Laboratory 8 – Week 10

## Debugging and Exceptions

### 8.1 Introduction

**Firstly, this worksheet *is* NOT one of the worksheets from which your laboratory worksheets portfolio of work will be assessed. Therefore this worksheet has no hand in date and it does not need to be assessed. However it will help you practice essential skills needed for the other worksheets and programming in general.**

This laboratory worksheet is based on a fictitious business scenario and covers the debugging and fixing of a pre-written Java program. This program contains a number of bugs which need to be identified, noted and corrected. There will be a number of terms that you may not be familiar with that have not been covered on the course. You can look these terms up (e.g. Google), ask a GTA or work out if you need to know them or not to complete the worksheet.

### 8.2 Preliminaries

You are a senior programmer for the global company **The Tiresias Corporation**. Your company employed the small software development company **Bodgitt and Scarper Limited Software Solutions** to develop a complex piece of software core to your company’s internal business processes. The software company ran vastly over time (the promised and negotiated delivery date) and rather disturbingly they did not ask any questions regarding the specification (see section 8.4). The project manager responsible for hiring the company (apparently off Rent-a-Coder) was promptly dismissed and your company paid off Bodgitt and Scarper and acquired the code.

This code does not work and thus you have been tasked to get it working. The code can be found in section 8.7.

**8.3 Your Task**

You need to get the code up and running. The initial project was a “proof of concept” project based on applying the **Hill** **Climbing** (HC) algorithm to solving the **Knapsack** problem. The aim was to see whether the HC algorithm performed well on this type of problem since a major resource allocation problem that the Tiresias Corporation has can be shown to be decomposable to the Knapsack problem.

|  |  |
| --- | --- |
| **Error Type** | **Description** |
| Syntax | Java language grammar violations |
| Run Time | Errors produced during the running of the program, probably generated by *Exceptions* |
| Semantic | Logical errors - probably through misunderstanding of the specification |
| Table 8.1. Error/Bug Types | |

You need to locate each problem in the code, fix it and identify what type of bug it is. The bug types can be found in table 8.1. You must fully document what you did to fix the program, since your company will be suing Bodgitt and Scarper at a later date.

**8.4 The Knapsack Problem**

The Knapsack problem is a combinatorial optimisation problem that has a wide ranging number of applications. The idea is that there are *n* objects (each object has a weight which is a real number *wi* > 0) that need to be put into a Knapsack that has a finite capacity *C*. Thus the sum of the weights of all of the objects put in the Knapsack cannot exceed *C*. The aim is to select which objects best fill the Knapsack without exceeding the capacity limit *C*.

Given that there are *n* weights there are 2*n*-1 possible combinations, i.e. the number of possible subsets of objects that can be placed into the bag. We need the combination that is the best, i.e. the one that fills the Knapsack as close to capacity as possible without exceeding limit *C*.

There is no known algebraic or simple solution to the Knapsack problem and it is in fact a well-known NP-Hard[[1]](#footnote-1) problem.

The program should solve the problem using the HC algorithm detailed below. The test data for the program is the first 1,000 prime numbers found in section 8.8.

**8.5 The Hill Climbing Algorithm**

The Hill Climbing algorithm is a simple Heuristic Search technique from Artificial Intelligence. The algorithm is used to solve difficult problems in an approximate manner, such as the Knapsack problem described above. The algorithm works by making an initial random guess at a solution to a problem. It then iteratively tried to randomly improve on the solution by making a sequence of small changes. Once a better small change is found, subsequent small changes are made to the better solution. The version of the Hill Climbing algorithm used in this project is the **Random Mutation Hill Climber** (RMHC).

The pseudo-code of the hill climbing algorithm can be found in algorithm 8.1 below.

|  |  |
| --- | --- |
| Algorithm 8.1. RMHC(ITER) | |
| Input: | ITER- the number of iterations to run for |
| 1) | Let S be a random point in the search space |
| 2) | Let F be the fitness of S |
| 3) | For i = 1 to ITER (number of iterations) |
| 4) | Let S’ be a random point close to S (small change) |
| 5) | Let F’ be the fitness of S’ |
| 6) | If F’ is better than F Then |
| 7) | Let S = S’ and Let F = F’ |
| 8) | End If |
| 9) | End For |
| Output: | S- a solution |

In the above algorithm, the search space is the collection of all possible solutions a solution is being searched from and the fitness or fitness function is a way of measuring the quality of a solution, i.e. how well a solution solves the problem being tackled.

The following sections describe the additionally features needed to implement the Hill Climbing algorithm.

**8.5.1 Representation**

In order to generate a random solution, a representation is needed. This is how the computer stores a solution to a problem. In the case of the Knapsack problem, this is which weights are in the Knapsack.

If there are *n* possible weights, *wi*, *i*=1,...,*n*, then a solution to the problem, say *S*, can be stored in an *n*-length binary string. For example if there are 5 objects/weights then *S* = 10101 means that objects 1, 3 and 5 are in the Knapsack.

**8.5.2 Small Change**

For the RMHC to work correctly, the small change operator must be the smallest possible change possible. The standard way of making a small change to a binary string is to randomly choose a single digit (say between 1 and *n* inclusive) and then flip that digit. I.e. if it was a '1' make it a '0' and if it was a '0' make it a '1'.

**8.5.3 Fitness Function**

A fitness function is used to rate how good a solution is at solving a problem and usually returns a number. Most fitness functions are a mapping from a solution to a real number. In the case of the Knapsack problem there is a solution (*S* = [*s*1,...,*si*,...,*sn*]), a set of objects weights (*W* = [*w*1,...,*wi*,...,*wn*]) and a capacity *C* (>0). The weights and capacity **never** change for a specific instance of the problem.

There are many fitness functions for the Knapsack problem. The one that will be used for this project is as follows:

Let Sum = the sum of all of the weights in the Knapsack

If Sum > C then

Let Fitness = C - Sum (a negative value)

Else

Let Fitness = Sum (a positive value)

End If

Note that we want the fitness to be as large as possible. Therefore the concept of ***better*** in the RMHC algorithm is whether *F'* > *F*. This is **VERY** important. More formally:



**8.5.4 Experiments**

The Tiresias Corporation wishes the program to be tested on 1,000 prime numbers, with a Knapsack capacity of *C*=1,000 and run ten times (repeats) for 10,000 iterations each run. If the algorithm is working correctly then the results should be very close to 1,000 for each run.

**8.6 System Architecture**

Note from previous project manager: Bodgitt and Scarper did not supply a flow chart of the system's architecture. The management will require one - could you draw one please...

**8.7 Source Code**

Note from previous project manager: Bodgitt and Scarper sent us the following. It seems to have become a little corrupted after being emailed from their “HQ” in Turkmenistan...

|  |
| --- |
| **Source Code - Caution. Here be Dragons...** |
|  |

**8.8 1000 Prime Numbers**

Note from previous project manager: Bodgitt and Scarper did not supply any test data, my suggestion is that you download the first 1000 prime numbers from <http://primes.utm.edu/lists/small/1000.txt>.

1. Note from previous project manager: This is quite a difficult concept, it is best thought of as being the collection of really hard to solve problems, of which any solution takes a very long time to find... [↑](#footnote-ref-1)