Algorithms and Data Structures
Project

Michele Piccolini
M.Sc. Stochastics and Data Science

June 19, 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 of 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.

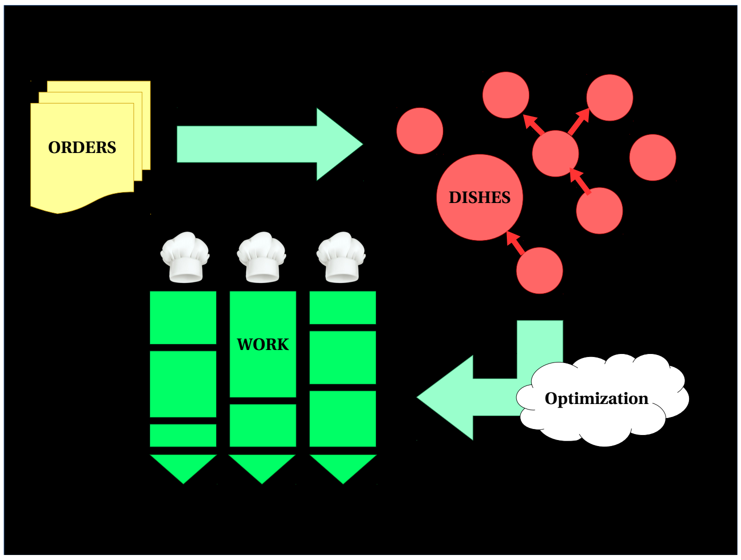

Figure 1: Overview of the problem

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 of 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.

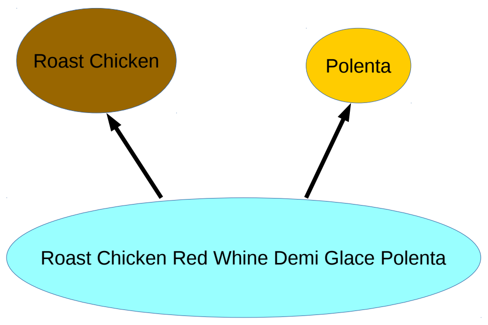

Figure 2: Example of complex dish preparation as tree

- **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.

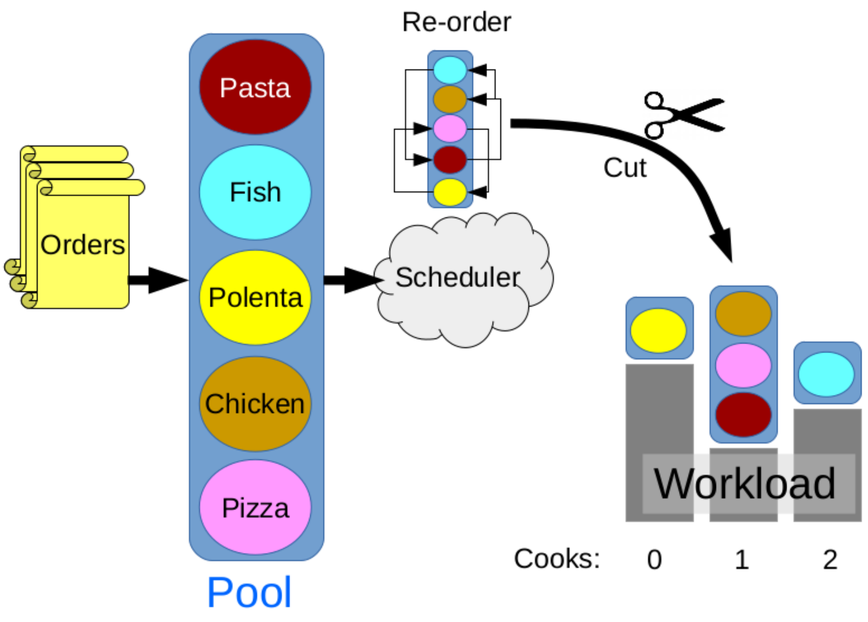

Figure 3: Overview of the algorithm

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
  - **oldestOrderTime** minimum timestamp among considered orders
  - **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** 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** this function takes the pool of dishes and the orders, then 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**.

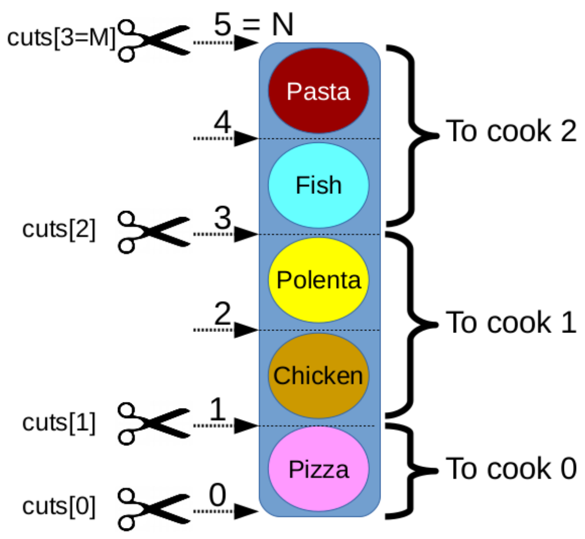

Figure 4: How cutting works: example with N = 5 dishes, M = 3 cooks

- **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. That is then checked by the optimizer, which first controls whether the M-schedule is legal, and finally computes its loss function. If a new better M-schedule is found, that one is stored in place of the former best one.
- **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 preparation of complex dishes can be done.

  The function **validity** works by traversing an M-schedule, and calling the function **check** only on the *final* dishes (the ones with prerequisites) encountered.
- **check** performs a depth-traversal on the tree of complex dishes. For each dish that 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 incon-

sistency, 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 needs 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, and 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, back to the root.

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

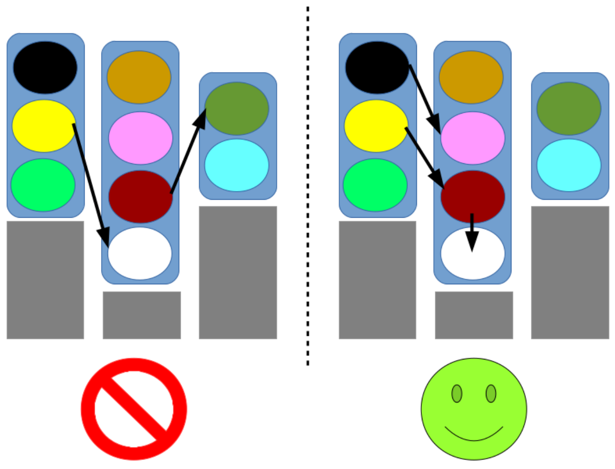

Figure 5: Invalid M-schedule and valid M-schedule

– **loss** the loss function. For a given M-schedule MS, it computes:

$$\alpha\,\mathbf{slowness}(MS) + (1-\alpha)\,\mathbf{coldness}(MS)$$

Now we see that $\alpha = 1$ means: give all the importance to the first factor. While $\alpha = 0$ all to the second.

- **slowness** first part of loss function. This function computes a "penalty value" for M-schedules, based on how slow will the analyzed M-schedule in completing the orders' requests. If, according to an M-schedule MS, every order o will be served in a time Δt_o, then

$$\textbf{slowness}(MS) = \max_o \Delta t_o$$

- **coldness** second part of the loss function, which considers the time span that the preparation of an order takes. We want all the dishes of one order to be prepared as close as possible in time, so not to let one dish get cold, waiting for the others to be prepared. (It is assumed that the dishes for an order go out of the kitchen *all together*, as soon as all of them are ready). Therefore, if for an order o, the first dish to be finished is finished at time t_o^{first} and the last is completed at t_o^{last}, we have:

$$\textbf{coldness}(MS) = \max_o (t_o^{last} - t_o^{first})$$

- **explain** prints the output

Complexity of the solution

The outline of the algorithm is in figure 6.

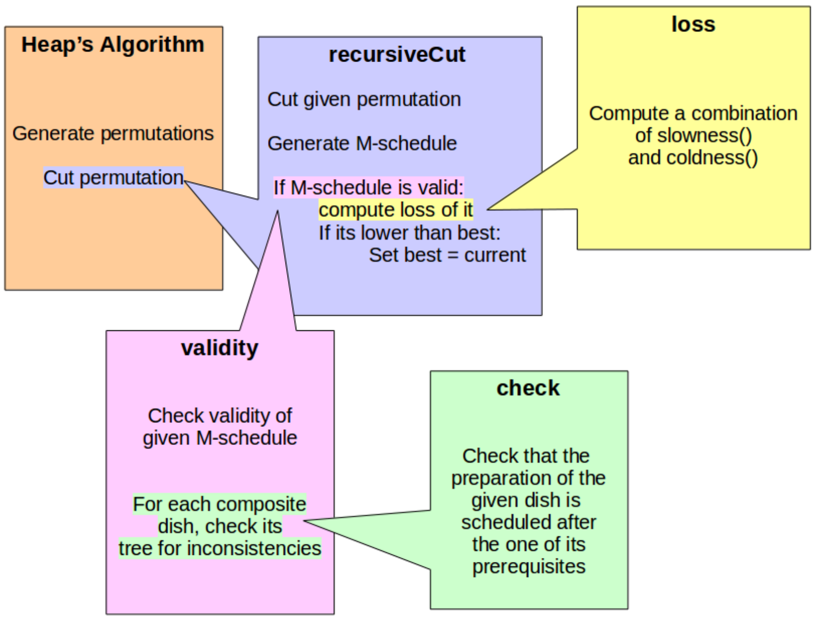
Heap's Algorithm
Generate permutations
Cut permutation
recursiveCut
Cut given permutation
Generate M-schedule
If M-schedule is valid:
compute loss of it
If its lower than best:
Set best = current
loss
Compute a combination of slowness() and coldness()
validity
Check validity of given M-schedule
For each composite dish, check its tree for inconsistencies
check
Check that the preparation of the given dish is scheduled after the one of its prerequisites

Figure 6: Outline of optimization algorithm

After the pool of dishes is filled, it can go through the scheduling algorithm. That will explore all the possibilities, discard the illegal ones, and return the best schedule possible.

Complexity of Heap's algorithm

First of all, we need a way to explore all possible ways to assign all the preparations in the pool to all the cooks. Each cook C (C in 1...M) can be assigned any number of dish (from 0 to n_C, with $\sum_{C=0}^{M-1} n_C = N = length(pool)$). The order in which the preparations are assigned to a cook matters! We are in front of the combinatorics problem of finding all the so called k-permutations (or weak permutations, or sequences without repetitions). We can break down this assignment problem into two separate, independent problems:

1) choosing *how many* preparations to assign to each cook (i.e. choosing n_C for each cook C)

2) choosing *what* dishes each cook must prepare (keeping in mind that for each cook we are picking n_C dishes from the pool).

The two subproblems are really independent, that is why we can deal with them in reversed order. More precisely, we notice that the problem is equivalent to:

1) finding all permutations of a list of N dishes

2) for each permutation found, cutting it into M pieces of varying length (as if it was a string, a rope that we can cut with our scissors). The resulting pieces will be – in order, starting from the beginning of the list – assigned to each cook. Notice that the length of a piece could also be equal to 0 (i.e. no preparations assigned to that cook).

See figure 4 for a visual idea of the cutting problem.

The latter approach is really the converse of the first one. Indeed, cutting a list (regardless of what its elements are) into M pieces is exactly choosing *how many* dishes we are going to assign to each cook. While caring about the contents of the list, and permuting them in all possible ways, is equivalent to *choosing* exactly *what* dishes are going to be assigned to each cook. It is like coloring the elements of the list (while the cut sees all the elements colorless).

Therefore, the skeleton of our algorithm is an algorithm for finding all the permutations of a given list: the so called **Heap's algorithm** (called **generate** in our code). The pseudocode for that is:

```
generate(n,L):
    if n==1:
        recursiveCut(1,L,numberOfCooks)
    else:
        for i in range(n):
            generate(n-1,L)
            if n is even:
                swap(L[i],L[n-1])
            else:
                swap(L[0],L[n-1])
```

It generates permutations of n elements of the list L in a recursive way. For each number i in 0,...,(n-1) it calls itself with parameter n-1. The idea is: before generating a permutation of n elements, generate a sub-permutation of n-1 elements. If n is 1, we have obtained one of the possible permutations. That is why the call to **recursiveCut** is nested in the "if n==1" block.

If n is 2, we need only to swap 2 elements to generate a new permutation. Indeed if n=2 the for loop reduces just to:

```
for i in [0,1]:
        generate(1,L) # i.e. outputs L as a permutation
        swap(L[i],L[1])
```

If n=3, the for loop will swap the first three elements in all the possible 3! ways, and so on.

This algorithm finds all possible permutations of a list of length N. Therefore, it will have a temporal complexity of $\Theta(N!)$ (ignoring the **recursiveCut** call). It works just by performing swaps on the list L, therefore its spatial complexity is $\Theta(N)$.

Complexity of Heap's algorithm + recursiveCut

On each permutation found by Heap's algorithm, we call this function:

```
recursiveCut(c,L,M):
        if c==1: cuts = [0]*M +[len(L)]
        # moves current cut to previous cut's position
        cuts[c] = cuts[c-1]
        # do stuff until this cut has reached the end
        while cuts[c] <= len(L):
            Mschedule = list of M lists
            # let the cuts above move first
            if c < (M-1):
                recursiveCut(c+1,L,M)
            # cut happens
            for j in range(M):
                Mschedule[j] = L[cuts[j]:cuts[j+1]]
            # check if the schedule violates contraints
            if validity(Mschedule):
                newLoss = loss(Mschedule)
                if newLoss < lowestLoss:
                    bestMschedule = Mschedule
                    lowestLoss = newLoss
            # advance the cut's position
            cls.cuts[c] += 1
```

The cutting part works by initializing M+1 "cuts' positions" (see fig. 4 to get the idea). The first and the last are fixed extremes, while the cuts indexed from 1 up to M-1 can be moved. And when the cut happens, their positions will determine "where will the scissors fall": all the dishes between the cut c and the cut $c + 1$ will be assigned to the cook c.

The cuts' positions must always satisfy the condition of being consecutive: cuts[c]>=cuts[c−1]. This is fundamental. Otherwise, think for example of having cuts=[0,3,1,5] (think about fig. 4): according to our rule for assigning dishes, we would have to assign Pizza, Chicken, Polenta (dishes between 0 and 3) to cook 0, and we would assign Chicken and Polenta (dishes between position 1 and 3) to cook 1, assigning the same dishes to different cooks!

The algorithm creates and empty M-schedule, then calls recursively the function on the cut c+1, until we reach the cut M-1 (the movable cut which is above all the previous ones). The list is cut (#cut happens), and then the cut M-1 moves up by one, then the list is cut again, and the cut moves up by one again, until it reaches the furthest position it can get to (=len(L), or N).

Every time the list is cut, an M-schedule is generated. That is immediately checked by **validity()**. If it is found valid, its **loss()** is computed, and if the value reaches a new minimum, the M-schedule is set as the current best M-schedule.

Let us now compute the complexity of the **generate** + **recursiveCut** algorithms (the two together), ignoring for the moment the complexity derivating from the **validity** and **loss** calls.

As we did for the Heap's algorithms, there is no need for computing explicitly the complexity. Instead, since this algorithm (**generate** + **recursiveCut**) explores all possible ways of choosing an M-schedule (with N dishes in the pool and M cooks), the complexity will be for sure O(number of possible ways), where we need to find the total "number of possible ways".

Turns out that this combinatorics problem can be solved very easily, if one has the right insight. Let us visualize our assignment problem as follows.

An M-schedule (with N dishes and M cooks) can be visualized as string like this:

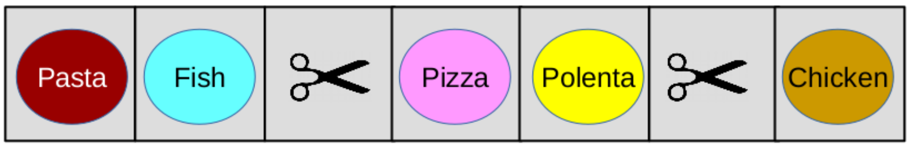

Figure 7: M-schedule as one string

Indeed this string could uniquely represent the choice of assigning [Pasta, Fish] to cook 0, [Pizza, Polenta] to cook 1, and [Chicken] to cook 2.

Notice that with N dishes and M cooks, we need a string that is N+M-1 long, since we only need M-1 "scissor symbols" to decide where to cut the string.

Then how can we visualize the problem of *choosing* an M-schedule? In fig. 9 we can see that the problem is this: we have an empty string with N+M-1 slots, and we want to assign N different labels (dishes) to N of those slots.

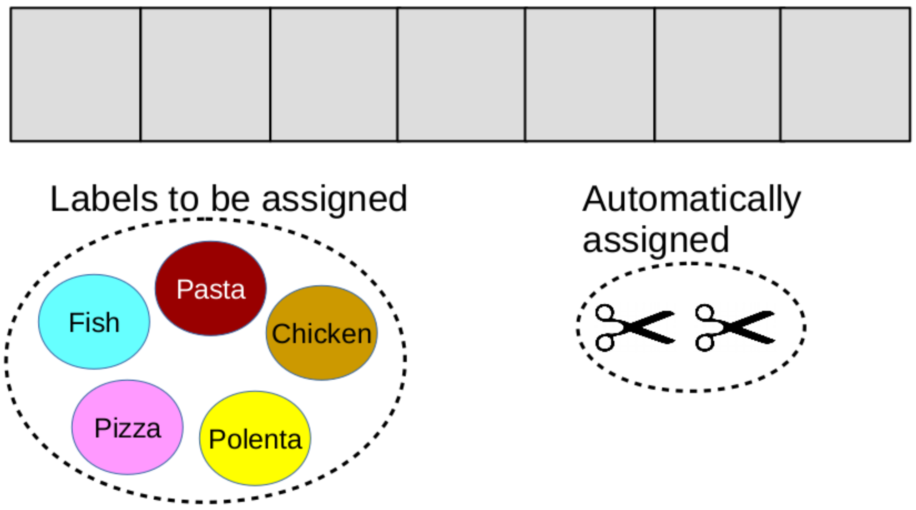

Figure 8: Choosing an M-schedule

We really only need to assign the dish labels, since then the ”scissor labels” will be automatically put in the remaining slots! (We could also avoid using such filler labels, and just think of breaking the string at the slots that remain empty after the dishes’ labels are assigned).

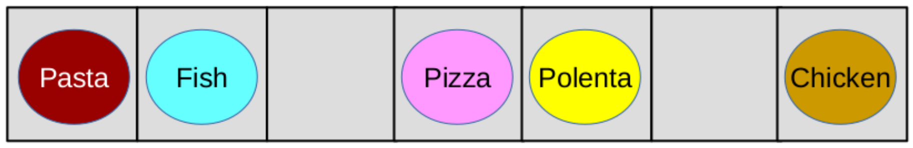

Figure 9: Just assign the dishes, and you get an M-schedule!

Then the question ”how many possible M-schedule can I build with N dishes and M cooks?” has clearly an answer, that is: ”the number of ways in which we can assign N objects in a container with N+M-1 slots”. This number is:

$$(N+M-1)\cdot(N+M-2)\cdot\ldots\cdot(N+M-M)=\frac{(N+M-1)!}{(M-1)!}$$

since we have N+M-1 ways of choosing the place for the first label, then N+M-2 possible places for the second, etc. up to the N-th label.

Therefore, the complexity of our algorithm is

$$O\left(\frac{(N+M-1)!}{(M-1)!}\right)$$

We can make an approximation, to get a more compact formula:

$$(N + M - 1) \cdot (N + M - 2) \cdot ... \cdot N < (N + M - 1)^N$$

which implies that the complexity is

$$O\Big((N + M - 1)^N\Big)$$

This is just the complexity for **generate** + **recursiveCut** algorithms. We ignored the **validity** and **loss** part, but we now proceed to include them into the calculation.

In the worst case scenario, all the generated M-schedules succeed in passing the validity test, then we have that for each of the $(N + M - 1)!/(M - 1)!$ M-schedules, our algorithm calls both **validity()** and **loss()**. Hence, an upper bound on the resulting complexity of the whole algorithm will be

$$O\Big((N + M - 1)^N \cdot (\text{complexity of validity} + \text{complexity of loss})\Big)$$

The space needed is very low though, since the function only needs to allocate a new local M-schedule (N elements), and a list for the cuts (M+1 elements). The spatial complexity is therefore $O(N + M + 1)$ which is basically $O(N)$ since the number of cooks won't grow for sure as the number of dishes. (Unless we want to bankrupt the restaurant by hiring an infinite amount of workers!)

Complexity of validity and check

```
validity(MS):
    for cook in range(numberOfCooks):
       for dish in MS[cook]:
          if i.is_final():
             # if there is a violation in a tree
             # then all the Mschedule is invalid
             if check(i,MS)==VIOLATION_FLAG:
                return False
         # if no violations were found, return True
         return True
```

```
check(i,MS):
    # set tfinish for dish i
    offsetSchedule = 0
        for precedingDish in MS[cook of precedingDish]:
            if precedingDish == i: break
            else: offsetSchedule += p.get_T()
    offset = workload of cook of i +
                 offsetSchedule + startTime
    i.set_tfinish(offset)
    # checks for violation in the tree of i
    for j in i.get_prerequisites():
        tFinish_j = check(j,MS)
        if tFinish_j==VIOLATION_FLAG:
            # if there was a violation somewhere
            # in the tree, backpropagates the
            # information until the initial
            # call of check() returns VIOLATION_FLAG
            return VIOLATION_FLAG
        else:
            # checks for violation: will j
            # be finised before i is started?
            if (tFinish_j > i.get_tstart()):
                # VIOLATION
                return VIOLATION_FLAG
    # if the dish has no prerequisites, or no violation
    # was found, return the tifinish of the dish
    return i.get_tfinish()
```

validity simply checks all the M-schedule for a violation, hence the commands inside the 2 for loops are executed N times. Let us suppose that in the worst case scenario we have to call **check** each of these N times. Then the complexity of **validity** is:

$$O(N \cdot \text{complexity of check})$$

check takes a dish i as input. It first gathers all the preparation times of all the dishes that are scheduled before the dish i, and uses this information to set the tfinish for i (time at which the preparation of i will be finished). In the worst case (all dishes scheduled in one single schedule, assigned to one single cook), this part has a cost of $O(N)$.

Then, **check** traverses the tree of prerequisites of dish i, by calling itself on "j in i.get_prerequisites()". After the call check(j,MS), only a pair of if statements are performed.

In practice, the depth of the tree of a dish will be very limited (it makes no sense considering the asymptotic limit for an infinitely-complex-dish that has a number of prerequisites growing towards infinity). Thus, we can assume the number of traversed nodes in the tree as a constant number K. The most computationally expensive part of the check function is the first one, which is at most linear in N, therefore we can roughly say that the whole cost of **check** is

$$O(KN) = O(N)$$

Therefore, **validity** is at most quadratic:

$$O(N^2)$$

The spacial complexity is negligible, considering that these functions allocate just few local variables.

Complexity of loss

```
loss(MS):
    return alpha*slowness(MS)+(1-cls.alpha)*coldness(MS)
```

```
slowness(MS):
    slowestPrepTime = 0
    for S in MS:
        for i in S:
             if i.get_tfinish() > slowestPrepTime:
                 slowestPrepTime = i.get_tfinish()
    return slowestPrepTime - oldestOrderTime
```

```
coldness(MS):
    for S in MS:
         for i in S:
              if i.get_tfinish() > i.get_ord().get_max():
                  i.get_ord().set_max(i.get_tfinish())
              elif i.get_tfinish() < i.get_ord().get_min():
                  i.get_ord().set_min(i.get_tfinish())
     worstRange = 0
     for o in orders:
          r = o.get_range()
          if r > worstRange:
               worstRange = r
     return worstRange
```

loss makes just a call to **slowness** and **coldness**.

slowness makes a simple operation for all the dishes in the given M-schedule. Therefore its complexity if $O(N)$

coldness also does few simple operations for each element in the M-schedule. then, it traverse all the orders to perform another constant number of operations. In a real case scenario, the number of orders won't grow as fast as N, the number of dishes in all the orders. Therefore the second part of the function is computationally negligible. The only heavy part is the first one, which gives $O(N)$.

To wrap up, **loss** is linear in N.

The space complexity is also in this case negligible.

Time complexity of whole algorithm

We finally got to the conclusion that the time complexity of the whole procedure is:

$$\begin{aligned}
&O\Big((N+M-1)^N \cdot (\text{complexity of validity} + \text{complexity of loss})\Big)\\
=\,&O\Big((N+M-1)^N \cdot (N^2+N)\Big)\\
=\,&O(N^2(N+M-1)^N)
\end{aligned}$$

The problem was really NP!

Testing

The algorithm is very slow at running, as expected. Some tests were done, will a small size of the input, to understand whether the program outputs reasonable schedules. In the test we tried many different values for the parameters (mainly number of cooks and α), and the results vary accordingly to those parameters. We put here just a couple of results, as examples.

6 dishes, 2 orders, 2 cooks

The code was tested with 2 small orders:

- order 1) [at 15:36] Pizza Margherita
- order 2) [at 15:46] Crispy Fish Tacos with Spicy Yogurt Sauce

The orders are made of just one dish, but the preparation of the second dish actually hides a tree of preparations with 5 nodes! Indeed, to prepare Crispy Fish Tacos with Spicy Yogurt Sauce, one has first to prepare Crispy Fish, Tacos, and Yogurt Sauce. And Crispy Fish need first Cleaned Fish to be prepared!

Therefore, the pool of preparations to be scheduled will contain 6 dishes.

In the listings below we can see the output of the program, which communicates the optimal schedule (with precise time indications for the preparation of each dish), the value of the loss function for that schedule, and the value of alpha used.

6 dishes, 2 orders, 2 cooks (no workloads), alpha = 0.5

```
Best M-schedule: (loss:     1710.0, slowness constant: 0.5)
cook 0
 0) from 15:46 to 16:00: PizzaMargherita* (ord. 15:36)
 1) from 16:00 to 16:07: CrispyFishTacosWithSpicyYogurtSauce* (ord. 15:46)
cook 1
 0) from 15:46 to 15:48: CleanedFish (ord. 15:46)
 1) from 15:48 to 15:53: CrispyFish (ord. 15:46)
 2) from 15:53 to 15:57: Tacos (ord. 15:46)
 3) from 15:57 to 15:58: YogurtSauce (ord. 15:46)
done! in 0.223242 seconds
```

The scheduling is well done. There are no violations in the preparations. The pizza is ready at 16:00, while the fish is ready at 16:07. By pure chance, it happened that the cooks have split the type of works: cook 1 does the intermediate preparations, while cook 0 does the final preparations (pizza and last step of fish), marked with a * at the end of their name.

8 dishes, 3 orders, 3 cooks with workload

A little more complicated example is the following, in which we have an order of pizzas, an order of gourmet dishes, and an order of japanese dishes (there is a lot of diversity in our restaurant!):

- order 1) [at 16:03] Pizza Margherita x 2, Pizza Diavola
- order 2) [at 16:04] Filet Mignon with Rich Balsamic Glaze (which requires Filet Mignon to be prepared first), Ultimate Gourmet Grilled Cheese

- order 3) [at 16:05] Miso Soup, Sashimi

We have a total of 8 dishes. We assume to have 3 cooks in our kitchen, and we assume that cook 0 has 10 minutes of workload, hence he won't be able to cook new dishes within the next ten minutes!

At 16:05 the scheduler takes in the most recent orders, and finds the optimal schedule for preparing them, which is the following:

6 dishes, 2 orders, 2 cooks (no workloads), alpha = 0.5

```
Best M-schedule: (loss:      2820.0, slowness constant: 0.5)
cook 0
 0) from 16:15 to 16:29: PizzaMargherita* (ord. 16:03)
 1) from 16:29 to 16:44: PizzaDiavola* (ord. 16:03)
cook 1
 0) from 16:05 to 16:19: PizzaMargherita* (ord. 16:03)
 1) from 16:19 to 16:34: FiletMignonWithRichBalsamicGlaze* (ord. 16:04)
cook 2
 0) from 16:05 to 16:10: FiletMignon (ord. 16:04)
 1) from 16:10 to 16:16: UltimateGourmetGrilledCheese* (ord. 16:04)
 2) from 16:16 to 16:22: MisoSoup* (ord. 16:05)
 3) from 16:22 to 16:27: Sashimi* (ord. 16:05)
done! in 131.228272 seconds
```

The schedule seems great. And we can notice the nice fact that less work was assigned to cook 0 (indeed he is scheduled to start cooking a pizza at 16:15, while the others start immediately!).

Bibliography

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