NeuroHunter - An Entry for the Balanced Diet Contest

Wojciech Jaśkowski Krzysztof Krawiec Bartosz Wieloch

July 14, 2008

Institute of Computing Science, Poznan University of Technology
Piotrowo 2, 60965 Poznań, Poland

1 Analysis of the task

The Balanced Diet task has a couple of important properties that lead to the design choices concerning our agent.

The agent cannot see food pieces, so it has no 'immediate incentive' that would directly determine the next move. It may only rely on probabilistic dependencies that relate grain/game occurrence with elevation. Whether it will pick a piece of food in the next move or not is matter of luck. However, one thing is certain: moving into an obstacle or a previously visited cell will gain it nothing. Therefore, the agent should definitely avoid visiting the same cells as this implies loss of time.

Most obstacles are two-cells wide. Agent's field of view (FOV), i.e., the part of the board passed to it by means of the `grid` parameter, is limited to two cells in each direction either. Thus, the range of agent's visibility does not span behind the obstacles, even when it is standing right in front of it. Therefore, without some extra mechansism (like long-term memory), an agent cannot tell apart obstacles from the board boundary. This is an important problem: an agent misinterpreting board end as an obstacle may waste precious turns trying to walk it around.

Game specification says that an attempt to make a forbidden move (i.e., into an obstacle or beyond the board/map) has no effect (apart from losing the turn), so agent's position does not change. Also, the agent is not explicitly informed about its absolute position. Therefore, without autonomous tracking of its absolute position, an agent cannot tell whether it has moved or not. Though in most cases the movement may be detected based on the change of elevations within the FOV, an autonomous evolution of such functionality would be probably difficult.

2 Agent's memory

The above observations clearly indicate that without a long-time memory, the risk of re-visiting the same cells and re-entering the areas that have been already explored is high. Thus, we decided to equip our agents in three types of long-term memory:

- Elevation memory: $e(x, y)$
- Obstacle memory: $o(x, y)$
- History memory (agent's track): $h(x, y)$

The elevation memory is intended to help the agent to maintain the balance between grain and game. The purpose of the obstacle memory is to ease detouring the obstacles. The history memory should help the agent avoiding entering of the already visited cells.

The agent also stores and updates its current absolute position (x, y), starting with $x = y = 256$; these coordinates are used to address the memory. However, as the initial position (the offset of (x, y) with respect to the actual board position) is not known to the agent, all memories are implemented as 511×511 arrays to safely store the data no matter what the agent's initial position is.

The memory arrays are intitially filled with zeros. At each turn of the game, they are updated based on the the current FOV, i.e., the argument `grid` passed to the agent by the caller of the `Next_Move` function:

```
int Next_Move(int *grid, int grain, int game,
              int last, int time);
```

These updates are carried out according to the following rules (for clarity, we use the same indexing scheme for e, o, and `grid`; technically, `grid` requires one-dimensional indexing expression):

$$e(x + x', y + y') = \begin{cases} grid[x', y'] - 128 & if\ grid[x', y'] \geq 0 \\ 0 & otherwise \end{cases} \quad (x', y') \in FOV$$

$$o(x + x', y + y') = \begin{cases} -1 & if\ grid[x', y'] \geq 0 \\ 1 & if\ grid[x', y'] < 0 \end{cases} \quad (x', y') \in FOV$$

What follows from these update rules is that in both memories zero encodes lack of information about a particular board cell – this applies to the cells that have not been 'seen' by the agent yet (i.e., have not appeared within its FOV). A non-zero memory value means agent's perfect knowledge about cell's elevation and obstacle. For e, it takes the value of 0 also when $grid[x', y'] = 128$, and then such a situation cannot be told apart from lack of information, but we assumed that this does not impact much the agent's performance.

History is simply updated based on agent's current position:

$$h(x, y) := 1$$

3 Agent's input variables

The memory updates detailed in the previous section are carried out at the very beginning of each execution cycle of `Next_Move` function. This nicely integrates the memory with the current FOV. The decision making process that follows is based exclusively on the memory state.

Agent's decision is based on a set of *input variables* that reflect the state of the memory. However, a one-to-one mapping of the memory state into input variables would imply lots of variables and render the evolution of the strategy infeasible. To keep the number of input variables within reasonable limits, we (i) limit agent's perception of the memory to a predefined neighborhood, and (ii) reduce the spatial resolution of that perception with the increasing distance from agent's current position.

Perception of obstacles is precise but low-ranged; the agent perceives a 7×7 patch of the obstacle memory surrounding its current position (x, y). Thus, the range of obstacle perception is 3 cells, one more than what the FOV provides. We expect that this will give the agent an advantage, especially for telling apart the obstacles and the board limits. There are therefore $7 \times 7 = 49$ obstacle input variables.

Perception of elevation and history deteriorates with distance from the agent's position. This multi-resolution perception may be visualized as a stack of grids overlaid over the memory, centered around the agents's current location. We use three such grids:

- The first (low-range) grid has the same resolution as the memory. The agent perceives a 5×5 square of its immediate vicinity from that grid.
- The second grid, responsible for middle-range perception, uses 5-fold reduction of resolution. Each variable from that grid corresponds to a 5×5 square of memory cells and averages their values. The agent perceives a 5×5 square from that grid (which corresponds to 25×25 square in terms of the original coordinates).
- The third grid implements the perception of the furthest cells and downgrades the original memory resolution 45 times. One variable (square) from that grid averages the state of $45 \times 45 = 2025$ memory cells. The agent receives a 11×11 square from that grid, corresponding to 495×495 cells of the memory. The range of this perception mode is therefore $495/2 = 247$, very close to the board size (256), allowing the agent a 'blurred vision' of the furthest parts of the board.

The total number of stack variables is therefore $5 \times 5 + 5 \times 5 + 11 \times 11 = 171$. Note that cells from different grids overlap; e.g., the middle element of the second grid averages all the variables from the first grid.

Figure 1 shows the central part of agent's FOV. The cell marked by 'A' marks agent's position. The 25 green cells depict the low-range grid. The yellow cells illustrate the second, middle-range grid. Finally, the 45×45 rectangle

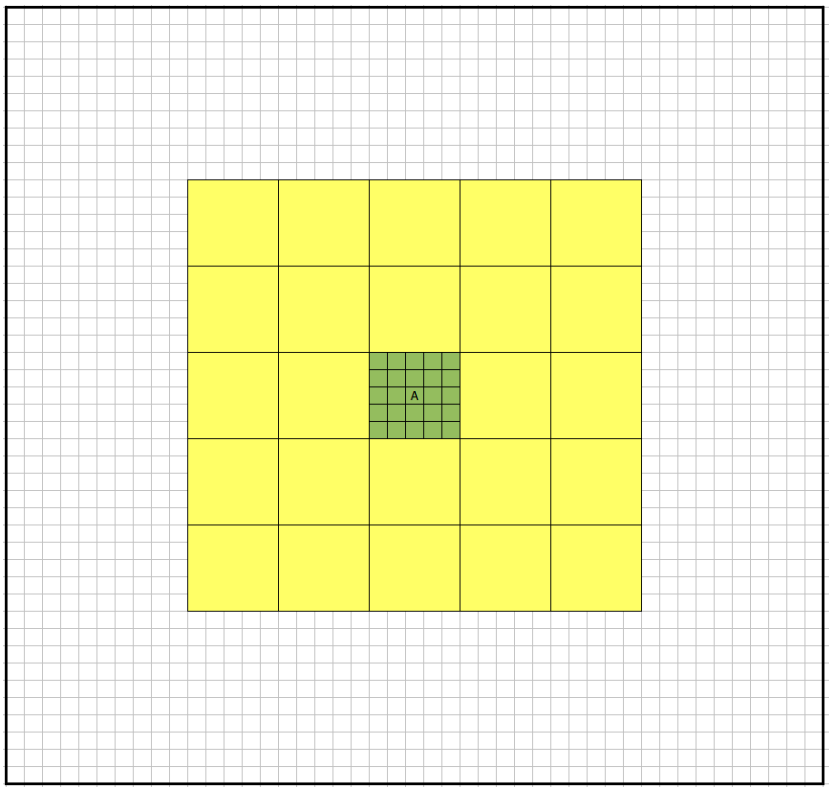

Figure 1: Organization of agent's field of view (FOV).

surrounding the figure depicts the central element of the third, low-resolution grid (the remaining elements of the low-resolution grid are not shown due to space limits).

A separate stack is built for elevation and history, so we have 171 input variables for both these memories. The set of variables related to memory state encompasses therefore 49 obstacle variables + 171 elevation variables + 171 history variables = 391 variables.

This set of variables is supplemented with three variables that are essential for meeting the task objectives, i.e., for keeping the balance between grain and game. These are:

- the grain variable – normalized value of the `grid` parameter passed to the `Next_Move` function,
- the game variable – normalized value of the `game` parameter passed to the `Next_Move` function,
- the time-left variable = (13107-`time`)/13107.

The total number of variables fed into our agent amounts therefore to 391+3=394. All input variables are normalized to interval [-1,1].

4 Agent's control

We decided to implement our agent as a neural network. This choice was motivated by the following properties of the task:

- large amount of input data that the agent has to 'perceive' to make reasonable moves (394 variables),
- the structured (planar) organization of agent's perception.

The agent' control module is a nonlinear multilayer perceptron with neurons in layers arranged in 2D rectangular grids. There are two non-linear layers in our network:

- the hidden layer: $7 \times 7 = 49$ neurons,
- the output layer: $5 \times 5 = 25$ neurons,

The input layer is built by the 394 input variables defined in the previous section. Each neuron has an extra bias weight and is equipped with a sigmoidal transfer function.

Technically, our network is fully connected: each pair of neurons from the consecutive layers has a connecting weight. In particular, each neuron in the hidden layer receives all 394 input variables (and the bias). However, the actual connection pattern may be much more sparse, as the process of phenotypic expression described in following can effectively remove some weights by assigning them the value of zero.

After feeding the network with the values of input variables and propagating the excitation through the network, the state of its output layer is directly interpreted as a recommendation for the next move. The higher the output of a particular neuron in the output layer, the more desired is the move into the corresponding board cell. The maximum output determines the agent's next move. However, the board cells containing obstacles are explicitly ignored when searching for the maximum; otherwise, the agent would easily fall into cycles, making the same move over and over again.

5 Evolution of agent's strategy

The total number of weights in our neural network amounts to 20605, which boils down to:

- (394 input variables + 1 bias) $\times$ 49 hidden neurons = 19355 weights connecting the input layer with the hidden layer,

- (49 outputs of the hidden neurons + 1 bias) × 25 output neurons = 1250 weights connecting the hidden layer with the output layer.

Such a large number of weights makes the direct encoding of network weights in the genotype infeasible (though technically possible, such an algorithm would probably need ages to converge). Thus, we decided to use an indirect network encoding, where the individual's genotype encodes the *method* (procedure) for assigning the weights to particular connections, and the phenotype is the network itself. As our network's topology is fixed, a method that provided weight values only was sufficient.

From the indirect neuroevolution approaches available in the literature, we considered the HyperNEAT by Stanley et al. [2] as the most suitable for our purpose. The main argument for this choice was HyperNEAT's ability to evolve neural networks with spatial organization of neurons in layers. This approach proved also effective in machine learning tasks [3] and multiagent learning [1].

HyperNEAT uses the NeuroEvolution of Augmenting Topologies (NEAT) algorithm to evolve the topology and weights of the so-called Compositional Pattern Producing Networks (CPPNs). CPPN is a class of specialized networks oriented towards generating symmetrical and repetitive spatial 'patterns'. In our case, CPPN determines the pattern of weights connecting the neurons in our network (termed 'substrate' by Stanley). Technically, CPPN is an oracle which, when queried with the coordinates of the source neuron and the destination neuron, returns the weight to be assigned to the connecting weight. Note therefore that, from the viewpoint of our approach, CPPN is the genotype of an individual.

Our CPPN has four inputs, denoted x_1, y_1, x_2, y_2. The pairs of inputs (x_1, y_1) and (x_2, y_2) are intended to receive the coordinates of the source and destination neuron, respectively. These coordinates are expressed in terms of the memory cells, with respect to agent's current position.

There are nine outputs from our CPPN, one for each type of weight present in our network, that is the weights connecting:

1. the obstacle variables with the hidden neurons,
2. the elevation variables with the hidden neurons,
3. the history variables with the hidden neurons,
4. the grain variable with the hidden neurons,
5. the game variable with the hidden neurons,
6. the time-left variable with the hidden neurons,
7. the hidden neurons with the output neurons,
8. the bias neuron (constant output=1) with the hidden neurons,
9. the bias neuron (constant output=1) with the output neurons.

By dedicating separate CPPN outputs to particular weight types, we enable semi-independent evolution of the ways in which our agent perceives elevation, obstacles, history, etc.

The coordinates x_1, y_1, x_2, y_2 fed into CPPN are normalized with respect to the agent's range of perception, so that the relative range of coordinates $[-247, 247]$ is mapped to the $[-1, 1]$ interval. Querying the CPPN for the weights of type 4, 5 and 6 needs special care, as the source input/neuron has no coordinates. In such a case, we set $(x_1, y_1) = (0, 0)$. We also follow the HyperNEAT's default setting consisting in expressing only the weights with magnitude greater than .3 (a smaller absolute value returned by the CPPN results in zero weight).

6 Experiment setup and technical implementation

Most of HyperNEAT parameters have been set to the default values. Appendix A lists the settings of all HyperNEAT parameters.

The starting point for our evolutinary system was the original implementation of HyperNEAT, i.e., HyperNEAT 2.0 C++ by Jason Gauci. However, we decided to use only the evolutionary part and CPPN part of the library, and to develop our own implementation of the neural network (substrate). In summary, we extended Gauci's software by the following components:

- neural network implementation,
- game simulator,
- distributed computing modules.

In initial experiments, we were evaluating our agents on a single map only; in other words, our training set contained one map. However, we observed significant overfitting: the evolved individuals performed much worse when evaluated on test maps. This clearly indicated that the differences between maps cannot be ignored and that a more representative training set is a must. Therefore, in the main evolutionary experiment (the one that produced our contestant) an agent is tested on each of the 10 maps provided by the organizers. Also, each game starts always from a random initial position of the agent, so two fitness function calls may return different fitness values for the same individual.

We introduced also an extension that increases the precision of fitness measurement with evolution time. Technically, we gradually increase the number of games played by the individuals. For the first 100 generations the agents play once on each map, then for the next 100 generations they play twice, and so on. From the 1500th generation we limit the number of played games on each map to 15 for the purpose of efficiency (so one agents is evaluated on 150 games in total). We also stop a game if we detect that an agent enters a cycle, what dramaticaly speeds up the evaluation of low quality solutions.

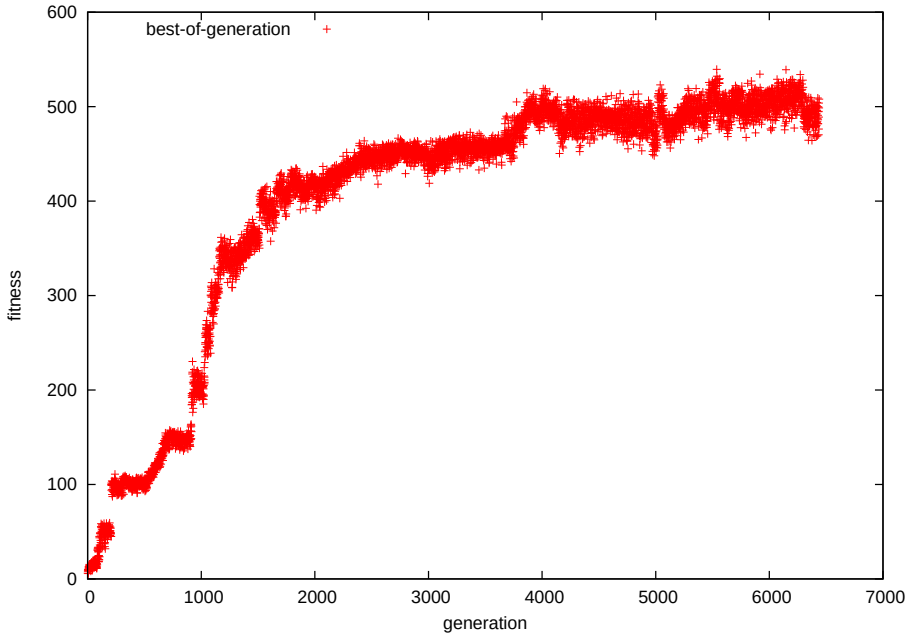

Figure 2: The fitness graph of the evolutionary run that produced our contestant.

The evolutionary runs were carried out on a cluster of computers. The machines were organized in star architecture, with the master running the evolutionary process (maintaining population, performing selection, crossover, and mutation), and the slaves carrying out the most time-consuming operations: genotype-phenotype mapping (i.e., building the neural net by querying the CPPN) and running the game simulations. The communication between the master and the slaves relied on the TCP/IP protocol. As only the genotype (CPPN) is sent over the network, the network load is reasonable: a typical CPPN evolved in our experiments has no more than a few dozens of nodes, so it is much more compact than the 20-thousand-weight neural net it produces.

We carried out a series of evolutionary experiments, with the number of slaves varying from 6 to 64 in the main experiment that evolved the submitted contestant. A single evaluation cycle (genotype-phenotype mapping plus 150 game simulations) of a good agent in the 2000th generation took about 170s on a 3GHz PC slave machine. Thus, a simulation of a single game takes slightly more than one second.

7 Experiment results

The evolutionary runs were able to maintain steady progress. In Figure 2 we can observe several rapid advances in fitness of the best-of-generation agents.

Figure 3 shows the track of our contestant on one of the maps. The cells marked by '#' are the obstacles. The empty cells have not been visited by the agent. The numbered cells have been visited by the agent, and the number tells how many times it happened. If the agent visited a cell more than 9 times, the cell contains digit 9.

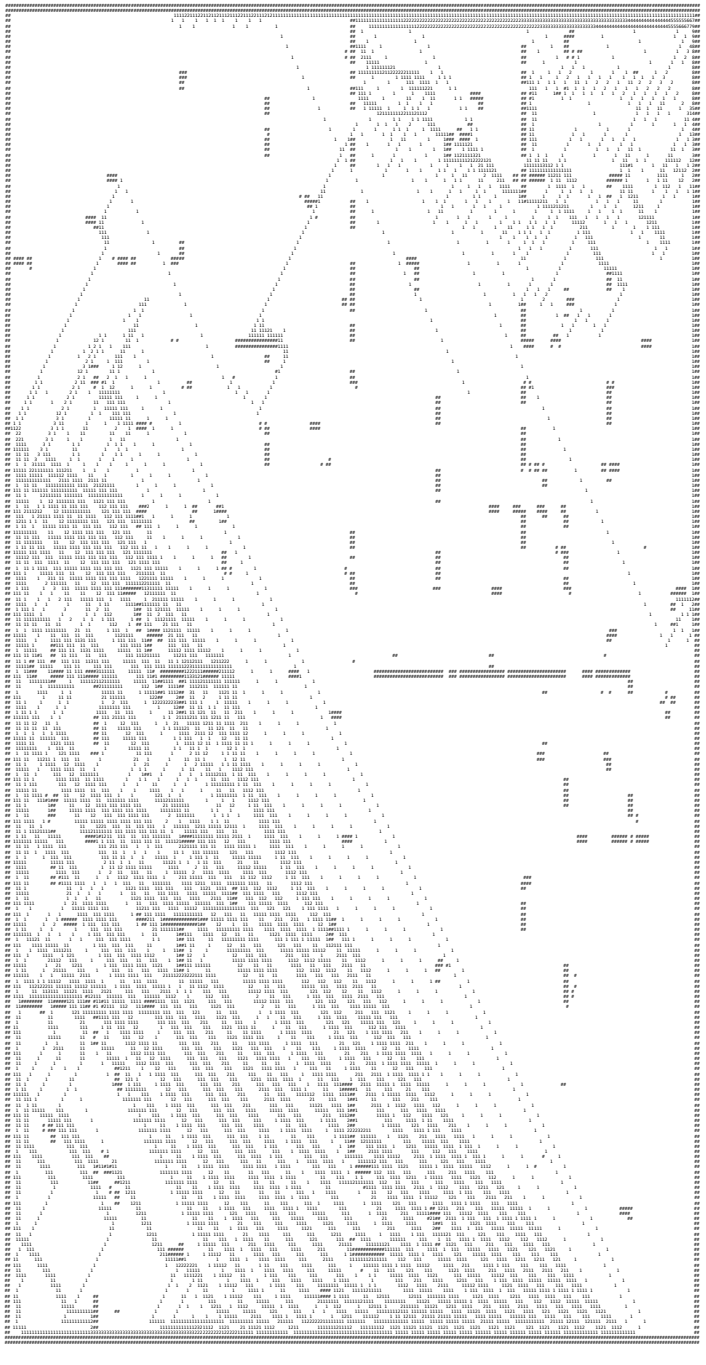

Figure 3: Track 10

Appendix A: HyperNEAT parameter settings

PopulationSize 20.0
MaxGenerations 100000.0
DisjointCoefficient 2.0
ExcessCoefficient 2.0
WeightDifferenceCoefficient 1.0
FitnessCoefficient 0.0
CompatibilityThreshold 6.0
CompatibilityModifier 0.3
SpeciesSizeTarget 20.0
DropoffAge 15.0
AgeSignificance 1.0
SurvivalThreshold 0.2
MutateAddNodeProbability 0.03
MutateAddLinkProbability 0.3
MutateDemolishLinkProbability 0.00
MutateLinkWeightsProbability 0.8
MutateOnlyProbability 0.25
MutateLinkProbability 0.1
AllowAddNodeToRecurrentConnection 0.0
SmallestSpeciesSizeWithElitism 5.0
MutateSpeciesChampionProbability 0.0
MutationPower 2.5
AdultLinkAge 18.0
AllowRecurrentConnections 0.0
AllowSelfRecurrentConnections 0.0
ForceCopyGenerationChampion 1.0
LinkGeneMinimumWeightForPhentoype 0.0
GenerationDumpModulo 10.0
RandomSeed 2.0
ExtraActivationFunctions 1.0
AddBiasToHiddenNodes 0.0
SignedActivation 1.0
ExtraActivationUpdates 9.0
OnlyGaussianHiddenNodes 0.0

References

[1] David B. D'Ambrosio and Kenneth O. Stanley. Generative encoding for multiagent learning. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2008)*, New York, NY, 2008. ACM. (to appear).

[2] Jason Gauci and Kenneth O. Stanley. Generating large-scale neural networks through discovering geometric regularities. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, New York, NY, 2007. ACM.

[3] Jason Gauci and Kenneth O. Stanley. A case study on the critical role of geometric regularity in machine learning. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-2008)*, Menlo Park, CA, 2008. AAAI Press. (to appear).