Ant Colony Optimization fine-tuned to solving Google Hashcode 2020 elimination round.

Anatol Kaczmarek 156038 group 2 and Wiktor Kamzela 156069 group 1

Combinatorial Optimization

The problem is similar to other problems – knapsack problem and traveling salesman problem (specifically time dependent versions of these problems). In both of which Ant Colony Optimization (ACO) is one of the best-known approaches.

ACO is a metaheuristic algorithm inspired by an ant colony trying to find the shortest path to the food. The essential element of this algorithm is the population of agents (ants). Each agent is looking for the most optimal path. When choosing the next node each ant considers distance to that node and number of pheromones left on the edge leading to this node by other ants. Once an ant finds a solution it places pheromones along the path proportionally to length of this path (or quality of the result). In the beginning ants are going randomly but at some point pheromones start guiding agents towards more optimal solutions.

One of the advantages of Ant Colony Optimization is that it approximates solutions to hard optimization problems very well. It seems to fit perfectly with this kind of problem. Moreover, it is fairly easy to implement. In contrast to many other techniques, it does not utilize searching the neighborhood space as the primary way of finding a solution. In this problem, unlike a standard TSP, swapping two, even consecutive libraries could have a significant impact on overall solution fitness, because it changes which books can be scanned in other libraries. Even though we ultimately decided to use mutation in our program it serves purpose to escape local optimum but is not only way we look for solution.

Our implementation:

In our implementation we decided to focus on finding the best sequence of libraries to sign up while we simply pick the most valuable and not already scanned books from each library. Therefore, we treat libraries like we would treat cities in TSP problem. Instead of normal weights our “distance” is value of books we can get by picking some library.

In normal ACO to pick next city we calculate the probability of choosing this city expressed by the equation: $\frac{1}{distanc\,e^{\alpha}} \cdot numberOfPheromone\,s^{\beta}$. In our solution we do not have to take the inverse of “distance”, because we want to maximize it instead of minimizing like in normal. So, our equation looks like this: $valu\,e^{\alpha} \cdot numberOfPheromone\,s^{\beta}$.

In our implementation of ACO number of pheromones added to each nodes are expressed by equation: $r_{xy} \leftarrow (1-p)\,r_{xy} + \sum_{k}^{m} \Delta r_{xy}^{k}$. To calculate delta, we modified equation taken from [1]:

$$\Delta r = \frac{1}{1 + \frac{best - curr}{min\,(curr\,,\,best)}}$$

First we tried to distribute pheromones in the edge between libraries. But then we did not take into account when the library was picked (so how many books were already scanned). Because of that we changed our strategy to distribute pheromones between the number of libraries already signed in and library. We also introduced other pheromones distributed directly to books.

We also introduced GreedyAnt, a subclass of ants which is used before the regular ants go in. When choosing the next library in uses a predefined greedy strategy. Its goal is to secure us a pretty good score regardless of the complexity of the test case.

Additionaly we decided to modify the usual ACO by using mutations. We mutate 10 bests and every 5 iterations by swapping random libraries in the solution. We accept only better solutions and every 10 iterations we mutate solutions more locally by swapping only consequtive libraries.

In our implementation we chose OOP approach, so our program is divided into multiple classes:

Book → has only two attributes, number, and value. We also overloaded the comparison operator, to sort books by their value.

Library → it implements all needed attributes and methods of libraries. i.e.:

- vector of books
- calculate approximate value in given time, we do not consider already scanned books.

We improved the proces of calculating aprrox value by counting prefix sums only at the beginning. Only books pheremones are calculated every iteration, but only once for each library.

Ant → abstract class, defining general template for GreedyAnt and PheromoneAnt which inherit from Ant. But we still implemented some functions which are used by both greedy and pheromone ant I.e:

- mutate → implements our mutation process.
- totalValue → return total value of books that can be scanned from libraries signed in by an ant.

GreedyAnt:

- nextLibrary → pick library with highest approx value / sign up time ratio.

PheromoneAnt

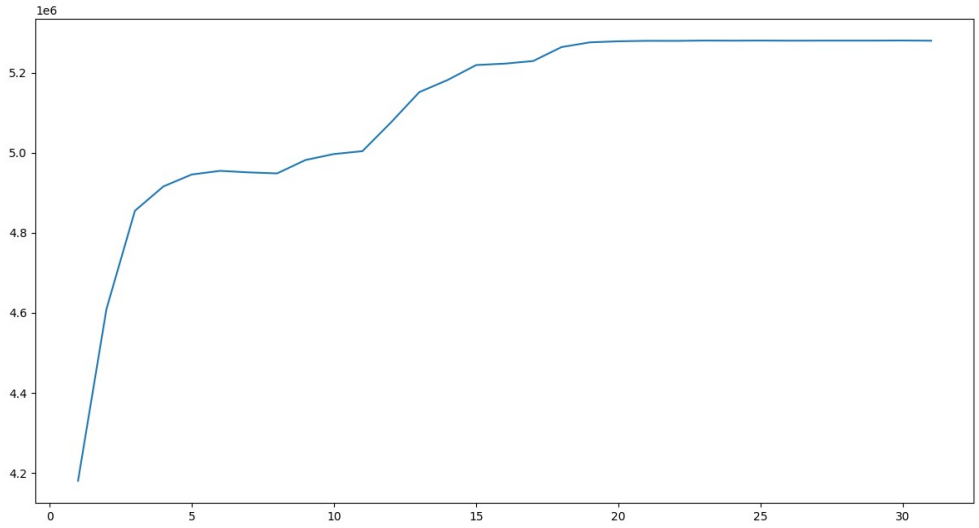
- nextLibrary → pick library with probability
- pickLibrary → get vector of probabilities as prefix sums and return one library with given probability.

ACO → Main class of our problem

- iteration → Main method of ACO, it carries out whole process of taking next libraries, calculating pheromones, and calculating best values.
- calculatePheromones → function that calls function to calculate pheremones deltas and then calculate pheromones.

Ultimately we achieved set goals, because not only we managed to score over 26 million which is a good result when compared to competitors, but also average result of whole population kept rising which proves that our ants are attracted by pheromones therefore our program correctly implements ACO.

The plot below shows how the average quality of the found solution rises throughout the whole simulation.



References:

[1] https://repozytorium.biblos.pk.edu.pl/redo/resources/30706/file/suwFiles/SchiffK_AntColony.pdf

[2] https://www.diva-portal.org/smash/get/diva2:1214402/FULLTEXT01.pdf

[3] https://repositorio-aberto.up.pt/bitstream/10216/123607/2/363694.pdf

[4] https://arxiv.org/pdf/1909.10427.pdf

[5] https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms