A Production Planning System

DS3500: Advanced Programming with Data

Purpose

- Leverage class inheritance 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 quanity. 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. (Manufacture all orders of a given product together as much as possible). The optimal solution is to schedule all orders for each of the five products together, but this might delay high-priority orders or produce some orders out of sequence.

2. **HIGH-LASTINDEX**: Produce The high-priority orders as soon as possible. The quantifiable goal is to minimize the total number of items produced before the LAST high-priority order is fulfilled. Producing all of the high-priority orders first will minimize this objective but at the expense of setups and other order delays.

3. **DELAYS**: Orders should be fulfilled in the order they were received to the extent possible. If order 5 is produced after order 2, then order 5 has been delayed. Count 1 delay penalty point each time an earlier-arriving order is produced after a later-arriving order. Sum across all orders. For example, if we had 10 orders numbered 1 to 10 and the order fulfillment sequence was 5,3,4,6,7,10,9,8,1,2 this schedule would get a penalty score of 4 + 2 + 2 + 2 + 2 + 4 + 3 + 2 = 21 because 4 orders should have been produced before order 5, 2 orders before order 3, and so on. The optimal approach is to schedule the orders 1,2,3,4,5,6,7,8,9,10 (penalty = 0) but this will make other objectives worse.

The general approach to finding optimal scheduling tradeoffs.

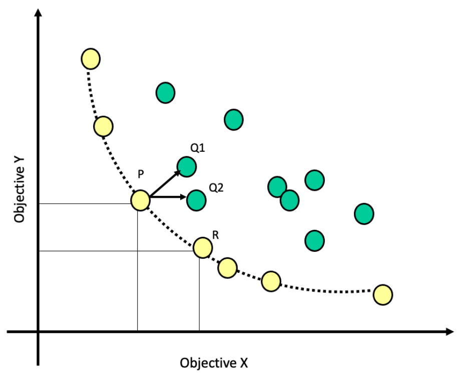
- Each production schedule is like an organism competing for survival in a dog-eat-dog world. Maintain a *population* of your solutions in some sort of data structure. Seed your population by generating and evaluating one or more random solutions. The solutions represent *organisms* living together in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

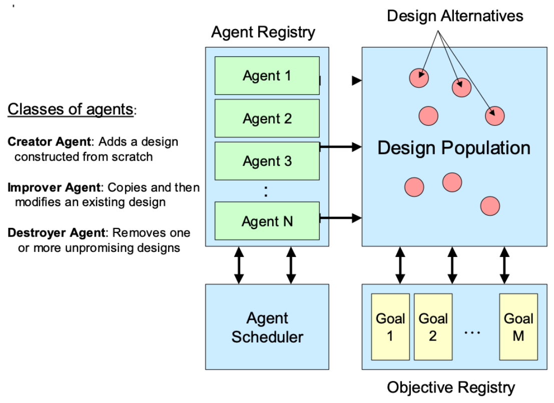
- Introduce variation into your organisms. Variation is a critical component of evolution. Create a mutator agent that picks a random solution from the population, and swaps two orders in the scheduling sequence. This produces a new child solution (offspring) that gets evaluated and added to the population. (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.

- **Optional**: Create other mutator agents that modify random solutions in more targeted ways. You might get better results if you do this but doing so is totally optional.

- 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
Agent Scheduler
Design Alternatives
Design Population
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 want to also add agents that randomly 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.*

3. Include a summary table where the rows are the solutions in your final *non-dominated set* and the columns are the scores for each of the three objectives. Include 1 column for a solution identifier (1,2,3,4…).

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.

Some General Guidelines

- There are no bad orders in this scenario.

- The order IDs are just integers 1 to 100.

- *Solution* is a generic class that contains an evaluation. The evaluation can be a simple dictionary mapping objective name to objective value. Each solution added to the population must be evaluated with respect to the three objectives defined above. A *Schedule* is a class that inherits from (i.e., extends) *Solution* while keeping track of the order-fulfillment sequence on the production machine.

- *Agent* is a base class. It has an abstract method *run()* which executes the agent. Problem-domain agents such as *CreateRandom* or *SwapOrders* or *RemoveDominated* all extend the base class *Agent* and must implement the *run()* method in a domain-appropriate way.

- *Population* is the class / object that contains solutions. Objectives are registered with the Population. Any solution that is added to the population should be evaluated automatically for each registered objective.

- *Objective* is a base class that has an abstract method *evaluate()*. Domain-specific objectives such as *MinimizeSetups* inherit from *Objective* and must implement the *evaluate* method for a given solution.

- The class hierarchy we have described here can be modified as you see fit but your framework should demonstrate the effective use of class inheritance at some level.