Programmation Dynamique - Project Partie 2

Ismael Martinez

February 2019

The following two sections have been lifted from Partie 1 of this project with minor changes.

1 Problem Description

The *container loading problem* is a sub-problem of [2] where the goal there was to find the optimal way of arranging containers onto the rail cars.

The problem we are attempting to solve is that of the *loading* of the containers from the stacks onto the rail cars. We are given the positions of all the containers in the stacks and the target position of all the containers on the rail cars after optimization of the problem in [2]. An example of our starting and ending container configurations is shown in Figure 1.

The largest cost associated with moving a container from the stacks to the rail cars is the cost of lifting the container; therefore, we ignore the distance travelled to move the container and say that a single *touch* of any container has a cost of 1.

Referring to Figure 1, grabbing container A from the stacks requires a single touch; however, grabbing container E requires a *double touch* meaning we have to first move container D before touching container E. We assume that there is ample space on the ground so we can place the first container on the ground in the event of a double touch. Since we are touching two containers, this move has a cost of 2.

The purpose of this project is to take the mathematical model from the previous part, and build a code that will give you the solution. Two codes were built, one exact solution and one heuristic.

1.1 Assumptions

All assumptions are as follows:

1. There is ample space on the ground to place intermediary containers.
2. Triple touch is not permitted. In fact, it can be shown that they are never optimal.
3. The maximum number of containers we can stack on the rail cars is 2.
4. The maximum number of containers we can stack in a lot is 3.
5. Only *valid* containers can be moved at a given step. More on this in Section 2.2.
6. The train has at most one rail car slot with a single container. In other words, every slot of the train is double stacked with at most one container on its own.

The physical layout of the containers in the stacks are separated into *lots* being groups of stacks; that being said, our model will only look at a container's *depth* - more on this in Section 2.1. The physical layout of the target is a set of rail cars which are separated into *slots*, each with at most one stack of containers. Both of these layouts in shown in Figure 1.

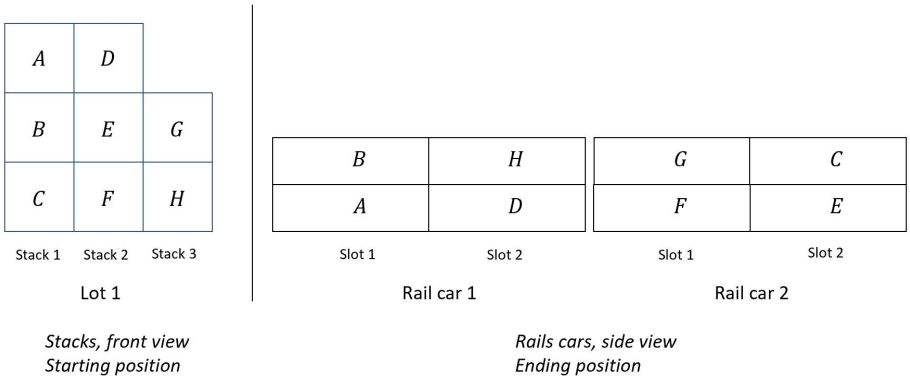

Figure 1: Our objective is to go from the stacks on the right to the rail cars on the left while moving as few containers as possible.

2 Modelization

The set of all containers is denoted C. A container $c \in C$ in the system lives in either the stacks or on the rail cars at any given time.

We define S as the set of slots with containers on the rail cars, and R as the target container configuration of the rail cars. Since S is the set of slots in R, $|S|$ is the number of slots in R. From Assumption 6, we can gather that $|S| = \lceil \frac{N}{2} \rceil$.

We define a function $t(c, R)$ to describe whether a container $c \in C$ in the target R is *top* or *bottom*.

$$t(c, R) = \begin{cases} 1 & \text{if } c \text{ is a top container in } R \\ 0 & \text{otherwise} \end{cases}$$

We also define the function $p(c, R)$ to describe in which slot $s \in S$ a container $c \in C$ belongs in R.

$$p(c, R) = s, \qquad s \in S$$

2.1 i - States

Let us denote

- $\boldsymbol{Z_k}$ as the set of containers in the stacks at step k.
- $\boldsymbol{Y_k}$ as the set of containers in the rail cars at step k.
- $\boldsymbol{x_k} = (Z_k, Y_k)$ as the state of the system at step k.
- $\boldsymbol{X_k}$ as the set of all possible states x_k at step k.

Each container in the system will either be a member of Z_k or Y_k at step k; that is, $C = Z_k \cup Y_k$ and $Z_k \cap Y_k = \varnothing$. As a result, x_k is a description of every container of the system at step k, and finding all possible states of the stack containers is equivalent to finding all possible states of the rail cars - this is important for Section 2.5.

We define a function $d(z, x_k)$ as the *depth* of a container $z \in Z_k$ in state x_k, which represents the depth of a container from the top of the stack. The values of this function for different containers are shown in Figure 2. This function's domain is the set of containers Z_k such that $d : Z_k \mapsto \{0, 1, 2\}$, and can perhaps be better thought of as the number of containers *above* our container $z \in Z_k$. Since we cap the height of the stacks at 3 by Assumption 4, the function's maximum value is 2.

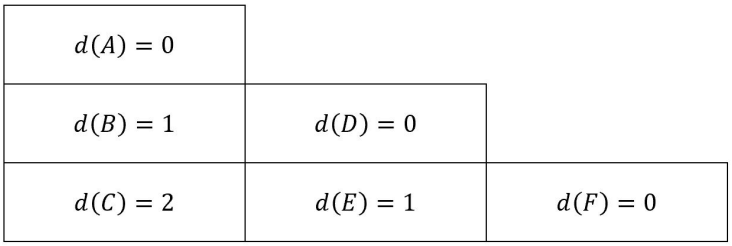

Figure 2: The depth function $d()$ for different containers in the stack with containers in an arbitrary set Z.

We define a function $h(s, R, Y_k)$ as the container height of slot $s \in S$ for configuration R given rail car container set Y_k. From Assumption 3, we can see that the output of $h()$ is in $\{0, 1, 2\}$.

Finally, we define a function $v(z, x_k, R)$ as the *validity* of a container $z \in Z_k$ in state x_k, which represents whether a container can be immediately moved to it's proper spot in R as shown in Figure 3. This function's domain is the set of containers in Z_k that we wish to move to Y_k such that $v : Z_k \mapsto \{0, 1\}$.

$$v(z, x_k, R) = \begin{cases} 1 & \text{if } t(z, R) = h(p(z, R), R, Y_k) \\ 0 & \text{otherwise} \end{cases}$$

For the above equation, we can see this is true since if the height of a slot is 0 and the container z belongs on the bottom, then it is valid. Similarly, if the height of the slot is 1 and the container belongs on the top, it is also valid.

Note, for each container $c \in C$, either $c \in Z_k$ or $c \in Y_k$ meaning we can derive Y_k from Z_k as $Y_k = C \setminus Z_k$.

2.2 ii - Actions

Let us denote u_k as the action of choosing and moving a container from the stacks to the rail cars at step k. For simplicity, we will also denote the choice of container for movement as u_k. Grabbing a single container for movement yields a cost of 1; since a container $z \in Z_k$ with $d(z, x_k) > 0$ requires all the above containers to be moved first, the cost of moving a choice u_k is $d(u_k, x_k) + 1$.

If we have to move an intermediary container z before we can move u_k, we assume we have ample space to move z on the ground, thus making $d(z, x_{k+1}) = 0$ after moving. Furthermore, for each container z' *below* our choice u_k, we decrement it's depth $d(z', x_{k+1}) \leftarrow d(z', x_k) - 1$.These two consequences are also apart of the action u_k.

Our choice of u_k is restricted to *valid* containers to move, with $v(u_k, x_k) = 1$. We also restrict our choice u_k to containers where $d(u_k, x_k) < 2$, meaning we only consider choices u_k with at most 1 container stacked on top of it.

The set of *all* possible choices containers to move from the lots is $U_k(x_k)$, and the set of containers per our restrictions is $U'_k(x_k)$; therefore, our available actions become $u_k \in U'_k(x_k) \subseteq U_k(x_k)$.

$$U'_k(x_k) = \{z \in U_k(x_k) \mid v(z, x_k, R) = 1 \text{ and } d(z, x_k) \leq 1\}$$

We define a function $u_k = \mu_k(x_k)$ as a mapping $X_k \mapsto U'_k$, or the choice u_k we make given $x_k \in X_k$. We denote the optimal choice of u_k given any x_k as $\mu^*_k(x_k)$, or simply as μ^*_k.

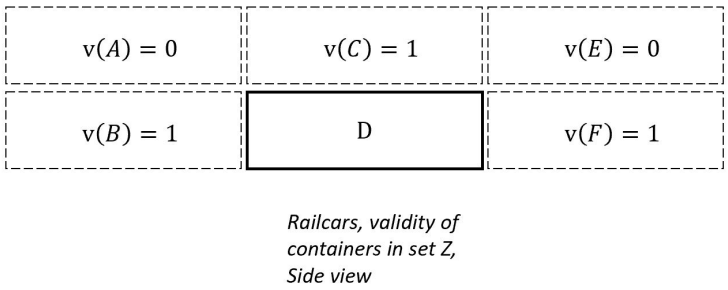

Figure 3: The *valid* choices of u for an arbitrary Y are containers B, C and F, where D is the only container currently on the rail cars (in Y) and both A and E are not currently able to be placed.

2.3 iii - Transition Function

From the previous two sections, we can see that given a state x_k any choice of u_k yields a unique state x_{k+1} of our system. Therefore, our transition function is as follows:

$$\begin{aligned} x_{k+1} &= f(x_k, u_k) \\ &= (Z_k \setminus u_k, Y_k \cup u_k) \end{aligned}$$

where $\setminus$ is the *set difference* operator. by the two consequences of u_k described in Section 2.2, a choice of u_k given x_k results in a unique state x_{k+1}.

2.4 iv - Forward Relation

We define a *policy* π as a set of μ_k for $k = 0, \ldots, N-1$, therefore $\pi = \{\mu_0, \mu_1, \ldots, \mu_{N-1}\}$. We define an *optimal* policy as $\pi^* = \{\mu_0^*, \mu_1^*, \ldots, \mu_{N-1}^*\}$.

As stated in the section above, the *cost* of moving our choice of container $u_k \in U_k'$ from state x_k is $d(u_k, x_k)+1$, therefore we can write the cost of u_k on x_k as

$$g_k(x_k, u_k) = d(u_k, x_k) + 1.$$

We define $D_{\pi,k}(x_k)$ as the cost of starting at x_0 and following the policy π from 0 to k. For an optimal policy π^*, we choose to write $D_{\pi^*,k}$ simply as D_k.

For N total containers, we define our recurrence relations as follows:

$$\begin{aligned} D_0 &= g_0(x_0) = 0 \\ D_k(f(x_{k-1}, u_{k-1})) &= \min_{u_{k-1} \in U_{k-1}'} \{g_{k-1}(x_{k-1}, u_{k-1}) + D_{k-1}(x_{k-1})\} \\ \mu_k^*(f(x_{k-1}, u_{k-1})) &= \arg\min_{u_{k-1} \in U_{k-1}'} \{g_{k-1}(x_{k-1}, u_{k-1}) + D_{k-1}(x_{k-1}))\} \\ &\forall x_{k-1} \in X_{k-1}, \quad k = 1, \ldots, N \end{aligned}$$

Our optimal solution is

$$D_N(x_N) = D(x)$$

where the first state x_0 is fixed with Z_0 containing all containers and Y_0 being empty.

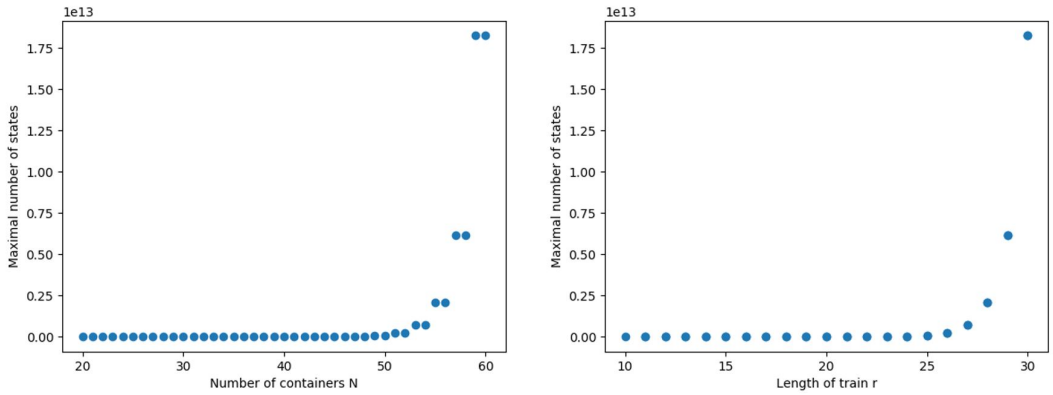

Figure 4: The maximal value X_k given $20 \leq N \leq 60$ and $|S| = \lceil \frac{N}{2} \rceil$.

2.5 v - Size of X_k and U_k

For X_0, the size is 1 since we are starting with all containers in the stack. Since maximal size of $|U_0|$ is $|S|$, we get $|X_1| \leq |S|$. For X_k, we are considering the number of ways to place k containers onto the rail cars without violating the target configuration R. For a length $|S|$ and step k, we find the number of ways to place b containers on the bottom of the rail cars, multiplied by the remaining containers on top of those bottom containers. In order to stack, we must have the number of bottom containers be bigger than the number of top containers. The state size becomes

$$|X_k| \leq \sum_{b=\lceil \frac{k}{2} \rceil}^{k} \left\{ \binom{|S|}{b} \cdot \binom{b}{k-b} \right\} \qquad \text{for } k = 0, 1, ..., N$$

After scripting the state size (Section **??**), this maximal value occurs at $k = |S|$. By calculating and plotting the maximum values of these equations, we can see these values grow exponentially with $|S| = \lceil \frac{N}{2} \rceil$ as shown in Figure 4 which shows the value space of our three instances.

In particular for our 3 instances, we get the following maximal sizes of $|X_k|$.

N	$k = \lvert S \rvert$	$\max \lvert X_k \rvert$
20	10	8 953
30	15	1 787 607
60	30	18 252 025 766 941

When we consider the size of the set of actions $|U_k|$, recall that we are only considering choices where $d(u_k, x_k) < 2$ and $v(u_k, x_k) = 1$. For U_0, there are at most $|S|$ choices here since we can only choose from the bottom containers of the rail cars. On the second round, for each remaining container $u_1 \in U_1$ there exists a state $x_1 \in X_1$ where u_1 is valid. Therefore,

$$|U_k| \leq N - k \qquad k > 0$$

3 Results and Analysis

I programmed my code in Python, using Pandas and Numpy libraries for data processing, and AnyTree library for building my solution.

Case 1:	Step $k-1$ has a single output node. There exists at least 1 container c with $d(c) = 0$. Move this container.
Case 2:	Step $k-1$ has a single output node. All valid containers c have $d(c) = 1$. Build a node for each possible choice u_k.
Case 3:	Step $k-1$ has multiple output nodes. At least one valid container c has $d(c) = 0$. Move this container. **Trim the graph of all other branches.**
Case 4:	Step $k-1$ has multiple output nodes. All valid containers c have $d(c) = 1$. Build a node for each possible choice u_k.

Table 1: Cases 1 - 4 of the improved algorithm.

3.1 Exact Solution

The problem as presented was quite easy for an algorithm. Once the validation and depth restrictions from Section 2.2 are enforced, the problem becomes much smaller. Furthermore, we notice two things:

1. If there exists a valid container with $d(c) = 0$ – a container on top of the stack – it is always an optimal choice to move it. In other words, there is no downside to moving such a container.

2. If all valid containers have $d(c) = 1$ – a container with one other container on top of it – then moving one of these containers will yield a valid container with $d(c) = 0$ on the next step.

This means that we can alter the Forward Chain algorithm slightly into four cases for step k, and yields a change from Algorithm 1 to Algorithm 2. Recall that we have defined the set of valid moves U'_k in Section 2.2 and the Transition Function $f(x_k, u_k)$ in Section 2.3

Algorithm 1 Basic forward-chain Dynamic Programming Algorithm

procedure ForwardChain
  $R \leftarrow$ Position of each container in railcar target
  $C \leftarrow$ Set of all containers
  $Z_0 \leftarrow C$ ▷ The set of containers in the stacks
  $Y_0 \leftarrow \varnothing$ ▷ The set of containers in the railcars
  $x_0 \leftarrow (Z_0, Y_0)$
  $D_0 \leftarrow 0$

  for k=1 to N **do**
    for all $x_{k-1} \in X_{k-1}$ **do**
      $U'_{k-1}(x_{k-1}) \leftarrow \{u_{k-1} \in U_{k-1}(x_{k-1}) \mid v(z, x_{k-1}, R) = 1 \text{ and } d(z, x_{k-1}) \leq 1\}$
      ▷ Set of valid nodes

      $D_k(f(x_{k-1}, u_{k-1})) \leftarrow \min_{u_{k-1} \in U'_{k-1}} \{g_{k-1}(x_{k-1}, u_{k-1}) + D_{k-1}(x_{k-1})\}$
      $\mu^*_k(f(x_{k-1}, u_{k-1})) \leftarrow \arg\min_{u_{k-1} \in U'_{k-1}} \{g_{k-1}(x_{k-1}, u_{k-1}) + D_{k-1}(x_{k-1})\}$

Algorithm 2 Updated forward-chain Dynamic Programming Algorithm

procedure FORWARDCHAIN

$R \leftarrow$ Position of each container in railcar target
$C \leftarrow$ Set of all containers
$Z_0 \leftarrow C$ ▷ The set of containers in the stacks
$Y_0 \leftarrow \varnothing$ ▷ The set of containers in the railcars
$x_0 \leftarrow (Z_0, Y_0)$
$D_0 \leftarrow 0$

for k=1 to N **do**
for all $x_{k-1} \in X_{k-1}$ **do**
$U'_{k-1}(x_{k-1}) \leftarrow \{u_{k-1} \in U_{k-1}(x_{k-1}) \mid v(z, x_{k-1}, R) = 1 \text{ and } d(z, x_{k-1}) \leq 1\}$
▷ Set of valid nodes

if $\exists c \in C \mid d(c) = 0$ and $|X_{k-1}| = 1$ **then**
$u_{k-1} \leftarrow c$ ▷ We only build one node at step k using any such c
break
else if $\exists c \in C \mid d(c) = 0$ and $|X_{k-1}| > 1$ **then**
$u_{k-1} \leftarrow c$
Cut all branches containing $x'_{k-1} \in X_{k-1}$, $x'_{k-1} \neq x_{k-1}$ ▷ This ensures we no longer look at other branches during computation
break
else
$D_k(f(x_{k-1}, u_{k-1})) \leftarrow \min_{u_{k-1} \in U'_{k-1}} \{g_{k-1}(x_{k-1}, u_{k-1}) + D_{k-1}(x_{k-1})\}$
$\mu^*_k(f(x_{k-1}, u_{k-1})) \leftarrow \arg\min_{u_{k-1} \in U'_{k-1}} \{g_{k-1}(x_{k-1}, u_{k-1}) + D_{k-1}(x_{k-1})\}$

In addition to Algorithm 1 and 2, the different cases used in the code are displayed in Table 1. Cases 2 and 4 are your traditional forward chain moves where we build on every possible option, and Case 3 is essentially taking the minimum of those new branches. Based on our previous two conclusions of this section, we can safely cut the other branches and reduce our problem significantly. Indeed, taking the lowest cost of the second set of nodes is equivalent to taking the lowest overall cost since all nodes at step k have cost 2, and nodes at $k+1$ have either cost 2 or 1. Case 1 is where the majority of our speed improvements come from; since grabbing a valid container with cost 1 has no negative effect on the system, it is always optimal to take such a container.

To verify the validity of my algorithm, I started with small examples which passed with my algorithm. Then, I focused on the double-touches in the solutions and verified that they were indeed necessary. Indeed, by first visually analysing the sets and then by modifying the data and re-running the scripts, I am certain that those double-touches could not be avoided.

The code output, optimal solutions, optimal costs, and time are given in Figures 5, 6, and 7. The output format for these Figures is in Table 2.

3.2 Heuristic Solution

For the heuristic solution, I employed a Greedy Algorithm approach. At each step k, we choose the valid container with the smallest cost and choose it as our u_k. Given the above four cases, we are only focusing on Case 1 and 2 since we will only ever be choosing a single u_k to build per step. If a cost of 1 exists, it is Case 1; if all containers have cost 2, we choose a random container as our u_k. The solutions are given in Figures 8, 9, and 10.

containerID of c	**cost** associated with moving c
c_1, 1^{st} container to move	g_1 = cost of moving c_1
c_2, 2^{nd} container to move	g_2 = cost of moving c_2
$\vdots$	$\vdots$
c_N, N^{th} container to move	g_N = cost of moving c_N
Optimal cost: $\sum_{i=1}^{N} g_i = J$	
Time to run script, in seconds: t	

Table 2: Format of the solutions in Figures 5, 6, 7, 8, 9, 10. The optimal solution is to move the containers in the given order for N containers.

```
The containers in order are as follows.

containerID      cost
BKMU_400293      1
BKMU_938598      1
BKMU_204624      1
BKMU_293977      1
BKMU_967034      1
BKMU_967249      1
TGHU_307621      1
WCKU_709029      2
WCKU_845523      1
WCKU_863156      1
FKCU_816675      2
FFLU_843666      1
FFLU_777328      1
CGMU_651777      2
WCKU_857500      1
NFNU_696988      1
CAMU_826316      2
CAMU_922734      1
ACLU_975060      1
APRU_572834      1
Optimal cost: 24

------------------------------------------------
Time taken to complete in seconds:
1.8426942825317383
```

Figure 5: Exact solution for the Small instance.

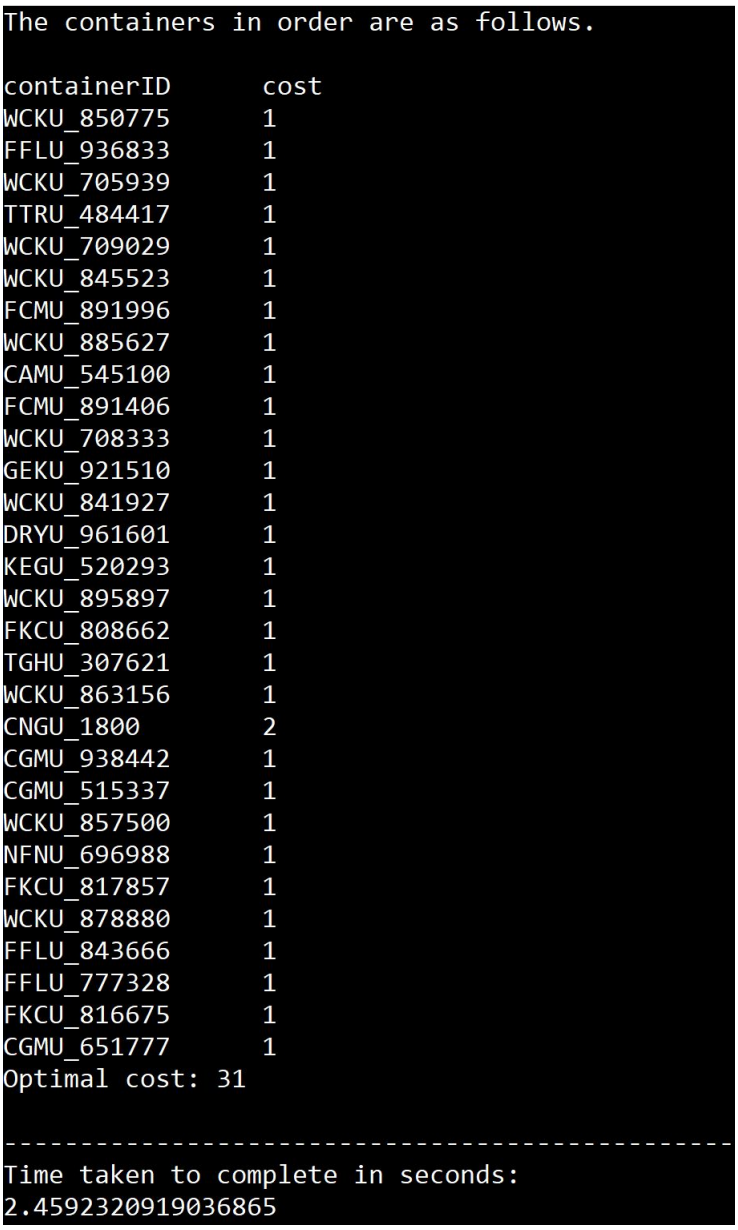

Figure 6: Exact solution for the medium instance.

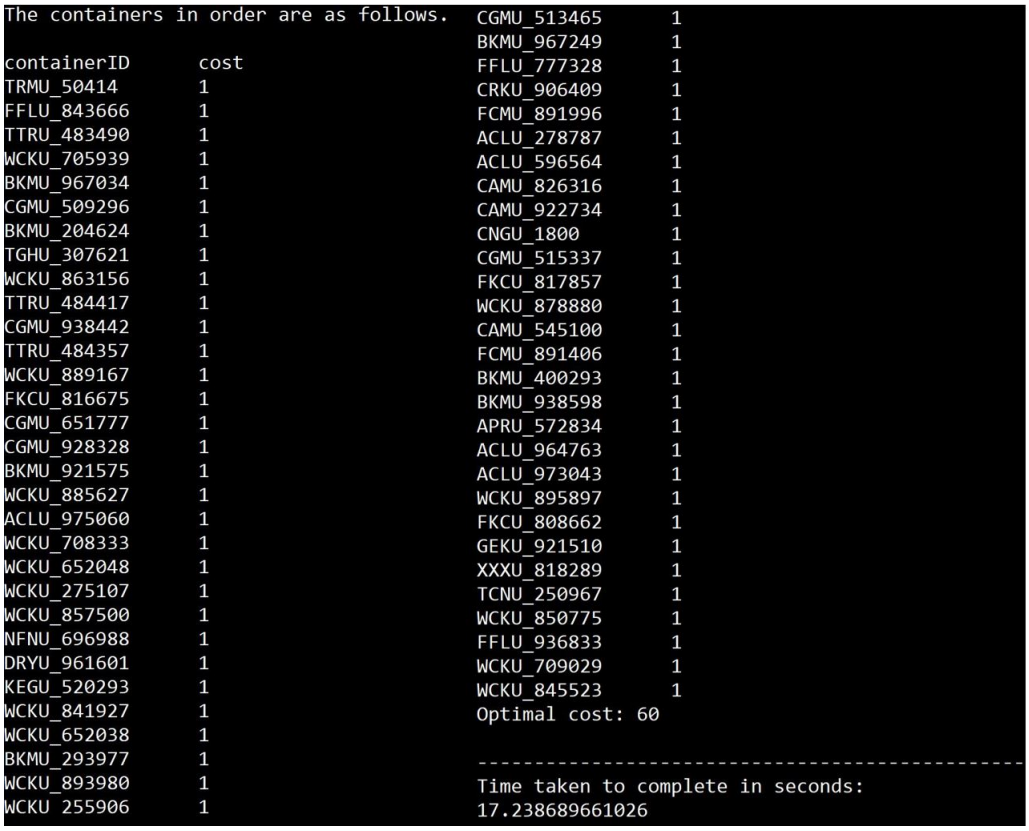

Figure 7: Exact solution for the large instance.

```
The containers in order are as follows.

containerID      cost
BKMU_400293      1
BKMU_938598      1
BKMU_204624      1
BKMU_293977      1
BKMU_967034      1
BKMU_967249      1
TGHU_307621      1
WCKU_709029      2
WCKU_845523      1
WCKU_863156      1
FKCU_816675      2
FFLU_843666      1
FFLU_777328      1
CGMU_651777      2
WCKU_857500      1
NFNU_696988      1
CAMU_826316      2
CAMU_922734      1
ACLU_975060      1
APRU_572834      1
Optimal cost: 24

------------------------------------------------
Time taken to complete in seconds:
1.739820957183838
```

Figure 8: Heuristic solution for the Small instance.

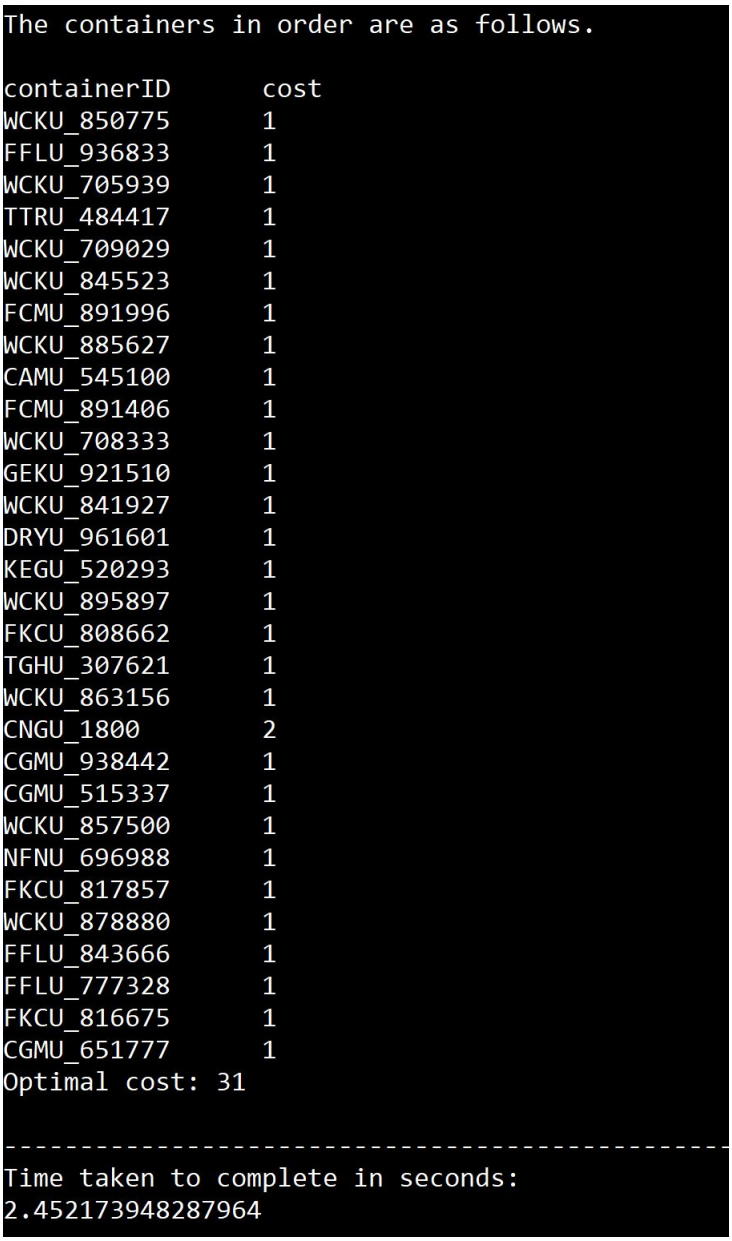

Figure 9: Heuristic solution for the medium instance.

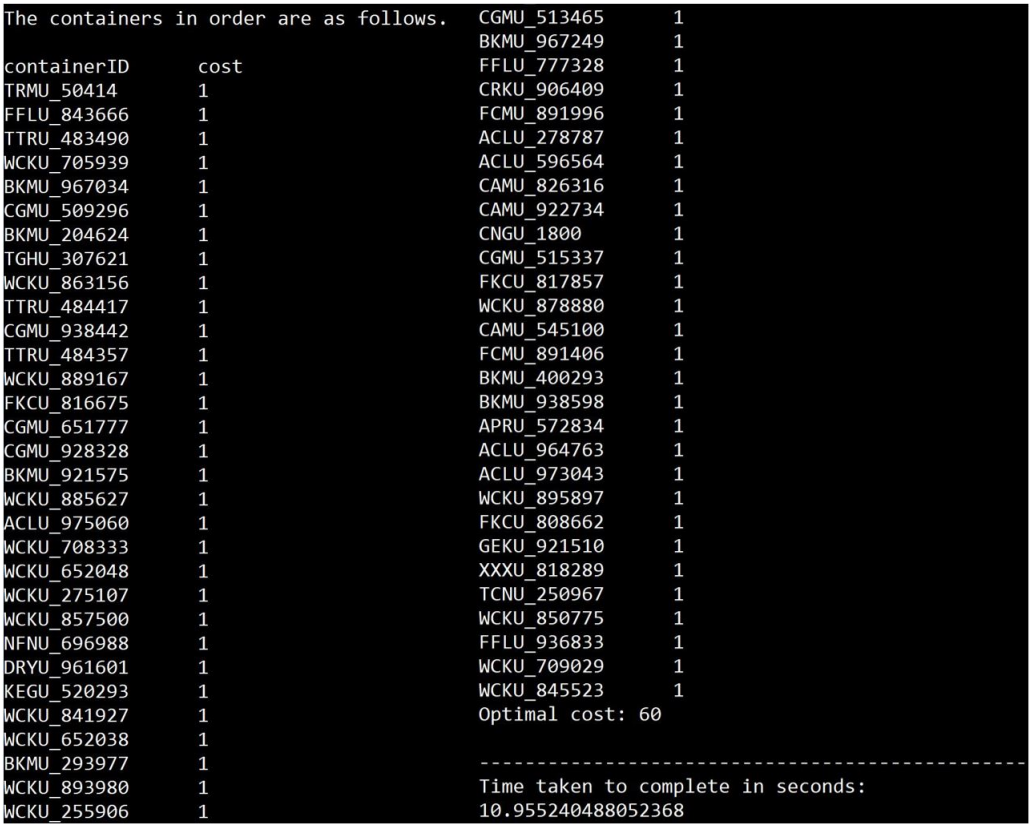

Figure 10: Heuristic solution for the large instance.

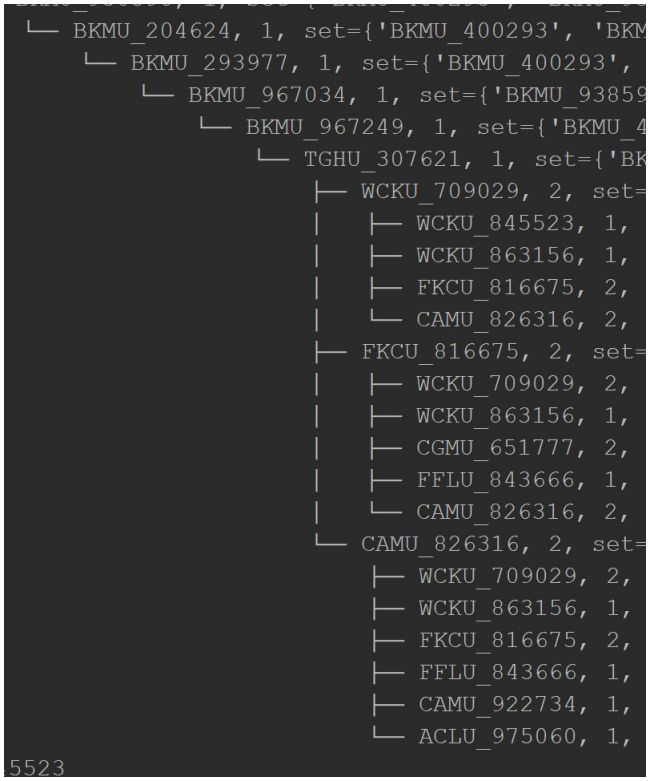

Figure 11: After seeing a cost of 2, we build one extra step and choose the lowest overall cost.

ProblemInstance	Exact (time in sec.)	Exact Solution	Heuristic (time in sec.)	Heursitic Solution
Small	1.8427	24	1.7398	24
Medium	2.4592	31	2.4522	31
Large	17.2387	60	10.9552	60

Table 3: Exact and Heuristic solution comparisons.

3.3 Exact vs Heuristic Solutions

The solution times for the different instances are as follows:

In addition to a different algorithm for the exact and the heuristic, there are also fewer variables in the heuristic. For the exact algorithm, the code kept track of the container states for the stacks and the railcars for each state, meaning a split in nodes could be costly; for the heuristic we don't need this since we are never splitting. We are choosing a single u_k per round.

If we look at the solutions, we can see that the optimal solution is not very far off a greedy solution. For the medium and large instances it is easy to believe both scripts came up with the same solution, but for the small it is merely a matter of coincidence. Whenever there is a cost of 2, the greedy algorithm chooses a random node to continue building off, and it was coincidental that this was the optimal path. If we look at Figure 11, we can see why this is. The nodes `WCKU_709029`, `FKCU_816675` and `CAMU_826316` all have a cost of 2, so we build one further step $k+1$; regardless of which node we choose to build a branch off, they all possess at least one subsequent node with a cost of 1. In all three cases, we reach an optimal solution.

Though the solution time seems similar for the Small and Medium instances, we see the difference in the Large instance based on the numbers in Table 3.

The hardest part of this problem was determining the best mathematical model and the *valid* choices of u_k. Once this was determined, everything else followed and the coding became quite straightforward. Without the restrictions of validity and depth on u_k, this problem could become very large and expensive to solve.

4 Work process

Once my mathematical model for Part 1 was established, I no longer needed to reference any external texts outside of Python function documentation online. The Greedy Algorithm approach is actually an approach I recall from my undergrad that I felt may act as a good heuristic for this project.

I spoke briefly with Emmanuel Eytan, Jean-Christophe Taillandier about coding ideas and the best approach to take. JC let me know about the AnyTree library which I eventually used to structure my solution.

5 Auto-Evaluation

5.1 Mathematical model

The variables are intuitively named based on variables used in class or by their purpose, and the descriptions are helpful. The only thing I would expand on is the target set R and what "proper placement" or "correct position" mean mathematically. Aside, I believe that the model, its variables and functions are well defined and well presented.

5.2 Motivations and Assumptions

The motivation and assumptions were built around not over-complicating the problem. In other words, I sought to reduce the problem to its most essential components and nothing else. Similarly, the assumptions made were to render the algorithm much easier and faster, without losing any key pieces of information or data.

5.3 Heuristic Relevance

For the heuristic, I chose to do a Greedy Algorithm because it worked well with the structure of the exact algorithm, being that we were building the graph as we were visiting nodes. This approach did not need for the graph to be computed prior to solving. As noted in Section 3 and Figure 11, a Greedy Algorithm will actually result in an optimal solution for this particular problem and given our data.

5.4 Quality of Results

Looking at Figures 5 – 10, the solutions are clearly presented with the order of containers, their costs, the optimal solution, and the time of execution. The time of execution is far less than I had expected, even for the Large problem. I had worked through the solutions of the Small problem by hand to ensure that the double-touches were in fact necessary, and I have no doubt the optimal solutions are presented accurately. The medium and large optimal costs surprised me, but on further look it makes sense since a larger container count means larger amount of valid containers and more likely to have a container of cost 1.

5.5 Quality of Analysis

I am happy with the manner I compared the exact and heuristic solutions, and with giving commentary and output examples on why the heuristic performed optimally. The only thing I wished I could have done is

```
C:\Users\Ismael Martinez\Desktop\IFT 6521 - DP\DynamicProgrammingProject>python ContainerLoadingDP.py -h
 -h Help
-s (string)_ <stack file path>
-r (string) <railcar file path>
 -d (optional) Show every step output
```

Figure 12: Options for running the script.

giving formal proofs as to why at least one container of cost 2 will yield a container of cost 1 after the first is moved.

5.6 Quality of Code

There's a brief description of the overall code purpose at the top of the script, as well as a description of the code, of inputs and of outputs for each function. Though I am concerned that some function names are slightly vague, the function descriptions are clear and the logic is mostly easy to follow. There are an adequate amount of comments throughout the program.

6 Appendix

6.1 Exact Solution

The code algorithm and all of its functions all reside within a single file, `ContainerLoadingDP.py`.

Referring to Figure 12, we can run the script in the command line. Ensure that you are in the correct directory as the python script. If not, use `cd` to change directory.

Type the following command:

```
python ContainerLoadingDP.py -s <path_to_stackInstance_file.txt> -r
<path_to_railcarInstance_file.txt>
```

These two files are assumed to be tab-separated .txt files. To view the output at every step including the list of valid containers, add `-d` tag at the end.

```
python ContainerLoadingDP.py -s <path_to_stackInstance_file.txt> -r
<path_to_railcarInstance_file.txt> -d
```

6.2 Heuristic Solution

The code algorithm and all of its functions all reside within a single file, `ContainerLoadingHeuristic.py`.

Similar to the previous script, we can run the script in the command line. Ensure that you are in the correct directory as the python script. If not, use `cd` to change directory.

Type the following command:

```
python ContainerLoadingHeuristic.py -s <path_to_stackInstance_file.txt> -r
<path_to_railcarInstance_file.txt>
```

These two files are assumed to be tab-separated .txt files. To view the output at every step including the list of valid containers, add `-d` tag at the end.

```
python ContainerLoadingHeuristic.py -s <path_to_stackInstance_file.txt> -r
<path_to_railcarInstance_file.txt> -d
```

References

[1] D.P. Bertsekas, Dynamic Programming and Optimal Control, Vol. 1, Chapter 1.3, 1.4, Athena Scientific, Belmont, Mass., 2005 et 2012.

[2] Mantovani, S., Morganti, G., Umang, N., Crainic, T.G., Frejinger, E., Larsen, E. (2017). The load planning problem for double-stack intermodal trains, European Journal of Operations Research, DOI 10.1016/j.ejor.2017.11.016.