IGGI GD2 Lab Exercise: Game Parameter Tuning

Introduction

The purpose of this lab is to gain experience of using a sample-efficient optimiser (N-Tuple Bandit Evolutionary Algorithm – the NTBEA) to tune game parameters.

The lab uses a purpose-designed tap-timing game called CaveSwing. The aim of the game is to swing through a cave as fast as possible using magic elastic rope (a bit like SpiderMan, but faster to attach), finishing at the end as high as possible, but it's game over if the avatar touches the roof or the floor.

The lab script should be doable within a couple of hours.

In the lab you will:

- Download the software (all in Java) from github (5 mins)
- Play the CaveSwing game using the keyboard (Space Bar to swing), and experiment with the effects of different parameter settings (10 mins)
- Run the NTBEA on a toy problem to get a feel for setting the exploration parameter (15 mins)
- Study the code in the package `cavegame` and run it with a random player (20 mins)
- Learn how to prepare a game for parameter optimisation (20 mins)
- Tune the game to maximise the score achievable by a random player (5 mins)
- Tune the game to maximise the difference between a skilful player and a random player (30 mins)

The tuning will be based on measures of player or game activity. We have to turn all these measure into a single number which aims to reflect how well the game satisfies a set of design objectives. This is a type of single-objective optimisation; multi-objective versions would also be possible, but for now we just do single.

Downloading and Running the Software

Download the software from the following Github:
https://github.com/ljialin/SimpleAsteroids

(we suggest you download the .zip file)

Open the project in your favourite Java IDE (we recommend Intellij Idea).

This is a large software repository with hundreds of classes, but you only need be concerned with some of them. All the files specific to the CaveSwing implementation are in package caveswing.

Open the file **`caveswing.test.KeyPlayerTest`**.

You'll see the first few lines of the main methods are something like this:

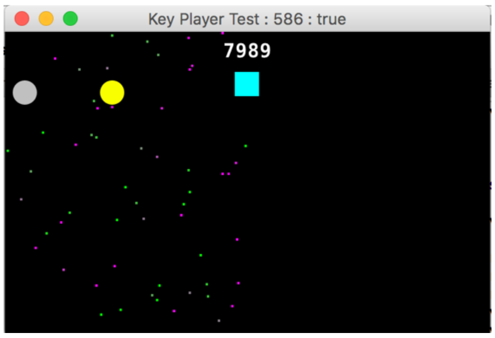
```
CaveSwingParams params = new CaveSwingParams();
// todo: adjust parameter setting and see how they affect game play for you
params.maxTicks = 5000;
params.gravity.y = 0.03;
params.gravity.x = -0.03;
CaveGameState gameState = new CaveGameState().setParams(params).setup();
CaveView view = new CaveView().setGameState(gameState).setParams(params);
```

The exact details may differ (versions may include various parameter settings)

Try playing the game, and observe the effects of setting smaller and larger values for gravity.y and Hooke's constant. Go to the **`CaveSwingParams`** source file to see the full set of parameters. Also try changing the number of anchors (nAnchors). Other things to try changing: the height of the game, the width of the game, and the scrollWidth (the latter is currently not a game parameter, but is set in CaveView). Note that CaveView has a boolean variable to select whether a side-scrolling view is shown, or the entire game window (you should be able to find it).

How do these affect the difficulty of the game for you? Can you find settings that make it easy and other ones that make it hard, but still possible?

Here is a successfully completed game run with a score of 7989, completed in 586 steps (the avatar is the light blue square).



Running the NTBEA

Open the Java file: **`ntbea.TestNTBEA`** and read the code.

The first part sets up the problems to solve:

```
int nDims = 5;
int mValues = 4;
double noiseLevel = 1.0;
boolean useTrap = true;
// EvalMaxM is like Noisy OneMax but generalised to M values
// instead of binary
EvalMaxM problem = new EvalMaxM(nDims, mValues, noiseLevel).setTrap(useTrap);
```

EvalMaxM is set up as a simple noisy optimisation problem in a space of nDims dimensions, with mValues possible in each one, ranging from 0 to (mValues-1) . The objective function sums the value in each dimension, then adds Gaussian noise. Hence, this would have a maximum possible score of 15, plus whatever the value of the random noise on that sample. *But, there is a catch!* If we set a trap, then picking the maximum value in each dimension will give a score of zero, plus noise.

The next part sets up the algorithm and its landscape model: here we choose which N-Tuples to use. Usually a good choice is to use 1, 2 and N, as done here:

```
double kExplore = 2;
double epsilon = 0.5;
NTupleBanditEA banditEA = new
NTupleBanditEA().setKExplore(kExplore).setEpsilon(epsilon);
// set a particlar NTuple System as the model
// if this is not set, then it will use a default model
NTupleSystem model = new NTupleSystem();
// set up a non-standard tuple pattern
model.use1Tuple = true;
model.use2Tuple = true;
model.use3Tuple = false;
model.useNTuple = true;
banditEA.setModel(model);
```

Run the code with the default values as given.

The printout gives important insight in to how the algorithm operates.

Here is a sample: this shows the statistics for each value in the first dimension. Each line shows the number of times that value was sampled, the mean score, and the standard deviation (NaN if it was only sampled once).

```
Search space has 5 dimensions
nPatterns observed: 4
[0]	20	8.881369161756407	2.3513275886841236
[1]	1168	11.849403698374624	1.2077887654641586
[2]	3801	12.4520200256144	1.2109922025655393
[3]	11	8.48488382426213	3.349187617971695
```

The next sample shows the stats for a pair of dimensions:

```
nPatterns observed: 16
[0, 0]		1	4.7485643752832045	NaN
[0, 1]		1	5.408243889054641	NaN
[0, 2]		1	7.682158474608535	NaN
[0, 3]		3	8.208609435327174	2.3225032666139214
[1, 0]		1	6.294616649742344	NaN
[1, 1]		1	8.898631862686198	NaN
[1, 2]		8	11.071365574169855	1.3085159817068766
[1, 3]		17	11.262404434138096	1.6402612648098414
[2, 0]		4	9.754357939614362	1.3764461343115553
[2, 1]		1	4.295630195535133	NaN
[2, 2]		171	12.619864236869907	1.3203726242410123
[2, 3]		1353	13.541031906042782	1.1353798743642982
[3, 0]		8	10.256045625853481	1.8752923104012205
[3, 1]		12	11.064462069796205	1.413810122726862
[3, 2]		1258	13.536381085813213	1.1672662457921197
[3, 3]		2160	13.761778366146128	1.11939108306784
```

Note the big discrepancy in the number of samples made for each pair of values, with some being sampled once or twice, others thousands of times.

Quickly scan over the printout, and see what else you can observe, especially around the maximum index values [3], [3, 3] and [3,3,3,3,3].

Now run the code again, but setting kExplore value higher – rerun with values of 20 and of 200. What do you observe?

Study the CaveGame code

You don't have to read every line, but look at the packages and classes in each one. In particular pay attention to the CaveSwingParams class, and also the CaveGameState class.

CaveGameState stores all the state information, and implements the AbstractGameState interface. Storing all the state variables in a single copyable object makes it easy to run statistical forward planning algorithms.

Now run the class: **`caveswing.test.CaveGameSpeedTest`**

The entire code is shown below:

```java
package caveswing.test;

import agents.dummy.RandomAgent;
import caveswing.core.CaveGameFactory;
import ggi.tests.SpeedTest;

public class CaveGameSpeedTest {
    public static void main(String[] args) {
        SpeedTest speedTest = new SpeedTest().setGameFactory(new CaveGameFactory());
        speedTest.setPlayer(new RandomAgent());
        int nGames = 10000;
        int maxSteps = 1000;
        speedTest.playGames(nGames, maxSteps).report();
    }
}
```

How long does it take to run?

Observe the stats. How many game ticks per second are executed? Try with and without game copying per tick (you'll have to set the appropriate flag in the SpeedTest class.

Optional Exercise: (10 mins)

Modify the code to use non default parameters and observe the effects on the game stats. *Hint: Create a CaveSwingParams object, just like in KeyPlayerTest, then pass that to the CaveGameFactory by calling its setParams method.*

Can you set up a game which is easy or hard for the random player?

Preparing a Game for Optimisation

There are two parts to this:

1. Preparing a search space
2. Implementing an evaluation function

We'll take them in turn.

In the code, we use the concept of an AnnotatedFitnessLandscape.

This is a model of a discrete search space: each dimension has a name (hence it is annotated), and a number of possible values. The values are selected via an index. The easiest way to understand this is by studying the code.

The set of possible values for each parameter is given in an array of the appropriate type. Currently there are parameter types for Double, Integer and Boolean.

```java
public Param[] getParams() {
    return new Param[]{
            new DoubleParam().setArray(xGravity).setName("x Gravity"),
            new DoubleParam().setArray(yGravity).setName("y Gravity"),
            new DoubleParam().setArray(hooke).setName("Hooke's constant"),
            new DoubleParam().setArray(lossFactor).setName("Loss factor"),
            new DoubleParam().setArray(anchorHeight).setName("Anchor height"),
            new IntegerParam().setArray(nAnchors).setName("Number of Anchors"),
    };
}

double[] xGravity = {-0.02, 0.0, 0.02};
double[] yGravity = {-0.02, -0.01, 0, 0.01, 0.02};

double[] hooke = {0.0, 0.005, 0.01, 0.02, 0.03};
double[] lossFactor = {0.99, 0.999, 1.0, 1.01};
double[] anchorHeight = {0.2, 0.5, 0.8};
int[] nAnchors = {5, 10, 15};

int[] nValues = new int[]{xGravity.length, yGravity.length,
        hooke.length, lossFactor.length, anchorHeight.length, nAnchors.length};
int nDims = nValues.length;

static int xGravityIndex = 0;
static int yGravtityIndex = 1;
static int hookeIndex = 2;
static int lossFactorIndex = 3;
static int anchorHeightIndex = 4;
static int nAnchorsIndex = 5;
```

In this case we define six parameters, from xGravity through to nAnchors.

It is fine to experiment with different possible values.

On to the second part: evaluation.

The first step is to unpack the game parameters from the candidate solution, and inject them into the parameters used to run the game. The following lines do this:

```
public double evaluate(int[] x) {

    // bundle extract the selected params from the solution vector
    // and inject in to the game design params

    GameRunnerTwoPlayer gameRunner = new GameRunnerTwoPlayer();

    // set up the params
    CaveSwingParams params = new CaveSwingParams();

    params.gravity = new Vector2d(
            xGravity[x[xGravityIndex]],
            yGravity[x[yGravtityIndex]]
    );

    params.hooke = hooke[x[hookeIndex]];
    params.lossFactor = lossFactor[x[lossFactorIndex]];
    params.meanAnchorHeight = params.height * anchorHeight[x[anchorHeightIndex]];
    params.nAnchors = nAnchors[x[nAnchorsIndex]];
```

Next we have to now run a game (or a set of games) – and measure something. Working out the best measure to optimise can be hard. Let's start with something easy: we'll optimise the game to make it easy for a random player.

To do this, we'll now set the parameters in to the game (in fact a Game Factory – and object to make a fresh game each time). Then we'll run the game using a random player, and return its score (or average score, to cover running multiple games).

This is done as follows:

```
    // using a Game Factory enables the tester to start with a fresh copy of the
    // game each time
    CaveGameFactory gameFactory = new CaveGameFactory().setParams(params);
    SpeedTest speedTest = new SpeedTest().setGameFactory(gameFactory);
    speedTest.setPlayer(new RandomAgent());

    speedTest.playGames(nGames, maxSteps);

    if (verbose) {
        System.out.println(speedTest.gameScores);
    }

    double value = speedTest.gameScores.mean();
    // return this for now, and see what we get
    logger.log(value, x, false);
    return value;
}
```

Note: the way the code is set up, every new fitness value **must be logged in the logger**: the logger is used to measure the number of calls to the objective function, and if we don't log them the optimiser will not realise that it's evaluation budget is all used up (*room for improvement in the code there*).

In order to run an optimisation trial, run the code in `hyperopt.TuneCaveSwingParams`. Experiment with different values of 'k'. Each optimisation run using a random player should be swift – of the order of a few tens of seconds,

Tuning to Reward Skilful Play

Now modify the evaluation function so that the value returned is the average score for a skilful player minus the average score for a random player.

In order to see a skilful bot in practice, view run the code in `test.EvoAgentVisTest`.
This shows a Rolling Horizon Evolution agent playing the game (usually with some success). Note that code has a method called getEvoAgent(). You can use this to create a skilful agent.

Now create a new version of `design.CaveSwingGameSearchSpace` based on the difference between the scores for the skilful agent minus the scores for the random agent. Note that evaluating this objective function will be slower than before, so set the number of games to play (nEvals) to a lower value.

A possible solution is shown in `solutions.CaveSwingGameSkillSpace`. Note the number of games played per evaluation.

You can modify the **`hyperopt.TuneCaveSwingParams`** to use this solution by uncommenting this line:

```
AnnotatedFitnessSpace caveSwingSpace = new CaveSwingGameSearchSpace();
// uncomment to run the skilful one
// caveSwingSpace = new CaveSwingGameSkillSpace();
```

The program will now take a minute or two to run, depending on the power of your machine. After running, perform a couple of further tests. Observe the final solution values, and try running the tests.KeyPlayerTest game with them. How well does the resulting game play?

Analyse the printout of the landscape model. There are some interesting things to note. Can you explain the score of zero below for index [0] of Hooke’s constant?

```
Hooke's constant

[0]    5      0.0
[1]    22     6053.905051831373
[2]    34     7142.263864254699
[3]    63     8129.168253660932
[4]    76     8212.8840759315
```

Summary

This lab has covered setting up a game design search space, and then searching it using two objective functions: to maximise the score a random player can achieve, and to maximise the score difference between a skilful player and a random player.

It is interesting to note that the skilful player is a general agent that had no knowledge of the game it was playing, other than through methods of the **`AbstractGameState`** interface.

This makes the approach very general, providing the game to be optimised can implement fast copy and next state functions.