EGE UNIVERSITY

Knapsack Problem

With Genetic Algorithm

Ayham Jaradat

91150000623

This report explains how I managed to solve Knapsack Problem using Genetic Algorithm, with implementation details about the project.

10/12/2016

Contents

Introducing the project 3

Knapsack Problem 3

Genetic Algorithm Components 3

Genetic Algorithm outline 3

Population 4

Chromosomes 4

Fitness function 4

Selection 4

Crossover 4

Mutation 5

Termination Conditions 5

Flow chart of the program 6

Using the Project 7

Inside the project 7

Running and Testing 9

Comparisons 10

Conclusion 11

Last word (Resources) 11

Automated Parameter Tuning 12

References 13

Knapsack Problem with Genetic Algorithm

Introducing the project

This report provides a description for a project that solves 0-1 Knapsack Problem using Genetic Algorithm, The Project is built as a web application that gives the user the ability to test GA for knapsack problem with different input data and GA parameters. The project uses *Javascript* as its main programming language.

The project is a front-end web application that works in any browser. Mainly build with *Javascript* and its main library *JQuery*. The project is built from scratch without using any GA library. The code is clean, well documented and forward, The project provides the user with many options to change for testing knapsack problem. The user can upload his own input data as text file. I worked on the project for three weeks, with minimum 2 hours per day including learning GA, developing GUI, implementing the algorithm and testing on different data and parameters.

Knapsack Problem

The Knapsack Problem is an example of a combinatorial optimization problem, which seeks to maximize the benefit of objects in a knapsack without exceeding its capacity. The Knapsack problem is a NP problem. In this project I implemented an optimization solution for KP using GA, The Algorithm seeks for finding the best solution from among many other solutions. The Problem is concerned with a knapsack that has positive integer capacity, and there are n distinct items that may potentially be placed in the knapsack. every item has a positive integer weight and positive integer benefit. Generally ,if for every item there is infinite number of available copies that can be placed in the knapsack, it is called *unbounded KP*. And if for every item there is a finite number of available copies that can be placed in the knapsack it is called *bounded KP*. The bounded KP when there is only one available copy of every item, is called *0-1 KP*. In this project I only focused on *bounded 0-1KP* where for every item there is only one copy available to be placed in the knapsack problem.

Genetic Algorithm Components

The Genetic Algorithm consists of some important components that can be modified and customize based on the problem nature that it solves. I provide here the components with explanation for the custom choices I made to solve the Knapsack problem.

Genetic Algorithm outline

1. Start: Randomly generate a population of N chromosomes.
2. Fitness: Calculate the fitness of all chromosomes.
3. Create a new population:
    a. Selection: According to the selection method select 2 chromosomes from the population.
    b. Crossover: Perform crossover on the 2 chromosomes selected.
    c. Mutation: Perform mutation on the chromosomes obtained.
4. Replace: Replace the current population with the new population.

5. Test: Test whether the end condition is satisfied. If so, stop. If not, return the best solution in current population and go to Step 2.

Each iteration of this process is called generation.

Population

Population is the set of candidate solutions in every generation. we start with a random population that evolve toward better solution in every generation. The population size is fixed in every generation. This project provides the user with the ability to set the generation size.

Chromosomes

The chromosomes in GAs represent the space of candidate solutions. For the Knapsack problem, I use binary encoding, where every chromosome is a string of bits 0 or 1. A chromosome is represented in an array having size equal to the number of the items. Each element from this array denotes whether an item is included in the knapsack (‘1’) or not (‘0’).

Fitness function

GAs require a fitness function which allocates a score to each chromosome in the current population. Thus, it can calculate how well the solutions are coded and how well they solve the problem. I calculate the fitness of each chromosome by summing up the benefits of the items that are included in the knapsack, while making sure that the capacity of the knapsack is not exceeded. If the volume of the chromosome is greater than the capacity of the knapsack then randomly one of the bits in the chromosome whose value is ‘1’ is inverted and the chromosome is checked again. For better performance I square the obtained value of the summation to be the final score value, by doing this I give a bigger chance for better chromosomes when doing the random selection of parents.

Selection

The selection process is based on fitness. Chromosomes that are evaluated with higher values (fitter) will most likely be selected to reproduce. This phase has an element of randomness just like the survival of organisms in nature. The most used selection methods, are roulette-wheel, rank selection, steady-state selection, and some others. Moreover, to increase the performance of GAs, the selection methods are enhanced by elitism. Elitism is a method, which first copies a few of the top scored chromosomes to the new population and then continues generating the rest of the population. Thus, it prevents loosing the few best found solutions.

In this project I implement selection using roulette-wheel, which is a simple weighted random selection method that implements fitness-proportionate selection. I gave the user the ability wither to use elitism or not in the selection process. When using the elitism we keep the first two best found solutions in the new population.

Crossover

Crossover is the process of combining the bits of one chromosome with those of another.

This is to create an offspring for the next generation that inherits traits of both parents. There are many crossover techniques exist. In this project I chose three different techniques which are :

- Random single-point crossover : for two parents we choose randomly a crossover point, and the child takes all bits before this point from one of the parents and the other bits from the other parent.
- Middle point crossover : For two parents we choose the middle point to be the crossover point. the child takes the first half of its bits from one of the parents and the second half of its bits from the other parent.
- Uniform crossover : The uniform crossover evaluates each bit in the parent strings for exchange with a probability of 0.5, the offspring has approximately half of the genes from first parent and the other half from second parent with 0.5 mixing ratio.

The project gives the user the ability to choose between these three crossover techniques in run time.

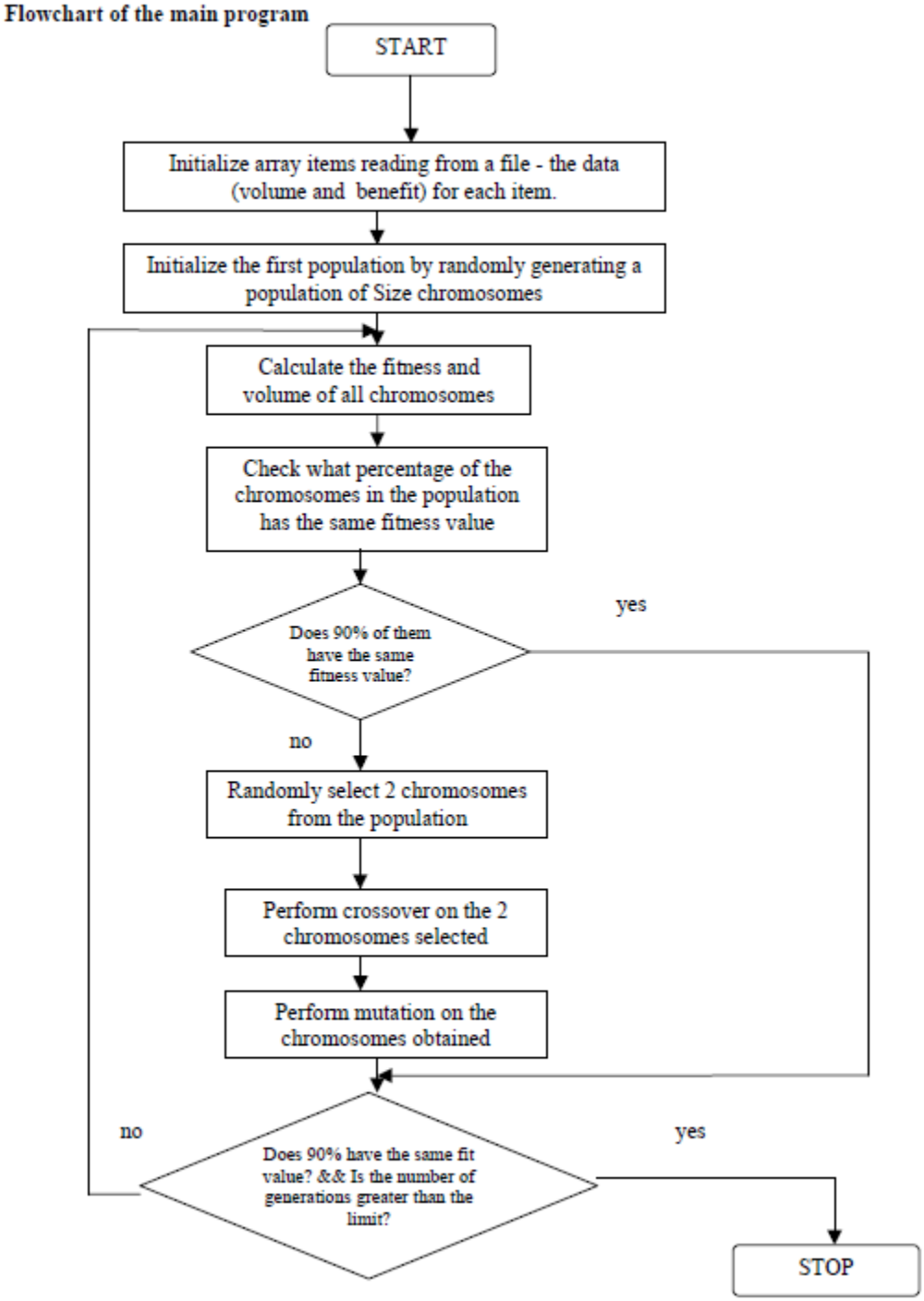
Mutation

Mutation is performed after crossover to prevent falling all solutions in the population into a local optimum of solved problem. Mutation changes the new offspring by flipping bits from 1 to 0 or from 0 to 1. Mutation can occur at each bit position in the string with some probability, usually very small (e.g. 0.001). This project provides the user with the ability to change the mutation ratio at run time.

Termination Conditions

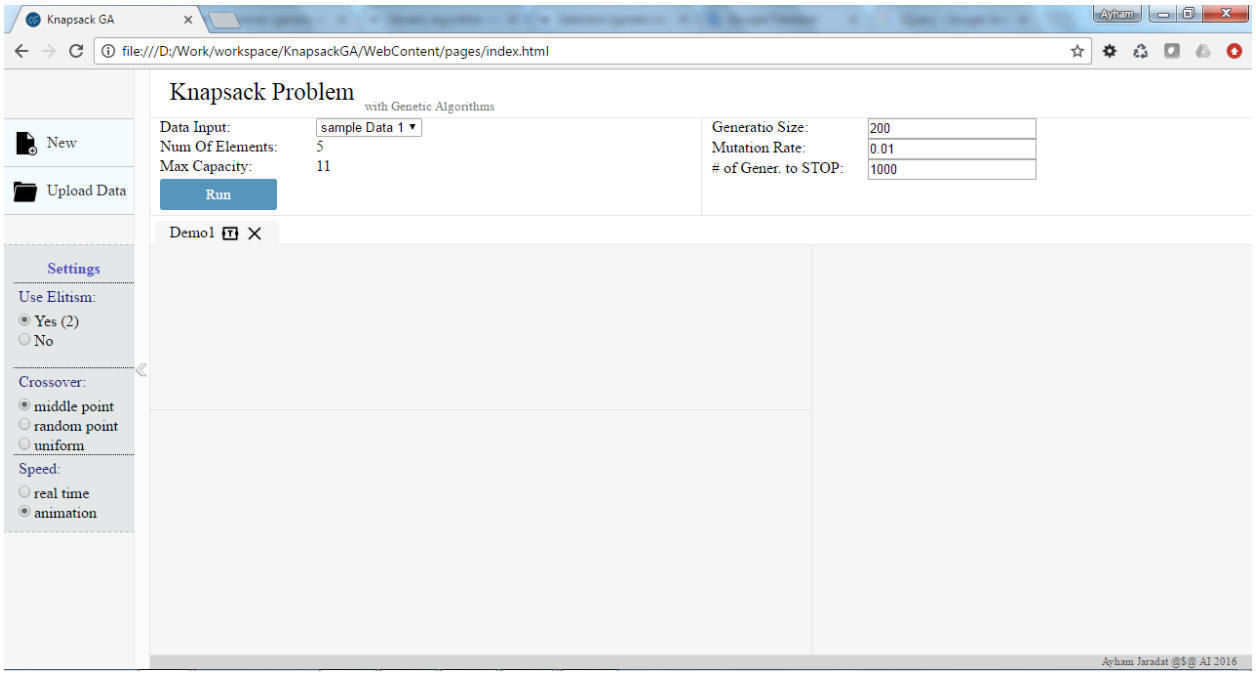
The population converges when either 90% of the chromosomes in the population have the same fitness value or the number of generations is greater than a fixed number. This project gives the user the ability to set the maximum number of generations at run time.

Flow chart of the program



Using the Project

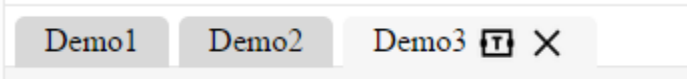
Using the project is as easy as opining a web page in any browser. Since the project is not yet up on the internet, to open it just double click on the index.html page provided with the source code. The web page should be opened in your default browser. and you will be seeing something like the next figure.



Inside the project

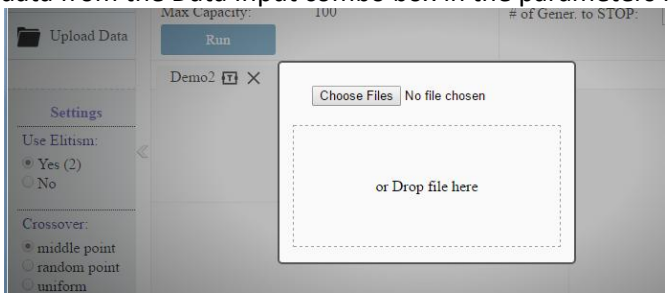
the project provides many features as follow:

- Running many instances of the problem at the same time, this is useful to compare with different parameters or different input data in parallel. I call every run instance a demo. To open new demo hit the New button on the left panel.

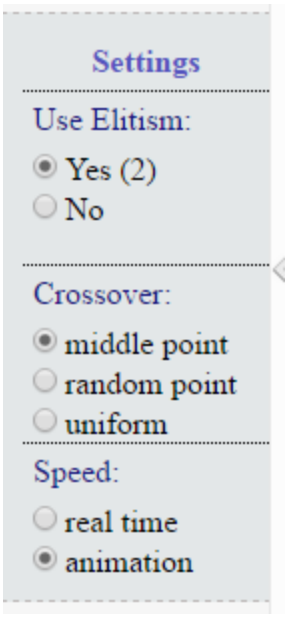


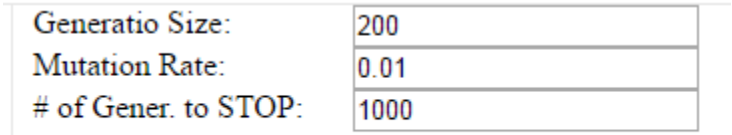
- Upload Input Data, The user can upload any input data of the knapsack problem using text file in the following format:
  first line has the maximum capacity of the knapsack. and each one of the other lines has two values separated by tab, the first value represents the weight of the item and the second value represents the benefit of the item.
  I provided some data input files with the source code as well as two sample data build-in with the code. To upload a file just click on Upload Data button. Then you can select the uploaded

data from the Data Input combo box in the parameters menu.

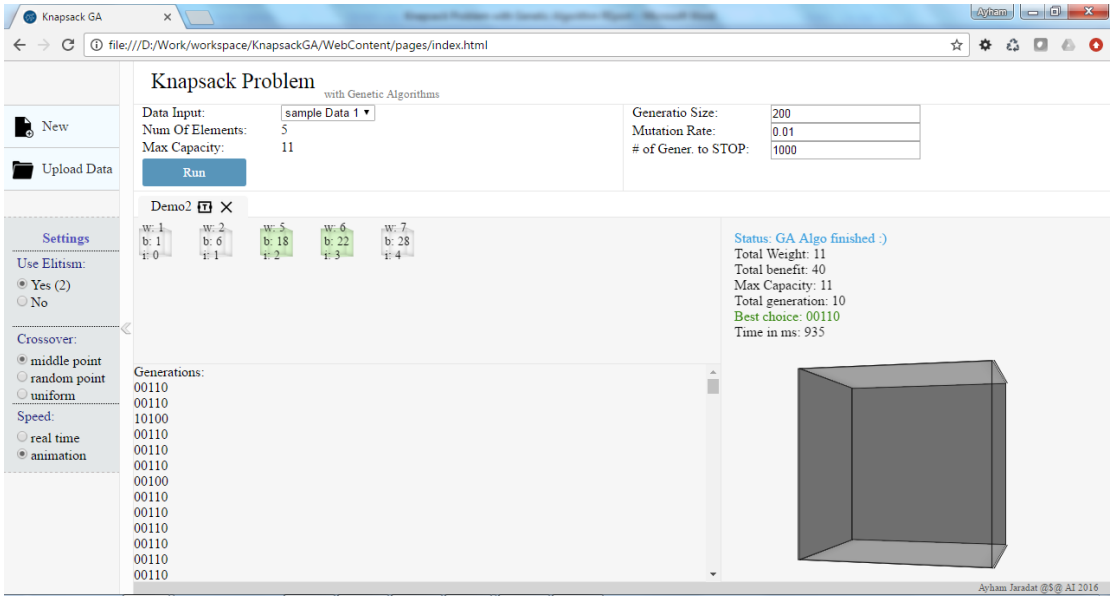


- Setting some GA Parameters: the user can set the Population Size in each generation, The Mutation Ratio, The maximum number of generation to stop, Use Elitism or not and Crossover technique.





- After selecting Input Data and parameters , it is time to hit the Run button to see the algorithm running as shown in the next figure.



As shown in the figure, under Demo tab, user can see the information about items to choose from (weight, benefit and index) represented as small boxes, All the chromosomes representation of current population, And the knapsack with some information about current generation like total weight and benefit of items in the knapsack, number of created generation for now, best gene representation and running time. Also the knapsack is represented as a big box under status panel showing the total weight in it.
In every generation the graphics are updated to indicate current obtained solution. User can choose animation speed from left panel to slow down the operation for better visualization.

Running and Testing

Our generic algorithm for solving Knapsack problem depends on many components and parameters that can be changed and test again to find the best configuration that will give us the best solution in a timely manner. Examples of these parameters are population size, mutation ratio, crossover technique, use elitism or not and input data size. However it is difficult to test and compare all of these inputs with their obtained results in this report. So I will focus on two parameters and set the others as fixed for the next experiments.
In my experiments I use the following parameters as fixed :

- Mutation Ratio of 0.01.
- Maximum number of Generation as 1000.
- Use Elitism.
- The same input data

I will run the code changing the following parameters :

- crossover technique (random crossing point, uniform crossover)
- population size (50,200).

The used input data is provided with the code as a text file named data3.(1) Which is a set of 24 different items to be placed in a knapsack with maximum capacity of 6404180. The optimal solution for

this data has a total benefit of 13549094, obtained with the next optimal selection string (110111000110100100000111).

The following Table summarize the obtained results in the four experiments:

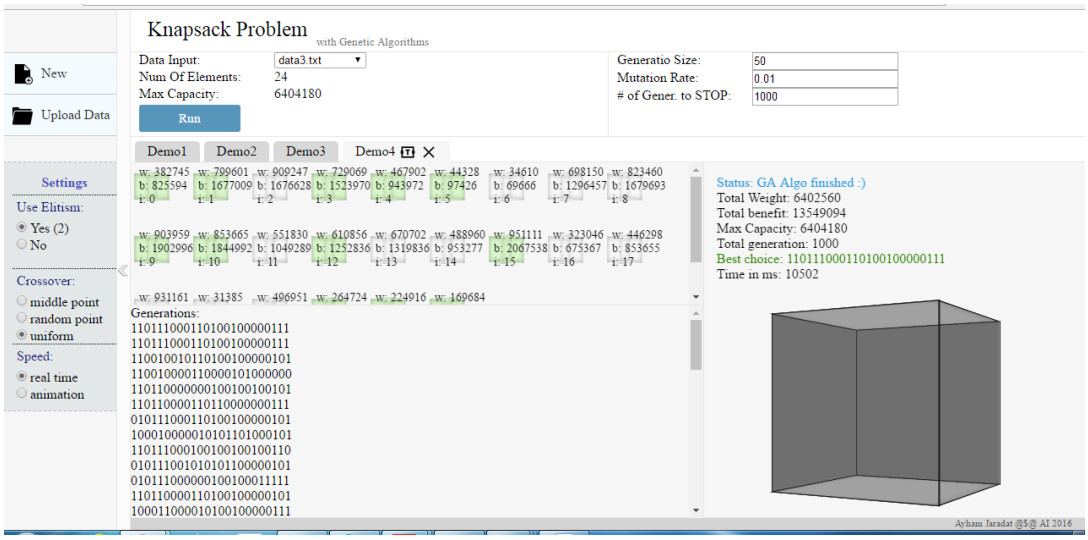
Experiment #	Crossover Technique	Used Population Size	Total Benefit of Solution	Time in MS	Chosen Solution as a string (Best Gene)
1	random point	50	13509264	12747	110110001110000100000110
2	random point	200	13518963	21080	110101101110100100000100
3	uniform	50	13549094	10502	110111000110100100000111
4	uniform	200	13549094	21036	110111000110100100000111

Comparisons

It is important to notice that all the experiments finished exactly after 1000 generation, which means our second termination condition did not achieved on any of these experiments, because the number of the elements in the input data is relatively big. On other experiments with 5 elements the algorithm usually finish before 1000 generation because it reached 90% of chromosomes with the same fitness.

Let us first Compare the population size, As seen in the previous table in the first two experiments when we run the code on random point crossover with different population size value (50 and 200), we get a better solution when we increased the population size in every generation. This means increasing the population size could help achieving better solution, but as noticed in the table the required time to finish 1000 generation increased significantly. This means there is a trade-off when selecting a suitable population size.
However , if we look at the last two rows in the table when we changed the population size with setting uniform crossover we could obtain the same answer using 50 and 200 as population size, this means if we chose better crossover technique we might be able to reduce the population size which will reduce the required time while still giving the best solution.
To compare the cross over technique let us compare them on the same size of population (Experiment 1- 3, and experiment 2-4). It is clear that the Uniform Crossover is doing better than the random single point crossing. It is also noticeable that using Uniform Crossover gave us the optimal solution (not just a better solution) when we run it on 50 and 200 population size.

The next figure shows the result of experiment number 3.



Conclusion

In This report , I show that the uniform crossover technique is better than random single-point crossover for knapsack problem. I also cleared that the population size is important parameters that could enhance the obtained solution by increasing it, but there is a trade-off with the operating time. However, I did some more testing using this project and noticed the following :

- Using Elitism is highly recommended, since it ensures to keep the best found solution during the generations evolving. And it does not affect the operating time.
- Mutation Ratio should be small as possible but not zero, to ensure not to stuck in a local maximum. Increasing mutation ratio will increase the randomness and might not help enhancing the generation towards a better solution.
- If the number of items is relatively small, there is no need to wait 1000 generation because the genes will converge to the better solution faster.

Last word (Resources)

While building this project I used these resources as follows :

- A Youtube video series about Genetic Algorithms, by Daniel Shiffman. I watched the series to learn everything about genetic Algorithms. (link is provided as reference 1).
- A paper titled "Solving the 0-1 Knapsack Problem with Genetic Algorithms" by Maya Hristakeva and Diptp Shrestha. I used the main over-flow of the program as they use with changes for used techniques. (link is provided as reference 2).
- A Youtube video for Weighted Random Algorithm, by Keith Perters. I used the technique he used when selecting Parents with probability. (link is provided as reference 3).
- A 0-1 Knapsack Problem Data set. I used the internet to collect some known data set for this problem. (link is provided as reference 4).

I know these resources could not be considered as strong references but they worth mention.

Automated Parameter Tuning

Automated Parameter Tuning is an optimization problem with the objective of finding good static parameter settings before the execution of an algorithm. Many algorithms have free parameters that can be modified to improve performance in practice, examples of such algorithm are :

- Genetic algorithms
- Bayesian optimization
- Evolutionary strategies
- Local search
- Racing strategies

These algorithms have parameters that affect their performance , such as numerical thresholds or discrete algorithm flags. The setting of these parameters often has a drastic impact, and thus algorithm designers optimize their parameters in practice, often manually which is a tedious, failure prone and time-consuming task. The Automated Parameter Tuning provides an automated approaches for this algorithm configuration problem (5). Examples of such approaches are :

- Model-Free
    - Revac.
    - ParamILS.
    - Gender based Genetic Algorithm
- Model-Based
    - DoE.
    - EGO.
    - F-Race.
    - Calibra.
    - SKO.
    - SPO.
    - SMAC.

A clear example of Automated Parameter Tuning is our Genetic Algorithm solving the knapsack problem, where we have many parameters that need to be set to give us the best solution. such parameters are Population Size, Mutation Ratio and Elitism number. Instead of figuring out manually these parameters we can use one of the Automated Parameter Tuning techniques to find out the appropriate parameters for us.

Many papers for Automated Parameter Tuning and Algorithm configuration are available , see reference (6).

References

For KP with GA:

(1) https://www.youtube.com/watch?v=9zfeTw-uFCw
(2) http://www.micsymposium.org/mics_2004/Hristake.pdf
(3) https://www.youtube.com/watch?v=MGTQWV1VfWk
(4) http://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html

For Automated Parameter Tuning:

(5) http://www.mff.cuni.cz/veda/konference/wds/proc/pdf10/WDS10_109_i1_Dobslaw.pdf
(6) http://www.cs.ubc.ca/~hutter/freiburg-seminar/papers.html