Cooking Time Optimization on Given Machine Constraints and Order Constraints

Chi Fang, Guoyi Jiang, Moqing Shi, Ling Xu, Lanxi Zhang

December 17, 2016

1 Abstract

In this project we investigate the problem of how food orders from restaurant customers should be processed and completed in particular kitchen settings in an optimal way. Our goal is to minimize the total preparation time for food orders (from when orders come in to when they are completed) in a realistic kitchen scenario. We explore this problem with assumptions and constraints with which we can apply Integer Programming techniques and create cooking guidelines for chefs by makeing sense of our program output. Real life scenario that we try to duplicate in our program is that the number of kitchen appliances is limited and the ordering of how each job of each dish should be cooked is pre-determined. We formulate and conquer the problem with two approaches: the Integer Programming Approach and the Graph Approach. As a result, the Integer Programming approach guarantees an optimal solution with respect to given constraints but sacrifices runtime. In contrast, the Graph approach is easier to set up but becomes costly under large number of tasks and does not guarantee optimality. In conclusion, the Integer Programming approach is robust even with complex problems. Based on our minimum time solution, we also try to reduce the number of chefs needed in the kitchen setting. The result can be used for restaurant owners for considerations over time and cost. We further extend our Integer Programming approach to solving other real life scheduling problems.

2 Problem Introduction

Eating plays a major part in our everyday life. Busy people sometimes find it hard to manage time for this activity, so they hope either to eat at restaurants that don't keep them waiting for too long or to be instructed about how to cook efficiently at home. Either way, the construction of a procedure guidance that satisfies all operational rules and equipments availability in a kitchen setting, while at the same time completes dishes as quickly as possibly is very much needed. In most restaurants, this task is left to experienced chefs to determine. This worked in the past, but now with more orders coming in everyday and the need to cut operational costs, some restaurants are seeking for an operational research approach to solve this problem. With these demands in mind, and in light of the progress achieved in restaurant industry, we hope to work on the cooking scheduling problem in order to develop an automated program for constructing efficient procedure guidances.

Formally, the cooking scheduling problem is defined as the process of assigning each kitchen equipment a specific job in a specific time period that completes orders from customers as quickly as possible. The basic goal of a cooking scheduling problem is to minimize the total time used to cook, subject to kitchen resource constraints and task requirements. Some resource constraints include kitchen equipment setting, manpower; and some task requirements include job priority within a dish, dish delivery time, etc.

For our project, we discuss two different approaches to solve this cooking scheduling problem. One method of finding an optimal solution is to set up linear constraints to formulate an integer programming problem; the other is a graph approach. We will explain more in the following sections.

3 Integer Programming Approach

3.1 Problem Formulation

3.1.1 Assumption

To reduce the complexity of a real-life problem, we have the following assumptions:

1. We have unlimited number of chefs with equal ability to do all jobs

2. We use natural numbers to represent all the time we need to do jobs (including the starting time and duration of each job)

3. We ignore the time taken in between different jobs

3.1.2 Problem Settings

Our objective is to schedule each job to each machine in a way that minimizes the longest running operation.

The basic idea of our integer programming model is as follows.

- We are given recipes for $n \in \mathbb{N}$ dishes respectively. Each recipe corresponds to only one dish and specifies how many jobs that dish requires.

- Since different dishes have different number of jobs to be completed, we define $j \in \mathbb{N}$ to be the maximal number of jobs among all dishes.
  Mathematically, we let $j = \max\limits_{1 \le i \le n}\{\text{number of jobs in dish}_i\}$.
  For example, when given two dishes with 5 jobs in dish_1 and 7 jobs in dish_2, we define $j = \max\limits_{1 \le i \le 2}\{5, 7\} = 7$.

- For representation purposes, we let $d_{i,j}$ denote the j^{th} of jobs of dish i. So we have $\{d_{i,1}, d_{i,2}, \cdots\cdots, d_{i,j}\}$ for each $i \in [n]$.

- We let $m \in \mathbb{N}$ denote m types of machines in our kitchen setting. Each machine is assigned with a distinct functionality which is mapped to a limited types of jobs.
  For example, a pan can only be used to fry.

- We let $t_{i,j}$ denote the duration of job $d_{i,j}$. Each job $d_{i,j}$ must be processed continuously on its corresponding machine during time $t_{i,j}$

- We let $T_{i,j}$ denote the starting time of job $d_{i,j}$. Consequently, the end time of job $d_{i,j}$ can be represented as $T_{i,j} + t_{i,j}$.

- We let C_{m_i} denote the capacity of machine m_i. The number of jobs each machine m_i can handle at one time is restricted to its capacity C_{m_i} jobs at a time.

  For example, if $C_{\text{pan}} = 2$, then there are two pans in the kitchen so at most two different jobs that require a pan can be processed at the same time.

3.1.3 Formulation

The Integer Programming model uses variables in the form of $x_{i,j,m,t}$. Each index of the variable indicates an attribute The value of each variable is a binary 0/1 value. When a variable $x_{i,j,m,t}$ is equal to 1, it represents that the job $d_{i,j}$ is being operated on machine m at time t. All variables can be visualized in a matrix with $i \times j$ rows and $m \times t$ columns and the first non-zero variable in each row is the starting time $T_{i,j}$ of the job. Therefore, the completion time for each job is $T_{i,j} + t_{i,j}$ as $t_{i,j}$ is the duration (processing time needed) for job $d_{i,j}$.

Here we take a look at an example:

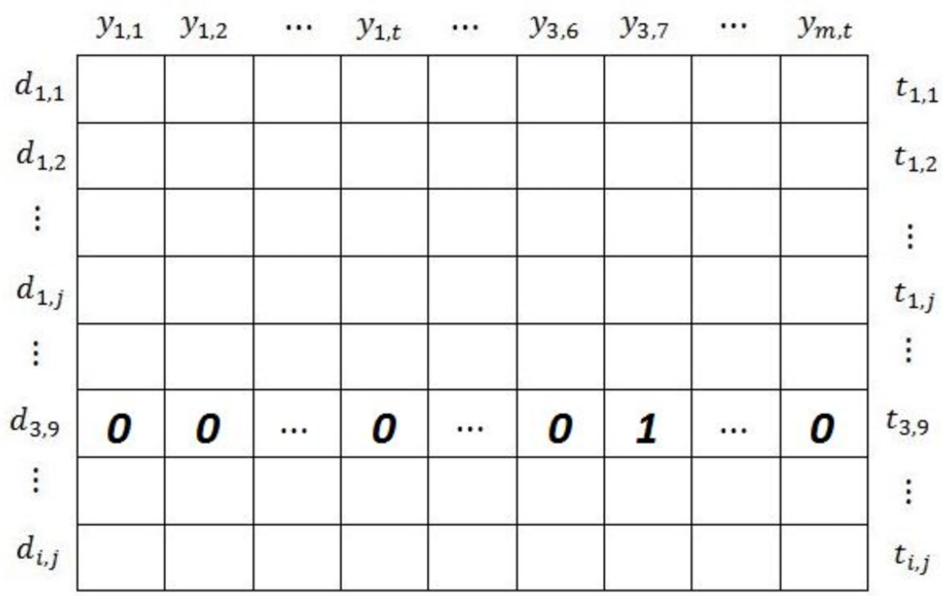


Figure 1: Visualization of all 0/1 variables in an $i \times j$, $m \times t$ matrix.

In Figure 1, since the variable $x_{3,9,3,7}$ is the first non-zero variable in that row, we know that the starting time for job 9 of dish 3 is the 7th minute.

Our objective then defined to minimize the maximum completion time T_{max},

$$\min T_{max} \tag{1}$$

where $T_{max} = max(T_{i,j} + t_{i,j})$ for all i and $j \in [n]$.

Based on such formulation in the Integer Programming model we are able to specify constraints in the model, which we will discuss in the next section.

To name a few, for priority constraint, if job d_{i,j_1} happens before job d_{i,j_2}, we require the starting time of job d_{i,j_2} to be larger than the completion time of job d_{i,j_1}:

$$T_{i,j_2} > max\{T_{i,j_1}\} + t_{i,j_1} \tag{2}$$

And for machine capacity constraint, the number of jobs being operated on each machine m should be less than or equal to the capacity C_m at any time:

$$\sum_i \sum_j x_{i,j,m,t} \leq C_m \quad \text{for all } t \tag{3}$$

3.2 Constraints and Objective Function

Note that there is no such representation for the starting time of each job $T_{i,j}$ in any linear form, so we try to find the corresponding linear version of the original objective function and constraints without the presence of $T_{i,j}$.

We try to find the minimum time needed to complete all dishes. We take the sum of completing each step sequentially to be the upper bound, and use binary search to find the first optiomal solution. That is, we run the integer programming solver using a fixed total completion time at each iteration. If the time provided were too short, any scheduling would be "undefined" or infeasible; on the other hand, if the upper bound gives plenty of time, we find the middle point between upper bound and lower bound and try to find an optimal solution within half of the previous range. Thus, the minimum completion time that we are looking for becomes the time at which *the first optimal solution come into place.*

All constraints formulated using the 0/1 variables are shown as follows:

1. Machine Assignment Constraint

2. Completeness Constraint

3. Machine Capacity Constraint

4. Continuity Constraint

5. Priority Constraint

3.2.1 Machine Assignment Constraint

If job $d_{i,j}$ is assigned to machine m:

$$\sum_t x_{i,j,m,t} = t_{i,j} \quad \text{for that particular } m \tag{4}$$

3.2.2 Completeness Constraint

Although the relatively local Machine Assignment Constraints have already ensured that all jobs would be operated and completed by certain machines, we still need the global Completeness Constraints to prevent a job that has already been finished from being processed by another one at a second time:

$$\sum_t x_{i,j,m,t} = t_{i,j} \quad \text{for all } i, j \text{ and } m \tag{5}$$

3.2.3 Machine Capacity Constraint

Same as equation (3).

3.2.4 Continuity Constraint

In order to make sure that each job is being processed on the assigned machine continuously, we require:

$$|x_{i,j,m,t_1} \cdot t_1 - x_{i,j,m,t_2} \cdot t_2| \leq t_{i,j} \text{ if both } x_{i,j,m,t_1} \text{ and } x_{i,j,m,t_2} = 1 \tag{6}$$

$$\text{for each job } d_{i,j} \text{ and its corresponding machines } m$$

However, as mentioned before, in order to be fit into the integer programming model, we need to convert the constraint into a linear form. Thus, we split the absolute value into two parts and using a penalty constant M (which should be very large compared to $t_{i,j}$, i.e. 10000) to limit the power of this constraint only to the case where both x_{i,j,m,t_1} and x_{i,j,m,t_2} equal to 1:

$$x_{i,j,m,t_2} \cdot t_2 - x_{i,j,m,t_1} \cdot t_1 \geq -t_{i,j} - M \cdot x_{i,j,m,t_1} - M \cdot x_{i,j,m,t_2} \tag{7}$$

and

$$x_{i,j,m,t_2} \cdot t_2 - x_{i,j,m,t_1} \cdot t_1 \leq t_{i,j} + M \cdot x_{i,j,m,t_1} + M \cdot x_{i,j,m,t_2} \tag{8}$$

$$\text{for } t_2 - t_1 \geq t_{i,j} \text{ and for each job } d_{i,j} \text{ and its corresponding machines } m$$

3.3 Priority Constraint

Since we cannot use the starting time T_{i,j_1} of job d_{i,j_1} and the completion time of job d_{i,j_2} for comparison, we used the midpoint between the starting and ending time for each job instead:

$$\sum_m \sum_t \frac{x_{i,j_2,m,t} \cdot t}{t_{i,j_2}} - \sum_m \sum_t \frac{x_{i,j_2,m,t} \cdot t}{t_{i,j_1}} \geq \frac{t_{i,j_1} + t_{i,j_2}}{2} \tag{9}$$

3.4 Case Study

In order to demonstrate the capabilities of the proposed model for the solution of a real-life cooking scheduling example, we choose an order with three simple dishes. The first dish is salad, the second dish is steak and the third dish is egg tarts.

We also specify the available kitchen appliances. We have 1 fry pan, 2 sinks, 1 microwave, 2 cutting boards, 2 boiling pots, 1 oven and 2 bowls in the kitchen setting. The pan can be used to fry; the sink can be used to wash; the microwave can be used to heat or defrost things; the cutting board can be used to cut meats or vegetables; the pot can be used to stew food or boil liquids; the oven can be used to bake; and the bowl is versatile as it can be used for resting hot food, seasoning, etc.

Note that in the third dish, baking crust and boiling milk and sugar share the same priority. This means that if we have more than one chef, we can do these two jobs in the kitchen in parallel, which reduces the total time.

In the following page, Table 1 specifies our kitchen settings and cooking priorities. In section 3.5 we provide computational results from the optimization process of our model.

Table 1: Case Study Example Specification

	Time	Priority	Machine
Dish1: Salad			
Wash vegetables	1 min	1st	sink
Chop vegetables	1 min	2nd	cutting board
Season vegetables	1 min	3rd	bowl
Mix vegetables	1 min	4th	bowl
Dish2: Steak			
Defrost Steak	5 mins	1st	microwave
Season Steak	1 min	2nd	bowl
Rest Steak	10 mins	3rd	bowl
Fry Steak	5 mins	4th	pan
Decorate Steak	2 mins	5th	cutting board
Chop Steak	1 min	6th	cutting board
Dish3: Egg Tart			
Mix pie crust	3 mins	1st	bowl
Bake crust	10 mins	2nd	oven
Boil milk and sugar	5mins	2nd	pot
Bake again	5 mins	3rd	oven
Rest to cool	2 mins	4th	bowl

3.5 Computational Results

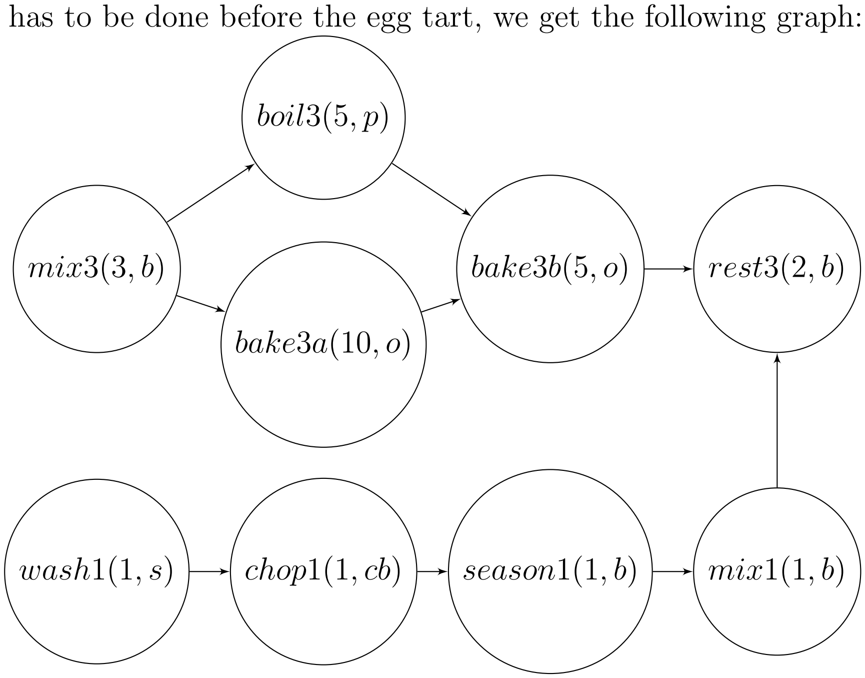
```
Status:  Optimal
Guidance for dish 1 is the following:
     ->Wash on sink from time 2 to 3
     ->Chop on cutting board from time 3 to 4
     ->Season on bowl from time 4 to 5
     ->Mix on bowl from time 5 to 6
Guidance for dish 2 is the following:
     ->Defrost on microwave oven from time 0 to 5
     ->Season on bowl from time 5 to 6
     ->Rest on bowl from time 6 to 16
     ->Fry on pan from time 16 to 21
     ->Decorate on cutting board from time 21 to 23
     ->Chop on cutting board from time 23 to 24
Guidance for dish 3 is the following:
     ->Mix on bowl from time 1 to 4
     ->Bake on oven from time 4 to 14
     ->Boil on pot from time 5 to 10
     ->Bake on oven from time 16 to 21
     ->Rest on bowl from time 22 to 24
```

4 Graph Approach

4.1 Graph Setup

To think about the same problem with a different approach, we can consider this same problem in a graph approach. The main goal for the problem is that we have to finish some amount of tasks, and we can draw the parallel between finishing these tasks with visiting nodes in a graph. Therefore, we can set up the nodes in the graph to be the different tasks we need to finish. For each node, we would have to give the nodes some weight to represent the amount of time it takes for us to fully visit the node (in parallel to finishing the task). And the objective for this graph problem would be to finish visiting all the nodes in the shortest amount of time possible. Notice that this graph problem would be different to a lot of the other problems in the sense that multiple nodes can be visited at the same time. And to represent the constraint that job i needs to be done before job j, we can use a directed edge going from node i into node j to represent that node i has to be visited before node j. And to impose the constraint with limited machines, we can just specify that some of the nodes cannot be visited at the same time.

For example, if we look at the salad and egg tart recipe from the previous example we used for integer programming approach, and to connect the two dishes, we specify that salad has to be done before the egg tart, we get the following graph:



As shown above, each of the job for each dish is a node in the graph, for example, wash 1 means the step to wash the vegetable in dish 1 (salad). Since there are two baking steps, we can represent the one that needs to be done first as bake3a (first baking step in dish 3, egg tart). The edges in the graph would represent the order constraint, for example there exists an edge going from wash1 to chop 1 because in the salad recipe, the vegetables need to be washed before chopped. And there exists an edge going from mix1 to rest3 because we want dish 1 to be done before dish 3. Moreover, we can label the node with the weight of the node, which represent how long it takes to finish the node or how long it needs to stay in the node (the number in the parenthesis). And we can label each node with the machine it uses (the character in the parenthesis). And our goal to solve this graph would be to visit every node with the shortest amount of time.

4.2 Graph Solution : Heuristics

4.2.1 Solution

To solve this graph problem from the beginning nodes to the end node by steps, we need to keep track of two sets at each step: (1) a *Working set* to keep track of which are the nodes that we are working on, and (2) a *Done set* to keep track of the nodes we have already visited.

We also need to keep track of the time we are at the end of each step. We can divide every step into two sub steps. For sub-step 1, we try to solve the nodes in the working set. To do so, we find the node in the working set that has the smallest weights and put it into the done set, at the same time reduce all the other working nodes' weight by this smallest weight. For sub-step 2, we try to add more nodes into our working set. To do so, we look at all the remaining nodes in the graph (that are not in working or done set), and add them to the working set if it satisfies 2 conditions:

- Condition 1: all of its previous nodes (define: i is a previous node of j, if there exists an edge from node i to node j) are in the done set or it does not have a previous node.
- Condition 2 : if it uses machine m, the number of nodes in the working set using machine m is less than the number of m we own subtracted by 1.

Notice that condition 1 takes care of the Priority constraint, and condition 2 takes care of the Machine Capacity constraint.

Finally at the end of the step, we increase our current time by the smallest weight. And we stop this process if and only if all nodes are in the done set.

To solve the above example, we start with set W (the working set, initially empty), set D (the done set, initially empty) and T (time) $= 0$. Since not all the nodes are in the done set. We start with the the first iteration of steps. For sub-step 1, we cannot do anything since it is empty. For sub-step 2, we can move mix3 and wash1 into the working set since they do not have previous nodes and they use different machine. So at end of the iteration, we have W = {mix3, wash1}, D = ∅ and T = 0, and the weights of nodes stay the same.

For the second iteration of the steps, in sub-step 1, we can see that wash1 has weight 1. Therefore, we can add it to the done set. Now, for sub-step 2, we can add chop1 into the working set because all its previous nodes are in the done set, and again chop1 and mix3 use different machines. Finally we have to add 1 to T. Therefore at the end of iteration 2, we have W = {mix3, chop1}, D = {wash1}, and T = 1, and the graph becomes the following (weight for wash1 becomes 0 and weight for mix3 becomes 2).

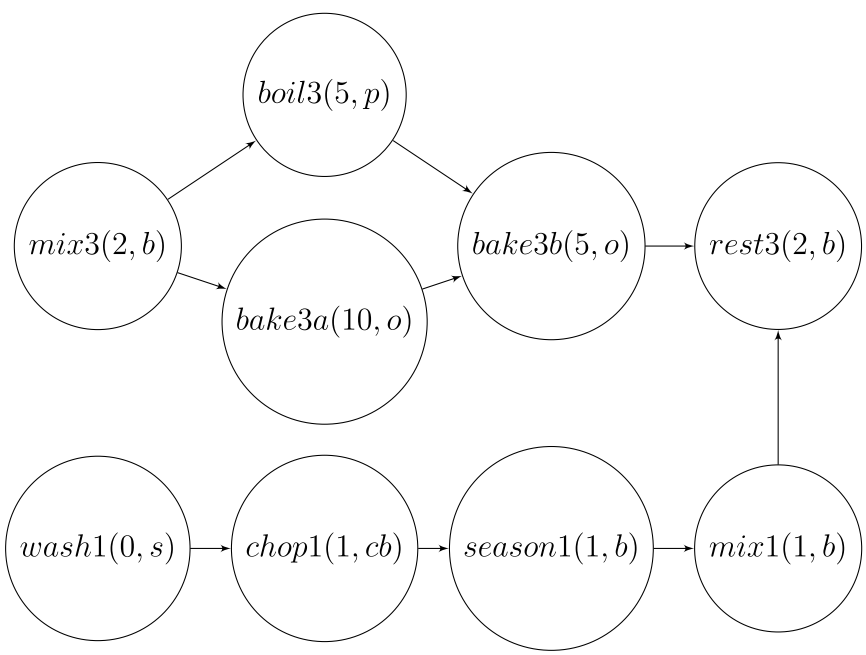


Table 2: Heuristic process

iteration	Working set	Done set	Time
0	{}	{}	0
1	{mix3, wash1}	{}	0
2	{mix3, chop1}	{wash1}	1
3	{mix3, season1}	{wash1,chop1}	2
4	{boil3, bake3a, mix1}	{mix3,wash1,chop1,season1}	3
5	{boil3, bake3a}	{mix3,wash1,chop1,season1, mix1}	4
6	{bake3a}	{mix3,boil3,wash1,chop1,season1,mix1}	8
7	{bake3b}	{mix3,boil3,bake3a,wash1,chop1,season1,mix1}	13
8	{rest3}	{mix3,boil3,bake3a,bake3b,wash1,chop1,season1,mix1}	18
9	{}	{mix3,boil3,bake3a,bake3b,rest3,wash1,chop1,season1,mix1}	20

The entire solving steps are shown in the above table.

4.2.2 Future Improvements

The heuristics can be turned into an algorithm that always produces the optimal solution using dynamic programming. So if we start to solve from the end node, potentially, we can always get the optimal solution by making the correct decision at every node. However, for the purpose of this paper, we want to focus on a simple heuristic that's easy to set up and solve.

5 Discussion

5.1 Integer Programming Approach

5.1.1 Limited Number of Chefs Scenario

We first start to solve the problem by assuming we have unlimited number of chefs in the kitchen. However, in real life scenarios, restaurants don't always have the money to hire so many chefs. Therefore, we try to find a way to increase total time needed for completing all dishes, so that we can reduce the number of chefs needed.

As a result, we find out that we only need 1 chef to finish all dishes in our previous example while increasing 4 minutes in total time. Such result can be used for future application. Restaurant owners can use this program to see how much sacrifice will we make in total completion time in order to reduce the number of chefs needed in the kitchen as a reference to their business decisions.

5.1.2 Extended Real-world Application: Rio City Cleaning

We extend our model to solve a real-world problem. Consider the post-Olympic period of this year's Rio Olympics. After the event, a huge amount of trash was left in stadiums to be cleaned. From statistics there was totally 488 tons of trash left at site during the event. The daily cleaning process is extremely costly so we want to minimize the total cleaning period when finishing up all the trash.

We formulate the problem in a similar way. We collect building capacities and areas of 6 main stadiums: Basketball gymnasium, natatorium, tennis court, football stadium and etc. The cleaning tasks of each stadium include (1) garbage collecting, (2) seat cleaning, (3) garbage sorting and (4) garbage recyling. Their priorities and designated process locations are assigned for each task. For example, the basketball gymnasium can contain $16,000$ audience and the golf court can contain $20,000$. The total number of workers is fixed and we also limit the total number of workers working indoors and outdoors, respectively.

From statistics we assume that there can be at most 1000 workers working in stadiums and 1000 workers in sewage factory. Our result shows that the entire cleaning process can be finished in 14 days. This is of course an "*lite*" version of the real-world problem. Changing the number of workers and increasing the number of stadiums will let us approximate an theoretically correct solution for this city cleaning problem.

5.1.3 General Advantages and Disadvantages

- Advantages: Our integer programming approach guarantees an optimal solution, and can give a clear and humanized cooking step instruction.

- Disadvantages:

  1. There are often multiple optimal solutions for the constraints given. For example, multiple solutions occur when jobs for certain dishes have no priority over one another so they can be done interchangeably. They can also occur when processing a "*slow*" job takes a long time (e.g. 30 minutes) while processing a "*quick*" job can be done fast (e.g 2 minutes). Thus the fast job can be arranged in parallel: either at the start, in between, or at the end of that slow job as long as all other constraints are satisfied. However, the Pulp package for Python that we use for solving our integer programming model only provides one solution.

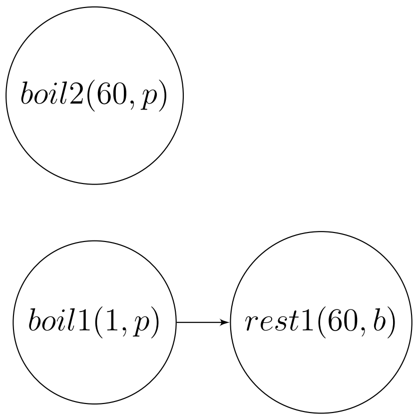
  2. Runtime: The running time is slow because of the large number of constraints assigned for the model. For example, the number of Machine Capacity Constraints is $m \times t$, which is quadratic; the number of Continuity Constraint is $\sum_{k=2}^{t} \binom{t}{k}$, which is exponential; the number of Priority Constraint is at most $j \times (j-1)/2$, which is about j^2 depending on the value j.

     Also, for the binary search process in our example (referring back to section 3.2), it takes about 4 minutes to complete. But for $t = 24$, which is the optimal solution, it only takes about 20 seconds.

5.2 Graph Approach

- Advantages: The advantage of this heuristics is that it's much easier to set up and much easier to solve comparing to the integer programming approach. The above example was solved by hand.

- Disadvantages: The disadvantage of this heuristics is that it does not always give the optimal solution. Most time as above, it gives a very good solution or sometimes the optimal solution itself. However, other times, it can perform really badly. The performance of this algorithm depends on the order we look at nodes when we add them to our working set. For example, consider the following simple example, assuming we only have one pan and one bowl,



When we are performing the first iteration, if we look at boil1 first, we would add boil1 into our working set first. Then when we look at boil2, we cannot add boil2 to our working set because there is only one p. If we follow this first iteration, then $T = 61$. However, if we look at boil2 first, and add boil2 to our working set, we would have to wait to perform boil1 , then rest1. In this second case $T = 121$, which is much worse than the optimal solution.

6 Conclusion

This paper presents two ways to solve the cooking scheduling problem with multiple dishes involved. We verified the accuracy of the estimated cooking time and the effectiveness of our algorithm through the simulation using three dishes, based on the assumption that we have unlimited numbers of chefs. We then try to reduce the number of chefs needed by increasing the total completion time. Such result can be used as a reference for restaurant decision making.

Future research may adopt a genetic algorithm to seek for equivalent optimal solutions given the solution we get from the Pulp solver. Moreover, we can apply our approach to other general industrial scheduling problems as we have shown in the Rio city cleaning example. For example, the industrial production line for automobile involves mainly 4 types of jobs: producing all components for the car, assembling components, assembling the car, and decorating the inside and outside of the car. We may consider different types of jobs in the similar way as different dishes. We also have different kinds of machines in the car factory which can be formulated in the same way as different kinds of machines in kitchen. We gather information about how many steps we have in each type of jobs, which machine and how much time each step needs. Finally we check the priority of each steps. With these information, we can apply this car production problem to our model and hence optimize the process.

7 Appendix

7.1 Python Code: Integer Programming Approach

```
from pulp import *
import sys
import time

def readcsv(filename):
    data = []
    for line in open(filename,'r').readlines():
        line = line.strip("\r\n")
        mylist = line.split(',')
        while '' in mylist:
            mylist.remove('')
        data.append(mylist)
    return data

def writeFile(filename, contents, mode="wt"):
    # wt = "write text"
    with open(filename, mode) as fout:
        fout.write(contents)

def addConstraint(j1,j2,prob, j,m,T,t,var):
    j1-=1
    j2-=1
    jt1=T[j1//j][j1%j]
    jt2=T[j2//j][j2%j]
    at1=[]
    at2=[]
    for col in range(m*t):
        at1=at1+[(var[j1*m*t+col],-(col%t+1)/float(jt1))]
        at2=at2+[(var[j2*m*t+col],(col%t+1)/float(jt2))]
    prob+= LpAffineExpression(at2+at1) >= (jt1+jt2)/2.0

def maxChef(i,j,m,t,var):
    cheflist = []
    for time in range(t):
        chef = 0
        for Machine in range(m):
            if (Machine ==3) or (Machine ==5) or (Machine ==6):
                cost = 0
            else:
                cost = 1
            for row in range(i*j):
                chef+=int(var[row*m*t+(Machine-1)*t+time].varValue)*cost
        cheflist.append(chef)
    if cheflist!=[]:
        return max(cheflist)
    else:
```

```
        return

def LP(i,j,m,T,c,assign,orderc, t, fo,Flag,ChefMax=0):
    #print(i,j,m)
    #initialize the problem
    prob=LpProblem("temp", LpMinimize)

    #dummy objective function
    prob+=t
    #initialize the variable name matrix,dim-i*j, m*t
    var_name=[]
    for dish_num in range(i):
        for job in range(j):
            tmp=[]
            for mach in range(m):
                tmp=tmp+['x%d%d%d%d'%(dish_num+1, job+1, mach+1, mt+1) for mt in range(t)]
            var_name=var_name+[tmp]
    #constraints for all 0-1 variables, dim-1, i*j*m*t
    var=[]
    for row in range(i*j):
        var = var + [LpVariable(var_name[row][col], lowBound=0, upBound=1, cat='Integer') \
        for col in range(m*t) ]

    #machine capacity constraint
    for col in range(m*t):
        prob+= lpSum([var[row*m*t+col]] for row in range(i*j)) <= c[col//t]

    M=10000
    #continuous contraint
    for row in range(i*j):
        dt= T[row//j][row%j]
        if dt>1:
            for mach in range(m):
                for length in range(dt, t):
                    for st in range(t-length):
                        sp=row*m*t+mach*t+st
                        ep=sp+length
                        prob+= var[ep]*(ep)-var[sp]*(sp)+1 <= dt+M*(1-var[ep])+M*(1-var[sp])
                        prob+= var[ep]*(ep)-var[sp]*(sp)+1 >= -dt-M*(1-var[ep])-M*(1-var[sp])

    #each job is assigned to a specific machine, according to
    for row in range(i*j):
        pos= assign[row//j][row%j]
        if pos>0:
            prob+=lpSum(var[row*m*t+(pos-1)*t+col] for col in range(t)) == T[row//j][row%j]

    #orderc
    for row in range(i*j):
        for col in range(row+1,i*j):
            if orderc[row][col]>0:
                addConstraint(row+1,col+1,prob, j,m,T,t,var)
```

```

    #each job gets finished once
    for row in range(i*j):
        prob+= lpSum([var[row*m*t+col] for col in range(m*t)]) == T[row//j][row%j]

    if Flag:
        for time in range(t):
            chef = 0
            varlist = []
            for Machine in range(m):
                if (Machine ==3) or (Machine ==5) or (Machine ==6):
                    cost = 0
                else:
                    cost = 1
                # print("cost is ",cost)
                for row in range(i*j):
                    varlist+=[cost*var[row*m*t+(Machine-1)*t+time]]
            # print("chef num is ....:",chef)
            # print("chef constraint",lpSum(varlist))
            prob+= lpSum(varlist) < ChefMax

    #prob.writeLP("broke.lp")

    prob.solve()

    # Print the status of the solved LP
    # print("Status:", LpStatus[prob.status])
    #writeFile(fo, LpStatus[prob.status]+'\n')

    output=""
    for row in range(i*j):
        for col in range(m*t):
            currVal = var[row*m*t+col].varValue
            #print type(currVal), type(currValStr)
            output=output+str(int(currVal))+'\t'
        #output+='\n'
    #writeFile(fo, output, α')
    state = LpStatus[prob.status]
    return state, output, maxChef(i,j,m,t,var)

def LPsolver(fn, fs):
    simpleExData = readcsv(fn)

    nrow = len(simpleExData)
    if nrow == 0:
        #print "Workstation Setup And Recipe are Empty!"
        sys.exit("Workstation Setup And Recipe are Empty!")
    elif nrow == 1:
        #print "Recipe Is Empty and WorkStation Setup Is inimcomplete!"
        sys.exit("Recipe Is Empty and WorkStation Setup Is inimcomplete!")
    elif nrow == 2:
```

```
        #print "Recipe Is Empty!"
        sys.exit("Recipe Is Empty!")
    elif not(nrow%3 == 0):
        #print "False Recipe!"
        sys.exit("False Recipe!")
    else:
        pass


    stepDic = {}
    for line in open(fs, "r"):
        line = line.strip("\r\n")
        if line == "fry":
            stepDic["fry"] = 1
        elif line == "wash":
            stepDic["wash"] = 2
        elif line == "defrost":
            stepDic["defrost"] = 3
        elif line =="season":
            stepDic["season"] = 7
        elif line =="stew":
            stepDic["stew"] = 5
        elif line =="decorate":
            stepDic["decorate"] = 4
        elif line =="bake":
            stepDic["bake"] = 6
        elif line =="chop":
            stepDic["chop"] = 4
        elif line =="mix":
            stepDic["mix"] = 7
        elif line == "boil":
            stepDic["boil"]=5
        elif line == "rest":
            stepDic["rest"]= 7

    #number of machines
    machine=len(simpleExData[0])
    m = machine

    kitchenDic = {}
    for i in range(machine):
        kitchenDic[i+1]=simpleExData[0][i]
    # print(kitchenDic)

    #number of dish
    dish = len(simpleExData)//3-1
    i = dish
    # print("dish:",i)

    #number of steps for each dish
    jobs = 0
```

```
    for jn in range(dish):
        if len(simpleExData[jn*3+3])>jobs:
            jobs = len(simpleExData[jn*3+3])


    #time for each job
    Time=[]
    for jm in range(dish):
        timelist = []
        for ti in range(jobs):
            if ti>=len(simpleExData[jm*3+4]):
                timelist.append(0)
            else:
                timelist.append(int(simpleExData[jm*3+4][ti]))
        Time.append(timelist)
    # print("time for each job",Time)
    T = Time

    #machine capacity
    capacity = []
    for index in range(len(simpleExData[0])):
        capacity.append(int(simpleExData[1][index]))
    # print("machine capacity",capacity)
    c = capacity

    #
    assign = []
    for a in range(dish):
        asslist = []
        for b in range(jobs):
            if b>=len(simpleExData[a*3+3]):
                asslist.append(0)
            else:
                asslist.append(stepDic[simpleExData[a*3+3][b]])
        assign.append(asslist)
    # print(assign)

    order = []
    for jm in range(dish):
        rowOrder = []
        for ti in range(jobs):
            if ti>=len(simpleExData[jm*3+5]):
                rowOrder.append(0)
            else:
                rowOrder.append(int(simpleExData[jm*3+5][ti]))
        order.append(rowOrder)
    #order matrix, dim-ij*ij
    orderMatrix = []
    for j in  range(len(order)):
        for x in range(len(order[0])):
            comp = order[j][x]
```

```
            xthrow = []
            for k in  range(len(order)):
                for y in range(len(order[0])):
                    if j==k:

                        if order[k][y]==0:
                            xthrow.append(0)
                        elif order[j][x] ==0:
                            xthrow.append(0)
                        elif comp<order[k][y]:
                            xthrow.append(1)
                        else:
                            xthrow.append(0)
                    else:
                        xthrow.append(0)
            orderMatrix.append(xthrow)
    orderc = orderMatrix
    j=jobs
    # print("number of jobs:",j)
    return (i,j,m,T,c,assign,orderc,simpleExData,kitchenDic)

def outputG(simpleExData, m, t, jobs, output,kitchenDic,fo):
    ##output guidance
    nrow=len(simpleExData)
    for i in range(3, nrow, 3):
        new="\nGuidance for dish "+str(i//3)+" is the following: "
        writeFile(fo,new+'\n','α')
        #print("\nGuidance for dish", i//3, "is the following: ")
        jobLen = len(simpleExData[i])
        jobList = simpleExData[i]
        index = i//3
        # print index
        for j in range(jobLen):
            for k in range(0, m):
                # print "this is j:", j
                # print "this is k:", k
                linenum = (index-1)*jobs + j
                linelen = 2*m*t
                # print "this is linenum:", linenum
                # print "this is linelen:", linelen
                id1 = linenum*linelen
                id2 = (linenum+1)*linelen
                # print "this is id1, id2:", id1, id2
                suboutput = output[id1:id2]
                templist = []
                templist1 = []
                for mi in range(m):
                    id3 = mi*t*2
                    id4 = (mi+1)*t*2
                    subjob = suboutput[id3:id4]
                    templist2 = []
```

```
                    tempstr = subjob.replace("\t", "")
                    for istr in tempstr:
                        templist2.append(int(istr))

                    templist1.append(sum(templist2))
                    templist2_rev = templist2[::-1]

                    for i in templist2:
                        if i == 1:
                            templist.append(templist2.index(i))
                            break
                    for i in templist2_rev:
                        if i == 1:
                            templist.append((len(templist2) - (templist2_rev.index(i))))
                            break

                    #templist.append(sum(templist2))
                #print templist
                if templist1[k] == 0:
                    continue
                else:
                    new= "→"+str.capitalize(str(jobList[j]))+' on\t'+\
                    kitchenDic[k+1]+" from time "+str(templist[0])+\
                    " to "+str(templist[1])
                    writeFile(fo,new+'\n','α')
                    #print("→", str.capitalize(str(jobList[j])), "on", kitchenDic[k+1], "from
(i,j,m,T,c,assign,orderc,simple,kitchenDic)=LPsolver('final.csv', 'step.txt')
maxt=0
for dish in range(len(T)):
    maxt=maxt+sum(T[dish])

def binarys(fn,fo,fs,lower, upper):
    flag = False
    (i,j,m,T,c,assign,orderc,simple,kitchenDic)=LPsolver(fn,fs)
    if lower>=(upper-1):
        finalStat,outputFinal,finalChef=LP(i,j,m,T,c,assign,orderc,upper,fo,flag)
        writeFile(fo,'Status: Optimal\n')
        outputG(simple,m,t,j,outputFinal,kitchenDic,fo)
        return (upper,finalChef)
    statUpper,outputUpper,upperChef=LP(i,j,m,T,c,assign,orderc,upper,fo,flag)
    statLower,outputLower,lowerChef=LP(i,j,m,T,c,assign,orderc,lower,fo,flag)
    print(statUpper,upper)

    mid=int(round(lower+(upper-lower)/2.0))
    statMidd,outputMidd,midChef = LP(i,j,m,T,c,assign,orderc,mid,fo,flag)
    if statMidd=="Optimal":
        return binarys(fn,fo,fs,lower, mid)
    else:
        return binarys(fn,fo,fs,mid, upper)

def linears(fn,fo,fs,lower, upper,ChefMax):
```

```
    (i,j,m,T,c,assign,orderc,simple,kitchenDic)=LPsolver(fn,fs)
    flag = True
    minimal = lower
    while minimal<=upper:
        print("minimal is :", minimal)
        status,output,chef=LP(i,j,m,T,c,assign,orderc,minimal,fo,flag,ChefMax)
        if status=="Optimal":
            return minimal
        else:
            minimal+=1
    return lower


minT = 0
start=time.time()
(minT,chefNum)=binarys('final.csv', 'result.txt',"step.txt",0,24)
finalTime =  linears('final.csv', 'result.txt',"step.txt",minT,maxt,chefNum)
print("We need at least %d chefs.” %chefNum)

end=time.time()
print("Number of Chef:",chefNum)
print("Finished calculation in %d seconds.” %(end-start))
print("\n\nAll dishes are prepared and served in", minT, "minutes")
```