Yang_Jinxin_1168646_homework-4-Q2

BY JINXIN YANG

brianyang1106@gmail.com

November 30, 2012

1 Description

This question is to use genetic algorithm to search for a maximum of a fitness function. We need to use streams of bits as a binary representation when dealing with sets of even and odd pairs. Apart from implementing the algorithm itself to solve the problem, we also should change the value of one of the parameters at a time to compare the performances of different parameters and check where the best results are from. For the directory of this homework-4-Q2:

- *src*: *h4* is a Java project file folder, you can just import this directory as "Existing Projects into Workspace" by Eclipse (Windows or Linux) and run *src→h4→genetic.java*
- *outputs*: all the direct outputs from eclipse
- *graphs*: all the Gnuplot generated eps images and screenshots
- *data*: organized data for Gnuplot and Gnuplot plotting script

I will go though the logic of my program to explain the work I did.

2 Data Generation

To make the parameter testing more flexible, I used dynamic array to store the sets of pairs `L[A][B]`, which means I can pass the sizes of examples to program and it will automatically run as the algorithm. In this question, `A` is an even and `B` is an odd (both of them from 6 to 27), I wrote a small function which can generate the specific range of random integers with controlling whether is even or odd. We want the program show the progress and performance of genetic algorithm, so I set (6∼18) as random generating range. The output of program starts with the parameters Fig. 1 and original generated pairs which shows in Fig. 2.

```
L = 80
Pco = 0.8
Pmut = 0.2
Threshold = 0.8
Mask: 1111100000
```

Figure 1. Initial parameters

3 Sort and Replication

In this program, I try to make everything dynamic and simple, so all the changes for testing is changing the parameters in the main function and the example numbers of the first member attribute (`L`) of class. So,

when generated the proper data, program steps into one iteration.

First we do the population sorting and replication. By calculating the fitness function, we got the results of different pairs and do the sort. Since I need to track every pair in the future (for mutation choice), I used an mapping array (`map[x][y],[x] stores index,[y] stores fitness value`) to record the original order of generated data, even the pairs will change, but the index of each pair will keep still. I used *selection sorting* algorithm to sort the `map[][y]` and trace the pair through `map[x][]`. After sorting (which is before crossover), Fig. 3:

0 iteration:

Origin solutions
(6,11) F= 179
(14, 7) F= 399
(10,13) F= 373
(10,17) F= 457
(10, 9) F= 289
(14,13) F= 573
(14,13) F= 573
(16,15) F= 751
(14,11) F= 515
(14, 9) F= 457
(16,17) F= 817
(14,17) F= 689
(12,15) F= 519
(10,11) F= 331
(10,15) F= 415
(12, 9) F= 369
(12,13) F= 469
(12,17) F= 569
(16,11) F= 619
(16, 7) F= 487

Figure 2. Original data

Before crossover
(16,17) F= 817
(16,15) F= 751
(14,17) F= 689
(16,11) F= 619
(14,13) F= 573
(14,13) F= 573
(12,17) F= 569
(12,15) F= 519
(14,11) F= 515
(16, 7) F= 487
(12,13) F= 469
(10,17) F= 457
(14, 9) F= 457
(10,15) F= 415
(14, 7) F= 399
(10,13) F= 373
(12, 9) F= 369
(10,11) F= 331
(10, 9) F= 289
(6,11) F= 179

Figure 3. Before crossover

After crossover
(16,17) F= 817
(16,15) F= 751
(14,17) F= 689
(16,11) F= 619
(12,13) F= 573
(14,13) F= 573
(12,15) F= 569
(12,21) F= 519
(14, 7) F= 515
(16,11) F= 487
(12,17) F= 469
(10,13) F= 457
(14,15) F= 457
(8, 9) F= 415
(14,13) F= 399
(10, 7) F= 373
(12,11) F= 369
(10, 9) F= 331
(14,11) F= 289
(6, 9) F= 179

Figure 4. After crossover and mutation

Though $(1-\mathrm{Pco})L$, we know the replication number which is fixed for one certain test. We don't do the replication here, because we can leave the first ones themselves and just do the crossover and mutation to other pairs. As the Fig. 3 shows, *first four* will be the replication ones (first four doesn't change). Also, we can see the boundary in Fig. 5 (see Sec. 4).

4 Crossover and Mutation

For crossover, first to turn all the integers into binary representation, Java has the function to do it automatically but it misses the leading zeros where I add a new function to fix it (`fixLeadingZeros()`). Since the rest of pairs except replication ones is even, *we could do the crossover between neighboring two pairs*. In this homework I set the mask as "1111100000" which means two pairs just do the cross exchange to get the new two off-springs. It is easy and reasonable. By using String operations, I cut the bits stream into half and recombined them.

After crossover, based on the Pmut $\cdot L$, we know the number of mutations, here we *randomly* choose the mutation pair and position. But for the correctness (A is even, B is odd, between 6 and 27) of pairs after mutation, I used a function to filter the mutations which lead to the wrong solution and return the right mutation position (10 bits, 1 position). Then we do XOR to mutate that position and return the new bits stream back to update all the pairs and do the threshold checking. Fig. 4 shows the result after crossover and *mutation*. To make a explicit comparison, I output a combined one (arrow ← shows the mutation pair):

```
before      after      bits:
--------------------------------
(16,17)    (16,17)    1000010001
(16,15)    (16,15)    1000001111
(14,17)    (14,17)    0111010001
(16,11)    (16,11)    1000001011
--------------------------------
(14,13)<-  (12,13)    0110001101
(14,13)<-  (14,13)    0111001101
(12,17)    (12,15)    0110001111
(12,15)    (12,21)    0110010101
(14,11)    (14, 7)    0111000111
(16, 7)    (16,11)    1000001011
(12,13)    (12,17)    0110010001
(10,17)    (10,13)    0101001101
(14, 9)<-  (14,15)    0111001111
(10,15)    ( 8, 9)    0100001001
(14, 7)<-  (14,13)    0111001101
(10,13)    (10, 7)    0101000111
(12, 9)    (12,11)    0110001011
(10,11)    (10, 9)    0101001001
(10, 9)    (14,11)    0111001011
( 6,11)    ( 6, 9)    0011001001
```

Figure 5. Comparison

Also, for each run, it will show the mutation pair and concrete mutation bit with a before and after mutation comparison just like below:

```
Mutation pairs:
(14,27)
before: 0111011011
 after: 0111010011

(14,15)
before: 0111001111
 after: 0110001111
```

Figure 6. Mutation pairs

For the record, the *after* condition Fig. 5 shows is an unsorted array, which will be calculated the fitness and sorted immediately when the next iteration begins. So when we choose the best solution, it will be the

best because we just pick up the first one which from a sorted array and reaches the threshold.

5 Parameters Test

Since the original generated data is between 6 and 18, it might be hard to reach the threshold within few iterations. But anyway, we have four parameters to change, test and check how many iterations they need respectively: *Pco*, *Pmut*, *number of examples* and *threshold*. The combination I used to test is:

```
Pco=0.8, Pmut=0.2, Threshold=2000, example=20/40/80
Pco=0.8, Pmut=0.1, Threshold=2000, example=20/40/80
Pco=0.6, Pmut=0.2, Threshold=2000, example=20/40/80
Pco=0.6, Pmut=0.1, Threshold=2000, example=20/40/80
Pco=0.8, Pmut=0.2, Threshold=1800, example=20/40/80
Pco=0.8, Pmut=0.1, Threshold=1800, example=20/40/80
Pco=0.6, Pmut=0.2, Threshold=1800, example=20/40/80
Pco=0.6, Pmut=0.1, Threshold=1800, example=20/40/80
```

Figure 7. Parameters combination

For every combination, the program will run the entire algorithm for 10 times and record each time's iteration that reached the threshold. Then after all the works done, it will show the pair which not only has the biggest fitness value but also reaches the threshold and tell the program to abort normally. Here is the screenshot of final result (the output file is too big to display here):

```
( 6, 7)<-  ( 6, 7)    0011000111
( 6, 7)    ( 6, 7)    0011000111
( 6, 7)    (22, 7)    1011000111

9 iteration:

Best solution:
(26,23)    1101010111

9 iterations reach the threshold
All works done!

Test 0: 13
Test 1: 21
Test 2: 9
Test 3: 22
Test 4: 7
Test 5: 10
Test 6: 7
Test 7: 13
```

Figure 8. Result Sample

The very last value stands for the average iterations each combination (Fig. 7) has. I collected all the three versions (Examples: 20, 40, 80) average iterations and plotted as below:

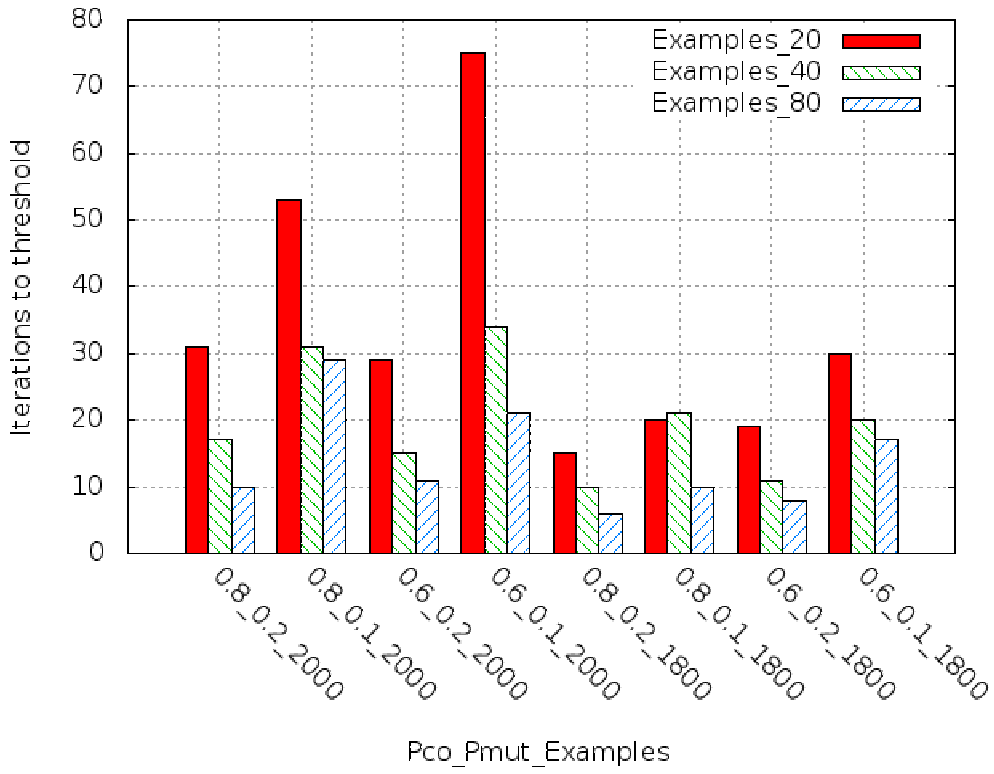

Figure 9. Result Data

6 Conclusion

Fig. 9 indicates the following:

- within one combination, the ***more*** examples, the fewer iterations
- for the same Pco, Pmut and number of examples, the ***lower*** threshold, the fewer iterations
- for the same Pco, number of examples and threshold, the ***larger*** Pmut, the fewer iterations
- for the same Pmut, number of examples and threshold, the ***larger*** Pco, the fewer iterations

7 Acknowledgment

Since this is the last homework we have, I want to give special acknowledgment to Kinjal, our Machine Learning TA. Thank you for what you did for all of us, you really help me a lot to through my first semester.

BEST WISHES

Brian Yang