# Program Search with a Hierarchical Variable Length Representation: Genetic Programming, Simulated Annealing and Hill Climbing

Una-May O'Reilly[1] and Franz Oppacher[2]

[1] Santa Fe Institute, Santa Fe, NM, 87505, USA
[2] School of Computer Science, Carleton University, Ottawa, CANADA

**Abstract.** This paper presents a comparison of Genetic Programming(GP) with Simulated Annealing (SA) and Stochastic Iterated Hill Climbing (SIHC) based on a suite of program discovery problems which have been previously tackled only with GP. All three search algorithms employ the hierarchical variable length representation for programs brought into recent prominence with the GP paradigm [8]. We feel it is not intuitively obvious that mutation-based adaptive search can handle program discovery yet, to date, for each GP problem we have tried, SA or SIHC also work.

## 1 Introduction

Genetic Programming (GP) [8] is a recent paradigm in the lineage of search techniques for performing program induction [2, 3, 4, 5, 10, 12]. The sources of its effectiveness are: 1) an evolution inspired exploitation and exploration mechanism and, 2) the use of a hierarchical, variable length representation for programs. In the former respect, GP uses fitness proportional selection (á la natural selection) as a basis for choosing which individuals among a population of programs will be the parents of the next generation and a genetics based crossover operation for transforming two parents into two novel offspring. This provides a robust heuristic search approach [7, 6, 8]. Regarding the latter source, the hierarchical representation is supported by a genetic crossover operator which ensures the syntactic closure of all programs and permits programs of different lengths to be generated when two "parent" programs are crossed over. Thus in GP a fixed size and structure of programs in the search space does not have to be *a priori* specified.

One of the goals of this paper is to isolate GP's representation scheme from its evolution-based adaptation to understand the degree to which each is crucial in solving a program discovery problem. Given that a hierarchical variable length representation is used, we would like to obtain a better notion of when GP is the most appropriate paradigm for solving a particular program discovery problem compared with search techniques which traverse with a single point and are not inspired by genetics and evolution. Our approach is to conduct experiments which compare GP to Simulated Annealing (SA) and Stochastic Iterated Hill Climbing (SIHC).

In Section 2 operators which are capable of transforming programs (represented as trees) while permitting program size and length to vary are described. For SA and SIHC we have developed and present a mutation operator "HVL-Mutate", (HVL = Hierarchical Variable Length). Section 3 describes the primitive sets, fitness functions and test suites of the problems we attempt to solve with GP, SIHC and SA. In Section 4 the results of the experiments are presented and discussed.

## 2 Crossover and Mutation on Programs as Trees

The crossover operator and the hierarchical representation chosen for programs in Genetic Programming are complementary. Hierarchical representation of programs is both intuitive and common. It is helpful in denoting program structure both in terms of specification and execution. A program is represented as a rooted, point-labeled tree with ordered branches. Using LISP as an example the name of the S-Expression is the root of the tree and the arguments of the S-Expression (which may recursively be S-Expressions) are the ordered children of the root. In GP the nodes in each of two parent programs are numbered in a depth first search ordering and then two values, each in the node number range on a parent, are randomly selected as the crossover points. Finally, the two subtrees of the programs, each rooted at the node designated by the crossover point, are swapped. The value of the hierarchical dissection is that the syntax of programs is automatically preserved (i.e. closure is ensured) and programs of different structure and size are generated from parents to offspring.

The GP crossover is clearly an influential element of GP since, as in this experimentation, it often provides the sole means of exploring the search space. Crossover effects the promotion and disruption of partial solutions through the deletion and substitution of subsequences of programs selected on a fitness basis. The selection of crossover points would appear to be a crucial factor in GP because it determines how much "genetic material" (i.e. how much of an S-Expression) will be removed from one program and transplanted to another. The crossover operator introduced by Koza, which we call GP-XO, selects crossover points with a 90% probability bias towards subtrees which are not leaves. The proposed rationale for this bias is robustness and the fact that exchange of leaves is more akin to point mutation than recombination and thus is not explorative enough [8]. The reason for investigating alternative crossover point selection biases is that the actual impact of the 90% probability bias really seems less than clear cut. This is because the process by which programs are initially generated, the blind aspect of crossover and the syntactic constraints of the repertoire of primitives all interact to generate unpredictable leaf to node ratios. In fact, when leaves comprise less than 10% of the size of a tree, GP-XO is actually biased towards leaf selection contrary to the rationale of the bias. As well, presuming that the point mutation of a leaf is a "tweak" or rather small change to a program implies that leaves are constants, variables or simple predicates. In fact they could be functionally complex expressions which simply have no parameters. In this case a point mutation will in fact be "sufficiently" explorative. When the leaves are constants, variables or simple predicates it is also possible that tweaks may be appropriate as localized exploration.

We have devised and experimented with two other crossover operators that only differ in how the crossover point is selected:

- Height-fair-XO groups subtrees of a program by height, with equal probability chooses a group and then randomly selects one subtree of this group as the crossover point. Thus, the root and leaves of a subtree may be chosen with equal probability.
- Fair-XO randomly selects any subtree for swapping.

The objective of experimentation with three different crossover operators is to determine whether crossover point selection (i.e. bias) generally affects convergence rates and whether one crossover point selection scheme is overall superior to the others.

Both SA and SIHC require a mutation operator. Given that a hierarchical representation is chosen, the operator we have designed and used, HVL-Mutate, is inspired from the method for calculating the distance between trees [11]. That approach defines distance as the minimum cost sequence of editing one tree, step by step, to become the other using elementary operations such as substitution, insertion and deletion. In a similar manner, HVL-Mutate changes a program into another by either substituting one node for another (within syntactic constraints), deleting a subtree or inserting one with equal probability. Mutations are intended to be small changes so HVL-Mutate tries to minimize the change to a program within the constraints of supporting a non-binary variable length hierarchical representation. When a subtree is selected as the point above which a new node will be inserted it will always be used as a child of the new node if one is required. All remaining children of the newly inserted node are leaves. In the case of a deletion the node selected for deletion is replaced by the largest of its children. As well, when a leaf is chosen for deletion it is replaced by a different randomly chosen leaf. This algorithm does not guarantee that mutation will not drastically change the size and shape of a tree, but does reduce the probability of that event.

The advantage of HVL-Mutate is that it facilitates search with techniques which are not population based and which do not use the same selection criteria as GP. It permits isolation of the hierarchical variable-length representation aspect of GP from the rest of the GP algorithm.

## 3 Description of Experiments

We experiment with 3 problems: the 6-bit Boolean Multiplexer (6-Mult), the 11-bit Boolean Multiplexer, (11-Mult), and sorting (Sort-A). The Multiplexer task is to decode address bits and return the data value at the address. 6-Mult uses the primitives IF, OR, NOT, AND which take 3, 2, 1, and 2 arguments respectively. There are 6 variables (i.e. primitives which take no arguments): A0, A1, D0,...,D3 which are bound before execution to the address bits and data values of a test case. All 64 possible configurations of the problem are enumerated as test cases. A program's raw fitness is the number of configurations for which it returns the correct data value for the given address. 11-Mult is simply a larger scale version of 6-Mult using 3 address bits (A0- A2) and 8 data values (D0-D7). The test suite consists of 2048 test cases.

The task of a program in sorting is to arrange in ascending order the elements in an array. A program is run 48 times, each time with a different array bound to the primitive *array*. The array sizes in the test suite range from 2 to 6 and among the arrays there are 198 elements in total. Among the test suite are 3 arrays which are initially in sorted order and 47 elements initially in their correct positions. The raw fitness of a program is the sum of the number of elements found in the correct position after running the program.

The primitives for Sort-A are Do-Until-False, Swap, First-Wrong, Next-Lowest, and *array* of 1, 3, 1, 2, and 0 arguments respectively. Do-Until-False executes its argument until it returns nil. Swap returns nil if its first argument is not an array. When its first argument is an array, and the second and third arguments evaluate to integers in the range of that array's size it exchanges the elements at the positions corresponding to the integers and returns true. First-Wrong returns nil if its argument is not an array or if the array is sorted. Otherwise it returns the index of the first element which is out of order in the array. Next-Lowest returns nil if its first argument is not an array and its second argument is not an integer within the extent of the array. Otherwise it returns the index of the smallest element in the array after the position indexed by the second argument. At the beginning of each test case the primitive *array* is bound to the array which is to be sorted.

In order to compare the results of GP, SIHC, and SA each run was permitted the same maximum number of evaluations. The GP runs were run with a population of 500 for 50 generations. Both SA and SIHC were given a maximum of 25500 evaluations as this corresponds to the evaluations in GP. The fitness values in the GP runs for all three problems were scaled by linear and exponential factors of 2. Whatever crossover operator used, it was applied 90% of the time with the remaining 10% of individuals chosen by selection being directly copied into the next generation.

The Simulated Annealing algorithm [1] used was basic. Fitnesses were normalized to lie in [0,1] which fitness drops to [0,1]. A starting temperature of 1.5 was always decreased to a temperature over a given number of evaluations according to an exponential cooling schedule. For 6-Mult, 11-Mult,and Sort-A the final temperature was set to 0.0001, 0.0001 and 0.0015 respectively. In SA whenever a mutant is better than its parent it is accepted as the next point from which to move again. The probability with which a mutant with lower or equal fitness than its parent is accepted decreases with the temperature of the system and depending upon the difference in fitness between them according to a Boltzmann distribution.

The algorithm for SIHC is to generate a program at random and apply the mutation operator to it. If the mutant is superior in fitness the search moves to it. Otherwise another mutation on the original point is tried. The maximum number of mutations to generate from a point before abandoning it and choosing a new one at random is a parameter of the algorithm which we call max-mutations. Values of 50, 100, 250, 500, 2500, and 10000 for this parameter were tried. At maximum evaluations or when a perfect solution has been found the algorithm terminates. In our terminology a "step" is the acceptance of a mutant because it is an improvement, a "climb" is a succession of steps and "evaluations" defines the number of mutations performed in a climb.

This experimentation demonstrates that different search strategies can be used with a hierarchical variable length representation. Inevitably, it is also a source of comparison among the strategies in terms of performance. The comparison is expressed in a specific manner however: i.e., there is no "tuning" of a technique to its best potential on a particular problem. For example, in GP, for a given problem and primitive set, while it is known that population size can affect performance, only a fixed population size was used in this experimentation. In SA, for example, the cooling schedule could also have been adjusted to improve effectiveness but was not. Even before experimentation, it seems clear that a greedy hill climbing strategy will never outperform GP or SA because it has no strategy for escaping from local optima.

## 4 Experiments Results

**6-Mult:** Table 1 summarizes the GP results obtained for the 6-Mult problem with the 3 different crossover operators. The experiment consisted of a minimum of 30 runs for each crossover operator. Under a t-test [13] with 95% confidence the results show that Height-Fair-XO and GP-XO have a significantly better expected probability of success than Fair-XO. Another significant difference was the average fitness of the population at the end of a run (a run was terminated as soon as a perfect individual was found) where, in contrast to the comparison in terms of probability of success, Fair-XO achieved a significantly better result and GP-XO and Height- Fair-XO were indistinguishable. The two results together might be explained by the fact that Height-Fair-XO is slower to converge because it requires a higher population fitness before it can create an individual fitter than the present fittest in the population. Both Height-Fair-XO and GP- XO needed a significantly less mean number of evaluations than Fair-XO but they did not differ with each other significantly.

Every run of SA to solve 6-Mult was successful and on average SA required 54.2% (7.7%) of available evaluations. After experimentation with various values for the max- mutations parameter (see Table 2) the best result for SIHC occurred when it equaled 10000. With this parameter SIHC solved 6-Mult 77% of the time. On average a successful climb was 9.4 steps and the number of evaluations per step in a successful climb was 7336 resulting in an average of 777.1 evaluations per step. One way of interpreting this value is that it quantifies the local nature of the landscape, i.e., with the given operator, how much effort is required to find

**Table 1.** GP Crossovers and 6-Mult

| 6-Mult and Genetic Programming (GP) | Height-Fair-XO | GP-XO | Fair-XO |
|---|---|---|---|
| Percentage of Successful Runs | 86.7 (34.0) | 79.5 (40.3) | 60.9 (49.8) |
| Confidence Interval (99%, 95%, 90%) | 16,12,10 | 17,13,11 | 23,18,15 |
| Fittest Individual at End of Run (% of Opt) | 98.3 (4.5) | 98.0 (4.5) | 96.7 (4.6) |
| Fitness of Population (% of Opt) at End of Run | 77.6 (4.6) | 78.2 (5.2) | 81.6 (3.9) |
| Evaluations in a Successful Run (% of 25500) | 51.4 (15.0) | 48.4 (21.4) | 57.4 (20.1) |
| Evaluations Over All Runs (% of 25500) | 55.2 (21.9) | 55.7 (27.8) | 72.0 (26.2) |
| Tree Height of Successful Programs | 7.9 (2.7) | 6.9 (3.2) | 8.4 (2.3) |
| Tree Size of Successful Programs | 40.4 (18.4) | 42.8 (32.0) | 48.8 (25.1) |

a higher point than the present one. The fact that SIHC was successful implies that there may not be many local optima to stymie the search or that there may be many peaks of optimal height.

**Table 2.** 6-Mult Hill Climbing

| Max Mutations | Prob of Success (%) | Best Fitness (% of opt) | Avg Steps per Climb | Avg Evals per Climb | Evals: Step | Evals: Run (% of 25500) | Successful Runs Evals: Run (% of 25500) |
|---|---|---|---|---|---|---|---|
| 50 | 10 | 89.8 (3.1) | 2.6 | 80 | 30.8 | 94.0 | 59.0 (25.9) |
| 100 | 20 | 91.4 (3.6) | 3.5 | 165 | 47 | 81.7 | 17.8 |
| 250 | 20 | 93.5 (3.5) | 4.4 | 425 | 97 | 90.0 | 58.7 (28.2) |
| 500 | 40 | 96.6 (2.3) | 5.4 | 855 | 159 | 81.8 | 61.3 (22.0) |
| 10000 | 77 | 98.8 (2.8) | 9.3 | 9787 | 1052 | 61.5 | 57.7 (30.2) |

**Table 3.** 6-Mult Comparison of GP, SA and SIHC

| 6-Mult Search Strategy Comparison | SA | GP Height-Fair XO | SIHC |
|---|---|---|---|
| % Successful Runs | 100 | 86.7 | 77 |
| Fittest Individual | 100 | 98.3 | 98.8 |
| Evals of All Runs (% of 25500) | 54 | 77.6 | 61.5 |
| Evals of Successful Runs (% of 25500) | 54 | 51.4 | 57.7 |
| Tree Height of Successful Programs | 8.6 | 7.9 | 5 |
| Tree Size of Successful Programs | 58 | 40.4 | 23.9 |

Evidently 6-Mult as a program induction problem is simple because all three of SA, GP (with any of these crossovers), and SIHC were able to solve it. Table 3 compares GP with Height-Fair- XO to SA and SIHC. SA had a significantly superior probabilility of success versus the best performing GP crosssover operator

(Height-Fair-XO) and SIHC. In terms of the evaluations expected over all runs or in successful runs no search technique was statistically superior. Interestingly, SIHC did find successful programs which were significantly shorter (i.e. had less primitives or fewer nodes) and of less depth (i.e. the height of the program trees was less).

**11-Mult:** Results for the 11-Mult problem are summarized in Table 4. GP could not solve 11-Mult with 25500 evaluations. Regarding the three different crossovers, both Height-Fair-XO and GP-XO obtained significantly better best fitness and population fitness than Fair-XO but they did not differ between themselves significantly. Height-Fair-XO found the fittest program (93.8% of optimal) while the fitness of the best program found by Fair-XO was 87.9% and GP-XO was 87.6%. Only SA among the 3 search techniques was able to obtain

**Table 4.** GP Crossovers and 11-Mult

| 11-Mult and Genetic Programming (GP) | Height-Fair-XO | GP-XO | Fair-XO |
|---|---|---|---|
| Percentage of Successful Runs | 0 | 0 | 0 |
| Best Fitness Found (%) | 93.8 | 87.6 | 87.9 |
| Fittest Individual at End of Run (% of Opt) | 80.9 (5.0) | 79.2 (5.5) | 76.2 (4.3) |
| Fitness of Population (% of Opt) at End of Run | 74.0 (3.65) | 74.1 (4.4) | 71.2 (3.9) |
| Evaluations Over all Runs (% of 25500) | 100 | 100 | 100 |

a perfect solution. One solution was a program with tree height of 14 and 1891 nodes. This it accomplished on 3 out of 25 runs. When the number of evaluations was increased by 4500 to 30000 (which effectively slowed the cooling) then 2 more solutions were found. On average SA found a solution which was 93.4% of optimal (adjusted fitness of 1914). The results for SIHC when the maximum mutations parameter was varied from 2500 to 10000 were not significantly different. They are displayed in Table 5. It is interesting to note that a very simple, greedy heuristic that can be coded in less than one page outperforms a complicated and computationally expensive algorithm such as GP.

**Table 5.** SIHC and 11-Mult

| SIHC | | | | | Best Climbs | | | |
|---|---|---|---|---|---|---|---|---|
| Max Mut | Avg Steps per Climb | Avg Evals per Climb | Evals: Step | Avg Best Fitness (%) | Avg Steps per Climb | Avg Evals per Climb | Evals: Step | Best Fitness (%) |
| 50 | 5.0 | 107.6 | 21.5 | 73.6 | 20 | 476 | 23.8 | 77.35 |
| 100 | 6.6 | 229.0 | 34.5 | 75.5 | 32 | 525 | 16.4 | 76.62 |
| 250 | 10.5 | 637.2 | 60.8 | 81.2 | 60 | 2273 | 37.9 | 87.95 |
| 500 | 15.0 | 1466.0 | 97.0 | 81.6 | 32.5 | 2436 | 74.4 | 84.38 |
| 2500 | 13.2 | 3939.0 | 296.6 | 74.1 | 57 | 12203 | 214.1 | 95.31 |
| 5000 | 5.9 | 2518.9 | 429.3 | 88.9 (34.2) | 8.4 | 22093 | 2630.1 | 95.31 |
| 10000 | 25.6 | 21059.0 | 822.6 | 88.4 (39.0) | 40 | 25000 | 625 | 96.88 |

**Table 6.** Comparison of SA, HC, and GP for 11-Mult

| 11-Mult Search Strategy Comparison | Simulated Annealing | GP Height-Fair-XO | SIHC Max-Mut = 5000 |
|---|---|---|---|
| % Successful Runs | 16.7 | 0 | 0 |
| Best Fitness Found (%) | 100 | 93.8 | 95.31 |
| Fittest Individual at End of Run (% of Opt) | 93.0 | 80.9 | 88.9 |

The 11-Mult problem experiments were interesting because the degree of success of the three search techniques varied. Table 6 compares the performance of GP, SA and SIHC. With the small number of runs it is not possible to conclude that SA is statistically superior to GP or SIHC, however, neither GP nor SIHC ever obtained a perfect solution in 30 runs. When the expected best fitness is compared, GP using Height-Fair-XO (which had the best result) and SIHC when max-mutations = 5000 do not differ statistically significantly. GP is slightly outperformed by hill climbing when the best values are compared: the best fitness ever obtained by GP (with Height-Fair-XO) was 93.8% of optimal and with SIHC it was 96.88% of optimal (max-mutations = 10000).

**Sort-A:** The first column of Table 7 shows how Sort-A was handled by Simulated Annealing. SA had approximately an 88% probability of success and the expected fitness at 25500 evaluations was very close to optimal (87.4%). This is superior to GP which is shown in the second column. Sort-A required approximately 63% of the allowed evaluations in all runs and about 55% of allowed evaluations in successful runs. This is more than GP with GP-XO. The size and height of the trees of successful programs in Sort-A with SA were less than those obtained with the GP. Table 8 provides the details of SIHC and Sort-A.

**Table 7.** Sort-A Comparison of GP, SA and SIHC

| Sort-A Search Strategy Comparison | Simulated Annealing | GP GP-XO | SIHC (max-mu = 100) |
|---|---|---|---|
| % Successful Runs | 88.3 | 80.0 | 46.7 |
| Fittest Individual | 87.4 | 49.7 | 72.6 |
| Evals of All Runs (% of 25500) | 62.2 | 51.3 | 72.2 |
| Evals of Successful Runs (% of 25500) | 54.6 | 39.1 | 37.5 |
| Tree Height of Successful Programs | 5.64 | 6.8 | 6.7 |
| Tree Size of Successful Programs | 12.9 | 25.0 | 48.3 |

It is possible to hill climb to a solution in the fitness landscape regardless of the value of the max-mutations parameter. In fact there is no significant difference in the probability of success for the values of the max-mutations parameter we tried. The greedy nature of SIHC seems likely to be responsible for a signicantly lower probability of success when compared to GP or SA. From the details of successful climbs one can see that most climbs take very few steps. It appears that there are many randomly spaced small equally fit peaks.

**Table 8.** SIHC and Sort-A

| SIHC | | | | Successful Climbs | | |
|---|---|---|---|---|---|---|
| Max Mut | Prob of Success | Best Fitness (% of Opt) | Evals:Run (% of 25500) | Size | Height | Evals:Run (% of 25500) |
| 50 | 50 | 67.6 (47.7) | 64.3 | 52 (7.6) | 7.8 (1.3) | 28.7 (15.3) |
| 100 | 46.7 | 72.6 (40.7) | 72.2 | 48.3 (7.8) | 6.7 (1.7) | 37.5 (27.6) |
| 250 | 50 | 62.1 (55.5) | 72.2 | 64.4 (24.4) | 8.2 (1.9) | 44.3 (16.7) |
| 500 | 46.7 | 63.2 (50.0) | 68.2 | 67.1 (6.8) | 7.4 (1.3) | 31.1 (25.9) |

## 5 Summary and Future Work

Our first goal was to isolate the variable length hierarchical representation of GP from its neo-Darwinian inspired search strategy. This we did by using that representation and the HVL-Mutate operator with SA and SIHC. A valuable lesson is that a variable length hierarchical representation may be a more fundamental asset to program induction than any particular search technique. We observed mixed or comparable differences between operator and search techniques across different problems. This confirms the notion that the suitability of a search technique depends upon the fitness function and primitives chosen for a particular problem since these influence the nature of the search landscape.

It came as somewhat of a surprise to us that SA and even SIHC are effective on problems in the domain of program discovery. The results clearly suggest something that is somewhat counter-intuitive: that adaptive mutation, with the important qualification is that the programs are represented as trees, is sufficiently powerful to find correct programs in large dimension search spaces. Typically one thinks that programs are so sensitive to context that tweaks are too radical. Since GP swaps sub-trees which are actually sub- programs there is some semantic level exchange of encapsulated function which makes crossover seem more intuitive than mutation. Yet as easily, HVL-Mutate seems to indicate that any intuition that this sort of exchange is necessary is incorrect: the hierarchical representation seems to allow tweaks to explore the search space in a efficient manner.

GP is a kind of Genetic Algorithm (GA). The crossover operator in GAs has been touted as an crucial factor in the power in GAs because of its combinative nature and as one reason GAs may be superior to other search techniques for certain problems (or certain fitness landscapes). Much recent GA research has focused upon finding out for what class of problems GAs are more effective than Hill Climbing and Simulated Annealing. Work in GP which is equivalent to this is very important and we believe this experimentation to a contribution in this respect. It is seems critical that program discovery methods are carefully compared and well understood to avoid premature conclusions regarding superiority. The role of crossover needs to be understood so that it can be determined whether there exist cases where GP is more effective than mutation-based search.

Related work we have in progress is 1) a search for some statistical measures which would indicate the superior technique for a particular problem or the superior crossover operator to choose in GP and 2) finding out how the search techniques compare on larger program discovery problems, i.e. how well do they scale? Automatic definition of functions (ADF) [9] is an important new technique in GP which is claimed to improve performance on big problems. ADF is a technique of representation rather than an operator so it seems likely that HVL-mutate can be modified to support it. We want to find out whether SA or SIHC with the extended HVL-Mutate are sufficiently powerful to handle problems GP with ADF.

## References

1. Aarts, E., Korst, J.: Simulated Annealing and Boltzmann Machines. Wiley. 1989
2. Cramer, N. L.: A Representation for the Adaptive Generation of Simple Sequential Programs. Proc of Ist Intl Conf on Genetic Algorithms. Lawrence Erlbaum Assoc. 1985.
3. Friedberg, R.M.: A Learning Machine: Part 1. IBM Journal of Research and Development. 2(1): 2–13 (1958)
4. Friedberg, R.M., Dunham B., North J.H.: A Learning Machine: Part 2. IBM Journal of Research and Development. 3(3): 282–287 (1959)
5. Fujicki, C., Dickinson J.: Using the Genetic Algorithm to Generate LISP Source Code to Solve the Iterated Prisoner's Dilemma. Proc. of the 2nd Intl Conf on Genetic Algorithms. Lawrence Erlbaum Assoc. 1987.
6. Goldberg, D. E.: Genetic Algorithms in search, optimization, and machine learning. Addison-Wesley, 1989.
7. Holland, J. H.: Adaptation in Natural and Artificial Systems: An Introductory analysis with applications to biology, control, and artificial intelligence. 2nd Ed MIT Press,1992. (1st Ed 1978)
8. Koza, J. R.: Genetic Programming; On the Programming of Computers by Means of Natural Selection. Bradford Books, 1992.
9. Koza, J. R.: Genetic Programming II. Bradford Books, 1994.
10. Lenat, D. B: The Role of Heuristics in Learning by Discovery: Three Case Studies. Machine Learning, Eds. R. S. Michalski, J. G. Carbonell and T. M. Mitchell. Tioga Publishing Inc, 1983.
11. Sankoff, S., Kruskal, J.B., Editors: Time Warps, String Edits and Macromolecules: the Theory and Practice of Sequence Comparison. Addison Wesley, 1983.
12. Smith, S., J.: Flexible Learning of Problem Solving Heuristics Through Adaptive Search. Proc of the 8th Intl Joint Conf on Artificial Intelligence. Morgan Kaufmann, 1983.
13. Press, W. H.: Numerical Recipes in C: the art of scientific computing. Cambridge University Press, 1992.