A Production Planning System 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

Overview

In our last assignment, we used a *priority queue* to determine how orders should be fulfilled. In many process industries such as paper and steel, one must optimize for the constraints of the production machines to determine the most optimal schedule. 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 solution *alternatives* that reveal the tradeoffs. By understanding these tradeoffs, the plant-floor scheduling team can decide which solution is the best.

In this scenario, we have, as before, an order ID, the customer order priority (Low or High – no medium this time) and an order quantity. But we will also specify one of five *product* codes. Whenever the factory switches from making one product over another, the factory incurs a machine *setup*. This takes time and should be avoided whenever possible. So, the plant floor manager must come up with an order fulfillment schedule where each schedule is evaluated for the following three objectives:

1. **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 produce some orders out of sequence.

2. **LOWPRIORITY**: Produce the high-priority orders as soon as possible. The quantifiable goal is to minimize the total number of low priority items (sum total of their quantities) 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.)

3. **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 – 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.

The general approach to finding optimal scheduling tradeoffs.

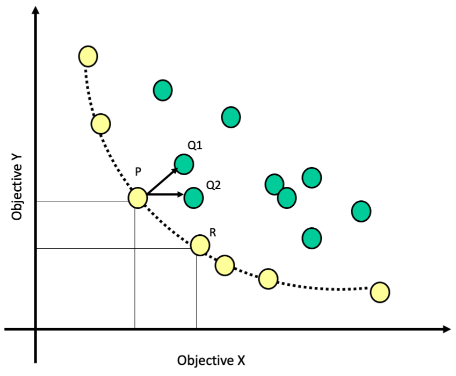
- Make free use of the evolutionary computing framework (evo.py) developed in class and attached to the assignment.

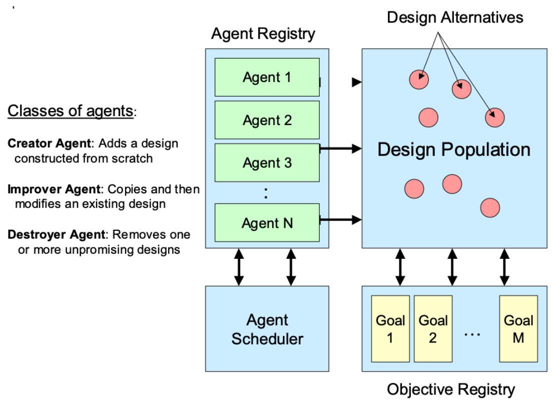
- Each production schedule is like an organism competing for survival in a dog-eat-dog world. Maintain a *population* of your solutions in some sort of data structure. Seed your population by generating and evaluating one or more random solutions. The solutions represent *organisms* living together in the natural world. To produce a random solution, the orders are simply shuffled in some random way.

- Introduce variation into your organisms. Variation is a critical component of evolution. Create one or more a mutator agents that picks a random solution from the population and modify the order sequence in some way. This produces a new child solution (offspring) that gets evaluated and added to the population. (The original parent solution is left unchanged.). The child solution might be better or worse. Not all mutations are beneficial. Most are harmful, *i.e.*, make the solution worse in some way.

- Periodically, a die-off occurs. (You must decide how often to trigger a die-off.) Solutions that are unfit are killed off. We are simulating the driving force behind evolutionary change: *natural selection.* The figure below shows the general approach. If a solution P is better than Q1 with respect to both objectives, then there is no reason to keep Q1. It should be culled. We say that P dominates Q1. But P and R can both be kept. P is better than R with respect to Objective X but R is better than P with respect to Objective Y. P does not dominate R and R does not dominate P. The idea extends naturally to three or more dimensions. Solution A dominates solution B if it is equal to or better with respect to every objective and strictly better with respect to at least one objective. Throw out the dominated solutions. Everything else is our *non-dominated set* which reveals the tradeoffs we care about.

- Continue this process repeatedly until no better solutions can be found. You will have discovered your optimal tradeoff curve!




Classes of agents:
Creator Agent: Adds a design constructed from scratch
Improver Agent: Copies and then modifies an existing design
Destroyer Agent: Removes one or more unpromising designs
Agent Registry
Agent 1
Agent 2
Agent 3
Agent N
Design Alternatives
Design Population
Agent Scheduler
Goal 1
Goal 2
Goal M
Objective Registry

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

1. Submit a zip file containing a directory with all your code.

2. Include some visualizations (images) that reveal the underlying tradeoffs. You might try using a **Seaborn PairPlot** for this purpose. A 3D scatter plot might also be interesting – it should reveal a tradeoff *surface.* Feel free to extend the evo library with real-time visualization capability.

3. 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. Column values should be listed in the following order: TeamName, Setups, High-LastIndex, Delays. Your team name should be 3-10 characters with no spaces. The TeamName identifies your team as the source of the solution. We will consolidate all the solutions to determine which team produced the best production schedules! If you do the extra credit, submit a *separate* solution summary of non-dominated solutions with a fifth column for the measured Imbalance.

4. Profile your code. Where is most of your time being spent when you execute your code? Document the percentage of time being spent running each agent and evaluating candidate solutions. We also want to know the total execution time of your program so be sure to profile your main method as well.

Some General Guidelines

- There are no bad orders in this scenario.

- The order IDs are just integers 1 to 100.

- Feel free to customize sorting.py (the sorting example given in class) to the requirements of the homework.

- You are free to modify evo.py but please *document the modifications clearly.*