Scheduling the categories for orienteering competitions

David Rajnoha

June 2021

1 Introduction

The goal of the project was to develop an application, that will schedule the categories and help to prepare the start lists for an orienteering competition.

1.1 Problem description

In an orienteering race, the competitors in a category usually start one after other in a fixed interval (typically ranging from 1 to 5 minutes). The competitors in one category have to start in subsequent interval, meaning a category forms a block of start times rather than the competitors being randomly scattered through the length of the start lists. Due to the organizational and logistical needs, it is desirable to reduce the number of competitors starting in the same minute. This means that we have to schedule the blocks of competitors in a way, that the number of concurrent starters would not surpass a chosen value. Due to the nature of the orienteering and the rules, there are further restrictions on the order of categories. The categories with the same first control can not start in the same minute, and the categories with the same route can not even overlap. These restrictions have to be fulfilled while creating the schedule. Further we can work with other constraints, that don't have to be necessarily fulfilled, but the schedule, which fulfills them can be considered better. To these constraints belong requests for two categories to start at the same time or requests for a category to start at a specific time. These constraints usually reflect the wishes of the competitors. Further we will call them soft constraints. The goal of the scheduling is to find a schedule with a given limit of competitors starting at the same time, that will meet all hard constraints and find an optimum between the fulfillment of the soft constraints and the total length of the schedule.

1.2 Flexible Job Shop Problem

The scheduling problem can be seen as a variant of the flexible job shop scheduling problem (FJSP). In the FJSP, we have to schedule a set of n jobs $J = \{J_1, J_2, ..., J_n\}$ to m machines $M = \{M_1, M_2, ..., M_m\}$. Each job is composed

from a number operations, where a given permutation of the operations imposes a restriction on how these operation have to be ordered. Each of the operation can be processed by any of the machine from M. [3]
If we consider our problem with no restrictions, it is a FJSP, where we will consider the categories as the jobs (each job consist of only one category) and slots as the machines. As a slot we mean a designated place in a starting minute in a given interval. The number of slots is therefor equal to the starting interval times the limit of runners in the same minute.
Further we will consider the constraints, that will limit the jobs being able to run in the slots belonging to the same minute or limit the jobs to run at the same time. The soft constraints will not serve as a hard limit but along with time will be a value we would aim to minimize.
The FJSP is a NP hard problem, meaning that it is not possible to efficiently find the optimal solution. Therefore it is necessary to use some form of heuristic, that will help us find a solution that is good enough. For this project, I have decided to use genetic algorithms. This decision was mainly influenced by my desire to learn more about evolutionary computing and try it in practice. That means there could be other approaches, which would solve the problem more efficiently.

1.3 Genetic algorithms

The genetic algorithm is an approach of solving the NP hard problem inspired from the principle of evolution. The possible solutions are encoded to a genome represented as a string or a list of characters. Initially are these genome created at random. In each round of the algorithm, a new generation of the individuals is created by combining the genomes of the individuals from the previous generation. Some mutations may be applied to improve the diversity and explore different solutions. Then are the created genomes evaluated using a specific evaluation function and based on this score are some kept and some discarded. After the evaluated score of the individuals converges or after a given number of generations is the best individual considered the solution to the problem. [2]

2 Existing solutions

Jan Novák is in his bachelor thesis "Algoritmus pro generování rozvrhu orientačního běhu" [4] also solving the problem of generating the schedule for the orienteering competitions. The difference from this project is the usage of a different algorithm (simulated annealing) and also the fact, the only the hard constraints are considered and the schedule can be optimized just on the length of the schedule.

3 Implementation

The application implements a genetic algorithm solving the described problem. In the following paragraphs are described the core classes and functions taking part in the implementation. The application also contains simple interface and a layer for parsing data. These layers are not described here, their basic description is provided in the docstrings directly in the code.

3.1 Representation of real world objects

The following classes are used to represent the real world objects.

Category

The `Category` class carries the information about the category. The most important are the number of competitors and the constraints for the category, data that are bring the necessary information for the schedule creation. The category is the equivalent of a job in the FJSP.

Slot

The `Slot` class is the equivalent of a machine in the FJSP. It contains the Category objects that start in the given slot. It provides the `iter_next_categories` and the `next_category` methods, allowing the iteration through the categories based on their order. This functionality is used in the evaluation of a particular solution.

Race

The Race class contains all information about the race including the empty `Slot` and `Category` objects associated with it.

3.2 Genetic Algorithm Logic

The LEAP [1] library is used for the implementation of the genetic algorithm. The `Decoder` and `Problem` classes from the library are extended as `RaceDecoder` and `RaceProblem` adding the functionality specific for our problem. With these two classes is also tied the class `RacePhenom` and also representation of the genome.

RacePhenom

The `RacePhenom` class serves as a representation of a particular solution of our problem. It contains Slots with Categories distributed among them. In addition it provides methods that return information useful during the evaluation, such as the `total_length` and the `efficiency` methods.
The later one is the division of the optimal length and the total length of the

schedule. The optimal length is the length of q start list without any restrictions - the number of the competitors divided by the number of concurrent starters. The efficiency is therefore 1 at best and decreasing with increasing length of the schedule.

Genome

In genetic algorithms, genome carries the information about the solution in a form that is suitable to mutate and combine the information with other individual. Here is the genome represented as a list containing the categories and numerical separators that encode the slots to which the categories belong. The separators are not identical due to the implementation of the crossover.

RaceDecoder

The `RaceDecoder` class extends the `Decoder` class from the LEAP library. Its main function is to take the genome of an individual and decode it into a `RacePhenom`.

RaceProblem

The `RaceProblem` class extends the `Problem` class from the LEAP library. Its main feature is the `evaluate` method, which takes a `RacePhenom` and returns its fitness- a number, which represents how good is the phenom. It also defines the ordering of the evaluated score, in the case of this project lower score means better evaluation.

evaluate

The evaluating method is one of the key components of the program. It computes the score which determines the fitness of the RacePhenom. It does so in the following way.
It computes the `time_score` as `(1 - efficiency) * 100` taking values from 0 to 100. Then it computes the `time_req_score` and the `same_start_req_score` as the number of unfulfilled same time requests and same start requests respectively.
These scores are then added using the formula `(time_score * 1.5) + time_req_score + same_start_req_score` and form the `soft_constraints_score`. The coefficients were established by creating schedules that were evaluated as better or worse and tuning the coefficients, so the result of the evaluating function would correspond to the human evaluation.
The `hard_constraints_score` is computed as the number of hard constraints that are not fulfilled. That means the number of categories starting in the same minute with the same first control and the number of categories with the same route starting concurrently.
The evaluating function than returns `hard_constraints_score * 100 + soft_constraints_score` as the evaluation of the RacePhenom object

Pipeline

The core of the functionality is in the `ga_pipeline` function. The function contains a while cycle, where the `toolz.pipe` together with the functionality from the LEAP library forms a pipeline, that produces the offsprings of the population. The pipeline takes the parent generation population, clones the individuals so the parents are not changed, performs the mutations, performs the crossover, computes the fitness score and collects the child generation. Then the best individuals from the parent and child generation are selected and these are passed to the next round of the cycle.

Mutation

The mutation is conducted by randomly selecting two position in the list representing the genome of the individual and flipping the values on these positions. The number of mutations is determined by the `mutations_probability`. When the fitness of the best individual is equal to the average fitness, the mutation rate is increased to two, meaning two instead of one mutations are performed. This leads to an increase in exploring further solutions and getting out of the local optimum. The particular value was determined experimentally, higher mutation rates weren't beneficial and start of the higher mutation rates before reaching the local minimum lead to slower overall convergence.

Crossover

The genome representation of our problem is a permutation by its nature. If a genome is valid, it must contain all categories and all separators exactly once. Therefor, we have to use a crossover that will preserve the property of permutation.
For our project, we have chosen the cycle crossover. The cycles are identified in a way, where we start at the first position not in any previous cycle in the first genome. We look at the element in the same position in the opposite genome. We find that element in the first genome, we look at the element in the second genome. We continue in this way until a cycle is formed. Then we continue with the next unused element. We can then alter the genomes by flipping half of the cycles. [2]

4 Installation and execution

The project is available at `https://github.com/DavidRajnoha/pa026`. To run the application, clone the repository and install the required packages using

```
pipenv sync
```

Then run the following command. (The value of the options here will result in a sample run).

```
python main.py schedule_categories
   --oris_id=5662
   --course_definition=resources/JML2020TBM.Courses.Edited.txt
   --same_start_req=resources/JML2020TBM_same_start_requests
   --specific_time_req=resources/JML2020TBM_time_requests
   --concurrent_slots=3
   --ignore_categories=HDR,T,P
   --interval=2
```

The description of the options is following:

oris_id

Id of the race in the ORIS - an information system of the Czech orienteering. It is used for downloading the number of registered competitors for the given race.

course_definition

Definition of the courses for the catagories in the following format.

```
ID1, ID2, ... IDn S1-C1-C2-C3-...-CN
```

First there are the IDs of the categories to which the course belongs to separated by comma and space. Then there is a space and the controls on the course separated by dashes. The first control is the start, and is therefor ignored. The course definitions are needed to specify the hard constraints.

same_start_req

The requests for categories that should start at the same time. Each line has the following format.

```
ID1 ID2
```

Meaning that on each line, there are the IDs of the categories that should start at the same time. This requirements are used for one of the soft constraints.

specific_time_req

The requests for the categories to start during a specific time. Each line has the following format.

```
ID STR1 STR2 STR3 ... STRN
```

That is the id of the category followd by a number of times that the category is expected to start during.

concurrent_slots

The maximum number of runners to start in the same minute.

interval

The interval in which the runners from the same category start.

generations

The number of generations after which the algorithm terminates.
Defaults to 200

initial_population

The number of individuals that form the initial population.
Defaults to 20

5 Run demonstration

When we run the example command

```
python main.py schedule_categories
--oris_id=5662
--course_definition=resources/JML2020TBM.Courses.Edited.txt
--same_start_req=resources/JML2020TBM_same_start_requests
--specific_time_req=resources/JML2020TBM_time_requests
--concurrent_slots=3
--ignore_categories=HDR,T,P
--interval=2
```

the algorithm start computing and gradually prints out the average fitness of the population and the fitness of the best individual.

```
best: 47, average: 49.8
best: 47, average: 49.8
best: 47, average: 49.8
best: 47, average: 49.8
best: 47, average: 49.75
best: 47, average: 49.7
best: 47, average: 49.55
best: 47, average: 49.55
best: 47, average: 49.55
best: 43, average: 49.2
best: 43, average: 49.0
best: 43, average: 48.55
best: 43, average: 47.95
best: 43, average: 46.85
```

```
best: 43, average: 46.2
best: 43, average: 45.45
best: 43, average: 44.85
best: 42, average: 44.25
best: 42, average: 43.5
best: 37, average: 42.3
best: 37, average: 41.95
best: 37, average: 41.3
best: 37, average: 41.0
best: 37, average: 40.55
best: 37, average: 39.85
best: 37, average: 38.85
best: 37, average: 37.7
```

After the run, the program saves the results in a json format and also in a graphical form.

```
{
"D16": {"entries": 8, "first start": 0, "offset": 0},
"H55": {"entries": 9, "first start": 16, "offset": 0},
"H16": {"entries": 10, "first start": 34, "offset": 0},
"H65": {"entries": 5, "first start": 54, "offset": 0},
"D21D": {"entries": 5, "first start": 64, "offset": 0},
"D14": {"entries": 20, "first start": 74, "offset": 0},
"H21C": {"entries": 29, "first start": 0, "offset": 0},
"H20": {"entries": 0, "first start": 58, "offset": 0},
"D35": {"entries": 24, "first start": 58, "offset": 0},
"D65": {"entries": 4, "first start": 106, "offset": 0},
"H12": {"entries": 22, "first start": 0, "offset": 0},
"H10": {"entries": 15, "first start": 44, "offset": 0},
"D55": {"entries": 6, "first start": 74, "offset": 0},
"H18": {"entries": 2, "first start": 86, "offset": 0},
"D20": {"entries": 1, "first start": 90, "offset": 0},
"D18": {"entries": 1, "first start": 92, "offset": 0},
"H45": {"entries": 24, "first start": 1, "offset": 1},
"D12": {"entries": 23, "first start": 49, "offset": 1},
"D10N": {"entries": 11, "first start": 1, "offset": 1},
"D10": {"entries": 16, "first start": 23, "offset": 1},
"D21C": {"entries": 17, "first start": 55, "offset": 1},
"D45": {"entries": 12, "first start": 89, "offset": 1},
"H21D": {"entries": 3, "first start": 1, "offset": 1},
"H35": {"entries": 28, "first start": 7, "offset": 1},
"H10N": {"entries": 6, "first start": 63, "offset": 1},
"H14": {"entries": 20, "first start": 75, "offset": 1}}
```

```
D16  ###  D16  ###  D16  ###  D16  ###  D16  ###  D16  ###  D16  ###  ...
H21C  ### H21C  ### H21C  ### H21C  ### H21C  ### H21C  ### H21C  ### ...
```

```
 H12  ###  H12  ###  H12  ###  H12  ###  H12  ###  H12  ###  H12  ### ...
 ###  H45  ###  H45  ###  H45  ###  H45  ###  H45  ###  H45  ###  H45 ...
 ### D10N  ### D10N  ### D10N  ### D10N  ### D10N  ### D10N  ### D10N ...
 ### H21D  ### H21D  ### H21D  ###  H35  ###  H35  ###  H35  ###  H35 ...
```

6 Testing and evaluation

The testing of the application can be divided into three parts - finding the best parameters for the algorithm, testing the total length of the schedule compared to the length of the schedule generated manually, and testing whether the application is able to create schedules with soft constraints fulfilled while not overly prolonging the total length of the schedule.

6.1 Testing against real data

The testing is conducted against the data from the real races. The program contains support for querying these data and is able to compute the efficiency of the manually created schedules. Based on the queried data (interval of the race, maximum number of runners in the same minute) is then generated a schedule. The efficiency of the generated schedule is compared with the efficiency of the manually created one and the difference is used in further evaluations.

6.2 Finding the best parameters of the algorithm

The genetic algorithm takes two parameters - the number of generations and the size of the population. We have run the algorithm with a combination of values 5 times on a sample of 10 real races and measured the efficiency difference. The results can be seen in the following table.

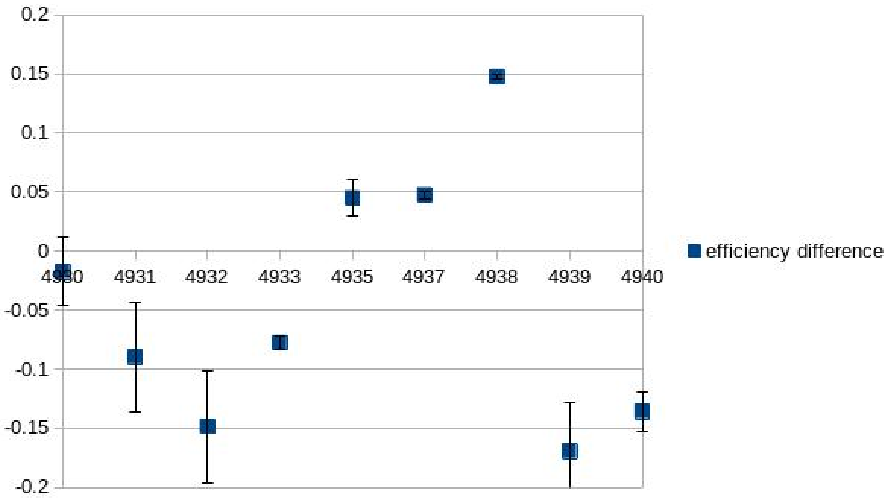
generations\population	20	40	80	100
200	0.11	0.0761	0.0589	0.0443
400	0.0597	0.0304	0.0093	0.0138
600	0.0361	0.0222	0.0086	-0.0011

The results show that the combination of the population size of 80 with 400 generations shows the most significant changes in the efficiency. The further increase of the parameters increases the run time with rather small reward in the efficiency.
To determine whether is more significant the size of the population or the number of generations, we have run another test, with results in the following table. We have again used the sample of 10 races and run the program 5 times. We have used multiple values of generations and initial population with the property that theirs product is constant (in our case 32 000).

generations / population	1600 / 20	800 / 40	400 / 80
mean	**0.0562**	**0.0478**	**0.0612**
std	0.0103	0.0105	0.0067

6.3 Testing the length of the schedule

After finding suitable values for the population size and the number of generation, we moved to testing the length of a generated schedule. We have created a schedule for 89 races, five times for each race and compared the efficiency with the efficiency of the manually created schedules. The average efficiency difference ended up 0.2 in favor of the manually created schedules. The standard deviation ended up 0.4. This was partly due to later discovered problems with the automatic querying the information about the races, where sometimes the values we got were not entirely accurate. That meant that the program sometimes tried to create schedules with more strict or more relaxed conditions then the real schedule.
We have decided to repeat the testing using a set of 9 selected races, where the information queried from the races were as expected. For each of the 9 races, we created the schedule 10 times. The average difference of the efficiencies ended up being -0.04 with the standard deviation of 0.03. That means, that the length of the generated schedules was about the same as the schedules created manually. In the following image are shown the efficiencies for the individual races.



6.4 Testing the soft constraints

Finally we have tested, whether is the program able to generate a schedule that fulfills the given soft requirements. Due to the fact, that collecting the requirements is a tedious manual job, the test has been conducted on a set of 5 races, with the average of 40 requirements for one race. For each race, we generated the schedule 5 times and observed the percentage of unfulfilled requests and the difference between the efficiency against the manual schedule. The results are in the following table.

ID	Efficiency difference	Total requirements	% req unfulfilled	stdev
4930	0.08	55	0.25	0.03
4931	-0.03	28	0.54	0.05
4932	-0.11	45	0.22	0.05
4933	-0.07	39	0.07	0.02
5662	-0.02	33	0.08	0.04
	-0.03	**40**	**0.21**	**0.04**

From it can be seen, that the schedules on average failed to fulfill one fifth of the requirements. The race with the id 4931 is an outlier with more then a half requirements unfulfilled. That was caused by the fact, that many of the requirements for the specific time were for time after the end of a sensibly efficient schedule.
In addition to the percentage of requirements fulfilled, two things are worth to be noted. First is the difference in efficiency, which is roughly the same as the difference for the generation of schedules without the requirements. The second is the average standard deviations of the percentage of unfulfilled requirements that ended up being 0.04. That is rather small value hinting at the fact, that the generated solutions are not getting stuck in a various different local optimums, but instead converging to a good solution.

7 Conclusion

The program is able to schedule the start lists for a orienteering competition using a genetic algorithm. The testing has shown, that the length of the generated schedules is similar to the length of the schedules created manually. When taking in account the soft constraints, the algorithm can fulfill most of them without significantly affecting the length of the schedule.
However, the testing of the soft constraints has also shown that there is still an area for improvement. The program is not well equipped to deal with requests to start at the end of the start list, as these have to be converted to a specific time request, and without the prior knowledge of the length of the start list can this conversion lead to requests, that can not be fulfilled.

References

[1] Mark A. Coletti, Eric O. Scott, and Jeffrey K. Bassett. “Library for Evolutionary Algorithms in Python (LEAP)”. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion.* GECCO ’20. Cancún, Mexico: Association for Computing Machinery, 2020, pp. 1571–

1579. ISBN: 9781450371278. DOI: 10.1145/3377929.3398147. URL: https://doi.org/10.1145/3377929.3398147.

[2] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. 2nd. Springer Publishing Company, Incorporated, 2015. ISBN: 3662448734.

[3] Xinyu Li et al. "Review on flexible job shop scheduling". In: *IET Collaborative Intelligent Manufacturing* 1 (June 2019). DOI: 10.1049/iet-cim.2018.0009.

[4] Jan Novák. "Algoritmus pro generování rozvrhu orientačního běhu". Masaryk University, 2015.