Designing New Algorithms Using Genetic Programming

Jinhwa Kim

School of Business, Sogang University
1, Shinsoo-Dong, Mapo-Ku, Seoul, 121-742, South Korea
Tel: +82-2-705-8860, Fax: +82-2-705-8519, E-mail: jinhwakim@sogang.ac.kr

Abstract

This study suggests a general paradigm enhancing genetic mutability. Mutability among heterogeneous members in a genetic population has been a major problem in application of genetic programming to diverse business problems. This suggested paradigm is implemented to developing new methods from existing methods. Within the evolutionary approach taken to designing new methods, a general representation scheme of the genetic programming framework, called a kernel, is introduced. The kernel is derived from the literature of algorithms and heuristics for combinatorial optimization problems. The commonality and differences among these methods have been identified and again combined by following the genetic inheritance merging them. The kernel was tested for selected methods in combinatorial optimization. It not only duplicates the methods in the literature, it also confirms that each of the possible solutions from the genetic mutation is in a valid form, a running program. This evolutionary method suggests diverse hybrid methods in the form of complete programs through evolutionary processes. It finally summarizes its findings from genetic simulation with insight.

Keywords:

Genetic Simulation; Generalizing Paradigm; Kernel; Combinatorial Optimization Methods; New Methods Development

1. Introduction

This study introduces the design and implementation of an evolutionary approach to designing new methods. It first analyzes the methods in the literature of combinatorial optimization. The main methods, along with their hybrid forms and deviations, are introduced. The methods in the literature are classified into several groups by their natures, such as the behavior of the search. The common and unique features of each method are studied. Sets of rules and steps to combine all these methods are considered. A unifying method combining all these methods is suggested by following the rules and steps we suggest in this study. The approaches to developing new methods are explained together with a classification of them by the nature of their approaches. The theory of genetic programming is the main methodology in this study. The diverse representation schemes of the evolutionary approach are explained. The application areas of genetic algorithms and genetic programming are also briefly introduced. A problem instance to evaluate the methodology is defined in relation to the kernel. The kernel is described in pseudo code form to indicate how it is implemented as a computer program. Function sets, the five basic building blocks of the kernel, are explained with their parameter values. This study also gives a general description of these functions and how they work inside the genetic frameworks.

2. Efforts in Designing New Methods in the Literature

There have been many approaches taken in designing new methods in combinatorial optimization since the 1960's. These approaches can be classified into three directions: inventing completely new methods, adding new features into the existing methods, and combining more than one method together.

The first group of approaches above can be divided into four categories (Morton and Pentico 1993): 1) manual approaches, 2) computer simulation approaches, 3) mathematical approaches, and 4) heuristics. Manual approaches depend on human problem solving ability. Simulation approaches have been studied extensively only after the advent of computers. Mathematical approaches are gaining in effectiveness with the continued development of computing hardware. Branch & bound, dynamic programming, and integer programming, are in this group. The heuristic group includes neighborhood search, simulated annealing, tabu search, genetic algorithms, and neural networks.

The second group, adding new features into existing methods consists of methods to improve the performance of the methods in the first group. Simulated annealing and tabu search are examples. Branch & bound has been designed in backward and forward versions, combined with many different bounding techniques. Beam search also has depth first and best first searches. There are several different versions of temperature functions in simulated annealing (Matsuo et al. 1989; Reeve 1993). Tabu search has many varieties in building and managing tabu lists. Its diversity in the literature is well summarized by Reeves

(1993). Genetic algorithms, in particular in their applications to scheduling problems, have also undergone diverse variations. New approaches on crossover (Webster et al. 1997; Norman and Bean 1994), mutation and parallel genetic algorithms (Jog et al. 1998) have been tried for better efficiency. Cleveland & Smith (1985) and Lee et al. (1990) summarized the different crossover methods.

The third group consists of hybrid methods. Tabu searches have been combined with genetic algorithms (Costa 1995). Genetic algorithms are mutated with simulated annealing. There are also hybrids of tabu search and simulated annealing (Malek et al. 1989). These hybrid methods in the literature also provide information on building a unifying algorithm that makes the existing methods mutable with each other.

There are several studies combining the existing methods from macro point of view. Ferland et al. (1996) offers an object-oriented approach to combine the methods in combinatorial optimization. Morton et al. (1993) generalize all neighborhood search-based methods as hill-climbing methods. The scope of their studies, however, covers a small number of methods. This study suggests more generalized framework covering most methods in all three groups previously introduced. The evolutionary approach we suggest in this study is similar to the third group in that both use existing methods. But there is a significant difference between these two. One depends on human intuition and the other systematically examines all possible combinations. Therefore, the approach we suggest in this study can be considered a fourth group, *a systematic evolutionary approach in designing new methods from existing methods.*

3. Genetic Programming for New Methods Design

Algorithms and heuristics from the literature of combinatorial optimization are analyzed as methods in genetic pool. An understanding of the existing methods will support discussion of the advantages and disadvantages of each of them. Diverse efforts on designing new algorithms or heuristics, generally in the form of hybrid methods, are also studied in the literature. The main features of each method are studied, along with its limitations.

3.1 Analysis of Algorithms and Heuristics in Combinatorial Optimization

The features of well-known algorithms and heuristics are studied with their behavior of search in the solution space. Each method has its own search technique. Some start the search from a random point and continue the search by the hill-climbing while the others start with a partial solution and complete the solution by filling in the locally best element one by one. Though it is not easy to find a unifying algorithm that explains all these diverse methods, this study tries to build the kernel that covers most methods in the literature.

We can largely classify the methods in combinatorial optimization into two groups: optimization methods and heuristic methods (Baker 1995). The optimization methods, such as Dynamic Programming (Srinivasan 1971) and some Branch & Bound algorithms (Greenberg 1968), must inevitably find the optimal solutions for given problems. Heuristics, however, are not guaranteed to produce optimal solutions. Examples of heuristics are Neighborhood Search (Wilkerson and Irwin 1971), Genetic Algorithms (Holland 1975), Beam Search (Loweree 1976), Simulated Annealing (Kirkpatrick et al. 1983), and Tabu Search (Glover et al. 1989). This study evaluates these methods according to two criteria: the quality of the solution and the time it takes to get the solution.

3.2 A General Paradigm Deriving a Kernel as a Unifying Representation Scheme of the Methods

The methods in the literature are classified into several groups following their characteristic of search behavior in the solution space: 1) neighborhood based hill climbing searches, 2) branching based searches, and 3) dynamic sets programming. Then, a generalized representation scheme for each of these groups are prepared. Finally, these three representation schemes are combined into a generalized kernel explaining all three types of methods. There are steps we followed to build the kernel.

3 Genearlization Steps

Step 1: Categorize all methods into three groups.
Step 2: In each category of methods, generalize the methods in a category into one based on the 6 rules below.
Step 3: Generalize the three categories into one unifying method, the kernel, based on the 6 rules below.

A method, either a method in the literature or a generalized method, consists of subparts. These parts can be divided into generalizable or optional parts. The generalizable part is the part that can be shared by other methods. The optional part is the part, which is unique to one method. A generalized method is derived from these generalized parts and optional parts when combined in the right order. To unify the methods introduced above, the following set of rules generalizing them has been considered.

6 Generalization Rules

Rule 1: Decompose each method into small steps. For example, a simple neighborhood search can be decomposed into the steps below:

Step 1: Find a solution randomly or by following another method.
Step 2: Generate solutions around the solution in

Step 2.

Step 3: Find the best solution among the solutions in Step 2.

Step 4: If there is an improvement in the performance of the new solution in Step 3, go to Step 1. Otherwise, stop the process and take the solution as the final solution.

Rule 2: The kernel consists of the generalized features among the methods and the unique feature of each method. For example, if method X = aBc and method Y = AbD, the generalized form of X and Y is abcD. The assumption here is that the upper case letter is contained in a lower case letter (a $\supset$ A). This means the letter *a* is a general form of the letter *A*. Each letter represents a fact or process in the methods.

Rule 3: If one feature can be generalized into the other, add the generalized one into the kernel. If one feature is a more generalized concept than the other, combine them into one feature group by generalizing them. For example, producing an initial solution set by general neighborhood search-based local search and branch-based search are different. Group them into one and save them as different features in one group such as *producing-initial-solutions*. aB** in X and Ab** in Y are generalized into ab**.

Rule 4: If a feature is unique, and thereafter which cannot be generalized, move it into the new combined string. The letter c and D in Rule 2 are examples as they are mutually exclusive features. For another example, tabu search can be generalized into general neighborhood search except for the tabu lists. The feature of a tabu list is added into the generalized method.

Rule 5: Categorize all feature sets into a similar group. If they generate new solutions around a search point, group them with a set name such as *generate-new-solution-around-a-search-point*. If they select the best solution from the neighborhood, group them with a set name such as *select-the best-solution-from-the-neighborhood*. If they are checking stopping condition, give a set name such as *stopping-condition*.

Rule 6: If the same features appear more than once, place them in the same method, and treat them as a separate feature. Therefore, AB and BA are different methods. For example, if method X = aBc and method Y = AbAD, the generalized form of X and Y is abAcD or abcAD.

Based on the 3 genrealization steps and 6 generalization rules, along with an analysis of the methods in the literature, a unifying method combining the methods in the literature of combinatorial optimization has been derived. Section 3.3 shows how this kernel works when it is implemented by computer programming.

A Suggested Kernel with 6 Steps

Step 1: Initialization
Initialize status variables.
Generate initial solution(s).

Step 2: Record "good" solutions
Select solutions to keep.

Step 3: Check if the algorithm should terminate or execute another iteration
If stopping condition is satisfied, exit the function.

Step 4: Select solution(s) that will be used to generate new solution(s)
Select solution(s) to reproduce.

Step 5: Generate new solution(s) from solution(s) in Step 4
Loop over solution(s) to reproduce.
{select solution(s) to reproduce
,generate solution(s) from selected solutions
, select child solution(s) to keep
, update memory on the old solutions
, remove selected on the top line inside this loop.}
End of loop.

Step 6: Update information on algorithm status
Update {Increment the loop counter, The iteration of improvement, The magnitude of improvement, The percentage improvement, True if improve on best solution, Update the cost of the best solution, Update Store total CPU time, Update algorithm status.}
Go to Step 2

Step 1 initializes all the variables in the program. It initializes the initial population. The trial solutions can be generated randomly for hill climbing methods such as a neighborhood search, simulated, annealing, tabu search, and genetic algorithm. It can also be generated following one of the methods in the literature. For example, the solutions in the initial population of the genetic program can be produced by neighborhood search method. For branch & bound, beam search, and dynamic programming, partial solutions are produced as an initial population. Variables checking the run time, iteration counter, and improvement of the performance are set to zero.

Step 2 records the best solution set from the population combining the solutions in the previous iteration and the solutions in the current iteration. The new solutions in the current iteration are produced from the old solutions in the previous iteration. As the new solutions do not always produce better solution, the best solution is selected from both old and new solutions set. In this study, we call this solutions set a population. In branch & bound, beam search, and dynamic programming, the best partial solutions are saved.

Step 3 checks the termination condition. If the condition is met, it stops the iteration. Otherwise, it evaluates the performance of the solutions in Step 2 to check the

improvement of the performance. For example, if the performance of the newly generated solutions in genetic algorithms does not improve after a given number of iterations of evolution, the iteration stops here. In branch & bound, beam search, and dynamic programming, it stops when the best solution is in complete solution form.

Step 4 selects solutions that will be used to generate new solutions. In neighborhood search, for example, the best solutions set is selected to generate neighborhood solutions. In genetic algorithms, the solutions with better performance have more chance to be selected for this set.

Step 5 generates new solutions using the solutions in step 4. In neighborhood search, the neighborhood solutions are generated a method such as pairwise exchange. In genetic programming, solutions are generated by mutating solutions in the previous step.
In branch & bound, beam search, and dynamic programming, solutions are automatically generated following their rules.

Step 6 updates the variables initialized in Step 1. The iteration goes back to Step 2. Step 2 through Step 6 is iterated until the stopping condition is satisfied.
The kernel is comprised of *if-then* statements, looping control structures, arithmetic operations, set function operations, and five main functions. The logic of these functions is controlled by the parameter values in the set P. The specific values of the parameters, which essentially define a specific algorithm, are created and reproduced in a manner that is analogous to the process of evolution in nature.

3.3. Implementation of the GP to Deriving New Methods

Human cell has a DNA, which is a huge string of repeating nucliotide units. Each of these units are comprised of a phophate group, a sugar, and bases of ademine (A), thymine (T), cytosine (C), and guanine (G) (Primrose 1997). As these four base elements are the recipe for the machinery of human life. Five function elements group of *Generate*, *Select*, *Ealuate*, *CPUtime*, and *Stop* are the basic building blocks of chromosome in this study. A chromosome, which corresponds to a method, is represented by the parameter values in set P. This can be a method or hybrid of among the methods in the literature. Each chromosome in the population solves randomly generated problem instances. In our testing problems, the size of the population is between 10 and 500 and the number of problem instances is between 20 and 100. The fitness of each P set is assessed by the relative quality of the solution generated by a given P set. The measure of quality for this set, for example, would typically be an additive function of CPU (Computer Processing Unite) time, cost of being late, and possibly memory requirements (e.g. *w1**CPU time + *w2**solution quality + *w3**memory requirements). Members of the population die and reproduce in a biased random way according to the fitness of a solution.

There are various choices reflected in the parameter values of P for each step in the kernel. Different combinations of parameter values in the kernel replicate wide variety of algorithms in the literature such as Neighborhood Search, Tabu Search, Branch & Bound, Simulated Annealing, Simulated Learning, and Genetic Algorithms. The number of unique algorithms that can be created by this approach is currently 10^{20} theoretically from combinations based on the number of selection functions, generation functions, evaluation functions, and stopping functions in their positions in the kernel.

If the kernel is the basic structure of a house, functions are building blocks. The five function groups that are executed in the kernel are: **Select**, **Generate**, **Evaluate**, **CPUtime**, and **Stop**. Each function group is summarized below (detailed steps of the functions will be presented after defining the kernel logic). As a matter of convention, we use the lower case letter 'x' (e.g., x_a) to denote fixed parameters and the upper case letter 'X' (e.g., X_a) to denote sets of data that may change during the execution of the kernel.

Kernel for Implementation

Step 1: Initialization
$CPUtime=Count=LastImprovement=Improvement=\%Improvement=0$
$X_{pop} = X_{perm} = X_{good} = X_{bad} = X_{status} = \varnothing$

X_{new} = **Generate**(I,$x_{measure}$,$x^1_{gen\#}$,$x^1_{gen\_method}$,X_{good},X_{bad},X_{status},X_{pop})
$MinCost$ = min{**Evaluate**(I,$x_{measure}$,X_{new})}
X_{status} = {*CPUtime*,*Count*,*LastImprovement*,
Improvement,*%Improvement*,*MinCost*}

Step 2: Record "good" solutions
$X_{temp} = X_{pop} \cup X_{new}$
X_{perm} = **Select**(I,$x_{measure}$,$x^1_{sel\#}$,$x^1_{sel\_method}$,X_{good},X_{bad},X_{status},X_{perm},X_{temp})

Step 3: Check if the algorithm should terminate or execute another iteration
:if **Stop**(I, $x_{stop\_condition}$, X_{status}, X_{new})
X_{temp} = **Evaluate**(I,$x_{measure}$,X_{perm})
:exit while returning the values of X_{status}, X_{temp}, & X_{perm}
:endif

Step 4: Select solution(s) that will be used to generate new solution(s)
X_{pop} = **Select**(I,$x_{measure}$,$x^2_{sel\#}$,$x^2_{sel\_method}$,X_{good},X_{bad},X_{status},X_{pop},X_{new})

Step 5: Generate new solution(s) from solution(s) in X_{pop}
$X_{new} = \varnothing$
$X_{temp} = X_{pop}$

:while $|X_{temp}| > 0$
X_{rep} = **Select**(I,$x_{measure}$,$x^3_{sel\#}$,$x^3_{sel\_method}$,X_{good},X_{bad},X_{status},X_{temp})
X_{kid} = **Generate**(I,$x_{measure}$,$x^2_{gen\#}$,$x^2_{gen\_method}$,X_{good},X_{bad},X_{status},X_{rep})
$X_{new} = X_{new} \cup$ **Select**(I,$x_{measure}$,$x^4_{sel\#}$,$x^4_{sel\_method}$,X_{good},X_{bad},X_{status},X_{kid})
X_{good} = **Select**(I,$x_{measure}$,$x^5_{sel\#}$,$x^5_{sel\_method}$,X_{good},X_{bad},X_{status},X_{new},X_{kid})

X_{bad} = **Select**(I,$x_{measure}$,$x^6_{sel\#}$,$x^6_{sel\_method}$,X_{good},X_{bad},X_{status},X_{new},X_{kid})
$X_{temp} = X_{temp} \backslash X_{rep}$
:endwhile
Step 6: Update information on algorithm status
$Count = Count + 1$
X_{temp} = min{**Evaluate**(I,$x_{measure}$,X_{pop})}

```
:if min{Evaluate(I,x_measure,X_new)} < X_temp
   LastImprovement = Count
   Improvement = X_temp - min{Evaluate(I,x_measure,X_new)}
  %Improvement = [X_temp - min{Evaluate(I,x_measure,X_new)}]/X_temp
   :if min{Evaluate(I,x_measure,X_new)} < MinCost

      MinCost = min{Evaluate(I,x_measure,X_new)}

   :endif
:endif
CPUtime = CPUtime

X_status = {CPUtime,Count,LastImprovement,
            Improvement,%Improvement,MinCost}

:goto Step 2
```

The variables in above kernel are explained below:

. *Iteration* contains the current iteration number of the major loop in the algorithm (the major loop is comprised of Steps 2 through 6)
. *LastImprovement* contains the most recent iteration that resulted in a solution with lower cost than the previous best solution
. *Improvement* contains the magnitude of improvement found in iteration *LastImprovement*
. *%Improvement* contains the percentage of improvement found in iteration *LastImprovement*
. *MinCost* contains the cost of the best solution attained
. *CPUtime* contains the CPU time since the kernel began execution
. X_{pop} is a set variable that contains the population of solutions being considered in the current iteration (solutions are specified as a sequence of job numbers)
. X_{perm} = is a set variable that contains solutions that are not to be discarded (this set is typically the set of the best solutions found so far)
. X_{temp} is a set variable for temporary storage
. X_{rep} is a set variable that contains solutions that will be the basis for generating new solutions. For example, through pairwise interchange of jobs, or through recombination, or any number of other means. In the terminology used in discussions of genetic algorithms, X_{rep} is the set of solutions that are allowed to *reproduce*
. X_{kid} is a set variable that contains new solutions that are generated from existing solutions (i.e., in a sense, these solutions are "children" of an "older generation" of existing solutions)
. X_{new} is a set variable that contains new solutions that are generated from existing solutions and that are to be kept for further consideration
. X_{good} and X_{bad} - see description under the select function

4. Tests and Analysis

The following describes a design to test an experiment with genetic programming for new methods design. The maximum population size is 1000, the crossover rate is 0.7 or 0.6, and the mutation rate is 0.03. The crossover method is a PMX method. The solution here represents not only a method, but it also a complete computer program solving combinatorial problems. Tests have been performed with different sets of problems, maximum time limit in evaluating one solution, initial node size, and methods of initializing the population. The evaluation of a solution is based on weighted performance and time. Tests have been performed with different sets of weight values assigned to performance and time. A problem set has one to five instances and each instance has a maximum of 10 jobs in it. The maximum time allowed to evaluate a solution ranges from 12 seconds to 20 seconds. This time limit helps the program run efficiently by cutting out inefficient solutions. The population is initialized in three different ways: 1) random initialization, 2) intentional initialization with replication of existing methods, and 3) a mix of methods 1 and 2. The population in the genetic programming is normally initialized randomly. But, the experiments show that a large portion of the solutions in an initial population is very low in performance quality and this leads to poor quality of solutions due to a lack of good schema at the beginning stage of the genetic search. To boost the quality of solutions, some algorithms or heuristics are intentionally inserted into the initial population. In this way we artificially mix good schemas into the solution pool to take advantage of them. Sometimes the evolution starts only with replicated methods to check how methods in the literature can evolve together. For testing purposes, 7 selection functions, 4 generation functions, and 3 stop functions are selected into the gene pools, where there are a set of selection functions and a set of generation functions. The generation functions either generate a set of solutions from a given set of solutions or generate an initial population. The first generation method randomly generates a number of solutions into the population. This function is generally used to initialize the populations. The second generation method generates a set of solutions around a set of input solutions. This function feature is from neighborhood search-based searches. The third generation method generates a set of solutions by mutation and crossover from a set of input solutions. This function feature is from genetic algorithms. The fourth generation method generates a set of solutions from a set of partial solutions. This function feature is from branch & bound and beam searches. When the generation function receives parameter values corresponding to other than these four methods, the default is to return the input solutions as the output.

4.1 Definition of the Problems for the Test

A set of job shop problems, as in Table 4.1, with processing time, due date, arrival time, and weights on the jobs are randomly created. The objective function is to minimize total weighted tardiness. A problem set consists of several sets of sequencing problems, each with 10 jobs in it. A candidate solution, in the form of string of numbers, is applied to this set of problems. Each number in the string means a function or method. The performance of the solution is the sum of tardiness from all the problems in a problem set.

Job	1	2	3	4
p_j	5	6	9	8
d_j	9	7	11	13
w_j	1	1	1	1

Table 4.1 An Example of Job Shop Problem

4.2 Test Results and Analysis

As the number of problems in a problem set or the population size increases, the more time is needed to evaluate one solution in populations. Table 4.2 shows the test result with five input parameters: number of instances, maximum time to evaluate a method, the method of initializing population, weight on the performance, and weight on the time.

Following the definition of functions in the previous chapter, the algorithm represented by the solution of the string is analyzed. The first column in Table 4.2 represents output numbers. The second column represents the number of problems in a problem set. The third column represents the maximum time limit in evaluating one solution in populations. The fourth column represents the population size, which is the number of solutions in a population and the method of initializing the population. *Rep* in the fourth column means the population is initialized with methods in the literature. *Random* in the column means the population is initialized with random generation. *Both* means the population use both rep and random initialization methods. The fifth and sixth columns represent the weights on performance and time when evaluating solutions. The last column represents a solution that is actually a method in the form of a numeric string. A solution consists of 17 numbers: the first 6 numbers represent the number of solutions to be selected from a given parent population. The next 6 numbers represent selection methods for six select functions in the kernel. The next 2 numbers represent the number of solutions generated by a generation method. The next 2 numbers represent generation methods. The last number represents the stopping condition. The solution in Output 1, for example, has a solution of { 1, 5, 1, 1, 1, 2, 1, 1, 2, 1, 0, 0, 2, 1, 1, 2, 1}. Based on this method, the kernel first generates two solutions randomly in Step 1 of the kernel. Then it selects the best solution from these two in Step 2. In Step 3, it evaluates the solution. It then selects the best in Step 4. In Step 5, it picks the one it chose in Step 4 for reproduction. Then, it generates solutions in the neighborhood of the selected solution. Then it selects the best of the neighborhood solutions. It does not use tabu memory or simulated learning for optimization.

. # : Serial number of new methods as outputs from tests
. *NOP*: Number of Problem Sets
. *MTL*: Maximum Time Limit in Seconds
. *IPS*: Initial Population Size
. *WOP*: Weight on Performance
. *WOT*: Weight on time

#	NOP	MTL	IPS	WOP	WOT	Methods
1	1	12	12 (rep.)	0.9	0.1	1 5 1 1 1 2 1 1 2 1 0 0 2 1 1 2 1
2	1	12	12 (rep.)	0.5	0.5	1 2 1 1 1 2 1 1 2 1 6 0 2 2 1 2 1
3	1	12	12 (rep.)	0.1	0.9	1 2 1 1 1 2 1 1 2 1 0 0 2 1 1 2 1
4	1	12	12 (rep.)	1.0	0.0	1 5 1 1 0 1 4 1 1 2 7 0 7 2 2 4 4 3
5	2	12	12 (rep.)	0.9	0.1	1 1 1 1 1 1 1 7 2 1 0 1 2 2 1 2 1
6	2	12	12 (rep.)	0.5	0.5	1 3 1 1 1 3 1 1 2 1 0 0 2 0 1 2 1
7	2	12	12 (rep.)	0.1	0.9	1 1 1 1 1 1 1 1 2 1 6 0 2 2 1 2 1
9	3	18	12 (rep.)	0.9	0.1	1 1 1 1 1 1 1 1 2 1 5 0 2 1 1 2 1
10	3	18	12 (rep.)	0.5	0.5	1 3 1 1 1 3 1 7 2 1 0 0 1 2 1 2 1
11	3	18	12 (rep.)	0.1	0.9	1 1 1 1 1 1 1 1 2 1 0 0 2 1 1 2 1
12	5	20	12 (rep.)	0.9	0.1	1 2 1 1 1 2 1 1 2 1 6 1 2 2 1 2 1
13	5	20	12 (rep.)	0.5	0.5	1 2 1 1 1 2 1 1 2 1 6 0 2 2 1 2 1
14	5	20	12 (rep.)	0.1	0.9	1 1 1 1 1 1 1 1 2 3 0 0 2 1 1 2 1
15	2	20	80 (ran.)	0.9	0.1	1 2 1 1 1 2 1 1 2 1 0 0 3 2 1 2 1
16	2	20	80 (ran.)	0.5	0.5	1 1 1 1 1 1 1 1 2 1 7 7 1 7 0 1 1 2 1
17	2	20	80 (ran.)	0.1	0.9	1 1 1 1 1 1 1 1 2 1 0 2 2 5 1 1 1 2 1
18	2	20	80 (both)	0.9	0.1	1 5 1 1 1 5 1 2 2 1 0 7 2 9 7 2 1 2 1
19	2	20	80 (both)	0.5	0.5	2 2 1 1 1 2 1 1 2 1 2 1 2 8 9 2 1 2 1
20	2	20	80 (both)	0.1	0.9	2 2 1 1 1 2 1 3 2 7 6 3 8 3 2 1 2 1

Table 4.2 Test Results with New Methods

Both solutions in Output 2 and Output 3 also represent neighborhood search method. The solution in Output 4 represents branch and bound with a tabu list. The number 4 in 15th and 16th represent the branching method and the number 7 in 10th and 12th position shows tabu list and its implementation. The solutions in Output 5, 6, 7, 9, 10, 11, 12, 13, and 15 represent a deviated version of a neighborhood search method. The solution in output 14 is a neighborhood search. But, when it selects the best solution from neighborhood search, it chooses the solution with a probability based on performance. The solutions in Output 17 and 19 are all neighborhood searches. They, however, produce large number of solutions in the initial population. The solutions in Output 16 and 18 is a tabu search with a large size of initial population. The solution in Output 20 is a simulated anneal with a large size of initial population.

Below facts are derived from analysis of test results:

Fact 1: If the Weight on Time is less than 0.1, Branch & Bound with Tabu list is the best.

Fact 2: If the Weight on Time is greater than or equal to 0.1 and Number of Problems is less 4, then Neighborhood Search is the best.

Fact 3: If the Weight on Time is greater than or equal to 0.1 and Number of Problems is greater than or equal to 4 and Weight on Time is less than 0.3, then Probability based Neighborhood search is the best.

Fact 4: If the Weight on Time is greater than or equal to 0.1 and Number of Problems is greater than or equal to 4 and Weight on Time is greater than or equal to 0.3, then Neighborhood search is the best.

These rules can be used to make a decision on the method selection in the future. When the time allowed to make a method selection is enough, it is possible to produce the best method from genetic simulation. In case, however, the decision has to be made in a short time, we can choose the best method following the rules from past simulation data.

4.3 Findings

Neighborhood search is overwhelming in most of the solutions reported in Table 4.2. The reason for the dominance of the neighborhood search may be its generality. If a method is specialized instead of generalized in a small highly competitive population, genetic programming can hardly catch the specialized one. It is a lot easier to grow weeds than flowers. Generality is adaptability in natural selection. It means a greater chance the general ones to survive in competition. But, when the time factor is totally ignored, the competition changes in favor of mathematical approaches. Now they have a greater chance to be selected. As is shown in the case of Output 4, if we assign all weight to the performance, branch and bound will be the best method for a problem. It means when we totally ignore the time factor in evaluating methods, branch based methods, especially branch and bound, will be dominant methods. It also seems that tabu list helps the search of branch and bound. The last five cases in Table 4.2 shows that when the population size increases, the number of solutions in the initial population dramatically increases. When the weight of time is close to 1, neighborhood search is dominant. An intuitive result shows that a simple heuristic seems to be efficient under time pressure. We can also see that tabu searches emerge in three cases of output 4, 16, and 18. A summary of findings is below:

- Generality wins over specification in intense competition.
- Generality wins under a time limit.
- Mathematical methods are dominant in performance.
- Tabu list helps efficiency in most cases.

5. Conclusion

In this study, we confirm that the kernel, suggested by generalizing paradigm, works as a generic framework in the evolutionary approach. Within the kernel, all possible solutions, in the form of programs, from genetic combinations are in valid forms. This genetic system also adapts to new problem sets by changing its solutions for the problems. It also demonstrates that this evolutionary approach can be successfully applied to the design of new algorithms. The test, however, shows that the time to derive solutions by genetic simulation is relatively long.

The main contribution of this study is on the theoretical side. It suggests a generic framework enabling mutation among heterogeneous members in a genetic population. The major contributions of this study can be summarized as below:

- Classification of Main Algorithms and Heuristics in Combinatorial Optimization into Groups by Their Features
- A General Paradigm Deriving a Kernel with Steps and Rules as a Guideline
- A Kernel as a General Structure Covering the Main Methods in Combinatorial Optimization
- Facts Found from Genetic Simulation: the performance of each method with different condition factors such as size of problems, weights on performance & time
- An Evolutionary Approach Implemented to Designing New Methods

This suggested generic system not only produces solutions in the form of programs, it also adapts itself to new problem environments. As problems change, the system finds the best method from combinations in its genetic pool. The solutions produced from the system from genetic cultivation are algorithms in a program form. They also return the best solution along with the best method cultivated in a genetic pool for a problem set.

References

[1] Aarts, E. and Korst J. (1989). *Simulated Annealing and Boltzmann Machine: A Stochastic Approach to Combinatorial Optimization and Neural Computing*, New York, Wiley.

[2] Baker, K. R.(1995). *Elements of Sequencing and Scheduling*, Forthcomming.

[3] Biegel, J. E., Davern, J. J. (1990). "Genetic Algorithms and Job Shop Scheduling," *Computers and Interstrial Engineering*, Vol. 19, pp.81-91.

[4] Booker, L. B., Goldberg, D. E., Holland, J. H. (1989) "Classifier Systems and Genetic Algorithms," *Artificial Intelligence*, Vol. 40, No. 1-3, pp.235- 282.

[5] Cleveland, G. A., Smith, S. F. (1989). "Using Genetic Algorithms to Schedule Flow Shop Releases," ICGA' 89, pp.160- 169.

[6] Cona, J. (1995). "Developing a Genetic Programming System," *AI Expert*, Feb., pp.20-29.

[7] Costa, D. (1995). "An Evolutionary Tabu Search Algorithm and the NHL Scheduling Problems," *INFOR*, vol. 33, no. 3, pp.161-177.

[8] Davis, L. (1991). *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York.

[9] Dev, K., and Murthy, C (1995). "A genetic algorithm for the knowledge base partitioning problem," *Pattern Recognition Letters* 16, pp. 873-879.

[10] Ferland, J., Hertz, A. and Lavoie, A. (1995). "An Objective-Oriented Methodology for Solving Assignment-Type Problems with Neighborhood Search Techniques", *Operations Research*, Vol. 44, No.2, pp.350.

[11] Fogel, D. B. (1994). "An Introduction to Simulated Evolutionary Optimization," *IEEE Transaction on*

Neural Networks, Vol. 5:1, pp.3-14.

[12] Glover, F. and Laguna, M. (1989). "Target Analysis to Improve a Tabu Search Method for Machine Schduling," *Advanced Knowledge Research Group*, US West Advanced Technologies, Boulder, Co.

[13] _____ (1993). "Tabu Search," *Modern Heuristic Techniques for Combinatorial Problems*, Colin R. Reeves (Ed), Blackwell Scientific Publications, Oxford, pp.70-150.

[14] Goldberg, D. E. (1988). *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison Wesley.

[15] Goldberg, D. E., and Deb, K. (1991). "A Comparative Analysis of Selection Schemas Used in Genetic Algorithms," *Foundations of Genetic Algorithms,"* Morgan Kaufmann.

[16] Greemberg, H. (1968). "A Branch-and-Bound Solution to the General Scheduling Problem," *Operations Research* 16, pp.353-361.

[17] Holland, J. H. (1992). "Genetic Algorithms," *Scientific American*, Vol. 267, no. 1, pp.66-73

[18] Hove, H. V., Verschoren, A. (1996). "Genetic Algorithms and Recognition Problems," *Genetic Algorithms for Pattern Recognition*, Pal, S. K.,and Wang, P.P., pp.145-165.

[19] Jog, P., Suh, J., and Van Gucht, D. (1991). "Parallel Genetic Algorithms Applied to the Traveling Salesman Problem," *Siam Journal of Optimization*, 4, Nov., pp.515-529.

[20] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). "Optimization by Simulated Annealing," *Science* 220, pp.671-680.

[21] Koza, J. R. (1992). *Genetic Programming*, Cambridge, Mass. MIT press.

[22] _____(1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press.

[23] _____ (1996). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press.

[24] _____ (1999). Bennett III, F.H., Andre, D., and Keane, M.A., *Genetic Programming III: Darwinian Invention and Problem Solving*, San Francisco, Morgan Kaufmann Publisher.

[25] Lee, I, Sikora, R. and Shaw, M. J. (1992). "Joint Lot Sizing and Sequencing With Genetic Algorithms for Schduling: Evolving the Chromosome Structure", *Proceeding of the Fifth International Conference on Genetic Algorithms and Their Applications*, pp.383-389.

[26] Malek, M., Guruswamy, Pandya, M. and Owens, H.(1989). "Serial and Parallel Simulated Annealing and Tabu Search Algorithms for the Traveling Salesman Problem", *Annals of Operations Research*, 21, pp.59-84.

[27] Matsuo, H., Suh, C. J. and Sullivan R.S. (1989). "A controlled search simulated annealing method for the single machine weighted tardiness problem," *Annals of Operations Research* 21, pp.85-108.

[28] McKay, N. N., Safayeni, F. R., Buzacott, J. A.(1999). "Job-Shop Scheduling Theory: What Is Relevant?" *Interfaces* 18: 4 July-August, pp.84-89.

[29] Morton, T. E. and Pentico, D. W.(1993). *Heuristic Scheduling Systems with Applications to Production Systems and Project Management*, John Wiley & Sons.

[30] Norman, B. A., Bean, J. C. (1994). *Random Keys Genetic Algorithm For Job Shop Scheduling*, Technical Report 94-5, University of Michigan, College of Engineering, Ann Arbor, Michigan.

[31] Pinedo, M. (1996). *Scheduling: Theory, Algorithms, and Systems*, Prentice Hall, New Jersey.

[32] Reeves C. R. (1993). *Modern Heuristic Techniques for Combinatorial Problems*, John Wiley & Sons, Inc.

[33] Sridhar, J., and Rajendran, C. (1994). "A Genetic Algorithm for Family and Job Scheduling in a Flowline-Based Manufacturing Cell," *Computers and Industrial Engineering*, Vol. 27, pp.469-472.

[34] Srinivasan, V. (1971). "A Hybrid Algorithm for the One-Machine Sequencing Problem to Minimize Total Tardiness," *Naval Research Logistics Quarterly* 18, pp.317-327.

[35] Webster S., Jog, P., and Gupta, A. (1997). "A Genetic Algorithm for Scheduling Job Families on a Single Machine with Arbitrary Earliness/ Tardiness Penalties and an Unrestricted Common Due Date", U.W-Madison Working Paper.

[36] Wilkerson, L.J., and Irwin J. D. (1971). "An Improved Algorithm for Scheduling Independent Tasks," *AIIE Transactions* 3, pp.239-149.