Final project

Genetic Algorithm of Sliding Number Game

Team 535:

You Fu: 001497262

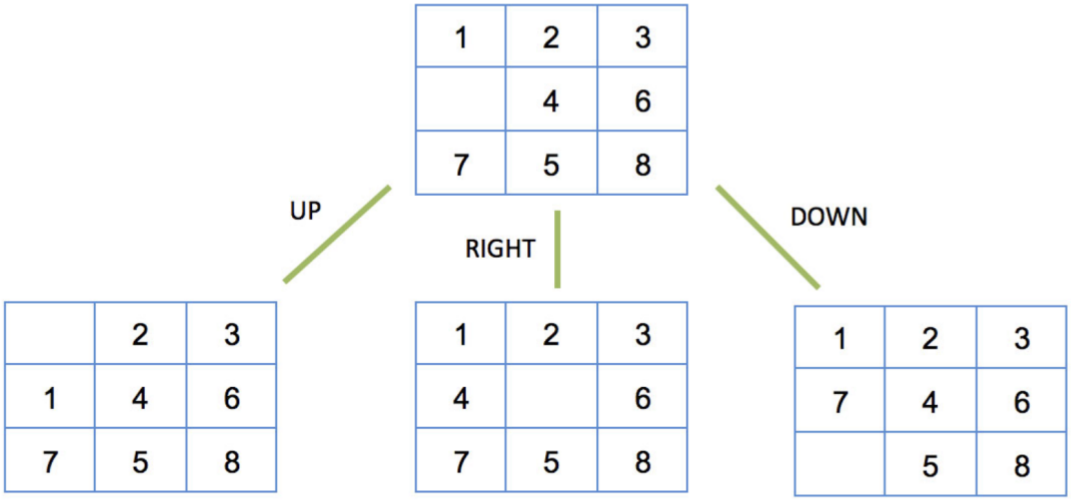
Yuzhen Han: 001443540.

Zenan Lin: 001403847.

4/20/2019

1. Problem Description

Sliding block is a traditional Chinese problem. There are n^2-1 numbers in an n-order square. Player can only move one block adjacent the only space in one time. The goal of this game is to place the random numbers in order within the shortest time (which can be regarded as in the lest steps)



1	2	3
	4	6
7	5	8

	2	3
1	4	6
7	5	8

1	2	3
4		6
7	5	8

1	2	3
7	4	6
	5	8

As known, genetic algorithm can be used to generate a better solution for optimization. So, our goal in this project is to solve the sliding block problem using genetic algorithm. We set the original state of the number panel as the start point and the reorder panel as the endpoint. Every step will be decided by genetic activities like mutation, crossover and selection. The output in the result shows when we get the answer and how to move those blocks in this solution.

2. Design and implementation

2.1. Problem Design

Based on the principle of this problem, the design of a problem panel has to be in some limitation. For example, if you want to only exchange the last two numbers, say seven and eight, that is impossible. As a result, to give out a reasonable initial problem panel, we can move a random combination of all kind of steps from an answer state.

2.2. Gene

Let’s think about the space block, there are five situations on it, say, going left, right, up, down and don’t move. For this problem, it is reasonable that the same space movement with different number blocks arrangement (because of different steps). So, we set other different operations as single steps. For example, after a left-up-right movement, the space block changes one step, but the result is different from a single up movement. In such consideration, we have 17 different genes which have been defined in program.

Because that we have genes in different length, the lengths of chromosomes are not fixed.

(To ensure a better performance, the combination of those steps cannot be separated.)

2.3. Fitness and Selection

Here, we use the classic selection method Roulette wheel selection. In the roulette wheel selection, the probability of selection of each individual is in proportion to its fitness. By formula, the probability to be selected is:

$$p_i = \frac{f_i}{\Sigma_{j=1}^{N} f_j},$$

Where, f_i is the fitness of individual i in the population, and N is the number of individuals in the population.

2.4. Multi-threads

In order to explore some hidden regular pattern, we use multi-threads in our project. When run the program, user can input a number of threads. Each thread run for the solution and return a number of generation when it get the answer. After running the program, we can use those data for further analysis. For example, the relationship between thread number and time to get a solution.

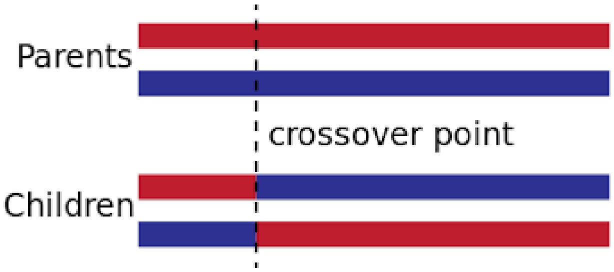
3. Evolution Process

3.1. Initialization and Configuration

As said above, the original state is a panel with a random movement from an ordered arrangement. As the instruction of this project, we set our initial population as 1000. The maximum number of generations is 10000. And as a result of that the length of chromosome is not fixed, we can only set the range of that. The minimum size is one and the maximum size is 25.

3.2. Crossover

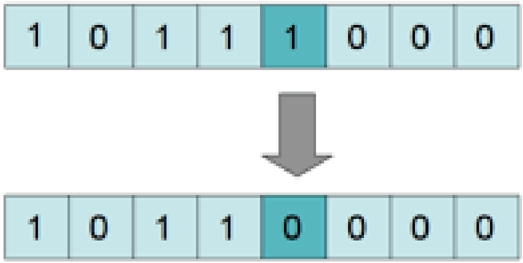
When the population generate off-spring, they will make a cross from a random bit of their chromosome. The part of two chromosome before the random bit will exchange to another chromosome at the same location as the following picture. Because we have lots of genes with different length, the length of chromosomes after cross may change a lot.



We set the crossover rate as 0.4, which means that 40% individuals in this population will make a cross and generate off-spring.

3.3. Mutation

When generating the next generation, there may occur mutation after crossover, which means a gene may be changed. The picture below shows an example.



Here, for the mutation, we allow three different actions: modification, deletion and insertion of a movement from/into a chromosome. It’s worth to be point out that we have a limitation on chromosome size. So, when there is an insertion or a deletion, we have to check that whether the new size is reasonable or not. In our project, we set the mutation ratio as 0.5.

3.4. Fitness Selection

As for roulette wheel selection, probability reflects the proportion of individual i's fitness in the total individual fitness of the entire population. The greater the individual's fitness, the higher the probability of being selected. After the individuals are selected, they can be randomly paired for later crossover operations.

3.5. Termination

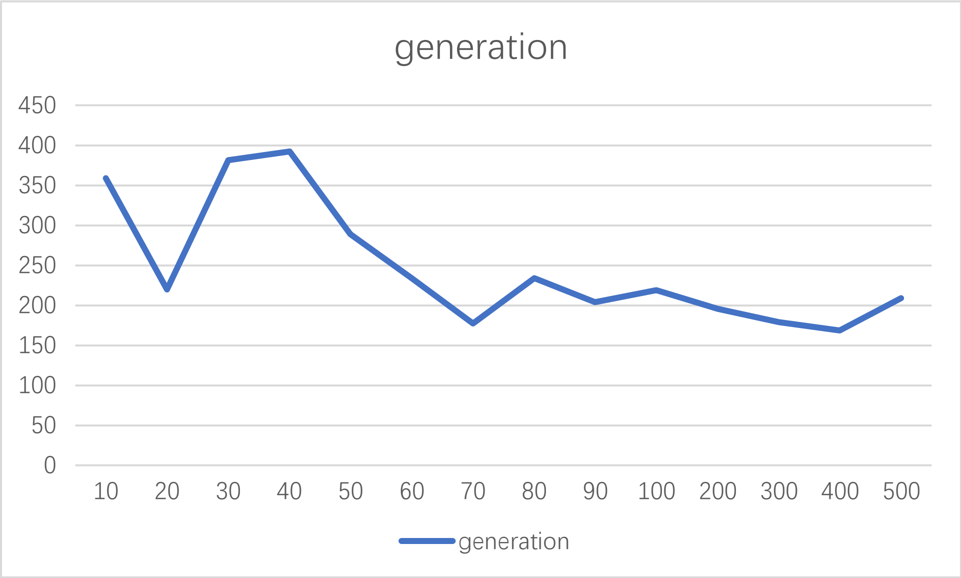
In the project, we calculate the best fitness at every step. If it remains constant for a number of generations, it is considered that the system has reached a local minimum. Here, we set the number of generations as 200. In this case, apply the best chromosome moves to the current board (environment) and create a new population that starts from this new panel (the new environment).

3.6. Remark

In the scenario of sliding block problem, we have different original state for every time. So, there is no general solution or best solution in total. When doing test, we can only talk about a specific situation.

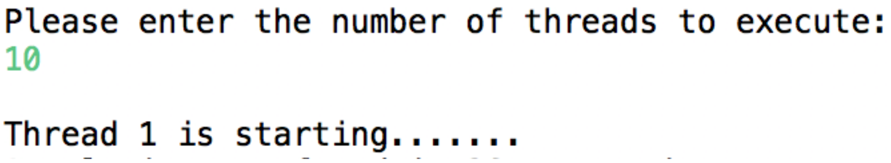
4. Conclusion

The purpose of this analysis is to find the average number of generation when we get a solution. We run the project with different number of threads, and with the improvement of the threads number, we can see that the generation number will become stable at near the 200 generation. The relation and trend can be easily get from the following diagram.



5. Result and evidence of running

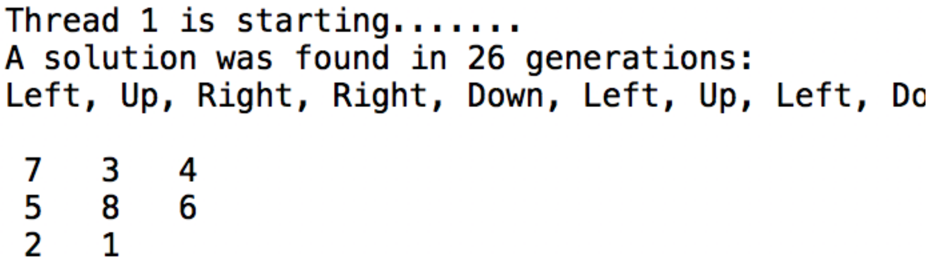
5.1. Running result



```
Please enter the number of threads to execute:
10

Thread 1 is starting.......
```

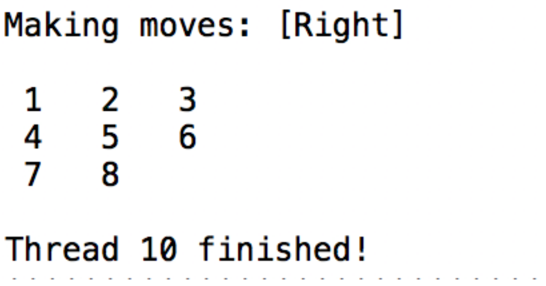
Firstly, we enter a thread number that the program will run.



```
Thread 1 is starting.......
A solution was found in 26 generations:
Left, Up, Right, Right, Down, Left, Up, Left, Do

 7   3   4
 5   8   6
 2   1
```

Here is a screenshot of one run. The first line is when the 210 generation, we get our solution and the following matrix is the original panel of this problem. (Following steps show how we get this state from an ordered matrix)

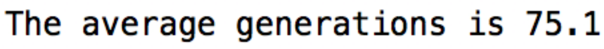


```
Making moves: [Right]

 1   2   3
 4   5   6
 7   8

Thread 10 finished!
```

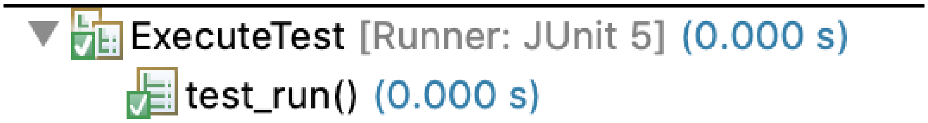
In the final state, we get a matrix in the right order(Followed by the thread number index)



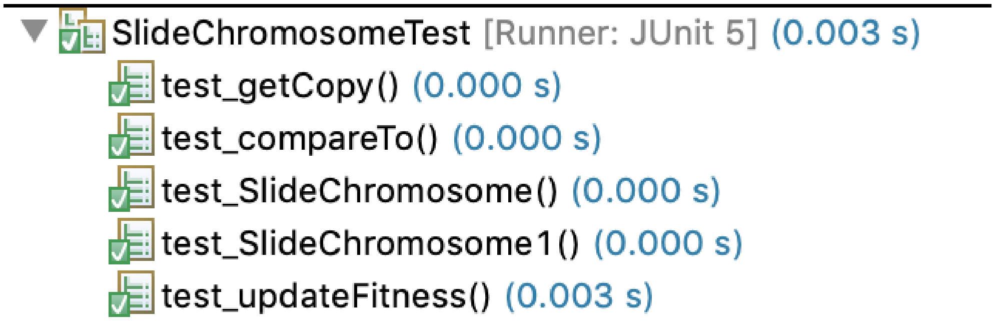
```
The average generations is 75.1
```

By default, the program will show the average generation the program used to get the final solution state.

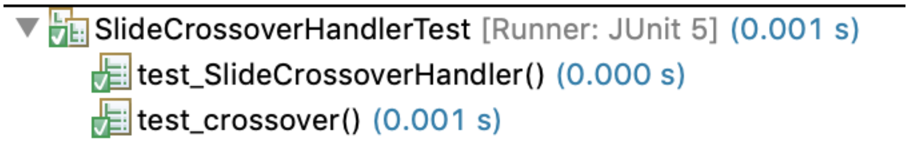
5.2. Evidence of test



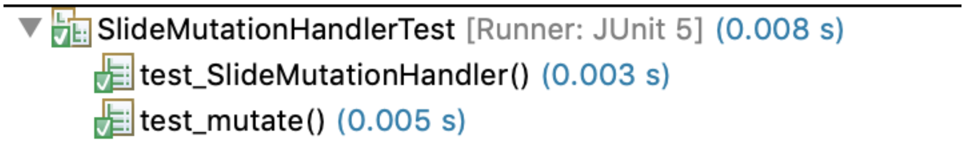
```
▼ ExecuteTest [Runner: JUnit 5] (0.000 s)
      test_run() (0.000 s)
```



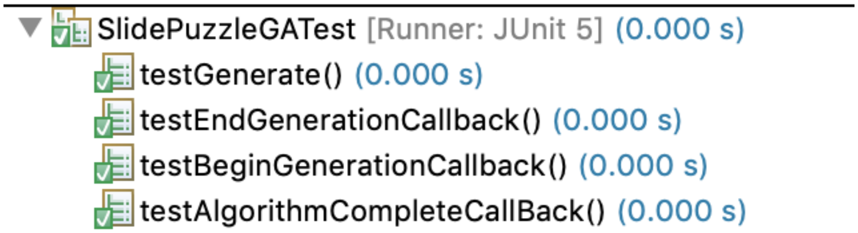
```
▼ SlideChromosomeTest [Runner: JUnit 5] (0.003 s)
      test_getCopy() (0.000 s)
      test_compareTo() (0.000 s)
      test_SlideChromosome() (0.000 s)
      test_SlideChromosome1() (0.000 s)
      test_updateFitness() (0.003 s)
```



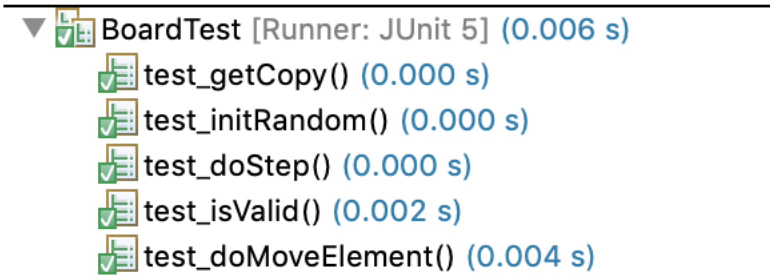
```
▼ SlideCrossoverHandlerTest [Runner: JUnit 5] (0.001 s)
    test_SlideCrossoverHandler() (0.000 s)
    test_crossover() (0.001 s)
```



```
▼ SlideMutationHandlerTest [Runner: JUnit 5] (0.008 s)
    test_SlideMutationHandler() (0.003 s)
    test_mutate() (0.005 s)
```



```
▼ SlidePuzzleGATest [Runner: JUnit 5] (0.000 s)
    testGenerate() (0.000 s)
    testEndGenerationCallback() (0.000 s)
    testBeginGenerationCallback() (0.000 s)
    testAlgorithmCompleteCallBack() (0.000 s)
```



```
▼ BoardTest [Runner: JUnit 5] (0.006 s)
    test_getCopy() (0.000 s)
    test_initRandom() (0.000 s)
    test_doStep() (0.000 s)
    test_isValid() (0.002 s)
    test_doMoveElement() (0.004 s)
```