PacWar Project

Linus Shih (yls2), Ege Ersu (ee31)

Problem Description

Project Background

This PacWar project was conducted over the course of the Fall 2018 Semester at Rice University.

Objectives

The goal of the project is to design the best possible gene sequence for a population of digital PacWar mites such that they consume as many members as possible of an opposing PacWar mite population.

Components

There are several components that make up the PacWar game - namely, the mites themselves, and the environment in which they compete.

Game Environment

The game and battles between mite populations take place in a 19x9 2-D grid world for a total space size of 171. Additionally, time operates on a discrete scale, so that mites act on a distinct turn-by-turn system.

PacWar Mites

Every mite possesses several attributes that influence how they interact with each other and the environment. These attributes include the following:

1. The mite's age (ranging from 0 to 3).
2. The direction it is facing in the gridworld (up, down, left, or right).
3. The genetic sequence of the mite and all other members of its population.
    a. Length = 50 single digits
    b. Possible range of values = {0, 1, 2, 3}
    c. Total possible number of gene sequences = 4^{50} = about $1.2676506*10^{30}$

Each mite follows a fixed set of behaviors. The one it performs on any given turn is determined by what value(s) it has for its attributes. Such behaviors include the following:

1. Aging (typically occurs if uncontested by mites of another species)
2. Changes direction (typically a function of its age and the age of a neighboring mite if one is present)
3. Births a new mite in a neighboring cell (either peacefully or by consuming and replacing a weaker enemy mite)
4. Dies due to reaching the maximum age
5. Gets consumed by an enemy mite

Although this outline is just a brief paraphrasing of the more detailed ruleset (which can be seen in the official project description page), one can quickly see the direct effect that the first two attributes - age and direction - have on the interactions a mite has with the environment and its peers. However, the connection between a mite's genes and its action set is far less obvious, and optimization of this relationship to improve mite performance forms the crux of this project.

With all the information given to us initially about the mites and their attributes, it would in theory be possible to determine exactly how each gene position and value impacts the behavior of a mite in any given situation. However, given the sheer number of possible genes (recall: 4^{50}, about $1.2676506*10^{30}$), it would be nearly impossible to solve this behavioral optimization problem through brute force search algorithms with the limited time and computational resources at hand. As a result, the best option to find the "killer gene" is to perform a limited, yet educated, exploration of the state space of possible gene sequences.

Battle Scoring

The last important aspect of PacWar is how combat results are scored. Battles are always one versus one, with point distributions as follows:

Points	Outcome of a duel
20-0	Destroying the opposing species in under 100 rounds
19-1	Destroying the opposing species in 100-199 rounds
18-2	Destroying the opposing species in 200-299 rounds
17-3	Destroying the opposing species in 300-500 rounds
13-7	Outnumbering the opponent by at least 10:1 after 500 rounds
12-8	Outnumbering the opponent by between 3:1 and 10:1 after 500 rounds
11-9	Outnumbering the opponent by between 1.5:1 and 3:1 after 500 rounds
10-10	If neither species outnumbers the other by more than 1.5:1 after 500 rounds

Note that the total combined score of both sides always adds up to 20, with perfect ties being a 10-10 split of points.

PacWar Program API

The PacWar program provides a Python API for interacting with the combat system (which itself is compiled in C). Two critical functions from the API are the following:

1. The *battle* function conducts a fight between two input gene sequences and outputs the number of remaining units from each population and the number of rounds taken before the fight is complete.
2. The *score* function takes as input two gene sequences and returns the score that the first input gene achieves against the second, using the battle results returned from the *battle* function and the scoring rubric presented in the previous section.

Ultimately, the project objective can be represented as an optimization problem where we attempt to maximize the value returned by *score* for a given gene sequence against as many other gene sequences as possible.

Solution Description

Now, we discuss the workflow we used for solving this optimization problem, the various algorithmic components used, and the software implementation. We developed our solution in Python 2.7 using the PyPacWar interface supplied to us. Our primary computational devices were our two laptops.

Overview of Workflow Structure

The general structure of our workflow and software implementation looks as follows:

1. Population Initialization
2. Tournament Phase with Bracket-Stage style competition
3. Genetic Algorithm

These three components have subcomponents and further details that we describe in more detail below. This workflow is repeated many times in order to uncover the best performing genes from as many random populations as possible.

Strategy

We first initialize a random population. Each gene in this population is randomized.

Our core algorithm consists of three major phases. First is the tournament phase, second is the GA phase and third is the champion update phase. Each new population goes through these two phases and individual genes are scored and ranked according to their performance.

Tournament Phase

After initializing an initial population of size 1024 (or any other power of two), we split the population into groups of two. Since all the genes are random, it doesn't really matter how we split them. Then we make them duel and look at the number of survivors remaining. We form a new population of genes by picking the gene with higher number of survivors, basically halving the whole population. We repeat this until we go down to a population of size 32 (or any other power of two).

The reason we went with this approach is the fact that the GA phase consumes too much time and computational power, with all its crossovers and score calculations. If we are going to spend all this time on a single population, we don't want to spend it on crossing over really bad genes. Since we are only looking at the number of remaining survivors, the tournament doesn't actually take a lot of time. For a population size of 1024, it's simply 512 + 256 + 128 + 64 + 32 = 992 games.

GA Phase

Once the tournament phase is over, the population size is reduced to 32. Now that we have a decent number of relatively better genes, we will perform crossover and mutation on them to create diversity within the population. The number of iterations of the GA algorithm is specified as a hyperparameter, and on each iteration the following operations are performed:

1. **Calculating Scores**

   Given a single gene, there are three scores we calculate. To rank genes we use (c) which makes use of (a) and (b).

   a. **Population Score**

      Given a population, we iterate over each gene and create a score matrix S.
      S[i][j] on the score matrix corresponds to gene i's vanilla score against gene j.
      After we compute scores for half of the population, we make use of symmetry and calculate the rest without doing any battles.
      For any given i,j: S[i][j] = 20 - S[j][i]

      After the matrix is complete to get the population score of gene i, we sum over row i and divide it by the size of the population. This way we get a number that represents that gene's average performance against the rest of the population.

      This stage is computationally expensive since each gene battles every other gene in the population, but the obtained score is valuable since we know how the gene ranks within the population.

   b. **Champion Score**

      Champions score represents how good a gene is with respect to the current champions. Each gene battles every champion and the vanilla scores it gets are averaged. A bad gene will almost all the time loose to the champions, so it's wouldn't be surprising that this score is 0 most of the time.

   c. **Real Score**

      Real Score is a weighted average of Population Score and Champion Score:

      *w1 * Population Score + w2 * Champion Score where: w1 + w2 = 1*

      We picked *w1 = w2 = 0.5,* but different weights will yield different results.

      A higher *w1* will emphasize the population score, making a gene's performance against its own peers more valuable.

      A higher *w2* will emphasize the champion score, making a gene's performance against the champions be more valuable. The reason we need the Champion Score is to make our Real Score be a better representation of how good our gene is in the whole solution space. A gene might beat every other gene in the population but it wouldn't be a good representation to assign it a score of 20. Champion Score helps us reduce that number.

2. **Crossover**

   In order to perform crossover we first sort the genes based on their scores and take top 60% of the list. Then we iterate over all genes and with prob of 0.3 swap that gene's i-th index with a random gene's i-th index that is picked from the top 60%.

   Another idea we tried was to first pick two random genes, iterate over the indexes of these two genes only. But our current crossover picks a random gene from the top 60% for each individual index. This increases the diversity of the population a lot more.

3. **Mutations**

   Once the crossover phase is over, the population goes through a mutation phase. We implemented the version discussed in class. We simply iterate over all genes and change an index to another number with a certain probability.

Champion Update Phase

The champions are kept in a text file line by line. After the GA phase is over and we hopefully reach a local minima, we use this population to update our champions. We make our genes duel every other champion and if it beats even a single one, we take that gene and add it to the champions after calculating it's real score. In the end we have a list of genes greater than the size of the champions, so we sort them according to their scores and take the first 12 (or whatever the champion size is). This way our champions get better and better after each iteration and it gets harder to score higher. This means that our score function becomes a better representation of a gene's performance as time passes.

Hyperparameters of our model

Number of Generations: This is a direct function of how long you want to run the program. The ideal number would be infinity. Each generation first goes through a tournament phase, then a GA phase.

Number of GA Iterations: This determines how many iterations a single GA should run. Each GA iteration consists of a scoring phase, a crossover phase and a mutation phase. Picking a high number would make the

Champion Pool Size: This increases the quality of our score function, but drastically slows down the whole process. The reason is that our score functions and champion updates iterate over all genes in the champion pool.

Initial Population Size: This only affects the speed of the tournament phase. We went with 1024, but any power of two is sufficient. Increasing this will make the population entering GA better, but hurt the time needed.

Population Size after Tournament (n): Population size exponentially increases the complexity of our algorithm. Our score function makes every gene duel every other gene in the population, therefore our score matrix is n^2. We first experimented with n=32, but it was taking a lot of time for GA to end. We then reduced it down to n=16 so that we can search through the solution space a bit more.

Results

Initial Foray

The first benchmark we met was being able to defeat two fixed-value genes: all 1's and all 3's. We were advised in blog post feedback that our performance against them would be good early indicator of whether our early gene were viable or not. Our initial few populations did not manage to meet this benchmark, and it was only with several trials that we produced a population containing genes that could. We believe that the population initialization is the most critical factor influencing this benchmark since the randomness of this step has a large impact on the overall quality of the population moving forward into the next steps of the workflow. If the population was initialized too far away from any local or global optima, then it was less likely for that population to achieve even rudimentary success. We continued to run our workflow, seeking other initializations capable of passing the first benchmark. We kept the best genes from only the viable populations to keep in our database of "champions". During the first round-robin tournament, we submitted our best gene from this initial list and ranked in the upper third of the class, which was a promising result.

Phase 2

We felt comfortable with the approach we had taken up to this point. We continued using our existing workflow to find more genes that could match our existing champions and found several more that performed better than the one we submitted for the first tournament. However, when we submitted our new best gene for the second tournament, we did not achieve much success. Looking back, we had neglected to realize that many of our best performing genes were similar in structure to each other, and that we had essentially reached a plateau that we had not gotten out of. We had not properly explored the solution space, and the resulting lack of genetic diversity contributed to our low ranking.

Phase 3

Based on the lack of success from the second tournament, we attempted to produce more diverse genes by generating a new set of champions independent of the previous database we had acquired. We ran our workflow using a new, empty champion database with the reasoning that, by ignoring our preexisting champions, we could remove any bias towards these genes that might be affecting our genetic diversity. Unfortunately, given the short time between the second tournament and the final submission, we were unable to find a gene that was both sufficiently diverse from the previous top performers and capable of defeating them as well. Eventually, we reverted to finding a new gene from the same genetic cluster as our previous champion and submitted it as our final result. Ultimately, we did not achieve a high final ranking, prompting us to discuss with our peers and between ourselves to find the glaring flaws we did not see in our approach to the problem before.

Improvements

Upon further discussions and thinking, we thought of several possible improvements that we could have done.

1. Removing the bracket phase competition and using only round-robin competition. When we designed our workflow, we added in the bracket style tournament step as a way to quickly eliminate unviable genes and reduce the population size to improve the computation time of the genetic algorithm step and the round-robin style competition. However, this approach has the potential effect of eliminating candidate genes without fully testing their combat potential. Unlike round robin style competition, bracket style tournaments only lets each gene fight against a limited number of other genes before eliminating it from contention. While more computationally expensive, round robin

competition does a deeper analysis of each gene's combat potential and thus is less likely to eliminate good candidates like our bracket style tournament phase. Removing the tournament phase would increase the population size going into the genetic algorithm step, but would likely yield more descriptive results and would better represent each gene's potential before eliminating them.

2. Placing a penalty on each gene if it is too similar to previous champions. This was a potential idea that we discussed with our peers following the final tournament, which would essentially reward genetic diversity as we loop through the genetic algorithm step.

Conclusion

The Pacwar problem was a hard problem because we didn't have a really intuitive understanding of how individual genes affected the actions of the agent. That's why we had to treat the genes simply as lists of numbers and basically hope that we can reach local minimas in the solution space. We think our solution would have been more effective if we had (1) implemented a hill climbing algorithm, (2) ran the code for a longer time. The tournament idea did not really help us find a good solution and we probably should have spent that time on more experimentation and on implementing the hill climbing algorithm.

Since we had to treat the game as kind of a black-box, we thought Genetic Algorithms were the way to go. We probably should have trusted the algorithm less and spent more time on analyzing how certain genes affected the gameplay and target those specific genes at optimization. An honest confession is that we didn't really use the GUI that much and approached it more of a mathematical problem. Understanding the game on a more intuitive level certainly would have helped us come up with more task-specific optimization strategies.