Algorithms and Data Structures Project

Michele Piccolini
M.Sc. Stochastics and Data Science

June 16, 2017

The problem

The problem that will be addressed in the present project is a discrete optimization problem about efficient scheduling of work units in a kitche. Namely: organizing the best schedule of preparations of dishes to assign to cooks in a kitchen.

During a typical day's work in a restaurant, several orders continuously arrive to the kitchen, containing requests about the preparations of several dishes. When the orders are many, or simply arrive too often, the organization of the work inside the kitchen becomes critical, especially if there are several cooks, since coordinating several minds is not an easy task. It often happens that there are some rush hours in which the restaurant is crowded, and the orders start to pile up, paving the way for confusion, in the case the requests are not well handled.

The present work aims to model the problem, taking account for realistic details, and then trying to address the issue by developing an algorithm that can handle orders and dishes, that can split complicated preparations into simple steps (or chunks of work), and that can assign those preparations in an optimal way to an arbitrary number of cooks.

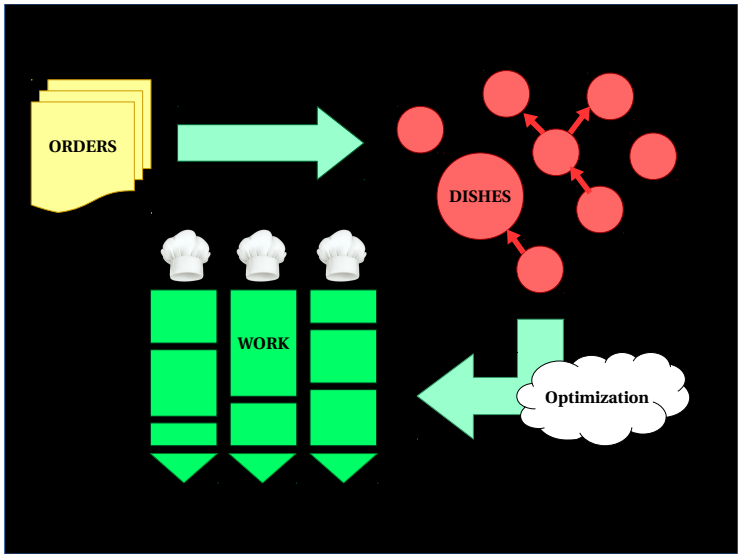

Figure 1: Overview of the problem

In figure 1 the overview of the modeled problem is represented. When implementing the code, several parts will be needed:

- Something that receives the orders and enucleates the working units that must be performed (preparation of dishes or of intermediate steps for complicated dishes)
- A proper data structure for representing the dishes
- A scheduler, which can take the "raw pool" of dishes/preprations as input, and that can come out with a solution which is an optimized scheduling of all the preparations

- A proper definition of the contraints and the loss function that the schedules should attempt to respect and minimize
- A way to represent the cooks in the kitchen. We will need just a simple representation of them, in particular we are just interested in taking account for their "workloads" (we want to know how much time they need to finish their jobs before being able to cook the new dishes).

Class of algorithm and Data Structures

The addressed problem is unfortunately very complex. It is a sibling of a well known optimization problem, called Job Shop Scheduling Problem. With reference to this, it can be cited: *The JSSP is not only NP-hard , but it is one of the worst members in the class.* [1]

Nevertheless, in this project we will try to solve the problem with a brute force approach. From the point of view of algorithmic complexity calculation, this is a challenging and interesting problem. On the other hand, in regards to the applicability, the inefficiency of this approach is hopefully a non-issue, since the practical aim is to run the algorithm *online*, meaning that in a real case scenario, one would only need to run the algorithm every time one or few new orders arrive. Therefore, the sheer amount of the number of dishes (= size of input) that the program has to "crunch" is bounded.

Data Structures

How are the data represented? The proposed solutions goes along the following lines (more details in the actual code):

- **Dishes** There is an abstract class Dish from which all the other (concrete) dishes inherit. The class Dish is a static one, which implements a factory design pattern (a static function) to easily instantiate actual dishes. Each actual dish is represented as a class, which inherits basic attributes and methods from Dish:
  - **self.ord** order to which the dish belongs.
  - **self.recipePrerequisites** a list containing the *names* of the dishes which are *prerequisites* for this dish. Only for complex/gourmet dishes.
  - **self.prerequisites** list that can contain the *actual instances* of the objects representing the prerequisites.
  - **self.tfinish** the time at which the dish is supposed to be ready after having being assigned to a cook and cooked.
  - **self.final** True if a dish is no prerequisite for any other dish. The "final" dishes are the ones that a customer can find on the menu.
  - **self.temporaryCook** index of the cook to which the dish is temporarily assigned during the optimization procedure.

(the methods are mostly getters and setters). Each concrete dish (derivating from Dish) implements its own values for **self.recipePrerequisites**, **self.tfinish**, and moreover it has its own preparation time: **self.T** (in seconds). A note on T: if a dish is composite (i.e. it has prerequisites), T is just the time needed for *ultimating this dish*, once the prerequisites are ready (each of them will have its own preparation time T).

Notice that the dishes, once instantiated, might show a tree structure, thanks to the list self.prerequisites. For example, when the dish "Roast Chicken Red Whine Demi Glace Polenta" (very tasty and complex!) will be ordered, an object of kind RoastChickenRedWhineDemiGlacePolenta will be instantiated, and also an object of kind RoastChicken and one of kind Polenta will be instantiated (those two happen to be the prerequisites for the original dish). Then the first objects will point to its prerequisites through self.prerequisites, and therefore the whole recipe shows a small tree structure (see 2. Children are drawn on the top, since they represent dishes that must be prepared *in advance* with respect to the dish below). Notice that the tree might be more deep and not necessarily binary.

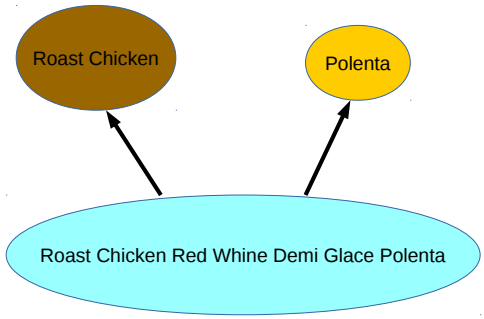

Figure 2: Example of complex dish preparation as tree

- **Orders** An order is represented with a class Order, containing the following members:
  - **self.requests** strings with ordered dishes' names
  - **self.timestamp** time at which the order is taken. The format of the time is the one outputted by time() (from python's "time" module), that represents the exact current date and time (in seconds units). This is also used, for simplicity, as a unique identifier for the order, since it is assumed that no two orders are taken at the same exact moment.
  - **self.min** minimum time at which one of the ordered dishes will be ready
  - **self.max** maximum time at which one of the ordered dishes will be ready (therefore the whole order will be ready to come out of the kitchen)

(the last two are used by the optimizer to compute the loss function). The class then has setters and getters.

New orders are also stored in a variable called "orders", a dictionary from which we can access to orders using their timestamps as keys.

- **Pool** The "pool" of dishes is the container that will contain dishes' objects coming from new orders. (It is the one that should contain all the red circles in the overview figure 1). The Pool is the list in which the ordered dishes (objects in the program) are stored. This list in then passed to the scheduler, which will reorder its items in all possible ways, and then it will cut this "long strip" into pieces (one for each cooks) and will assign those pieces to the cooks. See figure 3 for a visual representation of the workflow of the algorithm.

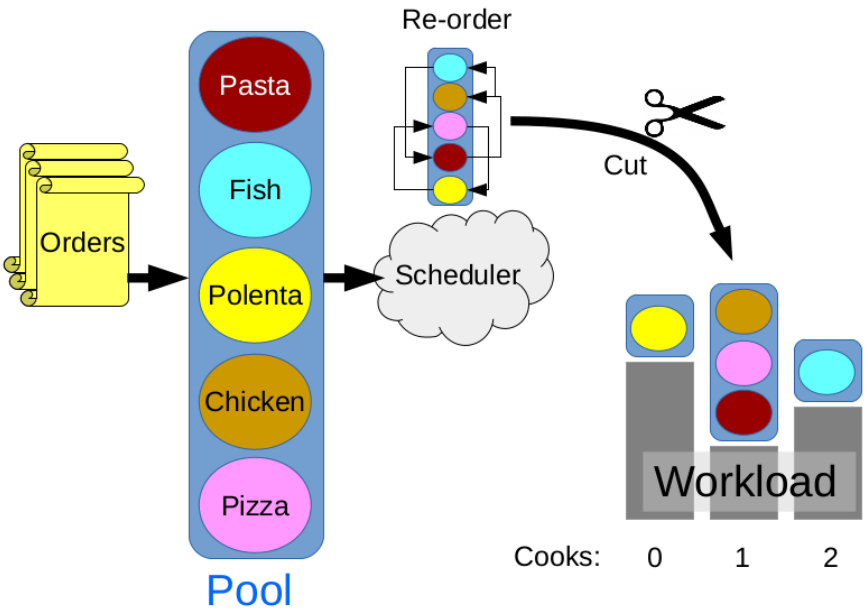

Figure 3: Overview of the algorithm

Then there is a function (called **inPool**) used for adding orders to the pool. This works by calling a recursive function (called **searchadd**) which scans a name of a dish in an order, instantiates one object of the requested type, and – if the requested dish is complex – calls itself on the prerequisites of the dish, so that it can instantiate also those as objects. This is the function that is responsible for creating the trees such as the one in fig. 2.

inPool is also responsible of registering an order in the dictionary "orders".

- **Schedule and M-Schedule** Once the pool is "cut", we will call the pieces "schedules". Since we will have M cooks, we will call the whole structure of m schedules: "M-schedule", with lack of fantasy.

For example, in fig. 3 an example of cut of the list of dishes is shown. From that, 3 schedules arise (which are shown directly over the cooks they are assigned to), and they are: [yellow], [red, pink, brown], and [cyan].
The corresponding M-schedule is the list of these 3: $\Big[$[yellow], [red,pink,brown], [cyan]$\Big]$.

In the implementation, a schedule will be represented by a python list, and an M-schedule by a list of lists. (A dictionary containing lists as values could be used as well, but we decided to index the cooks with numbers from 0 to M-1, hence the choice of list of lists).

- **Scheduler** The scheduler is implemented as a static class (since we only a singleton of it, and we need just its methods to run).

  The Scheduler class has several static variables, to store useful data for the optimization:

  - **numberOfCooks** the number of cooks in the kitchen
  - **bestMschedule** the list of lists used to store the optimal M-schedule found
  - **localCooksWorkloads** to make our problem realistic, we need a way to represent how much (already previously assigned) work the cooks have to finish *before* they can work on the new dishes coming from the optimal M-schedule. This variable is a list containing those times (in seconds). In fig. 3 this is depicted as gray rectangles.
  - **localOrders** reference to the "orders" variable
  - **optimizationStartTime** time at which the optimization procedure is started (set with time() call)
  - **cuts** list of the positions where to "cut" the pool. There are always M+2 entries. cuts[0] and cuts[M] are fixed at the values 0 and N respectively (where N is the total number of dishes in the pool). While the intermediate cuts (from 1 to M, one for each cook) can move between those values. See figure 4.
  - **lowestLoss** to store the current lowest value of the loss function
  - **alpha** "mixing" constant. Parameter of the loss function, with values in $[0, 1]$.

  The Scheduler then has the methods:

  - **initialize** to initialize the values for α, number of cooks, and workloads of cooks
  - **optimum** function that takes a pool of dishes, the list of orders, calls the scheduling functions, and returns the best M-schedule found. In practice, **optimum** only calls the function **generate**.
  - **generate** recursive function that explores all possible permutations of the list of dishes in the pool. For each permutation found, it calls the cutting function: **recursiveCut**.

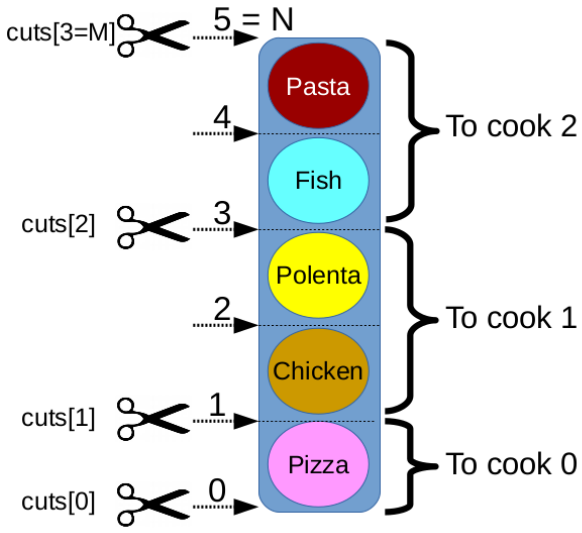

Figure 4: How cutting works: example with N = 5 dishes, M = 3 cooks

- **recursiveCut** as the name suggests, this is the function that recursively cuts the received permutation in all possible ways (by moving the position of the cuts). Every time the permutation is cut, a new M-schedule is created, and that is then passed to the actual optimizer, which checks if the M-schedule is legal, and then it computes its loss function.
- **validity** checks if an M-schedule is valid or not. A valid M-schedule is one in which all the dishes are scheduled to be prepared only *after* their prerequisites are supposed to be ready. The preparation of complex dishes may be split over different cooks (this is indeed the purpose of all this machinery). What matters, is that prerequisites are cooked first, since only when they are ready the assembly/preparation of complex dishes can be performed.

  The function **validity** works by traversing an M-schedule, and calling the function **check** only on the *final* dishes encountered.
- **check** this performs a depth-traversal on the tree of complex dishes. For each dish it encounters:
  - it computes the time at which the dish is supposed to be ready (sets **self.tfinish** for that dish)
  - calls itself on the prerequisites of that dish (continuing like this, it gets to the leaves of the tree, performing therefore a depth-traversal)
  - finally compares the tfinish of the prerequisites (returned by the recursive call) with the tstart of the dish it's visiting. If the former is greater than the latter, there is a scheduling inconsistency, and therefore the function returns an invalidity flag (a -1). Otherwise, if there are no problems, it returns the tfinish of the dish

it is currently visiting.

For example, suppose that **validity** is called on the left-hand M-schedule in fig. 5. Then it would start traversing the dishes (let's say, starting from the left-most schedule, from bottom to top). The first complex dish it encounters is the yellow (which has white as prerequisite), and therefore it calls **check** on it. **check** computes the time at which yellow should be started. Then it calls itself on the white. The tree was traversed (it was just a link), the time at which white should be finished is computed, and it is returned back to the original call. The original call compares the starting time of yellow, and the finish time of white, and finds them to be compatible. (The white is finished on time to allow for the preparation of the yellow, few minutes later). Where are the problems then? When **validity** then checks the next complex dish (red), it calls **check** on it. **check** calls itself on the prerequisite of red: dark green, extracts its finish time, and then returns it to the **check** called on red. The times are now incompatible! The red cannot be prepared in that position, since it need the dark green to be completed first! Therefore, **check** returns back an invalidity flag.

Notice that if red wasn't the root of the tree, but it was just an intermediate node, than what would have happened is that the invalidity flag would be "backpropagated" from the point where the first inconsistency was found, up to the root.

In the right-hand M-schedule of fig. 5 there are no inconsistencies, as one can easily check by eye.

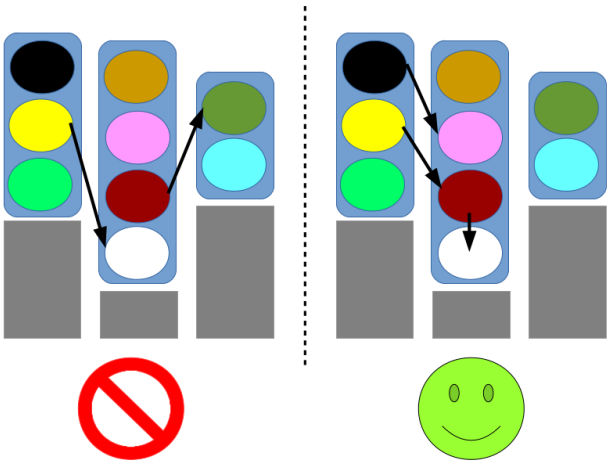

Figure 5: Invalid M-schedule and valid M-schedule

Bibliography

[1] Takeshi Yamada and Ryohei Nakano(1997), *Genetic Algorithms for Job-Shop Scheduling Problems*