Learning What to Ignore: Memetic Climbing in Topology and Weight space

Julian Togelius, Faustino Gomez and Jürgen Schmidhuber
Dalle Molle Institute for Artificial Intelligence (IDSIA)
Galleria 2, 6298 Manno-Lugano
Switzerland
{julian, tino, juergen}@idsia.ch

***Abstract*— We present the memetic climber, a simple search algorithm that learns topology and weights of neural networks on different time scales. When applied to the problem of learning control for a simulated racing task with carefully selected inputs to the neural network, the memetic climber outperforms a standard hill-climber. When inputs to the network are less carefully selected, the difference is drastic. We also present two variations of the memetic climber and discuss the generalization of the underlying principle to population-based neuroevolution algorithms.**

Keywords: neuroevolution, reinforcement learning, network topology, memetic algorithms

I. Introduction

Neuroevolution, or the training of neural networks using evolutionary algorithms, is conceptually simple, has very broad applicability, and has been shown to outperform classical reinforcement learning algorithms on difficult benchmark problems [1], [2]. Most neural networks can be defined by their topology (the set of neurons and connections between them) and connection weights, and the genetic crossover and mutation operators can in principle be applied to both.

Although the majority of neuroevolutionary algorithms stick to a fixed topology and evolve only the weights (see e.g. references 26–112 in [1]) as they represent a relatively smaller search space, a large body of work suggests that the topology of a neural network interacts in a nontrivial way with its evolvability, i.e. the ability of evolutionary algorithms to find weight settings for networks that produce a desired behavior or approximate a given function. It is not as easy as getting the size of the network right: two networks with the same number of neurons and connections can have drastically different evolvability for a given problem. Specifically, a fully connected network (e.g. a standard MLP) can often be made *more* evolvable by simply removing a few key connections.

The simplest hypothesis that explains this phenomenon is that the availability of certain information at certain points in the network leads evolution into local optima. Calabretta et al. call this effect *neural interference*. For those of us who wish to use neuroevolution for learning control (e.g. for game agents or robots) with minimal human domain knowledge, this poses a problem. We would rather leave it to the learning algorithm to decide which inputs to use and which to ignore. Not knowing how to deal with this effect may explain, in part, the lack of published successful results for evolving agents that use high-dimensional input data, e.g. vision. Almost all published papers in evolutionary robotics and game playing use networks with few (on the order of 10 to 20) inputs (with the notable exception of [3]).

Realizing the importance of topology, many researchers have devised algorithms that evolve topology and weights of networks in tandem. Some of the more well-known efforts include those of Gruau (Cellular Encoding; [4]) and Stanley (NEAT; [5]). These algorithms can create networks with very unusual topologies that perform significantly better than fully-connected topologies with evolved weights, at least on specific tasks.

There is a problem with evolving the topology, though: changing the topology of a network is almost always very disruptive. Critical links can be disabled, and enabling previously inoperative links can be equally destructive. If the network has a reasonably high fitness, it will often drop immediately to near the level of a random network after this kind of structural modification, so simply applying a mutation operator that changes topology as well as weights is unlikely to work very well.

In this paper, we explore a simple way of addressing this issue by evolving the structure and weights of a neural network at different time scales, where "global" search in topology space is interleaved with "local" search in weight space. In other words, after changing a topology, try to find a good weight combination for a little while before deciding whether to keep the new topology or revert to the old one. The hope is that this scheme will yield a family of *memetic algorithms* [6], that might initially learn more slowly than methods that search for weights only, but ultimately reach higher fitness by avoiding topology-induced local optima. The authors are not aware of any previous applications of memetic algorithms to evolving neural networks.

The next section, provides some additional background on the interaction between evolvability and network topology. In section III, we present five algorithms that are compared experimentally in a race car control task in section IV. Section V discusses our results and future directions, and section VI summarizes our findings.

II. Related Work

Several studies have demonstrated that often increasing evolvability is simply matter of removing a single neural connection (e.g. Nolfi was able to evolve better robot localization by applying such a minimal lesion to a recurrent network [7]), or removing a certain input, as shown by Lucas and Togelius' work in evolving waypoint-following behavior for a holonomic agent in simulation [8]. In that study, the authors found that effective controllers would only evolve when a specific input representing the angle between the agent's direction of movement and the direction to the waypoint was *absent*.

Restricting network topology can also encourage modularity to evolve for tasks where multiple, relatively orthogonal functional competencies are required. Calabretta et al. attempted to evolve networks to perform two different image processing tasks, resembling the "what?" and "where?" tasks in human neurobiology, simultaneously. They found that for this to work, the two tasks needed to be performed by largely separate networks, otherwise networks would evolve that could solve only one of the tasks [9]. Similarly, De Nardi et al. found that to evolve successful helicopter control it was crucial to enforce some modularity—it was necessary to keep the yaw stabilization module separate from the networks that controlled other aspects of the helicopter's flight [10]. If a network with access to all inputs was able to control the yaw of the helicopter, it quickly learned to keep hovering while spinning, which is a local optimum as goal-directed flight requires stable yaw. Keeping the yaw stabilization module separate prevented evolution from taking the easy way out.

Of course is it not always possible to identify the best connectivity experimentally, and searching topology space can be problematic since mutations that affect the connectivity of a network can often be very disruptive. There has been some work addressing this problem, most notably the NEAT algorithm [5], through a mechanism called *innovation protection*. Whenever a network with a sufficiently different new topology is created, it is assigned its own "species", and it or its offspring (with the same topology but different weights) are allowed to stay in the population for a few generations, even if its fitness is much below that of the best networks in the population. If, at the end of this "grace period", weight settings have been found that give networks of this topology a fitness among the best in the population, it can stay on, otherwise the topology is removed. This feature is similar to what we are proposing here, though NEAT is a considerably more complex algorithm.

III. Methods

This section describes the neural network representation, the five search algorithms, and test domain used in the experiments in section IV.

A. Masked neural networks

In all of the experiments, solutions are represented by a Multi-Layered Perceptron (i.e. a feedforward network) with one hidden layer, where each neural connection has an associated boolean variable that determines whether it is on or off. The network as a whole is thus defined by n real numbers denoting connection weights and n booleans denoting which connection weights are active (i.e. the "mask"). When an input vector is propagated through the network, only those connections whose mask bit is set are used to compute the network output. The search algorithms operate on the mask networks via two mutation operators: *weight mutation* and *topology mutation*. Weight mutation adds values drawn from a Gaussian distribution with mean 0 and standard deviation 0.1 to all connection weights; this includes weights which are currently marked as off. Topology mutation consists in considering each bit in the mask, and flipping that bit with probability 0.05. In the pseudocode for the algorithms presented below, weight and topology mutation are invoked by the WeightMutate() and TopologyMutate() functions, respectively.

In all experiments, all connection weights were initialized to 0; the initialization of the mask varied for each experiment, as described below.

Algorithm 1: Hill-Climber (n)

```
1  INITIALIZE (champion)
2  f_champ ← EVALUATE (champion)
3  for i=1 to n do
4      contender ← champion
5      WEIGHTMUTATE (contender)
6      f_cntder ← EVALUATE (contender)
7      if f_cntder >= f_champ then
8          champion ← contender
9      end
10 end
```

B. Algorithms

Algorithm 1: Hill-Climber
This algorithm is equivalent to a $(1+1)$ evolution strategy which has a population of one individual (the *champion*), and each update (generation), it evaluates the fitness of the champion, generates a new individual (the *contender*) by copying the champion and mutating the copy, and evaluates the fitness of the contender. If the fitness of the contender is higher than or equal to that of the champion, the champion is replaced by the contender, otherwise the contender is discarded and the champion remains.

Algorithm 2: Simultaneous Climber
This algorithm is the same as Algorithm 1, except that on each iteration the topology is also mutated, not just the weights. Arguably, this constitutes the simplest possible topology-evolving algorithm. However, given that topology mutations are typically destructive, this algorithm is not be expected to work very well; the probability of a beneficial weight mutation and a beneficial topology mutation co-

Algorithm 2: Simultaneous Hill-Climber (n)

```
1  INITIALIZE (champion)
2  f_champ ← EVALUATE (champion)
3  for i=1 to n do
4      contender ← champion
5      WEIGHTMUTATE (contender)
6      TOPOLOGYMUTATE (contender)
7      f_cntder ← EVALUATE (contender)
8      if f_cntder >= f_champ then
9          champion ← contender
10     end
11 end
```

Algorithm 3: Memetic Climber (n,m)

```
1  INITIALIZE (champion)
2  f_champ ← EVALUATE (champion)
3  for i=1 to n do
4      contender ← champion
5      TOPOLOGYMUTATE (contender)
6      for j=1 to m do
7          f_cntder ← EVALUATE (contender)
8          subcontender ← contender
9          WEIGHTMUTATE (subcontender)
10         f_subcnt ← EVALUATE (subcontender)
11         if f_subcnt >= f_cntdr then
12             contender ← subcontender
13         end
14     end
15     f_cntder ← EVALUATE (contender)
16     if f_cntder >= f_champ then
17         champion ← contender
18     end
19 end
```

occurring is simply too low.

Algorithm 3: Memetic Climber
This is the memetic version of the hill-climber. Each generation, a contender is generated by copying the champion and applying topology mutation, which typically causes a large drop in fitness. Algorithm 1 is then applied for m iterations (lines 6–14) in order to find better weights for the mutated topology.

Algorithm 4: Constrained Climber
Algorithm 3 puts no restrictions on topology mutation, and thus on how many connections can be on at a particular point in time. This means that the topologies are not searched in any particular order, or with any bias for a particular network size. However, there are at least two orthogonal reasons for ordering solution candidates such that simple ones are considered first: (1) testing simple candidates tends to consume less computation, (2) Occam's Razor suggests that small networks tend to generalize better. The constrained

Algorithm 4: Constrained Memetic Climber (n,m,p, k)

```
1  INITIALIZE (champion)
2  f_champ ← EVALUATE (champion)
3  for i=1 to n do
4      contender ← champion
5      TOPOLOGYMUTATE (contender)
6      PRUNECONNECTIONS (contender, p)
7      for j=1 to m do
8          f_cntder ← EVALUATE (contender)
9          subcontender ← contender
10         WEIGHTMUTATE (subcontender)
11         f_subcnt ← EVALUATE (subcontender)
12         if f_subcnt >= f_cntdr then
13             contender ← subcontender
14         end
15     end
16     f_cntder ← EVALUATE (contender)
17     if f_cntder >= f_champ then
18         champion ← contender
19     end
20     if i >k then
21         p ← 2 * p
22         k ← 2 * k
23     end
24 end
```

climber uses a principled scheme for incrementally allocating the total search time borrowed from universal program search methods [11], [12]: spend twice the time on programs of size $2l$ that is spent on programs of size l. This algorithm starts by searching for weights for a topologies of an initial "size" specified by the parameter p: the probability that a connection is active. After k generations both k and p are doubled (lines 20–23) so that topologies with, on average, twice the number of connections are searched for twice as many generations. The network size limit is enforced after every topology mutation by the PRUNECONNECTIONS() function (line 6) which randomly switches connections off until only as many connections as allowed are active.

Algorithm 5: Inverse Climber
This is the same as Algorithm 3 (memetic climber) except that the two types of mutations are swapped. Each generation, the algorithm makes one weight mutation and then searches topology space for m steps, in order to find a good mask for that particular configuration of weights.

C. The Race Car test domain

The five algorithms were tested in the "simplerace" simulated car racing domain, previously used for the 2007 IEEE CEC car racing competition[1]. A complete description can be found in [13]; source code is available on the car racing competition web page. The objective of this task is to drive a

[1]http://julian.togelius.com/cec2007competition

Algorithm 5: Inverse Memetic Climber (n,m)

```
1  INITIALIZE (champion)
2  f_champ ← EVALUATE (champion)
3  for i=1 to n do
4      contender ← champion
5      WEIGHTMUTATE (contender)
6      for j=1 to n do
7          f_cntder ← EVALUATE (contender)
8          subcontender ← contender
9          TOPOLOGYMUTATE (subcontender)
10         f_subcnt ← EVALUATE (subcontender)
11         if f_subcnt >= f_cntdr then
12             contender ← subcontender
13         end
14     end
15     f_cntder ← EVALUATE (contender)
16     if f_cntder >= f_champ then
17         champion ← contender
18     end
19 end
```

car through as many waypoints as possible from a randomly generated sequence in a continuous environment with simple physics. Fitness is defined as the number of waypoints passed in 500 time steps, averaged over several trials.

We use the "competition" version of the task, where a car is evaluated using three different scenarios: (1) on its own, (2) against an opponent employing a speed-limited greedy strategy (going straight for the current way point), and (3) against an opponent employing a more sophisticated strategy that selects which way point to aim for based on which car is closest to the current way point. When racing against an opponent, the task gains a strategic component: choosing the best next waypoint can require predicting which one the opponent is heading for. The fitness of a controller is calculated as the average number of way points passed over ten trials in each of the three scenarios. This version of the task was used for the initial ranking of competitors in the CEC competition, meaning that there are plenty of good controllers with which to compare our results.

Depending on the experiment, we use one of two different sets of inputs: standard or extended. The standard set of inputs is a vector of eight real values:

INPUT 1: constant bias term
INPUT 2: speed of the car
INPUT 3: angle to the current way point
INPUT 4: distance to the current way point
INPUT 5: angle and
INPUT 6: distance to the next waypoint
INPUT 7: angle and
INPUT 8: distance to the other vehicle (if present)

These inputs are chosen based on the authors' considerable familiarity with the domain, and provide sufficient information for neural networks to solve the task competently. Note that the standard inputs have a "first-person perspective": all of the values could be obtained from sensors actually mounted on the car.

The extended set of inputs consists of the 17 real values; the standard inputs plus:

INPUT 9: orientation of the car (in Cartesian space)
INPUT 10: angular velocity of the car
INPUT 11: speed of the other vehicle
INPUT 12: x-coordinate of car position
INPUT 13: y-coordinate of car position
INPUT 14: x-coordinate of opponent car position
INPUT 15: y-coordinate of opponent car position
INPUT 16: current way point in Cartesian coordinates
INPUT 17: orientation of the car (static reference frame)

These extra inputs provide valuable information about the state of the system that complements the standard inputs, and could potentially be used to construct a better performing controller than would be possible using only the standard inputs. However, because most of the extra information cannot be easily gathered from sensors present on the actual car (i.e. have a "third-person perspective") more computation would be needed to make effective use of this information. Still, one would expect a competent learning algorithm to disregard information it cannot handle, and start by using those inputs that can easily be exploited.

The two outputs of the network are always interpreted as the steering and driving command of the car; in both configurations the network has a hidden layer of six neurons, making for a total of 60 and 114 neural connections (and thus the same number of bits in the mask) for the standard and extended inputs, respectively.

IV. EXPERIMENTS

In this section, we describe a series of experiments performed with the hill-climber and variations of the memetic climber. Note that only single-search-point (i.e. ontogenetic) search algorithms will be explored here; section V will discuss extensions of the core technique to population-based search, e.g. genetic algorithms.

In every experimental run, a total of 20000 networks were evaluated. For the hill-climber (algorithm 1) and simultaneous hill-climber (algorithm 2), this means running the algorithm for $n = 20000$ generations; for the memetic climbers (algorithms 3, 4, and 5), the number of local search steps m was set to 50, for a total of $n = 20000/50 = 400$ generations. For the *constrained hill climber* the initial size-probability p was set to 0.05, and the number of generations to search this initial size k was set to 4.

Each experiment was repeated 50 times, and the graphs show the best fitness and standard deviation per generation averaged over all 50 runs.

A. Hill-climbing in weight space

In order to investigate the effect of different levels of connectivity in the network, the hill-climber was not only run with fully connected networks, but also on networks with

random connectivity, i.e. where not all of the bits in the mask are set. For the runs with less than full connectivity, a new mask is randomly generated for each run, with a probability p of each bit being set.

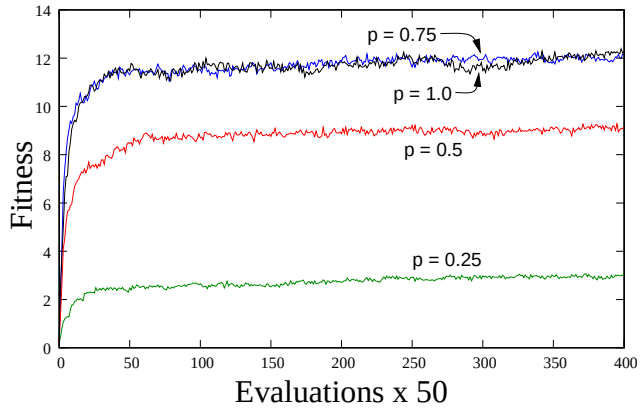

Fig. 1. **Hill-climber with standard inputs**. Each curve shows the average of 50 runs for a different proportion of connections in the network switched on. Each tick on the x-axis represents 50 generations.

Figure 1 shows the performance of the simple hill-climber for four different mask probabilities (1.0, 0.75, 0.5 and 0.25), using the standard set of inputs. The simple hill-climber performs well on this task when searching the space of weights for fully connected networks using a hand-picked set of inputs. The value at which the fitness levels off is just below the lowest fitnesses in the league table for the CEC car racing competition, and indicates well-tuned, though probably not tactical, driving.

In general, the more connections are turned on in the mask, the better solutions the algorithm is able to find. So simply turning off random connections in the hope of improving evolvability is, unsurprisingly, not a good idea.

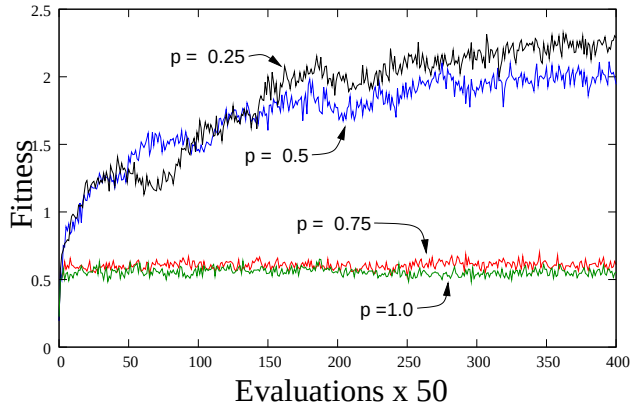

Fig. 2. **Hill-climber with extended inputs**. Each curve shows the average of 50 runs for a different proportion of connections in the network switched on. Each tick on the x-axis represents 50 generations.

In figure 2, the same four hill-climber configurations are plotted working in the space of the larger networks that make use of the extended inputs. Here, we see a radically different picture. None of the configurations manage to find good weights (a fitness of around 3.0 is only marginally better than random driving; compare to fitness in figure 1), though those with fewer connections ($p = 0.25$ and $p = 0.5$) reached markedly higher fitness than those with most of the connections switched on. Apparently, the extra inputs cause the hill-climbers to get stuck in local optima very early on, before any sensible behavior has evolved. Furthermore, the final fitness of the runs with low p had a standard deviation that was about as high as the fitness itself (3.2 and 2.3, respectively), meaning that randomly removing connections might in some cases lead to a topology that allows for reasonably good fitness to evolve, but might just as well lead to one where evolvability is virtually zero.

B. Simultaneous hill-climbing in weight and topology space

As expected, this algorithm does not perform well for either version of the task. In fact, no significant fitness growth was seen over many runs of this algorithm. We omit the graph.

C. Memetic climbing in weight and topology space

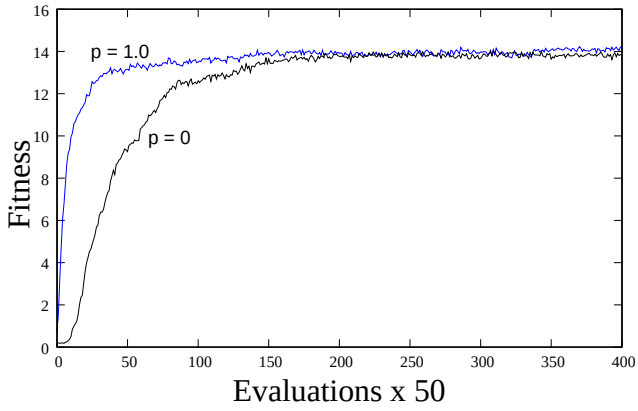

Fig. 3. **Memetic climbers with standard inputs**, starting with either all connections switched on or all switched off. Each tick on the x-axis represents one topology mutation and 50 steps of hill-climbing in weight space. Each curve is the average of 50 runs.

Figure 3 shows the progress of the memetic climber for networks with standard inputs. We tested two different ways of initializing the runs: having all connections switched off in the mask, and having all connections switched on. As is apparent from the graphs, the memetic climber works well under both conditions. The only significant difference is that fitness grows more slowly when connections are initially switched off.

In figure 4, we plot the performance of the memetic climbers for networks with extended inputs, again activating either all or none connections at the start of each run. It is immediately clear that there is a qualitative difference between the performance of the memetic climber and that of the hill-climber with this larger set of inputs; the memetic climber reaches an order of magnitude higher fitness. The ongoing search in topology space helps to avoid local minima when

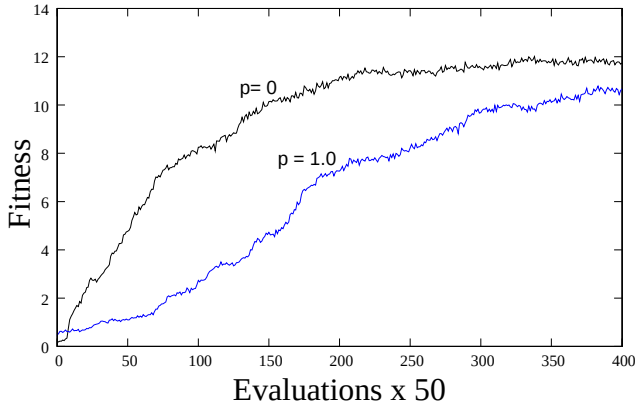

Fig. 4. **Memetic climbers with extended input**, starting with either all connections on or all connections switched off. Each tick on the x-axis represents one topology mutation and 50 steps of hill-climbing in weight space. Each curve is the average of 50 runs.

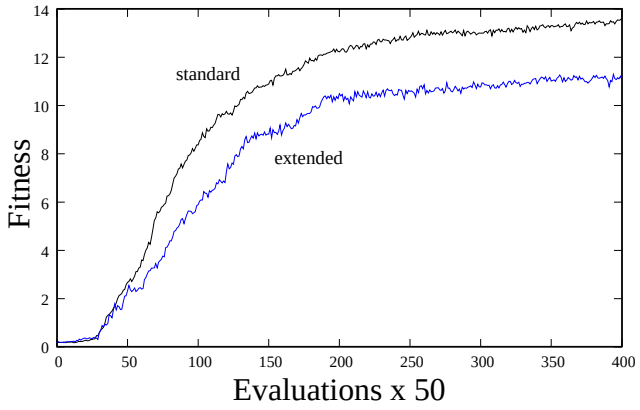

Fig. 5. **Constrained growth memetic climbers** with both standard and extended inputs. Each curve is the average of 50 runs.

searching in weight space. It is also clear that initializing a run with all connections active results in slower learning than initializing it with all connections off.

What the graphs do not show is the number of connections active at the end of each run, or how these connections are distributed. It turns out that regardless of the input set used or the number of connections turned on initially, about half of the connections are turned on at the end of a successful run. So for the standard inputs with $p = 0.0$, on average 31.0 (s.d. 3.4) connections are turned on at the end of the run, and for $p = 1.0$, 31.0 (3.71) connections are on, out of a total of 60. Using extended inputs and networks with 114 connections, runs that start with $p = 0.0$ finish with 55.0 (5.6) connections on, and starting with $p = 1.0$ finish with 54.6 (5.0) active connections.

Looking at the distribution of connections switched on in the masks of the evolved networks, it is hard to see a clear pattern. The probability is 0.51 that any given outgoing connection from any of the first 8 input neurons (corresponding to the standard inputs) will be switched on. This probability drops to 0.45 for the 9 input neurons that handle the extra inputs for the extended input set, a smaller difference than we expected. No individual input neuron has a much lower probability of having outgoing connections than any other. Therefore, it is not the case that evolution simply decides to turn off certain inputs. It is however possible that certain inputs are more unanimously turned off at earlier stages of the search process, something we have not investigated.

D. Memetic climbing with constrained network growth

Figure 5 plots the performance of the constrained memetic climber on networks with standard and extended inputs. The algorithm works well in both cases; however, final fitness is on average slightly higher for networks with standard inputs than those with extended inputs. One notable difference compared to the standard memetic climber is that the constrained memetic climber learns more slowly, i.e. takes longer time to reach the same fitness. Another difference is that the constrained memetic climber learns sparser networks than the memetic climber. With standard inputs, the networks of the final generation had on average 22.3 (s.d. 1.1) active connections, and the extended input networks average 44.4 (0.9) active connections. For networks with extended input, the connections from the first 8 input neurons had probability 0.33 of being switched on, and the corresponding probability for the 9 other input neurons is 0.28. Further study is needed to determine whether these simpler networks yield better generalization.

E. Inverse memetic climbing

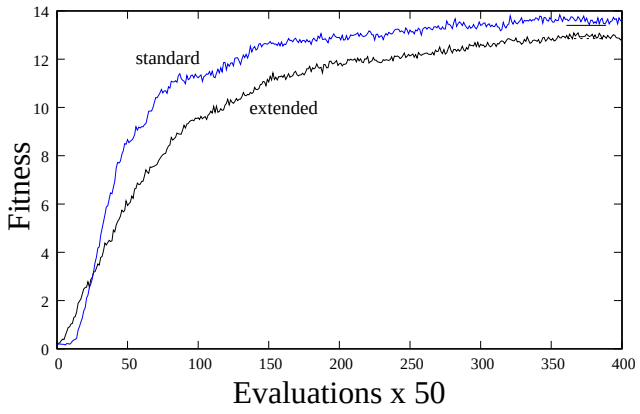

Fig. 6. **Inverse memetic climbers** with both standard and extended input. Each curve is the average of 50 runs.

Figure 6 shows the performance of the inverse climber on both standard and extended input networks. Just like the other two memetic climbers, this algorithm manages to reach high fitness in both conditions, and it reaches slightly higher fitness using the standard inputs compared to the extended inputs. The resulting networks are somewhat smaller than those produced by the standard memetic climber, but somewhat larger than those produced by the constrained memetic climber: 27.7 (4.2) for the standard inputs, and 50.8 (5.0) for the extended inputs. For networks with extended inputs,

connections from the first 8 input neurons had probability 0.48 and from other inputs had probability 0.42.

F. Comparison of different climbers

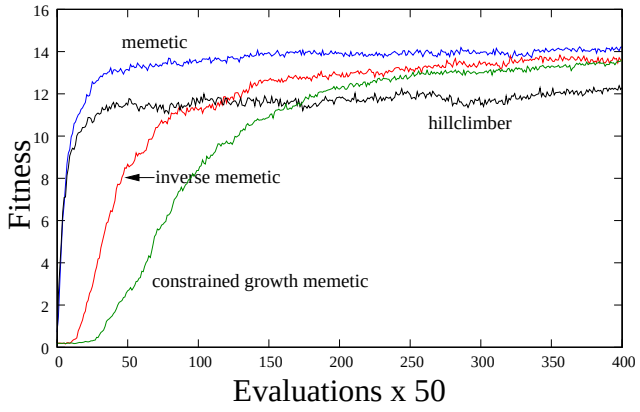

Fig. 7. **Algorithm comparison for standard inputs**: hill-climber ($p = 1.0$), memetic climber ($p = 1.0$), inverse memetic climber ($p = 0.0$) and constrained growth memetic climber ($p = 0.0$). Each curve is the average of 50 runs.

In figure 7 we plot the fitness growth all the climbers (except the simultaneous climber), using the best parameter setting found (for the case where more than one parameter setting has been tested). All three memetic climbers learn significantly better solutions than the hill-climber. The differences in final fitness between the different types of memetic climbers are very small (though the standard variety seems marginally better); the differences in learning speed, however, are quite large. Only the standard memetic climber was able to match the hill-climber's speed, with the inverse climber being much slower, and the constrained climber slower still. The standard variety reaches close to final fitness after 50 topology mutations, whereas it is unclear whether the constrained variety has leveled off after 400.

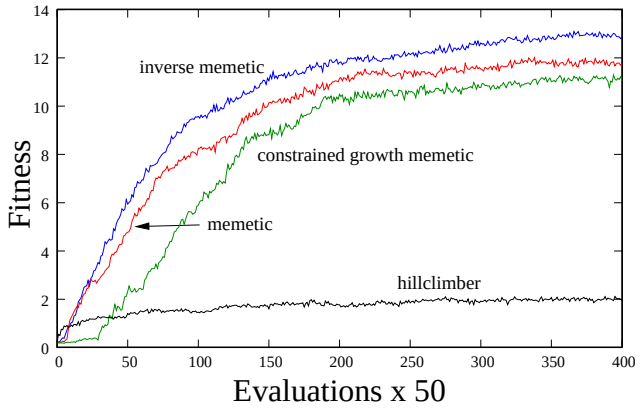

Fig. 8. **Algorithm comparison for extended inputs**: hill-climber ($p = 1.0$), memetic climber ($p = 0.0$), inverse memetic climber ($p = 0.0$) and constrained growth memetic climber ($p = 0.0$). Each curve is the average of 50 runs.

In the same manner, all four types of climbers are compared on networks with extended input sets in figure 7 (note that the memetic climber starts with $p = 0.0$ for these networks). The most obvious effect here is that all three memetic climbers vastly outperformed the standard hill climber. The difference between the memetic climbers was less pronounced in terms of learning rate, and more pronounced in terms of final fitness, than is the case for networks with standard inputs. The inverse memetic climber came out slightly better than the other two memetic climbers on both measures; additionally, the standard deviation in final fitness for the inverse memetic climber was lower (2.1) than for the standard (3.9) or constrained (4.0) varieties. The main result, however, is that all three memetic climbers solved the problem reliably, whereas the hill-climber never solved it.

V. Discussion

As hypothesized the memetic climber outperforms the standard hill-climber on both versions of the benchmark. The modest performance advantage on the standard input version of the problem may reflect that the score attained by the hill-climber is already very close to optimal for a reactive controller using this limited subset of inputs (those controllers that scored above 15 in the 2007 CEC competition accessed larger subsets of the game state, and often included a simulation of the complete game environment within the controller).

The magnitude of the performance increase for the memetic climber compared to the standard hill-climber when the using the extended inputs is somewhat surprising, however. Even more so is the good performance of the inverse memetic climber, suggesting that the important factor for success of this type of algorithm is that the two types of search occur at different time scales, and that which type happens at which of time scales is less important.

A. Parameter settings

Most of the parameter settings for the algorithms presented above where selected based on intuition, without much search for other settings. Most significantly, this includes the number of local search steps, m, per global search step, n, (e.g. number of weight mutations per topology mutation in the standard memetic climber), and the rate of growth in the constrained memetic climber.

The number of local search steps per global mutation is probably the most important parameter of this class of algorithms, and should be explored further, including a self-adaptive variation where the ratio between local and global search changes during the search. Another appealing possibility is to not have a fixed number of steps to search, but rather continue the local search until no progress has been made for a specified number of steps.

The modest impact of constraining network size in the memetic climber may very well be a result of poor settings for the growth constraint parameters. For example, the absence of fitness growth at the very beginning of runs of the constrained memetic climber point to increasing the initial proportion of allowed connections. Again, this merits further investigation.

B. Possible extensions to population-based search

While the memetic climber performed very well in the race car task, it still searches topology space using a single search point (*structural hill climbing*) and is therefore susceptible to local minima [14]. Applying the principle to a population-based framework would have a clear advantage in this respect. The simplest such approach would amount to parallelizing the memetic climber, with successful networks having a number of offspring and unsuccessful networks being removed from the population. Introducing crossover into such an algorithm, we could either choose to see each network as composed of two "natural" building blocks (the mask and the connection weights) and perform crossover so that a mask from one network was combined with the weights of another, or restrict crossover to networks with the same or similar masks, in an effort to battle the competing conventions problem. Alternatively, a single mask could be used for the whole population, and population-based search used for the weights only. A yet more interesting prospect is to cooperatively coevolve masks and weights.

VI. Conclusions

This paper explored the very simple idea of evolving topologies and weights of neural networks on different time scales. The hypothesis being that by only keeping a topology mutation if a subsequent hill-climb in weight space yielded an improvement relative to the previous topology, the destructive effects of topology mutation could be avoided. At the same time, the search in topology space would find topologies that avoided the sort of neural interference that often causes local optima for weight space search. Three variations of the this memetic climber were compared to a standard hill-climber on two versions of an established car racing benchmark. The memetic climbers were very competitive when networks were fed low-dimensional sensor input, and vastly outperformed the hill-climber when high-dimensional input was used. The memetic climber is a simple algorithm with broad applicability, and the core idea can easily be combined with population-based evolutionary algorithms.

VII. Acknowledgments

This research was supported in part by the Swiss National Science Foundation (SNF) grant number 200021-113364/1.

References

[1] X. Yao, "Evolving artificial neural networks," *Proceedings of the IEEE*, vol. 87, no. 9, pp. 1423–1447, 1999.

[2] F. Gomez, J. Schmidhuber, and R. Miikkulainen, "Efficient non-linear control through neuroevolution," in *Proceedings of the European Conference on Machine Learning (ECML)*, 2006.

[3] D. B. D'Ambrosio and K. O. Stanley, "A novel generative encoding for exploiting neural network sensor and output geometry," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*. New York, NY: ACM, 2007.

[4] F. Gruau, "Neural network synthesis using cellular encoding and the genetic algorithm," Ph.D. dissertation, Ecole Normale Superieure de Lyon, 1994.

[5] K. O. Stanley, "Efficient evolution of neural networks through complexification," Ph.D. dissertation, Department of Computer Sciences, University of Texas, Austin, TX, 2004.

[6] P. Moscato, "On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms," Caltech Concurrent Computation Program, Tech. Rep., 1989.

[7] S. Nolfi, "Evolving robots able to self-localize in the environment: the importance of viewing cognition as the result of processes occuring at different timescales," *Connection Science*, vol. 14, no. 3, pp. 231–244, 2002.

[8] S. M. Lucas and J. Togelius, "Point-to-point car racing: an initial study of evolution versus temporal difference learning," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2007.

[9] R. Calabretta, A. Di Fernando, G. P. Wagner, and D. Parisi, "What does it take to evolve behaviorally complex organisms?" *BioSystems*, 2002.

[10] R. De Nardi, J. Togelius, O. Holland, and S. M. Lucas, "Evolution of neural networks for helicopter control: Why modularity matters," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2006.

[11] L. A. Levin, "Universal sequential search problems," *Problems of Information Transmission*, vol. 9, no. 3, pp. 265–266, 1973.

[12] J. Schmidhuber, "Optimal ordered problem solver," *Machine Learning*, vol. 54, pp. 211–254, 2004.

[13] J. Togelius, "Optimization, imitation and innovation: Computational intelligence and games," Ph.D. dissertation, Department of Computing and Electronic Systems, University of Essex, Colchester, UK, 2007.

[14] P. J. Angeline, G. M. Saunders, and J. B. Pollack, "An evolutionary algorithm that constructs recurrent neural networks," *IEEE transactions on Neural Networks*, vol. 5, pp. 54–65, 1994.