

Lab 2

CS 3793 – Fall 2008
Tom Bylander, Instructor

assigned October 6, 2008
due midnight, October 24, 2008

In Lab 2, you will program an agent for generic algorithms that will work for the “allones,” “nqueens,” and “knapsack” environments. These environments return the fitness of solutions encoded as bit strings. Your agent will generate candidate bit strings using a standard generic algorithm. Your grade on the lab will depend on the performance of your lab when I run it on test problems.

Environments

The “allones” environment implements a fitness function for testing genetic algorithms. The maximum fitness is achieved when the bits are all ones. Here is an example where I am entering bit strings:

```
> $A/allones -n
start 5
01010
2
11100
3
00000
0
11111
5
quit
maximum fitness 5
```

The “nqueens” environment implements a fitness function for the n -queens problem. The maximum fitness is achieved when there is one queen in each row and column, but no two queens are on different rows, columns, and diagonals. Here is an example where I am entering bit strings (note that the option is -q, not -n):

```
> $A/nqueens -q 4
start 16
1111111111111111
-23
1000010000100001
13
0100000110000010
16
1010101010101010
0
quit
maximum fitness 16
```

The “knapsack” environment implements a fitness function for the knapsack problem. The maximum fitness is achieved when the selected items maximize the value without exceeding the weight. The values can be seen by selecting one bit at a time, but the weights are hidden. Here is an example where I am entering bit strings:

```
> $A/knapsack -n 4
start 4
1100
-1
1001
6
1101
-3
0101
8
0111
-3
q
maximum fitness 8
```

For each environment, using the -h option will provide an explanation how the bits encode a candidate solution.

Genetic Algorithms

The generic algorithm agent should follow this pseudocode (cf. Figure 4.17)

```
currentPopulation ← set of 100 random bit strings
determine fitness of each individual in currentPopulation
loop 100 times
  nextPopulation ← empty set
  loop 50 times
    parent1 ← random selection from currentPopulation
    parent2 ← random selection from currentPopulation
    with probability = crossover probability do
      child1, child2 ← crossover of parent1 and parent2
    else
      child1, child2 ← parent1, parent2
    randomly change bits in child1 and child2 according to mutation probability
    add child1 and child2 to nextPopulation
  currentPopulation ← nextPopulation
  determine fitness of each individual in currentPopulation
return individual with the highest fitness
```

Larger populations and more generations might produce better results, but please adhere to 100 generations with 100 individuals each.

Random Selection

I suggest using *tournament selection* for random selection. The way this works is to randomly draw k individuals and select the one with the highest fitness. I suggest $k = 2$ or $k = 3$.

Crossover

The book discusses one-point crossover, but two-point crossover or uniform crossover might be more appropriate. In one-point crossover, each parent is split into two strings at the same random point and the suffixes are switched:

parents		children
10101010		101 10 101
+	⇒	+
01010101		0100 10 10

In two-point crossover, each parent is split into three strings at the same random points and the middle part is switched:

parents		children
10101010		101 10 110
+	⇒	+
01010101		0100 10 01

In uniform crossover, each bit is randomly switched with a 50% probability.

parents		children
10101010		00111 100
+	⇒	+
01010101		11000 011

Crossover and Mutation Probabilities

When two parents are selected, the probability that crossover is performed is governed by the crossover probability. If crossover is not performed, then the children are identical to the parents. I suggest trying crossover probabilities between 0.5 and 1.

Mutation of a child can be performed bit by bit. That is, for each bit, the probability of changing the bit is the mutation probability. Typically, this is set to be a low probability such as 0.01, but if the crossover probability is low and the bit strings are short, you might get too little change from one population to the next. Maybe 1 divided by the string length is a better choice.

Turning in Your Lab

Somewhere in your directory, you should create a `lab2` subdirectory. This directory *must* have a `Makefile` (or *must* have an executable file `make_lab2`) that compiles and links your agent programs, which *must* be named `lab2`. Any source code that is linked to these programs *must* be in this directory. That is, you should be able to copy the files in your `lab2` directory to another directory and be able to run the following command sequence.

```
setenv A /home/bylander/agents/linux/bin or A=/home/bylander/agents/linux/bin
/bin/rm lab2 *.o
make
$A/interact $A/allones ./lab2
$A/interact $A/nqueens ./lab2
$A/interact $A/knapsack ./lab2
```

Zip or tar the entire directory and submit it using WebCT. If you submit multiple times, only the last one is kept. Be sure that WebCT has registered your submission.

In addition, provide a brief report describing the performance of your program. It should describe its performance on each environment and different size bit strings in each environment.

I will be running your lab on successively larger bit strings and will be looking at the quality of your answers and the amount of time it takes. You will get a grade of 0 until I can run your lab.