Otto-Friedrich-Universität Bamberg

Professur für Kognitive Systeme

Master Project Cognitive Systems

Learning Grammars with Swarm Genetic Approach

Submitted by:

Cano Fco.Javier

Macias Fernando

Milovec Martina

Supervisor: Prof. Dr. Ute Schmid

Bamberg, Summersemester 2012

Abstract

We present a program for exploring pain grammar with approach of genetic algorithms using swarm intelligence. The pain grammar contains sequences which are actually gestures that people (in our case young female) do when they suffer under moderated pain so called 'action units' (facial signals). According to the genetic algorithm approach we use these principles which helps us with mutation and fitness function. Mutation helps us to delete or add rules and symbols showing different possibilities and options for implementation. Fitness function selects the sequences and defines which grammar is the best, also punishes grammars which are not good. Swarm intelligence approach helps us that grammars converge to the best grammar and which in evolutionary way creates a compact grammar, that defines the structure of the sequences.
Keywords: genetic algorithms, swarm intelligence, grammars, mutation, fitness function.

Contents

1 Introduction

Research work in Genetic algorithms and programming approach for exploring pain grammars (Schmid, Siebers, et. al., 2012) has a big importance for this project. According to their research results, we used their data for our program developing. The program is based on genetic programming using swarm intelligence approach, which in evolutionary way creates a compact grammar, that defines the structure of the sequences (gestures that people do when they suffer under moderated pain). Genetic algorithms help us to use pain grammar, do the mutation part with sequences and select them according to fitness function[4]. Fitness function has for importance to define good grammars and to punish grammars which are not good. Swarm intelligence is used to converge grammars to the best chosen grammar. The result is the best compact grammar with sequences.
In the following section we will discuss how the swarm intelligence and its approach works. Afterwards are discussed genetic algorithms and context independent grammars. In the third chapter we describe how we use genetic algorithms according to the program structure and we will show how we implemented the swarm algorithm, fitness evaluation algorithm and mutation in the program.

2 Swarm Intelligence and Grammars

A single ant is not smart, but ants colonies are. If we would take a closer look at one individual ant, we would see that one ant as an individual can not accomplish much and it would act as confused. Ants colonies are smart, they respond quickly and effectively to solve problems, like defending territory from other bugs, finding food, etc. What they do is called swarm intelligence. Where such intelligence comes from and the study of swarm are important questions for humans to get inside of the swarm intelligence and to manage complex systems. Swarm intelligence works on a principle of simple rules being applied on local information, so called self-organized system. This principle works also for solving particularly complex human problems, for example a strategy to manage a complex business problem [7], which can save business from bankruptcy. Cooperation methods of swarm intelligence have been applied also to collective mobile robotics and multi-agent systems [5]. In before mentioned ant colonies, through specialization the ants colonies allocate different tasks among individuals adaptively and flexibly, resulting in improvement of their fitness to the environments, etc. These are important tasks how specialization can be advanced in Multi-Robot Systems, helping the robots to adapt themselves to the unknown dynamic environments and fulfil their missions efficiently[5]. For example in [6], swarm intelligence approach is used to show how cellular robotic systems are capable of 'intelligent' behaviour cooperating to accomplish global task.

In our project we are dealing with a problem of pain grammars using principles as genetic algorithms and programming as swarm intelligence. Particle swarm optimization has a multidisciplinary character such as artificial intelligence, psychology, engineering, computer science and evolutionary computation[2]. The principle of swarm helps us to generate short and good grammars. In our project, the populations are individual grammars that explore through the problem hyperspace. When the population is initialized, individual grammars are given initial empty values. According to the fitness function, the best individual must be chosen and others individuals (not that good as the best one) from the rest of population must converge to the best one. Here, the goal of fitness evaluation is to define how good is an individual grammar and to generate direction, for having short but good grammars.

3 Genetic Algorithms and Context Independent Grammars

In this chapter we discuss how we use GA approach for the context independent grammars.
Genetic Algorithms (GA) define elements, like population of grammars, selection according to fitness, and in our case random mutation. Genetic algorithms create a string of numbers that represents the solution and each possible solution so called individual. GA follow rules to generate useful solutions for optimization and search problems, by an heuristic search that mimics the process of natural evolution. GA help us to solve grammar problem, which we explain in following.
A formal grammar is a set of rules for strings in a formal language, which describes how to form strings from the language's alphabet that are valid according to the language's syntax. A grammar describes the form of the strings. Context free grammar is a formal grammar in which every production rule is of the form

$$S \longrightarrow a$$

where S is a single non terminal symbol, and a is a string of terminals and/or non terminals (a can be empty). Context free grammar describes the structure of programming languages and other formal languages. A context-sensitive grammar is a formal grammar in which the left sides and right sides of any production rules may be surrounded by a context of terminal and non terminal symbols. A formal grammar $G = (N, \epsilon, P, S)$ is context-sensitive if all rules in P are of the form

$$\alpha\beta \longrightarrow \upsilon$$

where $A \epsilon N$ (i.e., A is a single non terminal), $\alpha, \beta \epsilon (N \cup \sigma)*$(i.e., α and β are strings of non terminals and terminals) and $\upsilon \epsilon (N \cup \sigma)+$ (i.e., υ is a non empty string of non terminals and terminals).
In the following will be discuss how we use these principles in our program.

4 Program Structure

Figure 1 shows the program structure and important classes of our program. Swarm class is the main class in our program which implements grammar, sequence storage which implements parser class where are sequences reeded from a file, then fitness function with breadth algorithm and mutation classes. We will take a closer look to this classes later in this chapter.

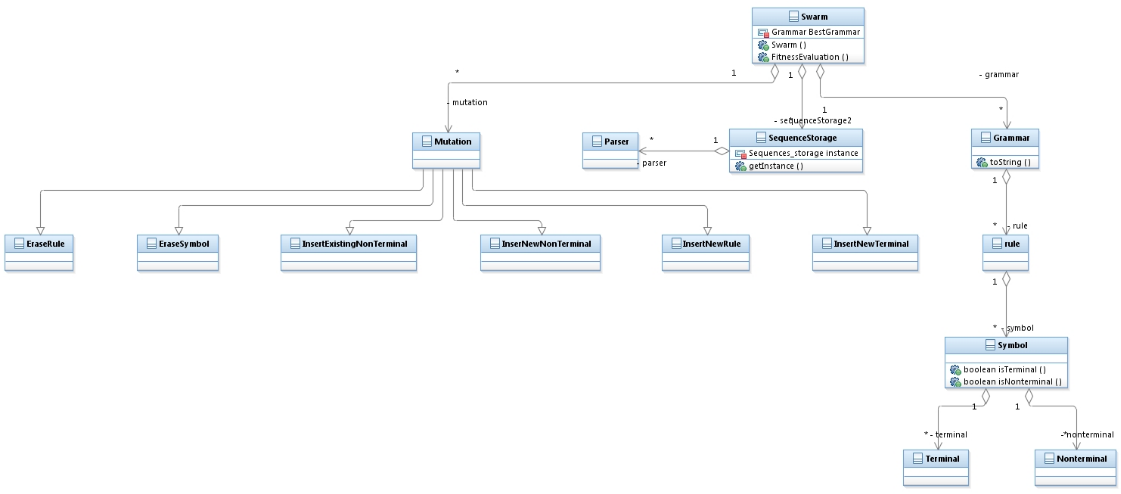

Figure 1: Program structure

4.1 The Swarm Algorithm

Swarm algorithm is developed to select grammars, apply randomly mutation, choose only the one best grammar and converge the half of remaining grammars to the best one. In following,we are showing how swarm algorithm works.

Input:
Number of generations n_{gen}, Offspring frequency o_{freq}, List of sequences S
G $\longleftarrow$ List of the individuals (grammars), initialized empty;
M $\longleftarrow$ Array of all the possible mutations;

```
while i<$n_{gen}$ do
```

```
    for each grammar g in G
        select pseudo randomly one mutation m from M;
        apply m to g;
     end
     if i%$o_{freq}$=0 then
        select grammar $best_g$ from G with highest fitness value (best-so-far individual)
            for half of the remaining grammars g in G
                g $\longleftarrow$ $best_g$
            end
    end
  end
return $best_g$ from G
```

4.2 Fitness Function and Grammar Initialization

For the ***Grammar Initialization***, at the beginning all rules are initialised as empty.

For the ***Fitness Evaluation***, our program calculates the fitness of each grammar by applying it to the input AU sequences, using the 'breadth' algorithm. How does 'breadth' algorithm works? For each grammar, it generates all the possible AU sequences that particular grammar allows. Afterwards, counts how many of the input AU sequences are actually generated, and subtracts to that number of rules of grammar. If the individuals are 'too big' they will be punished, because the goal is to have grammar as short as possible. The result of 'breadth' algorithm is an integer, and the given individual with the best result is the best-so-far individual and other individuals can converge to the best-so-far individual. In the following, we can see the fitness function algorithm.

```
Input Grammar G(N,T,S,P) and List of Words
    Initialize variables
    calulate value of maxLength
    add to Queue G(s)// Initial Symbol of the grammar
    while (Queue not empty) and List of Words not empty
        s = popQueue
        add to seen s
        if CheckTerminals (s) then
        while remove from List of Words do
            increase matches
        end
    else
        if $|s|$ <maxLength then
            split in prefix, first and suffix
            if prefix is contained in the list of Words ithen
                foreach rule in the Grammar do
                    if first is the nonTerminal in the rule then
                        add to list: preffix , rule rightSide and suffix
                        if list has never been seen and is not in Queue then
                            add list to Queue
                        end
                    end
                end
            end
        end
    end
end
return matches
```

4.3 Mutation

Mutation is one of the complex and important parts of program. We will show different mutations that can be used to generate the offspring. After applying any of this transformations, the program must check the consistency of the new grammar, and erase some of the rules if necessary, for example:

- rule alone must be valid (e.g. S $\longrightarrow$ aX is valid, but SS $\longrightarrow$ is not),
- rule has only one initial symbol 'S',
- we considered that they are not directly recursive, for example, if the non terminal symbol appears in the left side of a rule, it can not appear again in the right side,
- if a non terminal symbol appears in the right side of a rule, then it must also appear at least in the left side of another rule,

- grammar must contain at least one rule with only terminals in the right side (if it does not, the grammar gets into a loop that generates new non terminals).

Now we will present different types of mutation that will be randomly chosen by the program, with some probability, on each mutation step on the main algorithm for each individual.

New rule : inserting a valid new rule

Example of valid rules:

n0 $\longrightarrow$ S
S $\longrightarrow$ au7 X au6 au5 Y
X $\longrightarrow$ au 4 au8 au18-19
Y $\longrightarrow$ au3-au5 au10 Z

Example of invalid rule:

SS$\longrightarrow$

Erase rule : erasing an existing rule (program should also check if erased rule had non terminals on the right side, etc.(see the previous step))

Example of 'Erase Rule':

n0 $\longrightarrow$ S
S $\longrightarrow$ au7 X au6 au5 Y
X $\longrightarrow$ au4 au8 au18-19
Y $\longrightarrow$ au3-au5 au10 Z //if we want to erase this rule we will also erase a 'Z' rule

New symbol : adding a new terminal or non terminal (not existing ones) and adding an existing terminal and non terminal to the right side of an existing rule

Example of 'Add New Terminal':

X $\longrightarrow$ au4 au8 au18-19
result:
X $\longrightarrow$ au4 au8 au18-19 au76

<u>Example of 'Add New Non Terminal':</u>

Y $\longrightarrow$ au3-au5 au10 Z
result:
Y $\longrightarrow$ au3-au5 au10 Z W

<u>Example of 'Add Existing Non Terminal':</u>

Y $\longrightarrow$ au3-au5 au10 Z
result:
Y $\longrightarrow$ au3-au5 au10 Z Z

Erase symbol : erasing a terminal on the right side of the rule or non terminal also on the right side of the rule (program should also check if it occurs on the right side or not)

Example of 'Erase Existing Terminal':

X $\longrightarrow$ au4 au8 au18-19
result:
X $\longrightarrow$ au8 au18-19

Example of 'Erase Existing Non Terminal':

Y $\longrightarrow$ au3-au5 au10 Z
result:
Y $\longrightarrow$ au3-au5 au10

5 Conclusion and Results

The fitness function has been very difficult to implement, we have found a lot of different problems, for example with it's complexity and the number of calculations. So we have used some functions less 'complex' that solved our problems. For example, instead of the Lemma2 (check if there are words with correct prefix and do not add the word to q if is not the right one), we compared with the first non terminal found in the rule with the non terminal of the rule which we are checking. This helps us to avoid complex functions and therefore less calculations.

Due to the large number of calculations that program executes we had to choose between execution time or memory. Therefore we have chosen the execution time, as the following example of the code demonstrates:

```
 boolean prefixExists = false;

for (List<Terminal> l : ListWords) do
  if(l.size() >= prefix.size() && l.subList(0, prefix.size()).equals(prefix)) then
      prefixExists = true;
      break;
   end
end
```

Here we can see that if it is possible to generate a *Word* which we are working with, i.e. if the list of words contains this *Word*.

Another way in which we saved some memory was the use of Set< *List* < *Symbol* >> in which we inserted all the words we have seen. Duplicate words were then removed from the list.

In the next step, we have checked whether the queue contains the *word* or not. Here is an example of saving memory for execution time:

```
if(!seen.contains(list) && !Queue.contains(list) && !list.isEmpty() && list.size()
<= maxlength)
```

Therefore, the presented version is the most optimal time implementation that does not overflow the memory. However, the running time is still very long, and after running the program with the "sequences_young_female_4.csv" file (see Appendix) for almost 48 hours, we only got about 7 results, being 5 out of 59 being the highest coverage that we got. As an example, in the case of 500 grammars, 50 iterations and 15 offspring frequencies, the memory increased up to 1MB and was running for a very long time. After having done these verifications we could not identify the problem nor solve it, and, accordingly, we were not able to obtain the expected results. After checking all the errors detected in our program, we had to face the problem of the memory/time balance. The first corrected version was oriented towards time efficiency but in that case, the stack could not handle it and we got an overflow.

For further development, we think the fitness evaluation should be optimized and the full program checked. Besides, the parameters like number of grammars, number of iterations, probability of each mutation, etc. should be adapted. Once this is done, the program could be adapted to context dependent grammars without any problems.

In order to be able to present some actual results, we created a simple input file "test.csv" (see Appendix) and ran again the program. The grammar the program should generate will be similar or equivalent to:

```
n0 -> a
n0 -> a n1
n1 -> b
n1 -> n1 n2
n2 -> c
```

The language recognized by this grammar is a + abc*, so all the sequences in "test.csv" have this format.

After some time, we got an output file "results_test.csv" (see Appendix) that we could observe and furthermore use to generate the following graphics, which compare the average fitness value (punished with the number of rules of the grammar) for each parameter.

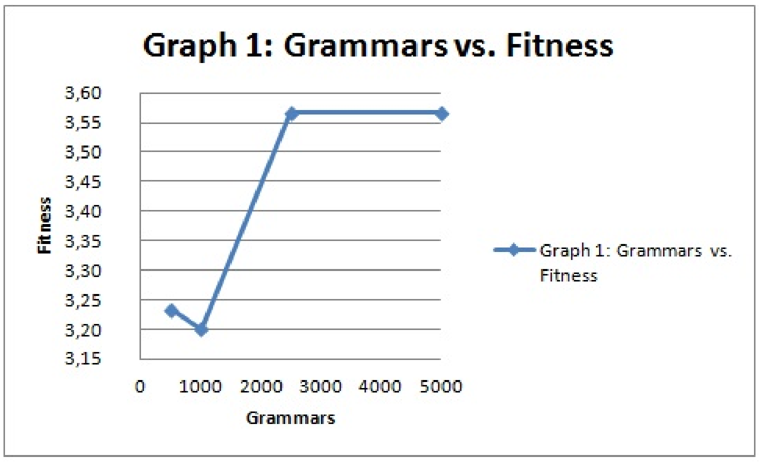

Figure 2: Results of test.csv file

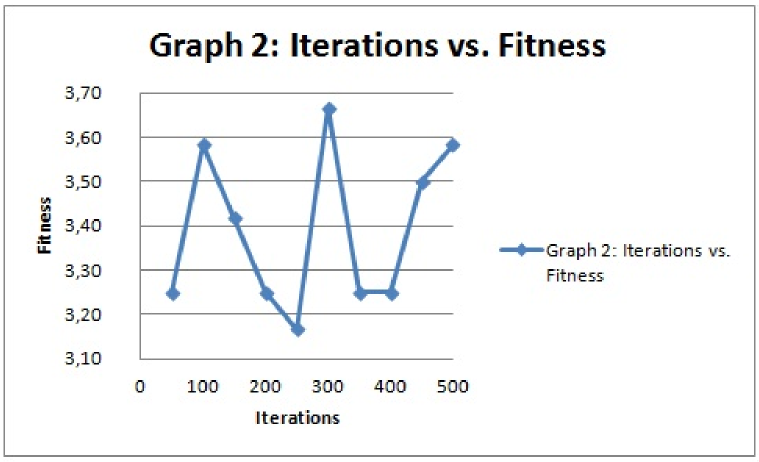

Figure 3: Results of test.csv file

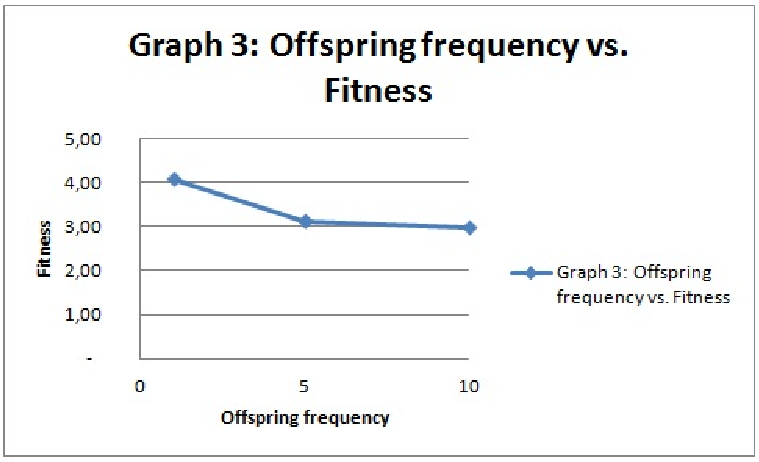

Figure 4: Results of test.csv file

References

[1] Schmid, U., Siebers, M., Seuß, D., Kunz, M., Lautenbacher, S. (2012). *Applying Grammar Inference To Identify Generalized Patterns of Facial Expressions of Pain.* JMLR: Workshop and Conference Proceedings, 1-13

[2] Kennedy, James F. (2001). *Swarm intelligence.* San Francisco: Morgan Kaufmann Publishers

[3] Miller Peter (2007). *The Genius of Swarms.* National Geographic. Source: http://ngm.nationalgeographic.com/2007/07/swarms/miller-text/1

[4] Mitchell, M. (1997). *An Introduction to Genetic Algorithms.* Massachusetts: Cambridge Publisher

[5] Zhang,D., Xie, G., Yu, J., Wang, Long. (2007). *Adaptive task assignment for multiple mobile robots via swarm intelligence approach.* Robotics and Autonomous Systems 55, 572–588

[6] Beni, G., Wang, J. (1993). *Swarm Intelligence in Cellular Robotic Systems.* Computer Science Robots and Biological Systems: Towards a New Bionics?, Vol. 102, Part 7, 703-712

[7] Bonabeau, E., Meyer, C. (2001). *Swarm Intelligence: A Whole New Way to Think About Business.* Harvard Business Review, 104-114

Appendix

sequences__young__female__4.csv file:

1 au38,au12,au7,au25,au6,au4,au9
2 au14,au38
3 au7,au4,au12,au39,au4
4 au4
5 au1-2,au12,au7
6 au7,au17
7 au7,au10
8 au6-7,au14,au15-17,au4
9 au4,au7,au6
10 au7,au10,au4,au1
11 au25,au25
12 au1-2,au6,au7,au1
13 au1,au2
14 au7,au1-2,au12
15 au7,au9,au4
16 au2
17 au12,au25,au6
18 au12
19 au4,au7
20 au4-7,au24,au4,au4,au10,au14,au12-25,au6,au1,au26,au4
21 au4-7,au10,au7,au20,au25,au12,au24,au7,au4
22 au4,au10,au6,au20,au7-25,au15,au43,au18,au43,au25,au20,au1,au2,au4,au1,au2,au4
23 au4-7,au38,au4,au4-7,au6,au9,au1,au25,au32
24 au7,au9,au4,au6,au20,au12,au25,au1,au12,au4,au9
25 au10,au4,au7,au6,au14,au9,au25,au12
26 au7,au7
27 au7
28 au7
29 au4
30 au6-7,au43,au4,au9,au25,au12,au26,au5
31 au7,au9
32 au7,au4
33 au6-7,au4,au6,au6,au9,au6-7,au9
34 au7,au6
35 au6-7,au6-7
36 au17,au14
37 au7

38 au7
39 au7,au4,au10-25
40 au7
41 au4,au7
42 au7,au6,au4,au12
43 au4-7,au6,au12
44 au4,au7
45 au12
46 au7,au7,au4
47 au2
48 au1,au2,au7
49 au1-2,au7,au14,au1,au2
50 au2,au1,au26,au18,au23,au14,au23,au14,au2,au1,au7,au4
51 au2
52 au26,au1-2,au14,au7
53 au14
54 au1-2,au7,au12
55 au14
56 au14
57 au4,au7
58 au17,au12,au25,au19,au12
59 au17,au14,au10

test.csv file:

```
1 a,b
2 a,b,c
3 a,b,c
4 a,b
5 a,b,c,c
6 a,b,c,c,c,c,c,c
7 a
8 a
9 a,b
10 a,b,c,c
11 a,b,c,c,c,c
12 a,b,c,c
13 a,b
14 a
15 a,b,c,c,c
16 a,b,c
```

results_test.csv file:

```
GRAMMARS;ITERATIONS;OFFSPRING_FREQ;FITNESS
500;50;10;3
1000;50;10;3
2500;50;10;3
5000;50;10;3
500;50;5;3
1000;50;5;3
2500;50;5;3
5000;50;5;4
500;50;1;3
1000;50;1;3
2500;50;1;4
5000;50;1;4
500;100;10;3
1000;100;10;2
2500;100;10;3
5000;100;10;3
500;100;5;3
1000;100;5;3
2500;100;5;3
5000;100;5;3
500;100;1;8
1000;100;1;4
2500;100;1;4
5000;100;1;4
500;150;10;3
1000;150;10;3
2500;150;10;3
5000;150;10;3
500;150;5;3
1000;150;5;3
2500;150;5;3
5000;150;5;3
500;150;1;3
1000;150;1;4
2500;150;1;6
5000;150;1;4
500;200;10;3
```

39 1000;200;10;3
40 2500;200;10;3
41 5000;200;10;3
42 500;200;5;3
43 1000;200;5;3
44 2500;200;5;3
45 5000;200;5;3
46 500;200;1;3
47 1000;200;1;4
48 2500;200;1;4
49 5000;200;1;4
50 500;250;10;3
51 1000;250;10;3
52 2500;250;10;3
53 5000;250;10;3
54 500;250;5;3
55 1000;250;5;3
56 2500;250;5;3
57 5000;250;5;3
58 500;250;1;3
59 1000;250;1;3
60 2500;250;1;4
61 5000;250;1;4
62 500;300;10;3
63 1000;300;10;3
64 2500;300;10;3
65 5000;300;10;3
66 500;300;5;3
67 1000;300;5;3
68 2500;300;5;3
69 5000;300;5;3
70 500;300;1;3
71 1000;300;1;4
72 2500;300;1;7
73 5000;300;1;6
74 500;350;10;3
75 1000;350;10;2
76 2500;350;10;3
77 5000;350;10;3
78 500;350;5;3
79 1000;350;5;3

80 2500;350;5;3
81 5000;350;5;3
82 500;350;1;4
83 1000;350;1;4
84 2500;350;1;4
85 5000;350;1;4
86 500;400;10;3
87 1000;400;10;2
88 2500;400;10;3
89 5000;400;10;3
90 500;400;5;3
91 1000;400;5;3
92 2500;400;5;3
93 5000;400;5;4
94 500;400;1;3
95 1000;400;1;4
96 2500;400;1;4
97 5000;400;1;4
98 500;450;10;3
99 1000;450;10;4
100 2500;450;10;3
101 5000;450;10;3
102 500;450;5;3
103 1000;450;5;3
104 2500;450;5;4
105 5000;450;5;4
106 500;450;1;3
107 1000;450;1;4
108 2500;450;1;4
109 5000;450;1;4
110 500;500;10;3
111 1000;500;10;3
112 2500;500;10;3
113 5000;500;10;4
114 500;500;5;3
115 1000;500;5;3
116 2500;500;5;4
117 5000;500;5;3
118 500;500;1;4
119 1000;500;1;4
120 2500;500;1;4

121 5000;500;1;5