project

December 20, 2020

1 Introduction

1.1 Authors

Magdalena Kobusińska (145463) and Jacek Karolczak (145446)

1.2 Abstract

The following document is a code review for the solution of NP-Hard problem, described by Google in Google Hashcode 2020 problem statement. This problem has been assigned as a semester project for Combinatorial Optimization course by M.Sc. Jarosław Synak. The given problem is NP-Hard, we decided to implement the Genetic Algorithm. We also decided to update GA principals to meet our needs. The key difficulty occurred finding good heuristic function.

2 Preparation

2.1 Modules

To develop our solution we used following Python built-in modules:

- `time` - to limit time in which our program will be running;
- `random` - to choose random index in creating individual; to choose individuals which will do crossover, according to mating pool;
- `statistics` - to compute heuristic of library, using variance.

[1]:
```
from time import time
import random
import statistics
```

2.2 Time measurements

One of the key requirements of the project is to return a result within 300 seconds from the start of code running. To meet this criterion we decided to store the duration of each epoch (actually time stamp of epoch completion time) in a list.

[2]:
```
epochs_duration = []
epochs_duration.append(time())
```

2.3 Library heuristic

To compute heuristic value for library we use following formula:

$$heuristicValue = \frac{\sum_{i=0}^{size} bookScore_i}{max(0.01, var(bookScores)) * signUpTime}$$

We want to maximize the sum of scores of books in the given library. That is why we put this value in the numerator. Another factor which we decided to take into account was time necessary to sign up the library - we want to minimize this factor, so we put it in the denominator. The last factor, which changed the results of our program was the variance of books' scores in the library. In some test cases, for some libraries variance was equal to zero, so we added substitution with 0.01 if variance equals 0.

[3]:
```
def library_heuristic(library, books_scores):
    if len(library[1]) == 1:
        return library[1][0]
    books = library[1]
    scores = [books_scores[x] for x in books]
    total_score = sum(scores)
    variance = statistics.variance(scores)
    return total_score / (max(0.01, variance) * library[0][1])
```

2.4 Evaluation function

In many places, it is useful or even necessary to know how good is the particular solution. Thus we decided to implement a function, which will return fitness value for the solution - fitness value is a precise value which is described in Google Hashcode problem description.

[4]:
```
def evaluate_solution(solution, books_scores):
    books_scanned = set()
    points = 0
    number_of_libraries = solution[0]
    for i in range(1, number_of_libraries+1):
        number_of_books = solution[i][1]
        for j in range(number_of_books):
            book_to_add = solution[i][2][j]
            if book_to_add not in books_scanned:
                books_scanned.add(book_to_add)
    solution_score = 0
    for i in books_scanned:
        solution_score += books_scores[i]

    return solution_score
```

2.5 Get complete solution from list of solutions

In many places, like mutation and crossover operators, our program operates on a set containing libraries chosen to sign up in particular solution, which is ordered according to order in the sign-up queue. Following function transform such a list into complete and valid solution. For each day, for each library, as long as there is time to send the book, is chosen a book (or books) with the highest score, which is available in the library and hasn't been sent by another library.

[5]:
```
def get_complete_solution(solution_indices, libraries, days_number,␣
↪books_scores):
    time_pointer = days_number
    used_books = set()
    complete_solution = [len(solution_indices)]
    for i in solution_indices:
        time_pointer -= libraries[i][0][1]
        books_per_day = libraries[i][0][2]
        available_books = list(libraries[i][1])
        particular_solution = [i, 0, []]
        available_books.sort(key = lambda x: books_scores[x], reverse = True)
        for x in available_books:
            if len(particular_solution[2]) > time_pointer * books_per_day:
                break
            if x not in used_books:
                particular_solution[2].append(x)
                particular_solution[1] += 1
                used_books.add(x)
        particular_solution[2] = tuple(particular_solution[2])
        complete_solution.append(tuple(particular_solution))
    return tuple(complete_solution)
```

3 Genetic Operators

3.1 Selection

3.1.1 Initial population

Each solution is created by adding arbitrary, not used the library to the solution as long as the time necessary to sign up all library in solution doesn't exceed available time. Choosing arbitrary library is determined by its heuristic values - the higher the value the more preferable library.

[6]:
```
def get_random_solution(libraries_number, days_number, libraries,␣
↪libraries_heuristics, books_scores):
    libraries_to_use = list(range(libraries_number))
    libraries_to_use_heuristics = list(libraries_heuristics)
    solution_indices = []
    time_pointer = 0
    while True:
        if not len(libraries_to_use):
            return get_complete_solution(solution_indices, libraries,␣
↪days_number, books_scores)
        random_index = random.choices(list(range(0, len(libraries_to_use))),␣
↪weights = libraries_to_use_heuristics, k = 1)[0]
```

```
        del libraries_to_use[random_index]
        del libraries_to_use_heuristics[random_index]
        if time_pointer + libraries[random_index][0][1] > days_number:
            break
        solution_indices.append(random_index)
        time_pointer += libraries[random_index][0][1]
    return get_complete_solution(solution_indices, libraries, days_number,␣
 ↪books_scores)
```

The evolutionary process begins with initialization, wherein an initial population of candidate solutions is generated. We decided to create the initial population as long as it's size is equal to set size or the half of available time runs out. The last element of the initial population is a greedy solution (see below).

```
[7]: def get_initial_population(population_size, libraries_number, days_number,␣
 ↪libraries, libraries_heuristics, books, timestamp, max_duration):
    population = []
    for i in range(population_size):
        population.append(get_random_solution(libraries_number, days_number,␣
 ↪libraries, libraries_heuristics, books))
        if i % 2 and time() - timestamp >= max_duration / 2:
            return (i, population)
    return (population_size, population)
```

3.1.2 Get greedy-like solution

One of project requirement is to return the result which is not worse than the greedy solution. To meet this constraint we decided, that program will add the greedy-like solution to the initial population, to be sure, that our final solution won't be worse than the greedy solution.

```
[8]: def get_greedy_solution(libraries_number, days_number, libraries,␣
 ↪libraries_heuristics, books_scores):
    libraries_to_use = list(range(libraries_number))
    libraries_to_use.sort(key = lambda x: library_heuristic(libraries[x],␣
 ↪books_scores), reverse = True)
    solution_indices = []
    time_pointer = 0
    for i in range(libraries_number):
        if time_pointer + libraries[libraries_to_use[i]][0][1] >= days_number:
            break
        solution_indices.append(libraries_to_use[i])
        time_pointer += libraries[libraries_to_use[i]][0][1]
    return get_complete_solution(solution_indices, libraries, days_number,␣
 ↪books_scores)
```

3.1.3 Mutation

The algorithm swap nucleotides in three pairs. Thus, it's prone to omit optima, but after our first experiments we figured out, that on our instances that gives better results if our time is not limited.

[9]:
```python
def mutate(solution, libraries, days_number, books_scores):
    solution_indices = [solution[i][0] for i in range(1, len(solution))]
    for _ in range(3):
        nucleotide_a = random.randint(0, len(solution_indices) - 1)
        nucleotide_b = random.randint(0, len(solution_indices) - 1)
        solution_indices[nucleotide_a], solution_indices[nucleotide_b] =␣
↪solution_indices[nucleotide_b], solution_indices[nucleotide_a]
    return get_complete_solution(solution_indices, libraries, days_number,␣
↪books_scores)
```

Our Genetic Algorithm mutate each solution, we do not use any probability to determine chance of mutation occurence like in a typicial GP.

[10]:
```python
def do_mutations(population, libraries, days_number, books_scores):
    mutated_solutions = []
    for solution in population:
        mutated_solutions.append(mutate(solution, libraries, days_number,␣
↪books_scores))
    return mutated_solutions
```

3.2 Crossover

Crossover is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next. Two strings are picked from the mating pool at random to crossover to produce superior offspring. Our program uses a single-point crossover. It means that the first child consists of the first part of first parent and second from the second one and the second child consists of the first part of the second parent and second from the first one. The algorithm takes care also for validness of the solution.

[11]:
```python
def get_children(parent_a, parent_b, libraries, days_number, books_scores):
    parent_a_indices = [parent_a[i][0] for i in range(1, len(parent_a))]
    parent_b_indices = [parent_b[i][0] for i in range(1, len(parent_b))]
    split_point = random.randint(0, min(len(parent_a_indices),␣
↪len(parent_b_indices)) - 1)
    child_a_indices = parent_a_indices[:split_point] + [x for x in␣
↪parent_b_indices[split_point:] if x not in parent_a_indices[:split_point]]
    child_b_indices = parent_b_indices[:split_point] + [x for x in␣
↪parent_a_indices[split_point:] if x not in parent_b_indices[:split_point]]
    for parent in (parent_a_indices, parent_b_indices):
        time_pointer = 0
        for i in range(len(parent)):
            if time_pointer >= days_number:
                parent_a_indices = parent[:i]
                break
            time_pointer += libraries[parent[i]][1]
        time_pointer = 0
    child_a = get_complete_solution(child_a_indices, libraries, days_number,␣
↪books_scores)
    child_b = get_complete_solution(child_b_indices, libraries, days_number,␣
↪books_scores)
    return (child_a, child_b)
```

Our algorithm creates new individuals in such a way, that offspring size equals population size. The mating pool consists of values of the population.

[12]:
```
def get_offspring(population, libraries, days_number, books_scores):
    offspring = []
    mating_pool = [evaluate_solution(population[i], books_scores) for i in␣
↪range(len(population))]
    for i in range(len(population), 2):
        offspring.append(get_children(random.choices(population, weights =␣
↪mating_pool, k = 2)), libraries, days_number, books_scores)
    return tuple(offspring)
```

4 Main

4.1 Parameterization

Time is specified by a project requirements.

[13]:
```
max_duration = 300
```

The population size and operators should not be kept very large as it can cause a GA to slow down, while a smaller population might not be enough for a good mating pool. Therefore, an optimal population size needs to be decided by trial and error. We did many experiments to find operators which will do best in finding the best solution. This same had been done for the size of the population.

[40]:
```
population_size = 80
```

4.2 Read data

The following part of a code is responsible for decapsulation input data for useful and understable for the program form.

[ ]:
```
books_number, libraries_number, days_number = [int(x) for x in input().split()]
books_scores = tuple([int(x) for x in input().split()])
libraries = [None] * libraries_number
for i in range(libraries_number):
    libraries[i] = (tuple([int(x) for x in input().split()]), tuple([int(x) for␣
↪x in input().split()]))
libraries = tuple(libraries)
```

4.3 Getting the initial population

The following snippet is responsible for creating a tuple, which stores heuristic values for libraries. Such values are necessary to create an initial population. The mechanism is widely described above function which is in charge of creating the initial population.

```
[ ]: libraries_heuristics = tuple([library_heuristic(library, books_scores) for␣
      ↪library in libraries])
```

Here program is obtaining initial population. Process of generating population is limited to half of the time given to execute the program, to prevent exceeding time with no solution or with quite a poor solution. Thus, the final population size may vary from given in parameters declaration.

```
[ ]: population_size, population = get_initial_population(population_size - 1,␣
      ↪libraries_number, days_number, libraries, libraries_heuristics,␣
      ↪books_scores, epochs_duration[0], max_duration)
```

One of the requirements of the project is to return solution which is not worse than the greedy solution. To meet this requirement program add the greedy solution to the initial population. We are aware that adding greedy solution may make program prone to stuck in local optima, but we want to be sure, that in a worst-case out solution will be at least as good as a greedy solution.

```
[ ]: population_size += 1
     population.append(get_greedy_solution(libraries_number, days_number, libraries,␣
      ↪libraries_heuristics, books_scores))
```

4.4 Main loop - epochs

Epochs are executed as long as time remaining for program execution is greater or equal to doubled last epoch's duration + 5 seconds. This guarantee, that program will not only finish searching for the best solution but also print the best solution in a given time. The program repeat the following procedure:

```
In each epoch:
    extend population with mutated individuals
    extend population with offspring
    choose population_size best indicidual s to create new population
```

```
[ ]: epochs_duration.append(time())
     while epochs_duration[0] + max_duration - 5 >= time() + (epochs_duration[-1] -␣
      ↪epochs_duration[-2]) * 2:
         mutations = do_mutations(population, libraries, days_number, books_scores)
         offspring = get_offspring(population, days_number, libraries, books_scores)
         population.extend(mutations)
```

```
    population.extend(offspring)
    population.sort(key = lambda x: evaluate_solution(x, books_scores), reverse␣
↪= True)
    population = population[:population_size]
    epochs_duration.append(time())
```

At the end, when program is running out of time, best solution as first individual of population. Thus last step to obtain solution is to print it out. Our program do this in a format specified by project requirements.

[ ]:
```
print(population[0][0])

for i in range(1, len(population[0])):
    print(population[0][i][0], population[0][i][1], end = " ")
    print(" ".join([str(x) for x in population[0][i][2]]))
```

5 Sum of all instances

After applying everything that was explained above, we run the program for six different data sets which were given. The final answer is the sum of the best scores for the individual data sets. The results are showed below.

- `a_example.txt` solution is: 21
- `b_read_on.txt` solution is: 5831900
- `c_incunabula.txt` solution is: 5414171
- `d_tough_choices.txt` solution is: 4815200
- `e_so_many_books.txt` solution is: 4610173
- `f_libraries_of_the_world.txt` solution is: 4116926

So it gives us the total solution of 24788391 points.