Evolvers of Catan Technical Report

Bat-Orgil Batjargal, Alvin Bierley, Andrew Fitch, and Daniel Kleber

I JSettlers

JSettlers is a Java program that consists of Settler of Catan (SOC) playing bots and framework for other SOC-playing AIs to train. You can run the program locally from a machine of your choice. Also, it includes graphics enabling humans to play and visualize the games that bots are playing.

Because JSettlers is openly available to the public, it allows AI creators to compare their performances, track their improvement as a community and build a census around what works in creating SOC-playing AI, enabling replicability of the previous research. In short, JSettlers empowers the game AI community to advance itself and develop better AIs.

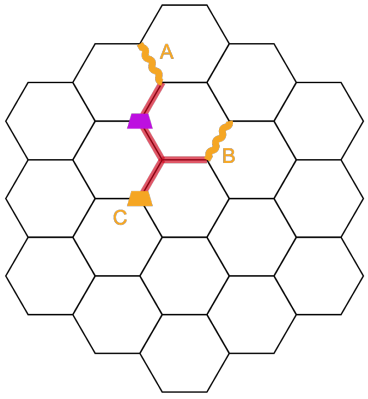
Current JSettlers provides two types of Heuristic bots to play with: fast and smart. Heuristic means they follow rules of thumb defined by humans to play SOC to a certain degree in their models.

Smart bot is mostly a better player than the Fast bot. Smart bot wins 30% of the time when it plays SOC with three Fast bots. Fast bot is named fast because it has a heuristic algorithm that is simpler and faster than that of the Smart bot. Smart bot is named smart because it was designed to win the Fast bot therefore smarter than the Fast bot that JSettlers originally had I assume.

In general, the Fast and Smart bots systematically analyze the game state, future dice probabilities, possible actions by itself and its opponents, and more to answer the question: which choice would increase its chance of winning?

We wanted to create an AI that can play against the Smart Bot and win. We extended the Smart bot.

The strategy of the smart bot:



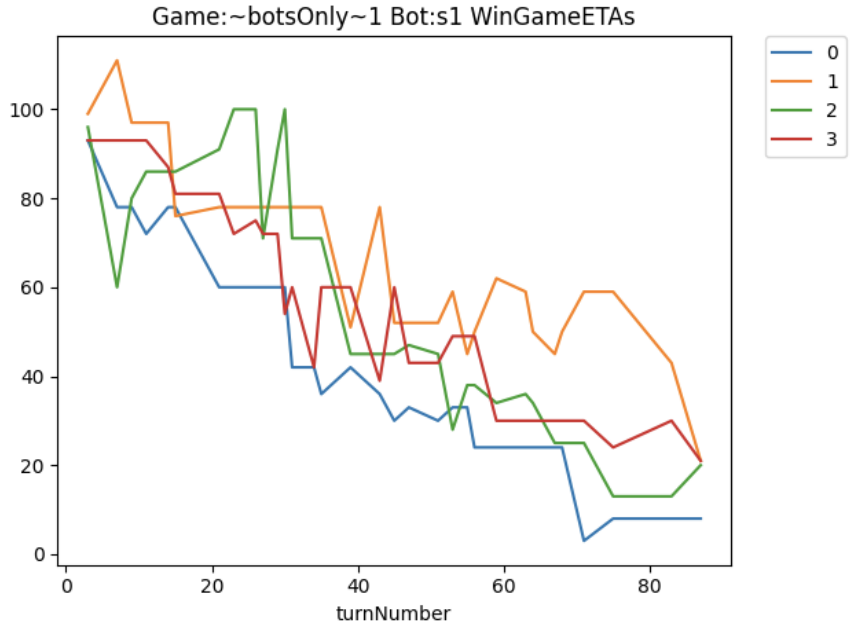
Let’s take an example. On the SOC board, a Smart bot has four roads in red and a settlement in pink. There are also possible actions to take. In yellow, A and B are options of building a road whereas C is an option to build another settlement.

In order to decide the next move, the Smart bot simulates simplified games for A, B, and C choices and calculates a quantity called WinGameETA. WinGameETA is an expected number of turns required for winning after the choice is made. In the simplified simulations, the smart heuristic algorithm considers 6 scenarios in parallel with “a while loop” which includes, finishing the game with “2 settlements (including necessary roads' ETA)”, “2 cities”, “1 city, 1 settlement (+roads)”,”1 settlement (+ roads), 1 city”, “Buy enough cards for Largest Army”, and “Build enough roads for Longest Road.” When Smart bot simulates a simplified game for the choices A, B, and C, the simulation runs possible game scenarios till one of them reach 10 victory points first in a while loop. Simulation of a choice returns WinGameETA as the number of turns that were required for the best scenario.

As a bigger picture, observe that each choice gets a simplified game simulation. Each simulation further considers possible actions to take according to the six scenarios. Then finally, the expected WinGameETA of each choice is considered by Smart bot’s overarching heuristic algorithm.

Simply, Smart bot could choose the move with the least WinGameETA. But Smart bot’s algorithm is more complex than that considering how its choice impacts WinGameETA of itself and its opponents. The image below shows the WinGameETA expectation a Smart bot had at the end of each turn for itself in blue and for its opponents in red, orange and green.

This chart shows all the win ETA predictions made by a single smart bot for itself and each of its opponents over the course of a single game of Catan. The X-axis gives the turn number and the Y-axis is the win ETA prediction in turns:



The fact that WinGameETA calculation is requested in every turn shows that it is a crucial component of the Smart Bot's algorithm. If we could create a WinGameETA calculator for Smart bot that is better than its current methodology, performance of Smart bot should improve.

We chose genetic programming to evolve one polynomial that calculates WinGameETA throughout the game because

- we didn't see in the scholarship the use of evolutionary AI with genetic programming in building SOC-playing AI;
- a polynomial WinGameETA calculation had an advantage of being faster than Smart bot's current approach; and
- polynomial WinGameETA calculations are easier to interpret than Deep Neural Neutral approaches we found in the literature.

Our Evolutionary AI replaces JSettler's WinGameETA calculator method.

II Trainer Walkthrough

The most interesting and complicated part of the Trainer class (in python/trainer.py) is the train function. This function takes the following input parameters:

- mutation_percent: A value between 0 and 1, which determines what proportion of the bots in each generation should be generated using mutation. The rest of the bots will be generated using crossover.
- generations: A positive integer, which determines how many repetitions of the general training structure the bots will go through.
- games_per_bot: A positive integer, which determines how many games each bot will play per generation to determine its fitness score.
- fast_count: 0, 1, 2, or 3, the number of fast bots which each evolutionary bot will be playing against. Each will have 3 total opponents, and any which are not fast will be smart.
- bots_per_sim: A positive integer, which sets how many bots will be in each simulation. (Excess bots will be in a non-full simulation.) We recommend roughly 100 for fully fast bot games and roughly 30-40 for smart bot games. If this is too large, games may begin to fail.
- operator_probability: String form of a number between 0 and 100, which sets the percent chance that a mutated node will become an operator, assuming it isn't at the maximum depth.
- max_children: Integer, which sets the maximum total descendent count a node can have before we exclude it from the pool of possible mutations. If this is set to -1, there is no such restriction.
- constants_only: Boolean, which, if true, forces all mutations to be changes to constants rather than altering or producing anything else.

It converts one set of bot trees and their associated fitness scores into a new set of bot trees, using the initial set as a basis, in the following fashion.

The first several lines of the train function are set up, copying variables around (lines 93-110). The function then proceeds to do the actual training, looping through it a number of times equal to the desired number of generations. Each generation, the trainer first sets up the simulations (lines 120-135), then runs them and records the scores (lines 137-146). Each of these simulations holds several instances of JSettlers, each containing one copy of one of the evolutionary bots in addition to some number of fast bots and some number of smart bots. There is one simulation for each bots_per_sim evolutionary bots. The number of fast bots is equal to the parameter fast_count. All games will be 4 players, with any remaining players being smart bots. Each JSettlers instance runs a number of games equal to the parameter games_per_bot. The simulations then average the scores of each bot over those games, adding an amount equal to the win_bonus parameter to any games in which the bot won, and uses that mean as the bot's fitness. In the final run or if the generation is a generation where it should print, it prints out all the bots (lines 149-154). After that, the train function begins the actual training part.

The training begins by dividing the bots into two groups, the high performers and the low performers (lines 157-167). It does this by taking all the bots, calculating the total fitness, and then multiplying that by the performance cutoff, which was input as the parameter performance_cutoff (lines 157-160). The goal is to have some fraction of the total fitness classified as high performers, with the fraction set by the caller, so this allows the function to know how much total fitness it should put in the high performer bin. Having done that, it sorts the bots by fitness (line 161) and goes through them one by one, putting trees into the high performer list until their total fitness has at least reached the target value (lines 163-165). The trees up to that one become high performers (line 166), while the rest are low performers (line 167).

Having done this, the fitness scores now need to be converted into weights. To do this, they are normalized (lines 170-173), by calculating the total fitness among high and low performers, then dividing the high performer fitnesses by the high performer total (line 172) and doing the equivalent for the low performers (line 173). This will be useful later.

The code now computes how many crossovers and how many mutations to do (lines 176-184). This is reasonably straightforward, multiplying the total number of bots to produce by the mutation_percent parameter to get the mutation count and letting the rest be crossovers (lines 176-177). Notable is that crossovers produce two trees, so if there are an odd number of crossovers, one mutation is replaced with a crossover, unless there are zero mutations, in which case one crossover is instead replaced by a mutation (lines 178-184).

The next step is to produce some storage space, as the trainer will need to use the files containing the current bots to hold the next generation, and the parent bots need to live somewhere in the meantime. To do this, the code first copies over the pointers to the files and the fitnesses, separating them from the original values in gen_results (lines 188-189), then actually copies the files, first of the high performers (lines 190-194) and then of the low (lines 195-199). The copies are done by a copyfile call, and the names of the prior files are retained in gen_resilts, where they will be input into the next generation's call.

Finally, the trainer calls the actual mutation (lines 202-211) and crossover (lines 214-229). The mutation and crossover operations are done separately (see section 3 for details), only being called by the trainer (lines 211, 229). The trainer's job here is to determine which bots to mutate/crossover from and where to put the results. Selecting of bots is done by two weighted random selections, first picking whether to use a high or low performer with weights high_performer_sample_rate and 1-that, then picking a bot within the selected group with weights equal to the normalized fitnesses (mutation: lines 204-209, crossover: lines 215-226). Crossover, of course, does this twice, once for each parent. As gen_results is no longer being used to store parents, after the prior storage space steps, the code places the resultant trees in the next tree in gen_results (mutation: line 210, crossover: lines 227-228). With that, the creation of the next generation of bots is complete.

Lastly, the trainer has to clean up all the files it created along the way. As it has the names of all those redundant files stored in the relevant lists, this simply requires going through those lists and removing the files they reference (lines 232-237). After this, the trainer proceeds to the next generation, and repeats (returning to line 111).

III Training Operations (Initialization, Mutation, Crossover)

Some of the operators used by the training method were programmed in the jsettlers code itself. The calls to the jsettlers code are in the file python/test_set_up.py and the call main methods in src/main/java/soc/robot/evolutionaryBot/EvolutionaryBotBrain.java. We set it up this way because it allowed us to reuse resources from the Evolutionary AI tree simplify the code. Having these operations in Java also let us take advantage of the gson library which allows you to read in a text file to a Java class. This made it so we wouldn't have to parse evolutionary tree files manually. One important thing to node is that each genetic parse tree has a maximum depth allowed to prevent trees from growing infinitely large

Mutation

Mutation takes 5 parameters:

1. The name of the bot to mutate (must correspond to an existing bot file)
2. The name of the bot after mutation
3. The probability that the mutated node becomes an operator (a string from 0 - 100)
4. The maximum number of descendents the mutated node is allowed to have (String)
5. Whether or not only constant value nodes can be considered for mutation ("true" or "false")

The mutation method starts on line 972 of Evolutionary bot brain where it reads all the parameters in (973 - 977), then reads in the bot tree (978 - 979). Then it calls mutate on the bot tree (980) before printing to the new file (981)

The tree mutation method called in line 980 is declared on line 662. This method gets a random node from the tree that satisfies the max_children and constants only_constraint (line 666). If no node satisfies the conditions then the function returns and the tree is not mutated (667 - 669). The node to be mutated is then given a pointer to the genetic tree itself (670). A boolean expression is

evaluated to determine whether or not the node to be mutated meets the conditions to be a constant value (672) before the mutation method on the node is called on line 673 passing in the operator probability and whether or not the node can be a constant. A node is a constant if and only if the following conditions are met:

- It is a leaf node
- It is a right child
- Its parent is a multiplication operator

The node mutation method is declared on line 271. This method starts by randomly determining whether or not the mutated node will be an input node or an operator node weighted by the operator probability. If the node is assigned to be an input node, if canBeConstant is set to True we assign the node to be a random constant, otherwise we assign the node to be a random input node and finish. If the node is assigned to be an operator, we assign it a random operation then create two random children for it. The same operator probability a parent used is also used in determining whether or not the children are operators or inputs. If one of the random child nodes is an operator then that node also gets two random children. This process repeats until all nodes are inputs. Throughout this whole process the max depth of each node is tracked and if any node's children would exceed this limit, then that node is automatically set to be either input or constant type.

Initialization

Initialization takes a single parameter: The name of the new robot you are creating. It starts on line 967 of evolutionary bot brain where it initializes a new bot brain in line 969 before creating a new GeneticTree in line 970. In the genetic tree, all the input values and operations are set up (519 and 520). Afterwards the root of the new tree is assigned to some random input. This node doesn't matter, however, because in the very next step (line 522) this node is mutated with an operator probability of 55%. After mutation finishes the tree is printed to a file (line 523). To summarize, a new tree is created by creating a tree with one node in then mutating that node.

Crossover

Crossover takes 5 parameters:

1. The name of the first bot to crossover (must correspond to an existing bot file)
2. The name of the second bot to crossover (must correspond to an existing bot file)
3. The name of bot 1 after crossover
4. The name of bot 2 after crossover

Crossover starts in line 983 of the EvolutionaryBotBrain. It starts by reading in the parameters and then initializing the two bot trees before calling crossover by passing in the 2 trees (line 992) and printing the new trees to a file.

The crossover method itself is declared on line 860. First, it determines which of the two trees to get the random node from first (863 - 870). Then, we perform depth first search on the first tree, creating a list with each node, its depth, its branch depth (the depth of the node's subtree - the node's depth), and its parent (873). Afterwards, if the node is just a root, we return without doing

crossover (875). Otherwise we choose a random node from the list to be the first candidate from crossover. Call this node S1 (selected node 1).

Next we get a node from the second tree for crossover. Similar to the first tree, we create a list of candidate nodes using the second tree's nodes with each node's parent, depth, and branch depth (880). One difference is that a node from tree 2 is only added to the candidate list if that node meets the following conditions:

- Selected node 1 depth + tree 2 node branch depth < max depth allowed
- Selected node branch depth + tree 2 depth < max depth allowed

These conditions ensure that after crossover both new trees will not be violating the max depth rule. Once we have the candidate list from tree 2 we select a random node from it in line 884. In lines 886-896 we actually do the crossover, by changing the pointers of the selected node's parents while also noting if the selected nodes are left or right children. Lastly, we have to recalculate the depths of all nodes in both trees (lines 899 and 900).

IV Experimental Results

The results from all of our experiments are found in a folder titled "Experimental-Results". Within this directory each folder contains the results of one experiment and follows the naming convention [date]_[test-name]. In addition to these results folders there are two other folders titled "bad_tests" and "old_trainer_tests". The bad_tests folder contains results from experiments that ran when there was a major bug in the Evolutionary robot logic that made it use more of the smart robots logic. Old_trainer_tests used an old method of selection that did not weight the trees before selecting which ones were to be used for reproduction.

Within an experimental results folder itself are several possible items:

- A folder named "bots": This folder contains all the text files for each robot in the population after the final round of mutation/crossover
- A folder named "tree-output": This folder contains PDFs of all the trees in the population at various different points in the training process. The number at the start of the pdf file's name indicates which the generation the tree was from.
- A folder named "pre_trained_bots": If the experiment relied on pre-existing trees to initialize its population, all of the trees used for initialization will be in this folder.
- A results text file: This file will contain several rows of testing parameters indicating all the different parameter sets this experiment used (there are multiple because parameters can be changed after a set number of generations). After each row of parameters is a row of results represented by a Json. The keys in this Json are generation numbers. The value is another dictionary where the key is the name of the robot and the value is that robot's average score in that generation. At the end of the file is a summary of the results in 3 columns. The first column indicates the generation, the second column represents the average score among all bots in that generation, the third column gives the average score of the best performing robot in that generation.

The following table gives a summarizes the results of each of the non-buggy tests that used the final training method

Test ID	Test Name	Motivation	Conclusions
1	02-02-2021-17-37-47_new_trainer_first_run	This was the first attempt with the new trainer. No parameter updates. Node penalties had not been implemented yet. No crossover	Overall poor performance
2	02-02-2021-21-59-01_parameter_update_test	First test where we updated parameters over the course of generations. Every ten generations the max_children variable was reduced and operator probability was decreased. No crossover	Performed better than test 1but still resulted in massive trees
3	03-02-2021-11-14-33_new_trainer_with_crossover	Same as test 1 but also had mutation-percent set to 0.5 in order to test the viability of crossover	Performed better than test 1 but had average results overall
4	03-02-2021-15-29-54_parameter_update_test_with_crossover	Same as test 2 but with mutation percent at 0.5	Similar results as test 2
5	03-02-2021-22-26-20_new_trainer_with_node_penalties	Added node penalties that were used in most future tests. Same as test 1 but had a node penalty of 0.025 in order to keep trees smaller	Performed much better than test 1 but was only average overall
6	04-02-2021-04-52-52_parameter_update_test_with_node_penalties	Same as test 2 but had a node penalty of 0.025	Good performance that was not reproducible when individual tests were run. All the final trees were nearly identical.
7	04-02-2021-14-12-56_higher_cutoff	Similar to test 6 but had a much more gradual decrease of operator probability and max children. Increased the number of games each bot played from 20 to 30. Increased the performance cutoff initially to try to encourage more diverse bots	Performance was back to average levels, worse than in test 6. Bots had still converged by the end
8	05-02-2021-16-31-32_lower_sample_rate	Same idea as test 7 but used a lower high_performance_sample rate to try to make trees more diverse	Performance was nearly the same as test 8. No diversity in final population

9	06-02-2021-12-28-03_periodic_shakeups, 06-02-2 21-12-50-48_periodic_shakeups_gradual_cutoffs, 07-02-2021-14- 37-41_periodic_shakeups_higher_cutoff	Each of these three tests had an occasional generation with max mutation settings. The hope was that by scrambling up the trees every so often, they could escape local maximas	The shake-ups only hurt overall performance
10	08-02-2021-11-29-47_no_update	Large mutation only test with no parameter updates, similar to test 1 but with node penalties. Wanted to observe convergence behavior when maximum mutation was allowed the entire test run	Poor performance, similar to test 1. The final population did have a lot of variety, however. Would indicate that convergence was caused by limiting either operator_probability or max_children
11	08-02-2021-23-24-19_high_crossover	Similar to test 10 but had mutation percent set to 0.1 to test the viability of crossover. Node penalty was still 0.025	Results were slightly better than test 10 but all the trees were extremely small, just a couple of nodes
12	09-02-2021-10-51-00_high_crossover_high_op	Similar to test 11 but increased operator probability and increased mutation percent (to 0.2) in order to prevent all trees to becoming just a of couple nodes large	Score was similar to test 10. Was, for the most part, unsuccessful at creating larger trees. Most trees involved just a root node. While there were some larger ones, they may have just been due to mutation in the last couple rounds. Node penalties probably need to be decreased
13	09-02-2021-20-25-57_no_win_bonus_base_test	Added negative constants that were in all future tests. Identical to test 6 except with a slightly lower node penalty and no win bonus. Win bonus might be skewing some of the results because if a tree randomly wins a couple of games it then it will have a disproportionate affect on how much that tree is selected for reproduction	Overall performance was poor, as expected. It did seems like the best bots in, subsequent, test performed closer to their scores from training

14	10-02-2021-01-29-37_no_win_bonus_param_update	Identical to test 7 except with a slightly lower node penalty, no win bonus, and slightly less training of constants only. Purpose was to see how win bonus would make impact performance	Performance did not seem to have improved, but the discrepancy between best bot score and average bot score on the final generation is smaller. The evolutionary does perform better relative to fast bots with no win bonus indicating that the evolutionary is okay at getting a decent amount of VP but bad at winning. The best bots, in subsequent tests, performed closer to their scores from training than bots did in other tests. Population still completely converged.
15	10-02-2021-11-06-39_param_update_with_printing	Very similar to test 7 but the high_performance_cutoff value wasn't changed and there is no win bonus. We printed the trees every 3 generations to see where in the process the trees converged	Bots performed slightly better than in test 14. Trees started huge then immediately optimized by getting as small as possible. shows the initialization value should probably be smaller. Trees were not converged by gen 30, looked to be mostly converged around gen 40 and completely converged by the end
16	10-02-2021-21-55-58_smaller_init_operator_prob_greater_cutoff	Since all the robots start with massive trees which are then mostly trimmed to nothing due to the node penalty, the operator_probability on tree initialization was set to 0.55 (originally 0.9) this new value was used for all future tests. Increased the higher_performance_cutoff after gen 30 to try to prevent convergence in later generations	Decreasing the initialization probability caused trees to have a much better start when compared to previous tests. Things had mostly evened out by generation ten. Increasing the cutoff after gen 30 caused scores at the end to be much worse. There were 3 or 4 different tree structures at the end though meaning that it did succeed in preventing complete convergence
17	11-02-2021-11-35-39_smaller_init_crossover	Wanted to see if crossover would be more viable with a lower node penalty and a smaller tree initialization value. Kept the operator probability during initialization at 55% and decreased the node_penalty to 0.01. mutation_rate was 0.2 and operator_probability was 0.6	Poor performance. All trees were quickly reduced to almost just the root node
18	12-02-2021-00-04-11_higher_sample_rate	Wanted to try a test that selected almost exclusively from high performers and spent more time evolving trees at mid level depths. Mutation only	While performance was not great it was better than some of the earlier results. Comparable to test 15
19	12-02-2021-00-39-59_big_test	Tried a test of 200 generations with no parameter updates to see if enough generations would be able	Improvement slowed around generation 30 and mostly stopped after generation 100. Did not perform that well

		to create a good tree structure by change. operator_probability was set to 45	
20	14-02-2021-01-12-24_more_input_updates	It seemed like generations with a non-zero max_children count don't provide any benefit over just setting max_children to -1. So, the new tests now only have 3 parameter updates. 60 gen with max_children = -1, 30 with max_children = 0, and the last ten with constants_only set to true	Performance was better, similar to test 18. All of the final trees are pretty small and have very few constants.
21	15-02-2021-11-35-06_smaller_node_penalty	To address the issue of the trees being too small and lacking constants constants, I turned the node penalty way down to 0.005	Performance was very bad until the evolving constants step which resulted in an improvement bringing it up to pretty much the score of test 20. Final result still didn't have constants though, showing the improvement at the end was just from tree convergence
22	16-02-2021-14-26-33_smart_bot_test	Wanted to have an experiment where we trained against smart bots. Apart from the opponent it had the same parameters as test 20	Good overall performance against smart bots. In additional tests the best tree from this experiment was able to beat smart bots more than 25% of the time. Still performed poorly against fast bots.
23	16-02-2021-14-41-39_pretrained_bots_test	Same as test 20 except all robots were initialized using 8 of the best fast bots from earlier experiments	Pretty good performance, better than test 20. Still is not better than fast bots however.
24	17-02-2021-23-33-56_pretrained_bots_test_win_bonus	Same as test 23 except it had a win bonus of ten	Performance seemed slightly worse than test 23.
25	18-02-2021-11-30-44_pre_train_win_bonus_fine_tuning	Took the best tree from test 24 and only trained the constant values	Surprisingly, performance got worse.
26	19-02-2021-01-53-44_hand_made_bot_test	Hand crafted a tree that had one of every input value added together. These input values were each multiplied by a constant. Then the constants were all trained for 150 generations	Not a viable approach. Performed worse than tests 23, 24, and 25

V WinGameETA recording:

Evolutionary bot has everything same as the Smart bot except that we extended the SOCBotBrain, SOCBotClient, and SOCPlayerTracker classes.

Each bot possesses four PlayerTrackers that calculate winGameETA for its bot and three opponents. The CSV file recording the WinGameETA calculations from these PlayerTrackers was named after a simulation name originated in simulation.py where you can specify the number of games and types of bots to simulate.

Simulation name passes into the Java side from simulation.py through SOCBotBrain, SOCBotClient, and SOCPlayerTracker classes. After accessing the simulation name, src/main/java/soc/robot/evolutionaryBot/EvolutionaryPrintWinETA.java creates a CSV file to record winGameETA for that simulation. EvolusionaryPrintETA.java then appends to the CSV file every time winGameETA is recalculated by an evolutionary robot, recording the following parameters:

"GameName," "BotName," RoundNumber," "TurnNumber," "PlayerTrackerNumber," "winETA"

Smart bot WinGameETA recording:

SOCPlayerTracker.java has a method called recalcWinGameETA. At the end of the method, we record WinGameETA with the same procedure as the Evolutionary bot. This happens in the file: src/main/java/soc/robot/PrintWinETASmart.java

Turning OFF and ON of WinGameETA recording:

Change “false” to “true” in the line 86 of SOCPlayerTracker to turn on Smart Bot’s WinGameETA recording.

RECORD_ETAS is a variable passed on from SOCPlayerTracker to EvoPlayerTracker, WinGameETA recording will be activated for Evolutionary bot as well.
To record WinGameETA for only SmartBot, consider commenting recording code in EvoPlayerTracker. It is the same for Evolutionary bot.

WinGameETA visualization:

First, run the simulation.py.
Then, evo_simulationName.CSV file is created in the python folder.
Second, run the plot_etas.py.

There are two different plotting option:

- First: plot a single robot’s four playTracker’s WinGameETA predictions over an entire game.
- Second: plot each robot’s winGameETA predictions for a single robot over the course of

a game.

To store WinGameETA of each playerTracker of each bot for every turn, we loop through the CSV one time to create the dictionary of dictionaries of lists. Then with a second loop over the CSV, we put the values into the data structure. The data structure was later used by the Python visualization library.

NOTE: matplotlib must be downloaded in order to generate the charts:
https://matplotlib.org/downloads.html

VI Tree printing instructions

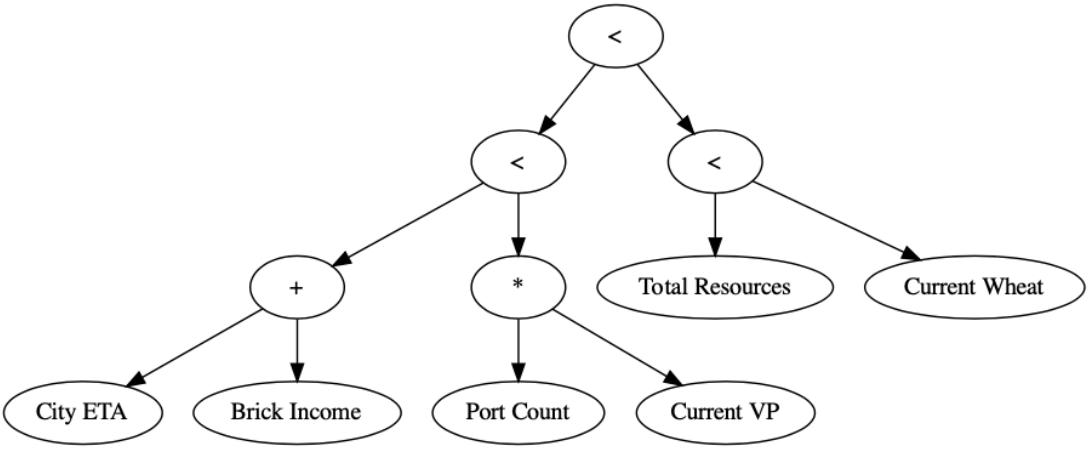
print_tree.py

- This file generates a pdf image of a bot's tree given a bot's text file
- Graphviz, a graph visualization package that may need to be installed:
    - https://graphviz.org/
    - https://pypi.org/project/graphviz
- To use:
    1. Make sure print_tree.py is in the same directory as the file you want to use to pass in a bot's text file
    2. At the top of this file, import print_tree.py ("import print_tree")
    3. To generate a pdf image of a bot's text file, call print_tree.main()
        a. print_tree.main() has 4 parameters
            i. simulation name
            ii. bot's text file
            iii. generation number
            iv. boolean to open the pdf image when it is generated (defaults to false)
        b. example 1: print_tree.main("sim1", "bot1.txt", "1")
        c. example 2: print_tree.main("s10", "bot2.txt", "g10", True)
        d. example output: sim1-bot1-1.pdf
    4. You can also print trees directly from print_tree.py, by calling main() directly in the file with the appropriate parameters
- Generated images are added to a folder named "tree-output"

Sample tree file input:


```
{"type":0,"nodeDepth":1,"left":{"type":0,"nodeDepth":2,"left":
{"type":0,"nodeDepth":3,"left":{"type":1,"nodeDepth":4,"value":{"inputName":"City
ETA"}},"right":{"type":1,"nodeDepth":4,"value":{"inputName":"Brick
Income"}},"operator":"+"},"right":{"type":0,"nodeDepth":3,"left":
{"type":1,"nodeDepth":4,"value":{"inputName":"Port Count"}},"right":
{"type":1,"nodeDepth":4,"value":{"inputName":"Current
VP"}},"operator":"*"},"operator":"\u003c"},"right":{"type":0,"nodeDepth":2,"left":
{"type":1,"nodeDepth":3,"value":{"inputName":"Total Resources"}},"right":
{"type":1,"nodeDepth":3,"value":{"inputName":"Current
Wheat"}},"operator":"\u003c"},"operator":"\u003c"}
```

Sample print tree output:



VII Genetic Tree Inputs

Time To Longest Road
- The estimated time to own the longest road
- getLongestRoadETA(), SOCPlayerTracker
-

Roads To Go
- The number of roads to build to obtain the longest road
- getRoadsToGo(), SOCPlayerTracker

Knights To Go
- The number of knights needed to obtain the largest army
- getKnightsToBuy(), SOCPlayerTracker

Largest Army ETA
- The estimated time to obtain the largest army
- getLargestArmyETA(), SOCPlayerTracker

Resource Income
- The number of different ways a player can obtain a resource. For a given resource, it is the number of that resource you get summed over all 36 possible dice rolls.
- getResourceIncome(resource), EvolutionaryPlayerTracker
- Function takes in a resource as a parameter, or “total” for all resource income combined
    - Resources: brick, log, ore, sheep, wheat

Current Resources
- The amount of a resource that a player owns

- getResourceCount(resource), EvolutionaryPlayerTracker
- Function takes in a resource as a parameter: brick, log, ore, sheep, wheat

Total Resources:
- The total amount of resources that a player owns
- getTotalResources(), EvolutionaryPlayerTracker

Current VP
- The current number of victory points that a player has
- getCurrentVP(), EvolutionaryPlayerTracker

Port Count
- The number of ports a player has access to
- getPortCount(), EvolutionaryPlayerTracker

Dev Card Count
- The number of development cards a player has
- getDevCardCount(), EvolutionaryPlayerTracker

Build Location Count
- The total number of remaining possible settlement locations
- getBuildLocationCount(), EvolutionaryPlayerTracker

Ready Build Spot Count
- The total number of playable settlement locations in the next turn
- getReadyBuildSpotCount(), EvolutionaryPlayerTracker

Build ETA's
- The estimated time to build something
- getBuildETA(type), EvolutionaryPlayerTracker
- Type parameters: road, settlement, city, dev card

VIII Instructions for building Jsettlers and running experiments

Note that the "Readme.md" file in the base directory and the "Readme.developer.md" file in the "doc" folder are great places to look if you are confused or want more details regarding how Jsettlers works and what its parameters are. There are several different ways you can run jsettlers games. After reading the instructions for building jsettlers, I recommend you look at all the different methods before you start with one.

Building Jsettlers:

1. Download IntelliJ and clone/open the repository. I recommend opening the "python" folder with PyCharm as well but if you want you can use IntelliJ for the entire project (this would require you to configure a python interpreter in IntelliJ). You're free to use

other IDEs as well but certain steps in this guide may not be applicable to other environments

2. The project uses gradle to build. This should be installed by default. The project may automatically connect to gradle on start-up or you may see a pop-up asking you to connect to gradle (if you do, select yes). If your project didn't connect to gradle, try closing it and reopening it. If that doesn't work you'll have to try troubleshooting. Here is the IntelliJ documentation on gradle which is a good place to start. When this is done you should have an automatically generated gradle wrapper file in your root "gradlew"
    a. You will know you are successfully connected to gradle if you have a sidebar on the right hand side of your screen that says gradle. Clicking this will open the gradle tool window. You can also open the gradle tool window via "View→Tool Windows→Gradle".
3. To build the project, open the gradle tool window, open the drop down, and select "Tasks→build→build". You'll have to build anytime you edit the java code. I recommend saving this as a run configuration. While you're building it may say some tests are failing, just ignore these warnings. After building you should have a new, auto generated "build" folder in your root directory.
    a. Optionally, you should now be able to run Jsettlers via the gradle tool window via "Tasks→application→run"
4. If you want to run simulations with the evolutionary robot, you must copy and paste the file named "gson.jar" (found in the root) into the folder with the other jar files "build/libs/"

Running robot only games (default method):

- Note: I generally don't recommend running games this way. The default method of running robot only games is very bad and gives you very little control over which robots are in which games.

1. Ensure that all the steps in the building Jsettlers section succeeded
2. Check the Jsserver.properties file in the root directory. These are all command line arguments that the jsettler server uses on start-up. For more details regarding these parameters and other unused parameters, refer to "src/main/bin/jsserver.properties.sample". To run a single robot only game, uncomment the line that reads "jsettlers.bots.botgames.total=1". (the other parameters are all fine).
    a. If you want the server to shut down when the bot only games have finished, set the line that reads "jsettlers.bots.botgames.shutdown" to be equal to "Y"
3. Right click on and run the jsettlers server jar file which should be in the automatically generated build folder: "build/libs/JSettlersServer-2.4.10.jar". I recommend setting this as a run configuration.

Running robot only games from an input file:

1. In the Jserver.properties file, uncomment the line that reads "jsettlers.bots.botgames.total=input.csv"

    a. Optionally, if you want to see the results of the game, uncomment the line that reads “jsettlers.bots.botgames.total=input.csv,test_results.txt”
    b. Note that input.csv and test_results.txt are just default file names, you can use whatever file names you want as long as they correspond to actual files that have already been created
2. Set up your input.csv file. There is a sample file in the base directory but feel free to edit it. Note that input files must be very precise and have no extra spaces or blank lines. Input files have the following format:
    a. The first row is where you declare all of the built-in jsettlers robots separated by commas. Note that if a robot’s name starts with “s” it will be a smart bot and if it starts with “f” it will be a fast bot.
    b. The second row is where you declare third party robots. Every two indices in this row are part of a pair where the first value indicates the name of the third party robot and the second entry indicates the location of that robot’s client in the code. Note that if you want to use evolutionary robots, the evolutionary robots name’s must correspond to existing text files (without the .txt extension) that have a robots encoded genetic tree.
        i. If you don’t want to include third party bots. Leave this line blank
    c. All subsequent lines represent an individual robot only game. Each of these lines will have four bot names (among those that you specified in lines 1 and 2) separated by commas
3. Optionally, set up the result file. If you want to record the results of the game be sure to have initialized an empty text file with a name that matches the value you passed into the jsserver.properties file.
4. Run the jsserver.jar file: “build/libs/JSettlersServer-2.4.10.jar”

Important Notes:

The python folder contains files that run tests and perform experiments automatically, if you want these files to work properly, be sure to mind the following:

- If you want to run simulations with the evolutionary robot and you haven’t already done so, copy and paste the file named “gson.jar” (found in the root) into the folder with the other jar files “build/libs/”
- Running simulations and experiments via the python code: The python code has a lot of functions that automate the process of creating input files and running experiments. In order for these functions to work, your path to the jsettlers jar files **must** match ours. So, the root directory must contain a “python” folder and a “build” folder. All the python files should be in the python folder. The build folder should contain a folder called “libs” that has all the jar files:
    - Root
        - python
            - [Python_file_1]
            - [Python_file_2]
            - [etc.]..
        - build
            - libs

- gson.jar
- JSettlers-2.4.10.jar
- JSettlersServer-2.4.10.jar

Running the simulation.py file

The python code has a built in class that automatically generates input files, runs simulations, and parses results files:

1. Navigate to python/simulation.py
2. Create your simulation object in the main method. All the parameters are explained at the top of the class. Note that any evolutionary bots you include must have their text files initialized with their trees.
    a. If you only want to run games with jsettler robots, set "no_evo" to true when you create your simulation object
    b. To initialize evolutionary robots, navigate to the python/test_set_up.py file. In the main function run the "initialize_new_bot()" method passing in whatever name you want for the new evolutionary robot. Note that this file is set up so 1 evolutionary robot will play against 3 built in robots. It does not support evolutionary robots playing each other
3. Run the simulate method on your simulation object
    a. If you want to see the results, be sure to print out the result of the four get methods after simulation is ran
4. One important note is that if you want to rerun a simulation with the same name, be sure that the input and results file from the last run are deleted

Running Evolutionary robot experiments:

1. There are a couple of dependencies that must be installed for experiments to work.
    a. Time outs: timeout and retry logic will work by default on Linux. The "time_out" parameter in "python/simulation.py" has instructions for how to set up timeouts on mac.
    b. Print tree: If you wish to have the experiment print the population of trees as it runs, you will need to have all the dependencies for the "python/print_tree" file. Refer to section 6 for details. I recommend you test this out before trying to print trees in the experiment.
2. Set up a Json configuration file with all the parameters of your experiment. "python/trainer_config_sample.json" provides the sample format these configuration files use. To summarize some of the different components of it:
    - "Tree-depth" is the max depth that evolutionary trees are allowed to have. **NOTE:** changing this value is just for bookkeeping purposes and does not actually change the tree depth. To change the tree depth, navigate to "src/main/java/soc/robot/evolutionaryBot/EvolutionaryBotBrain.java" and set the instance variable "MAX_DEPTH" to whatever integer you want
    - "Training_runs" is a list of different experiments. Each experiment is a json and they will be automatically run one after the other.
    - "name" is the name of the experiment

- "bot_count" is the number of evolutionary robots you want in the experiment
- "print_rate" is how frequently the generation of genetic trees is printed. If you don't want the trees printed automatically, set this to 0
- "initial_bots" is a list of pre-existing bot files used to initialize your new robot files. If you want to start with new, random evolutionary robots, leave this empty
- "training_sessions" is a list of dictionaries where each dictionary contains a set of training parameters. This allows you to modify the parameters of an experiment in the middle of its run. E.g. you could do 10 generations with one set of parameters and then train for another 20 generations with a different set. All of these parameters come directly from the evolutionary bot trainer so refer to the Trainer section or the "python/trainer" file for reference.
    - One note is that the number of games done in a generation is the games_per_bot * bots_per_sim. Jsettlers will run out of memory if too many games are running simultaneously (I've found that it's best to keep the number of fast bot games running simultaneously under 100 and the number of smart bot games under 30). To make sure you don't have too many games running at once, you can set the bots_per_sim value to whatever you want. Changing bots_per_sim will have no effect on the results of the experiment.

3. Go to "python/trainer_automator.py" and in the main method call the run_experiments function while passing in the path of your configuration file
4. After running, you should have a new folder for each experiment that has the results. See the experimental results for details regarding the contents of these folders

IX Summary the Repository:

Top level directory:

- "Experimental-Results" directory: Contains the results of all of the Jsettlers simulations we ran. See section 4 for more information.
- "python" directory: Contains all of the high-level code for the evolutionary algorithm (i.e. all the code that isn't directly part of the Catan game). This includes code for running simulations, different trainers, files for analyzing/visualizing results and configuration files.
- "jserver.properties": parameters for the jsettlers server. See the instructions for relevant parameters and refer to "src/main/bin/jsserver.properties.sample" for a sample.
- "Input.csv": a sample input file used to initialize bot games. See the "running robot only games from an input file" part of section 8 for details.
- "deploy.sh": A script that uploads all the necessary files to the Carleton cs servers. Preserves the necessary file path outlined in the instructions.
- "gson.jar" file: A jar file that the evolutionary bot uses to write its tree to a file and load in its tree from a file. Must be copied and pasted into the same directory as the other jar files.
- "src" directory: contains all of the actual settlers of catan code.

- “doc” directory: more documentation by the jsettlers creators.
- “target” directory: jsettlers directory used by gradle.
- “Evolutionary_Bot.txt” file: a sample evolutionary robot text file.
- “Readme.md” file: Instructions written by jsettlers creator. Is not up to date with the new additions to the code base.
- “test_results.txt” a blank file you can use to write game results to.
- “build.gradle” file: handles building the project.
- “.gitignore” file: Keep track of files you don’t want in version control

Additions to Jsettlers code:

- src/main/java/soc/robot/evolutionaryBot/EvolutionaryPrintWinETA.java: Helper file for printing out the win ETA for evolutionary bots. See section 5 for more details
- src/main/java/soc/robot/evolutionaryBot/EvolutionaryBotClient.java: Extends SOCRobotCleint. Handles the initialization of new Evolutionary robots. Creates the evolutionaryBotBrain.
- src/main/java/soc/robot/evolutionaryBot/EvolutionaryPlayerTracker.java: Extends SOCPlayerTracker: Overrides the recalcWinGameETA method with the evolutionary tree strategy. Has all the different methods for getting tree input values (see section 7 for more details)
- src/main/java/soc/robot/evolutionaryBot/EvolutionaryBotBrain.java: Extends SOCBotBrain. Primary logic for the evolutionary robot. Keeps track of all the different constants, tree input values, and operators used by genetic trees. Has a GeneticTree subclass which is used to recalculate the winGameETA. The geneticTree subclass also has subclasses for TreeNodes and TreeInputValues. Logic for initialization, mutation, and crossover are also part of the Genetic Tree and are called via the main method (see section 3 for more information). Genetic trees are either randomly generated or read in from an evolutionary bot’s text file. EvolutionaryBotBrain also overrides the relevant methods from SOCBotBrain in order to ensure that it is using the Evolutionary player trackers but also following the same logic as smart bots. See comments in the code for more details.

Modified files in the Jsettlers code:

- src/main/java/soc/game/SOCGame.java: made some instance variables public
- src/main/java/soc/game/SOCGameOption.java: Some of the unused game options were causing the client to crash when trying to initialize a bots only game while connected to the server. These are now excluded from possible game options.
- src/main/java/soc/game/SOCPlayer.java: potentialSettlements was made public.
- src/main/java/soc/robot/SOCBuildingSpeedEstimate.java: some methods were made public.

- src/main/java/soc/robot/SOCPlayerTracker.java: some methods were made public and the getSOCPlayerTrackerCopy method was modified to be able to make new Evolutionary Player Trackers. Optional Win ETA printing for smart bots was added as well.
- src/main/java/soc/robot/SOCRobotBrain.java: Gson was imported, a geneticTree instance variable was created, ourPlayerName was made public.
- src/main/java/soc/robot/SOCRobotClient.java: an instance variable was created for the simulation name.
- src/main/java/soc/robot/SOCRobotDM.java: getETABonus was made public.
- src/main/java/soc/robot/PrintWinETASmart.java: Helper file for printing out the win ETA for smart bots. See section 5 for more details
- src/main/java/soc/server/SOCGameHandler.java: Printing game results to a file was added (lines 2719 - 2741).
- src/main/java/soc/server/SOCLocalRobotClient.java: Created a new client constructor that lets the simulation name be passed to the robot clients.
- src/main/java/soc/server/SOCServer.java: Implemented initialization bots via an input csv file. See section 8 for instructions on how the input csv works. Most of the relevant new code is in if statements that start "if (GAMES_FROM_FILE)". Some warnings were also commented out to allow the evolutionary bot to be playable through the client.
- src/main/java/soc/server/SOCServerMessageHandler.java: Some warnings were commented out to allow the evolutionary bot to be playable through the client.
- src/main/java/soc/util/SOCRobotParameters.java: setStrategyType was made public
- src/test/java/soctest/util/TestGameList.java: Some warnings were commented out to allow the evolutionary bot to be playable through the client.

"python" directory:

- "old_trainer_stuff" python package: Contains files relating to a deprecated robot training method.
- analysis.py: A script for providing quick analyzing the results dictionary that is output from training.
- jsserver.properties: same as the properties jserver.properties file in the top level directory but the botgames.total left blank because we declare this explicitly in simulation.py.
- plot_etas.py: File that plots winETAs for robots. See section 5 for details.
- print_tree.py: File that visualizes tree text files. See section 6 for details.
- simulation.py: Automatically writes input files, runs Catan simulations, and analyzes results. See comments in the file for details on parameters and see the simulation.py notes in section 8 for instructions.
- test_set_up.py: Contains calls to the main method of the EvolutionaryBotBrain used for initializing evolutionary robots, performing crossover, and performing mutation.
- timer.py: Tests how long different simulations take.

- trainer.py: The training algorithm. See section 2 for more details.
- trainer_automator.py: Reads in a json configuration file and automatically queues and packages experiments. See the experiment part of section 8 for details.
- trainer_config_sample.json: a sample configuration file used by the trainer_automator. See the experiment part of section 8 for details