Darwin's Pancakes

Martim Viana

Faculdade de Ciências da Universidade de Lisboa

Abstract

Darwin's pancakes is an iterative deepening genetic toolkit that aims to solve one or more pancake sorting problems with as minimum moves as possible. The algorithm is guaranteed to find at least one sub-optimal solution on 'start until found' mode and, by tinkering with the various options in the 'start' mode, is guaranteed to find the optimal solution.

Introduction

The pancake sorting problem (Dweighter, 1975) is an approach to the sorting problem, in which it is only allowed to "flip the pancakes" or, in a more general term, it is only allowed to reverse a selected array from the stack, and insert it back to the stack. Initially, the pancake sorting problem was used for educational purposes only, however, it was later discovered that it could also be used for fault tolerant routing algorithms in the computer networks industry (Gargano, 1993).

For example, the array

1 4 3 2

Can be sorted by flipping the position 2 and 1, generating the following moveset:

0: 1 4 3 2
1: 1 2 3 4
2: 4 3 2 1

One of the main problems of the pancake sorting problem is that the amount of moves that it takes to sort is unknown, however, it is known that the worse combination of size n pancakes is lesser than $2n$ moves to sort the array. It can be said then that a "good enough" solution is lesser or equal than $2n$. On the same note, another major problem of the pancake sorting problem is to device a way of telling the algorithm when the individual when to stop searching because, if no stopping mechanism is created, the individual will keep generating new moves until the program runs out of memory.

All these topics will be touched upon in the 'Approach' section.

Approach

All individuals contain a list of problems that they need to solve, and a list of integers that represent the points that the individual used to "flip the pancakes", in order to solve the problems. The genes are, at first, randomly generated until a solution is found, or that the maximum moves threshold is reached. The maximum moves threshold originated as a stopping mechanism to prevent the individual from keep searching for the solution forever.

For a more detailed overlook on how the program works, the project's javadoc is also present in the project folder.

An Iterative-Deepening Genetic Algorithm

As a maximum moves threshold is applied to the individuals, a way to search for solutions that require more moves than the maximum move threshold allows is needed. As such, I've adapted the iterative principles of the Depth-First Iterative-Deepening algorithm (KORF, 1985) to solve this problem, where the maximum moves threshold is increased like the tree depth on the Depth-First Iterative-Deepening algorithm. Such implementations already existed for other problems (Yao, 2005), however, I couldn't found any IDGA implementations for the pancake sorting problem. On each threshold increment, the algorithm executes a genetic algorithm with a brand-new population. While this implementation does not guarantee the optimal solution, practical results show that the "quality" of the solutions found is greatly increased.

Alternative Genetic Algorithms

On the toolkit, two other types of genetic algorithms are also present: The $rununtil found$ genetic algorithm, as the name implies, will attempt to find any solution, no matter it's quality, while the $rungenerations$ genetic algorithm will only run a specific amount of generations, and will return the best solution that it found.

Implementation

In this section, I will detail how the In this section, I will explain how the program works "under the hood", detailing

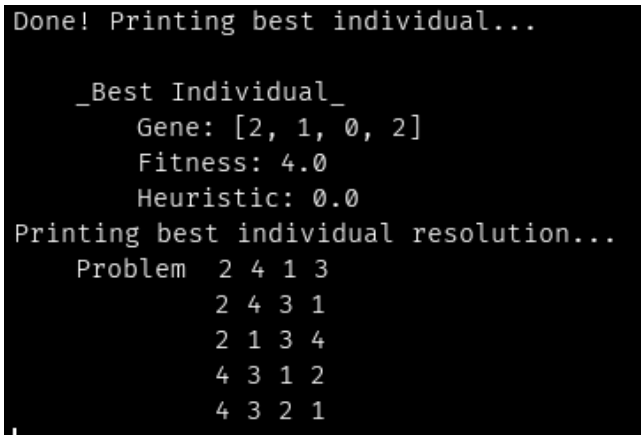

Figure 1: A screenshot taken from the program, where it found a solution for the problem assigned.

only the methods that are called, however, it is advised for the reader to call other operators, be it already present in the $Population$ class or created by themselves.

The Iterate method

The iterate method is where all the genetic operators are called, it is where all population-handling methods are called, following an architecture based on what was lectured by professor Luís Correia, during the Artificial Life discipline. The following pseudo-code is an abstraction of what is happening in the iterate method:

Result: Population with selected genetic operators applied
$selectedamount \leftarrow initialsize/(generation + 1)$
$mutatedamount \leftarrow pop.size - 100$
$decimatedamount \leftarrow selectedamount$ Randomly select $selectedamount$ individuals;
Perform single point recombination on selected individuals;
Perform bit-flip mutation on random individuals on a 80% chance;
Decimate $decimatedamount$ worst individuals;

Algorithm 1: The Iterate method.

Where $generation$ is an integer indicating the current generation that the iterate method is in.

As the reader can observe from the algorithm, I've chosen the single point as the recombination operator, whose selected individuals for recombination are chosen without taking into consideration it's fitness, this decision was made for two reasons:

- The fact that a individual is a solution or not is doesn't affect the probability of possibly holding a better solution;
- The fitness function chosen may lead to premature convergence, as there is a huge discrepancy between individuals that found a solution and individuals that didn't.

Single Point Recombination

The following pseudo-code is an abstraction of what is happening to the selected individuals during their single point recombination:

Result: Offspring generated by recombination
$minsize \leftarrow min(gene.size, partner.size)$;
$point \leftarrow randominteger(0, minsize)$;
$geneA \leftarrow gene[0, point] + partner[point, size]$;
$geneB \leftarrow partner[0, point] + gene[point, size]$;
create new individual using $geneA$;
create new individual using $geneB$;
return new individuals

Algorithm 2: The singlePoint method.

Where $size$ is the size of the genes of the individual that is calling it.

If the new individual can't find a solution based on the genes given, continue to generate points on top of the gene given. This will be common to all recombination and mutation operators.

Bit-Flip Mutation

The following pseudo-code is an abstraction of what happens to the individual during the bit-flip mutation:

Result: Mutated individual
$point \leftarrow$random[0, size];
$value \leftarrow$gene[random[0, size]];
gene[$point$]$\leftarrow value$;

Algorithm 3: The bitFlip method.

It is important to mention that the $value$ variable is taken from already mentioned points because, the real objective of the bitFlip method is to generate other alternatives by slightly changing the combination order.

Results

In the end, the program developed is capably of finding a "good enough" solution for all pancake sorting problems of all sizes, given enough time and computational resources.

Population analysis

Based on the information given previously, one can deduce that:

$$\Delta populationsize = offspring - decimated \quad (1)$$

Where $offspring$ is the amount of offspring that the selected individuals generate and $decimated$ is the amount of individuals that are decimated at the end of the iteration. Upon further analysis, we can deduce each of the variables.

$$offspring = 2 \times \sum_{i=1}^{selected} i \quad (2)$$

$$decimated = selected \quad (3)$$

As you can see, both variables are strictly dependant on the $selected$ variable, which is the amount of individuals that are selected from the random selection operator. Concluding the analysis, we can define $selected$ as

$$selected = \frac{initialsize}{generation + 1} \tag{4}$$

Where the final equation is

$$\Delta populationsize = 2(\sum_{i=1}^{\frac{initialsize}{generation+1}} i) - \frac{initialsize}{generation + 1} \tag{5}$$

Where $initialsize$ is the initial population size before any iterations, and $generation$ is the current generation the iteration is in.

Figure 2: $selected$ function behaviour, where X is the current generation and Y is the selected amount of individuals for.

As it is possible to see, the final equation depicts an equation that, by calculating it's limit, it converges to 0, thereby generating a solution whose computational complexity is polynomial.

Recombination vs. mutation

Due to the low correlation between fitness and a solution, there needs to be a further emphasis on the mutation operator, since it's purpose is to enrich the gene pool of the population, hence the high threshold on the mutation operator.

As you can see from figure 2, the population grows rapidly on the first generations by using the recombination operator, however, after the population grow stabilizes, the mutation operator will be the primary operator, since it will alter the individuals genes in order to search for an ideal solution.

Comments

One possible future work, based on a statement made by the professor, was of a co-evolutionary algorithm where, besides the individual stated in this paper, there would be another type of individual whose function would be to produce harder problems to solve, and observe emergent interactions. Furthermore, more recombination and mutation operators could be tested for any performance enhancements, as the current program lacks an efficient way of searching for a solution in problems with larger sizes.

References

Berman, P.; Karpinski, M. H. S. (2002). "1.375-approximation algorithms for sorting by reversals". *Proc. 10th ESA*, 2461:200–10.

Berman, P.; Karpinski, M. H. S. (2015). "pancake flipping is hard". *Journal of Computer and System Sciences*, 81(8):1556–1574.

Bulteau, Laurent; Fertin, G. R. I. (2015). "pancake flipping is hard". *Journal of Computer and System Sciences*, 81(8):1556–1574.

Dweighter, H. (1975). "elementary problem e2569". *Amer. Math. Monthly*, 82(10):1009–1010.

Gargano, L.; Vaccaro, U. V. A. (1993). "fault tolerant routing in the star and pancake interconnection networks". *Information Processing Letters*, 45(6):315–320.

KORF, R. E. (1985). "depth-first iterative-deepening: An optimal admissible tree search*".

Panchal, Gaurang; Panchal, D. (2015). "solving np hard problems using genetic algorithm". *International Journal of Computer Science and Information Technologies*, 6(2):1824–1827.

Yao, L. (2005). *IEEE TRANSACTIONS ON POWER SYSTEMS*, 20(3).