CS492D
Search Based Software Engineering

Coursework: Stochastic Engineering

Due: 13:00, 29 October 2015
LATE SUBMISSION

20130551 Youngjae Chang
youngjae.chang@kaist.ac.kr

Each problem is approached with 5 steps:

1. Initialization — **generate** function in **ga.py**
2. Crossover operator — **Crossover** class in **ga.py**
3. Mutation operator — **Mutation** class in **ga.py**
4. Other optimization techniques — **ga** function in **ga.py**
5. Tuning parameters — running arguments

The explanation also follows the same order.

(* Note that all the possible options can be found by adding -h when executing)

1. Traveling Salesman Problem

1.1 Initialization

Assumption:

The distance from one city to all other cities will form a normal distribution.

The shortest path can only be achieved by visiting near cities together.

So, when initialize, I only allow cities with distance less than average subtracted by standard deviation. To do this, I append cities to path one by one, checking the constraints. Only if there's no option, random city have been appended. Thus extreme cases are pruned from exploration.

1.2 Crossover Operator

The standard pmx crossover operator is used for the problem. Nothing much changed from the given implementation.

1.3 Mutation Operator

In case of mutation, I've applied best 2-opt algorithm, a variation of 2-opt algorithm. The basic role of algorithm is same, if there's a intersection, the algorithm will unwind it. The difference of best 2-opt algorithm is that it just swap the two point rather than reversing the intermediate path. This reduce the computation cost greatly. The mutation is only applied when the results improve, giving sense of exploitation. The algorithm can be applied multiple times; it can be adjusted using -2 (—repeat_2opt) option. The default is 3.

The mutation is always applied.

1.4 Other optimization techniques

Restart algorithm has been applied. If there's no improvement for certain number of generation — it can be specified using -r (—restart_after) option, default is not restarting — solve the problem again with completely new population.

Elitism is also applied, we can cut-off certain percentage of population and use the only the good ones for crossover. Use -e (—elitism) option to set the percentage of population you want to use for crossover. The default is 1, not using elitism.

1.5 Tuning parameters

Since the mutation operator have high computation costs, increasing -2 option value should be considered carefully. if the number of city goes over 100, using the value over 2 made computation unfeasible, it use too much time on exploitation. The population size does not have a big effect on the results; 100~300 was enough. Though the use of elitism is highly suggested. -r option was adjusted with trials.

2. Sudoku Solver

1.1 Initialization

Sudoku solution can be represented with various notations, here I represented it as a list of values that fills up the sudoku. This approach reduce the search space dramatically, and also guarantee the pre-defined values to stay there with no-extra computation costs.

When the sudoku puzzle is parsed, first the naked-single and hidden-single algorithms are applied to fill up the holes deterministically. Then now the board is ready for genetic magic.

The initial population is earned by generate function which call solve function in sudoku.py. It randomly choose the value among the candidates that does not break the rule. The algorithm doesn't care about values chosen before, it only depends on board configuration.

The problem definition is widely covered in **sudoku.py** file, so please have a look at it.

1.2 Crossover Operator

The simple cross operator is used.

1.3 Mutation Operator

Resampling two points is used as a mutation operator.

1.4 Other optimization techniques

Elitism and Restarting.

Occasional Hall-of-fame.

1.5 Tuning parameters

Omitted.