Assignment 3: General Games - Part 2

Core Body of Knowledge classification (http://tinyurl.com/acscbok): Abstraction (5), Design (5), Teamwork concepts & issues (3), Data (5), Programming (5), Systems development (3)

Due date: Week 12, Weight: 10 % of the course

1 Overview

Assignments should be done in groups consisting of five to six (5–6) students. Please use the Groups feature in myUni. If you have problems finding a group use the Discussions forum to search for group partners.

Each student has to take major responsibility for one of the exercises and collaborate with the team members on the remaining exercises. Each exercise needs 1–2 students taking major responsibility. For Exercise 4, you should have 2–3 students taking major responsibility. The group has to make sure that the workload is evenly distributed.

2 Assignment

This assignment uses the General Video Game AI framework (GVGAI). Detailed information about the competition and GVGAI is given at http://www.gvgai.net/. You can get the code at https://github.com/GAIGResearch/GVGAI. **You can only use Java for this assignment.**

Assignment 3 requires that you compute and test sequences of inputs to be played for each game level. We only consider the **Single Player Planning Track** in this assignment. For a quick start, take a look at **src/tracks/singlePlayer/Test.java** for an example on how to use the framework.

In this assignment, you have to consider the following 4 games: **Bomber, Boulderchase, Chase, Garbagecollector**. These games are deterministic, meaning performing a sequence of inputs results in the same outcome every time. Note that each game includes 5 levels (from "lvl0" to "lvl4"), so you will be looking at 20 game levels in total.

Since a sequence of inputs can be simulated using the **advance** function (see "One Step Look-Ahead" controller) without having a game instance running, there will be no wall-clock time constraint. Such a sequence is to be applied to the initial game state, and its score is measured after applying the last input, or when a terminal game state (win or lose) is reached during the simulation, whichever comes first. The score should be taken directly from the game state without modification from heuristics.

For this assignment, we consider a call to the **advance** function to be the time unit for the benchmark, as it is the bottleneck operation in the optimisation. Note that if the simulation of a given sequence reaches a terminal game state, subsequent steps will not change the game state, or incur computational cost. This means different sequences can have different evaluation costs, which must be accounted for in the termination criteria of your algorithms.

Exercise 1 *Team work: who has done what? (zero points)*

Just like in Assignment 1, we'd like each team member to write one paragraph about what he or she has contributed to this assignment. We will not mark this, and it will not have any effect on the marking of the other exercises. You might now ask, "why do this then?" – well, through this no-stakes approach, we'd like to encourage self-regulation within the group and cooperative learning. You can't lose; you can only win.

Exercise 2 *Single-objective inputs optimisation (20 points)*

For this exercise, you have to evolve a sequence of input that maximises the score on a given game level.

1. Design an evolutionary algorithm to compute an input sequence maximising the score on a game level. Describe and justify its design in the report. It should make use of (and count the number of calls to) the **advance** function. You may use existing techniques if appropriate.
2. Run the algorithm on each of the 20 game levels 10 times (or more) with 5 000 000 calls to **advance** each time, and report for each game level the average and standard deviations of scores after 200 000, 1 000 000, 5 000 000 calls.

Notes:

- You may have to implement a method to obtain the initial game state. For a quick start, check the **runOneGame** function in **ArcadeMachine.java**. Note that the **prepareGame** function in **Game.java** may need to be run to have the game state properly initialised.
- You can replay an input sequence with the **replayGame** function in **ArcadeMachine.java**, given that the sequence is stored in a certain format. Alternatively, you can implement your own replaying function. One the other hand, for efficiency, your algorithm should use the game state object directly.
- If the 200 000th call occurs during, and not at the end of, an iteration of the algorithm, record the score achieved in the previous iteration. The same applies to 1 000 000th, 5 000 000th calls. For population-based algorithms, "one iteration" refers to "one generation" (i.e. one iteration of the generational loop).
- If the final sequence is stored after each run, remove redundant inputs after the one where the terminal state is reached (if applicable).

Exercise 3 *Multi-objective inputs optimisation (30 points)*

This exercise extends Exercise 2 by considering an additional objective: minimising sequence's length. Here, "length" refers to the effective part of the sequence, i.e. not counting inputs after terminal state (if reachable).

1. Design a bi-objective evolutionary algorithm to compute input sequences maximising the score on a game level, while being as short as possible. Describe and justify its design in the report. It should make use of (and count the number of calls to) the **advance** function. You may use existing techniques if appropriate.
2. Run the algorithm on each of the 20 game levels 10 times (or more) within 1 000 000 calls to **advance** each time, and report for each game level the averages and standard deviations of hypervolumes after 200 000, 1 000 000, 5 000 000 calls.

Notes:

- The same convention in Exercise 2 applies regarding the how scores are recorded and how sequences should be stored.
- You can specify your own reference points to compute the hypervolumes. While different reference points may be used for different games, the same point must be used for all levels within a game. Include the reference points you choose in the report.
- The choice of hyperparameters (e.g. population size) is up to you. Include these in the report.

Exercise 4 *Level design (50 points)*

In this exercise, you are required to evolve levels of a game. You are to choose one existing game for this task, and it does not have to be among the 4 games mentioned above.

- Briefly describe the game, including sprite interaction rules, winning and losing conditions, and available inputs.
- Decide on at least 2 objectives to optimise. Additionally, define the constraints on the game levels as appropriate. For instance, some games require the levels be surrounded by wall sprites.
- Design a multi-objective evolutionary algorithm to evolve levels optimising the objectives. Describe and justify its design in the report.
- Run the algorithm 10 times (or more), and report the averages and standard deviations of hypervolumes of the final outcomes.
- Include screenshots of the initial game screens from 10 (or more) final levels in the report, along with their respective objective values. These levels should represent the Pareto front well. Discuss the results in the context of the chosen objectives.

Notes:

- The chosen objectives should be justifiable, and they need to conflict with each other to some degree to make for interesting trade-offs.
- For this exercise, you may choose to run with any parameter setting, including run-time budget. Describe the chosen setting in the report.
- To take the screenshot of the first frame of the game, you can try running the game with a human player.

3 Procedure for handing in the assignment

Work should be handed in using the course website. The submission must include:

- pdf file of your solutions for theoretical assignments
- all source files (if you created any): if your code does not compile or if it is not sufficiently well documented, **we will cap the code-related marks at 50%**
- descriptions and additional files as required in the statement of the exercises
- a file name README.txt that contains instructions to run the code (if any), the names, student numbers, and email addresses of the group members
- for each group, there should be only 1 submission

Failure to meet all requirements of the 'General procedure for handing in the assignment' will lead to a reduction by twenty (20) marks.

Note: there will be a progress presentation session for this assignment. This is a big opportunity for you to get feedback on your progress, and to make last adjustments for the final submission. In these progress presentations, we are expecting each group to briefly outline their achievements. You should not repeat what the assignment is about – we all should know this by then. Outline your approach, your results (screenshots), and tell us about your challenges – maybe we can help you right away.

The following link contains general information that can be helpful:

- Feedback that we have given in the past:

  `https://cs.adelaide.edu.au/~markus/teaching/feedback.txt`

  (not everything is applicable here)