The 101 flag is set to 1 (true).

The slides flag is not set to 1 (not set to true).

The Exen 550 flag is not set to 1 (not set to true).

This is the full (if incomplete) version, to be used by sok for OPIM 101 during the regular semester.

File (master): imsnotes.tex
Contents

Preface xi

I Preliminaries 1
1 Introduction & Overview 3
   1.0 Slides .............................................. 3

II Information Retrieval 7
2 Basics of Information Retrieval 9
   2.0 Slides ............................................ 9
   2.1 Bibliographic Note .............................. 24

3 Information Retrieval Exercises 25

III Decision Support: Basics and Theory 27
4 Decision Support Systems 29
   4.0 Slides .......................................... 29

5 A Brief Introduction to Decision Analysis 35
   5.0 Slides ............................................ 35
   5.1 Overview ......................................... 56
   5.2 Decision Trees and Their Analysis .................. 63
   5.3 Conditional Probability ........................... 67
   5.4 More Information .................................. 68
   5.5 Utility Theory ..................................... 82
   5.6 Multiattribute Utility Theory (MAUT) Models ........ 88
   5.7 Comments on the Use of Decision Analysis ...... 91
   5.8 Decision Analysis with Spreadsheets .............. 92
   5.9 Bibliographic Notes ............................... 99
5.10 Exercises ................................................. 99
5.11 Version Notes ............................................ 100

6 Case: DSS Evaluation with MAUT 103
6.1 Introduction ............................................. 103
6.2 The Basic Model and Its Attributes ....................... 105
6.3 An Example: Responses to Congressional Questions .... 106
6.4 Discussion ................................................ 110
6.5 Exercise .................................................. 110
6.6 Bibliographic Note ....................................... 111
6.7 Version Notes ............................................ 111

IV Elements of Database 113

7 Database 115
7.0 Slides .................................................. 115

8 QBE: Query by Example 125
8.0 Slides .................................................. 125

9 SQL and Access 131
9.0 Slides .................................................. 131

10 E-R Modeling 145
10.0 Slides .................................................. 145
10.1 Bibliographic Note .................................... 171

11 Database Normalization 173
11.0 Slides .................................................. 173

12 Database and Excel 199
12.0 Slides .................................................. 199

13 Database Exercises 201

V Database Case 207

14 Modeling EDI Messages 209

VI Elements of Programming 211

15 Visual Basic for Applications: A Brief Tutorial 213
15.0 Slides .................................................. 213
15.1 First Steps ............................................. 243
15.2 Second Steps ........................................... 244

iv
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.3 Variables</td>
<td>246</td>
</tr>
<tr>
<td>15.4 Boolean Operators</td>
<td>249</td>
</tr>
<tr>
<td>15.5 Control Structures</td>
<td>251</td>
</tr>
<tr>
<td>15.6 Arrays</td>
<td>254</td>
</tr>
<tr>
<td>15.7 Miscellaneous Topics</td>
<td>255</td>
</tr>
<tr>
<td>15.8 Bibliographic Note</td>
<td>259</td>
</tr>
<tr>
<td>15.9 Version Notes</td>
<td>259</td>
</tr>
<tr>
<td>16 VBA Exercises</td>
<td>261</td>
</tr>
<tr>
<td>17 VBA Programming Exercises</td>
<td>277</td>
</tr>
<tr>
<td>VII Programming Case: GAs and Post-Evaluation Analysis</td>
<td>279</td>
</tr>
<tr>
<td>18 Candle-Lighting Analysis</td>
<td>281</td>
</tr>
<tr>
<td>18.0 Slides</td>
<td>281</td>
</tr>
<tr>
<td>18.1 Introduction</td>
<td>293</td>
</tr>
<tr>
<td>18.2 Sensitivity Analysis</td>
<td>294</td>
</tr>
<tr>
<td>18.3 Model Validation</td>
<td>297</td>
</tr>
<tr>
<td>18.4 Option Discovery</td>
<td>298</td>
</tr>
<tr>
<td>18.5 Reducing Infeasibility</td>
<td>302</td>
</tr>
<tr>
<td>18.6 On the Role of the GA</td>
<td>303</td>
</tr>
<tr>
<td>18.7 Summary</td>
<td>305</td>
</tr>
<tr>
<td>18.8 Version Notes</td>
<td>309</td>
</tr>
<tr>
<td>19 Case 5, VB: Instructions, OPIM 101, Spring 1996</td>
<td>311</td>
</tr>
<tr>
<td>19.1 Task 1: Obtain the GA Run Parameters Interactively</td>
<td>311</td>
</tr>
<tr>
<td>19.2 Task 2: Obtain the Model Parameters Interactively</td>
<td>312</td>
</tr>
<tr>
<td>19.3 Task 3: Place Outputs in the Workbook</td>
<td>313</td>
</tr>
<tr>
<td>19.4 Task 4: Label and Format the Output Sheets</td>
<td>313</td>
</tr>
<tr>
<td>19.5 Task 5: Report Progress Information</td>
<td>314</td>
</tr>
<tr>
<td>19.6 Task 6: Add a New Model</td>
<td>314</td>
</tr>
<tr>
<td>19.7 Task 7: Respond to an Opportunity</td>
<td>316</td>
</tr>
<tr>
<td>19.8 Hints and Suggestions</td>
<td>316</td>
</tr>
<tr>
<td>19.9 Grading Policy</td>
<td>317</td>
</tr>
<tr>
<td>19.10 An Inventory Opportunity</td>
<td>317</td>
</tr>
<tr>
<td>19.11 Version Notes</td>
<td>319</td>
</tr>
<tr>
<td>20 BasicGA: Code for Genetic Algorithms</td>
<td>321</td>
</tr>
<tr>
<td>20.1 Introduction</td>
<td>321</td>
</tr>
<tr>
<td>20.2 Declarations</td>
<td>321</td>
</tr>
<tr>
<td>20.3 Sub DoTheGA: Code Structure Overview</td>
<td>326</td>
</tr>
<tr>
<td>20.4 PrepareGA: Detailed Code Structure</td>
<td>330</td>
</tr>
</tbody>
</table>
20.5  Sub RunGAUnit1Done: Detailed Code
       Structure .................................................. 332
20.6  Sub PostpareGA: Detailed Code
       Structure .................................................. 333
20.7  Complete Code Listing ........................................ 333
20.8  File Notes .................................................. 351

VIII  Conclusion ........................................ 353

21  Upward and Onward ........................................ 355

A  Answers to Selected Exercises 357
   A.1  VBA, Chapter 16 .......................................... 357

References ........................................ 359

Index .................................................. 362
## List of Figures

5.1 Decision Tree for the Parking Meter Problem ........................................ 65
5.2 Venn Diagram of Probabilities .......................................................... 68
5.3 Decision Tree under Perfect Information ........................................... 70
5.4 Subtree with Street Person’s Report  (Note: $c$ denotes the cost of the report.) ................................................................. 72
5.5 Decision Tree for the Parking Meter Problem ....................................... 95
5.6 Input parameters for the parking meter problem, plus table of names and named ranges for the Excel implementation (in files: parkdt.xls and parkdtform.xls). ......................................................... 99
5.7 Formulas (equational expressions) for the parking meter problem, implemented in Excel (in files: parkdt.xls and parkdtform.xls). ......................................................... 100
5.8 Formulas (equational expressions) for the parking meter problem, implemented in Excel, Excel formulas shown (in files: parkdt.xls and parkdtform.xls). ......................................................... 100
5.9 Chance node for the parking meter problem, implemented in Excel (in files: parkdt.xls and parkdtform.xls). ......................................................... 101
5.10 Chance node for the parking meter problem, implemented in Excel, Excel formulas shown (in files: parkdt.xls and parkdtform.xls). ......................................................... 101
5.11 Decision node for the parking meter problem, implemented in Excel (in files: parkdt.xls and parkdtform.xls). ......................................................... 101
5.12 Decision node for the parking meter problem, implemented in Excel, Excel formulas shown (in files: parkdt.xls and parkdtform.xls). ......................................................... 102

13.1 Illustration for question 10 ................................................................. 204
13.2 Illustration for question 11 ................................................................. 205

15.1 Contents Tab in the Help Menu for VBA ............................................ 242
15.2 Index Tab in the Help Menu for VBA .................................................. 244

16.1 Table for the sub “Question13” .......................................................... 263
16.2 Code for Question 15 ....................................................................... 267
16.3 Definitional information on Mod from Microsoft’s online help for VBA ................................................................................. 268
16.4 Code for Question 16 ....................................................................... 268
16.5 Code for Question 17 ....................................................................... 270
Illustration of Pythagorean theorem. The triangle has three sides: $a, b, c$. The angle between sides $a$ and $b$ is $90^\circ$. The lengths of the sides are as follows: length $a = ||a||$, length $b = ||b||$, length $c = ||c||$. According, then, to the Pythagorean theorem, $||c||^2 = ||a||^2 + ||b||^2$. 

Barrier of length $l$, patrolled by a vessel with radius of detection, $r$. The Coast Guard vessel, located at the center of the circle, is traveling at a speed of $v$ to the “west” and a target vessel is approaching with a speed of $u$, heading “south.” (The height of the barrier is $2 \cdot r$. The patrol area has area $2 \cdot r \cdot l$.)
## List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Outcome Matrix for the Parking Meter Problem</td>
<td>58</td>
</tr>
<tr>
<td>5.2</td>
<td>Desirabilities Matrix for the Parking Meter Problem</td>
<td>58</td>
</tr>
<tr>
<td>5.3</td>
<td>Example Outcome Matrix (with Specific Outcome Values) for the Parking Meter Problem</td>
<td>59</td>
</tr>
<tr>
<td>5.4</td>
<td>Conditional Probabilities Matrix for the Parking Meter Problem, Given Choice</td>
<td>59</td>
</tr>
<tr>
<td>5.5</td>
<td>Numerical Example of Conditional Probabilities Matrix for the Parking Meter Problem, Given Choice</td>
<td>59</td>
</tr>
<tr>
<td>5.6</td>
<td>Expectations Matrix for the Parking Meter Problem</td>
<td>60</td>
</tr>
<tr>
<td>5.7</td>
<td>Tabular Approach to Bayes’s Rule: Conditional Probabilities of Rows, Given Columns</td>
<td>79</td>
</tr>
<tr>
<td>5.8</td>
<td>Tabular Approach to Bayes’s Rule: The Given Information, Plus One Step</td>
<td>80</td>
</tr>
<tr>
<td>5.9</td>
<td>Table 5.8 Simplified: Joint Probabilities of Rows and Columns</td>
<td>80</td>
</tr>
<tr>
<td>5.10</td>
<td>(Almost) Final Table</td>
<td>80</td>
</tr>
<tr>
<td>5.11</td>
<td>Table 5.10 Simplified: Conditional Probabilities of Columns, Given Rows</td>
<td>81</td>
</tr>
<tr>
<td>5.12</td>
<td>Parameters for the Parking Meter Problem</td>
<td>94</td>
</tr>
<tr>
<td>5.13</td>
<td>Formulas for the Parking Meter Problem</td>
<td>96</td>
</tr>
<tr>
<td>5.14</td>
<td>Chance Nodes: Layout and Key Formulæ</td>
<td>96</td>
</tr>
<tr>
<td>5.15</td>
<td>Decision Node dP: Layout</td>
<td>97</td>
</tr>
<tr>
<td>5.16</td>
<td>Decision Node dP: Expanded Layout to Include the Optimal Choice</td>
<td>97</td>
</tr>
<tr>
<td>15.1</td>
<td>Truth Table for And</td>
<td>250</td>
</tr>
<tr>
<td>15.2</td>
<td>Truth Table for Or</td>
<td>250</td>
</tr>
<tr>
<td>15.3</td>
<td>Truth Table for Not</td>
<td>251</td>
</tr>
<tr>
<td>15.4</td>
<td>Truth Table Showing Definition of And in terms of Not and Or</td>
<td>251</td>
</tr>
<tr>
<td>16.1</td>
<td>Input data for Questions 15 and 16</td>
<td>266</td>
</tr>
<tr>
<td>16.2</td>
<td>Input data for Question 17</td>
<td>269</td>
</tr>
<tr>
<td>18.1</td>
<td>From BestNSaveSet: Top solutions found by CLAP-NT for test knapsack problem with $m = 80$ and $n = 30$. Values for $w_i$ and $p_i$ were randomly drawn. Found values of $z$: $z_1 = 274.93$, $z_2 = 266.99$, $z_3 = 266.20$, $z_4 = 261.66$, $z_5 = 261.63$.</td>
<td>306</td>
</tr>
</tbody>
</table>
18.2 Summary statistics for the 30 best distinct results on runs of 12 random knapsack models, for the CLAP-NT GA (population size = 100) versus a random search in the same space. Max and Min values are averaged across the 30 best distinct solutions found in each of two runs, at 5,000 and 10,000 function evaluations.

19.1 Assumed Parameter Values
18.1 Introduction

Achieving any single objective typically requires consumption of scarce, or not fully sufficient, resources. Further, goals and objectives rarely exist in isolation. Partial conflict among them, and mutual competition for limited resources, is the norm. Thus, the world presents us with nearly unrestricted opportunities for optimization in the presence of constrained resources.

Optimization, as a practical endeavor, is properly concerned with much more than merely a recommendation for the allocation of scarce resources. Any such recommendation must be based upon assumptions, and models, that are at best approximate and that nearly always must leave out important information. Optimization procedures optimize models, not the original problem. Thus, much of the decision maker’s work, at least the thoughtful, deliberative part of it, occurs during the interpretation phase of the modeling and optimization effort. Interpreting and seeking to understand what a model has to say is variously called post-evaluation analysis or post-solution analysis or post-optimality analysis, although, as we shall see, it also has a useful role in model validation.

There are essentially three kinds of techniques for supporting post-solution analysis:

1. Analytic. Taking partial derivatives of a model for purposes of sensitivity analysis is an example. This may work well for some problems, especially for unconstrained optimization of differentiable models, but it is too limited for general practical purposes.

2. Solver by-product. When, e.g., linear programs are solved using the simplex method, a great deal of information for post-evaluation analysis is produced as a by-product of solving the problem. This information includes dual variable values, reduced costs, and basis-maintaining ranges on variables and parameters.

3. Multiple-solution. There are several popular computational techniques for supporting post-solution analysis. These techniques all require repeated solution of the model in question, using perturbed input parameter values. The main multiple solution techniques are: what-if analysis (usually manually directed), goal-seeking, data tables (a feature supported in most spreadsheet programs), and grid searches. Although much used and quite valuable, these multiple solution techniques are limited in important ways. Manually-directed what-if questioning is slow, unsystematic, and at risk of distortion because of human biases and cognitive limitations. Complete grid searches may in principle produce a wealth of information for post-solution analysis, but they are often computationally impracticable.

With no essential loss of generality (for present purposes), we may think of the post-solution analysis task for an optimization problem as presenting the
decision maker with a finite evaluation or objective function hypersurface, which we shall refer to as the decision surface. Each point on the decision surface represents the value of the objective function for some permitted setting of the variables for the problem.\textsuperscript{4,5} Post-evaluation analysis may be thought of as the task of exploring the decision surface for the sake of gaining insight and coming to a decision.

Clearly, the decision maker will want to know which point or points on the decision surface are optimal.\textsuperscript{6} Providing this information is what optimization algorithms are mainly about. Post-evaluation analysis is about asking and answering other important questions regarding the evaluation hypersurface.

This paper is about heuristic, multiple-solution mapping of decision surfaces for purposes of supporting post-evaluation analyses, a process we call \textit{candle-lighting analysis} or CLA (cf., [12, 14, 15, 13, 6, 7]). In general, heuristic mapping of decision surfaces is needed because the computational cost of exhaustive mapping is prohibitive. In what follows we implicitly describe our CLA concepts by directly describing a series of examples and results obtained from an existing implementation, CLAP-NT (candle-lighting analysis program in NT).\textsuperscript{7,8}

18.2 Sensitivity Analysis

Given a model, a set of data (input parameters) for the model, and a solution for, or evaluation of, the model, there are a number of general questions we need to ask before a decision is made and a course of action taken. These questions include the following.\textsuperscript{9}

1. How sensitive (or robust) is the model to minor changes in its recommendations (particularly to minor changes in the values of its decision variables)?

\textsuperscript{4}We assume that ranges are given for the decision variables. A permitted setting is one in which each decision variable is set at a value within its range. This setting need not result in a feasible solution to the optimization problem.

\textsuperscript{5}The variables used to generate the decision surface will normally include the decision variables for the optimization problem. Also, for purposes we explain in the sequel, the surface-generating variables may include model parameters, e.g., right-hand side constants, which are allowed to vary over a specified range.

\textsuperscript{6}Here and in the sequel we assume without loss of generality that we have a maximization problem, so that the decision maker is seeking the “highest” points on the decision surface.

\textsuperscript{7}Space is greatly restricted in this paper. See [7] for additional information on CLAP-NT and other aspects of this work.

\textsuperscript{8}One reviewer of the paper commented that the results we achieve involve more than heuristic mapping of the decision surfaces. Yes, but—and there is insufficient space to go into this point—all of the results we report are obtained by extraction of information from heuristically-mapped decision surfaces. The key point to note is that these surfaces may be generated by more than variations in the decision variables. We also examine ranges of input parameters, e.g., right-hand side constants, in generating our surfaces. In short, given the evaluation purposes at hand, we will often blur the distinction between decision variables and input parameters in the model.

\textsuperscript{9}For more detailed development of these concepts, see [6].
2. How sensitive (or robust) is the model to minor changes in its assumptions (particularly to minor changes in the values of its input parameters)?

3. Are there attractive alternative decisions, and, if so, what are they and what are their sensitivity properties?

For each of these questions, CLAP-NT provides general, heuristic approaches for finding answers. These approaches apply to linear and nonlinear models with or without integer-valued decision variables.

To illustrate, consider the knapsack problem having the form given in Expression 18.1:

\[
\text{max } z = \sum_{i=1}^{n} p_i \cdot x_i
\]

where \( \sum_{i=1}^{n} w_i \cdot x_i \leq m \)

\( x_i \in \{0, 1\} \)  \hspace{1cm} (18.1)

Knapsack problems offer easily-interpreted, simply-created integer programs that typically are solved by creation of an integer linear program whose relaxed LP version is solved with the simplex algorithm, and whose integer constraints are then enforced with the branch-and-bound method. Because the simplex algorithm is not the final step in solving knapsack problems this way, the extensive set of sensitivity information generated in its solution process is effectively lost.\(^{10}\)

CLAP-NT presents a way of creating sensitivity analysis information as a residue of the solution mechanism. Using genetic search (in the current version of CLAP-NT), the user directs the system to save the best \( N \) solutions and corresponding fitness values \( \text{that it finds} \), thereby producing the BestNSaveSet for the run. These data may then be explored by the user, both graphically and in tabular form with filters of various sorts, in order to yield sensitivity information.

In a test implementation of the knapsack problem, we randomly assigned weights, \( w_i \), in the range \([1, 10]\) and profits, \( p_i \), in the range \([1, 20]\) for \( i \) from 1 to 30, i.e., \( n = 30 \). Since the 30 decision variables were binary, the problem had a basic search space of \( 2^{30} \).\(^{11}\)

Using an implementation of the model in the CLAP-NT system, we conducted a search using a genetic algorithm with 5,000 generations and a population size of 50, for a maximum of 250,000 = 5,000 \cdot 50 sample points. Thus, CLAP-NT examined at most \((250,000/2^{30}) \approx 0.023\%\) of the search space. In each of ten runs, CLAP-NT found the optimal solution. Moreover, it returned important sensitivity information. In what follows, we report on the run in which \( m = 80 \).

\(^{10}\)But see [2].

\(^{11}\)This is, of course, a small problem for current optimizers. Our focus is on using heuristics to find interesting feasible alternatives to the–or an–optimal solution of a model, and this is, we believe, something worth investigating even for comparatively small models.
18.2.1 Shadow Prices on \( m \)

The maximum value of \( z \) in our model, Expression 18.1, is constrained by the given value of \( m \). If \( m \) is changed, at what rate does \( \max z \) change? This information, called the shadow price on \( m \), is often invaluable for decision making. Let us see how CLAP-NT may produce shadow price information, i.e., heuristically-obtained information for estimating shadow prices.

The data in the BestNSaveSet reveal a difference of 7.9427 in fitness between the best solution found by the CLAP-NT system (and optimum for the model) and the second-best (see Table 18.1). The best found (and optimal) solution differs from the second-best solution (found by CLAP-NT) only in the value of \( x_{17} \), which is 1 in the best solution and 0 in the second-best. The weight required to carry the optimal mix of items whose total weight is less than or equal to 80 is 78.231. Should \( m \), the weight constraint of the model, drop below 78.231 yet remain above 77.095, which is required to carry the second-best-found mix, then the (estimated) optimal solution will shift to that found in the second-best solution. Thus, the interpretation of this information is that the (estimated) shadow price on \( m \) is 0 up to the allowable decrease of \( 7.9427 \) to 80; after that it has a shadow price of 7.9427 and a zero shadow price for the next \( 1.136 = 78.231 - 77.095 \) units of change.

All of this information is easily extractable, given the original objective function for the model (cf., Expression 18.1) and the data in the BestNSaveSet. We note that the shadow price information on \( m \) is available here only for decreases in \( m \), i.e., only for tightening, rather than relaxing, the constraint (but see §18.4, below). However, the number of (estimated) shadow prices for tightening of \( m \) is here determined (in large part) by the size of the BestNSaveSet. The data may (and in our test case, do) probabilistically yield useful results for many levels of tightening on \( m \).

18.2.2 Reduced Costs

The BestNSaveSet data (see Table 18.1) also yield (heuristic, estimated) insight into the reduced costs of several items in, or considered for, the knapsack. For example, the best and third-best vectors in the BestNSaveSet (ranks 1 and 3 in Table 18.1) have identical decision variable values, except that one includes \( x_{20} \), while the other instead includes \( x_{21} \). It turns out that \( x_{21} \) fills a larger chunk of the remaining space and increases the profit of the knapsack by 8.729 units. Thus, adding \( x_{20} \) to the knapsack creates a net opportunity cost—or reduced cost—of 8.729 and results in \( x_{21} \) being left behind.

We note that this reduced cost, or opportunity cost, information is also available for variables that are included in the optimal solution. For example, \( x_{27} \) is included in the optimal solution. What is the reduced cost of excluding \( x_{27} \)? From Table 18.1 we see that CLAP-NT (via the BestNSaveSet) estimates that the cost is \( z_{1} - z_{5} = 13.30 \).
18.3 Model Validation

CLAP-NT may be used to detect flaws, or suspect features, of models during the model validation phase of the modeling life-cycle. This aspect of CLA has received initial exploration for closed-form models [6] as well as for discrete-event simulation models [19]. More important than detecting absolute invalidity in a model, is the ability of CLA to ascertain ranges of validity and invalidity. Very many, otherwise quite valid and useful, models will give untoward results for certain joint ranges of parameter values. It is crucial for using such models that these ranges be detected and avoided—or at least recognized—in actual operations.

By way of illustration, consider the Basic Barrier Patrol model, which was developed for the U.S. Coast Guard in order to estimate the probability of detecting a target vessel attempting to cross a patrol barrier [15, 6].

The basic geometry of the patrol regime is given in Figure 18.1. Under reasonable assumptions, \( P(D|A) \), the probability of detecting a target vessel given that the Coast Guard vessel is available and on patrol is:

\[
P(D|A) = \frac{2 \cdot r}{l} \cdot \left(1 + \frac{v}{u}\right)
\]  

(18.2)

Given the intended interpretations of this model’s parameters—\( r, l, v, u \)—it would not be surprising if an invalid input value, for example a negative magnitude for one of the parameters, produced an anomalous value for \( P(D|A) \). What would be surprising, and certainly interesting, is if individually valid input parameter values produced anomalous values for \( P(D|A) \). A general procedure for attempting to find such interesting anomalies is as follows.

1. Define a measure of anomalyhood (or weirdness) for the model at hand.
2. Specify a (usually fairly broad) search space for all the parameters of interest.
3. Direct the CLA search process to find anomalous results in this search space.

John Miller, who independently developed some very similar ideas [19], has felicitously named this technique “model busting.” If anomalous results are found, i.e., if the technique breaks the model, then the analyst has work to do in order to understand and possibly revise the model or restrict its use. If no anomalies are found, after extensive search, then the analyst is given a reason to increase the stock of confidence in the model.

The CLAP-NT software supports model busting. The technique, and CLAP-NT’s support of it, is quite general and hardly limited to the Basic Barrier Patrol model. We use that model here only for illustration. Typical ranges for the model’s parameters in practice are: \( r \in [5, 20] \), \( l \in [75, 300] \), \( u \in [15, 35] \), and \( v \in [10, 30] \). We directed the genetic algorithm (GA) in CLAP-NT to search for high values of \( P(D|A) \) in the joint search space corresponding to these parameter ranges. In one run, we ran the GA for 100 generations with a population size of 50. We saved each individual in each generation, for a total of 5,000 solutions (although some 958 are duplicates). A large number of solutions was found—1,820—for which \( P(D|A) > 1 \). For example, CLAP-NT found that at (roughly) \( r = 19.9 \), \( l = 75.9 \), \( u = 15.3 \), and \( v = 30.0 \), the value of \( P(D|A) \) is 1.6. A very high probability indeed!

What has gone wrong? Further examination of the model’s equational form, Expression 18.2, reveals that these computational findings are not mathematically anomalous. There is nothing in the results to indicate that the implementation is incorrect. (Had it been the case, however, that the model was improperly implemented, then the model busting technique might well have helped diagnosed this condition.) Examination in CLAP-NT of the 5,000 saved solutions reveals quite reassuring behavior of the model: the expected directional changes occur (e.g., increasing \( v \) tends to increase \( P(D|A) \)), response of \( P(D|A) \) is generally smooth, and so on. In short, there is nothing in the data, other than \( P(D|A) > 1 \), that indicates a problem. This might suggest the hypothesis that the model is roughly accurate for parameter values resulting in \( P(D|A) < 1 \), and that higher values indicate a degree of overkill, or margin of safety. Because the Basic Barrier Patrol model is so simple algebraically, it is easy to confirm this hypothesis by adverting to the mathematics (Expression 18.2) and the geometry (Figure 18.1) of the model. In more complex cases, we can expect the benefits—of examining large data sets of model busting efforts—to be correspondingly greater.

### 18.4 Option Discovery

Models can support decision making, or the taking of a course of action, in two quite different but complementary ways. First, and standardly, a model may be used to find optimal or near-optimal courses of action under a given set of assumptions. This might be called the decision-oriented use of models. Here, key issues are model validity and sensitivity analysis. If we are to make a decision
18.4. OPTION DISCOVERY

based on the recommendations of the model, we need to be confident that the model is valid. Also, we need to explore for alternative recommendations that on balance might be better (hence, sensitivity analysis).

Under the second way of using a model, a model may be used to search for courses of action that may be taken in order best to alter the given set of assumptions. This might be called the *option-discovery-oriented* use of models. Here, the key issues are the costs and benefits of altering assumptions, particularly parameter values and constraints, that normally are assumed fixed for the purpose of solving the model. Given a set of assumptions, and a resulting set of decision options, often the best thing to do is to act so as to alter the assumptions. “It is,” as the saying goes, “better to light one candle than to curse the darkness.” As in the case of sensitivity analysis (§18.2, above), CLAP-NT provides general, heuristic approaches for option-discovery-oriented use of models. Note: Option-discovery-oriented use of models (our term) is standard practice as reported in the OR/MS literature, although this mode of using models is usually not distinguished terminologically from the technique usually employed to pursue it, sensitivity analysis. For further discussion of option-discovery use of models, see [15, 6].

We now take up an example—the ASSIST model and associated decision making—which has appeared in the general literature. We will rely extensively on the published record in order to describe the essentials of the decision problem [5, 8].

In response to the mounting human and economic toll from tobacco smoking, the National Cancer Institute (NCI) has, since 1982, funded a series of studies and experiments designed to find effective mechanisms for reducing smoking prevalence … These have culminated in the planning of the American Stop Smoking Intervention Study (ASSIST), the largest public health initiative ever undertaken by the National Institutes of Health. [5, page 1040]

ASSIST was given a budget of $114 million and states were invited to submit funding proposals that addressed the goals of the ASSIST program.

The process of issuing a complex request for proposal (RFP) and reviewing the resulting proposals defined an interesting decision problem: how to make awards among many competing contract proposals while balancing a number of considerations critical to the long-term viability and effectiveness of ASSIST. [5, page 1041]

Thus, NCI’s decision problem was framed as an optimization problem, and a 0–1 integer programming model was built (see [5, 8] for the—quite nontrivial—details). The model had 23 decision variables (1 fund, 0 do not fund), corresponding to the 23 proposals received (each from a distinct state). Objective function coefficients represented proposal ranks for technical merit, and “Other important criteria such as budget, diversity in smoking prevalence, and [sic]

---

12Motto of the Christopher Society.
diversity in decline in smoking rate, and diversity in geographical area,” were represented in the model as constraints [5, page 1046].

The published reports [5, 8] emphasize the extensive sensitivity analyses that were performed with the model.

The overall purpose in formulating and solving a model for the project funding problem is to provide a short list of solutions. Those solutions should be within, or very close to, the estimated budget figure, and also among the best available solutions with respect to total rank function and other important criteria. Decision makers at NCI will then have the option to select from among the recommended solutions [note: decision-oriented use], or seek additional information from the model [note: option-discovery-oriented use]. [5, page 1045]

A number of sensitivity analyses were performed, mainly by grid search: the model was repeatedly reoptimized/resolved across different ranges of parameter values. Perhaps the most interesting result from this process concerned the budget constraint.

Our initial goal had been to stay within a $114 million budget, and we found a very competitive solution at $114. However, when we examined [the results of the grid search] we found a superior solution for less than 0.5% over budget ($114.485). . . . This solution was quite attractive even when compared to other options with costs approaching $115.5 million.

Our model produced many solution sets. Solutions with the $114 million and $115.5 million constraints were shown to the decision-making team. The superiority of the $114.485 million solution, in terms of smokers reached and geographical balance, was obvious to the decision makers and thus contributed to this combination of states being awarded contracts. The selected solution set, however, was not an obvious one initially, and almost certainly would not have been discovered without the use of the model. [8, pages 119-120]

So we have here a testimonial to the value of post-solution analysis. How much of the achievement reported in the ASSIST papers could have been obtained via heuristic mapping of decision surfaces and the sort of software exemplified by CLAP-NT? Unfortunately, while the functional form of the ASSIST model is published [5, page 1051], confidentiality requirements prevent us from having access to the parameterization of the model. The papers do present a specific, scaled-down version of the model, using a notional parameterization, Expression 18.3. We have implemented this model in CLAP-NT and will now present our discussion in terms of it.\footnote{The parameterization in Expression 18.3 is not significant and was meant by the authors to be taken only for illustrative purposes. We note, in particular, that the constraints are variously redundant.} Note: In Expression 18.3 the right-hand-side value of the budget constraint is 37.
max \( z = \max_{x_1, \ldots, x_8} \) subject to
\[
\begin{align*}
6.3x_1 + 4.5x_2 + 8.0x_3 + 5.2x_4 + 4.7x_5 + 7.0x_6 + 9.1x_7 + 7.7x_8 & \leq 17.0 \\
9.8x_1 + 7.4x_2 + 4.9x_3 + 3.9x_4 + 8.1x_5 + 6.1x_6 + 7.3x_7 + 5.6x_8 & \leq 37.0 \\
x_1 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 & \leq 1
\end{align*}
\]

We implemented the model to incorporate a soft budget constraint (\( \leq 37 \)). The penalty we chose for violating the budget constraint was simply equal to a weight, \( w_1 \), times the absolute amount of constraint violation. Thus, for example, if a solution yielded an objective function value of 8.5 but had a budget of 37.7, then the net fitness would be 8.5 - \( w_1 \cdot (37.7 - 37.0) \).

Setting \( w_1 = 1 \), we ran the model with a genetic algorithm in CLAP-NT, using 50 generations and a population size of 50. The optimal (hard constraint) solution to the model (found both by an integer programming solver and by CLAP-NT) is to fund all proposals except \( x_1 \) and \( x_2 \), which happen to be the most-preferred projects according to the rank function (i.e., \( x_1 \) and \( x_2 \) have the highest objective function coefficients). Decision makers will naturally ask, “What happens as the budget constraint is relaxed?” CLAP-NT (with \( w_1 = 1 \)) also finds a good solution that slightly violates the budget constraint: fund \( x_2 \) through \( x_7 \). This yields an objective function value of 8.5 with a budget of 37.7. Thus, this solution produces an increase of \( 8.5 - 7.7 = 0.8 \) in the objective function value for a budget increase of \( 37.7 - 35.9 = 1.8 \). So, just as in the original ASSIST study, there is a happy solution just beyond the budget constraint.

One naturally also wonders what the consequences would be if the budget constraint could be tightened. This information is not reported in the ASSIST

---

14The preference function constraint (\( \geq 17 \)) turns out to be quite slack, so we will not discuss it further here.
15We also performed a number of runs in which the GA also searched on \( w_1 \), e.g., in the range \([1,2]\). The results corroborate the general findings we report here, but space limitations prevent us from giving details.
papers, and would have required different grid searches than those described in the papers, but it is easily available from CLAP-NT (see §18.2). As it happens, the optimal solution to the original problem (Expression 18.3) has an objective function value of 7.7 and a budget expenditure of 35.9. The next best found solution yields funding for \(x_1, x_2, x_4, x_5,\) and \(x_7,\) with an objective function value of 7.6 and a budget of 36.5. All of this (and other) information is easily and quickly obtained by using CLAP-NT to view and filter the BestNSaveSet for the model runs.

18.5 Reducing Infeasibility

An extreme case in which the need for an option-discovery-oriented use of models is appropriate is when an optimization model is infeasible due to conflicting requirements (rather than errors of implementation). Such events are far from rare. Standard optimization and model solution techniques are then pretty much at a loss as to what to do, unless, of course, the models are reformulated and solved as goal programs, a option not without difficulties of its own.\(^\text{16}\) CLA offers a general, computational, and heuristic approach to finding changes in assumptions that are attractive (or minimally unattractive) and that can (probabilistically) result in feasible solutions.

We take our example for this section from a goal programming case in a popular textbook by Ragsdale (Davis McKeown example, [20, pages 254-263]). Briefly, Davis McKeown is considering expanding his hotel and convention center and has purchased a consulting study.

The results of this study indicated that Davis’s facilities should include at least 5 small (400 square foot) conference rooms, 10 medium (750 square foot) conference rooms, and 15 large (1,050 square foot) conference rooms. Additionally, the marketing research firm indicated that if the expansion consisted of a total of 25,000 square feet, Davis would have the largest convention center among his competitors—which would be desirable for advertising purposes. [20, pages 254]

Small, medium, and large conference rooms cost $18,000, $33,000, and $45,150 respectively, and Davis has a budget of $1,000,000. Given this, we can formulate the optimization problem as in Expression 18.4.

The model, as formulated, is infeasible. A standard response, and that undertaken in Ragsdale [20], is to reformulate the model as a goal program. This results in a new objective function which here is linear, but involves weights that are chosen on largely subjective grounds. Ragsdale’s suggested objective function

\(^{16}\)We note, however, important work on certain aspects of this problem done by Harvey Greenberg [4] as well as others [3].
18.6. ON THE RÔLE OF THE GA

\[ \min z = 18000x_1 + 33000x_2 + 45150x_3 \]

where

\[ \begin{align*}
x_1 & \geq 5 \\
x_2 & \geq 10 \\
x_3 & \geq 15 \\
400x_1 + 750x_2 + 1050x_3 & \leq 25000 \\
1800x_1 + 3300x_2 + 45150x_3 & \in [0, 1, \ldots] \\
x_i & \in \{0, 1, \ldots\} \end{align*} \]

(18.4)

for the goal programming formulation is:

\[ \min z = \frac{w_1}{5}d_1^- + \frac{w_2}{10}d_2^- + \frac{w_3}{15}d_3^- + \frac{w_4}{25000}d_4^- + \frac{w_5}{25000}d_4^+ + \frac{w_6}{100000}d_5^- \]

(18.5)

where the \(d_i^\pm\) are the new decision variables and represent the amounts under or over (±) goal \(i\). The \(w_i\) weights are constants, but would normally be subject to the kind of sensitivity analysis by grid search we saw in §18.4. Solving this goal program produces a solution that is about 10% over budget and 1% over the square feet goal of 25,000, but that overwise meets the goals/contraints.

By now it should be clear how we would approach this problem with CLA concepts and CLAP-NT. We have a choice of either relaxing the right-hand-side goals in the original problem and performing search over appropriate ranges, or of solving the goal programming reformulation, but with ranges on the \(d_i^\pm\). In fact, we implemented the latter approach, and much as described in previous sections, we not only obtained the goal programming optimal solution, but a rich data set of sensitivity information.

18.6 On the Rôle of the GA

As we have seen, CLAP-NT uses a genetic algorithm (GA) to generate heuristic maps of the decision surfaces we investigate for post-evaluation analysis. Why use a GA? Are GAs fundamental to the CLA (heuristic mapping of decision surfaces) enterprise? On the latter question, the answer is clear and immediate: no. If a decision surface—or some relevant part of it—could be exhaustively searched, that would be better. The consequence of using a heuristic is that our results—shadow prices, etc., described above—can only be estimates. But for problems sufficiently large, such exhaustive enumeration is not practicable, and some heuristic will be required. Further, if heuristics are found for CLA that are better than genetic algorithms, then certainly one would want to use them. Then why use a GA?

GAs, for present purposes, have two important merits: (1) they are general; and (2) they work by finding good solutions and improving on them. Regarding (1), given a GA search engine, encoding a particular model for it is a fairly
straightforward thing, regardless of whether the model is linear or not, and whether or not the decision variables are integer. (This is not to say that the GA approach will be generally successful.) Regarding (2), GAs may be thought of as working via an implicitly parallel hill-climbing effort, which typically produces many good solutions. Our approach has been to save the best of these solutions for examination during post-evaluation analysis. The assumption has been that for decision making the most interesting alternatives to the optimal solution are those that are also reasonably close to being optimal. For many purposes, we think this is a reasonable assumption, but to the extent that it is not, the GA approach would seem to be unpromising.

These two reasons, however, are only plausibility reasons. They are only reasons for exploring use of GAs for post-evaluation analysis. It is hard to see why this should not be done. But what is more important is (a) to define appropriate goodness, or success, criteria for CLA (heuristic mapping of decision surfaces), and (b) to determine, either analytically or computationally, which heuristic approaches are most likely to best satisfy these criteria in a given situation.17 We have begun such investigations, and here offer some preliminary thoughts and initial results.

Here are two criteria (really criteria schema) that answer to (a). Suppose we have two heuristics for generating decision surface maps, \( h_1 \) and \( h_2 \). Then, for a given level of computational effort, \( h_1 \) is better than \( h_2 \) if

**Best N criterion:** The \( N \) best distinct solutions found by \( h_1 \) have associated objective function values that are generally closer to the optimum than are the \( N \) best distinct solutions found by \( h_2 \).

**Best within P criterion:** There are more and better distinct solutions found by \( h_1 \) than by \( h_2 \) that are within \( P \) percent of the optimum objective function value.

Using the Best N criterion, we performed computational experiments to test the CLAP-NT GA against a random search for 12 randomly-generated knapsack problems. Our measure of computational effort was the number of function (or fitness) evaluations performed. Thus, for example, a GA with population size 100, run for 50 generations, requires 5,000 fitness evaluations. This was, we assumed, computationally equivalent to randomly generating 5000 solutions (which may or may not be feasible) and determining their values. (Our findings are not very sensitive to this assumption.) The randomly generated solutions used the same variable ranges that the GA used, so the spaces searched were identical.

In the experiments we are reporting here, \( N \) was 30. Of our 12 knapsack problems, 3 had 20 decision variables, 3 had 25, 3 had 30 and 3 had 35. Fixing the decision variable count, we randomly generated 3 problems for each of the 4 cases (numbers of decision variables). Finally, we had two settings for computational effort: 5,000 function evaluations and 10,000 evaluations. In all, then, we had 48 runs: 12 models \( \times \) 2 heuristics \( \times \) 2 levels of computational effort.

---

17For very thoughtful, and useful, general surveys on assessment and measurement of contributions to decision making made by a system such as CLAP-NT see [1] and [9].
18.7. SUMMARY

As should be expected, the GA generally performed better, at least in a gross sense. Table 18.2 summarizes the results. In all 12 cases, the GA averages better than the random search in the sense that the best 30 distinct solutions it finds have objective function values on average higher than their counterparts from the random search. In fact, the average maximum value from the random searches is just slightly higher than the average minimum (i.e., 30th best) value found by the GA searches.

These computational results are hardly definitive. The performance of this GA versus this random search is hardly stunning, yet there does seem to be some advantage, provided we find the Best N criterion a relevant one. Clearly, much remains to be done if we are to find the most effective heuristic for mapping decision surfaces.

18.7 Summary

There is more to the story of candle-lighting analysis, but we may summarize the main points of this paper as follows:

1. Optimization problems, ubiquitously encountered, present to the decision maker objective function hypersurfaces—decision surfaces—that are typically rugged and complex, and that need to be explored in order to support high-quality decisions.

2. If these objective function hypersurfaces could be fully mapped and the resulting information made effectively available to the decision maker, then a rich, if not complete, body of information for post-solution analysis would be available to the decision maker.

3. It is generally not possible, for realistic problems, to produce anything approaching a complete mapping of an objective function hypersurface. Computational complexity forbids this.

4. Short of a full mapping of a decision surface, it may be hoped that a partial mapping, based upon intelligently-directed heuristics, can reliably produce much valuable information, and be generally superior to alternative methods for post-solution analysis (e.g., manually-directed what-if questioning, goal-seeking, grid search, and so forth).

5. Genetic algorithms are a promising form of heuristic for mapping decision surfaces for purposes of post-evaluation analysis. Other heuristics need to be investigated.

6. Much work—both conceptual and computational—remains to be done in order to understand adequately effective means of generating and exploring decision surfaces.

---

18See [6] and other CLA references for additional details.
Table 18.1: From BestNSaveSet: Top solutions found by CLAP-NT for test knapsack problem with $m = 80$ and $n = 30$. Values for $w_i$ and $p_i$ were randomly drawn. Found values of $z$: $z_1 = 274.93$, $z_2 = 266.99$, $z_3 = 266.20$, $z_4 = 261.66$, $z_5 = 261.63$. 

| $x_i$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | $x_9$ | $x_{10}$ | $x_{11}$ | $x_{12}$ | $x_{13}$ | $x_{14}$ | $x_{15}$ | $x_{16}$ | $x_{17}$ | $x_{18}$ | $x_{19}$ | $x_{20}$ | $x_{21}$ | $x_{22}$ | $x_{23}$ | $x_{24}$ | $x_{25}$ | $x_{26}$ | $x_{27}$ | $x_{28}$ | $x_{29}$ | $x_{30}$ |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
|       | 1     | 1     | 1     | 1     | 1     | 0     | 0     | 0     | 0     | 1     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     |
| $z_1$ | 1     | 1     | 1     | 1     | 1     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     | 0     |
Table 18.2: Summary statistics for the 30 best distinct results on runs of 12 random knapsack models, for the CLAP-NT GA (population size = 100) versus a random search in the same space. Max and Min values are averaged across the 30 best distinct solutions found in each of two runs, at 5,000 and 10,000 function evaluations.
Bibliography


18.8 Version Notes

Chapter 19

Case 5, VB: Instructions,
OPIM 101, Spring 1996

Your assignment in this case is to modify a simple application template, written
in Excel and Visual Basic for Applications (Excel VBA, or just VBA when
the context removes any ambiguity). In doing so, you are to produce a more
satisfactory application, per our instructions, below. The file name for the
template that you are to modify is gavbstub.xls. (If subsequent versions are
necessary, they will be named gavbstba.xls, gavbstbb.xls, and so on.)

The goals of this case are to provide you with some practical exposure to
VBA, to acquaint you with some principles and issues for good application
design, to give you exposure to using genetic algorithms (GAs) to solve problems
in management, and to give you some representative experience in dealing with
“real world” programming problems. You will not need to master the details of
the GA code we give you, although the more you learn about it the better.

Your assignment is divided into a series of tasks, as follows.

19.1 Task 1: Obtain the GA Run Parameters
Interactively

The template, gavbstub.xls, has one worksheet, “Run GA,” and one VBA code
module, “GA Code,” in addition to the “Answer Sheet” and “Grade Sheet”
worksheets. The “Run GA” worksheet has a button on it, entitled “Do the
GA.” When you click this button GA code from the code module is executed,
using the Basic Barrier Patrol model as a fitness function (see the sub Evaluate).
(The purpose of this model is to estimate the probability that a patrol vessel
will sight a target vessel, e.g., drug dealer, as it moves across the patrol area.
You do not need to know the details of this model, for the purpose of this case.)

One generation of population size 10 is run, with a crossover rate of 0.77 and
a mutation rate of 0.23. Output is written to two files. c1g1.txt is a text file that
contains output from the GA run, giving the current generation (generation 1). b1f1.txt is a text file that contains output from the GA run, giving the best 5 solutions and their fitnesses for the current generation (generation 1). (See the sub, PostpareGA.)

All these values—1 (generation) 10 (population size), 0.77 (crossover rate), 0.23 (mutation rate), and 5 (best solutions found)—are "hardwired" in the GA code, in the declarations section (at the top). This shouldn’t be. Instead, a user should be able to set these values interactively. Your job in this task is to change the template, gavbstub.xls, so that this happens.

Specifically, you should modify the workbook (gavbstub.xls) so that these five GA run parameters are set interactively in the following way. The sub, GetGARunPars, which is now mainly a stub, should read these parameter values from the "Run GA" worksheet, in cells C7 and below, and set them. The user should then be able to change them by changing values in these cells and clicking on the "Do the GA" button.

Pay good attention to formatting and programming style. Regarding formatting, the cells that the user can change (and that your program reads from) should be colored green. Everything should be labelled and the entire area should be headed with "GA Run Parameters" in bold and underlined. Regarding style, you should name the cells for each of these five parameters, and should use a prefix of "ws." For example, if your program variable is NumberOfGenerations, then the cell holding this value should be named ws.NumberOfGenerations, and you should refer to the cell in your VBA code using the name of the cell, not the address (e.g., B52).

19.2 Task 2: Obtain the Model Parameters Interactively

This task closely resembles task 1, in that we are replacing “hardwired” elements in the template code with a facility to allow users to input values interactively. Here, we need to set parameter values for the model itself, rather than the GA code that uses the model (for its fitness function).

The model parameters that need to be initialized are NumberOfDecisionVariables and OutputSize, as well as the high and low values for each of the model’s decision variables. (See the sub GetModelRunPars.) In the Basic Barrier Patrol model there are four decision variables:

1. r, the radius of detection for the patrolling ship’s radar
2. v, the speed of the patrolling ship
3. u, the speed of the target vessel
4. l, the length of the patrol barrier
(See the Evaluate sub for details. Note that absolute fitness is calculated and stored twice. This is simply to demonstrate that a fitness function can, in this code, return more than one value. Hence, OutputSize is set to 2.)

The high and low permitted values for these four variables are stored in the program array DecisionVariableInfo. Using the principles of good layout and program design (e.g., use named ranges) discussed in task 1, modify the template so that the information needed to populate DecisionVariableInfo and to set OutputSize and NumberOfDecisionVariables is read from a worksheet, and so that the user may conveniently change these values.

Specifically, add a new worksheet immediately after the “Run GA” worksheet. Call it “Model Inputs” and give it the same (light blue) color that the “Run GA” worksheet has. Then lay out the input information for the Basic Barrier Patrol model in a clear and pleasing way. The four decision variables should be clearly labelled:

1. Radius of Detection
2. Speed of Patrol
3. Speed of Target
4. Length of Patrol

Similarly, the other two inputs should be clearly labelled as well.

### 19.3 Task 3: Place Outputs in the Workbook

In the template, program output is written to two files. See the sub, PostpareGA. One of the files contains the current generation of the GA (with fitness numbers). The other file contains the best n (= BestNSaved) solutions found by the GA, over however many generations it has run.

Instead of writing this information to two different files, write it to two different worksheets in the workbook. Specifically, add two worksheets to your Excel workbook, “Best Finds” and “Final Generation,” putting them immediately after the “Model Inputs” worksheet. Modify the program so that instead of writing to the two files, the output is sent to these two new worksheets. Follow the basic format of the existing file output, in terms of column order and meaning, but begin writing on row 11 of the output worksheets.

Add a sub to clear the two worksheets entirely at the start of each new run. (Be sure to tell us the name of this sub and where it is called.)

### 19.4 Task 4: Label and Format the Output Sheets

There are two parts to this task.

First, you are to add code that, each time the GA is run, puts column labels in row 10 of each of the two output worksheets. Specifically:
1. For the “Best Finds” worksheet and the Basic Barrier Patrol model, column A should be labelled ID, column B should be labeled Radius of Detection, column C should be labelled Speed of Patrol, column D should be labelled Speed of Target, column E should be labelled Length of Patrol. Each of these labels (for B-E) should not be hardwired in your program. Instead, they should be read off of the “Model Inputs” worksheet. Columns F and G (for the Basic Barrier Patrol model) should be labelled Absolute Fitness. The labels for columns A, F and G can be hardwired in your program.

2. For the “Final Generation” worksheet, the labels should be as in the “Best Finds” worksheet, expect that column H should be labelled Absolute Fitness.

Note: Be sure to indicate in your answer the subroutine in which you put this code.

Hint: Design for generality. What if, as below, you need to handle different models with different output sizes, different numbers of input variables?

Second, you should add a sub to your code that automatically resizes the column widths in the two output worksheets, so that the columns are wide enough to display the labels completely. This sub should be called automatically with each run of the GA.

19.5 Task 5: Report Progress Information

Report progress of the GA run on the “Run GA” worksheet. Do this by keeping a running tally of the number of generations processed in cell C2 of that worksheet. Specifically, set the count to 0 before the sub PrepareGA is called and set it to 1 immediately after. Increment the counter as, in the sub RunGAUntilDone, the generations are completed. Put an appropriate label in the cell C1. The effect should be that after the user clicks the “Do the GA” button, a running count is displayed in C2 of the number of completed generations.

19.6 Task 6: Add a New Model

The application you have now developed has a reasonable set of features, but it can handle only one model at a time, here the Basic Barrier Patrol model. Your job in this task is to add the capability to hand a second model, specifically an inventory model, which is specified in Appendix A (below). In doing so, you are to reuse as much of the existing code as possible and you are to retain the current feature set.

Here’s what you need to do.

First, you need to set up a list box that will prompt the user to pick which model is to be run. The two models are “Basic Barrier Patrol” and “Inventory
Opportunity.” Based on the user’s response a global program variable, ModelNumber (an integer), is set. The user should also have the opportunity to end the program execution by pressing the “Cancel” button on the dialog box eliciting the model number.

Second, you need to set up choice points at several places in the code, so that, e.g., the model inputs are taken from the worksheet, “Model Inputs,” for the appropriate model. You should use a “Select Case” construction for this. You will need to do this at three points:

1. Getting the model inputs (see the sub GetModelRunPars)
2. Calling the model for evaluation (see the sub CalculateFitness)
3. Writing the results to the output worksheets (see the sub PostpareGA)

Note:

1. The sub Evaluate in the template is called for evaluating the Basic Barrier Patrol model. You will need to add a new sub to evaluate the Inventory Opportunity model. Call this sub EvaluateInvOp.

2. The GA code in the template only works for maximizing a positive-valued function. Here, you are asked to minimize a positive-valued function. To circumvent this problem, and to avoid making changes to the guts of the GA code, you should do the following. As with the Basic Barrier Patrol model, the OutputSize should be set to 2 and the AbsoluteFitness array will then have two columns. (See the Evaluate sub for an example.) The first column of AbsoluteFitness is what the GA code uses to direct evolution. You should store in this column what we will call the Scaled Absolute Fitness. That is, the true absolute fitness is $I_T$ (see Appendix A) and this goes in column 2 of the AbsoluteFitness array. In column 1 put $1,000,000 - I_T$. This will be a positive number that the GA code can maximize and in doing so will minimize $I_T$.

3. You will need to change your output routines so that the columns in the tables in the “Best Finds” and “Final Generation” tables are properly labelled.

4. In the “Model Inputs” worksheet, you should add information about the Inventory Opportunity model that the GA can read it in, as it now does for the Basic Barrier Patrol model. Specifically, there will be 5 input variables: $C_o, D, Q, C_c, r$ (see Appendix A). To test your implementation, you should set the high and low values identically for all variables except $Q$, which should range from 500 to 1000. If you then run it for the values indicated in Appendix A, you should be able to find the approximate value of $Q^*$. 

5. If input variables range too broadly, e.g., so that certain quantities in the model go negative when they should not, then the GA code will produce
weird results. Stay reasonably close to the input values discussed in these instructions.

6. Again: duplicate the features, functionality, and style we used with the Basic Barrier Patrol model.

19.7 Task 7: Respond to an Opportunity

Having completed the previous tasks, you are in possession of a useful tool. You have determined $Q^*$ and the corresponding $I_T$ under the assumptions given in Appendix A. Now, suppose you learn that your supplier is somewhat flexible in committing to a delivery schedule. In fact, the supplier wants to commit to a single value of $r$ (now at 950), but is indifferent between any value of $r$ ranging from 700 to 1250. Your task is to perform an analysis of this opportunity and recommend what value of $r$ your organization should select and given that, what the new $Q^*$ and $I_T$ will be. Provide a brief analysis, including a discussion of sensitivities.

Your boss is also concerned about how sensitive the model is to minor changes to the fixed input assumptions, e.g., that $D = 100,000$. In a brief memo to your boss, describe how you might use your nifty new program to investigate these concerns.

19.8 Hints and Suggestions

1. Be sure to document everything: tell us how you did each task, where the code is, etc. Put all this on the “Answer Sheet,” associated with the tasks in which you made the changes.

2. If cell A1 is named wsbob, then the following expression reads the value of that cell into the VBA program variable, bob:

   ```vba
   bob = Range("wsbob") . Value.
   ```

3. Option Explicit is declared in the template. It should remain declared in all your modifications. That is, you must declare all your variables. Also, do not use the variant data type unless it is absolutely necessary.

4. Each time the program is run from scratch, e.g., by launching Excel and clicking the “Do the GA” button, the random number generator is seeded with the same random number (in spite of the Excel and VBA documentation!). This allows you to test your program, since, e.g., minor modifications in how settings of parameters are done should not affect the values of the outputs. But you have to exit and restart Excel to see this. (The documentation on Randomize is incorrect.)

5. A listing of the GA template code (or something very close to it) is given in your bulkpack. Also, your bulkpack discusses the main elements and
strategy of the code. Online, pressing F2 when the code module is active is a very useful thing to do. In class, we will cover such important topics and concepts as ReDim, how the GA code works in general, and so on.

19.9 Grading Policy

1. Tasks 1-4, 2 points each.

2. Task 5, 1 point.

3. Tasks 6-7, 4 points each.

19.10 An Inventory Opportunity

Here we work with a particular, fairly simple, but widely used version of the EOQ (economic order quantity) model, called the noninstantaneous receipt model (under deterministic and constant demand). EOQ models are widely discussed in management science textbooks and many good descriptions are available, but the discussion here stands by itself.

A main purpose of the EOQ model is to determine the optimal amount of inventory to order in a given ordering instance. The EOQ model seeks to balance the ordering costs with the holding, or carrying, costs of inventory. The basic model is:

\[
I_T = C_o \cdot \frac{D}{Q} + C_c \cdot \frac{Q}{2} \left(1 - \frac{d}{r}\right) \tag{19.1}
\]

where

\(I_T\) Total annual inventory cost—which we wish to minimize

\(C_o\) The cost of placing a single order for inventory—a constant, which does not vary with the amount of inventory ordered

\(D\) Total annual demand, in units

\(Q\) The number of units ordered per ordering instance—this is normally the decision variable for the model

\(C_c\) The annual carrying cost per unit in inventory

\(d\) The daily rate of demand—equal to \(D/365\).

\(r\) The daily rate of replenishment of inventory—after inventory has been ordered, this is the rate at which it arrives.

\footnote{For the record, we are following the treatment in \textit{Introduction to Management Science, 3rd ed.}, by Bernard W. Taylor, III (Allyn and Bacon, Boston, 1990, chapter 17).}
Thus, we can express our model in the form of an optimization (now, minimization) problem, as follows.

\[
\text{Choose } Q \text{ so as to } \min \ z = I_T = C_o \cdot \frac{D}{Q} + C_c \cdot \frac{Q}{2} \cdot \left(1 - \frac{d}{r}\right) \\
\text{subject to } Q \geq 0 \quad (19.2)
\]

Note:

1. We assume with this model that all quantities are known and constant over the time horizon of the model.

2. We assume that inventory ordered does not necessarily arrive instantaneously. Instead, it arrives at a rate captured by \( r \). New inventory must typically be ordered before the current stock is entirely empty.

3. The maximum amount of inventory on hand is consequently

\[
Q_{max} = Q - \frac{Q}{r} \cdot d
\]

4. This is a nonlinear optimization problem, but because of the simple constraint structure and the fact that the objective function is continuous, the problem can be solved by standard techniques in the calculus. \( Q^* \), the optimal ordering quantity, is:

\[
Q^* = \sqrt{\frac{2C_oD}{C_c(1 - d/r)}}
\]

Assume, further, that we have the parameter values before us given in table 19.1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_o )</td>
<td>$1,500</td>
</tr>
<tr>
<td>( D )</td>
<td>100,000</td>
</tr>
<tr>
<td>( C_c )</td>
<td>$750</td>
</tr>
<tr>
<td>( r )</td>
<td>950</td>
</tr>
</tbody>
</table>

Table 19.1: Assumed Parameter Values

For these data:

1. \( Q^* \approx 750 \).
2. \( Q_{max} \approx 534 \).
3. \( I_T \approx $400,140 \).
19.11 Version Notes

Chapter 20

BasicGA: Code for Genetic Algorithms

Note April 8, 1998: This listing is only approximate. I have a running version of the code in the Excel98 file gavbsolexcel98.xls, which is the solution to the programming case for DOPIM 101, spring 1996.

20.1 Introduction

The purpose of this appendix is to lay out and discuss the code for BasicGA. BasicGA is a program having some (very) basic genetic algorithm capabilities. It was written in Visual Basic (Microsoft) and works in the Visual Basic for Applications (VBA) dialect. BasicGA works in Excel. The purpose of BasicGA is to serve as a shell or starting point for developing applications using genetic algorithms, especially in a classroom environment. My intention, thus, has been to make BasicGA as implementation-independent as possible.

Note: For program development, debugging, and other purposes, I have often substituted a stub routine in BasicGA for what would be an actual routine in a full application. Such stubs will have “stub” appended to the actual name, sometimes prefixed and sometimes postfixed. For example, the stub for InitializeGA is InitializeGAStub but could be StubInitializeGA. Also, in this documentation, names of code objects, e.g., procedures and variables, will be given in a typewriter font, as we have just seen with InitializeGA.

20.2 Declarations

The purpose of this section is to describe the declared, global or public, variables for BasicGA. Here follows the declarations section of the BasicGA code. It is

\footnote{A stub is a procedure that is intendedly not in final form, but is used during program development and testing.}
written in Microsoft Visual Basic 3.0 and has also been tested in the Microsoft Excel 5.0 environment.

'sok 951127: This is in file: GACODE.BAS in
'the folder clapopo3
'Am freezing this for now. Call it the BasicGA program,
'version 951127.

Option Explicit

'**************************************************
'*************** Below, constants declared that ***
'*************** should be read in. ***************

'sok 951126: Note: These program variables,
'in a non-stubbed
'environment, need to be declared in the declarations
'section. They are so declared, but I have commented
'out the declarations (see below).

'+++++++++++++++++++++++
'++++ from GetGARunPars ++++++++

Const NumberOfGenerations = 20
'GetNumberOfGenerations
Const PopulationSize = 100
'GetPopulationSize
Const CrossoverRate = .77
'GetCrossoverRate
Const MutationRate = .23
'GetMutationRate
Const bestNSaved = 100
'GetBestNSaved
' ++++++++++++++++++++++++++++++++
' +++++ from GetModelRunPars ++++++++
'Const NumberOfDecisionVariables = 4

'GetOutputSize
Const OutputSize = 2

' ++++++++++++++++++++++++

' ++++++++++++++++++++++++
' +++++ from/for InitDVarInfo/StubInitDVarInfo +++++
Dim DecisionVariableInfo(1 To NumberOfDecisionVariables, => 1 To 4) As Double

Const DecisionVariableInfo11 = 5 'r, low
Const DecisionVariableInfo12 = 20 'r, high
Const DecisionVariableInfo13 = 0 'r, not integer
Const DecisionVariableInfo14 = 0 'r, no grid search

Const DecisionVariableInfo21 = 10 'v, low
Const DecisionVariableInfo22 = 30 'v, high
Const DecisionVariableInfo23 = 0 'v, not integer
Const DecisionVariableInfo24 = 0 'v, no grid search

Const DecisionVariableInfo31 = 15 'u, low
Const DecisionVariableInfo32 = 25 'u, high
Const DecisionVariableInfo33 = 0 'u, not integer
Const DecisionVariableInfo34 = 0 'u, no grid search

Const DecisionVariableInfo41 = 200 'l, low
Const DecisionVariableInfo42 = 300 'l, high
Const DecisionVariableInfo43 = 0 'l, not integer
Const DecisionVariableInfo44 = 0 'l, no grid search

' +++++++++++++++++++++++++++++++++++++
'**************************************************
'*************** Above, constants declared that ***
'*************** should be read in. ***************

' Global variables

'++++ from GetGARunPars
' **** but explicitly declared above ++++++++8

' Dim NumberOfGenerations as Integer
' Dim PopulationSize as Integer
' Dim CrossoverRate As Double
' Dim MutationRate As Double
' Dim bestNSaved as Integer

' ++++++++++ from GetModelRunPars
' ++++++++but explicitly declared above +++++++++++

' Dim NumberOfDecisionVariables As Integer
' Dim OutputSize As Integer

' +++++++++++++++++++++++++++++++++++++++++++++++++++++++
Dim Index  As Integer
Global CurrentGeneration() As Double
Global AbsoluteFitness() As Double
Dim ChromosomeCopySpace() As Double
Dim RelativeFitness() As Double
Dim CrossoverLikelihood() As Double
Dim BestNCURRENTSavEset() As Double
Dim LowestAbsoluteFitness As Double
Dim HighestAbsoluteFitness As Double
Dim CurrentIdNum As Double
Dim NumberOfGenerationsSoFar As Integer
Dim CrossoverPoint As Integer
Dim NoisyOutput As Integer ' 1 = show lots of output;
'0 = don’t

The general structure and plan for the program is simple. Everything re-
volves around two arrays.

First, the array CurrentGeneration holds the current generation of chromo-
somes, one chromosome per row. CurrentGeneration has rows running from
1 to PopulationSize, where PopulationSize is the number of individuals or
chromosomes maintained in each generation. CurrentGeneration has columns
running from 0 to NumberOfDecisionVariables, where
NumberOfDecisionVariables is the number of variables at play in the model
for the GA runs. Column 0 of CurrentGeneration holds the ID of the corre-
sponding chromosome.

Second, the array AbsoluteFitness holds the results of the fitness evalua-
tions for each chromosome in the current generation. AbsoluteFitness has
rows running from 1 to PopulationSize and a row of AbsoluteFitness cor-
sponds to a row of CurrentGeneration. AbsoluteFitness has columns from 1
to OutputSize, where OutputSize is the number of distinct values returned for
a single chromosome by evaluation of the fitness function. Usually, OutputSize
will equal 1, that is, only 1 value is returned: the absolute fitness of the chro-
mosome at hand. Sometimes, however, it is useful to have the fitness function
return several values. If so, then their number is indicated by OutputSize and
it is the responsibility of the fitness function, Sub Evaluate(I), to organize the
response. By convention, column 1 of AbsoluteFitness must hold the absolute
(or raw) fitness of the chromosome at hand.

BasicGA works by initializing CurrentGeneration, calculating fitnesses with
Evaluate(I) and thereby populating AbsoluteFitness. Then the next gen-
eration is created. Crossingover is performed, mutation is performed, and the
cycle continues until the stopping condition (a count of the generations in this
code) is encountered. All this mostly happens through Sub RunGAtillDone.

Now some specific comments about these declarations.

1. The following parameter is set in InitializeGA:
20.2. DECLARATIONS

(a) **CurrentIDNum.** An integer, representing the ID number, or count, of a given chromosome or solution.

2. The following parameters are set in **GetGARunPars:**

(a) **NumberOfGenerations.** Integer, should be $\geq 0$.
(b) **PopulationSize.** Integer, should be $\geq 1$.
(c) **CrossoverRate.** Floating point, should be $\in [0,1]$.
(d) **MutationRate.** Floating point, should be $\in [0,1]$.
(e) **BestNSaved.** Integer, should be $\geq 0$.

3. The following parameters are set in **GetGAModelRunPars:**

(a) **NumberOfDecisionVariables.** Integer, should be $\geq 1$. This is the number of input variables sent to the fitness evaluation function.
(b) **OutputSize.** Integer, should be $\geq 1$. This is the number of output values returned by the fitness evaluation function.

4. The following parameters are set in **ReDimGAArrays:**

(a) **CurrentGeneration.** Declared here as nonstatic, i.e.,
   Dim CurrentGeneration() As Double.
(b) **AbsoluteFitness.** Declared here as nonstatic, i.e.,
   Dim AbsoluteFitness() As Double.
(c) **RelativeFitness.** Declared here as nonstatic, i.e.,
   Dim RelativeFitness() As Double.
(d) **BestNCurrentSaveSet.** Declared here as nonstatic, i.e.,
   Dim BestNCurrentSaveSet() As Double.
20.3 Sub DoTheGA: Code Structure Overview

Sub DoTheGA is the intended entry point to this program. Its structure is quite simple and the source code is given in Figure 20.1.

'S **************************** Main Program ******
', Sub DoTheGA ()

Randomize (17)
ChDir "c:\clasave\"
NoisyOutput = 1

'1. Make preparations to run the GA.
 PrepareGA

' 2. Run the GA until the stopping condition is met
 RunGAUntilDone

' 3. Postpare the system
 PostpareGA
End Sub

Figure 20.1: Sub DoTheGA Source Code: Main Entry Point

A few comments are in order. The purpose of Randomize (17) is to initialize the random number generator. This guarantees that on each run the same sequence of random numbers will be generated, regardless of which machine the program is run on.

ChDir "c:\clasave\"

is for the IBM PC (MS DOS) environment and will need to be changed or commented out on the Macintosh. It assumes that a directory called clasave exists on the C drive. The program writes its output files to this directory.

NoisyOutput is set to 1, turning on various comments during the running of the program. Set it to 0 to turn these off.

Now, briefly, to the three subroutines called in Sub DoTheGA.
20.3.1 PrepareGA

The purpose of this subroutine is to initialize the program and to generate the first generation of chromosomes. The source code for this subroutine is given in Figure 20.2.

```vba
Sub PrepareGA ()
    ' 1. Initialize the system
    InitializeGA

    ' 2. Validate the input data
    ValidateGAINput

    ' 3. Generate the initial population of chromosomes
    MakeGAGenOne

    ' 4. Calculate the absolute and relative fitnesses for each chromosome.
    CalculateFitness

    ' 5. Initialize the save sets
    InitializeSaveSets
End Sub
```

Figure 20.2: Sub PrepareGA Source Code
20.3.2 RunGAUntilDone

This is the subroutine that does the main work in the program. Its source code is given in Figure 20.3.

Sub RunGAUntilDone ()
Do Until NumberOfGenerationsSoFar >= NumberOfGenerations
   If (NoisyOutput = 1) Then
      MainForm.ProgressBox.Text = "NumberOfGenerationsSoFar = " & NumberOfGenerationsSoFar
   End If
   ' Now to the main business:
   PerformCrossover
   PerformMutation
   CalculateFitness
   UpdateTheSaveSets
   SortBestNCurrentSaveSet
   NumberOfGenerationsSoFar = NumberOfGenerationsSoFar + 1
Loop
   If (NoisyOutput = 1) Then
      MainForm.ProgressBox.Text = "NumberOfGenerationsSoFar = " & NumberOfGenerationsSoFar
   End If
End Sub

Figure 20.3: Sub RunGAUntilDone Source Code. Note: Lines artificially broken with my continuation symbol: ==>.
20.3.3 PostpareGA

Sub PostpareGA cleans things up once the GA has run its course. The program does two things: writes out CurrentGeneration to a file and writes out BestNCurrentSaveSet (the array holding the best N chromosomes found to this point in the GA run) to a file. The source code is given in Figure 20.4.

Sub PostpareGA()

' Print out final generation.
Print2FileCurGen
' Print out the best finds overall.
Print2FileBestOverall

If (NoisyOutput = 1) Then
' MainForm.ProgressBox.Text = "All done."
End If
End Sub

Figure 20.4: Sub PostpareGA Source Code
20.4 PrepareGA: Detailed Code Structure

20.4.1 InitializeGA

InitializeGA initializes CurrentIDNum to 0, then calls three subroutines. The first, GetGARunPars, is for obtaining information needed to make this run of the GA. The second, GetGAModelRunPars, is for obtaining particular information about the model (fitness function) that is to be applied in this particular run of the GA.

The third, ReDimGAArrays, only has the function of setting the sizes of various dynamic arrays (see declarations section, above).

1. CurrentGeneration(1 To PopulationSize, 0 to NumberOfDecisionVariables) As Double.
2. AbsoluteFitness(1 To PopulationSize, 1 To OutputSize) As Double.
3. RelativeFitness(1 To PopulationSize) As Double.
4. BestNCurrentSaveSet(1 To BestNSaved + PopulationSize, 1 To NumberOfDecisionVariables + 1 + OutputSize) As Double.

20.4.1.1 GetGARunPars

The following program variables need to be initialized in this subroutine:

1. NumberOfGenerations.
3. CrossoverRate.
4. MutationRate.
5. BestNSaved.

20.4.1.2 GetGAModelRunPars

The following program variables need to be initialized in this subroutine:

1. NumberOfDecisionVariables.
2. OutputSize.

In addition the following array must be initialized:

1. DecisionVariableInfo.

Specifically,

ReDim DecisionVariableInfo(1 to _
   NumberOfDecisionVariables, 1 to 4) As Double.
should be declared and `DecisionVariableInfo` initialized.

In `DecisionVariableInfo` each row corresponds to a decision variable. Column 1 holds the LowValue, column 2 the HighValue for the row’s variable. Column 3 is 0 if the variable is not required to be an integer, and 1 otherwise. Finally, column 4 holds grid search information. (BasicGA does not have any grid search capability, but is designed to be expanded.) A 1 indicates that no grid search is being done on that variable. A number larger than 1 indicates that if a grid search is to be done, then the number represents the number of grid points to be examined for that variable. The array holds floating point numbers, and grid search counts are integers. It is up to the grid search program to make the conversion. By convention, we truncate, e.g., 3.1 stored goes to 3.

### 20.4.2 ValidateGAINput

The purpose of this subroutine is to validate the information collected in the `InitializeGA` subroutine. In the current version of the software, little or nothing is done here. Beware!

### 20.4.3 MakeGAGenOne

Declare:

```vbscript
ReDim CurrentGeneration(1 to PopulationSize, 0 to NumberOfDecisionVariables) As Double
```

Each row holds a chromosome of the current generation. Columns 1 through `NumberOfDecisionVariables` hold values for the corresponding decision variables. Column 0 holds the ID number of the solution.

This subroutine is very simple. It merely uses `DecisionVariableInfo` to load up `CurrentGeneration`, with the aid of a random number generator. Also, each member of the generation (i.e., each row) is given a unique ID.

### 20.4.4 CalculateFitness

This routine calls `Evaluate(I)` for each member (row) of `CurrentGeneration`. `Evaluate(I)` then calculates the fitness of that row and stores it in `AbsoluteFitness`. By convention, the first column of `AbsoluteFitness` is the absolute fitness of the corresponding row or solution. If the fitness function, `Evaluate(I)`, returns more than one value, additional values are stored in the second, third, and so on columns of `AbsoluteFitness`.

Following this, `CalculateRelativeFitness` is called, which calculates the relative fitnesses from the absolute fitnesses and stores them in `RelativeFitness`, a one-dimensional array.

### 20.4.5 InitializeSaveSets

In the basic program, only one save set is used. `BestNCurrentSaveSet` stores the best `N` solutions so far, plus the current generation. In this subroutine,
CurrentGeneration and AbsoluteFitness are read into BestNCurrentSaveSet, which is then sorted on absolute fitness in the subroutine SortBestNCurrentSaveSet.

20.5 Sub RunGAUnit1Done: Detailed Code

Structure

As is clear from the code for Sub RunGAUnit1Done (Figure 20.3 and §20.7) this procedure has five main subroutine calls. We now briefly describe each and refer the reader to the complete code listing in §20.7.

20.5.1 PerformCrossover

This is the most complex of the five subroutines, but the basic idea is simple. Using fitness proportional section, two chromosomes are randomly drawn from CurrentGeneration. If crossover is drawn via a random number, then the two chromosomes are crossed over and the results read into the holding array, ChromosomeCopySpace. If crossover is not drawn, then the two chromosomes are simply copied into ChromosomeCopySpace. This continues until PopulationSize is reached, at which time ChromosomeCopySpace is copied back into CurrentGeneration.

20.5.2 PerformMutation

In this subroutine, the program loops through the entire array CurrentGeneration. For each entry a random number is drawn to determine whether there shall be a mutation. If there is to be a mutation, a uniform random number is drawn between the declared high and low values for the decision variable in question.

20.5.3 CalculateFitness

This routine calls the sub Evaluate which is a model-specific procedure that calculates the values for a row of the array AbsoluteFitness.

20.5.4 UpdateTheSaveSets

Only one save set is present in the program: BestNCurrentSaveSet. The program reads CurrentGeneration into columns 0 through NumberOfDecisionVariables and AbsoluteFitness into the remaining higher-order columns, all this beginning at line BestNSaved + 1. This has the effect of writing over the worst rows of BestNCurrentSaveSet, leaving the best BestNSaved rows intact. The program then (next sub) sorts BestNCurrentSaveSet on absolute fitness.
20.5.5 SortBestNCurrentSaveSet

The program uses a simple bubble sort on column $\text{NumberOfDecisionVariables} + 1$ of $\text{BestNCurrentSaveSet}$. This column is presumed to hold the absolute fitnesses of the various rows.

20.6 Sub PostpareGA: Detailed Code Structure

Sub PostpareGA calls two subroutines in order to write to files the current generation and the overall best $N$ (= $\text{BestNSaved}$) chromosomes found during the run of the GA. Source code for these two subroutines is given in Figures 20.5 and 20.6.

20.7 Complete Code Listing

There follows the complete listing of the code. Following the declarations section, the procedures, whether subs or functions, are in alphabetical order.

Note: For purposes of fitting the listing on the typeset page, I have occasionally broken lines. When I do this, I use the continuation symbol, $=>$, which is not part of Visual Basic.
Sub Print2FileBestOverall ()
Dim I, J As Integer
Dim FNameBestOverall, FNumBestOverall
Dim msg

FNumBestOverall = FreeFile
FNameBestOverall = 
==> "B" & NumberOfGenerationsSoFar & 
==> "F" & FNumBestOverall & ".TXT"
Open FNameBestOverall For Output As FNumBestOverall
For I = 1 To bestNSaved
    msg = ""
    For J = 0 To NumberOfDecisionVariables + OutputSize
            msg = msg & " " & BestNCurrentSaveSet(I, J)
    Next J
    Print #FNumBestOverall, msg
Next I
Close
End Sub

Figure 20.5: Sub Print2FileBestOverall: Source Code. Note: My continuation symbol, not in Visual Basic: ==>. 
Sub Print2FileCurGen ()
Dim I, J As Integer
Dim FNameCG, FNumCG
Dim msg

FNumCG = FreeFile
FNameCG = "C" & NumberOfGenerationsSoFar &
== "G" & FNumCG & ".TXT"
Open FNameCG For Output As FNumCG
For I = 1 To PopulationSize
    msg = ""
    For J = 0 To NumberOfDecisionVariables
        msg = msg & " " & CurrentGeneration(I, J)
    Next J
    For J = 1 To OutputSize
        msg = msg & " " & AbsoluteFitness(I, J)
    Next J
    msg = msg & " " & RelativeFitness(I)
    Print #FNumCG, msg
Next I
Close

End Sub

Figure 20.6: Sub Print2FileCurGen Source Code.
' sok 951127: This is in file: GACODE.BAS in
' the folder clapopo3
' Am freezing this for now. Call it the BasicGA program,
' version 951127.

Option Explicit

'**************************************************
'*************** Below, constants declared that ***
'*************** should be read in. ***************

'sok 951126: Note: These program variables,
' in a non-stubbed
' environment, need to be declared in the declarations
' section. They are so declared, but I have commented
' out the declarations (see below).

'+++++++++++++++++++++++++++++++++++++++
'++++ from GetGARunPars +++++++++++++

Const NumberOfGenerations = 20
'GetNumberOfGenerations
Const PopulationSize = 100
'GetPopulationSize
Const CrossoverRate = .77
'GetCrossoverRate
Const MutationRate = .23
'GetMutationRate
Const bestNSaved = 100
'GetBestNSaved
' ++++++++++++++++++++++++++++++++++++++++ ++++++++++++ from GetModelRunPars +++++++++++++
Const NumberOfDecisionVariables = 4

'GetOutputSize
Const OutputSize = 2

' ++++++++++++++++++++++++++++++++++++++++

' ++++++++++++++++++++++++++++++++++++++++ +++++ from/for InitDVarInfo/StubInitDVarInfo ++++++

Dim DecisionVariableInfo(1 To NumberOfDecisionVariables,
=> 1 To 4) As Double

Const DecisionVariableInfo11 = 5 'r, low
Const DecisionVariableInfo12 = 20 ' r, high
Const DecisionVariableInfo13 = 0 ' r, not integer
Const DecisionVariableInfo14 = 0 ' r, no grid search

Const DecisionVariableInfo21 = 10 ' v, low
Const DecisionVariableInfo22 = 30 ' v, high
Const DecisionVariableInfo23 = 0 ' v, not integer
Const DecisionVariableInfo24 = 0 ' v, no grid search

Const DecisionVariableInfo31 = 15 ' u, low
Const DecisionVariableInfo32 = 25 ' u, high
Const DecisionVariableInfo33 = 0 ' u, not integer
Const DecisionVariableInfo34 = 0 ' u, no grid search

Const DecisionVariableInfo41 = 200 ' l, low
Const DecisionVariableInfo42 = 300 ' l, high
Const DecisionVariableInfo43 = 0 ' l, not integer
Const DecisionVariableInfo44 = 0 ' l, no grid search

' +++++++++++++++++++++++++++++++++++++
'**************************************************
'*************** Above, constants declared that ***
'*************** should be read in. ***************

' Global variables

'++++ from GetGARunPars
' **** but explicitly declared above ++++++++++++  

' Dim NumberOfGenerations as Integer 
' Dim PopulationSize as Integer 
' Dim CrossoverRate As Double 
' Dim MutationRate As Double 
' Dim bestNSaved as Integer 

' ++++++ from GetModelRunPars
' ++++++but explicitly declared above ++++++++ 

' Dim NumberOfDecisionVariables As Integer 
' Dim OutputSize As Integer 

' ++++++++++++++++++++++++++++++++++++++++++++++++++++

Dim Index As Integer
Global CurrentGeneration() As Double
Global AbsoluteFitness() As Double
Dim ChromosomeCopySpace() As Double
Sub CalculateFitness()
  Dim I As Integer
  For I = 1 To PopulationSize
    Evaluate(I)
  Next I
  CalculateRelativeFitness
End Sub

Sub CalculateRelativeFitness()
  Dim I As Integer
  Dim Interval As Double
  Dim LowestAbsoluteFitness As Double
  Dim HighestAbsoluteFitness As Double
  Interval = FindHighest() - FindLowest()
  If Interval < .00000001 Then
    MsgBox "Whoa! In CalculateRelativeFitness, highest absolute fitness = " & HighestAbsoluteFitness & " and lowest absolute fitness = " & LowestAbsoluteFitness & " and interval = " & Interval & " which is too small to divide by."
    HighestAbsoluteFitness = " & HighestAbsoluteFitness & " and LowestAbsoluteFitness = " & LowestAbsoluteFitness & " and interval = " & Interval & " which is too small to divide by."
  End If
  For I = 1 To PopulationSize
    If Interval > .00000001 Then
      RelativeFitness(I) = (AbsoluteFitness(I, 1) - LowestAbsoluteFitness) / Interval
    Else
      RelativeFitness(I) = 1
    End If
  Next I
End Sub
Sub CopyStrings (String1, String2, Index)
Dim I As Integer

For I = 0 To NumberOfDecisionVariables
    ChromosomeCopySpace(Index, I) =
    => CurrentGeneration(String1, I)
    ChromosomeCopySpace(Index + 1, I) =
    => CurrentGeneration(String2, I)
Next I

End Sub

Function Crossover () As Integer
Dim ReturnValue As Integer

If Random01Value() <= CrossoverRate Then
    ReturnValue = 1
Else
    ReturnValue = 0
End If
Crossover = ReturnValue
End Function

Sub CrossoverStrings (String1, String2, Index)
Dim I As Integer

CrossoverPoint = Int((Random01Value() *
=> (NumberOfDecisionVariables - 1)) + 1)
If CrossoverPoint >= NumberOfDecisionVariables Then
    MsgBox "Whoa! In CrossoverStrings."
End If

' Copy up to the crossover point
For I = 1 To CrossoverPoint
    ChromosomeCopySpace(Index, I) =
    => CurrentGeneration(String1, I)
    ChromosomeCopySpace(Index + 1, I) =
    => CurrentGeneration(String2, I)
Next I

' Copy past the crossover point to the end
For I = CrossoverPoint + 1 To NumberOfDecisionVariables
    ChromosomeCopySpace(Index, I) =
    => CurrentGeneration(String2, I)
CHAPTER 20. BASICGA: CODE FOR GENETIC ALGORITHMS

ChromosomeCopySpace(Index + 1, I) =  
=> CurrentGeneration(String1, I)
Next I

' Assign new IDs to the chromosomes
ChromosomeCopySpace(Index, 0) = GetCurrentIDNum()
ChromosomeCopySpace(Index + 1, 0) = GetCurrentIDNum()
End Sub

' ***************** Main Program *********
',
Sub DoTheGA ()

Randomize (17)
ChDir "c:\clasave\"
NoisyOutput = 1

'1. Make preparations to run the GA.

PrepareGA

' 2. Run the GA until the stopping condition is met

RunGAUntilDone

' 3. Postpare the system

PostpareGA
End Sub

Sub Evaluate (I)

' Note: This is a model-specific routine.
' And should be revised, e.g.
' p1 goes to r
Dim p1, p2, p3, p4 As Double
p1 = CurrentGeneration(I, 1)
p2 = CurrentGeneration(I, 2)
p3 = CurrentGeneration(I, 3)
p4 = CurrentGeneration(I, 4)

AbsoluteFitness(I, 1) = 2 * p1 * (1 + p2 / p3) / p4
AbsoluteFitness(I, 2) = 2 * p1 * (1 + p2 / p3) / p4

End Sub
Function FindHighest () As Double
Dim I As Integer
Dim Highest As Double
Highest = AbsoluteFitness(1, 1)
For I = 1 To PopulationSize
    If AbsoluteFitness(I, 1) > Highest Then
        Highest = AbsoluteFitness(I, 1)
    Next I
FindHighest = Highest
End Function

Function FindLowest () As Double
Dim I As Integer
Dim Lowest As Double
    Lowest = AbsoluteFitness(1, 1)
    For I = 1 To PopulationSize
        If AbsoluteFitness(I, 1) < Lowest Then
            Lowest = AbsoluteFitness(I, 1)
        Next I
FindLowest = Lowest
End Function

Function GetCurrentIDNum () As Double
    CurrentIdNum = CurrentIdNum + 1
    GetCurrentIDNum = CurrentIdNum
End Function

Sub GetGARunPars ()
    ' This is a stub right now, with the program variables to be
    ' initialized here declared as constants in the
    ' declarations section.
    ' But here they are:
    'Const NumberOfGenerations = 2
    'GetNumberOfGenerations
    'Const PopulationSize = 50
    'GetPopulationSize
    'Const CrossoverRate = .77
    'GetCrossoverRate
    'Const MutationRate = .23
    'GetMutationRate
    'Const BestNSaved = 50
    'GetBestNSaved
    NumberOfGenerationsSoFar = 0
End Sub
Sub GetModelRunPars()
' This is a stub right now, with the program variables to be
' initialized here declared as constants in the
' declarations section.
' But here they are:

'NumberOfDecisionVariables = 4
   'GetNumberOfDecisionVariables
'OutputSize = 2
   'GetOutputSize

StubInitDVarInfo
   'for InitDVarInfo
End Sub

Sub InitializeGA()
CurrentIdNum = 0

GetGARunPars
GetModelRunPars
ReDimGAArrays

End Sub

Sub InitializeSaveSets()
Dim I, J As Integer

' Number of rows is the number in the best N save set
' plus the population size
' Number of columns is no. decision variables + ID +
' absolute fitness

' Read in CurrentGeneration array
For I = 1 To PopulationSize
    For J = 0 To NumberOfDecisionVariables
        BestNCurrentSaveSet(I, J) = CurrentGeneration(I, J)
    Next J
Next I

' Read in AbsoluteFitness array
' Note: In the BestNCurrentSaveSet array
' the absolute fitness is
' kept in column number NumberOfDecisionVariables + 1.
For I = 1 To PopulationSize
    Next I
End Sub

Sub MakeGAGenOne()
    Dim I, J As Integer
    Dim LowValue, HighValue As Double
    For I = 1 To PopulationSize
        For J = 1 To NumberOfDecisionVariables
            LowValue = DecisionVariableInfo(J, 1)
            HighValue = DecisionVariableInfo(J, 2)
            CurrentGeneration(I, J) = RandomBetween(LowValue, HighValue)
        Next J
        CurrentGeneration(I, 0) = GetCurrentIDNum()
    Next I
End Sub

Sub PerformCrossover()
    Dim I, J As Integer
    Dim String1, String2 As Integer
    Dim SumFitnesses As Double
    SumFitnesses = 0
    For I = 1 To PopulationSize
        SumFitnesses = SumFitnesses + RelativeFitness(I)
    Next I
    ‘ CrossoverLikelihood accumulates the probabilities of crossover. So,
    ‘ CrossoverLikelihood(PopulationSize) should = 1.
    CrossoverLikelihood(1) = RelativeFitness(1) / SumFitnesses
    For I = 2 To PopulationSize
        CrossoverLikelihood(I) =
            ‘ (RelativeFitness(I) / SumFitnesses) +
            ‘ CrossoverLikelihood(I - 1)
    Next I
    For I = 1 To PopulationSize Step 2
        String1 = RandomStrings()
        ‘ get a random string that can be crossed over
CHAPTER 20. BASICGA: CODE FOR GENETIC ALGORITHMS

String2 = RandomStrings()
' get a random string that can be crossed over
If Crossover() = 1 Then
' If we do crossover here, then
CrossoverStrings String1, String2, I
Else ' We don’t do crossover and
' we just copy the chromosomes to the next generation.
Call CopyStrings(String1, String2, I)
End If
Next I

' copy back into the CurrentGeneration array
For I = 1 To PopulationSize
    For J = 0 To NumberOfDecisionVariables
        CurrentGeneration(I, J) =
=>ChromosomeCopySpace(I, J)
    Next J
Next I
End Sub

Sub PerformMutation ()
Dim I, J As Integer

For I = 1 To PopulationSize
    For J = 1 To NumberOfDecisionVariables
        If Random01Value() < MutationRate Then
            CurrentGeneration(I, J) =
=>RandomBetween(DecisionVariableInfo(J, 1),
=>DecisionVariableInfo(J, 2))
            CurrentGeneration(I, 0) = GetCurrentIDNum()
        End If
    Next J
Next I
End Sub

Sub PostpareGA ()

' Print out final generation.
Print2FileCurGen
' Print out the best finds overall.
Print2FileBestOverall

If (NoisyOutput = 1) Then
    MainForm.ProgressBox.Text = "All done."
End If
End Sub

Sub PrepareGA ()
   ' 1. Initialize the system
   InitializeGA

   ' 2. Validate the input data
   ValidateGAInput

   ' 3. Generate the initial population of chromosomes
   MakeGAGenOne

   ' 4. Calculate the absolute and relative
   '    fitnesses for each chromosome.
   CalculateFitness

   ' 5. Initialize the save sets
   InitializeSaveSets

End Sub

Sub Print2FileBestOverall ()
   Dim I, J As Integer
   Dim FNameBestOverall, FNumBestOverall
   Dim msg

   FNumBestOverall = FreeFile
   FNameBestOverall = "B" & NumberOfGenerationsSoFar & 
   "F" & FNumBestOverall & ".TXT"
   Open FNameBestOverall For Output As FNumBestOverall
   For I = 1 To bestNSaved
      msg = ""
      For J = 0 To NumberOfDecisionVariables + OutputSize
         msg = msg & " " & BestNCurrentSaveSet(I, J)
      Next J
      Print #FNumBestOverall, msg
   Next I
   Close
End Sub

Sub Print2FileCurGen ()
Dim I, J As Integer
Dim FNameCG, FNumCG
Dim msg

FNumCG = FreeFile
FNameCG = "C" & NumberOfGenerationsSoFar & ">" & FNumCG & ".TXT"
Open FNameCG For Output As FNumCG
For I = 1 To PopulationSize
  msg = ""
  For J = 0 To NumberOfDecisionVariables
    msg = msg & " " & CurrentGeneration(I, J)
  Next J
  For J = 1 To OutputSize
    msg = msg & " " & AbsoluteFitness(I, J)
  Next J
  msg = msg & " " & RelativeFitness(I)
  Print #FNumCG, msg
Next I
Close

End Sub

Function Random01Value ()
' Note: Here and only here we use the 0-1
' random number generator built into Basic.

  Random01Value = Rnd
' return a random value from the interval [0,1]

End Function

Function RandomBetween (Low, High)

  RandomBetween = (Random01Value() * (High - Low)) + Low

End Function

Function RandomStrings ()
' The purpose of this routine is to pick
' a chromosome to contribute to the next
20.7. COMPLETE CODE LISTING

' generation. The likelihood of being picked
' is proportional to the relative fitness of the
' chromosome
Dim I As Integer
Dim PointOnUnitInterval As Double

PointOnUnitInterval = Random01Value()
I = 1
While CrossoverLikelihood(I) < PointOnUnitInterval
    I = I + 1
Wend

RandomStrings = I

End Function

Sub ReDimGAArrays ()
ReDim CurrentGeneration(1 To PopulationSize,
=> 0 To NumberOfDecisionVariables) As Double
ReDim ChromosomeCopySpace(1 To PopulationSize,
=> 0 To NumberOfDecisionVariables) As Double
ReDim AbsoluteFitness(1 To PopulationSize,
=> 1 To OutputSize) As Double
ReDim RelativeFitness(1 To PopulationSize) As Double
ReDim BestNCurrentSaveSet(1 To bestNSaved +
=> PopulationSize, 0 To NumberOfDecisionVariables +
=> OutputSize) As Double
ReDim CrossoverLikelihood(1 To PopulationSize)
=> As Double
End Sub

Sub RunGAUntilDone ()
Do Until NumberOfGenerationsSoFar >= NumberOfGenerations
    If (NoisyOutput = 1) Then
        MainForm.ProgressBox.Text =
        "NumberOfGenerationsSoFar = " & NumberOfGenerationsSoFar
    End If
    ' Now to the main business:
    PerformCrossover
    PerformMutation
    CalculateFitness
    UpdateTheSaveSets
    SortBestNCurrentSaveSet
    NumberOfGenerationsSoFar = NumberOfGenerationsSoFar + 1
End Do

End Sub
Loop
    If (NoisyOutput = 1) Then
        MainForm.ProgressBox.Text =
        => "NumberOfGenerationsSoFar = "
        => & NumberOfGenerationsSoFar
    End If
End Sub

Sub SortBestNCurrentSaveSet ()
    Dim CurrentRow, I As Integer
    Dim ArraySize As Integer
    Dim SortIndex As Integer
    Dim NumberSwapped As Long
    NumberSwapped = -1
    ArraySize = bestNSaved + PopulationSize
    SortIndex = NumberOfDecisionVariables + 1
    ' Above: note that in InitializeSaveSets that
    ' the absolute fitness is read
    ' into column NumberOfDecisionVariables + 1
    While NumberSwapped <> 0
        NumberSwapped = 0
        For CurrentRow = 1 To ArraySize
            I = CurrentRow
            While I <= ArraySize
                If BestNCurrentSaveSet(CurrentRow, SortIndex) <
                => BestNCurrentSaveSet(I, SortIndex) Then
                    SwapRows CurrentRow, I
                    NumberSwapped = NumberSwapped + 1
                End If
                I = I + 1
            Wend
        Next CurrentRow
    Wend
End Sub

Sub StubInitDVarInfo ()
    Dim I, J As Integer
    ' Load up the array DecisionVariableInfo
    'For I = 1 To NumberOfDecisionVariables
For J = 1 To 4
' Next J
Next I

DecisionVariableInfo(1, 1) = DecisionVariableInfo11
'r, low
DecisionVariableInfo(1, 2) = DecisionVariableInfo12
'r, high
DecisionVariableInfo(1, 3) = DecisionVariableInfo13
'r, not integer
DecisionVariableInfo(1, 4) = DecisionVariableInfo14
'r, no grid search

DecisionVariableInfo(2, 1) = DecisionVariableInfo21
'v, low
DecisionVariableInfo(2, 2) = DecisionVariableInfo22
'v, high
DecisionVariableInfo(2, 3) = DecisionVariableInfo23
'v, not integer
DecisionVariableInfo(2, 4) = DecisionVariableInfo24
'v, no grid search

DecisionVariableInfo(3, 1) = DecisionVariableInfo31
'u, low
DecisionVariableInfo(3, 2) = DecisionVariableInfo32
'u, high
DecisionVariableInfo(3, 3) = DecisionVariableInfo33
'u, not integer
DecisionVariableInfo(3, 4) = DecisionVariableInfo34
'u, no grid search

DecisionVariableInfo(4, 1) = DecisionVariableInfo41
'l, low
DecisionVariableInfo(4, 2) = DecisionVariableInfo42
'l, high
DecisionVariableInfo(4, 3) = DecisionVariableInfo43
'l, not integer
DecisionVariableInfo(4, 4) = DecisionVariableInfo44
'l, no grid search

End Sub

Sub SwapRows (Index1, Index2)
Dim I As Integer
Dim Temp() As Double
ReDim Temp(0 To NumberOfDecisionVariables + OutputSize) As Double

For I = 0 To NumberOfDecisionVariables + OutputSize
    Temp(I) = BestNCurrentSaveSet(Index1, I)
    Next I

For I = 0 To NumberOfDecisionVariables + OutputSize
    BestNCurrentSaveSet(Index1, I) = BestNCurrentSaveSet(Index2, I)
    Next I

For I = 0 To NumberOfDecisionVariables + OutputSize
    BestNCurrentSaveSet(Index2, I) = Temp(I)
    Next I

End Sub

Sub UpdateTheSaveSets()
Dim I, J As Integer

' Basically, this subroutine dumps the
' CurrentGeneration array
' and the AbsoluteFitness array into the
' BestNCurrentSaveSet array, by appending them after the
' current bestNSaved. Later, we sort the entire array.
' This is done as the next subroutine call in RunGAUntilDone.

' Right now only the best N overall save set
' Number of rows is the number in the save set plus
' the population size
' Number of columns is no. decision variables +
' ID + absolute fitness
For I = 1 + bestNSaved To PopulationSize + bestNSaved
    For J = 0 To NumberOfDecisionVariables
        BestNCurrentSaveSet(I, J) = CurrentGeneration(I - bestNSaved, J)
        Next J
    For J = 1 To OutputSize
        BestNCurrentSaveSet(I, NumberOfDecisionVariables + J) = AbsoluteFitness(I - bestNSaved, J)
        Next J
    Next I
End Sub
Sub ValidateGAInput ()
    ' Note: There's a lot more that needs doing here.
    If NumberOfGenerationsSoFar > NumberOfGenerations Then
        MsgBox "NumberOfGenerationsSoFar > NumberOfGenerations " &
    End If
End Sub

20.8 File Notes

April 8, 1998: Brought this file, dt-basicga.tex (formerly latex), over from the
Macintosh. It's history there: [File: dt-basicga.latex. Created: November 27,
1995, from clam1-code.latex. Revised: 951222, 951128.]