Contents

Video 2

Solution 2

Overall Concepts 2

Development Procedure 6

Advantages and Limitations 7

Improvement and Evidence 7

Best Suited Robotic Arm 9

Video

The desired position of parcels in the following video is:

desiredParcels = [

[0 0 0 0 0 0];

[1 0 0 0 0 0];

[2 0 0 0 0 0];

[3 0 0 0 0 0];

];

The video shows the arm starts from the home position, then picks up and put down parcels to their desired positions and returns to the home position.

The arm firstly moves to box1, picks up parcel1 and puts it down to box2. It then returns to box1 to pick up parcel2 and puts it down to box3. Then it returns to box1 to pick up parcel3 and puts it down to box4. Finally, it returns to the home position.

During the picking and putting movements, the robot arm raises a bit up after the parcel is gripped to avoid hitting other parcels. It also moves down a bit when putting the parcel down to avoid any damage caused by dropping from high place.

The link to the video stored in google drive is:
https://drive.google.com/file/d/1wXM5TrCOh_Mwx2f6YkHbSUgOmg3vN_Xv/view?usp=sharing

Please note that it is suggested to open it with "Video player for Google Drive" for higher video resolution, otherwise the resolution is restricted to 360p by google drive, which may be quite blur.

Solution

The submitted files include "project2.m" for the simulation algorithms, and the "project2Real.m" is used for the real robot simulation in laboratory.

Overall Concepts

This algorithm is a dynamic solution, which mainly contains three parts: setting up initial variables, "put" function to remove all the parcels on the goal parcel and spare the desired box, and finally "hanoi" function to move the goal parcel to the desired position. They will be explained in detail below:

1. Set initial variables

Before the algorithm, the desiredPosition matrix is loaded to get the number of provided boxes and the number of parcels, so that the algorithm is scalable.

2. "Put" function

For n parcels, a loop is applied from the largest parcel n to smallest parcel 1. For each loop, if the number of parcels is n, the program separates the n parcels into two stacks: parcel 1~n-1, and parcel n, which is the parcel at the bottom and the parcels above it. The parcel 1~n-1 is considered as one stack, by doing this, the parcel n can be moved to desired position after moving those parcels above it away and after the destination position becomes available.

There are several limitations when finding the available end positions for those n-1 parcels. Firstly, the start position of parcel 1~n-1 and end position of parcel n becomes unavailable, so that the parcel 1~n-1 will not remain at the start or occupy the desired position of parcel n. Secondly, the program checks the availability of other boxes based on two conditions: 1. If the box is empty; 2. If the parcel on top of that box is larger than parcel n-1. If the box satisfies these two conditions, the box is marked as available. Thirdly, the program checks whether the desired position of parcel n-1 is available based on the availability checked before, if it is available, the end position of parcel 1~n-1 is set to be that box, otherwise, the program will randomly pick a box that is available to be the end position. This step is to save steps to move parcel n-1 to desired position for next loop.

After removing the n-1 parcels above the parcel n, the program now moves parcel n from start to desired position. This program loops until n = 1, so that all parcel n can be moved to desired positions. Thus, the program for this stage can be concluded as:

```

For parcel n in parcels 1~n (from largest to smallest):

    if the parcel is not at the desired position:

        find available end position for n-1 parcels

        move n-1 parcels to that end position

        move parcel n to desired position

    end

end

```

The example below shows when there are five parcels if the desired position of parcel 4 is at box3 and the desired position of parcel 5 is at box4. So, the program separates parcels in to “parcel 5” and a stack of parcels “1~4 parcels”, and since box3 is empty, the “1~4 parcel” can be moved there. The next step is to move parcel 5 to box4 and this function is finished.

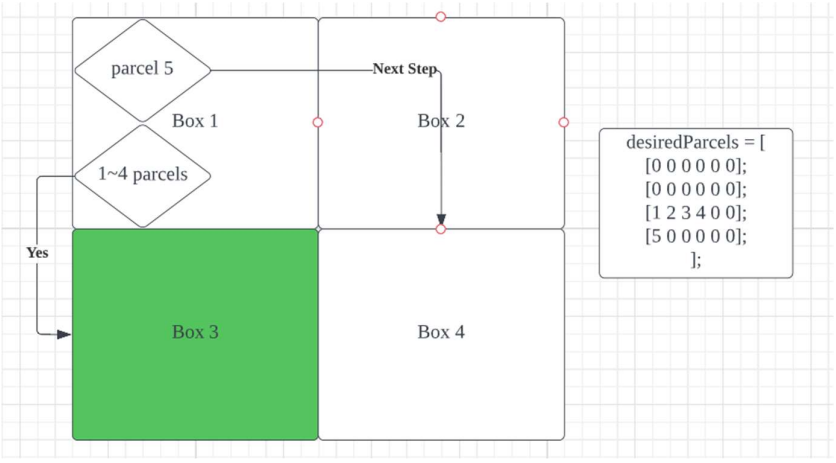

Figure 1: 5 parcels example 1

Other conditions remain the same, if the desired positions are the same for parcel 4 and parcel 5, then the program randomly chooses other available boxes, in this case is box2 or box3. The example is shown below:

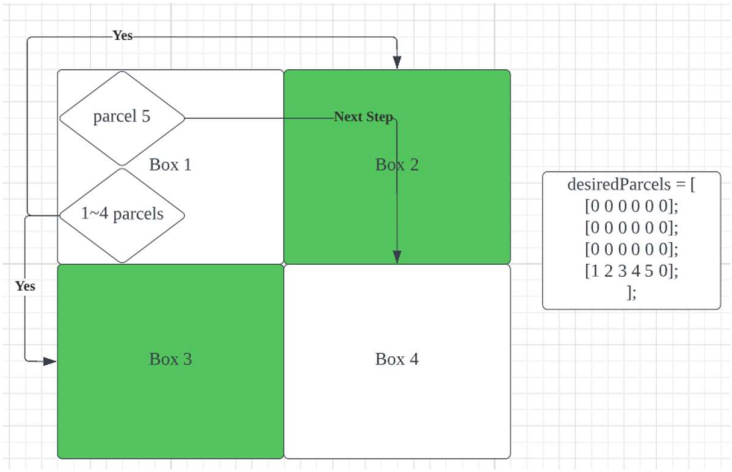

Figure 2: 5 parcels example 2

3. Hanoi function

To implement the "move" action, "hanoi" function is used recursively to move a stack with m number of parcels from a start position to an end position. It also separates the stack into two parts: 1. 1~m-1 parcels; 2. Parcel n. It firstly moves parcel 1~m-1 to available boxes, then moves parcel m to the desired position and finally moves parcel 1~m back to the top of the parcel m. An example of 5 parcels is shown as below, where the three steps are marked, and the green areas show available boxes for 1~4 parcels to move.

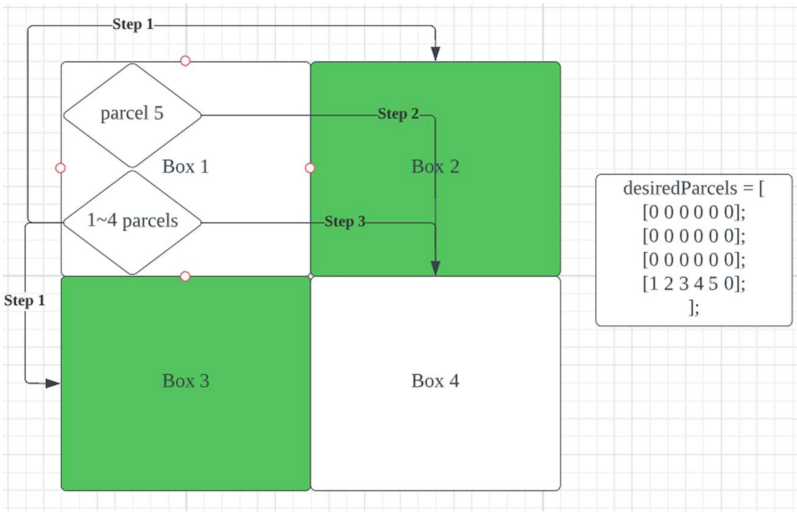

Figure 3: 5 parcels example 3

Besides the m number of parcels, start and end positions, this function takes another input as "parcel". This is the "parcel n" from the "put" function, the goal parcel to be moved to the desired position. If m is larger than 1, the program checks whether there is no parcel above the goal parcel n and whether the desired position of parcel n is available now, if these two conditions are met, the program will move on to next recursive. The reason to check this is because the parcel n could have

been able to move to the desired position before the above m parcels being moved together as a stack to somewhere else.

The following example shows when moving 1~4 parcels to box3, this situation may happen where parcel 1 is not yet been moved to 2~4 parcels, while the parcel 5 is free to move to box4 now. In this case, the program will stops moving parcel 1 to box3, and start moving parcel 5 to box4, which saves steps.

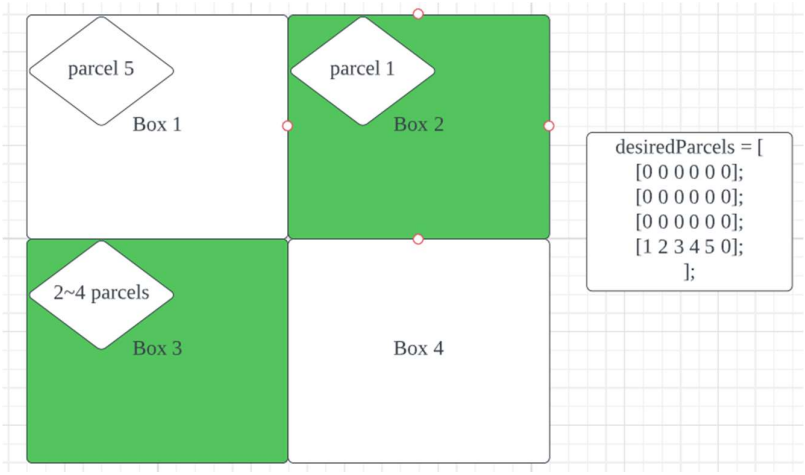

Figure 4: 5 parcel example 4

Moreover, the desired position of parcel n is always marked as unavailable if there is other box for those m parcels to move to. By doing this, the number of steps can be reduced to move the parcels on the desired position away to spare the space for parcel n.

Thus, the program can be concluded as:

```
Function hanoi(m, parcel n, start, end)
If n = 1:
        Pickup(start)
        Putdown(end)
Else:
        If there are no parcels on parcel n and the desired position of parcel n is available:
                Return;
        find available end position for m-1 parcels
        hanoi(m-1, start, available end point)
        hanoi(1, start, desired position)
        hanoi(m-1, available end point, desired position)
```

Development Procedure

1. Simulation Algorithms

At the beginning, I was inspired by the puzzle of Hanoi, and tried to complete this algorithm by only writing the recursive function to move n parcels. Then, I found another function was needed to provide the value of n for each recursive round. To figure out the n values, I firstly tried to use an array to store the continuous parcels in each row of desired positions. A loop was also created to loop the list for each box to try to move n parcels to each box recursively. This worked for some cases but failed with case “box2: parcel 2; box3: parcel 1, parcel 3”. In this case, the list that stored values of n was: “box2: 1; box3: [1, 1]”, which made the program put parcel 3 on parcel 1, because when placing n = 1, the program did not check the availability of the desired position.

After observing this problem, I thought about looping from the bottom of n values of each box, but this caused another problem. Considering the same situation as mentioned above, the desired position was: “box2: parcel 2; box3: parcel 1, parcel 3”, even if the program started from the bottom of each box, the program did not know how to put parcel 3 from the start position box1 to the desired position box3, because the n value provided was based on the desired position, while the program started from the start position.

Thus, instead of looking into the desired position to find n values, I turned to investigate the start position. In this way, I tried to get n value based on how many parcels need to be removed from the largest parcel, so that the largest parcel could be moved to the desired position. Then each parcel was able to move to the desired position after removing all the parcels above them. This was how I developed the basic concept of “put” function.

The basic concept of “hanoi” function was basically the same as taught in the tower of hanoi puzzle, which was to firstly move n-1 parcels to a spared box, then move parcel n to the desired position and finally move the n-1 parcels back to the desired position.

Despite the challenge in developing the basic concept of “put” function, it was also a challenge to make use of spared boxes. Since in the provided tower of Hanoi puzzle video, there was only one spared position, while in this assignment, there was totally four boxes and the parcel could also be put in box1, thus, there was totally three spared boxes. Moreover, the parcels could only be put on the larger boxes, so, it was challenging to define which box was available. To achieve this, I used a global list named “spare” to store “0 – for available; 1 – for occupied” for each box. At the beginning, the values are all “0”, then it was designed to mark the end position “1” and the start position “0” after moving one parcel. But this soon made all values to “1” especially when there were many parcels like 5 parcels. Thus, I created a function called “checkAvailable” to also check each box if the parcel is larger than the current parcel, if the parcel at the top of the box is larger than the current parcel, it is marked as “0”. By adding this “spare” list, before moving n-1 parcels, the program checks the availability of each box and can choose to move to the spared boxes instead of moving to a specific box.

2. Real Robot Programming

For the real robot programming, I hard coded the whole program for soring 3 parcels. But at the beginning, I used scalable algorithms, like retrieving the number of parcels from the provided desired position matrix and calculate the height of picking up the parcels and putting down the parcels based on the number of parcels. However, this did not work as expected when tested it out in the laboratory. There were always problems with the values of the gripper height. The gripper sometimes was too high to grip the parcel at the expected height, or the gripper crashed into the remained parcels when returning to the box1. The first problem may be caused by the minor error

with each parcel thickness since they may not always be the same as 3mm. And this was solved by testing out the required height to pick up each parcel. The second problem was caused by firstly, the error in expected height as mentioned in the first problem, and secondly, the enough height may not be provided during the movement from the desired position to start position. This was solved by firstly moving up to a safe height after putting down the parcel and then moving back to other position to ensure other parcels are safe.

Advantages and Limitations

1. Simulation Algorithms

As mentioned in the overall concepts, the numbers of provided boxes and the parcels are counted by reading the desired position matrix, so one of the advantages is they are scalable.

Secondly, since as described before, a list called "spare" is used to store the availability of each box, this is one advantage not only because it records the availability dynamically and the worker or the system can supervise the status all the time, the system can also manually change and monitor the status of each box if some position becomes occupied by other things or being damaged, so that the robot can avoid using that position.

Thirdly, since the robot checks whether the top of the goal parcel to be moved is empty and whether the desired position is available for the goal parcel for each round, the program saves steps as shown in "Figure: 5 parcel example 4". This also saves energy and improves efficiency in real life operation.

Besides checking the availability of moving the goal parcel, the program also marks the destination of the desired position of the goal parcel as "occupied", so that the program will spare that place to save steps moving away the parcels on the goal desired position if there is another available box to move to. This can be one advantage, since it does save steps compared with before. But it can also be a limitation, since in some cases, several steps can be further saved by firstly making use of the desired position and then put the parcel to other boxes, which also does not occupy the goal parcel's place. If the difference between the number of parcels and that of provided boxes increases, this limitation will have larger drawback on the efficiency of the program and in real life operation.

2. Real Robot Programming

As mentioned before in "Development Procedure", the real robot program was hard coded due to the uncertainty and unexpected error in parcel thickness. The advantage of doing this is that the accuracy of implementing this specific task was high and very reliable, while the limitations of the hardcoded program are obvious. It is unscalable and therefore has low repeatability and is not flexible for different situations. The possible improvements and related reasons will be discussed in "Improvement and Evidence" section below.

Improvement and Evidence

1. Simulation Algorithms

There were several improvements done to reduce unnecessary steps and improve the efficiency. The first improvement was to check whether the parcel was placed at its own desired position during placing the n-1 parcels and remove its own desired position from the "spare" list if there is another place to put. The reason to do this is because when moving the n-1 parcels, since it is not only moving one parcel and there is a pre-check whether the parcel need to be moved again, so the parcel being moved will be moved again to another place. In this case, it is better to not placing it to

the desired position, otherwise, more steps will be taken to spare places for other larger parcels than itself. This improvement has been tested for desired position: “box2: parcel1, parcel3; box4: parcel2;”. The number of steps was reduced by 1, which was caused by avoiding putting parcel1 to box2 at the beginning, and instead parcel1 was moved to box3, so that parcel 3 can be directly put to box2 later and then parcel 1 was moved back. This improvement saves more steps as the number of parcels increases.

The second improvement was to check whether the goal parcel is free to be moved and whether the desired position is free for the parcel to move to. Before the improvement, the n-1 parcels were programmed to finish moving to another place that was not the desired position, so that the parcel n had no parcels on it and the desired position was spared for parcel n to move to. During the movement of n-1 parcels, there are several steps, firstly, the parcels need to be moved to other places, and secondly, regroup the parcels together to put in one place. Then among these steps, there can be a point where there are no parcels on parcel n, and the desired position is spared to be moved to, while the n-1 parcels have not been put together yet. Thus, to reduce the unnecessary steps which were used to put n-1 parcels together, another check was added to check whether the parcel n is free to move to the desired position at each step when moving n-1 parcels, if it is, then the program will terminate the movement of regrouping the n-1 parcels and move on to moving the parcel n to desired position. This improvement was tested on desired positions: “box2: parcel3; box3: parcel 1, parcel 2, parcel 4; box4: parcel 5”. The steps were reduced by 11, which were used to removing parcels 1~4 and regrouping the parcels 1~4. However, during the movement, the parcel 5 was free to be moved to box4, and the improvement stops regrouping the parcels 1~4 and reduces these steps.

These improvements can reduce the time and energy cost in real life robot project and thus improve the efficiency of the factory.

2. Real Robot Programming

As for the hardcoded program, one improvement about the height was applied. Originally, the robot moved directly to the start position after putting down the parcel. This may crash into the remaining parcels at the start position. Thus, the improvement was to add a safety boundary. This boundary was added to the expected height to pick up the next parcel at the start position. In this way, the robot was firstly moved up to the height, which is the result of combing the expected height for next parcel and the safety boundary together, and then move horizontally to the start position to pick up the next parcel. By moving up to safety height and moving horizontally, the robot avoids crashing the parcels when moving to different height, and the additional safety boundary reduces the danger caused by the parcel thickness error. This was tested in the lab ur5e robot and it successfully picked up the parcel without crashing into anything.

To improve the repeatability, reliability, and efficiency of the program, it firstly should be changed from hardcoded to dynamic programming. This can be done by combining the simulation algorithm with the real robot pick up and put down functions. Thus, the most important improvement should be related to solving the uncertainty and error of parcel thickness to ensure the robot could pick up the parcel in an expected height and put down the parcel

This can be solved by providing a test boundary, which is used to move the gripper further down if no parcel was gripped. For example, if the test boundary is set to 1mm, and the gripper does not grip the parcel at the expected height, it will go 1mm further down and add the test boundary further till the parcel is gripped. This can solve the error in parcel thickness and by setting the test boundary, the worker can adjust the boundary anytime based on different size of parcels.

Furthermore, to improve the repeatability and efficiency, the program can remember and adjust the latest tested height and update it based on the tested boundary. For example, if the robot gripped the parcel after going 1mm further down, then the new height is recorded as the pickup height. For next round, the robot will calculate the expected height to pick up and put down parcels based on the updated height. The same can be used when the actual height is larger than the expected height, so, if the end effector detects force that exceeds a force boundary, the robot can stop moving down and update the expected height and start gripping the parcel.

Since this program needs a sensor to check whether the parcel is successfully gripped and the force in the end effector. But if these improvements were used in real life project, the test height boundary can adjust the height dynamically, reduce the expected height error no matter how many rounds it executes, and thus improves the repeatability and reliability. The force boundary can also adjust the expected height and at the same time avoids pushing down too hard to crash the parcel or hurt anyone.

Best Suited Robotic Arm

To compare type of robotic arm among all the robotic arms mentioned in lectures, including cartesian, scara, articulated & delta, the requirements of this task need to be defined.

The parcel picking up and putting down task requires picking up and putting down parcels in straight line, the speed of movement can be relatively quick to improve efficiency and the parcels are not heavy. The following comparisons discuss these points around different types of robotic arms.

The articulated robot arms have the most flexible movement, but is slower, more complicated, and more expensive than other types of robotic arms, and it can also lift heavy goods. One example of using articulated robot arm is Articulated Robot Palletizer LOGICO (https://www.youtube.com/watch?v=F5UJrS982R0), where the robot is required to play multiple roles flexibly, including picking up and putting down boxes with movement in straight lines, adjusting positions quickly with movement in curve lines (movej). With these mentioned, the articulated robotic arm can do this parcel picking job, but is not best suited, since this task does not require much flexibility or moving in circular line, and the parcel is not heavy to pick, and also the articulated robot has slow speed.

The Cartesian robotic arm can lift heavy parcels, and it is accurate, less expensive, meanwhile, it is slow, not flexible. There is an example of Cartesian robot palletizer mod (https://www.youtube.com/watch?v=FlUhbt67Xzs), where it picks up and puts down parcels moving in straight line slowly. This type of robotic arm can also finish the task, but there is no need for heavy lifting or large workplace and the movement of this type is slow, thus, the Cartesian robot is also not suitable for this task.

The delta robotic arm is designed for the fastest movement, but it is limited to lightweight parcels and limited reach. One example of Delta palletizer example is: https://www.youtube.com/watch?v=Qd09PRfMIv0, where the Delta robot was used to pick and place chocolates from one side to the box. The task seems to be similar to the parcel placing task. The SCALRA robot can be compared to the Delta robot, which also has limited flexibility and lightweight goods, the movement is also fast but not as fast as the Delta robot. One example of SCARA palletizer example is: https://youtu.be/vKD20BTkXhk, where the SCARA robot places the small ball in different positions with high accuracy.

These two robotic arms both have similar industrial employment examples, but although the delta robot is quicker, it normally needs security cover, which takes larger space and more expensive.

Although SCARA robotic arm is not as fast as Delta, but it is fast enough for this task and requires less working space and less expensive. Moreover, since the vibration for each arm needs to dissipate, the Delta robotic arm is not accurate as SCAEA robot, which is not suitable for high repeatability. Thus, by comparing these robotic arms, SCARA is chosen to be the most suitable type for this specific task.