

# Assignment #7 – Genetic Algorithms

## The Knapsack Problem

Due: November 10<sup>th</sup>, 2011

The goal of this assignment is to write a genetic algorithm that solves the Knapsack Problem. Briefly stated, the Knapsack Problem goes like this:

You have a collection of  $N$  objects of different weights,  $w_1, w_2, \dots, w_n$ , and different values,  $v_1, v_2, \dots, v_n$ , and a knapsack that can only hold a certain maximum combined weight  $W$ . You would like to get a set of objects of maximal value into the knapsack.

As a search problem, the knapsack problem turns out to be intractable – there is no way to search that is efficient, reducing the search to an exhaustive check of all possible combinations of objects, and the time to solve it grows exponentially with the number of objects.

As a genetic algorithm, however, solutions that come extremely close to the maximum, while not guaranteed to actually be the maximum, can be found very quickly.

### Your Assignment

Write a GENETIC ALGORITHM (using LISP or any programming language you prefer as long as you have checked with the TA in advance for non-LISP languages) that solves the Knapsack problem. If you write it in LISP, your program should run with the following top-level calls (in other languages, suitable arrangements must be made to permit adjustment of the same set of parameters:

**(knapsack NumIterations ObjectList MaxWeight PopulationSize MutationPct)**

Your program will start by creating **PopulationSize** random members of the population (including a computation of the fitness function for each one). It will then loop iteratively **NumIterations** times, performing the following functions:

- Randomly select a pair of parents to breed.
- Pick a random spot for crossover, and breed two new children (with fitness computed).
- Randomly decide whether to mutate based on **MutationPct**, and if so, mutate one gene.
- Kill off the two weakest members of the population, to keep the size constant.

The genome can be simply a list of 1's and 0's, indicating whether each element of **ObjectList** either is or is not in the knapsack (see example below). You'll also want to store its fitness value.

The fitness function can be equally simple: it is simply the total **value** of the objects in the knapsack, unless the **weight** of the objects would be higher than **MaxWeight**, in which case the fitness is 0 (or at any rate, smaller than any legitimate non-overweight knapsack; you might find some clever uses of negative numbers for very large populations, but typically the overweight knapsacks will disappear from your population very quickly).

Your program should end by reporting the best remaining member of the population at the end of the run, along with the actual **weight** and **value** of that member.

## Example

Suppose you've got four objects:

Name	Weight	Value
A	45	3
B	40	5
C	50	8
D	90	10

and a knapsack that can support a maximum weight of 100 pounds. Let a genome be a pair showing the fitness value and then a list of 0s and 1s showing whether each item is in the knapsack.

Here are some of the genomes you might randomly generate, along with their fitness values:

(3 (1 0 0 0)) [A is in the knapsack; total value is 3]  
(8 (1 1 0 0)) [A and B are in the knapsack; total value is 8]  
(0 (1 1 1 1)) [A, B, C, and D are all in; fitness value is 0 because the bag is overweight.]

If you picked the first and third to cross over, and you chose randomly to cross them over starting at the third item, your two new child genomes would be

(0 (1 0 1 1)) and (8 (1 1 0 0))

If you then randomly chose to mutate the last gene of the first child, it would turn into

(11 (1 0 1 0)) [A and C are in the knapsack; total value is 11]

at which point, if your maximum population size was 3, you would kill off the members valued at 0 and 3, keeping the other three (even though two of them are the same). Then you would repeat the breeding, mutation, and death cycle until you had gone through the specified number of iterations.

## What to Hand In

After you've written the program, it is up to you to experiment with population size and number of iterations that produces good results. Since not very much happens in a single iteration, you may need a very large number to get a good result, but that is what you will determine.

As usual, turn in a well-commented listing of your program, and a runtime script showing your programming running on the test data. **Use the test data file from on the website (also available in ~lib220/asst7).**

Your transcript should be that of the best solution you have found for the sample data **that takes no more than two minutes to run** (real time, not CPU cycles). Please clearly indicate the population size and number of iterations you chose for that run.

## **Optional Extensions**

There are numerous extensions to this very basic genetic algorithm, as discussed in class. If you are interested in genetic algorithms, I would suggest you consider incorporating your ideas into a final project rather than working more on this particular assignment. But if you'd just like to work on this a little more and still do a separate final project, you can add different mechanisms for crossover, or mutation, roulette or rank fitness, random death (with or without elitism), etc. We will give up to 20% extra credit for assignments that go beyond the basic requirements, depending on the amount of effort and independent learning involved.