# Social Learning Strategies Tournament II Entry: BlueGenes

G-J van Rooyen
Dept. E&E Engineering, Stellenbosch University, South Africa

28 February 2012

## 1 Overview

*BlueGenes* is a genetically programmed entry to the Social Learning Strategies Tournament II. The agent script is mostly machine generated, based on parametric trait algorithms. The solution experiments with a new idea in genetic programming, where evolution is not based on a tree of primitives, but rather on a state graph of parametric algorithms (traits) and transition conditions that can be suggested by the designer.

The entry itself is largely machine generated, and should ideally not be debugged directly. The `move()` function is split into eight parts, corresponding to all possible combinations of the `canChooseModel`, `canPlayRefine` and `multipleDemes` options. Each algorithm inside such a combination has been individually evolved to best fit the constraints.

Each algorithm is defined as a state graph, which allows an individual agent to change its behaviour over its lifetime. This is achieved by defining each state's completion criteria (`*_done()` function) based on only its observed history, and by selecting a current state based on previous states' exit criteria and exit rounds. Since no local information can be stored by an individual, this information must be built up by the individual during each invocation, by using the provided history lists.

Furthermore, some traits allow individuals to diversify across their current population. This is achieved by playing a random move during certain key moments in an individual's history (usually in terms of its state graph). These random choices can be detected in subsequent rounds, and be used to individualise the agent's behaviour, depending on that initial choice.

Although the provided agent script is quite long (around 2500 lines of code), much of the code is traits repeated across the eight mode combinations. It also includes dormant traits that the solution acquired over its evolution, but which may not ever be reached in this solution. These have been left untrimmed, since they contribute very little overhead, and the machine-generated code was rather left intact to avoid the manual introduction of errors. Despite the long code length, an agent will quickly hone in on the relevant code segment during its round, and the average time taken per round was found to be on par with the reference solution given in the tournament rules.

## 2 Traits

In searching for model solutions, the genetic program randomly combines self-contained algorithms provided by the programmer. These algorithms are called *traits*, and each trait represents a state in an evolved agent's state machine. During this state, the trait's algorithm defines the sequence of moves that are played.

Traits' algorithms may depend on parameters that can be optimised over generations, called *evolvables*. Although a programmer may provide typical starting values, these parameters are evolved over time to improve the fitness of the agent.

Since traits represent stages in an indiviual agent's state machine (i.e. an agent can move through many states in its lifetime, each represented by a unique trait), the trait definition should include a `*_done()` function to indicate the conditions where a state can transition to a next state. States may have multiple output conditions, each representing a different outgoing edge on an agent's state graph, and the `*_done()` function should indicate which edge to to take when the state finishes.

This section will provide a brief description of the various traits used to evolve the submitted solution.

## 2.1 Pioneering

This is an initial state, which means that it can only occur at the entry point of an agent's state graph (and consequently at the start of an individual agent's life). The agent plays OBSERVE on its first round. If it observes no other agents playing EXPLOIT, it assumes that it is part of the cohort of pioneers – the very first group of agents alive in a simulation. It then proceeds to play INNOVATE for a number of rounds before exiting the state. If it determines that it is not a pioneer (i.e. it observed other agents playing EXPLOIT) it exits this state immediately.

**Evolvables:**

- `N_rounds`: The number of rounds a pioneer should play INNOVATE.

## 2.2 PioneeringBi

This trait is identical to *Pioneering*, except that it defines two exit conditions: if it determines that the agent is a pioneer, it playes `N_rounds` of INNOVATE, and exits via the first output edge. Otherwise, it exits via the second output edge.

## 2.3 DiscreteDistribution

The simplest strategy: in this state, the agent randomly plays moves, based on an evolvable distribution. It is a terminal state, which means that it has no output edges. This trait has several duplicates (*DiscreteDistributionB*, *DiscreteDistributionC*, etc.) which allows the state to occur multiple times in an agent's state graph.

**Evolvables:**

- `Pi, Po, Pe, Pr`: Weights defining the probability that INNOVATE, OBSERVE, EXPLOIT and REFINE are played. The weights need not be normalised (i.e. they need not sum to unity).

## 2.4 Specialisation

This single-round state randomly plays a move based on an evolvable distribution. It then exits on one of four output edges, corresponding to the move chosen. This allows agents across the population to diversify (i.e. move into diverse subsequent states) based on the random distribution.

This trait has a duplicate (*SpecialisationB*) which allows it to occur twice in an agent's state graph.

**Evolvables:**

- `Pi, Po, Pe, Pr`: Weights defining the probability that INNOVATE, OBSERVE, EXPLOIT and REFINE are played. The weights need not be normalised (i.e. they need not sum to unity).

## 2.5 ExploitGreedy

Exploit the maximum-payoff state until the payoff drops below its initial value (increases in payoff are happily ignored). Then, exit the state (possibly returning again later). This trait has no evolvable parameters. It has a duplicate (*ExploitGreedyB*) which allows it to occur twice in an agent's state graph.

## 2.6 Study

The perfect counterpart to *ExploitGreedy*, the *Study* state allows an agent to learn new acts for a few rounds. It plays INNOVATE, OBSERVE and REFINE from an evolvable distribution for a number of rounds, and then exits.

This trait has a duplicate (*StudyB*) which allows it to occur twice in an agent's state graph.

**Evolvables:**

- `N_rounds`: The number of rounds to remain in this state.
- `Pi, Po, Pr`: Weights defining the probability that INNOVATE, OBSERVE and REFINE are played. The weights need not be normalised (i.e. they need not sum to unity).

## 2.7 InnovationBeat

This is the most intricate trait. Its premise is that a population of agents could maximise information exchange if they could synchronise their moves in some or other way. Since there is no absolute reference of round numbers for the individual agents, and the only way that agents can learn about their peers' actions is through the OBSERVE action, *InnovationBeat* attempts to synchronise agents' behaviour by forcing regular rounds where no EXPLOIT acts can be observed. This is achieved by letting agents who have already synchronised, play INNOVATE at regular intervals. New agents can play OBSERVE for a while, and try to detect the round at which no (or few) models are observed; this is assumed to be the INNOVATE synchronisation beat. The agent then falls into this rhythm.

This behaviour is achieved by evolving two sequences of moves. Each sequence starts with the INNOVATE move; the next move (EXPLOIT / OBSERVE) is played randomly, which effectively splits the population into two groups. Group A continues to play the one sequence, and Group B the other. The remaining acts in each sequence are chosen from {EXPLOIT, OBSERVE, REFINE} so that the two groups never play OBSERVE simultaneously. Ideally, one group will learn from the other at some points in the sequence.

When an agent enters the *InnovationBeat* state, it plays OBSERVE for `N_Seq` moves, picks the round when the lowest payoff was observed as the synchronisation round, and then picks a group (A or B) and starts playing its sequence, synchronised to the rest of the population.

*InnovationBeat* is a terminal state; once an agent has entered it, it remains in this state for the rest of its lifetime.

**Evolvables:**

- `N_Seq`: The length of a sequence (i.e. the number of rounds between synchronised INNOVATE moves).

- `seq_A, seq_B`: The evolved move sequences.
- `Pa`: The probability of an agent picking Group A as its own.

### 2.8 InnovationBeatSpatial

This variation on *InnovationBeat* takes the *multipleDemes* scenario into account. If an agent migrates across demes, its earlier synchronisation is no longer valid. In this case, the state exits (potentially re-entering itself immediately to resynchronise, or perhaps spending a few rounds in the *Study* state).

## 3 Submission

The tournament submission includes:

- This document, which provides an overview of the solution.
- `BlueGenes.py`, the submitted solution.
- `individual_modes.zip`, for information only. This contains the sub-solutions that were merged into `BlueGenes.py`. The filename of the sub-solution indicates whether it applies to *canChooseModel* ('O'), *cumulative* ('R'), *multipleDemes* ('D') or some combination thereof.
- The full genetic programming framework, including a reference simulator that implements the tournament rules to measure an agent's fitness (defined as average total payoff per round) is available at https://github.com/gvrooyen/SocialLearning. 

## 4 Declaration

By submitting this entry, I hereby accept the rules contained in the "Rules for entry" document provided by the organisers on the tournament website.