Name: Wimal Perera
Admission No: 09/10008

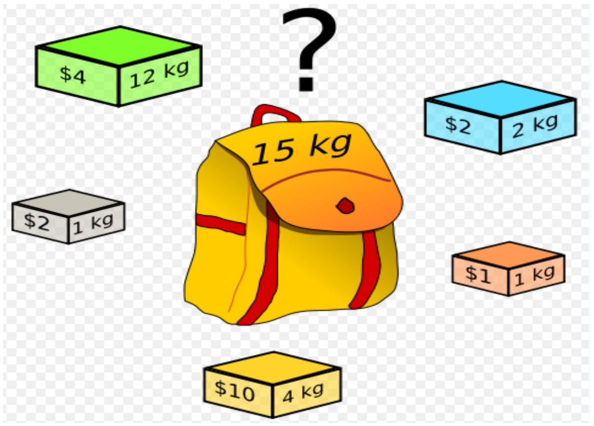
Evolutionary Computing – Assignment

Part I

1. Sample Problem

The sample problem which I selected to solve using Genetic algorithms is *0/1 Knapsack Problem* [1].

When you're given a set of items with each item having its own weight and own monetary value, how would you select a subset out of these items so that the total weight is less than a given maximum weight, but the total monetary value is as large as possible?



e.g. What are the boxes that you must choose out of the given 5 boxes so that you're able to take maximum amount of money while all the boxes you select can be put into your bag (Knapsack) which can hold any total weight less than or equal to 15 kg?

2. Encoding

To encode a set of n-items we use a *BitVector* which indicates for each item whether we take it or not. A set of such *BitVectors* (each representing a particular selection of items) serve as the set of chromosomes (individuals) in a particular generation according to the genetic algorithm.

e.g. Suppose we have 10 items. We may represent our intended selection of the subset of items as a *BitVector* with 10 bits as follows. This is a single chromosome.

1101010001

The meaning of the above chromosome (i.e. BitVector) is that we select only 1st, 2nd, 4th, 6th & 10th items to be put into our bag (Knapsack) out of all the 10 items.

3. Fitness Function

From the evolutionary computing view, the fitness function works as an indicator which can be used to determine how a particular chromosome (i.e. a candidate solution/individual) is close to the goal (i.e. a collection of items giving maximum amount of money, while still the total weight of items is less than or equal to the maximum weight that can be carried in the bag) which we intend. When the solution (i.e. chromosome) is getting closer to our goal the fitness function must be able to indicate a higher fitness value. Therefore we use the following algorithm to implement the fitness function when determining the optimality for a candidate solution (i.e. a chromosome) for the Knapsack problem.

```
Input:
A BitVector (i.e. a chromosome) indicating the items those are selected to be
put into the bag (i.e. Knapsack)
e.g. 1101010001 (indicating that we select only 1st, 2nd, 4th, 6th and 10th items
to be put into the bag)

Variables:
Number: Fitness_Value, Total_Weight, Total_Monetary_Value, Capacity

Algorithm:
Initialize Fitness_Value = 0, Total_Weight = 0, Total_Monetary_Value = 0

Capacity = "Maximum weight that can be carried by the bag (Knapsack) at once"

For Each item i
   If item i is Selected Then
        Total_Weight = Total_Weight + weight of i th item
        Total_Monetary_Value = Total_Monetary_Value + value of i th item
   End If
End For Each

If Total_Weight <= Capacity Then
        # This means our sack is underweight/equal weight.
        # Thus the solution is acceptable.
        # Therefore we return a positive value for fitness.

        Fitness_Value = Total_Monetary_Value

Else
        # This means our sack is overweight.
        # Thus the solution is not acceptable.
        # Therefore we return a negative value for fitness.

        Fitness_Value = -(Total_Weight - Capacity)
End If

Return:
Fitness_Value
```

4. Implementing a solution using a selected GA Toolkit

I used ECJ [3], an open source GA toolkit written in java by EC labs (Evolutionary Computation Laboratory) at George Mason's University, USA [2].

You can download ECJ from EC labs website [3].

ECJ comes as 2 zip files [3];

1. ecj.zip (containing source code and build for the GA toolkit)
2. libraries.zip (containing jar files of dependencies)

ECJ is a highly configurable GA toolkit. Configuration can be specified by adhering to the syntax of java property files with file extension *.params. In a *.param file we can specify [4];

1. Nature of a Chromosome (number of genes; whether genes are bits, integers, floating point numbers etc.)
2. Number of Generations
3. Selection Methods (I've used *Tournament Selection*)
4. Crossover Methods (I've used *One Point Crossover*)
5. Crossover & Mutation Probabilities
6. The problem

Please note that in the above configuration parameters, "the problem" is the custom class which must be written according to our problem. This problem class contains the definition for the fitness function to evaluate chromosomes.

Please refer to the appendix found at the end of this document for in depth details in terms of implementing a solution for the knapsack problem using ECJ.

5. Reusing the GA Solution of Knapsack Problem to solve Local Examples

The major reason for me to select Knapsack problem for this assignment is its applicability for certain problems that can be found from local context; related with "*resource allocation while maintaining optimal conditions and adhering to a certain constraint*". Following are some scenarios from the local context those are analogous to the "*knapsack problem*"; so that we can reuse our genetic algorithm solution for "*knapsack problem*" to model any of them.

1. Recruiting Employees for a project – Different employees will have different skills. Highly skilled employees are more demanding. But in the meantime their productivity is high as well. Different employees will have different levels for their demand and productivity. However allocation of employees must be done with respect to a total allocated project budget. Therefore the resulting problem is "*how would you allocate the correct set of employees to gain maximum productivity while adhering to the total pre-allocated project budget*"?
2. Allocating water for farm lands from a reservoir with limited capacity – Most of the time production from a farm land will be not be proportional to its water supply. Some lands may produce more production although they consume a less water supply (may be soil has high levels of required nutrients). In the meantime some lands may produce in large scale but also consume a large water supply (nature of the crop being cultivated, like rice). So our problem is "*how can we find an optimal schedule in order to maximize the production of crops while*

adhering to the maximum capacity of our main water reservoir"? Therefore this kind of an optimized schedule can be used in a frequent manner when releasing water from the main reservoir.

Therefore both of the above problems can be modeled in a *similar fitness function* which we have used to solve the *knapsack problem*.

Part II

Genetic Algorithms for Solving Optimization Problems

The use of Genetic Algorithms is most popular when it comes to solving problems which will have more than one solution but we're interested in finding the most optimum solution or set of solutions out of all the available solutions.
e.g. In the 0/1 knapsack problem we can select any of the items and put them in our sack as long as the total weight of items is less than the maximum weight which our sack can hold. So there are many solutions. But what we want to select is the set of items those provide the highest monetary value while their total weight is able to support the maximum capacity of our bag. Therefore although there are many solutions there is only one or very few *optimal* solutions.

In the GA approach, the optimum solution is reached by starting from generating an initial set of randomly generated solutions (i.e. we call them the initial population or the 1st generation). Then each of these initial set of solutions are treated as guesses and evaluated against a certain statistic which we call the fitness function. After each initial solution is evaluated by the fitness function, we may find the suitability of each solution (guess) for our problem. So solutions which are high-fit will be given a more attention initially. So we have started from an initial population (i.e. 1st generation) with random set of guessed solutions and what we found as high-fit solutions are the best solutions ***relatively*** to those randomly guessed initial set of solutions. But the problem continues as that there can be other solutions for our original problem when we consider the entire possible solution space. Further out of these still unknown other solutions, some may be better than the best solutions which we have in our pockets already.

This is where the concepts of evolution (cloning, selection, crossover and mutation) play a vital role in our genetic algorithm. We select the individuals from our initial population (most of the time a mix of majority of best-fit individuals and minority worst-fit individuals). Then reproduce new individuals (crossover) by interchanging the characteristics between them while incorporating a novelty (mutation) in a ***minor set*** of newly reproduced individuals. Then we form our 2nd generation with these newly generated solutions and the very few utmost fit solutions (cloning) in the previous generation. So when we evaluate this entire 2nd generation again we can inspect whether there are any other solutions those are best fit than the best-fit solutions in our 1st generation. If there are such better solutions we can assume them as the best-fit solution/solutions for the moment. Similarly we can generate the 3rd, 4th and n th generation iteratively and observe that a convergence is happening for the best-fit solutions as it passes from generation to generation, ultimately ending up with the *optimal* solution which we really wanted.

The most interesting part of our discussion is why this GA approach is well suited than the traditional approaches in solving optimization problems?

One of the reasons to choose GA is that GA approach wins over traditional methods for optimization problems when the number of input parameters increases. For an example in the knapsack problem we have 2 deal with 2 varying input parameters; the weight of an item and the monetary value of an item. One traditional method to solve optimization problems is to construct a mathematical model in the form of a function and try to find the maximum of a monetary value while adhering to our constraints. What if we consider a higher form of a knapsack problem; this time we have to select the items generating a maximum monetary value with 2 constraints where not only the total weight, but also the total volume of our items should be able to meet the maximum weight and maximum volume that can be held by our bag. The main point which I want to emphasize here is that as the total number of parameters increase problem solving with mathematical models gets more complicated and time consuming. But in terms of the GA approach the only additional effort is adjusting the fitness function accordingly after inspecting several individual solutions. So in terms of GA we can still start from generating an initial population of solutions and based on fitness see whether they are converging towards an optimal solution as it passes from generation to generation.

Another reason for the popularity of GA approach is that the adaptability of them when the entities in our problem space are changing dynamically. What if the weights and volumes wary of items are not fixed? It seems not practical when it comes to the knapsack problem. A more practical example would be a virtual private network which is connected across different network paths and promised to serve different levels of quality of services for its clients. But in any network all clients won't be using the network at the same time. Therefore the maximum bandwidth is always set to less than the sum of bandwidth required by all users (clients). The behavior of users with respect to parameters like network usage will vary from time to time. Therefore our problem is how to find an optimized schedule for everyone using the network with the limited bandwidth. In this case if the network usage changes from time to time for certain users our problem is more complicated since we need to find the optimal solution based dynamic entities. In a case like this the GA approach will still be able to find an optimal solution as the optimal schedule will be found after being evolved for the new environment of users by iterating across several generations ahead.

References

[1] Knapsack Problem – Wikipedia - http://en.wikipedia.org/wiki/Knapsack_problem
Last viewed on: 5th December 2009
[2] EC Lab – George Mason's University - http://cs.gmu.edu/~eclab/
Last viewed on: 5th December 2009
[3] ECJ – A Java based Evolutionary Computation Research System - http://cs.gmu.edu/~eclab/projects/ecj/
Last viewed on: 5th December 2009
[4] Tutorial 1: Build a Genetic Algorithm for the MaxOnes Problem - http://cs.gmu.edu/~eclab/projects/ecj/docs/tutorials/tutorial1/index.html
Last viewed on: 5th December 2009

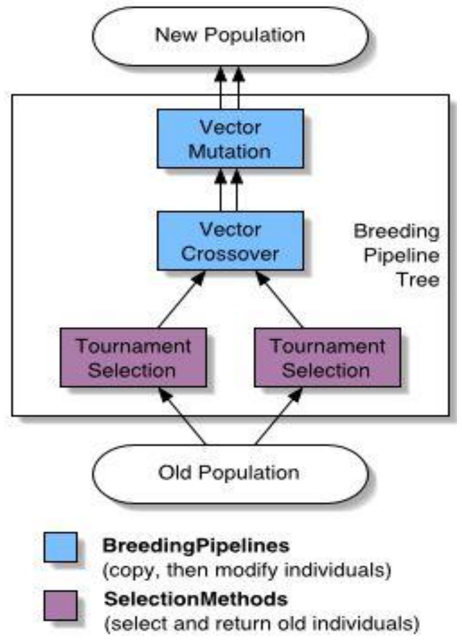
Appendix

Section A – Configuration File (knapsack.params)

```
# My implementation contains the solution for the
# following knapsack problem.
#
# Item No | 1| 2| 3| 4| 5| 6| 7| 8| 9| 10
# Weight  |14|26|19|45| 5|25|34|18|30| 12
# Value   |20|24|18|70|14|23|50|17|41| 21
#
# Maximum weight supported by the Knapsack (Capacity) 100
#
# I'm using a BitVector of length 10 to represent chromosomes
# So each chromosome will have 10 genes
#
# Steady-state model is used for the execution of GA
# There are 20 generations
# There are 20 chromosomes in a generation
# Tournament Selection is used during selection
# One-point crossover is used during crossover
# Crossover probability is 0.8
# Mutation probability is 0.01
#
#

# Basic Settings
verbosity   = 0
breedthreads      = 1
evalthreads = 1
seed.0            = 4357

# Object responsible for maintaining the state of entire genetic algorithm
state       = ec.simple.SimpleEvolutionState

# Defining the Population
pop         = ec.Population
init        = ec.simple.SimpleInitializer
finish            = ec.simple.SimpleFinisher
breed       = ec.simple.SimpleBreeder
eval        = ec.simple.SimpleEvaluator
stat        = ec.simple.SimpleStatistics
exch        = ec.simple.SimpleExchanger

# No of generations
generations       = 20
quit-on-run-complete    = true
checkpoint        = false
prefix                  = ec
checkpoint-modulo = 1

# Output file to write statistics
stat.file         = $out.stat
```

```
# Settings for defining a generation
pop.subpops       = 1
pop.subpop.0            = ec.Subpopulation
pop.subpop.0.size             = 20
pop.subpop.0.duplicate-retries      = 0
pop.subpop.0.species          = ec.vector.VectorSpecies

# Nature of a Chromosome
pop.subpop.0.species.ind      = ec.vector.BitVectorIndividual
pop.subpop.0.species.genome-size    = 10
pop.subpop.0.species.crossover-type = one

# Crossover and mutation probabilities
pop.subpop.0.species.crossover-prob = 0.8
pop.subpop.0.species.mutation-prob  = 0.01

# Fitness configuration
pop.subpop.0.species.fitness        = ec.simple.SimpleFitness

# Breeding pipe line
pop.subpop.0.species.pipe                 =
ec.vector.breed.VectorMutationPipeline
pop.subpop.0.species.pipe.source.0        =
ec.vector.breed.VectorCrossoverPipeline
pop.subpop.0.species.pipe.source.0.source.0     =
ec.select.TournamentSelection
pop.subpop.0.species.pipe.source.0.source.1     =
ec.select.TournamentSelection
select.tournament.size        = 2

# Custom class to our problem, containing our custom fitness function
eval.problem            = ec.app.assignment.KnapSack
```

Section B – Breeding Pipeline and Selection Methods according to configuration in Section A



Section C – Custom Problem class with the fitness function (Knapsack.java)

```java
/*
  Wimal Perera
  09/10008
  Evolutionary Computing - Assignment
 */

package ec.app.assignment;

import ec.*;
import ec.simple.*;
import ec.vector.*;

public class KnapSack extends Problem implements SimpleProblemForm {
      /**
       *
       */
      private static final long serialVersionUID = 1L;

      /**
       * Weights of the 10 items
       */
      private int[] WEIGHTS = new int[] {
            14, 26, 19, 45, 5, 25, 34, 18, 30, 12
      };

      /**
       * Monetary values of the 10 items
```

```
         */
        private int[] VALUES = new int[] {
              20, 24, 18, 70, 14, 23, 50, 17, 41, 21
        };

        /**
         * Maximum weight supported by the Knapsack
         */
        private int CAPACITY = 100;

        /**
         * This is the java implementation of the fitness function
         * for the Knapsack problem
         * @param individual
         * @return fitness as a value
         */
        private int fitnessFunction(BitVectorIndividual individual) {

              int totalValue = 0;
              int totalWeight = 0;

              for (int i = 0; i < individual.genome.length; i++) {
                    boolean isSelected = individual.genome[i];
                    if(isSelected) {
                          totalValue += VALUES[i];
                          totalWeight += WEIGHTS[i];
                    }
              }

              /* if the sack is under weight return the value as the fitness */
              if (totalWeight <= CAPACITY) {
                    return totalValue;
              }
              /* else return the amount it is overweight as a negative fitness
*/
              else {
                    return -(totalWeight - CAPACITY);
              }
        }

        /**
         * Callback method of the ECJ framework which is used to evaluate
         * any individual against the fitness function.
         */
        public void evaluate(final EvolutionState state, final Individual ind,
                    final int subpopulation, final int threadnum) {
              if (ind.evaluated)
                    return;

              if (!(ind instanceof BitVectorIndividual))
                    state.output.fatal("Whoa!  It's not a
BitVectorIndividual!!!", null);

              BitVectorIndividual ind2 = (BitVectorIndividual) ind;
              float fitnessValue = (float)this.fitnessFunction(ind2);

              if (!(ind2.fitness instanceof SimpleFitness)) {
                    state.output.fatal("Whoa!  It's not a SimpleFitness!!!",
  null);
                    return;
              }

              SimpleFitness fitness = (SimpleFitness)ind2.fitness;
              fitness.setFitness(state, fitnessValue, false);
              ind2.evaluated = true;
        }

        /**
         * This method is optional. However it can be used to improve the
         * verbosity of the statistical output of GA.
         */
        public void describe(final Individual ind, final EvolutionState state,
                    final int subpopulation, final int threadnum, final int
  log,
                    final int verbosity) {
        }
  }
```

Section D – Running the program

The program was tested under java version 1.6.0_17 in Windows Vista operating system.

Copy the “**Implementation**” folder found in the CD to the hard disk.
Go to Assignment\Implementation\bin from command prompt.

Type the following command.
java ec.Evolve -file knapsack.params

If it worked you should get a sample output in the command prompt and also written to a newly created/modified file with name “**$out.stat**”

Section E – Sample Output ($out.stat)

```
Generation: 0
Best Individual:
Evaluated: T
Fitness: 128.0
 1 1 0 1 1 0 0 0 0 0

Generation: 1
Best Individual:
Evaluated: T
Fitness: 128.0
 1 1 0 1 1 0 0 0 0 0

Generation: 2
Best Individual:
```

```
Evaluated: T
Fitness: 128.0
 1 1 0 1 1 0 0 0 0 0

Generation: 3
Best Individual:
Evaluated: T
Fitness: 134.0
 0 0 0 1 1 0 1 0 0 0

Generation: 4
Best Individual:
Evaluated: T
Fitness: 134.0
 0 0 0 1 1 0 1 0 0 0

Generation: 5
Best Individual:
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1

Generation: 6
Best Individual:
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1

Generation: 7
Best Individual:
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1

Generation: 8
Best Individual:
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1

Generation: 9
Best Individual:
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1

Generation: 10
Best Individual:
```

```
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1

Generation: 11
Best Individual:
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1

Generation: 12
Best Individual:
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1

Generation: 13
Best Individual:
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1

Generation: 14
Best Individual:
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1

Generation: 15
Best Individual:
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1

Generation: 16
Best Individual:
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1

Generation: 17
Best Individual:
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1

Generation: 18
Best Individual:
```

```
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1

Generation: 19
Best Individual:
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1

Best Individual of Run:
Evaluated: T
Fitness: 155.0
 0 0 0 1 1 0 1 0 0 1
```