Tartarus – Problem and Code

In this homework, you will explore using genetic programming to evolve the behavior of a very simple robot in a very simple simulation. Following is a description of the Tartarus problem itself and detailed descriptions of how to use and modify the provided code that implements a GP algorithm to find solutions to the Tartarus problem.

Tartarus – the problem

The Tartarus problem is described in the first few pages of the Tartarus pdf on Moodle. One caveat is that the GP code you will be using expects fitness to be minimized instead of maximized, so the fitness calculation is changed. In the original problem,

origFit = number of block edges touching a wall

The maximum possible fitness using this equation is

maxFit = 2(number of blocks on grid)* if num blocks < 4

maxFit = 4+(number of blocks on grid) if num blocks >= 4

The new fitness equation to convert to a minimization problem is

fitness = maxFit – origFit + 1

The +1 is to avoid fitnesses of 0. You will use only tournament selection, so this shouldn't be a problem, but better to be safe. In the default experiments with 6 blocks on a grid, the fitnesses will range from 1 (best) to 11 (worst).

Tartarus – using the code

The GPJPP code is courtesy of Eric Siegel, with minor modifications. The Tartarus1 and Tartarus2 code was written by me, with code for printing files needed for visualization added by former Carleton students Emily Bauer, Emily Johnston, Laurel Orr, and S'ade Smith. The RobotDraw code was written entirely by those students (which is awesome and I thank them for making the project way more fun).

The Tartarus1 folder contains an implementation as described in the paper using the terminals/functions listed in tables 10.1 and 10.2 (with the exception of the random terminal). The Tartarus2 implementation is more like the Sante Fe Ant problem we discussed in class. The terminals are the 3 possible dozer movements (left, right, forward). The functions are if/elif/else style functions with 3 children of which precisely one will be evaluated. If the sensor returns a 0 (indicating an empty cell), the left child is evaluated. A return of 1 (indicating a box in the cell) causes the middle child to be evaluated, and a return of 2 (indicating a wall) means the right child is evaluated. A single tree will be evaluated completely before starting over at the root again, taking the appropriate dozer action each time a leaf node is reached, so there may be many dozer movements (steps) in the execution of one individual tree.

<u>To compile and run</u>

Navigate in terminal to inside Tartarus1 or Tartarus2 and type the command

```
javac –cp "../:." *.java
```

The *–cp* option (short for *–classpath*) tells Java to look in all of the paths given (: separates paths), or in this case one directory back and the current directory. To run use the same classpath option and Tartarus as the file with the main function. You must have an existing *data* folder inside the Tartarus1 (or 2) folder, and you should include a command line argument that will be the basename of the input file and all data files written to the data folder. If you forget, "tartarus" will be used as the basename. You must have a file *basename.ini* in the current directory with the input options. Any existing output files in the data folder with the same basename WILL BE OVERWRITTEN, so use a different basename each time you want new data without deleting the old (and copy the .ini file to one with that basename). For example, if I have my options in the file tartarus.ini, I would perform a new test run by using the commands

```
cp tartarus.ini t0.ini
java –cp "../:." Tartarus t0
```

This will copy my input options from tartarus.ini to an input file with the correct basename t0, read the file *t0.ini* from the current directory, and write the files *t0.stc*, *t0_simTime.txt*, etc. to the data folder.

Input file

The file *basename.ini* contains values for many options to use in an experiment.
Lines 1-27 set general gpjpp options. Documentation for these options is contained in *gpjpp.GPVariables.html* in the *docs* folder. Options you might want to change include lines 1, 2, 3, 6, 7, 8, 10, 13, 14, 18, 20, 21, 22, 23, 24, 27 (open the file in an app that shows you line numbers). I have not tested alternate values for any of the other gpjpp options; changes may just effect the nature of the evolution or they may completely break the program.
Lines 28-35 set Tartarus-specific options, all of which you may modify. These options are defined and described in code comments in the TartVariables.java file.

Output files

basename.stc: general data about the run, including

- list of the option values used
- the set of possible nodes
- general statistics for each generation
- data on the final best individual (may not be from final generation)
- real time for entire run and average per generation

basename1.gif: graphic representation of the parse tree for the final best genome.
basename_simTime.txt: used as input file to RobotSim visualizer, saves best genome each N generations, where N is set by *CheckpointGens* option.
basename_simBest.txt: used as input file to RobotSim visualizer, saves best genome from entire run.

Performing and saving data from multiple runs at once

Much of the actual day to day work in my research involves designing experiments, performing many runs with a given setup differing only in random number seed, and analyzing the data produced. The stochastic nature of EC means that you usually need to perform many experiments and look at the average or best result across all of them. You've seen the effect a good or bad initial population can have, so just performing one experiment and generalizing from that result is bad science.
To efficiently perform and analyze many runs, I write a lot of scripts, generally using bash to perform runs and Python to parse the resulting data. For your experiments in this HW, you might find some simple bash scripts helpful to run many tests. These are commands you can run directly in terminal, or save in a file and run that way. I will give you a few examples here, but I'm not going to go into lots of detail. However, these are useful skills and you should research on your own and feel free to ask me questions!

- Example script to loop through N experiments (replace N below with a number). This script assumes you have your starting options in the file *tartarus.ini* and the folder data exists. It will save the output files in the data folder, using the basenames t1, t2, … tN. To execute the script, type each line below and then press enter; when you press enter after the final "done" the script will run.

```
for i in {1..N}
do
cp tartarus.ini t$i.ini
java -cp "../:." Tartarus t$i
rm t$i.ini
done
```

- Same example all on one line – you can copy/paste this into terminal and the whole thing will run.
  ```
  for i in {1..2}; do cp tartarus.ini t$i.ini; java -cp "../:."
  Tartarus t$i; rm t$i.ini; done
  ```
- Finally, you can put this code into a file and run the file as a script. You will likely first need to set the permission of the file to allow you to run it as an executable, you can use chmod 744 to do this. For example, I put the code in a file named *test*. I run the command in terminal *chmod 744 test*. Now I can use the command *./test* to run the script.
- I will let you decide how to parse the output files most easily!

Visualizing simulated results

A group of Carleton students (named earlier) wrote a great tool that allows you to watch a visual simulation of the robot behavior for a given genome. To use this, you must run the original experiment with the visualization option turned on so that the necessary files will be printed. You then use these files as input to the simulator. Complete the following steps

1. Compile and run a single experiment of each Tartarus1 and Tartarus2 using the instructions given earlier. Just use the default settings in the provided .ini files. You should now have the files base_simTime.txt and base_simBest.txt in each data folder, where “base” is whatever you used as your base file name (“tartarus” if you didn’t include a command line argument).
2. Compile the RobotDraw code by navigating to inside the RobotDraw folder and using the command “javac *.java”.
3. Run the visualization from the RobotDraw folder using the command
   ```
   java RobotSim <sim_file> <TartType> <RobotType> <pause>
   ```
   The options are described below
   - *sim_file* is the name and path to one of the simulation input files (*base_simTime.txt* or *base_simBest.txt*)
   - *TartType* is 1 or 2 for which Tartarus folder was used to create the input file
   - *RobotType* chooses the image used for the robot: 1 for WALL-E or 2 for EVE
   - *pause* determines the speed of the visual simulation. It is number of seconds to pause between each robot action. A value in the range of .01 to .5 is works well.

Make sure you understand what the visualization is showing you for each type of input file. Note – the robots from both Tartarus1 and Tartarus2 will be rather terrible at performing the task, the initial GP representations are purposefully bad so that you can improve them!

Tartarus – modifying the code

Most of the code you will modify is in the Tartarus1 and Tartarus2 folders. Until you get to the mutation modification part of the assignment you should not touch the code in the other folders.

GPJPP

The gpjpp package provides all of the general code to use genetic programming to evolve solutions to a problem. It also provides abstract classes/functions that must be overridden for each specific problem the user wants solved. There is some documentation for this package (found in the docs folder), as well as comments in the code files themselves.

Tartarus

Each Tartarus folder provides an implementation of the Tartarus problem by extending the gpjpp base. Note that each class except Grid extends (inherits from) a base class, so if some class variable or function is used that you don’t see defined anywhere, it must be defined in the base class from gppjpp. Documentation for these implementations is here in this description, and in the code files themselves – there are lots of descriptive comments. The files include

- *TartVariables.java*: defines the Tartarus-specific options and methods to load/save/print them.

- *TartGene.java*: contains the important *evaluate* method that when called on an initial node, recursively evaluates the entire tree below that node, evaluating/executing each node as it goes.
    - Syntax note: the following line of code recurses down to evaluate child node "x", where x=0 is furthest left child, x=1 is child 2nd from the left, etc.

      ```
      ((TartGene)get(x)).evaluate(cfg, gp, os, out);
      ```

- *Tartarus.java:* creates the node set, contains the main function that sets up in/out file stuff and then actually starts the algorithm running.
- *TartGP.java:* handles overall evaluation of an individual genome. Creates and tests genome on many grids, calculates total & average fitness across all grids. Also includes functions to test genomes for file output purposes (not during actual evolution).
- *Grid.java:* implementation of actual Tartarus problem itself. Everything to set up and run a single simulation of a robot. Only file that does not inherit from anything in gppjpp, therefore flexible in how you may modify it. Everything in Grid is only used from Tartarus specific files, so any modification only requires syncing with those files, you don't have to worry about what's in gpjpp.

<u>Adding new node type</u>

You will add/modify several node types, both functions and terminals. To add a new node, you will take the following steps

1. Add the node with a reasonable name and the next available integer to the list of nodes at the top of *Grid.java*
2. Add the node to the list in *createNodeSet* in *Tartarus.java*. The first arg is the variable from *Grid.java*, the second is a string that will be printed when printing/drawing a genome tree, and the third is the arity if the node is a function (left out if a terminal).
3. VERY IMPORTANT - change the argument to *GPNodeSet* on line 33 of *Tartarus.java* to the new number of nodes!
4. Add a case to evaluate this node type in the *evaluate* method in *TartGene.java*.