Learning Agent Goal Structures by Evolution

H. Van Dyke Parunak[0000-0002-3434-5088]

Parallax Advanced Research, Beavercreek, OH 45431, USA
van.parunak@gmail.com

Abstract. When social models test theories and make predictions about real scenarios, they must be fit to observed behaviors. Realistic modeling frameworks offer multiple interacting mechanisms, each with parameters that can be fit. Previously, we demonstrated how to fit the *preferences* that SCAMP agents use to make tactical decisions. This paper extends that work by reporting experiments on fitting the *hierarchical goal networks* that guide more strategic decisions.

Keywords: Agent based modeling, genetic algorithm, model fitting, HGN.

1 Introduction

This paper extends the work in [11] on fitting social models in SCAMP [10] to observed behaviors.

SCAMP models several human decision mechanisms, including the agent's immediate *preferences* over features of accessible options, its longer term *goals*, *mental simulation* of the immediate future, the influence of the *geospatial environment*, and the *social influence* of other agents. Each mechanism has its own underlying structures, and any of them might be adjusted to fit the model to observed behavior. In [11] we recovered an agent's immediate preferences, using a genetic algorithm. Agents also have longer-term objectives, hierarchical goal networks (HGNs) that identify high-level goals and the lower-level goals into which they are decomposed [12]. This paper seeks to recover these HGNs from observed behavior, using (like [11]) stochastic iterative optimization [1].

SCAMP preferences are vectors that lend themselves to the standard genetic operators of mutation and crossover inspired by natural processes on chromosomes, and we recovered reasonable preference vectors with a genetic algorithm. An HGN is not linear, but a rooted directed acyclic graph. A promising approach is genetic programming (GP), developed to evolve parse trees for computer programs [3].

See [11] for a survey of prior work on evolving simulation models. We repeat a brief summary of the SCAMP causal modeling system (Social Causality using Agents with Multiple Perspectives), so that this paper can be read by itself (Section 2). Then we define the basic genetic constructs (genome, mutation, crossover, and fitness) in this space (Section 3), outline our test data (Section 4), report experiments (Section 5), and discuss lessons learned and next steps (Section 6).

This paper is preliminary in two ways.

1. Our main objective is to demonstrate the method we have developed, with experiments on a single test data set. We need to evaluate the method's ability to handle data with differing underlying parameters.
2. Agent preferences and goals interact to determine behavior. Concurrent evolution of both mechanisms is an important subject for later work.

2 The SCAMP causal model

SCAMP is an agent-based framework developed to generate realistic social data from an explicit causal model. Our original model, inspired by Syria's recent history, included groups of agents representing an oppressive government, its military, a democratic armed opposition, violent extremists, relief agencies, and civilians. We describe SCAMP in more detail elsewhere [7-9]. SCAMP can be viewed as a causal language, with advantages over more conventional formalisms such as Bayesian networks [10].

SCAMP is stigmergic: agents move through their environment, leaving marks and guiding their decisions by marks from other agents. Their environment consists of two graphs. The nodes in the bipartite directed *Causal Event Graph* (CEG) alternate between event types and relations between successive events indicating that an agent currently participating in one type of event could choose to participate in one subsequent event (*then*) or a group of concurrent events (*thenGroup*). Some types of events require a participating agent to drop into a *geospatial lattice* and move to a destination. The CEG has one START event where all agents begin, and one STOP evnt where they all arrive. A *trajectory* is a sequence of CEG nodes in the order visited; a *path* is such a sequence beginning with START and ending with STOP. When an agent reaches STOP, one option (adopted in our experiments here) is to go back to START and generate a new path. Such a restart does not necessarily repeat the initial path, because event features (urgency and presence) and agent preferences (wellbeing) can change as the system runs.

Nodes in both the CEG and geospace have vectors of *features* of three types. *Wellbeing features* are intrinsic consequences of participating in an event (e.g., impact on physical health; terrain difficulty). *Urgency features* capture the impact of participation on the goals of each group. *Presence features* record the level of recent participation by agents of each group. Urgency features on CEG nodes are computed by an HGN for each group, which translates participation on CEG nodes (event types) that support or block its leaf-level subgoals into an estimate of the urgency of those event types for satisfying the overall goal.

Each agent carries a vector of *preferences* defined over the same vector space in which features are defined. To choose its next action, the agent

- computes the dot product of its preferences and the features of each alternative,
- exponentiates each dot product to yield a positive number,
- raises each to the power of a determinism parameter,
- and normalizes them to form a roulette.

SCAMP's agents are polyagents [5]. Each actor has an *avatar* that sends out repeated waves of *ghosts* to explore possible futures. Each ghost considers the full fea-

ture vector of each alternative, and augments the presence feature of its avatar's group on the nodes that it visits. The avatar follows its group's presence features.

Fig. 1 shows how an HGN operates.[1] Each (sub)goal maintains four scalar values in [0, 1]: Satisfaction, Frustration, Urgency, and Tolerance.

1. The sum of presence features on each event node reflects the total recent participation in the event across all groups.
2. Each leaf subgoal in the HGN is "zipped" to one or more events. *Support* zips increase the subgoal's Satisfaction with increased event participation, while *block* zips increase its Frustration with increased participation.
3. Satisfaction and Frustration propagate up the HGN through *and* and *or* links. An *and* passes the minimum Satisfaction and the maximum Frustration of its lower-level subgoals up to the next higher goal, while multiple *or*s pass up the maximum Satisfaction and minimum Frustration of their subgoals.
4. At the root, Urgency = 1 – Satisfaction and Tolerance = 1 – Frustration.
5. Urgency and Tolerance propagate back down. Support zips increment the urgency of their events by Urgency, while block zips increment it by Tolerance – 1 (that is, decrementing it by a value in [-1, 0].
6. Urgency features on the events then guide agent choices moving over the CEG.

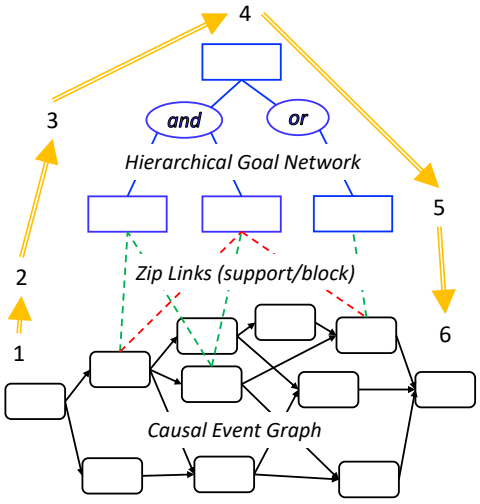

Fig. 1. Operation of HGN in SCAMP

3 Applying Genetic Programming to an HGN[2]

Genetic methods require a *population* of potential solutions ("genomes", 100 in the experiments reported here), *operations* that can modify one solution to produce another that is structurally valid, and a *measure* of how well a solution solves the problem being addressed.

[1] This mechanism is a refinement of TÆMS [2,6], which handles goal satisfaction and frustration symmetrically.

[2] J. Greanya developed the GP framework used here, like the GA framework used in [11].

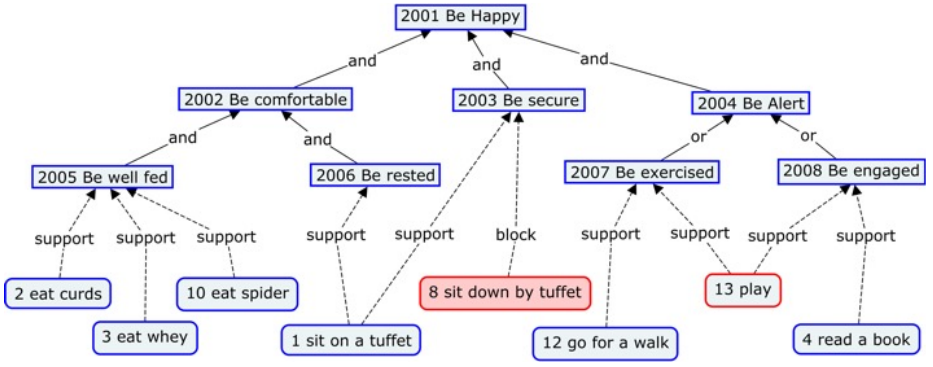

Fig. 2. Simple example of an HGN. Rounded rectangles are CEG nodes.

3.1 The Genome for an HGN

Fig. 2 is an example HGN from a toy model inspired by the children's rhyme "Little Miss Muffet" [8,10]. Each goal or subgoal is a rectangle with square corners, and has a unique identifying number. The leaf subgoals are zipped to events in the CEG (rounded rectangles), each also with a unique identifier.

Sometimes we need a string representation of the HGN. We use a preorder traversal, in this case visiting the goals in the order 2001, 2002, 2005, 2006, 2003, 2004, 2007, 2008. Fig. 3 shows the string for Fig. 2.

```
Goal(and(
   Goal(and(
      Goal(support(E2), support(E3), sup-
port(E10)),
      Goal(support(E1))),
   Goal(support(E1), block(E8)),
   Goal(or(
      Goal(support(E12), support(E13)),
      Goal(support(E13), support(E4))))))
```

Fig. 3. Textual Representation of HGN in Fig. 2.

To generate each genome in the initial population, we begin with a root node, and recursively add layers. At each layer, we add a stochastically chosen number of children, where the probability of adding children depends on a parameter *siblingChance* (0.3 in these experiments). This parameter, divided by the number of current children, gives the probability of adding a new child. The children can be either subgoals, or zipped events, where subgoals are the probability of a zipped event increases with the depth of the current node. The choice between *and* and *or* for connecting subgoals is also randomized.

3.2 Mutation and Crossover

Each generation creates one mutated child and optionally one crossover child from each individual in the current population, yielding a new population up to three times the size of the original. Then for population of size n (here, 100), we select the n most fit individuals.

In a genetic algorithm (e.g., [11]), the genome can be treated as a bit string and mutation and crossover are straightforward. A genetic program must respect the structural integrity of the genome, making it more complicated.

Mutation walks through a preorder list of the nodes in the HGN. It mutates each node with a configurable probability (0.5 in the runs reported here), by replacing it with a new randomly generated node. If the node being replaced is a leaf node, the new node grows a subtree. Otherwise it takes over the children of the replaced node.

Crossover first selects a partner, then selects a node from a preorder list of the receiving genome and replaces it (and thus its descendants) with a node (and descendants) from the partner. Crossover can select a partner based either on fitness or diversity from the host genome.

Crossover can reduce the diversity of the population. This problem did not arise in evolving preferences, but was dramatic in evolving HGNs. Defining diversity as the number of distinct genomes in the population divided by total population size, Fig. 4 shows the loss of diversity over ten generations. The result is that the algorithm wastes time evaluating the fitness of identical genomes, while failing to explore as widely as it should.

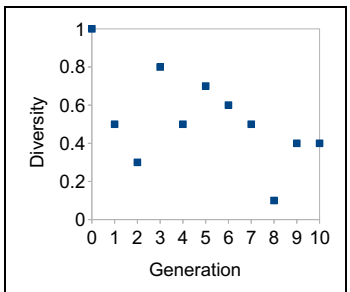

Fig. 4. Diversity over Time

The problem arises with HGNs because the space of genomes is smaller than with preferences. The preference genome [11] in our test model is a vector of eight real numbers in [-1, 1], and the chance of duplication is very low. For HGNs, the comparable structure is a preorder list of nodes, drawn from the alphabet {Goal, and, or, support, block, Enn*}, where Enn* represents the 300 event node names (Section 4). The length of the preorder list is not fixed, but can be as short as three (e.g., (Goal (support (E23))). In such a reduced space, the chance that crossover (or even mutation) will generate a genome already in the population is much greater.

To solve this problem, while generating a new population, mutation or crossover of a selected member is repeated until it generates a genome not already in the set.

3.3 Measuring Genome Fitness

Finding a good fitness function proved challenging.

As in [11], we seek to recover a model that can generate an observed sequence of behaviors, a trajectory through a SCAMP Causal Event Graph (CEG). Thus we need to compare the *test trajectory* (generated by the evolved HGN) against the *target trajectory* (which was initially observed).[3] To discourage the growth of monstrously large HGNs, we penalize each fitness result by the number of nodes in the HGN under test divided by 10^5. Without the length penalty, all fitness values are in [0, 1].

Fitness0: [11] achieved good results by comparing the overall trajectory (including any restarts) of an agent with an evolved preference vector against the overall trajectory of the original agent, using the Levenshtein or string edit distance [4]. This met-

[3] *Test* refers to a trajectory or subpaths generated by the *evolved* HGN. *Target* refer to trajectories and paths either input directly, or generated by the *input* HGN.

ric counts the minimum number of characters in one string (behaviors in one trajectory) that must be inserted, deleted, or change to match the other string.

This measure did very poorly in recovering HGNs. probably because it must deal with more noise than other functions, and because we are running fewer generations than we did in [11].

The Levenshtein metric has two sources of noise.

1. The trajectories include not only event names, but also the relations between them (potentially Then and ThenGroup, though our experiments use only Then). Since event names and relations alternate, fully half of each trajectory being matched consists of elements that vary only a little between trajectories.
2. The Levenshtein metric is efficient for alphabets on the order of [0-1a-zA-Z], or about 60 characters. A mature SCAMP model can have hundreds of event nodes, and our experimental CEG (Section 4) has 300 distinct event nodes. So our implementation hashes the event names in a trajectory into alphabetic characters before matching, with the result that each character can represent on average five different events. As a result, some differences in trajectory are invisible.

In [11] we evaluated candidate preference vectors against a historical record of event features without actually running the SCAMP algorithm. Evolution ran until it reached a fitness of 0 or 500 generations, and while most genomes required fewer than 10 generations, some ran much longer. To evaluate an HGN, we must actually run the SCAMP engine, which is much slower. We stop when the best candidate does not change for five generations, or we reach generation 20. Fitness0 succeeded with 500 generations, but not with 20.

Our other fitness measures avoid problem 1 by focusing on event names and not relations, and problem 2 by focusing only on the event names in a specific trajectory, ignoring the far larger number that are in the CEG but are never visited. These measures also focus on the set of event nodes visited and not their order, since the order is in any case largely fixed by the directed edges in the CEG. We omit START and STOP from the set of nodes considered, since they appear in all trajectories. A *path* is a sequence from START to STOP. The *path spectrum* is a function from each complete path to its frequency, and the *node spectrum* is a function from each event node to its frequency. The *length* of a spectrum is the size of its domain.

Fitness1: Define a vector space over the union of the domains of node spectra for the test and target trajectories. The target and test vectors give the frequency of each node in the overall trajectory. Their cosine similarity is Fitness1.

Fitness2: Like Fitness1, but using the counts of complete paths from the path spectra instead of the nodes from the node spectra (which may include partial paths).

Fitness3: The length of the longest prefix shared between the test and target trajectories, divided by the length of the target trajectory.

Fitness4: Divide the size of the intersection of the domains of test and target node spectra by the length of the target spectrum.

Fitness5: Define the miss ratio as the size of the set difference of the domain of the test node spectrum and the target spectrum, divided by the length of the test spectrum. Report Fitness4 times (1 - the miss ratio). We designed this function to improve on Fitness4 by penalizing false positives, but its behavior shows that the corrective factor interferes with recovery of the target HGN.

4 Test Data

To facilitate experimentation, a parameterized CEG generator constructs CEGs and groups from which agents are generated. It grows a directed acyclic graph of a specified number of nodes with a single START node, in which every node lies on a path from START to a single STOP node.

Table 1. Parameters of Test System

Parameter	Value
Number of event nodes (exc. START and STOP)	300
Target node degree	6
Max path length	30
Min path length	20
Number of groups	2

The algorithm attempts to enforce a maximum node degree and a minimum and maximum path length between START and STOP. However, the requirement that every node be on a path from START to STOP sometimes requires adding edges beyond the desired node degree, and short-cuts that emerge between trajectories lead to paths shorter than the desired minimum. Table 1 summarizes these parameters in our experiments. The mode of the actual degree distribution is 4, the mean node degree is 9.1, and the shortest paths (101 in number) have length 3.

The test CEG includes two groups. Both have access to all nodes. As in [11], the feature space is of length 6. The first four elements (including the third and fourth, representing urgency for the two groups) are initialized randomly in [-1, 1], while the last two (presence features) are initially 0. We seek an HGN for Group 0, whose preference vector is [.8, -.8, .8, -.8, .8, -.8]. That is, the agent is attracted to high urgency for Group 0 and the recent presence of other agents from Group 0. The overall population consists of four agents of Group 0, and four of Group 1, with the inverse preference vector, [-.8, .8, -.8, .8, -.8, .8].

We tested HGN evolution in two ways: by starting with a known HGN, and by giving the system a trajectory through the CEG. The latter represents our intended use case. But by using a known HGN to generate the target trajectory, we can also evaluate how close our recovered HGN is to the original.

Table 3 shows the paths generated by the CEG with no HGN, omitting the initial 'E' in node names. E0 is START, and E301 is STOP. The system is attracted to paths beginning 175-203-165.[4]

We ran the algorithm backward, inverting Fitness4 (our most promising candidate) to find an HGN whose trajectory differs maximally from no HGN. HGN1 is Goal(support(E126)), with the path spectrum in Table 2.

Table 3. Path spectrum of CEG with no HGN

Path	Count
0-175-203-165-35-301	4
0-175-203-165-301	3
0-175-203-165-94-301	2

Table 2. Path spectrum of HGN1

Path	Count
0-126-31-115-212-301	1
0-126-31-115-301	1
0-126-31-212-301	1
0-126-31-279-136-212-301	1
0-126-31-279-136-301	1
0-126-31-279-255-136-301	1
0-126-31-279-255-212-301	1
0-126-31-279-255-301	1

[4] We do not report the partial paths generated when the program terminates before the agent finishes its current traversal of the CEG.

This HGN avoids the paths generated by the CEG with no HGN, but its paths are very short. We inspected the CEG manually to find a longer path that is disjoint from Table 3, Beh(avior)1, E0-E293-E52-E47-E124-E263-E171-E301. We concatenate this path with itself to get a trajectory of the same length as the CEG with no HGN. We were able to evolve HGNs that fit most of this trajectory, but node E124 was extremely elusive.

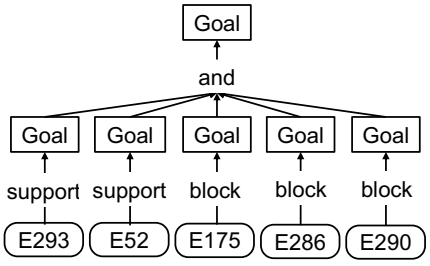

Fig. 5. Manually constructed HGN (HGN2)

So we defined HGN2 (Fig. 5). The two support goals should increase the urgency on the first two steps. The three blocking goals should generate negative urgency on their events, critical alternatives to E124 in the CEG (events that the agent uses to bypass E124 on its way to STOP).

The elusiveness of E124 led us to apply our best fitness function, Fitness4, to two additional input HGNs. HGN3 is Fig. 5 with an additional conjunct supporting E124, and HGN4 is Goal(support(E124)) by itself.

The next section describes our results in fitting HGNs to these test cases.

5 Experiments

Our experimental matrix has 18 cases: three inputs (HGN1, HGN2, and Beh1), with six fitness functions. For each case, we report:

- The number of generations. The algorithm runs until the fittest genome found is not surpassed for five generations, or until it reaches generation 20, whichever comes first.
- The fittest HGN found ('=' means that the input HGN was recovered; letters refer to the list in Table 4.
- The best fitness in the first and last generation, and the net gain in fitness
- Number of distinct nodes (excluding E0 and E301) in the target, the trajectory generated by the input
- Number of hits (nodes in the target also visited by the fittest HGN)
- Number of F(alse) P(ositives) (number of nodes visited by the HGN that are not in the target)

Table 4. Discovered HGNs

a.	Goal(block(E238))
b.	Goal(support(E41))
c.	Goal(support(E293))
d.	Goal(and(Goal(support(E171),support(E52)), Goal(support(E171),support(E293))))
e.	Goal(or(Goal(support(E293),support(E171))))
f.	Goal(support(E60))
g.	Goal(and(Goal(and(Goal(support(E45)), Goal(support(E291)))),Goal(support(E124))))
h.	Goal(support(E52),support(E293))
i.	Goal(or(Goal(support(E45),support(E124))))
j.	Goal(or(Goal(support(E124),support(E45))))

Table 5. Results for HGN1 (evolved to be maximally different from Table 3)

Input	Fit-ness	# Gen	Best	Initial	Final	Gain	Base-line	Hits	FPs
HGN1	0	10	=	2.8E-4	3.0E-5	2.5E-4	7	7	0
HGN1	1	6	=	2.8E-4	3.0E-5	2.5E-4	7	7	0
HGN1	2	5	a	1.00	1.00	0.00	7	0	5
HGN1	3	6	=	2.8E-4	3.0E-5	2.5E-4	7	7	0
HGN1	4	6	=	2.8E-4	3.0E-5	2.5E-4	7	7	0
HGN1	5	6	=	2.8E-4	3.0E-5	2.5E-4	7	7	0

All runs have population size 100, mutation probability 0.5, crossover on and only picking a partner with potentially better fitness while favoring maximum diversity. In constructing an initial genome, the base probability of adding a child to a node that already has one (*siblingChance*) is 0.3.

For efficiency, all runs start with the same image of the system, so all runs have the same random number seed. This would be a problem if we were doing multiple runs with the same parameters, but is helpful in comparing different configurations.

Table 5 shows the results of our fitness functions on HGN1, Goal(support(E126), which we evolved to be maximally different from the no-HGN case.

All fitness functions except Fitness2 recovered this very simple HGN exactly. The input HGN has only three nodes in its linear expansion. There is only one option for the first (Goal), two for the second (support, block), and 300 for the third (an event name). Since the system samples 300 genomes in each generation, we would be surprised if such a simple HGN were not recovered.

Fitness2 fails because it gives no credit for paths in the target and test trajectory that are similar but not identical. Fitness2 has no leverage in the early phases of evolution, and terminates in five generations, which means that the winning genome was part of the original population. This genome (Goal(block(E238))) appears several times in the following tables, always with runs that terminate in 5 generations. It is the same in each case because the runs have the same random seed. They all generate this genome and sort it to the first place when ties are broken. The lack of any leverage is seen in the gain of 0.00 for Fitness 2. HGN *a* generates the same path spectrum as no HGN (Table 3).

The low fitness values for successful cases show that the fitness is dominated by the penalty for solution length, since the correct solution is found early.

Table 6 shows the results for the manually constructed path. Not only does Fitness2 fail, but so do Fitness0 and Fitness1. Of the other three functions, Fitness4 is superior, capturing five of the six nodes in the target with fewer false positives than the others. The superiority of Fitness4 to Fitness5 is surprising, because Fitness5 was designed to penalize false positives, but the correction apparently interferes with the ability of Fitness5 to find its way through earlier generations to a superior solution.

Table 6. Results for Beh1 (manually constructed path)

Input	Fit-ness	# Gen	Best	Initial	Final	Gain	Base-line	Hits	FPs
Beh1	0	8	b	0.38	0.38	0.00	6	0	2
Beh1	1	5	a	3.0E-5	3.0E-5	0.00	6	0	5
Beh1	2	5	a	3.0E-5	3.0E-5	0.00	6	0	5
Beh1	3	9	c	0.98	0.96	0.03	6	2	5
Beh1	4	19	d	0.83	0.17	0.66	6	5	3
Beh1	5	19	e	0.98	0.78	0.20	6	3	4

Table 7. Results for HGN2 (manually constructed to reach E124)

Input	Fit-ness	# Gen	Best	Initial	Final	Gain	Base-line	Hits	FPs
HGN2	0	7	f	0.40	0.39	0.01	8	2	5
HGN2	1	20	g	0.58	0.35	0.23	8	4	5
HGN2	2	5	a	1.00	1.00	0.00	8	2	3
HGN2	3	9	c	0.98	0.96	0.02	8	4	3
HGN2	4	12	h	0.62	5.0E-5	0.62	8	8	0
HGN2	5	12	i	0.83	0.55	0.28	8	5	2

No fitness function could find E124. The node is not unreachable. Genomes (*g* and *i* in Table 4) that support E45 retrieve it, always as the third element in a trajectory that begins E45-E47, which does not match the rest of the target path.

Table 7 show the results for HGN2. Though we designed the HGN to block the events that most commonly bypass E124, the path that the evolved HGN generates does not include E124, as discussed further below. All of the fitness functions recovered some nodes from the target path, but Fitness4, which recovered all with no false positives, is clearly superior.

Though Fitness4 perfectly recovers the behavior generated by the input HGN, it does not exactly replicate the HGN. The input HGN (Fig. 5) was Goal(and(Goal(support(E293)), Goal(support(E52)), Goal(block(E175)), Goal(block(E286)), Goal(block(E290)))). We recovered Goal(support(E52), support(E293)), which has the same effect as the first two conjuncts in the original HGN. The blocking events have no effect on the resulting behavior.

Table 8 shows the results of Fitness4 on HGN3 (adding support for E124 to HGN2) and HGN4 (support for E124 alone).

Table 8. Fitness4 on HGN3 and HGN4

Input	Fit-ness	# Gen	Best	Initial	Final	Gain	Base-line	Hits	FPs
HGN3	4	14	j	0.57	0.28	0.29	7	5	2
HGN4	4	5	a	3.0E-5	3.0E-5	0.00	5	5	0

Adding Goal(support(E124)) to HGN2 makes a slight change to the path spectrum of the target (Table 9). Neither target includes E124. However, the difference leads to a new HGN (*j*). Like its functional equivalent *i, j* does generate some paths that begin E45-E47 and include E124.

HGN4 generates a target path spectrum identical to no HGN, and leads to HGN *a*, whose spectrum is the same.

Table 9. Target path spectra for HGN2 and HGN3

Path	HGN2	HGN3
0-293-52-263-301	1	2
0-293-52-263-35-301	1	2
0-293-52-263-94-301	0	2
0-293-52-47-55-301	2	0
0-293-52-47-72-263-301	1	0
0-293-52-47-72-263-94-301	1	1
0-293-52-47-72-55-94-301	1	0
0-293-52-47-72-94-301	0	1

6 Discussion

Can we learn SCAMP HGNs from behavioral traces? The answer has three parts.

1. As with preference vectors [11], we can recover an HGN that generates the same trajectory as a furnished HGN.
2. We are not always able recover the original HGN. We could recover HGN1, but with HGNs 2, 3, and 4, what we recovered differs to some degree from the original HGN (though the trajectory of the evolved HGN matches that of the original). Again, our experience parallels [11], where we invoked the metaphor of nature (there, the preference vector; here, the HGN) vs. nurture (the environment, in this case, the structure and features of the underlying CEG). Much of behavior is constrained by the environment, allowing the same behavior to emerge from different structures in the agent.
3. We can define a legitimate trajectory in the CEG that we cannot recover completely with evolution (Beh1). The problem is a single recalcitrant event (E124). It does appear in some paths generated by evolved HGNs, but not in the context of the overall behavior that we were seeking to recover.

E124 deserves further study, but we hypothesize that the manually defined trajectory Beh1 is simply inconsistent with the overall dynamics of the system (which includes not only the feature vectors on the nodes and the preference vectors on the agents, but the behavior of other agents modulating the presence features of other event nodes that attract the agent being monitored onto alternative paths). Again, the environment has a major impact on behavior.

An important lesson is the importance of the fitness function. The simple string edit distance is too noisy to converge under the stricter execution time limits imposed by HGN processing. We found good results by focusing on the events visited by a trajectory, recognizing that the order of the events was constrained by the CEG and so did not need to be evaluated. However, even here the function is tricky. What looks like an obvious improvement to Fitness4, a simple penalty function to discourage visiting nodes not in the target spectrum, actually decreases performance.

Our focus here was on developing methods for applying genetic programming to learning HGNs. We did not explore the parameter space of CEGs or of the GP algorithm, and future research should study these.

Perhaps the biggest open question is how these mechanisms can operate concurrently with those in [11]. It is unlikely that we will need to fit only the preference vector, or only the HGN. We will probably want to refine initial estimates of both structures concurrently. The (in)accessibility of E124 most likely results from the interaction of the agent's preferences and goals in determining its behavior.

7 References

[1] Corne, D., Dorigo, M., Glover, F., Editors: New Ideas in Optimization. New York, McGraw-Hill (1999)

[2] Horling, B., Lesser, V., Vincent, R., Wagner, T., Raja, A., Zhang, S., Decker, K., Garvey, A.: The TÆMS White Paper. Multi-Agent Systems Lab, University of

Massachusetts, Amherst, MA (2004). http://mas.cs.umass.edu/pub/paper_detail.php/182

[3] Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MIT Press (1992)

[4] Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady, 10(8):707-710 (1966)

[5] Parunak, H.V.D., Brueckner, S.: Concurrent Modeling of Alternative Worlds with Polyagents. In Multi-Agent-Based Simulation VII, LNCS, vol. 4442, pp. 128-141. Springer, Berlin, Germany (2007)

[6] Parunak, H.V.D., Belding, T., Bisson, R., Brueckner, S., Downs, E., Hilscher, R., Decker, K.: Stigmergic Modeling of Hierarchical Task Networks. In Proceedings of the Tenth International Workshop on Multi-Agent-Based Simulation (MABS 2009, at AAMAS 2009), pp. 98-109, Springer (2009)

[7] Parunak, H.V.D.: Social Simulation for Non-Hackers. In Van Dam, K.H., Verstaevel, N., editors, 22nd International Workshop on Multi-Agent-Based Simulation (MABS 2021), Springer, London, UK (2021)

[8] Parunak, H.V.D.: Psychology from Stigmergy. In Proceedings of 2020 Conference of The Computational Social Science Society of the Americas, pp. 203-216, Springer International Publishing (2021)

[9] Parunak, H.V.D., Greanya, J., McCarthy, M., Morell, J.A., Nadella, S., Sappelsa, L.: SCAMP's Stigmergic Model of Social Conflict. Computational and Mathematical Organization Theory, (2021)

[10] Parunak, H.V.D.: How to Turn an MAS into a Graphical Causal Model. Journal of Autonomous Agents and Multi-Agent Systems, 36 (2022)

[11] Parunak, H.V.D.: Learning Actor Preferences by Evolution. In Proceedings of The 2021 Conference of The Computational Social Science Society of the Americas, pp. 85-97, Springer International Publishing (2022)

[12] Shivashankar, V.: Hierarchical Goal Networks: Formalisms and Algorithms for Planning and Acting. Thesis at University of Maryland, Department of Computer Science (2015)