Genetic Programming Rice Categorization Problem with Pyramid Search

COSC 4P82 Final Project

Thanushan Pirapakaran 6890206
Adrian Binu 6970677
Brett Terpstra 6920201
COSC 4P82
Prof: Brian Ross
Brock University
April 20th, 2024

Introduction

Given a set of scrambled datasets of rice, train a genetic program to find a function that can accurately categorize the rice. In particular, the dataset is of seven descriptive variables: area, perimeter, major axis length, minor axis length, eccentricity, convex area, extent, and class. Thus, the program should utilize these first seven variables to develop a function that effectively classifies each rice sample as either Cammeo or Osmancik.

Furthermore, in every GP algorithm fashion, we must find the most optimal settings that provide the best performance. This is in terms of vanilla GP variables in addition to new pyramid search variables.

For this paper, we build upon the problem above by testing the effects of pyramid search. Additionally, we test whether the claims made in Ciesielski and Li's paper on Pyramid Search (listed in the Bibliography), are true for the following use case.

Pyramid Search Claims

Essentially, Pyramid Search is a training strategy that aims to overcome the limitations of basic GP systems. In particular, pyramid search is best for large-scale problems and complex fitness landscapes with multiple local optima. In other words, one of the major use cases includes improving premature convergence. For this reason, we chose a GP rice problem algorithm (from assignment 1 part b) with a premature convergence problem as illustrated in Figure 2 and Figure 3.

Furthermore, this technique is supposed to be faster than running a basic GP program, as badly performing subpopulations get pruned. This is also the reason that Pyramid Search consumes less memory than a basic GP, over a duration. All in all, Pyramid Search claims that there is a higher chance of success with fewer evaluations. Thus, we expect the program with Pyramid Search to be more accurate, faster, and less memory-demanding, than the basic GP. Additionally, we expect Pyramid Search to solve the algorithm's premature convergence problem.

Experiments

The Basic Genetic Programming is illustrated in Table 1.

Parameters	
Pop. Size	8000
Crossover Rate	90%
Mutation Rate	10%
Generations	50
# of Elites	2
Min Tree Size (Init)	2
Max Tree Size (Int)	17
Min Tree Size (Mut)	0
Max Tree Size (Mut)	17
Tournament Size	7
Testing Split	70%
Training Split	30%

Table 1: GP Parameters

These parameters are used in the experiments for both the rice problem with and without Pyramid Search. The specific Pyramid Search parameters are illustrated in Table 2.

Parameters	
num_pops	10
pop_size	800
num_gens	5
prune_ratio	0.2
max_gens	50

Table 2: Pyramid Search Parameters

The specific parameters will be explained in the Pyramid Search Algorithm section. Nonetheless, we decided to choose the same Pyramid Search parameters as Ciesielski and Li's paper. This was done to test the best parameters that they found and compare our findings to theirs.

Language

Similarly, the language is identical for both programs with and without Pyramid Search. This can be seen in Table 3 under Appendix.

Rice Problem Fitness Function

The fitness of an individual is based on how many hits an individual can get. A hit is measured by how much rice it can guess correctly. Essentially, it gets the predicted value from the GP after it was calculated with the 7 float variables listed in the float array and then just like shown in Figure 10

```
If (GP_ans >= 0.0  and ans[i] = "Cammeo")
   then hits = hits + 1;
Else if (GP_ans  < 0.0 and ans[i] = "Osmancik")
   then hits = hits + 1;
```

Figure 10: Fitness Function Pseudocode

we used the predicted value and categorized it. If it was greater than 0 then GP predicted Cammeo otherwise it was Osamancik. Then we compared the gp answer and the actual answer and if it was right then gave it a plus one hit.

Pyramid Search Algorithm

This technique takes after the island model, in that it uses subpopulations to fix certain basic GP program issues. Unlike the island model though, Pyramid Search uses subpopulations that work completely independently. The Pseudo Code in Figure 1 illustrates the process behind Pyramid Search.

```
def pyramid(num_pops, pop_size, prune_ratio, num_gen, max_generations)
  popsleft = num_pops
  while(not solved and popsleft != 1 and generations < max_generations
    evolve_pops(pop_size, num_gen)
    remove_least_fit_pops(round(popleft * prune_ratio))
    popsleft = popsleft - round(popleft * prune_ratio)
    generations += num_gen
  }
)
```

Figure 1: Pyramid Search Pseudo Code

There are 5 additional Pyramid Search variables to note:

1. Num_pops: the number of subpopulations within the run
2. Pop_size: the population size of each subpopulation
3. Num_gens: the number of generations between each prune/layer in the pyramid
4. Prune_ratio: a number used to calculate how many sub-populations get pruned every num_gens
5. Max_generations: the total number of generations that need to be evolved in the run (This variable is equal to the basic GP gen variable)

Essentially, the program loops until max_generations or a solution has been found. Within a loop, the program runs each sub-population independently, then when num-gens has been reached, it ranks the fittest individuals from each subpopulation and prunes the least-performing population(s). The number of population(s) to prune is calculated by multiplying the prune_ratio by num_pop and rounding up to the nearest whole number.

Dataset

The dataset is of about 3800 rice samples, with eight separate rice data points. The first seven data points are numeric descriptive variables, while the last one is the class of the rice (Cammeo or Osmancik). In other words, the first seven data points are the GP's input variables, and the last data point is the GP's target variable. As the data linked in the bibliography is ordered in terms of rice classification, we must scramble the data to ensure that the training set consists of a variation of data samples. This ensures that the training set includes a diverse range of data samples to achieve better performance with less data. Thus,

minimizing memorization and promoting generalization.

Variation of Experiments

Our experiments revolved around testing all Pyramid Search claims. As a result, there were four main tests.

The first experiment was about comparing the premature convergence found in the rice problem program without Pyramid Search to the program with it. This will allow us to test whether Pyramid Search truly fixes premature convergences.

The second experiment tests the claims about speeding up the GP program. As a result, we compared the duration of each run, so that we can compare the differences with and without Pyramid Search.

The third experiment revolved around memory usage. To test this we kept track of the memory demand throughout all runs. To then compare and contrast.

Finally, Pyramid Search claimed to create more accurate answers, as a result, we intended to test the accuracy of the rice identification process.

Results

Pyramid Search gave us slightly better results but very minor. Pyramid Search gave us an improvement in performance in terms of speed, and accuracy of the test and lowered memory consumption over time. Firstly, it reduced execution time from **6.786 seconds to 4.6055 seconds** using Pyramid Search. As a result, you can see that it is slightly faster and if you were to scale this problem up even more we believe you would see a bigger gap between the two algorithms. The big reason for the speed-up is because it prunes out the worst fit runs periodically, so they are removed from the search space. This allows the algorithm to focus its resources on more promising runs that have not prematurely converged yet. This avoids spending resources on runs that are not making satisfactory progress early on.

Figure 7: Pyramid Search Memory Usage

Not only does this help with speed but it also helps with lowering memory consumption over time (as can be seen in Figure 7). After every 10 generations, it prunes out the worst-performing run.

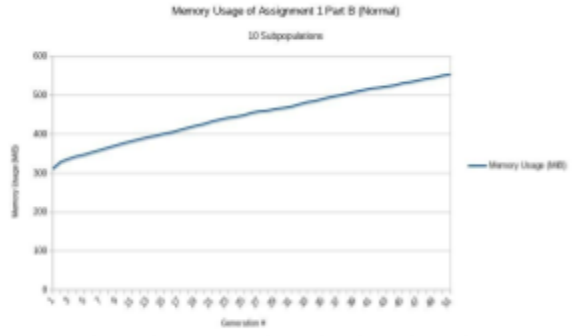

Figure 6: Basic GP Memory Usage Graph

For this reason, the memory consumption drastically decreases and then it slowly ramps up again. This is because the other runs take more space in memory as they keep getting bigger in terms of having bigger and more complex individuals.

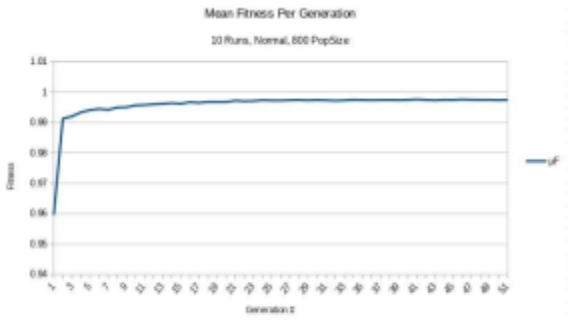

Figure 2: Basic GP Mean Fitness Per Generations Graph

This process continues until the algorithm stops and only one run is left. In the end, you can see in Figure 7 that the model uses about 110 megabytes compared to when it started using 320 megabytes.

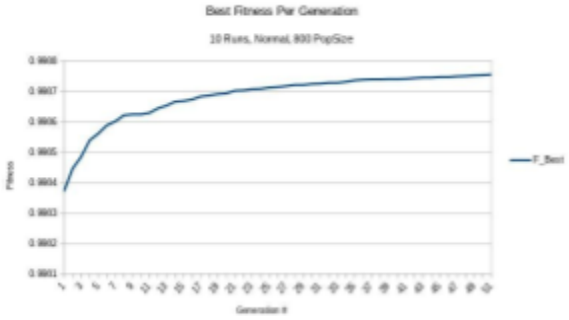

Figure 3: Basic GP Best Fitness Per Generation Graph

Compared to vanilla GP in Figure 6, it starts off using the same amount of memory as Pyramid Search in Figure 7. However, the Figure 6 graph slowly ramps up to 550 megabytes which is actually a big difference between the two algorithms.

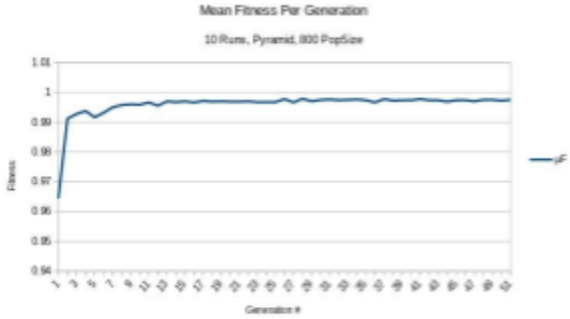

Figure 4: Pyramid Search Mean Fitness Per Generation Graph

However both algorithms converged and while Pyramid Search did give us slightly better results as seen in Figure 8 and Figure 9 where the accuracy went up from 80.31% to 91.99%,

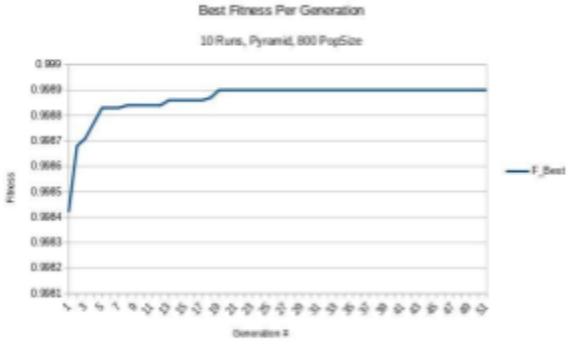

Figure 5: Pyramid Search Best Fitness Per Generation Graph

using Pyramid Search you can also see in Figure 2 and Figure 4 where both graphs prematurely converged in about the same area. Looking back at the confusion matrices you can see that in Figure 9 the Pyramid Search algorithm got better results in almost every aspect but by only slight margins; it got lower false positives and false negatives but it got bigger true positives and true negatives.

Testing Set			
TARGET / OUTPUT	Cammeo	Osmancik	SUM
Cammeo	945 33.65%	370 13.18%	1315 71.86% 28.14%
Osmancik	183 6.52%	1310 46.65%	1493 87.74% 12.26%
SUM	1128 83.78% 16.22%	1680 77.98% 22.02%	2255 / 2808 80.31% 19.69%

Figure 8: Basic GP Confusion Matrix

Testing Set			
TARGET / OUTPUT	Cammeo	Osmancik	SUM
Cammeo	1022 36.37%	121 4.31%	1143 89.41% 10.59%
Osmancik	104 3.70%	1563 55.62%	1667 93.76% 6.24%
SUM	1126 90.76% 9.24%	1684 92.81% 7.19%	2585 / 2810 91.99% 8.01%

Figure 9: Pyramid Search Confusion Matrix

There are two more graphs that we didn't mention before and those are the best fitness graphs, specifically Figure 3 and Figure 5 there is a difference to note here. Figure 3 is the best fitness graph without Pyramid Search and you can see that the model takes a couple of generations before it gets a fitness of about 0.9987. On the other hand, pyramid search quickly gets there and even explores a bit more of the search space. This would mean that it is a little better at prematurely converging than without Pyramid Search, but the difference is very small and negligible.

Conclusion

For the paper they concluded that the Pyramid Search strategy offers a variety of ways of managing a run. Some of these are better than others. The best pyramid strategies result in considerable improvement over the standard approach of multiple runs to some maximum number of generations. We found this to be true, the pyramid search model was faster, slightly more accurate and lowered memory usage that is why we recommend you try the Pyramid Search strategy.

Terms

GP: Genetic Program

Basic GP: The rice problem without pyramid search

Vanilla GP: The rice problem without pyramid search

Appendix

Function	Arity	Example
ADD	2	x + y
SUB	2	x - y
MUL	2	x * y
Protected_DIV	2	x / y (1 if y = 0)
NEG	1	-x

Table 3: Function and Terminal Sets

```
Hits: 2580, Total Size: 2810, Percent Hit: 91.814947
CC: 1072
CO: 64
OO: 1508
OC: 166
Fitness: 0.998914
Hits: 920


y = ((extent + minor) + log(exp(((log(major) / (0.22371 - minor)) * (convex - (major + log(exp(perimeter))))))))

 (+ (+ extent minor)
    (log (exp (* (/ (log major)
                    (- 0.22371 minor))
                 (- convex
                    (+ major
                       (log (exp perimeter))))))))
```

```
=== BEST-OF-RUN ===
            generation: 47
                 nodes: 20
                 depth: 8
                  hits: 920
TOP INDIVIDUAL:


-- #1 --
                  hits: 920
           raw fitness: 920.0000
    standardized fitness: 920.0000
       adjusted fitness: 0.9989
TREE:
 (+ (+ extent minor)
    (log (exp (* (/ (log major)
                    (- 0.22371 minor))
                 (- convex
                    (+ major
                       (log (exp perimeter))))))))
TREE-equ:
y = ((extent + minor) + log(exp(((log(major) / (0.22371 - minor)) * (convex - (major + log(exp(perimeter))))))))
```

Figure 11: Best GP Expression Evolved Non-Pyramid Search

```
...
Hits: 2605, Total Size: 2810, Percent Hit: 92.704626
CC: 1036
CO: 117
OO: 1569
OC: 88
Fitness: 0.998899
Hits: 907

y = (log(eccentricity) / (log(((major + minor) + eccentricity)) * ((log(perimeter) - eccentricity) * ((((0.62716 * 0.41230) + (minor * convex)) + (area / minor)) + ((exp(perimeter) + exp(-0.90311)) -
(-0.03000 * extent)))) * ((log(minor) - ((minor * convex) * (area / major))) * (((convex * 0.91184) / (extent / ((eccentricity - area) * major))) * (exp(major) * (major * major))))))))

(/ (log eccentricity)
  (* (log (+ (+ major minor) eccentricity))
    (* (- (log perimeter) eccentricity)
      (* (+ (+ (+ (* 0.62716 0.41230)
                (* minor convex))
              (/ area minor))
            (- (+ (exp perimeter)
                  (exp -0.90311))
               (* -0.03000 extent)))
         (* (- (log minor)
               (* (* minor convex)
                  (/ area major)))
            (* (/ (* convex 0.91184)
                  (/ extent
                    (* (- eccentricity area) major)))
               (* (exp major)
                  (* major major)))))))))
...
```

Figure 12: Best GP Expression Evolved Pyramid Search

Bibliography

[1] Rice (Cammeo and Osmancik). (2019). UCI Machine Learning Repository. https://doi.org/10.24432/C5MW4Z.

[2] Poli, Riccardo, et al. *A Field Guide to Genetic Programming*. Lulu Press, 2008.

[3] Ciesielski, V., & Li, X. (2003). Pyramid search: finding solutions for deceptive problems quickly in genetic programming. *CEC: 2003 CONGRESS ON EVOLUTIONARY COMPUTATION, VOLS 1-4, PROCEEDINGS*, *2*, 936-943 Vol.2. https://doi.org/10.1109/CEC.2003.1299767