PacWar World

And how to find the best genes

Alex Craik & Dejan Grubisic

Sep 17. - Dec 5. 2020.
Comp557 Artificial Intelligence, Rice University

ROADMAP

INTRODUCTION

"It is not the strongest or the most intelligent who will survive but those who can best manage change."

— Leon C. Megginson

All living things have one thing in common - their ability to survive. That ability is not necessarily connected with their strength, ratio, or some other quality, but rather with our ability to respond to all challenges they face. In the PacWorld [1], this ability is extremely important, as a PacMite can face an enemy at any step. To survive in such harsh conditions, PacMite must adopt their genes to respond to any challenges.

The PacWorld consists of 9x19 grids that are completely surrounded by walls and no PacMite can escape. PacMites can move one step in each round with a maximum life span of 4 rounds. In certain situations, a PacMite can fight against PacMite of different species, give birth, or being killed. All PacMite behavior is specified by 50 genes in the range from 0 to 3, and our job is to find the sequence of the gene which is going to perform the best.

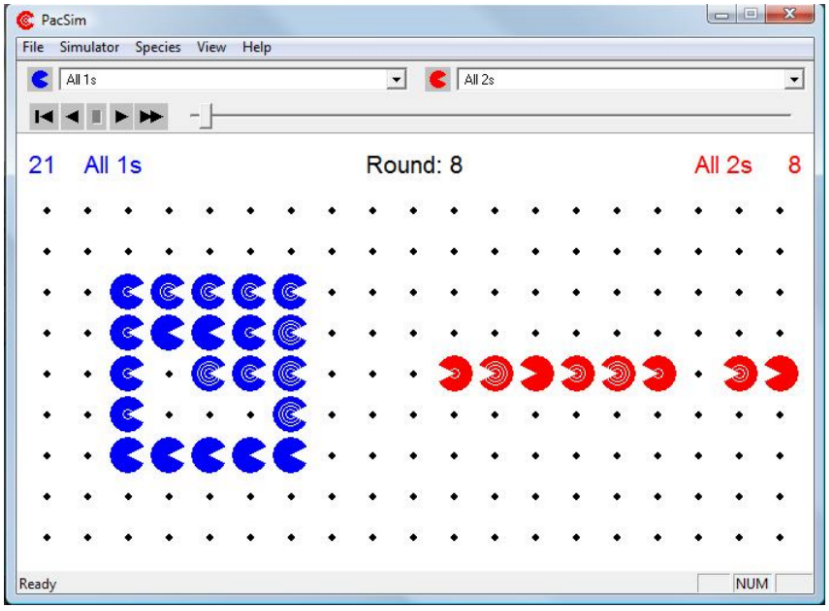

Figure1. PacWorld Simulator

Comparing every two genes and finding an optimal gene is computationally impossible and instead of that in this project, we focus on genetic and local search algorithms.

METHODOLOGY

To find the best performing gene, we used several methods to conduct the search of PacWorld space. To begin our search, we created a relatively simple genetic algorithm to locate our first contending champions, or genes that could beat other genes effectively. Then we used different mutation and crossover mechanisms to ensure diversity of genes population. During the semester we repeated this procedure many times and collected dissimilar champions on different local minima.

In order to make our gene champions more robust, we applied two additional search stages with including hill climbing and genetic algorithms. One of these was concentrated around genes that perform well against current champions, and others were focusing on unique genes in our population. We find that this approach was extremely fruitful as it allowed us to avoid overfitting and enable our best performing genes to achieve good results against genes that they have never seen in our dataset before. In the end, we clustered our genes, by using the K-means algorithm, found the best performing gene for each local minima, and finally find the gene that performs the best amongst all local champions. This way, we emulated the situation that we expect in the final tournament, as many genes are going to end up on different hills, and the winner is going to be a gene that can beat the most hills.

PROBLEM-SOLVING

8. How to create an initial set of strong genes
9. How to implement the best scoring function
10. How to make the population diverse
11. How to deal with too many champions
12. How to make champions robust
13. How to make iterations fast
14. How to select the best gene

1 How to create an initial set of strong genes

Specifically, we began with 25% elitism with random cross-over between 40-60% of the entire gene length. The best performing genes would be included in our tester genes, which was how we calculated a score table. While this was promising at first, we encountered a problem. Namely, while the champions would perform well against each other and random genes, the best performing gene failed against the standard case of all threes. We felt that, in failing against a standard case, our champions would not perform well against our classmates' genes. This indeed would have been the case as the weekly championships included the standard "All Threes" gene.

Our initial fix for this observation was to include the standard genes (zeros, ones, twos, three) in our tester gene set. This would help locate genes that could beat the standard genes in addition to other random genes. It was assumed that, if a gene was able to beat the standard gene, it would indicate a path to a local minimum with a higher chance of beating other local minima that were not able to beat standard genes.

2 How to implement the best scoring function

Here though we encountered another problem. Since there was no weight on different genes in the tester genes, the search for champions employed a score value that did not

differentiate between the best performing gene and a standard gene. Since we would like the score to more heavily weight the higher-performing champions, we included a weight factor when calculating the score table. Specifically, we applied the weight of one for the standard genes, and we weight the remaining tester genes based on their performance against all champions from previous generations. We manually found the most effective maximum weight multiplier for our best champion by changing the maximum weight and assessing the performance of the resulting champions.

Besides that, we discovered an interesting phenomenon of circularity in PacWorld. Even though every new iteration is stronger than the previous, that doesn't mean that it is stronger than all its ancestors. In Figure 2 we are comparing the top 10 genes of each iteration against the top 10 genes of iteration 40. We can see that the results of the genes before iteration 40 graph are negative which indicates that iteration 40 has stronger genes, while after iteration 40, scores have a positive slope and becomes positive which indicates that these iterations are stronger. The reason why the genes of iterations between 40 and 60 are negative is that our scoring function gives more points to the second genes (from iteration 40) in battle if they are even.

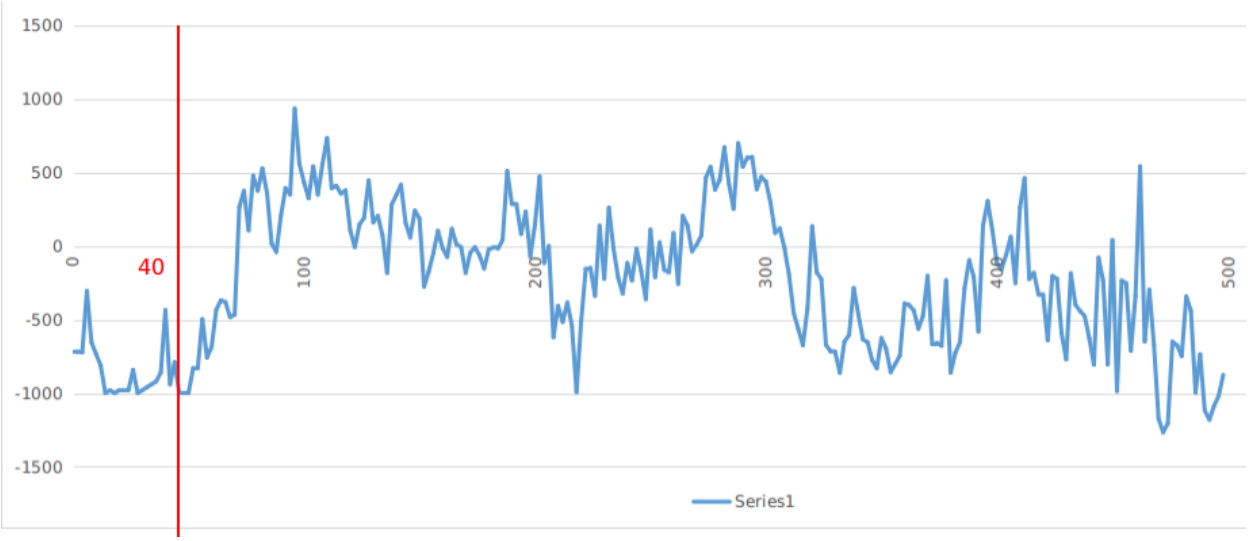

Figure 2. Comparison of each iteration against the iteration 40

3 How to make the population diverse

We eventually realized that our constraint of a 40-60% cross-over was limiting the ability to retain high-functioning segments of our best champions. To mitigate this error, we expanded the constraint to 2%-98% potential cross-over of the genes for each generation. Additionally, we included hill-climbing for our best 25% of genes. For example, rather than using cross-over for all genes in a generation, we performed cross-over with mutation (genetic search) and mutation alone (hill-climbing). The inclusion of hill-climbing means that we would be directly searching around our best performing genes, rather than simply using our best genes as components for new genes.

4 How to deal with too many champions

Eventually, the total size of our champion list was becoming too large for effective computation. We didn't feel it was appropriate to delete all champions below a certain number. Instead, we removed genes that were too similar to each other (with a priority for genes higher on the list), with the idea being that we would include as many different 'hills' as possible for our search. If we deleted all genes below i.e. 200, we may end up only searching a couple of areas and may have deleted potentially fruitful gene paths that just had not been adequately explored. We created a gene similarity mechanism with the assumption that, for genes that are very similar to each other, the gene that is higher in the list currently is likely a better gene to retain for future generations. The specific mechanism was whether the genes shared the same values for more than X% of the total gene length, with X manually varied depending on the number of genes and level of randomness in a particular group of champions. This similarity metric was applied whenever the champion list grew past 200.

5 How to make champions robust

After several weeks with this setup, we decided to include two additional searching stages, for a total of three, which the algorithm would then switch between as a function of the current iteration count:

1. Hill-climbing and genetic search around the dissimilar (unique) champions
   The reason: We wanted to expand the number of local minima that we would be searching

   The idea: Find the champions that are least similar to each other. These are now our tester genes.

2. Hill-climbing and genetic search around dissimilar adversarial genes
   The reason: We found that our best performing champions did not beat all champions.

   The idea: Find all genes within the champion list that outperform the best champion. Eliminate the similar genes within this list. This now becomes our tester genes as we will be searching for genes that can outperform our best champion.

3. Hill-climbing and genetic search around champions most similar to our best champion
   The reason: As the number of iterations performed by our search increased, we assumed that our best champion was indeed a relatively well-performing champion

   The idea: Perform a search around all genes similar to our best champion so as to locate the best of that local minimum

We found considerable success following the implementation of the above three strategies, but we were concerned that eventually, the search would center around only a few local minima. To counteract that possibility, we enacted several more modifications, which acted to introduce stochasticity to the search. Previously, we had selected a single cross-over point for each iteration of gene evolution. We instead produced a different cross-over point and a different number of genes to mutate for each gene in every iteration. To introduce further stochasticity, we included

randomly generated genes with our tester set with the idea being that a search for potential champions that must beat previous champions, standard genes, and random genes will be more effective than a search for those that beat only the previous champions. Finally, we saved our current champions and restarted the process from scratch. While this final step was not fruitful, it did lead to a number of potentially fruitful gene paths that our initial evolution had not found. Furthermore, the lack of highly effective champions in this restart contributed to a sense of confidence in our initial implementation.

To find additional adversarial genes we started several runs with an initial random population of 200 genes and set our champion genes as a reference point of the scoring function. In each iteration, we would save at most the top 100 unique genes, add 80 random genes, add 60 crossovered genes, and 40 mutated genes. By doing so we were exploring a wide space of possible genes that was leading to finding many adversarial genes. In Figure 3, we compare the top 10 genes of each iteration against our champion genes. As we started from random genes, they were losing in the beginning, but they eventually learned how to beat the scoring set.

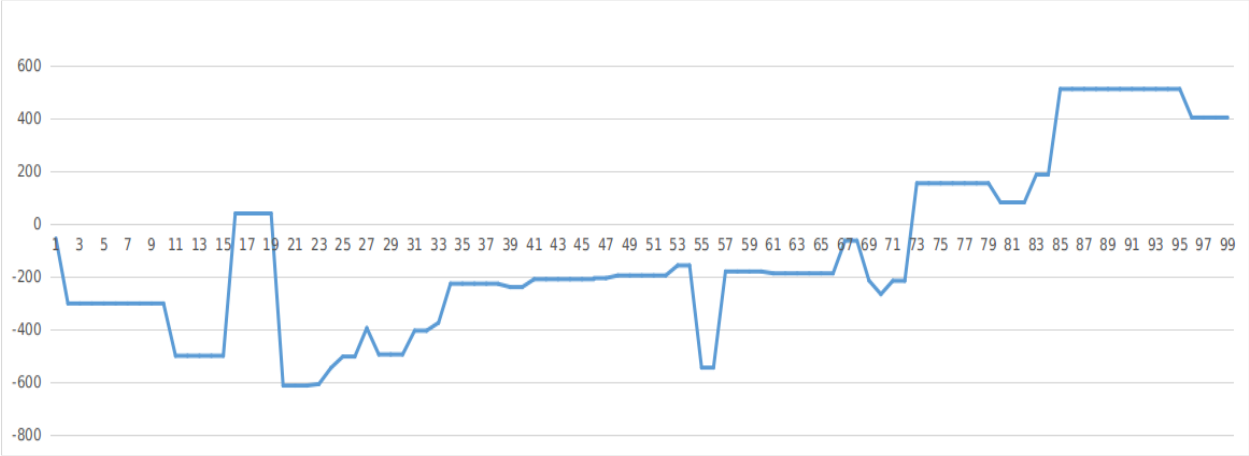

Figure 3. Comparison of the best 10 genes of each generation against the best genes that we are using in the scoring function.

As we reached a plateau, we tried to grow our population further creating the scoring function from the top 40 genes from the current population. We were running 500 iterations and tracking the score of the top 10 genes against previous champions. In Figure 4 we can see that not having

any of the previous champions in our scoring function results in losing the ability of the current population to fight precious champions, even though the top genes of every new population were beating previous.

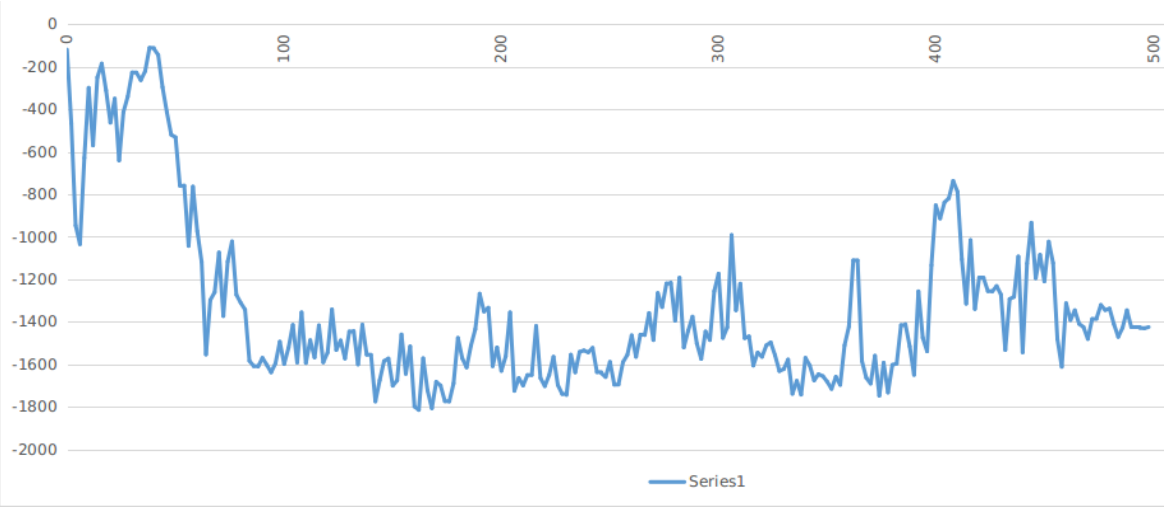

Figure 5. Comparison of top 10 genes in each population against previous champions across 500 iterations (with no champions in scoring function)

6 How to make iterations fast

When it comes to performance, gene evolution contains several tasks that could be easily parallelized in each iteration. When we are trying to generate scores of the current population, we perform a battle of each population against scoring genes. As each battle is independent of others we are using *ProcessPoolExecutor* from *concurrent.futures* library to submit all tasks in parallel. In the same way, we can calculate a similarity table, based on which we keep unique genes.

On the other hand, the basic operation of crossovers and mutation does not include heavy computations, and launching these steps in parallel doesn't benefit that much. Nevertheless, we decided to implement heavier versions of these functions with more computations and make these steps running in parallel too. Instead of just flipping some genes or pasting subsequences from one to another gene, we are running several iterations of creating a new gene and comparing it against its parent. If the new gene is stronger, then it is becoming a parent in the

next iteration, while if it is weaker, we dismiss it. In Figure 5 we are showing CPU utilization of 8 cores on an Intel i7 (10th generation) processor.

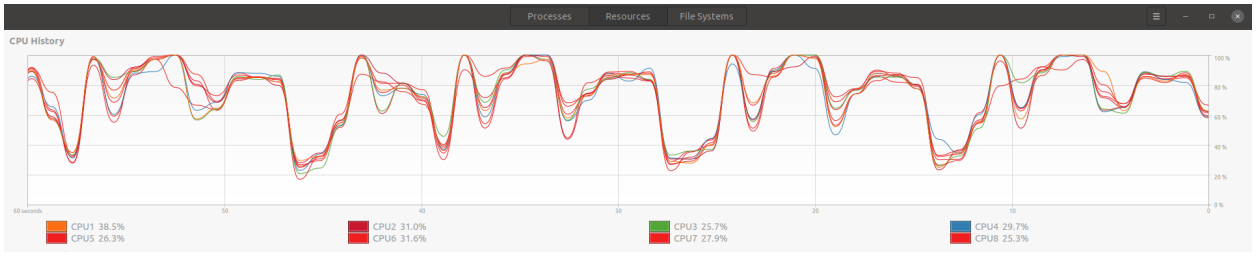

Figure 5. CPU utilization during the execution of our algorithm

7 How to select the best gene

Throughout the semester, competitions between student groups were held and our selected genes performed very well against other champions. However, with the recombination of our champion lists, the gene we submitted for the competition before the finals did not perform as expected. In Figure 6 we present the performance of our genes across the semester.

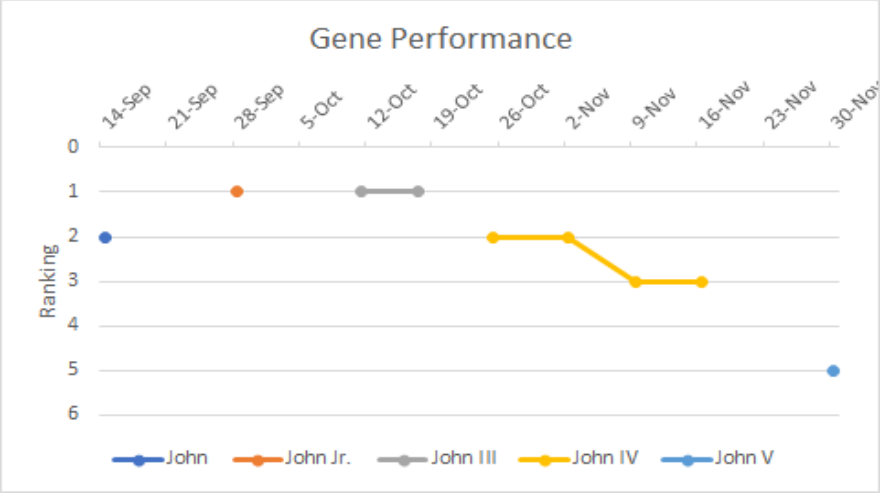

Figure 6. Gene performance in the weekly class competitions

Due to the relatively poor performance of John V, we removed the stochastic innovations from the previous paragraph with the idea being that finding and investigating any more local minima

was not feasible with the remaining time. Additionally, rather than removing similar genes once the number of champions grew beyond a certain number (due to computational demands), we kept a certain percentage of our top-performing genes and removed similar genes below that performance threshold. Then, investigated the current characteristics of our champion list so that we could invest our remaining time in a more selective manner.

The first idea was to try to focus our local search toward genes with higher entropy across our best performing genes. The assumption here was that genes with lower entropy form some kind of positive behavior that we should not change. In Figure 7 we can see how different gene types are distributed across the whole gene sequence of our best 50 genes. Based on these counts we were able to calculate the entropy for each position in Figure 8. By using this information we could better guide our search.

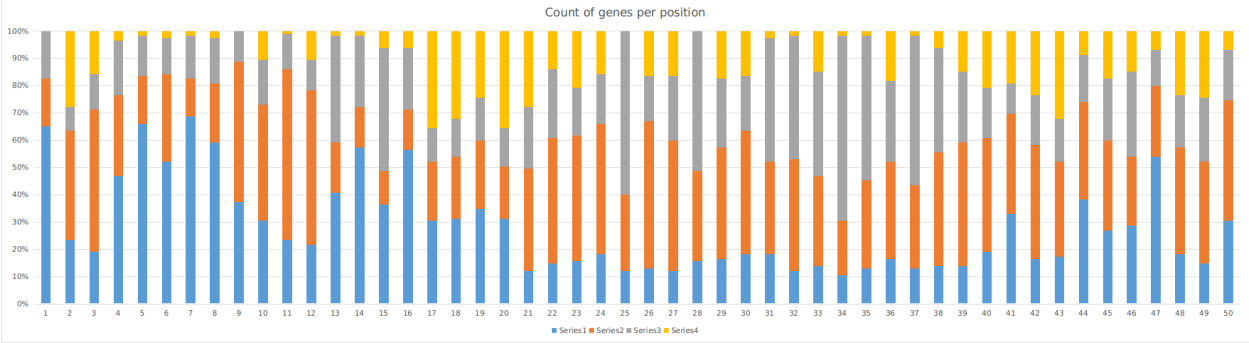

Figure 7. Accumulative count of gene types for each position for the top 50 genes

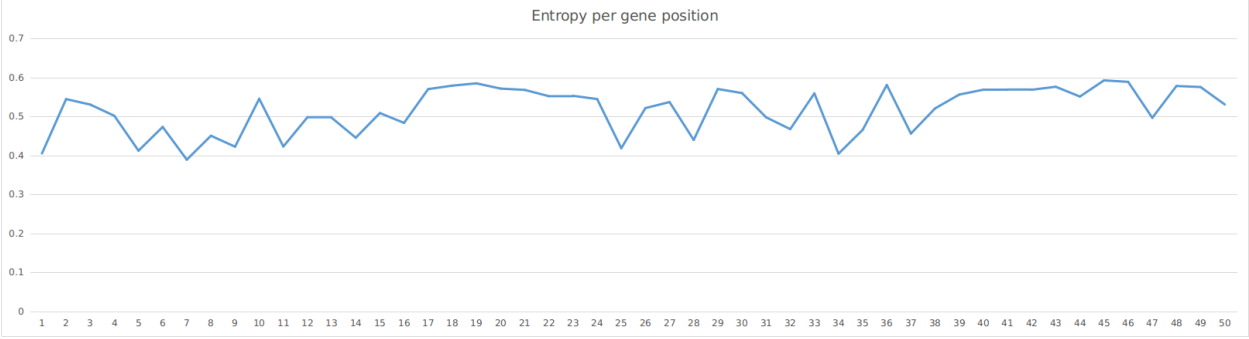

Figure 8. The entropy of each position in the gene for the top 50 genes

Finding local hills with K-means

The final stage in this process was to select a single gene for final submission to the competition. One simple approach would be to perform a competition between all of our produced genes and select the best performing gene. As the relative performance between genes may be biased due to the significantly higher representations amongst our gene population of the hills where we spent most of our exploration efforts and because John V performed poorly, this was not deemed to be an adequate selection approach. Another considered approach included creating clusters based on similarities, with clusters centered on genes higher on the champion list. But, as our final gene population included genes from multiple lists, this clustering mechanism would be biased on how we initially rank our gene population. Instead, we decided to implement a clustering algorithm mechanism for this selection, specifically, k-means clustering.

We hypothesized that k-means would provide information on the true number of local minima that existed within the gene population. The idea was to train a k-means clustering algorithm based on a variable number of k clusters. The individual clusters within each session would then compete with each other, leading to k champions. These k cluster champions would then compete against each other, resulting in a single champion for that specific selection of k. Finally, the single remaining champions from the variable number of k clusters would compete against the champions produced from other k-means training sessions. The assumption was that the resulting champion would perform well against the other local minima within this space, local minimas that our competing classmates would have potentially searched. With our initial k-means training, the highest performing champion across all k was reproduced in roughly three areas: 250 k clusters, 700 k clusters, and 900 k clusters, which is presented in Figure 9.

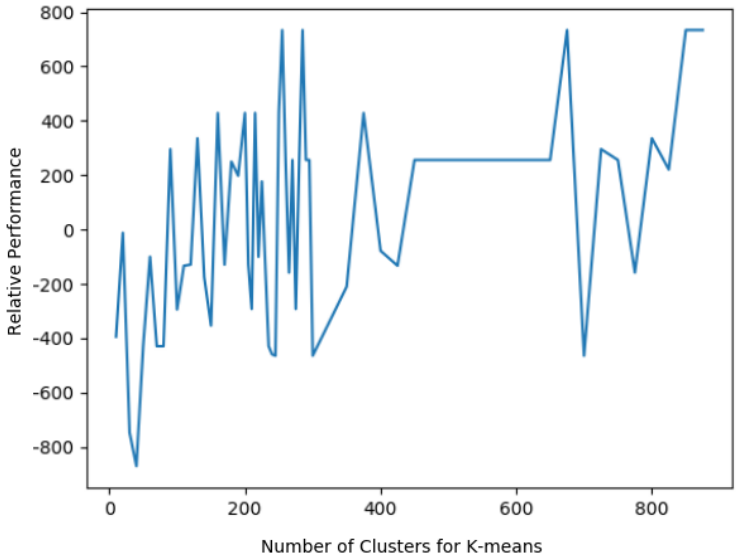

Figure 9. Performance of the cluster champions as a function of the number of k-means clusters

The highest performing gene at a k cluster count of 250 was not the most produced champion gene through this framework. Instead, there was a plateau from 500 to 700 k clusters. This could indicate that the true number of local minima within our gene population was within this range and that the resulting champion within this range would be more effective across a larger number of local minima. With this information, a final gene search session was performed.

Final search

In this final search, we collected the well-performing inter-cluster champions from the k-means clustering (k equal to 250, 500, and 900) and combined these with the current collection of champion genes. This group was included as a gene population for our genetic search with the idea that our final search would focus on searching only our best performing hills as produced by k-means clustering. Other genetic algorithm parameters for this final search include:

Parameters of the final search	
Gene population:	Champion genes from k-means, best performing genes from previous generations
Phases:	Local and genetic search around genes adversarial to best, Local search around unique champions, Local search around best performing gene
Iterations per phase:	30 iterations
Cross-over range:	Random over the range 8% (4 genes) to 92% of the gene length
Number of mutations:	Random within 1-4 genes per mutation
Stochasticity:	10% of new random genes
Elitist percentage:	80%

Finally, the k-means clustering analysis was performed again, which led to Figure 10.

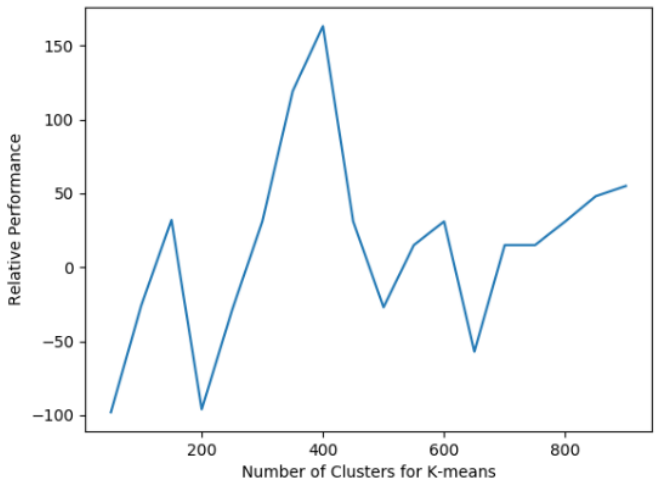

Figure 10. The final k-means analysis following the final gene search

By incorporating the genes produced from our final gene search based on our initial k-means analysis, the best performing gene is produced with a k equal to 400 clusters. Interestingly, by searching with the additional information from k-means, we no longer produce a plateau in the 500-700 range. Instead, our above results indicate that the approximate number of local minima within our total gene population is 400. Our final gene selection is:

King John:

[0, 3, 1, 3, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 3, 3, 3, 3, 3, 1, 1, 1, 2, 1, 1, 2, 1, 1, 2, 2, 2, 2, 2, 1, 2, 2, 1, 3, 1, 1, 3, 0, 1, 2, 1, 1, 3, 1]

CONCLUSION

In this report, we were describing our methodology in finding PacWar best performing gene. In the first place, we applied several genetic and local search algorithms to find initial groups of strong genes. Then we focus on exploring the space around good adversarial genes and unique genes that have the potential to reach an even higher hill. By including such genes in the scoring function, we were avoiding overfitting and making our genes more robust to unseen genes. In the end, we cluster our genes and found their local hill champions, so we could finally find the gene that performs the best across all hills. By using the described methodology we believe that our PacMite will achieve high performance and become the Lord of the Hills.

REFERENCES

[1] https://www.clear.rice.edu/comp440/old/fall2007/pac.html