Pacwar report

September 13, 2021

1 Introduction

We tried multiple local search algorithms to attack this problem which include the Hill Climbing, GSAT algorithm and Genetic algorithm. After trying different algorithms, we found that GA is the best way to attack this problem which we maintain a gene pool that contains various powerful genes and update the gene pool by adding a new gene that aims to beat all the old genes. To increase the diversity of the gene pool, we modify the evaluation function and generate genes with different tendencies. We also introduce two different crossover techniques and use variational autoencoders to initialize our algorithm in order to improve efficiency. We finally take the crossover of several optimal genes and explore their one-component and two-component neighbors to obtain the final result.

2 Solution Strategy

2.1 Hill climbing

2.2 GSAT

We apply GSAT algorithm to attack this problem. GSAT is a similar algorithm as the random hill climbing and hill climbing. However, GSAT traverses all the neighborhood and have two part randomness which come from random restart and random good neighbors. The disadvantage of GSAT is that we need a good evaluation gene set to evaluate gene and it is not easy and fast to get it from this method.

2.3 Genetic Algorithm

Our strategy is to generate a gene pool G that contains powerful genes, and then find an optimal gene that can beat all the genes in the gene pools. The procedure is as follows:

1 Using genetic algorithms, hill-climbing, simulated annealing algorithms that aim to beat all 0, all 1, all 2 all 3. We use the output genes to initialize the gene pool G.

2 Set the parameters of evaluation function (different parameters generate genes with different tendencies).

2 Run genetic algorithms with opponents G and put the optimal gene into G.

3 Go to step 2 until we have enough genes in the G.

5 Rank all the genes in the G by combating each other. Put the top n genes into an optimal pools G_{opt}.

6 Refine the gene in G_{opt} via crossover and explore the one-component and two-component neighbors.

Our basic idea is to maintain a gene pool with good performance and enough diversity. We refine this pool by repeating step 2 and 3. In step 3, we always put the optimal gene into G which guarantees the performance of G. In step 2, we modify the evaluation function in order to generate different types of genes (for example, defensive gene, attacking gene, etc.) and guarantee the diversity of G. Finally, we pick several good genes from the pool and refine them further to generate the final solution.

2.4 GA+GSAT

We coupled GA with GSAT algorithm by changing the random mutation into a GSAT algorithm which decrease some of the randomness from random walking on the gene space. However this make the iteration very slow and the overall result vs time is even worse than GA. So we finally abandoned this algorithm and start to improve the GA.

3 Genetic Algorithm

The core algorithm we used is genetic algorithm (GA) which is shown below:

1 Initialize population $G_{population}$ and opponent pool be $G_{opponents} = G$. Both are lists of genes.

2 Select top 10% of the genes in the population $G_{population}$.

3 Crossover and mutation.

5 Score and rank the new population. Put the best gene into $G_{opponents}$ every 30 iterations.

5 Go to step 2. Stop after 300 iterations or the standard deviation of the population is smaller than 0.1.

3.1 Evaluation Function (Fitness function)

The fitness of a gene $g \in G_{population}$ is defined as:

$$
\begin{aligned}
S(g) &= w_1 S_{population}(g) + w_2 S_{opponents}(g) \\
S_{population}(g) &= \frac{1}{|G_{population}|} \sum_{g' \in G_{population}, g' \neq g} score(g, g') \\
S_{opponents}(g) &= \frac{1}{|G_{opponents}|} \sum_{g' \in G_{opponents}} score(g, g')
\end{aligned}
$$

where $score(g, g')$ denotes the score gene g get after battling against g'. The fitness is a weighted sum of two part. The first part denotes the score obtained by fighting with genes in the populations and the second part represents the score of fighting the genes in the gene pools.

We test different combinations of weights: $(w_1, w_2) = (0.5, 0.5), (0.0, 1.0), (0.1, 0.9)$. In the case of $(w_1, w_2) = (0.5, 0.5)$, the algorithm put much effort on beating the other genes in the population. The $S_{opponents}$ score is relatively low. We also find that overall performance of this case is poor. Considering the fact that the average performance and diversity of $G_{opponents}$ is better then $G_{population}$, $S_{opponents}$ should give the correct evolution direction. So we increase the weight w_2 and try $(0.1, 0.9)$ and $(0.0, 1.0)$.

To test the efficiency of different parameters, we initialize $G_{opponents}$ with 10 genes selected from G. Then we use different parameters to generate the optimal gene. The result after 100 iterations is shown below:

(w_1, w_2)	$S(g_{opt})$	$S_{opponents}(g_{opt})$
$(0.5, 0.5)$	15.8	12.4
$(0.0, 1.0)$	16.1	16.1
$(0.1, 0.9)$	16.4	16.3

Include result

From the result, we find that even though the average score S is good for $(0.5, 0.5)$, it doesn't imply a good performance on opponents. The performance of $(0.0, 1.0)$ and $(0.1, 0.9)$ are similar. Finally, we decide to use $(0.1, 0.9)$. In this case, the $S_{population}$ would help us to pick out the better gene when two genes have similar $S_{opponents}$.

3.2 Score function

Our score function $score(g, g')$ also contains two part. $score(g, g') = score_{org}(g, g') + score_{cus}(g, g')$ with the first part is the original score and the second part denotes the customized score. Our customized score is defined as

$$score_{cus}(g, g') = s_{death} I(g \text{ has zero pacman left}) + s_{lose} I(g \text{ loses}) + s_{tot\_win} I(g' \text{ has zero pacman left})$$

The gene g would loss $-s_{death}$ score if it has zero pacman left after the battle and loss $-s_{lose}$ score if it loses and gain s_{tot_win} scores if its enemy has zero pacman left.

We tune the value of $s_{death}, s_{lose}, s_{tot\_win}$ to get different type of genes. Decreasing s_{death} would enforce the gene to survives even fighting with powerful enemy. Decreasing s_{lose} would generate defensive genes, which instead to try to reach a high score but to not lose. Increasing the s_{tot_win} gives more attacktive genes, they would try to kill all the pacmans in order to reach a high score. We test the genes generated by different parameter settings. We show the average behaviors of different genes on a opponent set with size 15 (150 iterations):

Type	$(s_{death}, s_{lose}, s_{tot\_win})$	$S_{opponents}$	Winning rate	# of Pacmites	# of opponent Pacmites
Normal	$(0, 0, 0)$	14.4	0.87	100	26.6
Don't die	$(-2, 0, 0)$	13.3	0.80	84.3	37.5
Defense	$(-1, -2, 0)$	16.8	0.87	110.1	17.2
Win	$(0, 0, -2)$	15.4	0.93	106.0	21.4
Attack	$(0, 0, 2)$	16.6	0.87	113.0	17.0

It's worth mentioning that, in the Win type, we reduce the score gained by kill all the opponents. Then instead of killing all pacmites for specific opponents, the gene during evolution would try to win as many opponents as possible.

3.3 Crossover and Mutation

We use two types of crossover: bit crossover and segment crossover. Given two gene g_1, g_2, for the bit crossover, for each position we pick one component from either g_1 and g_2 (50%-50% probability). For the segment crossover, we divide each gene into 6 part based on the battle ruls: $[U] = g[0:4], [V] = g[4:20], [W] = g[20:23], [X] = g[23:26], [Y] = g[26:38], [Z] = g[38:50]$, and we generate new gene by pick a segment of gene from either g_1 and g_2 (50%-50% probability). In the segment crossover, we try to preserve the optimal combination of components within each segment. In our experiment, we found the segment crossover outperform the bit crossover (shown below).

However, only using segment crossover may reduce the diversity of each segment. In order to explore more possibilities of each segment, we introduce both segment and bit crossover. At each iteration, we randomly pick segment crossover and bit crossover with probability $p_{segment} = 0.9, p_{bit} = 0.1$. We test the efficiency of bit crossover, segment crossover, and hybrid crossover as follows:

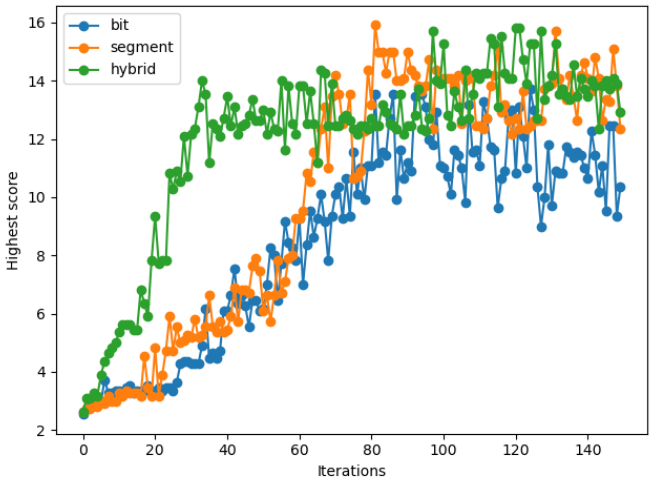

Figure 1: Performance of different crossover. The highest score denotes the highest score of the genes in the population at each iteration.

Obviously, The bit crossover basically has a poor performance and our choice gives a better score.

For the mutation, we set a mutation rate $p_{mut} = 0.05$. For each component of each gene, it mutates with probability p_{mut}.

3.4 Initialization with Variational Autoencoder

After the size of gene pool G (or the opponent genes $G_{opponent}$) reach the size of ~ 40, the convergence of genetic algorithm becomes quite slow. So we initializing the population via a variational auto-encoder instead of randomly initialization.
The architecture of vae we used has the following structure:

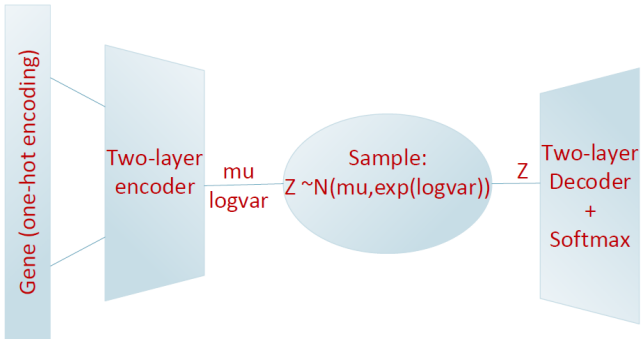

Figure 2: Autoencoder decoder. Both encoder and decoder are two-layer feed forward networks with ReLU activation and number of hidden units (30,30).

The input of the encoder is a gene x with one-hot encoding. The output of the encoder is two vectors $\mu, logvar$. Then we sample the input of decoder by distribution $N(\mu, e^{logvar})$. The output of decoder is a $y = 50 \times 4$ matrix with $y[i, :]$ denotes the probability distributions of component i. We feed the vae via the top 30 genes from the gene pool G and train the vae by minimizing the cross-entropy

$$L(x, y) = \sum_{i=0}^{49} \sum_{j=0}^{3} I(x[i, j] == 1) \log(softmax(y[i, :])[j])$$

After the training procedure, we use vae to generate gene. To do so, we first calculate the mean and variance of $\mu, logvar$ over the training data set: $mean(\mu), mean(logvar), var(\mu), var(logvar)$.

We then generate new $\mu, logvar$ from $N(mean(mu), var(\mu))$ and $N(mean(logvar), var(logvar))$. And finally we generate the new genes from encoder. We test the quality of the generated genes:

Number of generated genes	Average Score	Highest Score	Mean Std
30	7.4	15.6	0.90

where the Mean std is defined as the average of standard deviation for each component of gene.

Finally, we compare the efficiency of vae initialization and random initialization:

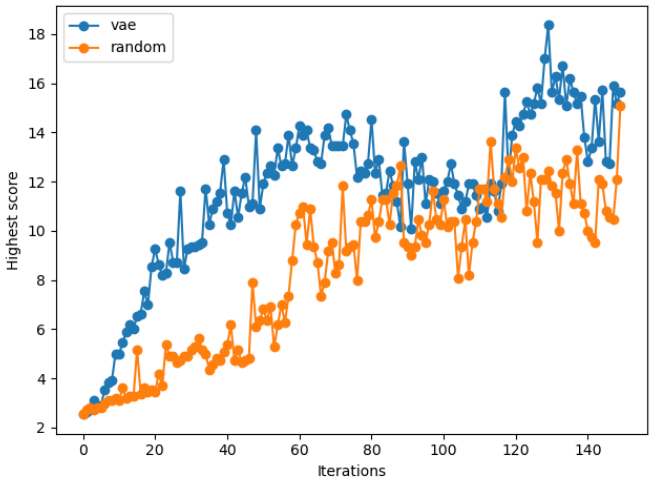

Figure 3: Highest score vs iterations with different initialization.

We observe a much faster convergence and better performance with vae initialization. But using vae initialization may not allow us to explore new types of genes, so we initialize the population half with vae and half with random.

4 Final refinement

We generate the gene pool G with 55 genes. They are generated with different evaluation function:

Number of genes	Type of evaluation
5	Defense
5	Attack
15	Don't die
10	Win
20	Normal

We also show the score of each gene vs the other genes in the G.

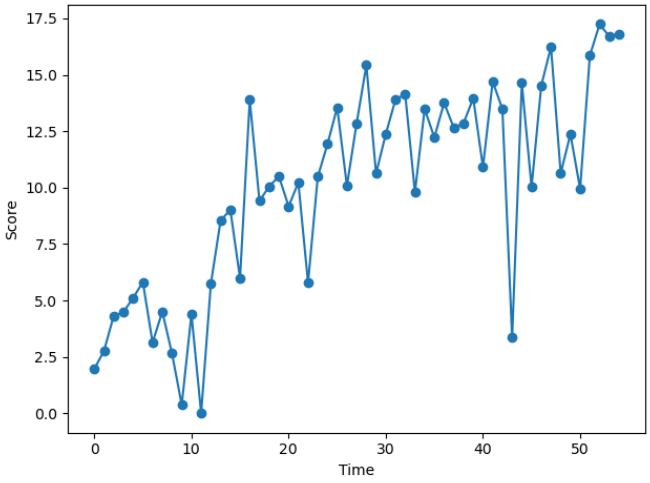

Figure 4: Score of all the genes. The time denotes the time order the gene is generated.

After setting up the target gene pools for final refinement, we apply 2 kinds of refreshment following the guidance of the last blog. One is searching the neighbourhood and the other one is crossover two distinct genes.

4.1 Search neighborhoods

Searching the neighborhoods is basically using Hill climbing algorithm and evaluate it with the gene discussed above. This ensures that our gene sit in a local maximum. We first apply the 1st neighbor local search to find the gene and we found that there is no improvement at all. The best gene returned is still the original gene. This means that right now our gene is sitting in a local minimum in 1st neighbor. The total iteration is 150 (50*3) in space.
Then we apply a second neighbor which asks a 11025 (50*49/2*3*3) iterations on gene space which cost about 10mins on my laptop. This increase the search space which create a slightly

better genes than previous one with 0.5 scores higher.

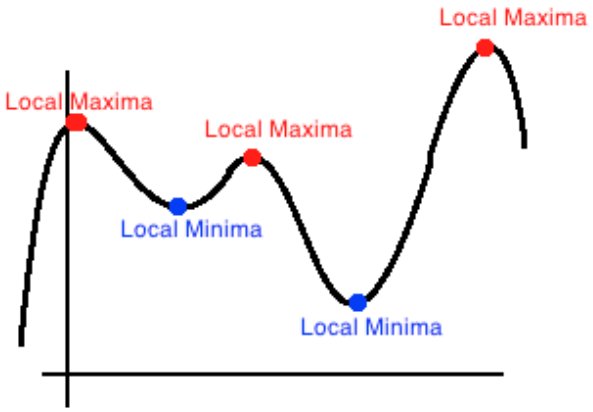

Figure 5: Schematic diagram for local maximum.

We also observe the symptom that most directions (50 dimensions) around our strong gene are cliff which get extremely low points. This might apply that more mutation is also hard to get a better gene from these genes.

4.2 Crossover two different genes

We apply a random crossover between two distinct gene set and both of them get good scores on the evaluation set that we discussed previously. These two genes are after polished by HC with both 1st and 2nd neighbor. We apply both random crossover for both segment wise (U, X, Y...) and bit wise. The segment crossover will create 2^6 different new genes, while the bit crossover will create 2^{30} since we have 30 different bits between these two genes. The segment crossover could be easily traversed. However, the search space for bit wise traversal is still too large. So, we make sure that the genes are the best among the segment-wise crossover genes and finite time 100k bit-wise crossover.

Gene score	gene
16.767	031300003003032303331121221111211221221211301201 31
16.317	000200000100202200301211231231231233232233131133 13