley Drive, Scotts Valley, CA., 1985.

Bradeau, M.L., and Chiu, S.S., "Sequential Location and Allocation: Characterization and Estimation of Globally Optimal Solutions", Department of Engineering-Economic Systems, working paper, Stanford University, 1984.

Cornuejols, G., Fisher, M.L., and Nemhauser, G.L., "Location of Bank Accounts to Optimize Float: An Analytical Study of Exact and Approximate Algorithms", Management Science, vol. 23, 789-810, 1977.

Cornuejols, G., Nemhauser, G.L., and Wolsey, L.A., "A Canonical Representation of Simple Plant Location Problems and its Applications", SIAM J. Alg. Disc. Math., vol. 1, no. 3, 1980.

Cornuejols, G., and Thizy, J.M., "A Primal Approach to the Simple Plant Location Problem", SIAM J. Alg. Disc. Math., vol. 3, no. 4, 1982.

Domich, P.D., Hoffman, K.L., Jackson, R.H.F., and McClain, M.A., "The Internal Revenue Service Post-of Duty Location Modeling System: Final Report", National Bureau of Standards Technical Report NBSIR 86-3482, Gaithersburg, MD, July, 1986a.

Domich, P.D., Jackson, R.H.F., and McClain, M.A., "The Internal Revenue Service Post-of Duty Location Modeling System: Programmer's Manual for the FORTRAN Driver", National Bureau of Standards Technical Report NBSIR 86-3473, Gaithersburg, MD, July, 1986b.

Domich, P.D., Jackson, R.H.F., and McClain, M.A., "The Internal Revenue Service Post-of Duty Location Modeling System: User's Manual", National Bureau of Standards Technical Report NBSIR 86-3471, Gaithersburg, MD, July, 1986c.

Erlenkotter, D., "A Dual-Based Procedure for Uncapacitated Facility Location", Operations Research, vol. 26, no. 6, 992-1009, 1978.

Fisher, M.L., "The Lagrangian Relaxation Method for Solving Integer Programming Problems", Management Science, vol. 27, no. 1, 1982.

Francis, R.L., and White, J.A., Facility Layout and Location, Prentice-Hall, Englewood Cliffs, NJ, 1974.

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

Garfinkel, R.S. and Nemhauser, G.L., Integer Programming, John Wiley & Sons, New York, NY, 1972.

Hoffman, A.J., Kolen, A.W.J., and Sakarovitch, M., "Totally-Balanced and Greedy Matrices", SIAM J. Alg. Disc. Math., vol. 6, no. 4, 1985.

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

Kolen, A., "Solving Covering Problems and the Uncapacitated Plant Location Problem on Trees", European J. of Operational Research, vol. 12, 1983.

Morris, J.G., "On the Extent to which Certain Fixed-Charge Depot Location Problems Can Be Solved by LP", Journal of the Operational Research Society, vol. 29, no. 1, 71-6, 1978.

Matula, D.W., Marble, G., and Isaacson, J.D., "Graph Coloring Algorithms," in R.C. Read, Graph Theory and Computing, Academic Press, New York, NY, 1972.

Ryan-McFarland Corporation, IBM Personal Computer Professional FORTRAN: Reference, for IBM Corp. Boca Raton, Florida, 1984.

Tamir, A., "A Class of Balanced Matrices Arising from Location Problems", SIAM J. Alg. Disc. Math., vol. 4, no. 3, 1983.

Teitz, M.B. and Bart, P., "Heuristic Methods for Estimating the Generalized Vertex Median of a Weighted Graph," Operations Research, 16, 955-61, 1968.

Wagner, H.M., Principles of Managment Science, Prentice-Hall, Englewood Cliffs, NJ, 1975.

APPENDIX: Program Listing

```
{$D-}     {Debugging on}
{$R+}     {Range checking on}
{$A+}     {byte alignment}
{$B+}     {Boolean complete evaluation on}
{$S+}     {Stack checking on}
{$I+}     {I/O checking on}
{$N+}     {Numeric coprocessor present}
{$E-}     {8087 emulation off}
{$M 3500,0,655360} {Modified stack and heap sizes}

Program Solver;
{    Version TURBO Pascal 5.0, stack and heap sizes for General Problem.}

{    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.}


Uses
  Crt;

Const
      MaxZips = 2000;     { maximum number of zip-code areas allowed
                            maximum = 1550 if valuearray of type real
                            maximum = 2050 if valuearray of type single }
      MaxPossible = 85;   { 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                                     }
      MaxRead=64;         { BlockRead Parameter for CreateDataStructures }

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                              { Each "Neighbor" record is one
}
                 Site, Target       : Zcode;   { entry in the sparse matrix of
}
                 Cost               : single;  { information relating zip
codes}
```

```
                 NextZip, NextPOD : Link;    { to POD sites.  The field SITE
}
               end;                          { 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;

    ColumnPointArray = array[1..MaxPossible] of PODsite;
    RowPointArray    = array[1..MaxZips] of Link;
    IndexArray       = array[1..MaxZips] of Zcode;
    IntegerArray     = array[1..MaxZips] of shortint;
    ValueArray       = array[1..MaxZips] of single;
    ZipCodeArray     = array[1..MaxZips] of integer;
    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, for each zip,
                                                the zip which is the nearest
                                                POD in the current solution.
                                                NextBestPOD holds the second-
                                                best current POD for each
zip.}
```

```
    mono_Xij                    :IntegerArray; { Used in the lagrangain
routine
                                                 to monitor the direction of
                                                 change for dual variables.  }
    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].
}
    Dual_var, Sum_Xij           : ValueArray;  { Lagrangian Work Arrays.
}
    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                      : ZipCodeArray;        { actual zip code
number
                                                             (in real
format).}
    CurrentNumber,zip,
```

```
    Nzips, Nposs, EndNumber    : integer;   {CurrentNumber is the number of
                                             POD sites assigned in the
current
                                             solution. 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 }
    Minimum_Zipcode            : longint;   { Smallest zip-code in state }
    StateName                  : string[2]; { Two-letter state code }
    response                   : string[1];
    StateNameFile              : text;
    ch                         : char;

{*********************************************************}

function exist(fn:FileString):boolean;
   { 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 assoc-
   iated 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
        begin
          done := true;
        end;
      while not done do
        if base=nil then { no POD is close enough, so this is illegal: }
          begin
            done := true;
            error:= true;
            writeln('feasiblity 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 { if POD in CurrentPODs...}
            else
```

```
              {writeln(' pod not in CurrentPODS ',pod:5);}
              base := base^.nextpod; { keep looking for a best POD }

        end; {while not done...}
{writeln('zip ',zip:5,' best ',bestpod[zip]:5,' cost
',currentcost[zip]:6:2);}
    end;  { for zip := 1 to ...}

end; { Procedure Match }

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

procedure ComputeCost;
   { Just add up all CurrentCost values, since fixed costs
     are stored in CurrentCost[ current POD site ].       }

  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({crt,}' 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
  for zip := 1 to Nzips do
```

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

{*****************************************************}

{$I init.pas   }      { Include array initializations }
{$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 Lagrangain Lower Bounds Algorithm}

begin  { M A I N   P R O G R A M }
  ClrScr;
  Assign(StateNameFile,'DISTRICT');
  reset(StateNameFile);
  read(StateNameFile,StateName);
  close(StateNameFile);

  Initialize;
  CreateDataStructures;
  Match;
  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;
      writeln;
      writeln (' ****************  FINAL SOLUTION  ****************** ');
      ComputeCost;
      ListCurrent;
      writeln;
      write(' Do you wish to produce lower bounds to the optimal solution (Y
or N)? ');
      readln(response);
      if (response = 'y') or (response = 'Y') then
        begin
          lagrangian_dual;
          writeln;
          write(' Please press the RETURN key to continue');
          readln;
        end;
      {Next section modified by M. McClain, 1/11/88}
      if exist('ADJACENT.'+StateName) 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;
         NextCost[i] := 1E+37;
         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: word;
           item2: double;
         end;

var
    pair_vec        : array[1..MaxRead] of pair;
    i,j,k,pntr      : word;
    pair_file       : file;
    hold            : Array [1..Maxpossible] of word;
    C_ij            : Array [1..Maxpossible] of single;
    filename        : FileString;
    Start_Memory,
    End_Memory      : longint;
    Number_Closed,
    count,POD,t,
    ipod,izip,start : word;
    PODnum,Number   : word;
    numread         : word;
    pt,p1,p2        : link;
    scanning        : boolean;
```

```
begin    { M A I N   P R O C E D U R E }

  count := 0;
  Nzips := 0;
  Minimum_Zipcode := 99999;
  Assign(Pair_file,'TRANSFER.'+StateName);
  reset(pair_file,10);

{ Determine the number of closed POD sites for this run }

  blockread (pair_file,pair_vec,3,numread);
  CurrentNumber:=pair_vec[3].iteml; { Number of existing POD's}
  Number_Closed:=pair_vec[2].iteml; { Number of new POD's currently
unopened}
  Nposs:=pair_vec[2].iteml + CurrentNumber; {Total number of POD's}
  writeln;
  writeln('  Total number of possible POD''s is    ',Nposs:3,'.');
  writeln('  Number of current and fixed POD''s is ',CurrentNumber:3,'.');
  repeat
    writeln;
    write('  Enter the desired number of POD''s in the final solution: ');
    error:=false;
    EndNumber:=0;
    read(ch);
    if ch<>#13 then
      repeat
        if ch in ['0'..'9'] then
          EndNumber:=EndNumber*10+ord(ch)-48
        else
          error:=true;
        read(ch)
      until ch=#13
    else
      error:=true;
    read(ch);
    if error then
       writeln(' Error in input -- please try again.')
    else if (EndNumber<1) or (EndNumber>Nposs) then
      begin
         writeln(' Error in input -- please enter a number between ',
                '1 and ',Nposs,'.');
        error:=true
      end
  until not error;
  {End of input buffering section}

  if Endnumber < CurrentNumber then  {Number of Sites Closed at termination}
     Number_Closed := Nposs - Endnumber;

  ClrScr;
  Writeln;
  Writeln(' Working ... ');
```

```
  Nposs := 0;
  CurrentNumber := 0;
  Start_Memory  := memavail;

  blockread (pair_file,pair_vec,MaxRead,numread);
  pntr := 1;

{ while ....   modified by PDD Nov. 17, 1988}

  while ( (numread=MaxRead) or (pntr < numread) )  do
    begin
      k := pair_vec[pntr].item1;
      k := k div 10 + 2;
      for j:= 1 to k do
        begin
          pntr := pntr+1;
          if pntr < numread+1 then
            begin
              hold[j] := pair_vec[pntr].item1;
              C_ij[j] := pair_vec[pntr].item2;
            end
          else
            begin
              pntr := 1;
              blockread( pair_file, pair_vec,MaxRead,numread);
              hold[j] := pair_vec[pntr].item1;
              C_ij[j] := pair_vec[pntr].item2;
            end;
        end;

{      for j := 1 to k do writeln(hold[1]:3,' ',hold[j]:3,C_ij[j]); }

      if pntr = numread-1 then
        begin
          pntr := 1;
          T := pair_vec[numread].item1;
          blockread( pair_file, pair_vec, MaxRead, numread);
        end
      else if pntr = numread then
        begin
          pntr := 2;
          blockread( pair_file, pair_vec, MaxRead, numread);
          T := pair_vec[1].item1;
        end
      else
        begin
          T := pair_vec[pntr+1].item1;
          pntr := pntr+2;
        end;

{ Start entering the i,j data.  Entries i,j are assumed to be in
  sorted order, except possibly last entry }

      number:= hold[1];
```

```
     count := succ(count);
     if Minimum_Zipcode=99999 then Minimum_Zipcode := trunc(C_ij[1]);
     ZCreal[number] := trunc(C_ij[1]-Minimum_Zipcode);
     if Nzips < number then Nzips := number;
     if T > 1 then { this is a possible POD site }
     begin
       if Index[number] = 0 then
         begin
           Nposs := succ(Nposs);
           Index[number] := Nposs;
           POD := Nposs
         end
       else
         POD := Index[number];

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

{ next 5 lines modified by PDD November 17, 1988 }

     if k > Number_Closed+1 then
        start := k-Number_Closed
     else
        start := 2;
     for j:=start to k do
       begin
          new(pt);
          with pt^ do
            begin
               site   := number;
               if hold[j]>maxzips then writeln(hold[j]);
               target := hold[j];
               cost   := C_ij[j];
               nextzip:= nil;
               nextpod:= nil;
            end;
          pl := Map[number];
          if pl = nil then
             Map[number] := pt
          else
            begin
              Map[number] := pt;
              pt^.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^.nextzip;
          if pl^.site <> pl^.target then
            begin
              CanBe[POD].next:=pt;
              pt^.nextzip:=pl;
            end
          else
            begin
              pt^.nextzip:=p2;
              pl^.nextzip:=pt;
            end;
        end;
    end;
  end;
 close(pair_file);

        { LST sends output to default list device (printer) }
        { Remove braces to change from console to printer }
 ClrScr;
 End_Memory:=memavail;
 writeln(' ');
 writeln('Starting Memory: ',Start_Memory:6,' bytes',
         ' Ending Memory: ',End_Memory:6,' bytes');
 writeln;
 writeln('      Total number of zipcodes is          ',count:4,'.');
 writeln('      Total number of possible POD''s is     ',Nposs:3,'.');
 writeln('      Number of current and fixed POD''s is
',CurrentNumber:3,'.');
 writeln('      Desired number of POD''s is            ',EndNumber:3,'.');
 writeln;
{ dumpstruct;} { diagnostic only--prints out entire data structure }
end;

procedure interchange;

{  This procedure performs the Teitz/Bart interchange heuristic
   for the 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 cur-
   rently assigned.  The best such exchange is performed, and
   the heuristic repeats until no advantageous exchanges exist. }

var
   done                           :boolean;
   POD,TestOut,SwapIn,SwapOut,i,ii :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,ww          : 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                      { permanent POD Site }
    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
           begin
            ww:=cost-currentcost[site];
            if (ww<0) then   { make new assignment }
              if (site<>BestPOD[site]) and (old_POD<>BestPOD[site]) then
                contr := contr + ww;
            end;
          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;
          base := base^.nextzip;
        end;

    end; (else clause from top)
  if illegal then
    Swapval := failure
  else
    Swapval := contr;
end; ( function swapval() )

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

( M A I N   P R O C E D U R E )
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;
                    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
                writeln(' Swapping ',Minimum_ZipCode+ZCreal[SwapOut]:5,
                             ' out, ',Minimum_ZipCode+ZCreal[SwapIn]:5,
                                    ' in at a cost of $',mincost:12:2);
                InsertPOD(SwapIn);
                RemovePOD(SwapOut);
                changes := true;
                done :=false;
              end;
          end;
      end;
  until done;
{  writeln(' Leaving interchange heuristic:');      }
end; {  interchange heuristic procedure  }



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,ww,wwl                     : 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];
          ww:=cost-currentcost[zip];
          wwl:=cost-nextcost[zip];
          if ( ww < 0 ) and ( old_POD <> zip ) then
```

```
            begin
              NextCost[zip] := CurrentCost[zip];
              CurrentCost[zip] := cost;
              NextBestPOD[zip] := old_POD;
              BestPOD[zip] := new_POD;
            end
          else if ( wwl < 0 ) then
            begin
              NextCost[zip] := cost;
              NextBestPOD[zip] := new_POD;
            end;
        end;  { with base^ do...}
        base:=base^.nextzip;
      end;
    NextBestPOD[new_POD] := BestPOD[new_POD];
    NextCost[new_POD]    := CurrentCost[new_POD];
    CurrentCost[new_POD] := new_cost;
    BestPOD[new_POD]     := 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
                if base2=nil then
                  begin
                    looking := false;
                    NextBestPOD[site] := 0;
                    NextCost[site] := 1E+37;
                  end
```

```
              else
                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
                    base2 := base2^.nextpod;
                end;
            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
              if base2=nil then
                begin
                  looking := false;
                  NextBestPOD[site] := 0;
                  NextCost[site] := 1E+37;
                end
              else
                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
                    base2 := base2^.nextpod;
                end;
            end;
        end;
      end;
    base := base^.nextzip;
  end;
end; { procedure RemovePOD() }

{-------------------------------------------------------------}
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,ww              : single;
        base                      : link;
        done                      : boolean;

begin                          { Procedure GreedyADD }
  opt := 1;
  impr := 1E+37;
  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
            with base^ do
              begin
                z := site;
                ww:=cost-currentcost[z];
                if (ww<0) and (z <> BestPOD[z]) then
                addval := addval - currentcost[z] + cost;
                base := nextzip;
              end;

          if addval < impr then  { this is the best choice so far }
            begin
              impr := addval;
              opt  := zip;
            end;
        end;  { if BestPOD... }
    end;      { for ind   ... }
    if impr < 1E+37 then          { add the new POD to the current set }
      begin
        InsertPOD(opt);
        writeln({LST,}' Adding POD ',Minimum_Zipcode+ZCreal[opt]:5,
                       ' at a cost of    $',impr:12:2);
      end
  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
      writeln(' Deleting POD ',Minimum_Zipcode+Zcreal[opt]:5,
              ' at a cost of $',minval:12:2);
      RemovePOD(opt);
    end
  else
    begin
      stuck := true;
      writeln(' No POD can legally be deleted from current allocation.' );
    end;
end;  {procedure GreedyDEL}

{************************************************************}
```

```
procedure greedy;

{
   This procedure, given an initial and a final number
   of POD sites, adds or subtracts sites using the
   greedy heuristic until the desired number remain.
   Procedures GreedyADD and GreedyDEL are called.}

{****************************************************}

begin  { M A I N   P R O C E D U R E  }
{ writeln(' Entering Greedy heuristic...');}
  stuck := false;
  changes := false;
  while (CurrentNumber <> EndNumber) and not stuck do
    if (CurrentNumber < 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; { Procedure Greedy }

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;
      CharacterArray = array[1..Maxpossible] of char;
Const
      w_eps = 0.001;
      max_iter = 150;
var
      Hold_POD_set          : zipset;
      Running_Average       : Array[1..5] of real;
      POD_Indicator         : CharacterArray;
```

```
      red_cost,work        : RealArray;
      pod_id               : IntegerArray;
{     Dual_var             : ValueArray;}
{     Sum_Xij              : ValueArray;} {Indexarray;}
{     mono_Xij             : Indexarray;}
      error,Existing,Fixed: boolean;
      base                 : link;
      dual_file            : file of real;
      filename             : FileString;
      Out_file             : Text;
      Number_Fixed         : integer;
      factor, save, delta_s, scale_factor, norm_factor,
      min_dif, w_target, w_previous, w_new, s, delta, Minus_infinity:
double;
      Sum_Xjj, monotone, nits, iter, pod, ipod, zip, i, j, tick  : integer;

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,izip    : 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; {while not done...}
    end;  { for zip := 1 to ...}
end; { Procedure Assignment }

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 ',l:5,r:5,m:5,x);}
          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 L>=R
      End;
   End { QuickSort };

{**********************************************************************}

begin
{                        Initialize Lagrangian Solution                }
  tick := 0;
  nits := 60;
  Fixed:= false;
  Minus_infinity:=-1e37;
  Error:= false;
```

```
  scale_factor:= 1;
  norm_factor := 1;
  w_previous  := 0;
  w_new       := 2*w_eps;
  w_target    := totalCost+10;
  hold_POD_set:= CurrentPODs;
  Number_fixed:= 0;
  for i:=1 to Maxpossible do POD_indicator[i]   := ' ';
  for i:=1 to 5           do Running_average[i] := 0;

{         Compute a dual value for the number of open POD sites          }

  delta_s:=totalcost;
  GreedyADD;
  ComputeCost;
  s:=totalcost-delta_s;
  CurrentPODs:=hold_POD_set;
  match;
{                        Perform File Initialization                     }

{  assign(dual_file,'dual.var');
  rewrite(dual_file); }

{       Initialize dual variables using best and nextbest costs          }

  for i:=1 to nzips do
    begin
    if map[i]<>nil then
      begin  {       determine an interval for the dual variable         }
        if nextbestPOD[i] = 0 then
          nextcost[i]   := 1.5*currentcost[i]
        else if currentcost[i] >= nextcost[i] then
          currentcost[i]:= nextcost[i]/2;
        factor  := CurrentCost[i]/nextcost[i];
        nextcost[i] := nextcost[i]-currentcost[i];
        if nextcost[i] > abs(s) then nextcost[i]:= abs(s);
        dual_var[i] := CurrentCost[i] + factor*NextCost[i];
        mono_Xij[i] := 0;
      end
    else
      dual_var[i]:=0.0;
    end;
{                          Check for Fixed Sites                   }
  for i:=1 to nposs do
    if CanBe[i].must then
      begin
        Fixed:=true;
        Number_Fixed:=Number_Fixed+1;
      end;
   if Number_Fixed >= Endnumber then
     writeln('ERROR: Illegal Number of Fixed Variables');

{                          Begin the main loop                     }
```

```
  clrscr;
  writeln('                    The Lagrangian Lower Bound ');
  writeln;
  writeln('      The best Greedy-Interchange solution value is
',delta_s:8:2);

  while ( (abs(w_previous-w_new) > w_eps) and (tick < max_iter)  )  do
    begin
      monotone:=0;
      for iter:=1 to nits do
        if norm_factor > 0.9 then
        begin
{                        Compute Reduced Costs                        }

          tick := tick + 1;
          if (iter = 1) or Fixed then
          begin
           if iter > 1 then for i:=1 to nzips do
           dual_var[i]:=dual_var[i]+Sum_Xij[i];
           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;
           end
          else  {          Compute the same thing only faster          }
           begin
              for i:= 1 to nposs do
                work[i]:=work[i]+delta_s*(Sum_Xjj-Endnumber);
              for i:= 1 to nzips do
                if Sum_Xij[i] <> 0 then { Examine only those which changed }
                  begin
                    dual_var[i] := dual_var[i]+Sum_Xij[i];
                    base:= map[i];
                    while base <> nil do
                      with base^ do
                        begin
                          ipod:= Index[target];
                          save:= cost-dual_var[i];
                          if site=target then
                            work[ipod] := work[ipod]-Sum_Xij[i]
                          else if save < 0 then
                            begin
                              if save+Sum_Xij[i]<0 then
```

```
                  work[ipod] := work[ipod] - Sum_Xij[i]
                else
                  work[ipod] := work[ipod] + save;
              end
            else if save+Sum_Xij[i]<0 then
              work[ipod] := work[ipod] - save - 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;
      if Fixed and canbe[i].must then
        red_cost[i]:=Minus_Infinity
      else
        red_cost[i]:=work[i];
    end;
  Quicksort(red_cost,pod_id,nposs);
  if Fixed then for i:=1 to nposs do
    if canbe[pod_id[i]].must then red_cost[i]:=work[pod_id[i]];

{     find an approximate feasible solution with k-1 k or k+1 PODs       }

  Sum_Xjj:=0;
  for i:= 1 to nposs do
    if (Red_cost[i] < 0) or CanBe[pod_id[i]].must then
      Sum_Xjj:=Sum_Xjj+1;

  if ( abs(Sum_Xjj - EndNumber) > 0 ) then
    delta_s := red_cost[EndNumber+1]/2 + red_cost[Endnumber]/2
  else
    delta_s:=0;
  s:= s+delta_s;

  w_previous := w_new;
  w_new      := s*EndNumber;
  CurrentPODs:= [];
  Sum_Xjj    :=  0;
  for i:= 1 to nposs do
    begin
      if ( delta_s <> 0 )then
        begin
          red_cost[i]:= red_cost[i] - delta_s;
          work[i]    := work[i] - delta_s;
        end;
        if (red_cost[i] < 0) or canbe[pod_id[i]].must then
        begin
          w_new:=w_new+red_cost[i];
          Sum_Xjj:=Sum_Xjj+1;
          CurrentPODs:=CurrentPODs+[pod_id[i]];
```

```
            end;
          end;

{                   Compute the Xij's and Objective Value            }

        for i:=1 to Nzips do
          if map[i]=nil then
            Sum_Xij[i] := 0
          else
            Sum_Xij[i] :=-1;

        for j:=1 to Nposs do
          begin
            base := CanBe[j].next;
            pod  := base^.site;
            ipod := index[pod];
            if ipod in currentPODs then
              while base <> nil do
                with base^ do
                  begin
                    save:=cost-dual_var[site];
                    if (site=pod) or (save < -w_eps) then
                      Sum_Xij[site] := Sum_Xij[site]+1
                     else
                      if save < w_eps then
                        if Sum_Xij[site] < 0 then
                          Sum_Xij[site] := 0;
                    base:=nextzip;
                  end;
          end;

        norm_factor:=0.0;
        for i:=1 to nzips do
          begin
            w_new:=w_new+dual_var[i];
            norm_factor:=norm_factor+abs(Sum_Xij[i]);
          end;

        if norm_factor = 0 then
           begin
             norm_factor:=0.9;
             w_previous:=w_new;
           end;

        Running_average[iter mod 3 + 1] := norm_factor;
        save :=
(Running_average[1]+Running_average[2]+Running_average[3])/3;

        { compute a new scale factor and compute new dual variables   }

        if w_new-w_previous < -w_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;

          factor:= scale_factor*(w_target-w_new)/save + w_eps;

{         compute the min_dif needed to change Sum_Xij by 1          }

         GotoXY(1,5);
         min_dif:=1E+37;
         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];
              if min_dif > factor then
              while base <> nil do
                with base^ do
                  begin
                    delta:=1E+37;
                    pod := target;
                    ipod:= index[pod];
                    save:= cost - dual_var[i];
                    if (ipod in currentPODs) and (save > 0) then
                        delta:=save  {;}
                    else
                    if not (ipod in currentPODs) then
                      begin
                      if site = target then
                        delta:=work[ipod]
                      else
                      if save > 0 then
                        delta:=save+work[ipod]
                      else
                      if save < 0 then
                        delta:=work[ipod];
                      end;
                    if delta < min_dif then
                      min_dif:=delta;
                    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];
          if min_dif > factor then
          while base <> nil do
            with base^ do
              begin
                pod := target;
                ipod:= index[pod];
                save:= cost-dual_var[i];
                if (ipod in currentPODs) and (save < 0) then
                  begin
                    if (-save/Sum_Xij[i] < min_dif) then
                      min_dif := -save/Sum_Xij[i];
                    if (-work[ipod]/Sum_Xij[i] < min_dif) then
                      min_dif := -work[ipod]/Sum_Xij[i];
                  end;
                base:=nextpod;
              end;
        end
      else
        mono_Xij[i]:=0;
      end;

     for j:=1 to Nposs do
       begin
         base := CanBe[j].next;
         zip  := base^.site;
         pod  := index[zip];
         if pod in currentPODs then
           POD_indicator[pod]:='1'
         else
           POD_indicator[pod]:='0';
       end;

    if(min_dif<1E+37) and (min_dif>factor)  then
        begin
{           writeln(lst,' min_dif applied ',min_dif:10:5,'
factor',factor:10:5);}
         factor := min_dif;
        end;

     save:=100.0*w_new/w_target;
{     writeln(POD_indicator);
     writeln(' iter w_target     w_new       norm_fact factor',
             '   min_dif     s      Sum_Xjj ');
     writeln(tick:3,' ',w_target:12,' ',w_new:12,' ',norm_factor:9,
             ' ',factor:12,' ',min_dif:9,' ',s:6:3,' ',Sum_Xjj:3);}
  writeln('      Upgraded Lower Bound for the Optimal Solution is
',w_new:8:2);
  writeln;
  writeln('      Percentage of the Greedy-Interchange Solution is
',save:4:2,'%');
  writeln;
  writeln('      Iteration ',tick:4);
```

```
          for i:=1 to nzips do
            begin
            if abs(mono_Xij[i]) >= 5 then
              begin
                Sum_Xij[i]:=Sum_Xij[i]*2;
{               writeln(' 2*Sum_Xij on ',i:4);}
              end;
            if min_dif = factor then
              Sum_Xij[i]:=-factor*Sum_Xij[i]
            else if factor < nextcost[i] then
              Sum_Xij[i]:=-factor*Sum_Xij[i]
            else
              Sum_Xij[i]:=-Sum_Xij[i]*scale_factor*Nextcost[i];
            end;

          if factor > 10 then
            delta_s:=scale_factor
          else if (Sum_Xjj-EndNumber>0) and (factor<-red_cost[Sum_Xjj]) then
            delta_s:=abs(red_cost[Sum_Xjj])
          else if (Sum_Xjj-EndNumber<0) and (factor<red_cost[Sum_Xjj+1])
then
            delta_s:=abs(red_cost[Sum_Xjj+1])
          else
            delta_s:=factor;
          s:=s-delta_s*(Sum_Xjj-EndNumber);
        end;
          Assignment;
          if error then
            begin
{              writeln(' current lagrangian solution not feasible ');}
            end
          else
            begin
              ComputeCost;
{              listcurrent;}
              if (totalCost < w_target) and (Sum_Xjj = EndNumber) then
                begin
                  if abs(totalCost-w_new) > w_eps then
                     save:= (w_target-w_new)/(totalCost-w_new)
                  else
                     save:=1;
                  if save > 2 then
                    begin
{                     writeln(lst,' scale_factor adjusted
',scale_factor:5:4);}
                      scale_factor:=scale_factor*save;
                      if scale_factor > 1 then scale_factor:=1;
                    end;
                  w_target:=totalCost;
                  hold_POD_set:=currentPODs;
                  save:=100.0*w_new/w_target;
 {                writeln('upgraded w_target',w_target);}
  clrscr;
  writeln('                      The Lagrangian Lower Bound ');
```

```
  writeln;
  writeln('      An Improved Greedy-Interchange solution value is
',totalcost:8:2);
  writeln;
  writeln('      Upgraded Lower Bound for the Optimal Solution is
',w_new:8:2);
  writeln;
  writeln('      Percentage of the Greedy-Interchange Solution is
',save:4:2,'%');
  writeln;
  writeln('      Iteration ',tick:4);
                  end;
              end;
      iter:=nits shr 1;
      nits:=iter+10;
      scale_factor:=scale_factor/2.0+w_eps;
{     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);}
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
  Computing", 1972 Academic Press, N.Y.  The idea for the algorithm
  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[count] := nodes;
        end;

end; { init_graph }

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

procedure write_out_graph (var graph : graph_type; num_nodes : integer);
var
    count : integer;
    temp : ptr_type;

begin   { write_out_graph }
        writeln; writeln;
    for count := 1 to num_nodes do begin
        write('adjacency 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;

{This procedure modified by M. McClain, 1/12/88}

const     maxadj = 25; {Warning - This constant must match the value
                        of MAXADJ used in the installation program
                        subroutine ADJ.  Changes in this value require
                        changes in the buffer record structure below.}

var       Zip1,Zip2 : string[5];
          ind,count,blocksize : integer;
          adj_file : file;
          buffer : record
                      zipindex : word;
                      adjnum : word;
                      zipcode : double;
                      blankreal1,blankreal2,blankreal3,blankreal4 : double;
                      blankint1,blankint2,blankint3 : word;
                      neighbor : array[1..maxadj] of word
                    end;

  {-------------------------------------------------------}
    procedure Add_to_list(z,n:integer);
      var pod1, pod2 : integer;
          ptr, p     : ptr_type;

      begin
        pod1 := Node_num[Index[BestPOD[z]]];
        pod2 := Node_num[Index[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,'ADJACENT.'+StateName);
  blocksize := 4*maxadj;
  reset(adj_file,blocksize);
  while not EOF(adj_file) do
    begin
      blockread(adj_file,buffer,1);
      ind := buffer.zipindex;
```

```
      if buffer.zipcode <> ZCreal[ind]+Minimum_Zipcode then
        begin
          str(buffer.zipcode:5:0,Zipl);
          writeln('ERROR: Zip code mismatch for index ',ind:5,' --');
          writeln('        ',Zipl,' in adjacency file,');
          writeln('        ',ZCreal[ind],' in solver.');
        end;
      if buffer.adjnum > 0 then
        for count := 1 to buffer.adjnum do
            if BestPOD[ind]<>BestPOD[buffer.neighbor[count]] then
              Add_to_list(ind,buffer.neighbor[count])
    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
         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
                  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; { find min degree }
{-----------------------------------------------------------}

procedure order_graph (var graph : graph_type; num_nodes : integer;
          var ordering : Node_array);
var
   deleted : 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; { for }
end; { order_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 current > 0 then
              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 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 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; { for }
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
     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;
               if(not (current IN visited))and
                 (color[current] IN sub_colors) then  begin
                      find_component(current);
               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]; }
                   trial_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 .. 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 }

{-------------------------------------------------------}
```

```
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 := 'SOLUTION.'+StateName;
{     GettingName := exist(filename);
      while GettingName do
        begin
          writeln(' NOTE: file ',filename,' already exists:');
          writeln(' Write over this file ? ');
          readln(answer);
          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);
      {Next section modified by M. McClain, 1/22/88}
      write(Writefile,totalcost:12:2,' ',CurrentNumber:3);
      writeln(Writefile);
      for pod:=1 to Nzips do
        if Index[pod] <> 0 then
          if (Index[pod] in CurrentPODs) then
            write(WriteFile,Canbe[Index[pod]].where:5);
      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[Index[pod]];
```

```
                    write(Writefile,count:5,ZCreal[count]+Minimum_Zipcode:6,
                    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 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,'SOLUTION.'+StateName);
  Rewrite(WriteFile);
  write  (Writefile,totalcost:12:2,' ',CurrentNumber:3);
  writeln(Writefile);
  for pod:=1 to Nzips do
    if(Index[pod]<>0)then
      if(Index[pod] in CurrentPODs)then
        write(writefile,Canbe[Index[pod]].where:5);
  writeln(Writefile);
  clr:=0;
  for zip:=1 to Nzips do
    if(map[zip]<>nil)then
    begin
      write  (WriteFile,zip:5,ZcReal[zip]+Minimum_Zipcode:6,
              clr:3,BestPOD[zip]:5);
      writeln(WriteFile);
    end;
  Close(Writefile);
end; { PODColor }
```

NBS-114A (REV. 2-8C)

U.S. DEPT. OF COMM.

BIBLIOGRAPHIC DATA SHEET (See instructions)

1. PUBLICATION OR REPORT NO.
NISTIR 86-3472-1

2. Performing Organ. Report No.

3. Publication Date
FEBRUARY 1989

4. TITLE AND SUBTITLE
The Internal Revenue Service Post-of-Duty Location Modeling System (Version 5.0) - 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
U.S. DEPARTMENT OF COMMERCE
GAITHERSBURG, MD 20899

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, NW
Washington, DC 20224

10. SUPPLEMENTARY NOTES

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

11. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)

This report is a programmer's manual for a microcomputer system designed at the National Institute of Standards and Technology 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.

12. KEY WORDS (Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)
Greedy heuristic; Interchange heuristic; Lagrangian Relaxation; Uncapacitated fixed charge facility location problem

13. AVAILABILITY

[X] Unlimited
☐ For Official Distribution. Do Not Release to NTIS
☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.
[X] Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES
65

15. Price
$14.95

USCOMM-DC 6043-P80