CS492D Search Based Software Engineering
Coursework: Stochastic Optimisation
Implementation Document

CS, 20155228, Nakwon Lee

October 29, 2015

1 Travelling Salesman Problem

The order of coordinates (permutation) is main concern for Travelling Salesman Problem (TSP). The intuition for solving TSP is that if two parents solutions have different partial optimal permutation, the combination of two partial optimal permutation should be better than each parent solution. The evolution therefore should proceed to retain the order of good solutions.

1.1 Crossover

Crossover operation that I use is focusing on how to keep the order of good parent solution. The "crossover" function takes two solutions as input, which represent parents A and B. Then, randomly choose two different change points. Get permutations btw change points from the parent A and insert that permutation to child B at same position. Do same action for parent B and child A. After that, remaining vacant positions in child A (child B) are filled with coordinates in parent A (parent B) by keeping their order.

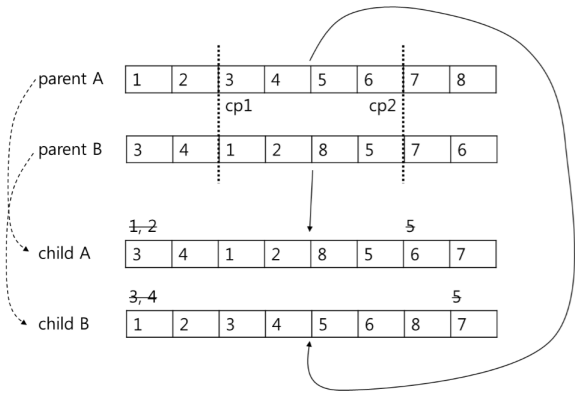

Figure 1: Example of crossover for TSP

For example in Fig. 1, permutation btw change points are add to directly. Remaining coordinates cannot be directly added to a child because there are duplicated numbers in given permutation. After inserting permutation in parent B to child A, vacant positions in child A are filled with coordinates in parent A. To keep the order, fill coordinates in order of parent A, (1,2,3,4,5,6,7,8).

When we putting 1 to child A index 0, it is not possible because there is 1 in the inherited part, which is from parent B. Two cannot also be added to child A because of same reason. Therefore, 3 added to child A. Through this procedure, the order of coordinates in parents are inherited to children.

1.2 Selection

I just use Binary Tournament selection mechanism for selection operator. It randomly select two solutions in population and return more fitted one. For each crossover, two parents are needed. Therefore, after crossover, two parents are selected by two selection operator calls.

1.3 Mutation

The Mutation operator I use just swap randomly selected two coordinates. Mutation rate is 0.5, which means that for every offspring, it is mutated by probability of 0.5. Mutation is not applied to the elite solutions, which is directly added to the next generation.

1.4 Input File Format

The input file, which stores the information of coordinates, must be a plain text format (filename extension is not concern). In file, there is no line that starts with digit except lines representing a coordinate data. The line representing a coordinate data must be structured as [label of coordinate][white space][x-coordinate][white space][y-coordinate]. The label of coordinate must be a digit and cannot be duplicated. There is no blank between a start of line and a label of coordinate.

1.5 Execution Manual

The file *ga_tsp.py* is the executable file for solving TSP.

~$ python ga_tsp.py -[option] [option parameter] [file name]

[option]
p [size of population]: set the population size
f [budget limitation]: set the budget limitation

Options are omissible. Default setting of population size is 200 and budget limit is 1000000. The [file name] must be a last command line parameter.

1.6 Elitism and Ageing

To accelerate the convergence of solution, I use elitism. The main mechanism of elitism is that more fitted solutions have higher probability of survival. Survived solution is also alive in next generation. The survival probability is established using below equation. The x is a rank of solution in population, which is low when fitness is high (start at zero). The p is a population size and, the y is a probability of survival for solution that is ranked as x.

$$y = 0.9 \times ((p - x)/p)^5$$

From the equation, we intuitively know that the most fitted solution have the highest survival probability and, when fitness is decreasing, survival probability is much more decreased until it become zero.

The ageing is complement of elitism. Although one solution have highest fitness, it does not mean that the highest solution is an/the optimal solution. Therefore, it is needed to ensure diversity of solutions. To promote diverging, I kill solutions in population that live many generations, which is supposed as local optimum. The probability of survival with regard to ageing is based on below equation.

$$y = \sqrt{\sqrt{0.8 - (x \times 0.2)}}$$

The x is a age (how many generations that the solution have been alive) of solution and, the y is the probability of survival for the solution that have age as x. This equation intuitively shows that young solutions are rarely dead but old solutions easily or must be dead.
In conclusion, the solution that is more fitted and young are easily alive at next generation.

1.7 Keep Best Solution of All Time

If the best solution of current generation has higher fitness than best solution of all time except current generation, the new best one is held and prior one is thrown out. Therefore, we can always keep the best solution of all time and it is returned at end of genetic algorithm.

1.8 Check the Details in Code

In sourcecode, I comment many details. So, please check the details in comments

2 Sudoku

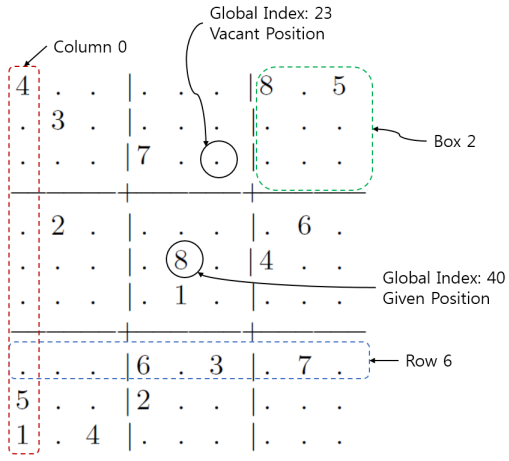

Figure 2: Terms used in this sudoku solving genetic algorithm manual

The sudoku problem is to fill the vacant positions with regard to the given positions following the sudoku rule. To solve the sudoku problem, I use fitness value (FV) as the number of non-assigned position. Non-assigned means that there is no possible number to fill a vacant position because all numbers (1∼9) are assigned to same column, row, or box. The solution of sudoku is therefore the zero FV solution, which means no non-assigned position. However, if the solution has low but non-zero FV, it is also not a correct solution for given positions as same as a high FV solution. I consider this problem with an assumption that, among solutions that have same FV,

a solution which has rarely appeared numbers as assigned numbers for each vacant position is better solution than others. Intuitively, if one number is appeared many times at same position without solving, that number might not be a part of a/the correct answer. It means that, despite of many assignments (exploration) with the number, the solution cannot be found. Therefore, it is reasonable to prefer a solution that have newly appeared numbers for each position.

2.1 Problem Format

A sudoku problem file must be a plain text format. The sudoku problem information are represented by digits and periods. A digit is the assigned number of a given position and, a period is a vacant position. Global indices are just the appearing order of digit or period. The file reading proceeds from byte to byte. Symbols except digits or period are ignored.

2.2 Solution and Fitness

A solution (individual gene of genetic algorithm) consist of list for vacant positions and their assigned number, and fitness value. The positions that does not have a given number are vacant positions. Because given positions differ from problem to problem, the vacant positions vary btw problems. Assigned numbers can also be various for the same vacant positions when the assigning order is vary. Both fixed and changeable order are possible. I implemented both and select changeable order because of empirical evaluation result.
FV is divided into two parts. The *fitness* is just the number of non-assigned position and, the *fitness2* is sum of frequencies (see details in section 2.2.1) of numbers assigned. The *fitness* has higher priority than *fitness2*. The solution, which has zero *fitness* is a/the correct answer.

2.2.1 Frequency

We define frequency to measure the freshness of solution. I consider two types of frequencies, first, how many times the numbers appeared in each vacant position and second, how many times the combination of numbers appeared in population.
Fig. 3 shows the example for calculating first type of frequency. The frequency is stored in table of index by assigned number. If one solution has assigned number 3 at global index 2, it is counted and, the value of position (2,3) in frequency table increased. In example I simply increase the value as one but, actually, the increasing value is relative to *fitness*. The assumption in the *fitness* relative increasing is that low but not zero *fitness* solutions are likely to be a local optimum. Therefore, I increase the frequency by the amount that is in inverse proportion to the *fitness*.
The second frequency called as combinatorial frequency because it consider the combination of assigned numbers and their frequency. It designed to reduce the search space. If we have another fixed valuations for some vacant positions, the algorithm can significantly reduce the search space. Therefore, we can assume some positions as fixed and search under them. Although many generations were done, if the correct answer cannot be calculated, we can change the fixed part as another combination. This change can be done by combinatorial frequency, which represent how many one combination is explored. However, combinatorial frequency is not only applicable to non-changeable order, but also worse than other setting (ex. changeable order with only *fitness2*). So, I select changeable order.

2.3 Crossover

The solutions have non-assigned vacant positions. If we fill non-assigned positions, problem is directly solved. From this intuition, it is reasonable to inherit the non-assigned positions from another good solution.

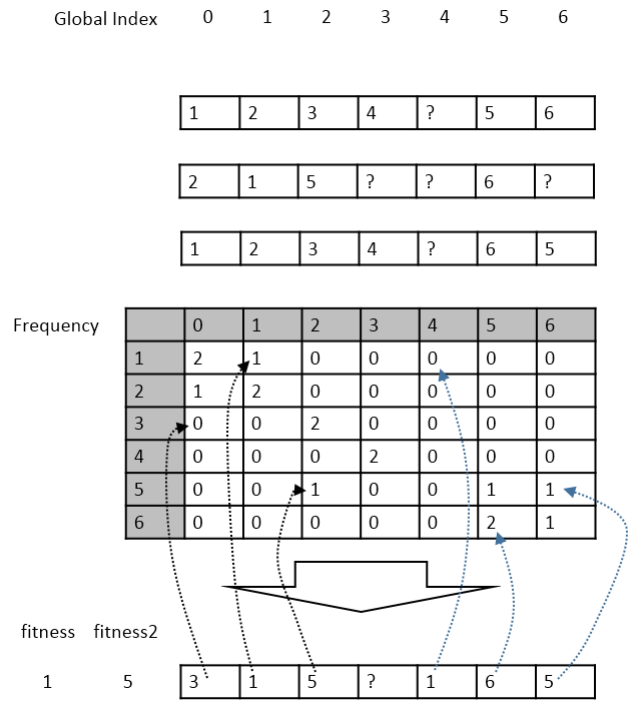

Figure 3: Example for calculating frequency

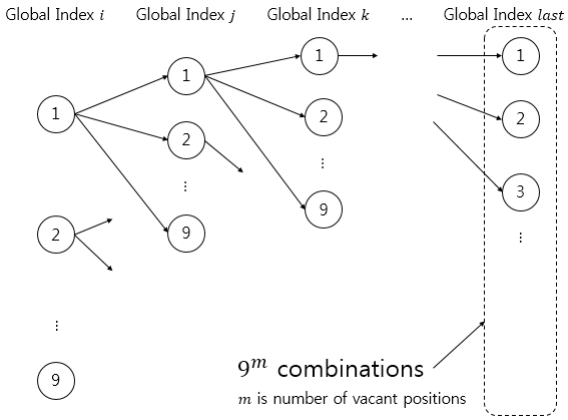

Figure 4: Example for calculating combinatorial frequency

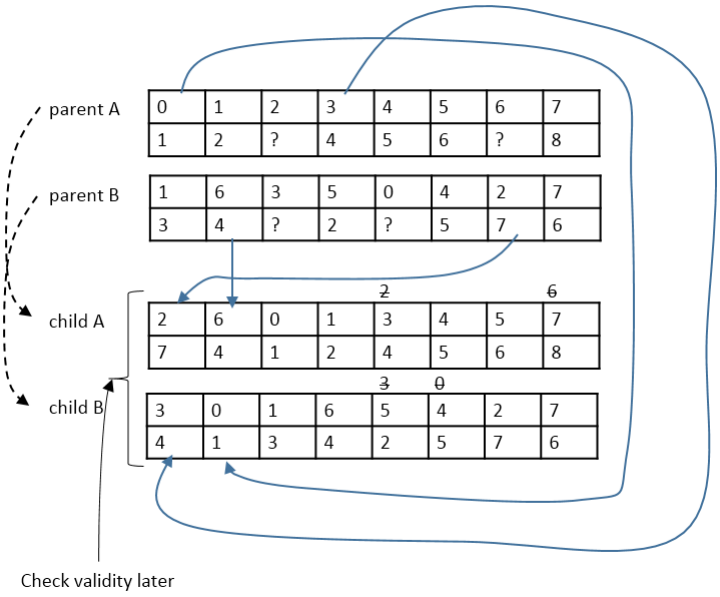

Figure 5: Example for crossover of sudoku problem

In Fig. 5, parent A has 2 and 6 as non-assigned positions. When making offspring, the non-assigned positions are inherited from the other parent, parent B. And, inherited non-assigned positions are inserted to the front of a child to retain assignment during the validation phase (see details in Section 2.6). The remaining parts of child A (child B) are inherited from the parent A (parent B).

2.4 Mutation

The mutation operator for sudoku problem just swap two vacant positions and their assignments in given solution.

2.5 Selection

It return the more fitted solution btw two randomly selected solutions from the population. In selection operation, we have to consider two disjunctive FVs. As I mentioned before, *fitness* is more important than *fitness2*. Therefore, return solution that has smaller *fitness*. If two solutions have same *fitness*, return solution that has smaller *fitness2*.

2.6 Generate Random Solution and Validate Offspring

To generate initial population, I randomly generate solutions. The finding candidate number for given global index is simple. I use set operator for it. Given positions and their numbers are already stored in relevant sets during the file reading. For example, if global index 40 is given and, the number is 8, number 8 is stored in the row set 4, column set 4, and box set 4. Indices of sets are calculated as (row set index) = quota of global index / 9, (column set index) = remainder of global index / 9, and (box set index) = calcBoxIdx(global index), which is implemented by me.
As I mentioned before, we use changeable order. Therefore, solutions can have different order of

vacant positions. For each random generation, first shuffle the order of vacant positions. After that, find the possible set of candidate numbers for global index using difference set operation following the shuffled order. Detailed formula can be seen in the source code. If the set of possible candidates are empty, it is the non-assigned position. If not, we choose and assign one number from candidate set. The chosen number is added to appropriate row, column, and box sets. And proceeds to the next vacant position.
Validation is for offspring. In contrast to random generation, the offspring already has assigned numbers for vacant positions. Therefore, we use the assigned numbers as so as possible. The basic procedure is same as random generation but, if the assigned number is in candidate set, we choose assigned number instead of other candidates. The other way, although there is assigned number, which is not zero, the candidate set can be empty. If so, just assign zero to the vacant position.
Both in random generation and validation, the evaluation counter is increased because they are the actual evaluation of solutions. Therefore, the calculation of *fitness* and *fitness2* are also conducted. The *fitness* is increased as one if the candidate set is empty. The *fitness2* is repeatedly increased as the value of assigned number and its global index in current frequency table. The update of frequency table is conducted at the same time with *fitness* relative manner (not to the current table, to the copy of table, at the end of generation, change the table to newer one).

2.6.1 Recalculation of Fitness

I also use elitism. Therefore, there are solutions that is not a random generated one or offspring. The elite solutions does not change any assignment and the *fitness* but, the *fitness2* can be changed. When the solution is selected as elite, it is sent to the re-calculation function and update *fitness2* and increase the frequency table. Evaluation counter also increased in here.

2.7 Elitism and Ageing

To accelerate the convergence of solution, I use elitism. The main mechanism of elitism is that more fitted solutions have higher probability of survival. Survived solution is also alive in next generation. The survival probability is established using below equation. The x is a rank of solution in population, which is low when fitness is high (start at zero). The p is a population size and, the y is a probability of survival for solution that is ranked as x.

$$y = 0.8 \times ((p - x)/p)^5$$

From the equation, we intuitively know that the most fitted solution have the highest survival probability and, when fitness is decreasing, survival probability is much more decreased until it become zero.
The ageing is complement of elitism. Although one solution have highest fitness, it does not mean that the highest solution is an/the optimal solution. Therefore, it is needed to ensure diversity of solutions. To promote diverging, I kill solutions in population that live many generations, which is supposed as local optimum. The probability of survival with regard to ageing is based on below equation.

$$y = \sqrt{\sqrt{0.8 - (x \times 0.2)}}$$

The x is a age (how many generations that the solution have been alive) of solution and, the y is the probability of survival for the solution that have age as x. This equation intuitively shows that young solutions are rarely dead but old solutions easily or must be dead.
In conclusion, the solution that is more fitted and young are easily alive at next generation.

2.8 Execution Manual

The file *ga_sudoku.py* is the executable file for solving sudoku.

~$ python ga_sudoku.py -[option] [option parameter] [file name]

[option]
p [size of population]: set the population size
f [budget limitation]: set the budget limitation

Options are omissible. Default setting of population size is 200 and budget limit is 1000000. The [file name] must be a last command line parameter.

2.9 Check the Details in Code

In sourcecode, I comment many details. So, please check the details in comments