Production Planning using Evolutionary Computing

DS3500: Advanced Programming with Data

Purpose

- Leverage functional programming to build a factory machine production scheduler.
- Learn about evolutionary computing as a problem-solving technique
- Explore an important field of AI: Intelligent Decision-Support
- Tackle a computationally challenging real-world problem

Overview

Imagine a factory that produces a variety of products. Orders for exactly one type of product are received and the orders now need to be fulfilled in some sequence. The plant manager, Becky, will determine the production schedule. Unfortunately for Becky it is not necessarily optimal to simply fulfill customer orders in the order that they were received. Some customer orders, for example, may need to be expedited and given higher priority. Also, switching the machine configuration from producing one product to another requires downtime and should be avoided whenever possible. This is exactly the situation faced by process industries such as paper and steel: one must optimize for the constraints of the production machines to determine the best, most efficient, most optimal schedule.

In real-world scheduling problems, there are multiple competing objectives, and while we could probably come up with some weighting factors to score a given solution. A better approach is to generate many solution *alternatives* that reveal the tradeoffs. By understanding these tradeoffs, Becky, our plant-floor manager, can use her domain expertise to decide which of the solutions is best. Remember, our goal is not to find the best solution for Becky. Our goal, instead, is to find for her the best *tradeoffs.*

In this scenario, we have a set of orders that need to be scheduled on the factory machine. Each order has the following attributes:

- An order ID
- A priority (High or Low)
- A quantity (integer)
- A product code. (Every order is for exactly one type of product).

Objectives

Every schedule must be evaluated with respect to three competing objectives.

SETUPS**:** Minimize the number of setups. A setup occurs whenever we switch from producing one type of item to another. Do not count the first item in the production sequence as a setup. Setups can be minimized by trying to manufacture all orders of a given product together as much as possible, but this might delay high-priority orders or require that we produce some orders out of sequence. For example, if the fulfillment sequence was for products AABBBAAACCCCBDDDDDD this would count as FIVE setups: A→B, B→A, A→C, C→B, and B→D. We are counting a setup whenever the product we are producing changes.

LOWPRIORITY: Produce the high-priority orders as soon as possible. The quantifiable goal is to minimize the total quantity (not number) of low priority orders produced before the LAST high-priority order is fulfilled. Producing all the high-priority orders first will minimize this objective to zero but at the expense of setups and other order delays. For example, if three low priority orders are produced before the last high-priority order is completed and their quantities are 100, 200, and 150 widgets, the LOWPRIORITY score would be 450. (We want it to be 0.)

DELAYS: Orders should be fulfilled in the order they were received to the extent possible. Use the order ids to determine when orders are fulfilled out of sequence. The order quantities are the measure the delay. Let's say order 10 (quantity 15) is produced before order 5 (quantity 20). Order 5 must wait while the 15 items for order 10 are produced, so add 15 delay points to the total delay. Sum across the entire production schedule. You only need to consider delays by looking at sequential pairs of orders in the production schedule. (It doesn't matter, for example, if order 9 was produced before order 10, further delaying order 5. The delay measure for each order is only based on the quantity of the order immediately preceding. The optimal approach is to schedule the orders in sequence (zero delay), but this will make other objectives worse.

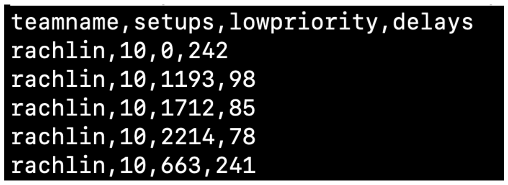
IMBALANCE (Optional / Extra Credit): The problem of scheduling a production machine could be generalized to scheduling *n* machines. Ten points extra credit will be awarded if you generalize your code to generate schedules for two machines instead of one. Let the *load* on any given machine be the total quantity of products being fulfilled on that machine. (Each order specifies the order quantity). A fourth objective must be included to minimize load imbalances between the machines. Formally, the objective can be defined as **IMBALANCE = max_load – min_load**. You will need to implement agents that move orders from one machine to the other in search of an optimal two-machine production schedule.

Deliverables

- Submit all code. Be sure to adhere to a functional style whenever possible. Favor list comprehensions and recursion over assignment statements and loops. Make sure your agents do more than simple swaps. Implement goal-directed agents also as it is the only way to produce competitive results.

- Include a visualization or visualizations that show the optimal tradeoffs. You might try using a **Seaborn PairPlot** for this purpose. A 3D scatter plot that shows the tradeoff surface would also be interesting. Feel free to extend the evo library with real-time visualization capability.

- Include a summary table (comma-delimited csv.) where the rows are the solutions in your final *non-dominated set* and the columns are the scores for each of the three objectives. Your CSV file must follow this EXACT format. Make sure your values are in the order shown.



```
teamname,setups,lowpriority,delays
rachlin,10,0,242
rachlin,10,1193,98
rachlin,10,1712,85
rachlin,10,2214,78
rachlin,10,663,241
```

If you do the optional extra credit, submit a *separate* solution summary of non-dominated solutions with a fifth column ("imbalance") for the measured load Imbalance.