Adaption of Operator Probabilities in Genetic Programming

Jens Niehaus and Wolfgang Banzhaf

System Analysis, Computer Science Department
University of Dortmund, D-44221 Dortmund, Germany
{niehaus,banzhaf}@ls11.cs.uni-dortmund.de

Abstract. In this work we tried to reduce the number of free parameters within Genetic Programming without reducing the quality of the results. We developed three new methods to adapt the probabilities, different genetic operators are applied with. Using two problems from the areas of symbolic regression and classification we showed that the results in these cases were better than randomly chosen parameter sets and could compete with parameter sets chosen with empirical knowledge.

1 Introduction

One of the characteristics of Genetic Programming (GP) [10,5] is the enormous number of free parameters of the algorithm. As different problems require different parameter sets GP requires a lot of experience and knowledge on side of the user.

In this work we are trying to reduce the number of free parameters. Our aim is to find adaptive methods that result in solutions that are as good as the ones gained with the traditional algorithm and empirically established parameter sets. Furthermore the solutions found with our new methods should be better than those found with the traditional algorithm and randomly chosen parameter sets.

While adaption of parameters is common in other areas like Evolution Strategies [13,9] and Genetic Algorithms [8,3,4,14] there are only very few attempts with GP [1,15].

When we later apply GP to a new problem it will be possible to use this algorithm without preceding parameter studies, which always require a lot of time. At least it will be possible to get an initial parameter set, which can be used for further tweaking.

Every new method incorporated into the GP paradigm has to be compared with the traditional method. Besides that a feature might work well on some problems and not on others, the methods might need different parameter sets. Therefore, if one method seems to be better than another one, the reason might be either due to an advanced method or it could be due to a difference in quality of the parameter sets used. Our adaptive methods rule out the second reason.

During a run the algorithm creates new individuals by copying old individuals and applying genetic operators. As there are several operators one of them has to be chosen. We examined three different methods to decide which of the operators is applied in certain situations.

In Section 2 we present the GP-system used including the implemented genetic operators. Section 3 describes the new adaptive methods and Section 4 includes our experiments regarding two different problem domains and the corresponding results. The results are discussed in Section 5.

2 GGP and Genetic Operators

2.1 Graph Representation

We used our own GP-system called *GGP*, which is capable of solving problems modelled as acyclic digraphs. Each node of a graph represents one operation such as *ADD* or *MUL* and has a certain number of inputs and outputs, and sometimes additional parameters. All outputs of each node have to be connected with exactly one input of another node.

Opposite to other systems we use graphs as internal representation instead of trees [11] or arrays [12]. This way it is easy to extend the system for using cyclic graphs without any restrictions.

A problem is modelled within the graph. The node type *Input* takes the input values of the problem and propagates them via the outgoing edge to the next node. The fitness function is part of the graph, too. The fitness value of an individual represented by a graph is the sum of all values propagated to the *Output* nodes of the graph over all scenarios.

The node type *GP*, referred to as GP-node, stands for a subgraph created by the GP-system during evolution. The number of incoming and outgoing edges of a GP-subgraph is variable and depends on the surrounding graph. Traditional implementations of GP would call only this node an *individual* and the surrounding graph would be part of the GP program code. The approach *GGP* uses is far more flexible:

- The calculation of the fitness value is part of the graph. If the user wishes to use a method other than, for example, squared differences he just has to modify the graph and does not have to modify the GP-program.
- One graph can contain more than one GP-node. For traditional GP this would mean that one individual consists of more than one tree. Problems dealing with more than one input value could use a tree for each input for some kind of preprocessing and the results could be combined in another tree. To achieve this kind of behaviour with a traditional GP-system the user would have to modify the GP programm code.

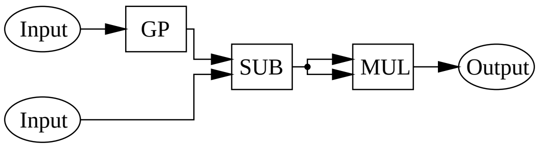

Fig. 1. A graph for symbolic regression

Figure 1 shows a graph used for symbolic-regression problems. For each scenario the first input takes the value from the function domain and the second input the value the function is supposed to calculate. The first value is used as input for the subgraph represented by the GP-node. The result of the calculation within the GP node is compared to the second input value. The fitness function is the sum of the squared differences over all scenarios.

In this work we allow only nodes with exactly one outgoing edge within the GP-subgraph. As the GP-node has only one *Output* node in our examples the contents of a GP-node can always be interpreted as a tree comparable to ordinary GP-systems.

During a run the number of nodes in a GP-subgraph within a GP-node is limited by a parameter the user has to choose at the beginning. The number of *Input* and *Output* nodes inside a GP-subgraph corresponds to the number of incoming and outgoing edges of the GP-node.

2.2 Genetic Operators

GGP includes one crossover operator and several mutation operators. Bear in mind that the operators modify only GP-subgraphs and not trees as in usual tree based GP. All the modifications result in acyclic graphs with all nodes of each GP-subgraph properly connected.

As each edge has one start and one ending the sum of all inputs of the nodes in a GP-subgraph must be the same as the number of outputs from these nodes making it impossible just to add or remove a node with a different number of inputs and outputs. Therefore most of the mutation operators modify subgraphs.

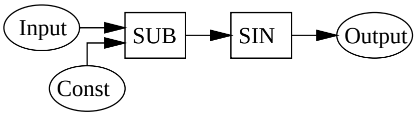

Fig. 2. An example graph

Figure 2 offers an example graph we use to explain the genetic operators described next.

1. We cannot remove the *Constant* node as it has no input but one output. The second input of the *SUB* node would not be connected afterwards. On the other hand it is possible to remove both the *Constant* node and the *SUB* node and to connect the *Input* node directly to the *Output* node. This mutation is called **delete path** as it removes the path between two nodes.
2. The opposite to **delete path** is the mutation operator **insert path.** As we are using only trees in this work the operation is reduced to the following: one edge of a GP-subgraph is split, a node with two inputs and one output is inserted, and the free input of this node gets connected to a newly inserted node with no incoming and one outgoing edge.[1]
3. The operator **insert node** inserts a node with one input and one output by splitting one existing edge.
4. As the name implies the operator **delete node** deletes a node with one input and one output by connecting the surrounding edges.
5. The operator **move node** moves a node with one input and one output to an adjacent position. For example the *SIN* node in Figure 2 could be moved to the position between *Input* and *SUB* node.
6. **Replace node** replaces a node with a different node with the same number of inputs and outputs. The *SUB* node in Figure 2 could be replaced by an *ADD* node.
7. **Subgraph crossover:** Given two individuals represented by two graphs with several GP-nodes each, one of the first individual's GP-subgraphs is replaced by the contents of the corresponding GP-node of the second individual. As our experiments use only models with one GP-node the operator is equivalent to replicating the first individual and we do not use it at all.

2.3 The Evolutionary Algorithm

GGP uses steady state tournament selection. There are always four individuals per tournament. The two winners overwrite the two losing individuals. Afterwards the new individuals will be mutated by one of the genetic operators. In the basic version of the algorithm the probabilities of the genetic operators are defined once at the beginning of a run. The sum of all probabilities must be smaller than or equal to one. As this work is about adapting these probabilities the next section will describe different methods to change the probabilities during runtime.

3 Different Methods of Adaption

We developed three new methods of adapting the probabilities of genetic operators and compared them to two static methods. Our aim was to find a method

[1] When we evolve graphs instead of trees the beginning of the new path can also be a node with two outputs and one input. The free input and output are inserted into another edge of the existing GP-subgraph. The operator also checks that the graph remains acyclic.

that is significantly better than randomly chosen static probabilities and not significantly worse than static probabilities that have empirically proved to result in good fitness values. We compare our results against the following methods without adaption:

Random Static Probabilities (RSP): Given n different genetic operators we compute an n-tuple $(p_1, ..., p_n)$ with $\sum_{i=1}^{n} p_i = 100$ and $p_1, ..., p_n \in \{0, 1, ..., 100\}$. Each of the $\frac{(n+99)!}{100!(n-1)!}$ possible tuples has the same probability of being chosen. The value of p_i represents the percentage of how often operator i is chosen for mutation.

Empirical Static Probabilities (ESP): During empirical tests some n-tuples have achieved significant better results than others. One of those was chosen.

Within all of the three following methods we calculate some values p_i, which represent the percentage of how often operator i is chosen for mutation. As the sum of all percentages might be smaller than 100 we introduce $p_0 = 100 - \sum_{i=1}^{n} p_i$ as the percentage for replication.

3.1 Population-Level Dynamic Probabilities (PDP)

The probabilities of the genetic operators are adapted based on the success rates of the operators.

- Twenty percent of all probabilities are shared equally distributed amongst all n genetic operators: $p_{all} = \lfloor \frac{20}{n} \rfloor$. This rule ensures that none of the probabilities can decrease to zero preventing the selection of the corresponding operator at a later time when its use is more suitable.
- To keep the next equation simple we introduce the ratio $r_i = \frac{success_i{}^2}{used_i}$ for each operator i where $used_i$ is the number how often operator i was applied and $success_i$ is the number how often these applications have lead to a fitness improvement compared to the parent individual.[2] [3]
- For each operator p_i is computed using p_{all} and a scaled value of r_i: $p_i = p_{all} + \lfloor r_i \frac{(100 - n p_{all})}{scale} \rfloor$ with $scale = \sum_{j=1}^{n} r_j$.

The name *'Population-level Dynamic Probabilities'* was chosen in reference to Angeline's categorisation of different classes for adaptive parameters. For an overview of other work on adaptive methods in Evolutionary Computation see [2].

[2] A squared value of success is used because a linear term always results in nearly equally distributed probabilities amongst the operators due to the unpleasant small success rate of genetic operators.

[3] To circumvent *division by zero*-errors $used_i$ is initialised with a value of one.

3.2 Fitness Based Dynamic Probabilities (FBDP)

Initial experiments have shown that different operators have different success rates depending on the fitness of their parent individual.

For each fitness improvement between a parent individual and the corresponding child the fitness of the parent and the operator used to achieve the improvement is stored in an array sorted by fitness values.

The probabilities are computed using the following method where twenty percent of all probabilities are again shared equally distributed amongst all n genetic operators: $p_{all} = \lfloor \frac{20}{n} \rfloor$.

– A probability of $p_{left} = 100 - \lfloor n p_{all} \rfloor$ must be distributed among n operators.
– We create a set of p_{left} $(fitness, operator)$-tuples from the array containing all the fitness improvements with the corresponding genetic operators. The set consists of those p_{left} pairs whose fitness value is next to the fitness of the actual parent.
– p_i is the sum of p_{all} and the number of tuples in the set that use operator i.
– If there have not been p_{left} fitness improvements yet p_i is calculated as $p_i = \lfloor \frac{100}{n} \rfloor$.

This method works only for operators using one parent. With GP systems using two parents for one genetic operator the fitness of the individual with the higher impact on the offspring should be chosen to create the set of tuples. If the impact is uncertain one of the two fitness values of the parents should be chosen at random.

3.3 Individual-Level Dynamic Probabilities (IDP)

Each individual j has its own parameter set for operator probabilities. For each genetic operator i there is a variable cnt_j^i counting the unsuccessful attempts to improve fitness with this operator. A counter is reset if an application of the corresponding operator leads to a fitness improvement. The relation among the values of all counters of one individual is used to calculate the probabilities of the genetic operators.

– Again twenty percent of all probabilities are shared equally distributed among all n genetic operators: $p_{all} = \lfloor \frac{20}{n} \rfloor$.
– p_i is calculated using equation (1). The more often an operator has failed the smaller is p_i.

$$p_i = p_{all} + \left\lfloor \frac{(\max_{1 \le k \le n} cnt_j^k + 1 - cnt_j^i)(100 - n\, p_{all})}{n(\max_{1 \le k \le n} cnt_j^k + 1) - \sum_{k=1}^{n} cnt_j^k)} \right\rfloor \tag{1}$$

– After a tournament is finished, a winning individual j is copied to the position of a losing individual k, and one of the operators is applied (based on p_i-values of the parent individual). Then the counter of the applied operator of the parent individual is updated. Afterwards all the counters of the parent j are propagated to the child individual k using equation (2).

$$cnt_k^i = \frac{1}{2}\left(cnt_j^i + \frac{1}{n}\sum_{l=1}^{n} cnt_j^l\right) \tag{2}$$

With equation (2) individuals can more easily adapt to new situations during a run. For example in a situation with one individual having a good fitness and one preferred operator, this individual might be the ancestor for a chain of descendants with slightly degrading fitness values. These individuals where all mutated with the same operator, which now seems to be a bad choice. Because of to equation (2), the difference between the probability of this operator and all the others will be reduced after each mutation so that the use of a different operator gets more probable soon.

For operators using two parents this method has to be extended. One way would be to propagate the smaller one of both values cnt_k^i from both of the parents but further research is needed on this topic.

4 Experiments and Results

This section studies the performance of the different methods introduced in Section 3 using a symbolic-regression and a classification problem. In both cases we used a population size of 100 individuals and a maximum number of 100,000 tournaments. With each problem and each method of adaption we tried four different values for the upper limit of nodes inside a GP-subgraph (40, 80, 100 and 140). For each of all possible combinations 60 runs were performed. As we only want to examine the influence of our methods on the solution we did not use advanced techniques such as ADFs or demes.

4.1 Symbolic Regression

We used GP to find a function $f : [0, 2\pi] \rightarrow \mathbb{R}$ that minimises the expression

$$\sum_{i=0}^{49}\left(f(\frac{6.3}{49}i) - \sin(\frac{6.3}{49}i)\right)^2 . \tag{3}$$

In other words we tried to evolve a sinus-function based on a training set of 50 equidistant points between 0 and 6.3. We used the graph shown in Figure 1. Table 1 lists all node types allowed in a GP-subgraph. The node type *Factor* scales the input by the value of its parameter. The output of the node type

Table 1. Nodes in GP-subgraph

Type	Inputs	Outputs	Params
ADD	2	1	0
SUB	2	1	0
MUL	2	1	0
DIV	2	1	0
Factor	1	1	1: N(0,4)
Const	0	1	1: N(1,4)
Inputval	0	1	0

Table 2. Average fitness of 60 runs

Nodes	RSP	ESP	PDP	FBDP	IDP
40	0.055	0.052	0.052	0.046	**0.045**
80	0.033	0.028	0.031	0.022	**0.022**
100	0.025	0.019	**0.012**	0.017	0.02
140	0.029	**0.01**	0.02	0.023	0.016

Inputval always has the same value as the *Input* node of the GP-subgraph. The GP-system uses the operators **insert path, delete path, replace node** and **delete node** as described in Section 2.2.

Table 2 shows the average fitness of 60 runs for each parameter set. All dynamic methods are in average better than RSP. With the exception of the experiments with GP-subgraphs with a maximum size of 140 nodes there were always two dynamic methods better than ESP.

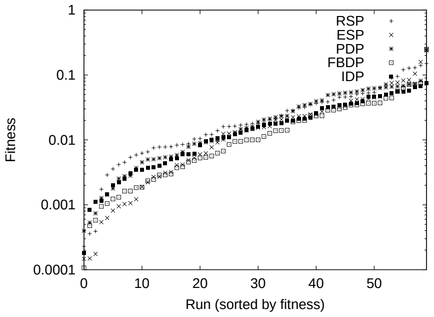

Fig. 3. Problem: Sinus, GP-subgraph with 80 nodes

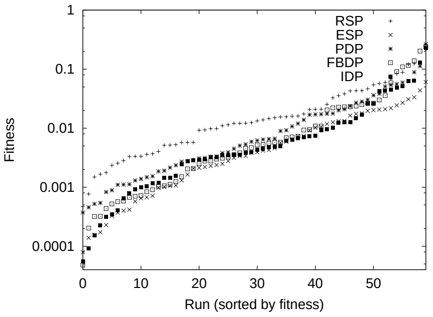

Fig. 4. Problem: Sinus GP-subgraph with 140 nodes

Figures 3 and 4 show the results of all runs with a maximum of 80 and 140 nodes inside the GP-subgraph. Due to the similarity to Figure 3 the figures for 40 and 100 nodes are omitted.

For each GP-subgraph-size we did a Kolmogorov-Smirnov test [7] to find out whether the distributions of the fitness results for the alternative methods might differ. As the plots propose this is only true in a very few cases. With a confidence-level of 95 percent only the results of RSP seem to differ from all other methods in the 140 nodes case. For all other cases no hypothesis can be accepted or discarded.

4.2 Classification

The classification problem we used is shown in Figure 6. The data set consists of 1000 points and was taken from [6]. The corresponding graph is shown in Figure 5. The fitness was the error rate of an individual given in percent.

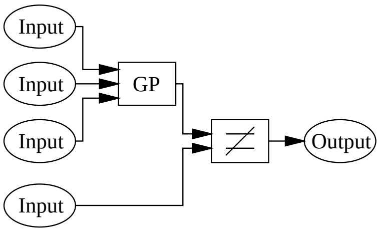

Fig. 5. Graph for classification

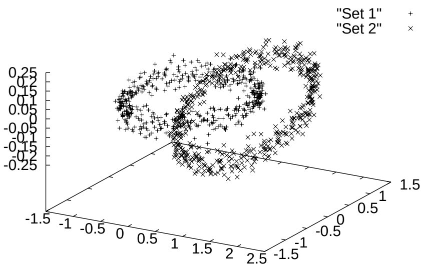

Fig. 6. The classification problem

Additional to the node types in Table 1 types called *SIN, COS* and *Threshold* were used, where *Threshold* has one parameter and sets the output to 1 if the parameter is smaller than the input value or -1 otherwise. Instead of *Inputval* we used the node type *Inputval2*. It uses a parameter to decide which input value of the GP-subgraph should be propagated to the output of the node.

The results are given in Table 3. Figures 7 and 8 show the results of the individual runs. The runs with 40 and 140 nodes within the GP-subgraph look similar to Figure 8 and are omitted.

Table 3. Average fitness of 60 runs

Nodes	RSP	ESP	PDP	FBDP	IDP
40	0.124	0.079	0.106	0.104	0.094
80	0.11	0.087	0.111	0.102	0.096
100	0.139	0.080	0.123	0.112	0.105
140	0.127	0.102	0.123	0.128	0.105

Table 4. Significant differences

Nodes	Methods
40, 80, 100:	RSP $\leftrightarrow$ ESP
40, 100, 140:	RSP $\leftrightarrow$ IDP
80, 100:	ESP $\leftrightarrow$ PDP
100:	ESP $\leftrightarrow$ FBDP

In both figures we see that the results of the adaptive methods lie between BSP and ESP. Of all the adaptive methods IDP seems to be most suited for this problem.

Kolmogorov-Smirnov tests offer some indicators that both ESP and IDP perform better than RSP. Table 4 lists the methods with different distributions at a confidence level of 95 percent.

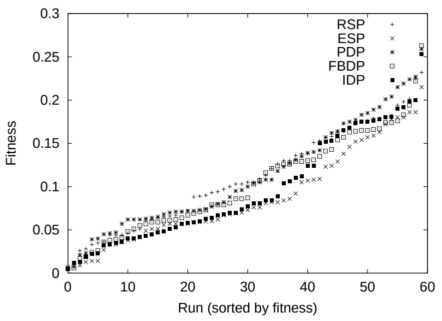

Fig. 7. Problem: Classification, GP-subgraph with 80 nodes

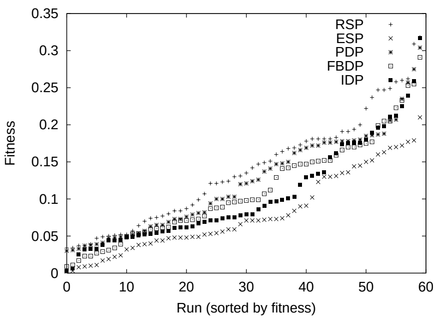

Fig. 8. Problem: Classification, GP-subgraph with 100 nodes

5 Discussion

The results given in Section 4 point out that the new methods seem to fulfil our aims.

Looking at the average fitness values the adaptive methods seem to be better than randomly chosen static parameter sets on both problems and even better than the empirically chosen parameter sets on the symbolic-regression task.

Kolmogorov-Smirnov tests show that our methods seem to be better than the randomly chosen parameter sets. As the distribution functions in both benchmarks chosen are not continuous the tests have to be taken with care because Kolmogorov-Smirnov tends to accept hypotheses for too long.

IDP seems to be the best of the three methods for adapting probabilities, at least for the benchmarks used. The reason for the differences might be explained as follows.

– PDP uses the same probabilities for all individuals. Those probabilities are based on all fitness improvements. Most of the successful mutations improve only the fitness of an individual relative to its parent but do not result in a new best fitness for the hole population. So the adaption might lead to a parameter set useful for improving fitness of average individuals without improving the fitness of the population.
– FBDP chooses probabilities based on the fitness of the parent. Therefore mutation of individuals with a good fitness does not interfere with mutation of those individuals with a worse fitness.
  The test results in Tables 2 and 3 show that the performance of FBDP decreases for large GP-subgraphs. The reason for this might be the following: On small GP-subgraphs individuals with a similar fitness tend to have a similar genotype (e.g. graph). As most operators can only be applied to certain positions in a graph it is quite possible that the mutation performed

is quite similar to those successful operations responsible for choosing this operator thus resulting in a fitness improvement. On bigger GP-subgraphs the chance to do a mutation completely unrelated to those responsible for choosing the probabilities grows, making a fitness improvement less probable.
- With IDP every individual has its own history of successful and unsuccessful mutation attempts. If an operator does not work for a certain individual it will not be used with it that much for later mutations. Other individuals, which might benefit from this operator, are not affected by this restriction.

6 Conclusion

In this contribution we have successfully applied new methods of parameter adaption to GP. In further studies we have to increase the number of runs to make the results statistically more significant, we have to validate the results using other problems and find methods to adapt other parameters such as population or GP-subgraph size.

References

1. P. J. Angeline: Adaptive and Self-adaptive Evolutionary Computations. In: M. Palaniswami, Y. Attikiouzel, R. Marks, D. Fogel and T. Fukuda (Eds.) *Computational Intelligence: A Dynamic System Perspective,* NJ: IEEE Press, 152–161, 1995
2. P. J. Angeline: Two Self-Adaptive Crossover Operators for Genetic Programming. In: P. Angeline, K. Kinnear (Eds.) *Advances in Genetic Programming II.* Cambridge, MA: MIT Press, 89–110, 1996
3. J. Arabas, Z. Michalewicz and J. Mulawka: GAVaPS – a Genetic Algorithm with Varying Population Size. In: *Procceedings of the First IEEE Conference on Evolutionary Computation.* Orlando, Florida: IEEE Press. 73–78, 1994
4. T. Bäck, A. E. Eiben and N. A. L. van der Vaart: An empirical study on GAs "without parameters". In: *Parallel Problem Solving from Nature - PPSN VI.* Berlin: Springer-Verlag, 315–324, 2000
5. Banzhaf, W., Nordin, P., Keller, R. E., Francone, F. D.: Genetic Programming: An Introduction. San Francisco, CA: Morgan Kaufmann, 1998
6. Brameier, M. and Banzhaf, W: Evolving Teams of Predictors with Genetic Programming. Technical Report, University of Dortmund, Computational Intelligence, Collaborative Research Center 531, 2001. To appear
7. W. J. Conover: Practical nonparametric statistics. New York: John Wiley & Sons, 309–314, 1971
8. L. Davis: Adapting Operator Probabilities in Genetic Algorithms. In: J. D. Schaffer (Ed.) *Procceedings of the Third International Conference on Genetic Algorithms and Their Applications.* San Mateo, CA: Morgan Kaufmann, 61–69, 1989
9. N. Hansen and A. Ostermeier: Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In: *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation.* 312–317, 1996

10. J. R. Koza: Genetic Programming: On the Programming of Computers by Natural Selection. Cambridge, MA: MIT Press, 1992
11. P. Marenbach and H. Pohlheim: Generation of Structured Process Models Using Genetic Programming. In: Fogarty, T. C. (Ed.) *Evolutionary Computing. Selected Papers,* volume 1143 of *Lecture Notes in Computer Science,* Berlin: Springer Verlag, 102–109, 1996
12. R. Poli: Some Steps Towards a Form of Parallel Distributed Genetic Programming. In: *The 1st Online Workshop on Soft Computing (WSC1).* `http://www.bioele.nuee.nagoya-u.ac.jp/wsc1/`, 1996
13. H.-P. Schwefel: Evolution and Optimum Seeking. New York: John Wiley & Sons, Inc., 1995
14. W. M. Spears: Adapting Crossover in Evolutionary Algorithms. In: R. Reynolds and D. B. Fogel (Eds.) *Procceedings of the Fourth Annual Conference on Evolutionary Programming.* MIT Press, 367-384, 1995
15. A. Teller and M. Veloso: PADO: A new learning architecture for object recognition. In: *Symbolic Visual Learning.* Oxford University Press, 81-116, 1996