SPEAR optimisation algorithm for production processes now available as open source

The optimisation of production schedules for multiple objectives such as make-span, use of renewable energies, or total energy usage, is a difficult and increasingly relevant task. While a large body of work exists for production optimisation in general, those do not always fit the requirements and constraints found in the real world. The goal of the ITEA project SPEAR (https://spear-project.eu) was the development of a comprehensive platform for measuring, simulating, and optimising production processes.

To achieve multi-objective optimisation of production processes, project partner TU Berlin developed an approach based on a genetic algorithm that allows users to optimise for make-span, energy consumption or cost, or a mixture of these goals. Approaching the end of the SPEAR project, the code has been made available as open source and can now be found in the GitLab repository of DAI-Labor at https://gitlab.dai-labor.de/spear/spear-optimisation.

The modules' underlying model includes concepts such as products, tasks and task-modes, machines and cells, resources for modelling dependencies between tasks, and variable energy sources, allowing a wide range of production scenarios to be modelled and optimised.

The algorithms were evaluated using different scenarios provided by partners in the SPEAR project and showed promising results, being able to find non-trivial solutions, trading e.g. a slight increase in make-span for a significant gain in energy efficiency.

Domain model

The computational representation of the **Production System** is subdivided into two sides: Entities and Activities. Entities are *Production Cells*, *Machines*, and *Resources,* which can be used for modelling, e. g., raw materials, pressurised air, intermediate products, or the state of charge of a buffer battery. Activities are *Products*, which are comprised of *Tasks*, which can be executed in one of several *Task Modes*, e. g., on different machines, or using different energy profiles.

Multiple **Energy Sources** can be available for use, each with a *price* and *availability*, both variable over time, defining the price for one unit of energy and the units available per unit of time. Both are optional, and 'free' and 'infinite' per default, respectively. This way, the same model can be used for simple energy from the grid (fixed price, unlimited), locally installed renewable energies (free, limited varying availability) and complex capped energy tariffs (variable price, limit on usage per time step).

The optimisation is subject to different **Constraints**, both *general constraints* that always have to be fulfilled, e. g., that two tasks cannot be executed on the same machine at the same time, and that the needed energy must be provided for by the energy sources, and *user-defined constraints*

such that, e.g., task A may not be executed at the same time as task B. Further, resources (see above) can be used for modelling more complex dependencies between individual tasks and task modes producing or requiring those resources.

The above parts, together with some configuration parameters such as the optimisation weights, make up the **Request** being sent to the optimisation algorithm. The **Result** then consists of the actual *Schedule*, i.e. which tasks to execute when in what task mode and on which machine, as well as some *Statistics* about energy usage over time.

Optimisation algorithm

The optimisation has been implemented as a Genetic Algorithm (GA) using the Jenetics library (https://jenetics.io). The general working principle is as follows:

1. Create an initial population of candidate solutions
2. Randomly select, combine, and mutate individuals from that population
3. Use a fitness function to choose the best individuals for the next generation
4. Repeat until some stopping condition is satisfied

In a GA, the candidate solutions are represented in the form of a **genetic code**, comprising multiple chromosomes for different aspects, and thus mutated on a very low level, by changing or swapping individual genes in that code. In this approach, the total number of tasks to be executed are predefined by the products that have to be produced. For each task, the following attributes are encoded: The *task mode* and *machine* to use for that task, the *order* within all the tasks to be executed on that machine, and the *pause time* between this task and the previous one. Compared with the alternative of encoding the start time of each task, this has the advantage that by definition two tasks cannot 'clash' on the same machine, significantly increasing the chance of successful mutations (those that result in an improvement) at the cost of a slightly more complex decoding of the schedules. Analogously, another set of chromosomes is used for encoding the assignment of products to cells, product ordering and pause times.

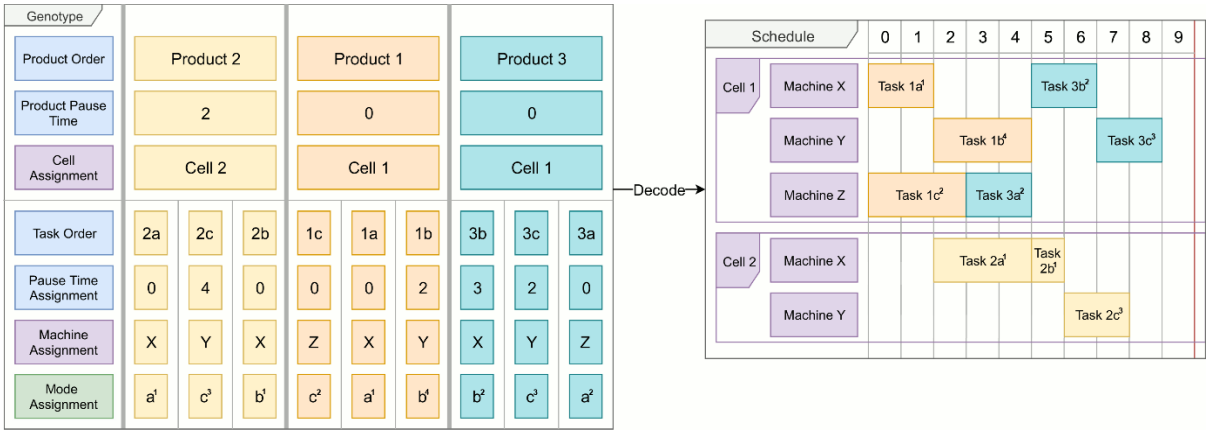


Illustration of how the schedules are represented in the genetic code.

Accordingly, the candidate schedules can be **mutated and recombined** in different ways:

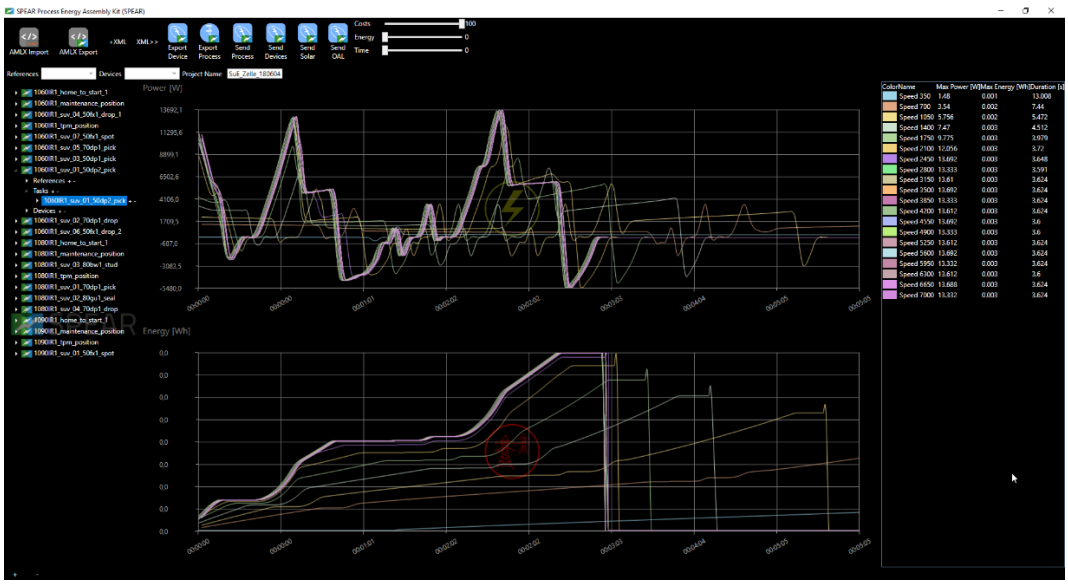
- Assigning a new value to a randomly chosen gene, e.g. a machine or task mode assignment
- Swapping two values in the task ordering chromosome, e.g. the order of two tasks on a machine
- Shifting a task (and all tasks that follow on the same machine) by altering a pause time gene
- Recombination as partially matched crossover and single-point crossover was tried, but did not result in a significant increase in quality

The **fitness function** used for assessing the quality of a candidate schedule is subdivided into two parts: the *hard score*, keeping track of the number of constraint violations making the schedule impossible, and the *soft score*, including the weighted user-defined optimisation criteria such as total make-span, energy usage, or energy costs. A schedule with hard constraint violations is not immediately discarded, but schedules with a lower hard score are always preferred to those with a lower soft score and higher hard score.

Interface

The algorithm is made available as a REST web service, providing different routes for configuration and the actual optimisation. The Requests and Results are encoded in JSON, allowing the optimisation to be invoked from and integrated into any other programming language. For local testing and debugging, the modules also include an insightful graphical user interface.

An instance of the REST Interface is currently running on the servers of DAI-Labor, TU Berlin, at https://spear.aot.tu-berlin.de:8082/, including a comprehensive description of the API. Alternatively, to host your own instance of the server, it can be built from the open-source code or used as a Docker image, found at https://hub.docker.com/r/dailab/spear-rest-interface.



RF::PEAK prototype showing the process description of the FFT demonstrator in the form of a tree structure (left), a graphical display of the power curve and corresponding energy consumption (center) and the most important parameters (right).