DEGREE PROJECT, IN SOFTWARE ENGINEERING OF DISTRIBUTED SYSTEM , SECOND LEVEL

STOCKHOLM, SWEDEN 2015

Optimizing Task Sequence and Cell Layout for Dual Arm Robot Assembly Using Constraint Programming

ZHENGYANG ZHAO

KTH ROYAL INSTITUTE OF TECHNOLOGY

INFORMATION AND COMMUNICATION TECHNOLOGY

Optimizing Task Sequence and Cell Layout for Dual Arm Robot Assembly Using Constraint Programming

Zhengyang Zhao

Supervisor:
Johan Wessén
(ABB Corporate Research, Sweden)

Advisors at KTH:
Prof. Christian Schulte
Gabriel Hjort Blindell

Examiner:
Prof. Christian Schulte

Master's Thesis
September 9, 2015

School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

Abstract

Nowadays, assembly robots are increasingly used in the manufacturing industry to replace or collaborate with human labors. This is the goal of the dual arm assembly robot developed by ABB. With the rapid upgrading in consumer electronics products, the lifetime of an assembly line could be only a few months. However, even for experienced programmers, to manually construct a good enough assembly sequence is time consuming, and the quality of the generated assembly sequence is not guaranteed. Moreover, a good robot assembly sequence is important to the throughput of an assembly line. For dual arm robots, it is also important to obtain a balance between the two arms, as well as handling scheduling conflicts and avoiding collisions in a crowded environment.

In this master thesis, a program is produced to automatically generate the optimal assembly sequence for a class of real-world assembly cases. The solution also takes the layout of the assembly cell into account, thus constructing the best combination of cell layout, workload balancing, task sequence and task scheduling. The program is implemented using Google OR-Tools – an open-source support library for combinatorial optimization. A customized search strategy is proposed and a comparison between this strategy and the built-in search strategy of Google OR-Tools is done. The result shows that the used approach is effective for the problem study case. It takes about 4 minutes to find the optimal solution and 32 minutes to prove its optimality. In addition, the result also shows that the customized search strategy works consistently with good performance for different problem cases. Moreover, the customized strategy is more efficient than built-in search strategy in many cases.

Keywords:
Constraint Satisfaction Problem, Constraint Optimization Problem, Vehicle Routing Problem, Dual Arm Robot Assembly, Search heuristics

Sammanfattning

Numera används monteringsrobotar alltmer inom tillverkningsindustrin för att ersätta eller samarbeta med människor. Detta är måluppgiften för den tvåarmiga monteringsroboten, YuMi, som utvecklats av ABB. Med den korta produktlivslängden för hemelektronikprodukter kan livslängden för en monteringslinje vara ett fåtal månader. Även för erfarna robotprogrammerare är det svårt och tidsödande att manuellt konstruera en tillräckligt bra monteringsordning, och dessutom kan resultatets kvalitet inte garanteras. En bra monteringsordning är nödvändig för genomströmningen i en monteringslinje. För tvåarmiga robotar, är det också viktigt att få en balans mellan de två armarna, samt hantering av schemakrockar och undvika kollisioner i en trång miljö.

I detta examensarbete har ett program skrivits, som automatiskt genererar optimala lösningar för en klass av verkliga monteringsfall. Lösningen tar hänsyn till utformningen av monteringscellen och arrangerar cellen på bästa sätt, balanserar arbetsbelastningen, ordnar och tidsbestämmer uppgifter. Programmet använder sig av Google OR-Tools – ett öppet kodbibliotek för kombinatorisk optimering. Dessutom föreslås en skräddarsydd sökstrategi, som jämförs med Google OR-Tools inbyggda sökstrategi. Resultatet visar att den använda metoden är effektiv för problemtypen. Det tar ungefär 4 minuter att hitta den optimala lösningen och 32 minuter för att bevisa optimalitet. Dessutom visar resultatet att den anpassade sökstrategin konsekvent har en bra prestanda för olika problemfall. Dessutom är den anpassade strategin effektivare än den inbyggda sökstrategin i många fall.

Nyckelord:
Constraint Satisfaction Problem, Optimeringsproblem med bivillkor, Ruttplanering, Tvåarmad robotmontering, Sökheuristik

Acknowledgements

I would like to thank my supervisor at ABB Johan Wessén for all the help and support throughout the thesis project, and for many instructive discussions. I would also like to thank my academic examiner and advisors at KTH Professor Christian Schulte and Gabriel Hjort Blindell for the invaluable feedback on the thesis work and report. Another thank you goes to Frej Knutar for the inspiring discussions and pleasant cooperation. In addition, I would like to thank everybody at ABB who has helped me during the thesis work. Finally, I want to thank my family for their support and encouragement.

Table of Contents

Acronyms & Abbreviations

COP	Constraint Optimization Problem
CP	Constraint Programming
CSP	Constraint Satisfaction Problem
CVRP	Capacitated Vehicle Routing Problem
CVRPPD	Capacitated Vehicle Routing Problem with Pickup and Delivery
DFA	Deterministic Finite Automaton
JSP	Job Shop Problem
TSP	Travelling Salesman Problem
VRP	Vehicle Routing Problem
VRPPD	Vehicle Routing Problem with Pickup and Delivery
VRPTW	Vehicle Routing Problem with Time Windows

1 Introduction

This thesis work was carried out at ABB Corporate Research in Västerås, Sweden. ABB is a global corporation mainly focusing on robotics and the power and automation technologies. It provides solutions to improve the efficiency, productivity and quality of its customers' operations. The ABB Group of companies operates in roughly 100 countries with about 140,000 employees. ABB Corporate Research develops technologies for future products and services for ABB's core businesses [1].

1.1 Overview

In recent years, due to the rising labor costs, assembly robots are increasingly used in manufacturing industries from large vehicles to small electronics. Some assembly work that requires high precision can only be done by robots. This brings a high demand for robot assembly [2].

ABB is one of the leading manufacturers of industrial robots and robot systems [3]. Since its first commercial electric robot introduced in 1974, ABB has installed over 200,000 robots worldwide in a wide range of fields such as consumer electronics, machine tools, metal fabrication, food and beverage industries, etc.

YuMi® is the latest dual arm assembly robot of ABB [4]. It is designed for small parts assembly that was previously intended for manual labors. The robot is small in size and suitable for the compact production environment.

A good assembly sequence and assembly cell layout is important to the throughput of an assembly line. On the assembly line, a single product assembly is called a cycle. In a common lean production [5] environment, due to the cumulative effect, even a small improvement in a cycle will result in a significant improvement in the total throughput and a large saving over the entire assembly line. Therefore, it is interesting to find the optimal assembly sequence and assembly cell layout so as to get the optimal cycle time.

1.2 Problem

In some manufacturing fields such as consumer electronics, an assembly line needs to frequently reinstall due to the rapidly changing market demands and technique upgrading. In some extreme cases, the assembly line could change every few months. Meanwhile, the installation for the new assembly line often takes a comparatively long time. Even an experienced programmer needs weeks to build up a new assembly line and in most cases with non-optimized assembly sequence. Therefore, finding a way to automatically and efficiently generate optimized assembly sequence becomes important.

1.3 Purpose and Goal

The purpose of this project is to evaluate the suitability of optimizing dual arm robot assembly using Constraint Programming. A Constraint Programming model and a customized search strategy are proposed and tested in this thesis.

The goal is to produce a program to automatically generate high quality assembly sequence for dual arm robot. This program can greatly shorten the installation time for an assembly line and improve the throughput.

1.4 Benefits, Ethics and Sustainability

1.4.1 Benefits

The ultimate goal of this thesis project is to shorten the installation phase of an assembly line meanwhile maximize the assembly throughput. If the outcome of this project is successfully applied in industry, both manufacturers and customers will enjoy the benefits.

With shorter installation time for robot assembly line and higher assembly throughput, new products will be put into market faster. And with cleverer assembly manner on the assembly line, the average energy consumption for each product will be reduced. This will lead to cost savings for the manufactures and might in turn lower the product price for customers.

1.4.2 Ethics

One important function of assembly robots is to free human from repetitive and dangerous jobs. Robots can keep the assembly line running continuously and operate on dangerous materials, such as toxic and radioactive substance, without worries. However, there are concerns that the increasing use of robots raises unemployment in the society [6]. With the development of technology, the advantage of robot over human labor will become even more prominent. In the future, it is inevitable that robots will replace human in more and more assembly work.

1.4.3 Sustainability

This thesis project aims to give the optimal assembly manner for a product assembly. The best assembly manner brings a higher throughput for the assembly line, which will in turn lower the number of robots needed, thus more sustainable.

In addition, since robots make less mistakes, the defect rate of robots is often much smaller than that of human labors. Therefore, the increasing use of robot in product assembly will reduce the waste of raw materials.

1.5 Delimitations

Although one important feature of YuMi robot is to work collaboratively with humans, the problem case in this thesis only considers the assembly scenario that a robot works autonomously without any interference from outside world.

In practice, the movement of robot arm is driven by the instant parameters of kinematics and dynamics. In this thesis, instead of using the instant data, the modeling uses the data of travel time and operation time collected off-line by software simulation.

1.6 Contributions

The author's contributions to this thesis project were as follows:

1. The author was highly involved in designing and implementing the CP model. Particularly, the author proposed and designed the conjunctive constraint and suction capacity DFA, and was highly involved in designing the precedence order DFA.
2. The author designed and implemented the customized search strategy.
3. The author carried out the experiment and evaluation of this thesis report.

1.7 Outline

This thesis report is organized in eight Chapters. Chapter 2 presents the real world problem case studied in this thesis. Chapter 3 gives necessary theoretical background in Constraint Programming for the thesis. Chapter 4 introduces two important problems related to the thesis – the Vehicle Routing Problem and Job Shop Problem. Chapter 5 presents the details of the model and search strategy, as well as the implementation. Chapter 6 and Chapter 7 present the evaluation results and conclusion. Finally, Chapter 8 gives some discussion and suggests future work.

2 Background

An industrial robot is an automatically controlled, reprogrammable, multipurpose manipulator programmable in three or more axes [7]. A typical application of industrial robots is production assembly. Generally, Assembly robots can increase productivity, quality and eventually lower the costs for manufactures.

With the development of technology, the modern assembly robots become more compact and highly flexible to meet the agile production scenarios. The state-of-the-art assembly robots are designed as dual arm humanoids to perform tasks like human beings.

In this chapter, I will introduce the latest dual arm assembly robot of ABB. A real world assembly case of the robot will be introduced. This assembly case is the target problem study case of the thesis.

2.1 YuMi® Robot

YuMi robot, as shown in Figure 2.1, is the latest collaborative, dual arm, small parts assembly robot of ABB. It has flexible hands, parts feeding systems, camera-based part location and robot control systems. One of the most important features of YuMi robot is the human-robot collaboration i.e. human workers and robots work together on the same assembly tasks. However, this thesis only focuses on the situation that a robot works independently in an isolated working environment with dedicated components input and assembly output.

Figure 2.1 YuMi dual arm assembly robot [4]

The robot takes a human size workplace. It has an accurate vision and two arms that contain 7 rotating joints, which allows the robot to work in a very precise and flexible way. In order to perform assembly like a human worker, the robot is provided with several optional hands as shown in Figure 2.2.

Figure 2.2 Optional hands for YuMi robot [4]

The hands consist of gripper tool and suction tool. Suction tool is used for picking up flat components. In addition to picking up, gripper tool is also used for the tasks such as peeling the cover off a component and tapping a component to stabilize it on a fixture. Since the tasks of peeling and tapping are performed on a fixture, this type of tasks is called fixture operation.

2.2 Problem Input and Output

The goal of this thesis is to produce a program that automatically constructs the optimal assembly sequence and assembly layout for the YuMi robot under a class of real-world problem cases. The program takes a number of parameters and instance data as input, and generates two outcomes.

The input parameters and instance data include:

- Assembly cell description
- Robot description
- Assembly Tree
- Travel time matrix
- Task durations

The outcomes include:

- Assembly sequence, including timing of each task
- Assembly cell layout

The input and output will be introduced in details together with the problem case in the next section.

2.3 Problem Case

In this thesis, the problem case is the same as a previous work [8] regarding the cell description, robot description, assembly graph, distance matrix, and task durations. The main difference between the input data in this thesis and that in previous work is that the assembly tree is proposed in this thesis and thus not used in previous work.

In this section, we will introduce the input and output of the thesis together with the target problem case in details.

2.3.1 Assembly Cell and Robot Description

An *Assembly cell* is a limited workspace in which the robot works with product assembly. Within the cell, the robot works autonomously without any interference from outside. A cell contains several auxiliary apparatuses such as trays, cameras, air gun, fixtures, and output buffer.

Figure 2.3 gives an overview of the assembly cell of the problem case in this thesis. The assembly cell contains two fixtures, five trays, five cameras, one air gun, and one output buffer. The output buffer is not shown in this figure but it is also a part of the assembly cell.

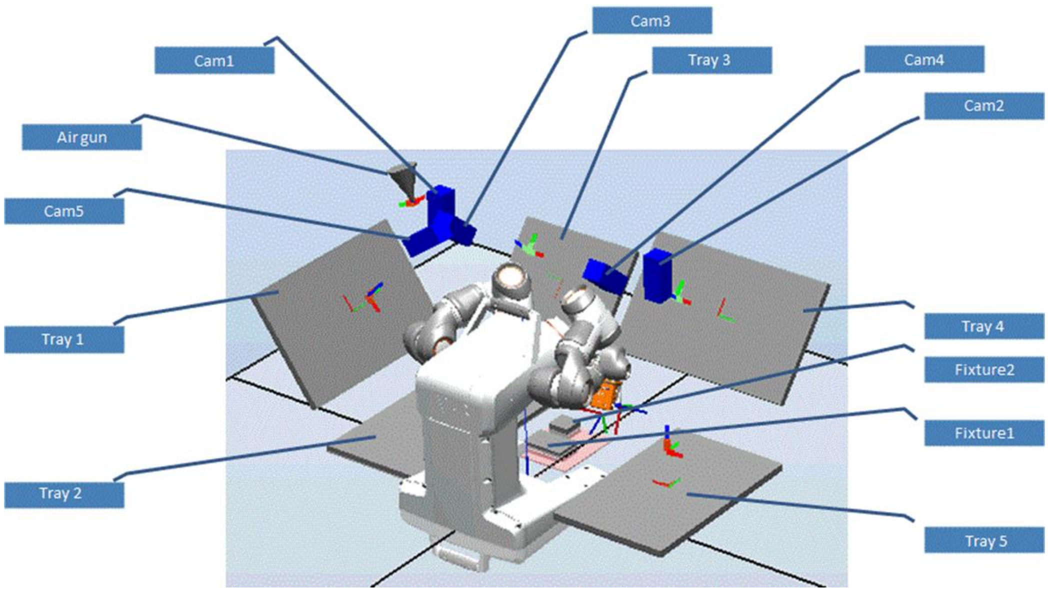


Figure 2.3 The assembly cell of the problem case [9]

The trays provide new parts for the robot to assemble. In the rest of the thesis, the parts to be assembled are also called *components*. After finishing the assembly, the robot puts the final assembly of the product on the output buffer. The trays are filled with components from outside of the cell. The final assembly on the output buffer is also handled by another part of the assembly line out of the cell. In this thesis, we only consider the operations performed in the assembly cell.

The components are initially laid on the trays. Each tray only provides one type of component. At the beginning of an assembly cycle, the robot picks up a component from a tray and places it on a fixture. The fixture is a stable place for the robot to perform further assembly operations. The robot could tap or peel the component that has been mounted on the fixture, put another component onto this mounted component, or put one sub-assembly (which is a combination of several components) onto another sub-assembly.

For each tray in the cell, a dedicated camera is used as a vision system for the robot to locate the component on the tray. A camera is also needed when a component is picked up by a suction tool. This is because a minor error exists in the placement using suction tool. To deal with the error, the robot needs to take a photograph of the component holding by the suction tool, and calculate the compensation angle of placing it on the fixture.

If needed, the air gun is used to clean a component before it is mounted onto the fixture.

Robot description contains the information of the hands used by the robot. In the problem case, the robot has two hands, each with one gripper tool and two suction tools, i.e. the second hand in Figure 2.2.

2.3.2 Assembly Graph

The *Assembly Graph* is used in previous work [8]. It describes how to assemble a product. It gives the breakdown of the entire assembly and the order in which to assemble the components. The assembly graph of the problem case is shown in Figure 2.4. This graph also contains the process steps over the components such as mount and sub-assembly.

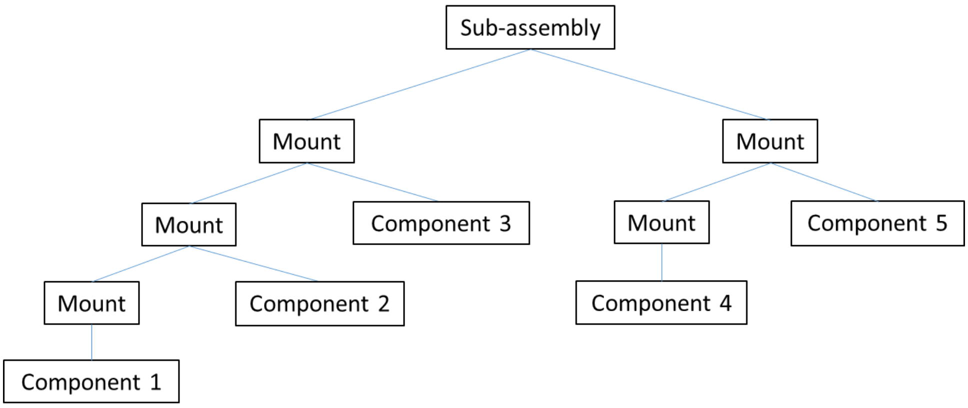


Figure 2.4 Assembly graph of the problem case

There are five components to be assembled in the problem case. To achieve a complete assembly, two subassemblies should be assembled beforehand. For the first subassembly, component 2 should be assembled on top of component 1, and component 3 should be assembled on the combination of component 1 and component 2. For the second subassembly, component 5 should be assembled on top of component 4. After obtaining the two subassemblies, the first sub-assembly should be placed over the second subassembly to get the final assembly. Also, some of the components require additional process operations such as taking a photo or cleaning by the air gun, which need to be performed before they can be assembled.

2.3.3 Assembly Tree

In this thesis, in order to use more information of the assembly process, the assembly graph is converted into an augmented tree structure description which is called *Assembly Tree*. The assembly tree of the problem case is shown in Figure 2.5.

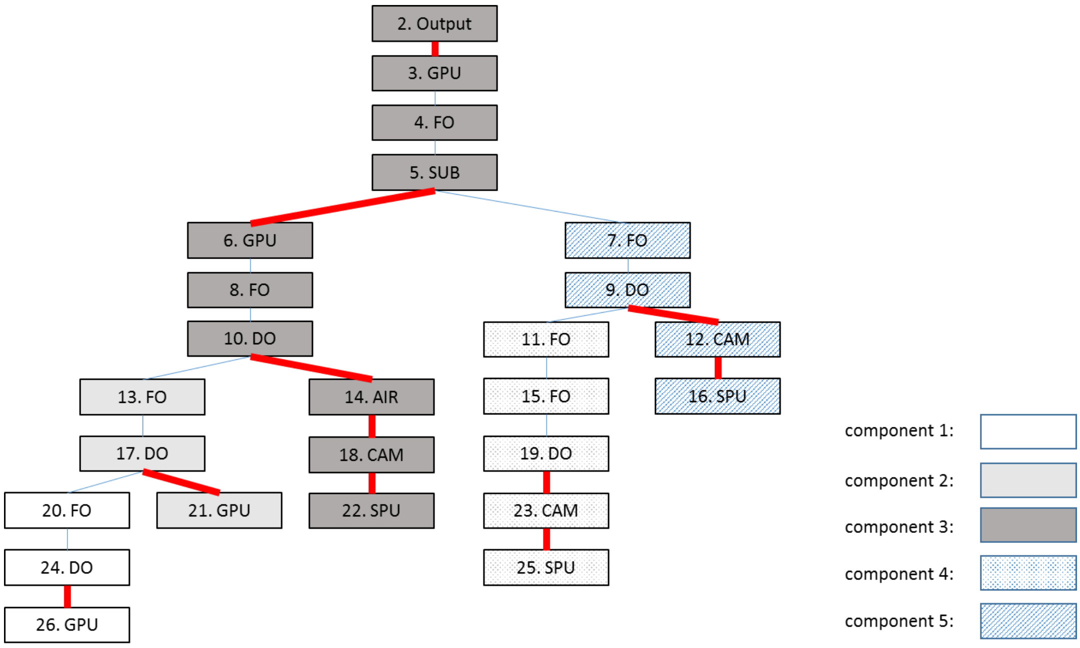


Figure 2.5 Assembly tree of the problem case

In the assembly tree, each node represents a task of the assembly process. The tree contains totally 25 nodes. Tasks on different components are distinguished by the color and texture. **GPU** represents picking up a component using gripper tool, and **SPU** represents picking up with suction tool. **DO** represents dropping off a component on a fixture. **FO** represents fixture operation which could be tapping or peeling on a component mounted on the fixture. **CAM** represents taking the component to a camera and taking a photograph. **AIR** represents taking the component to an air gun to clean it. **SUB** represents sub-assembly. **Output** represents placing the final assembly on the output buffer.

As shown in Figure 2.5, the left branch of node 5 is performed on fixture 1, and the right branch is performed on fixture 2. Node 3, node 4 and node 5 are performed on fixture 2, and node 2 is performed on the output buffer.

In the assembly tree, the precedence order of the tasks is easy to track. Each node should be performed before its parent node. The nodes that are connected by a bold line should be performed by a single arm. This is because if an arm picks up a component, it should also drop the component off, as well as performing the intermediate nodes between the pick-up and drop-off. The nodes that do not have precedence order can be performed in parallel.

2.3.4 Travel Time Matrix and Task Durations

The *Travel Time Matrix* is a three dimension array. The array takes the start location and end location as the first and second dimension, and arm ID as the third dimension. The value in the array is the travel time of the corresponding arm from start the location to end location. For example, $TTM(s, e, 1) = t$ indicates the travel time of arm 1 from location s to location e is t.

In the problem case, there are 14 locations in the assembly cell including five trays, five cameras, two fixtures, one output buffer, and a dummy location for modeling start nodes and end nodes in routing problem.

Task Durations contain the performing time of each task. The time of performing the same task by different arms could be different.

The data of travel time matrix and task durations is derived from the simulation of the problem case in Robot Studio [9] which is a software environment used to program and model ABB robot systems.

2.3.5 Assembly Sequence and Assembly Cell Layout

The outcome of the program produced in the thesis is the assembly sequence and the assembly cell layout.

Assembly Sequence is the description of how to perform the tasks of the assembly. First, for each node in the assembly tree, the sequence determines which arm to perform this node. Second, for all nodes performed by a single arm, the sequence determines in which order to perform the nodes. Third, the sequence also determines the start time of each node.

Assembly Cell Layout describes the tray location of each component, i.e. which tray provides which component, and the camera location of each camera node, i.e. which camera to use when taking a photo.

A manual installation for the problem case was carried out by an experienced engineer of ABB. The time of the entire installation was a couple of weeks and the assembly sequence generated was good enough but non-optimal.

In this thesis with the same problem instance, the tray location of the five components and the camera location of the three camera nodes are not fixed, which brings $5! \times 5^3 = 15{,}000$ times more combinations than the problem without this freedom. The total amount of combinations in the problem case makes it very challenging to find the optimal assembly sequence and assembly cell layout.

3 Constraint Programming

Constraint Programming (CP) is a programming paradigm for solving combinatorial problems. It has been widely used in a number of fields including artificial intelligence, computer science, and operations research. A powerful feature of the CP paradigm is its declarative nature, which means the user states the problem and the computer solves it [10].

To solve a real-world problem using CP, the problem needs to be translated into a *Constraint Satisfaction Problem* (CSP). This step is called *modeling*. In the modeling phase, the user needs to define a set of variables and post a set of constraints on these variables. Each variable in CP has a domain which consists of the possible values that can be set to this variable. Each constraint in CP describes a relation between the variables. A solution to the problem must satisfy all the constraints in the CSP. Sometimes, an objective function is defined over the variables to show a preference between solutions. In this case, the problem becomes a *Constraint Optimization Problem* (COP).

After the CP model is built, dedicated algorithms are used to prune and search the domains of variables. This step is called *solving*. The algorithms in solving phase could be either general or customized. The general algorithms are convenient to reuse for different problems. The customized algorithms usually have better performance in the specific problems. The program that implements these algorithms are called *solver*.

This chapter aims to give an introduction of constraint programming with an illustrative example. The necessary terminologies and definitions of CP will be given as well. The content is mainly based on [11] [12], readers who want to get more information are encouraged to read these literatures.

3.1 CP Modeling

As mentioned above, the real-world problem is converted into a CSP or COP in the modeling phase. This section will first give the formulation of CSP and COP, and then use a classic example to explain how to model a problem in constraint programming.

3.1.1 Constraint Satisfaction Problem

A *Constraint Satisfaction Problem* (CSP) is composed of a set of variables $X = \{x_1, \dots, x_n\}$, a finite domain $D(x_i) \subseteq \mathbb{Z}$ for each variable x_i which consists of the possible values that can be set to this variable, and a set of constraints $C = \{c_1, \dots, c_m\}$ in which each c_j is a relation on a subset of X, i.e. $\mathrm{Var}(c_j) \subseteq X$ where $\mathrm{Var}(c_j)$ denotes the variables involved in constraint c_j.

3.1.2 Constrained Optimization Problem

A *Constrained Optimization Problem* (COP) is a CSP together with an objective function $f: \prod_{i=1}^{m} D(x_i) \rightarrow \mathbb{Q}$ where $m \leq n$ and $\mathbb{Q}$ is the set of rational numbers. The user is interested in finding an optimal solution $d \in \prod_{i=1}^{m} D(x_i)$ which minimizes (or maximizes) the value of the objective function $f(x)$.

3.1.3 A classic problem: 4-Queens Problem

4-Queens problem is a classic combinatorial problem which is defined as to place four queens on a 4×4 chessboard such that no two queens can capture each other. In another word, no two queens are allowed to be placed on the same row, the same column or the same diagonal. A solution to the 4-Queens problem is shown in figure 3.1.

		Q	
Q			
			Q
	Q		

Figure 3.1 A solution to 4-Queens problem

3.1.4 4-Queens problem model in CSP

To convert the real-world problem into a CSP, the user needs to define variables, domains, and constraints on the variables.

Variables and Domains

Since no two queens can be placed on the same column and the number of queens is equal to the number of columns, we define a variable for each column to represent the position of the queen on this column, thus totally 4 variables. The value of each variable denotes the row of the corresponding queen. Figure 3.2 illustrates the definition of the variables on the chessboard.

	x_0	x_1	x_2	x_3
0		Q		
1				Q
2	Q			
3			Q	

Figure 3.2 Illustration of the variables on chessboard

Under this definition, all variables have the same domain:

$$x_1, x_2, x_3, x_4 \in \{0,1,2,3\}$$

Constraints

According to the definition of 4-Queens problem, three constraints are needed in the model. They are listed as follows:

C1: No two queens can be placed in the same row:

$$\forall i \neq j, x_i \neq x_j$$

C2: No two queens can be placed on the same diagonal:

$$\forall i \neq j, x_i + i \neq x_j + j$$

C3: Also, no two queens can be placed on the same back diagonal:

$$\forall i \neq j, x_i - i \neq x_j - j$$

Solution

An *assignment* of a CSP is a function that maps each variable x_i to a value in its domain D_i. If an assignment satisfies all the constraints in the CSP, this assignment is called a *solution*.

As shown in figure 3.2, the assignment $\{x_0 \to 2,\ x_1 \to 0,\ x_2 \to 3,\ x_3 \to 1\}$ is a solution to 4-Queens problem.

3.1.5 The optimization version of 4-Queens problem

Traditional 4-Queens problem is not an optimization problem. However, we can define an objective function that we are interested in to make this problem as a COP. For instance, the objective function is defined as follows:

$$minimize\ (x_0)$$

An optimal solution to the optimization version of 4-Queens problem must satisfy the three constraints mentioned above, meanwhile make the value of the objective function minimum.

3.2 CP Solving

After the CP model is built, the constraint programming system will automatically solve the problem mainly with two operations: *constraint propagation* and *search*. The common process of solving a CP model is iteratively executing these two operations. First, before starting the search, the constraint propagation is performed to prune values, which violate the constraints, from the domain of variables. During the search phase, the domains of the variables are explored in some specific order.

Each time when the domain of a variable x_i is changed, the constraints that involve x_i will perform propagation. If there is only one value left in the domain of a variable, this variable is said to be *bound*. At any time when every variable in the problem is bound, this group of values in the domains is recorded as a solution to the problem.

To solve COPs, the objective value is usually considered as an additional constraint. This constraint enforces that the potential optimal solution must have a better objective value than the current best objective value.

3.2.1 Constraint Propagation

Constraint propagation is an important part in constraint programming. During the search, it efficiently prunes invalid values from the domain of variables to reduce the search space. In a constraint programming system, the functions that perform constraint propagation are called *propagators*. Each propagator can be considered as an implementation of a specific constraint.

A *store* is a function that maps each variable x_i to its domain D_i. Before the search and at the end of each step in the search, the propagators are called to check the current store to prune the values that violate the constraints.

In the following, we use the 4-Queens problem as an example to illustrate how the constraint propagation works. The three constraints in the 4-Queens problem can be achieved by three propagators of AllDifferent constraints which will be introduced in the later section. The three propagators are described as follows:

Propagator 1: $AllDifferent(x_0, x_1, x_2, x_3)$

Propagator 2: $AllDifferent(x_0 + 0, x_1 + 1, x_2 + 2, x_3 + 3)$

Propagator 3: $AllDifferent(x_0 - 0, x_1 - 1, x_2 - 2, x_3 - 3)$

Propagator 1 guarantees that no two queens are on the same raw, propagator 2 and 3 enforce that no two queens are on the same diagonal or back diagonal. The process of constraint propagation on the 4-Queens problem is as follows:

Step 0: $x_0 \in \{0, \dots ,3\},\ x_1 \in \{0, \dots ,3\},\ x_2 \in \{0, \dots ,3\},\ x_3 \in \{0, \dots ,3\}$

Step 1: $x_0 \rightarrow 1,\ x_1 \in \{0, \dots ,3\},\ x_2 \in \{0, \dots ,3\},\ x_3 \in \{0, \dots ,3\}$

Step 2: $x_0 \rightarrow 1,\ x_1 \in \{0,2,3\},\ x_2 \in \{0,2,3\},\ x_3 \in \{0,2,3\}$

Step 3: $x_0 \rightarrow 1,\ x_1 \in \{2,3\},\ x_2 \in \{0,2,3\},\ x_3 \in \{0,2,3\}$

Step 4: $x_0 \rightarrow 1,\ x_1 \in \{3\},\ x_2 \in \{0,2\},\ x_3 \in \{0,2,3\}$

Let D_i denotes the domain of x_i. Step 0 is the initial store of the problem. In this store, none of the propagators can prune any value. Suppose x_0 is set to 1 in step 1. In step 2, propagator 1 removes 1 from the domain of the rest variables. In step 3, propagator 2 removes 0 from D_1. In step 4, propagator 3 removes 2 from D_1 and 3 from D_2.

3.2.2 Search

In general, constraint propagation is not adequate to find the solution to a CSP. The solver needs to search through the domain of the variables that are not bound. At the end of each step of the search, constraint propagation is performed on current store to prune invalid values from the domains. In this way, the search space is reduced continuously.

There are mainly two types of search approaches for solving CSPs – systematic search and non-systematic search. Systematic search ensures that every solution to a CSP will be found by going through all possible assignments. Such approach can be used to find a provably optimal solution to a COP. A typical example of systematic search is the backtracking tree search. Unlike systematic search, non-systematic search cannot tell whether a CSP has solutions if any aren't found, nor find provably optimal solution to a COP. However, such approach is often efficient in finding a feasible solution to a CSP, or an approximation to the optimal solution of a COP. Local search is a well-known example of non-systematic search. This thesis only involves backtracking search. Interested readers can read more about local search in [11] [12].

A backtracking tree search can be seen as performing an exploration on a search tree. The search tree is a virtualized description of the search process. It is generate by branching strategy together with constraint propagation.

Branching

Branching is the operation that decomposes the whole search into smaller sub search branches. A branching strategy defines the structure of a search tree by giving two orderings: the order in which the variables are selected for checking the domain (called *variable ordering*), and the order in which the values in the domain of the selected variable are explored (called *value ordering*).

Search Tree

The search tree is finite. At each node of the tree, the first unbound variable in the variable ordering is selected. The first value in the value ordering of the selected variable splits the search space into two disjoint partitions. Each partition is a decision that adds a branch in the tree and defines the store of the corresponding sub-trees. Under each decision, the affected propagators will do propagation on current store to get a new store as the child node. The root of the tree takes the resulting store of the initial propagation. Each leaf node is either a solution or a failure.

The search tree is explored in some specific order. The most commonly used exploring order is the depth-first left-most order. The solver first goes through the left-most branch until the leaf of this branch. If finding a solution or detecting infeasibility, the solver goes back and turns to the first untried branch on the right.

In the following, we use the 4-Queens problem to illustrate the process of search. The variable ordering is from x_0 to x_3, and the value ordering is to set the selected variable to the minimum value in its domain.

Search for 4-Queens problem as CSP

First, we search for solutions to the 4-Queens problem without considering the objective function. Figure 3.3 shows part of the search tree. The blue nodes are the nodes where unbound variables exist and require further branching. The red nodes are failed nodes where some variable has empty domain. The green nodes are solutions

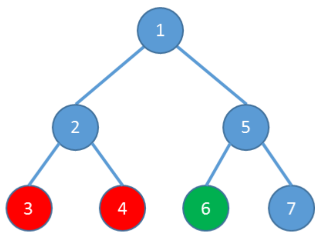


Figure 3.3 Search tree of the 4-Queens problem as CSP

At each node, the constraint propagation is done by the three propagators mentioned in 3.2.1. The process of search with propagation as follows:

Node 1: $x_0 \in \{0, \dots, 3\},\ x_1 \in \{0, \dots, 3\},\ x_2 \in \{0, \dots, 3\},\ x_3 \in \{0, \dots, 3\}$

Node 2: $x_0 \to 0,\ x_1 \in \{2, 3\},\ x_2 \in \{1, 3\},\ x_3 \in \{1, 2\}$

Node 3: $x_0 \to 0,\ x_1 \to 2,\ x_2 \in \emptyset,\ x_3 \in \{1\}$

Node 4: $x_0 \to 0,\ x_1 \to 3,\ x_2 \in \{1\},\ x_3 \in \emptyset$

Node 5: $x_0 \in \{1, 2, 3\},\ x_1 \in \{0, 1, 3\},\ x_2 \in \{0, \dots, 3\},\ x_3 \in \{0, \dots, 3\}$

Node 6: $x_0 \to 1,\ x_1 \in \{3\},\ x_2 \in \{0\},\ x_3 \in \{2\}$

Node 7: $x_0 \in \{2, 3\},\ x_1 \in \{0, 1\},\ x_2 \in \{0, \dots, 3\},\ x_3 \in \{0, \dots, 3\}$

As shown in figure 3.3, node 3 and node 4 are failed nodes with empty domain. The solver found a solution at node 6 and node 7 needs further search.

Search for 4-Queens problem as COP

As mentioned at the beginning of section 3.2, to solve COPs, the objective value can be used as an additional constraint which enforce that the potential optimal solution must have a better objective value than the current best objective value. During the search, when a solution with better objective value is found, the solver will update the current best objective value. At each step of the search, the additional propagator checks the upper bound or lower bound of the objective value of the next branch to explore. If the bound is worse than the current best objective value, prune this branch and turn to the next branch.

Recall the objective function we defined in 3.1.5, which is *minimize* (x_0). The search tree for the 4-Queens problem as COP is shown as figure 3.4.

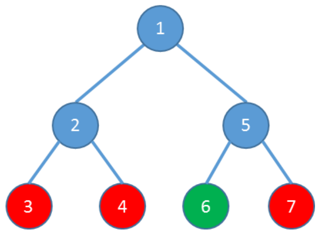


Figure 3.4 Search tree of the 4-Queens problem as COP

The process of search is as follows:

Node 1: $x_0 \in \{0, \dots, 3\},\ x_1 \in \{0, \dots, 3\},\ x_2 \in \{0, \dots, 3\},\ x_3 \in \{0, \dots, 3\}$

Node 2: $x_0 \to 0,\ x_1 \in \{2, 3\},\ x_2 \in \{1, 3\},\ x_3 \in \{1, 2\}$

Node 3: $x_0 \to 0,\ x_1 \to 2,\ x_2 \in \emptyset,\ x_3 \in \{1\}$

Node 4: $x_0 \to 0,\ x_1 \to 3,\ x_2 \in \{1\},\ x_3 \in \emptyset$

Node 5: $x_0 \in \{1, 2, 3\},\ x_1 \in \{0, 1, 3\},\ x_2 \in \{0, \dots, 3\},\ x_3 \in \{0, \dots, 3\}$

Node 6: $x_0 \to 1,\ x_1 \in \{3\},\ x_2 \in \{0\},\ x_3 \in \{2\}$

Node 7: $x_0 \in \{2, 3\},\ x_1 \in \{0, 1\},\ x_2 \in \{0, \dots, 3\},\ x_3 \in \{0, \dots, 3\}$

The search tree is almost the same as that of the 4-Queens problem as CSP. The only different is that node 7 in this tree is a failed node. This is because a solution is found at node 6 and the constraint regarding objective value becomes $x_0 < 1$. When the solver goes to node 7, it found that there will be no better solution under this branch because $x_0 \in \{2, 3\}$. As a result, the branch under node 7 is discarded and the optimal solution is $\{x_0 \to 1,\ x_1 \to 3,\ x_2 \to 0,\ x_3 \to 2\}$ found at node 6.

3.3 CP Variables

This section will introduce the four types of CP variables used in this thesis.

Integer Variable

Integer variable is the most common variable in CP. Such a variable has a finite domain that consists of integers.

Boolean Variable

A Boolean variable is a variable whose domain only contains two Boolean values: true and false. Note that Boolean variables are not integer variables because they usually have a different optimized implementation.

Interval Variable

An interval variable represents an time interval which is often used for modeling tasks in scheduling problems. An interval variable contains the start time, duration, and end time of the time interval. Besides, the interval variable also takes a Boolean variable *performed* indicating if the task has been performed or not.

Sequence Variable

A sequence variable is a variable whose domain consists of the possible orderings of a set of intervals. This variable is often used in scheduling problems. Sequence variables are created by posting disjunctive constraint on a set of interval variables. The disjunctive constraint guarantees that no two intervals overlap, thus can be given orderings.

3.4 CP Constraints

This section will introduce the general CP constraints used in this thesis.

3.4.1 Global Constraints

A global constraint is an expressive and concise constraint that implemented by means of a special propagation algorithm [13]. Modeling with global constraints is often more efficient than using the constraints with weaker propagation. The global constraints can take a number of variables and be used to model complex problems. The global constraints used in this thesis are listed below.

AllDifferent constraint [14]

$$AllDifferent(Vars) \tag{3.1}$$

This constraint enforce all variables in the collection $Vars$ to be pairwise different, i.e. $\forall\, i, j \in \{0,1, \dots, Vars.size - 1\}, i \neq j \Rightarrow Vars[i] \neq Vars[j]$.

Element constraint [15]

$$Element(Vars, index, value) \tag{3.2}$$

This constraint enforce that $value$ is equal to the $index^{th}$ item of the collection $Vars$, i.e. $value = Vars[index]$. The element constraint is also written as $value = Vars_{index}$ in the following chapters.

Disjunctive Constraint [16]

$I = \{(s_i, e_i) \mid s_i, e_i \in \mathbb{Z}, i \in \mathbb{N}\}$ is a set of intervals where s_i and e_i are the start time and end time of an interval. The $Disjunctive(I)$ posts constraint on the interval set I such that

$$\forall\ (s_i, e_i), (s_j, e_j) \in I \text{ where } i \neq j \Rightarrow e_i \leq s_j \text{ or } e_j \leq s_i. \tag{3.3}$$

(3.3) ensures that all intervals in the collection I do not overlap. This constraint is often used in scheduling problems. As a result, the disjunctive constraint returns a sequence variable which contains the possible orderings of the intervals in the collection I.

MapDomain constraint [17]

$$MapDomain(var, Vars) \tag{3.4}$$

This constraint maps the domain of the variable var onto the Boolean variable array $Vars$ such that $\forall\ i \in \{0,1, \ldots, Vars.size - 1\}$, $Vars[i] == 1 \Leftrightarrow i \in var.domain$ and $Vars[i] == 0 \Leftrightarrow i \notin var.domain$.

Circuit constraint [14]

$$Circuit(Nodes) \tag{3.5}$$

This constraint is posted on a digraph consisting of the vertices collection $Nodes$, to enforce covering the digraph with a circuit that visits each vertex in $Nodes$ exactly once. In routing problems, this constraint ensures that a route can only start and end at the depot and eliminates possible sub-tours. Sometimes, this constraint is also mentioned as No-Cycle constraint.

3.4.2 Reification

In a reified constraint (also known as meta constraint) $c \leftrightarrow b$ where c is a constraint and b is a Boolean control variable, the validity of the constraint c is reflected to the value of b [18].

For example, for integer variables x and y and a Boolean control variable b, the reified constraint $x + y = 0 \leftrightarrow b = 1$ enforces the following rules:

- If the constraint $x + y = 0$ holds, then $b = 1$ is posted.

- If the constraint $x + y \neq 0$ holds, then $b = 0$ is posted.
- If b is assigned to 1, then the constraint $x + y = 0$ is posted.
- If b is assigned to 0, then the constraint $x + y \neq 0$ is posted.

3.4.3 Deterministic Finite Automaton

A *Deterministic Finite Automaton* (DFA) is a finite state machine that accepts or rejects finite strings of symbols and only produces a unique computation for each input string [19].

A DFA M is a 5-tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$ consisting of

- a finite set of states Q
- a finite set of input symbols Σ called the alphabet
- a transition function $\delta : Q \times \Sigma \rightarrow Q$
- a start state $q_0 \in Q$
- a set of accept states $F \subseteq Q$

A sequence of transitions is accepted by the DFA only if the sequence begins from the start state and ends at an accept state, and all the transitions in the sequence satisfy the transition function.

Figure 3.1 shows a simple example of DFA.

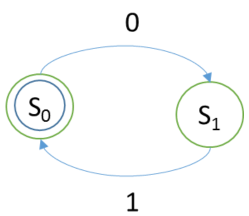


Figure 3.5 A simple example of DFA

In this DFA, the state set $Q = \{S_0, S_1\}$, the alphabet set $\Sigma = \{0,1\}$, start state $q_0 = S_0$, accept set $F = \{S_0\}$, and the transition set $\delta = \{\langle S_0, 0, S_1 \rangle, \langle S_1, 1, S_0 \rangle\}$. This DFA only accepts the sequences with the form of (01)* which represents any non-negative number of the symbols "01".

One application of DFA is to implement regular expressions [20] that define search patterns used for pattern matching with symbols.

4 Vehicle Routing Problem & Job Shop Problem

Vehicle Routing Problem and Job Shop Problem are two theoretical problems related to this thesis. A brief introduction for VRP and JSP will be given in section 4.1 and 4.2. In section 4.3, a discussion about the similarity and difference between the two problems will be presented.

4.1 Vehicle Routing Problem

The *Vehicle Routing Problem* (VRP) proposed by Dantzig et al. [21] is an important industrial problem in the fields of transportation, distribution and logistics.

4.1.1 Travelling Salesman Problem

The *Travelling Salesman Problem* (TSP) is a special case of the Vehicle Routing Problem. The problem is defined as finding the shortest path for a salesman to visit a set of customers and return to the start point. TSP is an NP-hard problem and is well-studied [22] [23] [24].

TSP can be modeled as a weighted complete graph [25]. The customers to be visited are the vertices of the graph. As a complete graph, each pair of vertices is connected with a weighted edge. If no edge exists between two vertices, add an edge with the value of infinity. A solution is a path forming a circuit that visits each vertex exactly once. The objective is to minimize the length of the circuit.

4.1.2 Definition of Vehicle Routing Problem

The VRP is a generalization of TSP. Consider a TSP in which a number of salesmen are added, then the problem becomes Multiple TSP and is equivalent to VRP when there is no constraint over the vehicle capacity. The problem is describes as to find paths for a number of salesmen to visit a set of customers such that each customer is visited exactly once, and the total length of the paths is minimized.

The Vehicle Routing Problem is defined as follows. A fleet of m vehicles are to serve a set of n customers. Each customer must be visited exactly once. All vehicles start and end their routes at a fixed depot. The traveling cost from customer i to j is c_{ij}. The objective is to minimize the sum of traveling costs of all vehicle routes. Finding the optimal solution for VRP is an NP-hard problem [26].

Figure 4.1 shows an example of VRP problem with 3 vehicles.

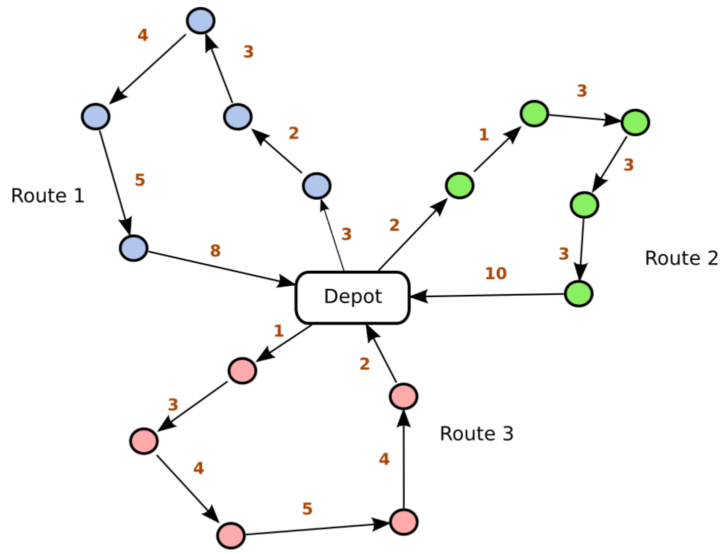


Figure 4.1 An example of VRP with 3 vehicles [27]

For different companies in industry, specific constraints can be added to respective VRP problems. For example, only particular vehicles may be allowed to visit some certain customers, or a customer may prefer the service of a particular driver. In some cases, additional objectives may be required in the problem such as to minimize the number of vehicles.

4.1.3 Variations of Vehicle Routing Problem

There exist a number of VRP variations with specific side constraints. The variants related to this thesis are introduced below.

Capacitated Vehicle Routing Problem (CVRP)

In CVRP, each vehicle has a limited carrying capacity. Let r_i denotes the demand of customer i. In pure CVRP, the demand r_i is a non-negative integer. The set of customers that assigned to vehicle k is C_k and the vehicle capacity is Q. Then the constraint that $\sum_{i\in C_k} r_i \leq Q$ must be satisfied.

In the VRP with multiple resources, the customers are allowed to demand a number of different resources and each vehicle has a specific capacity for each resource. For example, the demand of customer i on resource l is r_i^l and the vehicle k has a capacity Q_k^l on resource l. C_k denotes the set of customers assigned to vehicle k, then the constraint that $\sum_{i\in C_k} r_i^l \leq Q_k^l$ must be satisfied.

Vehicle Routing Problem with Pickup and Delivery (VRPPD)

VRPPD [28] [29] is another variation in which the goods are not only delivered from the depot but also picked up and delivered during the travel. Therefore, the demand of a customer can be either positive or negative. A positive demand indicates that the customer needs the visiting vehicle to bring an amount of goods to him. A negative demand means that the visiting vehicle picks up some goods at this customer and delivers the goods later in the travel.

For CVRP with Pickup and Delivery (CVRPPD), rather than considering the total demands of the customers assigned to a vehicle, the instant cumulative capacity of the vehicle is considered. Let $V_k \subseteq C_k$ denotes the customers that vehicle k has already visited at this moment, r_i denotes the demand of customer i, and Q denotes the capacity of vehicle k. Then the side constraint becomes that $\sum_{i \in V_k} r_i + Q \geq 0$ must hold for vehicle k at any time during the travel.

Vehicle Routing Problem with Time Windows (VRPTW)

In VRPTW [30] [31], a time window $[a_i, b_i]$ is specified for each customer i to be served. In some cases, a customer may have multiple time windows. If a vehicle arrives earlier than time a_i, it has to wait at the customer's location until the time window starts. The additional time constraints make the VRP much more difficult, even finding a feasible solution is NP-hard [32].

4.1.4 Methods for Vehicle Routing Problem

As a problem with a wide range of applications in industry, VRP has been well studied [33] [34]. Some popular methods for solving VRP are introduced below.

Heuristic method

Because the VRP and its variations are NP-hard problems, the most common way for solving large scale VRP is to use heuristic methods [35]. A common heuristic method contains three phases – constructing an initial solution, improving the initial solution through local search, and using meta-heuristic methods to escape from local optima [36].

In the first phase, a construction method is used to create an initial solution. The most well-known VRP construction method is the "Savings" heuristic method [37]. Another classic method is the "Sweep" heuristic method [38].

In the second phase, the local search method [39] [40] is used to look through the neighborhood of current solution to find better solutions. For each local search method, one or more move operators are defined to manipulate current solution to get new solutions. The neighborhood of a solution is a set of solutions generated by applying the move operators on this solution. Some move operators for routing problems are listed in [34]. Typically, a local search method employs several of these operators.

Because local search methods always look for better solutions by following the cost-reducing trend, which often leads the search to being trapped in a local optimal solution. Therefore, in the third phase, meta-heuristics are used to escape from the local optima. Basically, the meta-heuristic method works in two alternative ways. The first way is to allow cost-increasing moves and control the acceptance of the moves. The second way is to expand the neighborhood to avoid local optimal. The most famous meta-heuristic methods include Simulated Annealing [41], Tabu search [42] [43], Genetic algorithms [44], and Variable Neighborhood Search [45].

Constraint Programming

Constraint Programming is a powerful programming paradigm for solving combinatorial problems. Typically, the process of the Constraint Programming method is to construct a solution by incrementally traversing the search tree, backtracking when an infeasible branch is detected, until a feasible solution is found or the problem is proved to have no solution.

Constraint Programming is also applied to COPs such as VRP [34]. Unlike the heuristic method which focuses on efficiently finding cost-reducing solutions, Constraint Programming considers the quality of the solutions. This method performs a systematic search through a large number of feasible solutions to find the optimal or near-optimal solution. Thus, Constraint Programming is often used to prove the optimality.

Hybrid Method

Since the systematic search in Constraint Programming needs to explore every possible solution, Constraint Programming is often a slower approach compared with heuristic method. However, Constraint programming shows great advantages in pruning infeasible solutions. A common way to solve large scale VRP problems is to combine the two methods together [46] [47] [48].

There are mainly two ways to integrate Constraint Programming method with heuristic method. The first way is to let heuristic method take charge of the search. In this case, the constraint model only checks the feasibility of the solution. The second way is to encapsulate the local search movements into a lower level operator, and run the constraint system on top of the operator. In this case, the traditional systematic backtracking search will not be influenced by the movement of the local search. A well-known technique under this case is the Large Neighborhood Search [49].

4.2 Job Shop Problem

Job Shop Problem (JSP) is an optimization problem that has many similarities and is closely related to VRP.

In the classic JSP [11], n jobs are needed to be processed on m machines. Each job consists of a sequence of completely ordered activities. Each activity must be processed during an uninterrupted period of time on a specific machine. The ordering of the activities in a job must be preserved during processing, i.e. an activity cannot start processing until its predecessor in the job has been done. For each machine when processing an activity, no more activities can be processed on this machine until the current process is finished. In real-world problems, sometimes additional side constraints need to be considered, for example there could be constraints on the transition time which represents the time between two processes on the same machine.

The objective of JSP is to find a schedule for the activities to the machines that minimizes the makespan of the whole process. The makespan is the duration between the start time of the first activity among all machines and the finish time of the last activity among all machines. Since the first activity usually starts at time 0, the makespan is equivalent to the finish time of the last activity.

Both constraint programming and heuristic methods are applied for solving Job Shop problem [50] [51].

4.3 VRP vs. JSP

There is a great similarity between VRP and JSP. Many real-world problems fall into the gray zone between the pure VRP and pure JSP. Some criteria to distinguish the two problems are suggested in [52]. The main differences are listed below.

- In VRP typically a customer can be visited by many vehicles, but in JSP an activity can only be processed by a specific machine.
- In VRP the duration of a visit is much smaller compared to the travel time, however, the durations are dominant in JSP.
- In VRP the objective is to minimize the total travel cost of all routes. In JSP the objective is to minimize the makespan of the entire process which is the duration between the start time of the first activity and the end time of the last activity.
- In VRP the time windows of visiting different customers are usually independent, but in JSP the time windows of processing different activities are correlated in a precedence order.

5 Model and Implementation

This chapter will introduce the CP model, search strategy and some implementation details. The model will be described together with the problem case introduced in Chapter 2. However, since the implementation of the model is generic to different problem instances, some of the parameters, such as number of tasks and number of components in the assembly, are presented in a general way.

5.1 Problem Case Analysis

The problem case has been described in section 2.3. The case contains 25 tasks, 2 fixtures, 5 components, 5 tray locations, 5 camera locations and an air gun. A dual arm robot is working in the assembly cell with one gripper tool and two suction tools on each arm.

The problem of finding optimal assembly sequence for the robot is analogized to a Vehicle Routing Problem. In the VRP, each task in the assembly is considered as a node (customer) to visit. The two arms of the robot are modeled as two vehicles. The route for each vehicle is the ordering in which the arm performs the tasks assigned to it. For each arm, a start node and an end node are added as the depot which does not correspond to any task [53].

The problem in this thesis is a rich VRP which contains the features of a number of VRP variants. First, the problem is a Capacitated VRP. Each arm (vehicle) has a limited capacity – one gripper tool and two suction tools. Second, the problem is a VRP with Time Windows. The tasks (nodes) have time windows to be performed (visited). For example if a camera is occupied, the task of taking photographs on this camera has to wait until the camera becomes free. Third, the problem is a VRP with Pickup and Delivery. This is because a component needs to be picked up at some node and delivered to another node. Moreover, the problem contains the characteristics of Job Shop problem (JSP). This is because the task durations are considerable compared to the travel time between nodes, and the objective value of the problem is the makespan of the entire assembly which is also a typical objective value of JSP. As modeled in JSP, the two arms are considered as two machines. The tasks need to be scheduled on the machines disjunctively.

In addition to the restrictions discussed above, the problem contains a number of precedence order constraints and collision avoiding constraints. The collision avoiding constraints do not allow the two arms visiting the same location simultaneously. The precedence order constraints enforce some tasks to be performed in specific orders, for instance, the pick-up, camera, air-gun and drop-off nodes of one component should be performed one after another, as shown in the assembly tree of figure 2.5 with the bold lines.

The objective function of the problem is to minimize the makespan of the assembly i.e. the total cycle time.

5.2 Constraint Programming Model

The general process of building a CP model is to define variables, post constraints on the variables, and define search strategies. In this section, we will introduce the variables and constraints.

As a convention in this report, the name of a set begins with a capital letter. The name of a parameter in the problem case description begins with a capital letter. The name of a single variable and the parameter of a single node begin with a lowercase letter.

5.2.1 Problem Instance Data

The problem instance data are the constant parameters derived from the input data files describing the assembly tree, travel times for each arm and task durations for each arm. The parameters give a description of the assembly cell and the process of assembling a product.

In the model, a task node represents a task in the assembly tree and a start/end node represents the depot in the VRP which does not contain any assembly task. Each node has a unique integer identifier which is used to reference this node in the model. $node_i$ denotes the node with index of i.

The parameters of the assembly cell description are listed as follows:

- $NumberNodes \in \mathbb{N}$ is the number of nodes in the model. The problem case instance of this thesis contains 29 nodes including 25 task nodes, 2 start nodes, and 2 end nodes.
- $NumberRoutes \in \mathbb{N}$ is the number of arms of the robot, i.e. the number of routes to be scheduled in the VRP. In this thesis, $NumberRoutes$ is always equal to 2.
- $NumberComponents \in \mathbb{N}$ is the number of components to be assembled in the assembly.
- $NumberTrays \in \mathbb{N}$ is the number of trays in the assembly cell.
- $NumberCameras \in \mathbb{N}$ is the number of cameras in the assembly cell.
- $NumberFixtures \in \mathbb{N}$ is the number of fixtures in the assembly cell.
- $NumberLocations \in \mathbb{N}$ is the number of locations in the assembly cell. The locations include tray locations, camera locations, fixture locations, output buffer location, and a dummy location for start and end nodes.

- $NumberResources \in \mathbb{N}$ is the number of resource types on each arm. In this thesis, there are two types of resources – gripper tool and suction tool.
- $Q_{jr} \in \mathbb{N}, j \in \{0,1, \dots, NumberResources - 1\}; r \in \{0,1, \dots, NumberRoutes - 1\}$ denotes the capacity of resource j on route r. In this thesis, each arm contains one gripper and two suctions.
- $T_{ijr} \in \mathbb{N} \cup \{0\}, i, j \in \{0, \dots, NumberNodes - 1\}; r \in \{0, \dots, NumberRoutes - 1\}$ is the travel time from node i to node j on route r.

The model contains three types of nodes: task nodes, start nodes, and end nodes. As introduced above, start nodes and end nodes represent the depot in VRP which do not correspond to any assembly task. The indices of the model nodes are organized as follows:

- $S = \{0,1, \dots, NumberRoutes - 1\}$ is the set of indices of the start nodes, one start node for each route.
- $T = \{NumberRoutes, \dots, NumberNodes - 1 - NumberRoutes\}$ is the set of indices of task nodes.
- $E = \{NumberNodes - NumberRoutes, \dots, NumberNodes - 1\}$ is the set of indices of end nodes, one end node for each route.
- $I = S \cup T \cup E$ is the set of indices of all nodes in the model.
- $dummyNode = NumberNodes$ is a dummy index indicates the dummy state before start nodes or after end nodes. There is no model node corresponding to this index. The dummy index is used in posting DFA constraints on the task sequence.

For each node in the model, $node_i.*$ denotes the value of the corresponding attribute of the node. A node has the following attributes:

- $id \in \mathbb{N} \cup \{0\}$ is a unique identifier used to reference a node in the model. The id of a node is equal to the index of this node, i.e. $node_i.id = i$. In the rest of the report, node *i* also refers to the node with index of *i*.
- $component \in \mathbb{N} \cup \{0\}$ is an integer value that indicates which component this node handles. For nodes that do not deal with any components, e.g. start nodes and end nodes, the value is equal to 0.
- $durationRoute_r \in \mathbb{Z}, r \in S$ is an integer value which is the time takes to perform the node on route r, i.e. the time for arm r to perform the corresponding task. The value equal to -1 indicates arm r cannot perform this node.

- $isGripper \in \{0,1\}$ is a Boolean value that specifies if the node requires gripper tool.
- $isSuction \in \{0,1\}$ is a Boolean value that specifies if the node requires suction tool.
- $fixture \in \{0,1,\ldots,NumberFixtures\}$ is an integer value that indicates the index of the fixture on which this node is performed. If the node is not performed on any fixture, the value is equal to 0.
- $isTray \in \{0,1\}$ is a Boolean value indicating if the node is performed on a tray, i.e. a tray pick-up node in the assembly tree.
- $isCamera \in \{0,1\}$ is a Boolean value indicating if the node is to take a photograph, i.e. a camera node in the assembly tree.
- $isAirFlow \in \{0,1\}$ is a Boolean value indicating if the node is to visit the air gun, i.e. an air-gun node in the assembly tree.
- $isSubAssembly \in \{0,1\}$ is a Boolean value indicating if the node is sub-assembly in the assembly tree.
- $isOutput \in \{0,1\}$ is a Boolean value indicating if the node is output in the assembly tree.
- $isFixtureOperation \in \{0,1\}$ is a Boolean value indicating if the node is a fixture operation node in the assembly tree, i.e. tapping or peeling.
- $isDropOff \in \{0,1\}$ is a Boolean value indicating if the node is a drop-off in the assembly tree.

Recall the assembly tree in Figure 2.5, $node_i.parentNode$ denotes the parent of $node_i$ in the assembly tree, and $node_i.leftChild/rightChild$ denotes the left/right child of $node_i$.

Since the start nodes and end nodes are not involved in the assembly tree, the parent and child of these nodes are defined as follows.

For $i \in S \cup E$,

$$node_i.parentNode = node_i.leftChild = node_i.rightChild = null \tag{5.1}$$

The assembly tree shows the precedence order of performing the nodes. The symbol "$\prec$" represents the relation "should be performed before" between two nodes, i.e. $node_i \prec node_j$ denotes $node_i$ should be performed before $node_j$. Apparently, this is a partial order relation and is transitive. Figure 5.1 shows the assembly tree with precedence order.

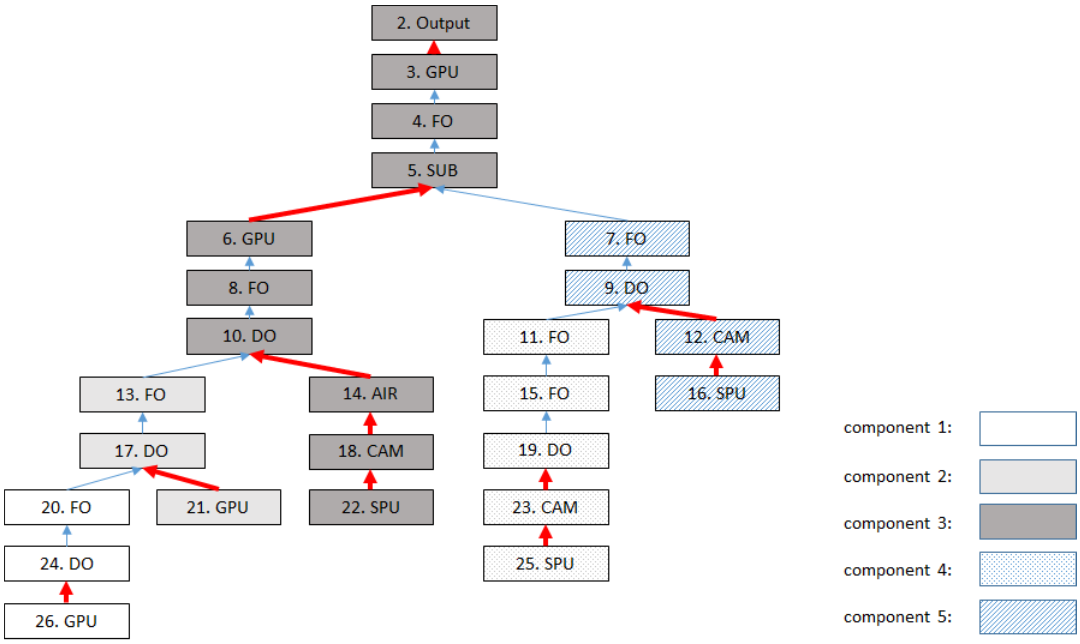


Figure 5.1 Assembly tree of the problem case with precedence order

As discussed in section 2.3.3, each node should be performed before its parent node in the assembly tree, therefore, we have

$$\forall\, i,j \in T, node_j = node_i.parentNode \Rightarrow node_i \prec node_j \qquad (5.2)$$

There exist some subtle precedence orders in the assembly tree. For example, if $node_{26}$ and $node_{21}$ are performed by the same arm, then $node_{26} \prec node_{21}$. This is because if the arm firstly performs node 21 (picking up component 2), due to the gripper capacity, it cannot perform node 26 (picking up component 1) until release the gripper tool at node 17. However, due to the transitivity of the relation "$\prec$", $node_{26} \prec node_{17}$ must hold, which makes a contradiction. This kind of precedence orders is not obvious thus require careful analysis on the problem case and the assembly tree.

According to the precedence order, two additional attributes for each node are defined as follows:

For $i \in T$,

$$node_i.InvalidNext = \{j \mid \mathrm{node}_j \prec \mathrm{node}_i\} \cup \{i\} \qquad (5.3)$$

$$node_i.InvalidPrev = \{j \mid \mathrm{node}_i \prec \mathrm{node}_j\} \cup \{i\} \qquad (5.4)$$

For $i \in S \cup E$,

$$node_i.InvalidNext = \{i\} \qquad (5.5)$$

$$node_i.InvalidPrev = \{i\} \qquad (5.6)$$

These attributes are used to pre-process the domain of variables, which will be introduced in the following section.

5.2.2 Variables

All variables are encapsulated in variable arrays. Each variable is mapped to a node in the model and referenced by the node index.

The decision variables in this thesis are listed below:

- $route_i \in S, i \in I$ is the index of the arm which performs node i.
- $next_i \in I, i \in I$ is the index of the immediate successor of node i.
- $location_i$ is the location in the assembly cell where node *i* is performed.
  $location_i \in \{0, 1, \dots, NumberTrays - 1\}$ where $node_i.isTray = 1$;
  $location_i \in \{NumberTrays, \dots, NumberTrays + NumberCameras - 1\}$
  where $node_i.isCamera = 1$;
  Otherwise, $location_i$ is a constant value.

  In this thesis, only the tray pick-up nodes and camera nodes have unfixed locations. For other nodes with fixed locations in the assembly cell, this variable is set to a constant value corresponding to the fixed location.

The $makespan$ which is the objective value is also a decision variable. This decision variable is used to push the start time of all tasks forward in order to make the task sequence tight.

The other variables used in the model are listed below:

- $sequenceOnRoute_r = \{sequenceOnRoute_{jr} \mid j \in I \cup \{dummyNode\}\}, r \in \{0,1\}$ is an integer variable array derived from the *next* variables. It contains the possible orders of the nodes assigned to route r.

  We interpret this array under an example with 5 task nodes and 2 routes to make it easier to understand. Suppose an assignment of the *next* variables: $(3, 6, 7, 5, 8, 2, 4, 9, 9)$. The routes generated from this assignment are shown in Figure 5.2.

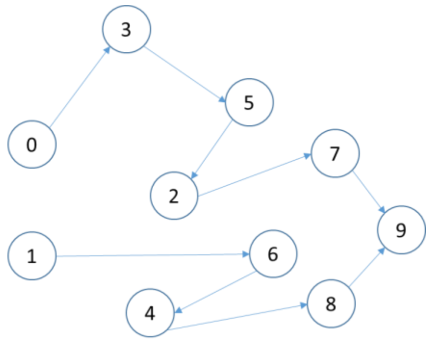


Figure 5.2 The routes generated from the example assignment of *next* variables

In this example, node 2, 3, 4, 5 and 6 are task nodes, node 0 and 1 are start nodes, and node 7 and 8 are end nodes. Route 0 starts from node 0 and passes through node 3, 5, 2, and 7. Route 1 starts from node 1 and passes through node 6, 4, and 8. Both $next_7$ and $next_8$ is pointing to 9 which is the dummy index indicating the routes end here. Then the assignments for the $sequenceOnRoute$ arrays for each route are

$$sequenceOnRoute_0 = (0,3,5,2,7,9),$$

$$sequenceOnRoute_1 = (1,6,4,8,9).$$

In general, the *next* variables form one route for each arm. The variable $sequenceOnRoute_{jr}, r \in \{0,1\}$ is the index of the jth node performed on route r. Then we have

$$sequenceOnRoute_{0r} = next_r,$$

$$sequenceOnRoute_{1r} = next_{sequenceOnRoute_{0r}},$$

$$\dots,$$

$$sequenceonRoute_{jr} = next_{sequenceOnRoute_{j-1,r}}. \tag{5.7}$$

The $sequenceOnRoute$ arrays are connected to the *next* variables and are used to post efficient constraints.

- $nextLocation_i$ is the location where the successor of node i is performed.
- $arrivalTime_i \in \{0,1,\dots,2^{63}-1\},\ i \in I$ is the time of arrival at node i.
- $startTime_i \in \{0,1,\dots,2^{63}-1\}, i \in I$, is the time of starting the operation on node i. The s ime and $arrivalTime$ are used for modeling the situation of waiting at each node.
- $departureTime_i \in \{0,1,\dots,2^{63}-1\}, i \in I$ is the time of departure from node i.

- $interval_{ir}, i \in I, r \in S$ is an interval variable representing the interval during which node i is performed on route r. The interval variable is introduced in section 3.3. Each node has two interval variables, one interval on each route. A constraint is used to enforce only one interval is performed for each node.
- $prev_i \in I, i \in I$ is the index of the immediate predecessor of node i. This variable contains the same information with the *next* variable, however, it is suggested to improve the model of VRP [34].
- $routeSequence_r, r \in S$ is a sequence variable that contains the information about possible ordering of the interval variables on route r. The sequence variable is introduced in section 3.3.

5.2.3 Pre-Processing of Variable Domains

After defining the decision variables, the domain of some variables could be refined according to the problem instance data and the assembly tree.

For each $i \in I$,

- If $node_i.duration_r = -1$, then remove r from the domain of $route_i$.

According to assembly scenario, the duration equals to -1 indicates that arm r cannot perform task i. Therefore, the value r should not appear in the domain of $route_i$.

- If $i \in S$, then

  $route_i = i,$

  $prev_i = dummyNode.$

Since each arm has one and only one start node, the index of each start node is equal to the index of its corresponding arm. Because a start node is the first node on its route, its predecessor is modelled as a dummy node.

- If $i \in E$, then

  $route_i = i - (NumberNodes - NumberRoutes),$

  $next_i = dummyNode.$

The index of each end node is also equal to the index of its corresponding arm. Since an end node is the last node on its route, its successor is modelled as a dummy node.

- If $i \in T$, then

  remove $E \cup \{dummyNode\} \cup node_i.InvalidPrev$ from the domain of $prev_i$,

  remove $S \cup \{dummyNode\} \cup node_i.InvalidNext$ from the domain of $next_i$.

Since an end node is the last node on a route, it cannot be the predecessor of any task node. A start node is the first node on a route, thus it cannot be the successor of any task node. For the same reason, the dummy node cannot be the successor or predecessor of any task node. According to the definition of *InvalidPrev* and *InvalidNext* in (5.3) – (5.6), the nodes in $node_i.InvalidPrev$ ($node_i.InvalidNext$) cannot be the predecessor (successor) of node i.

5.2.4 Constraints

In this section, the constraints posed in the CP model will be introduced.

5.2.4.1 Core constraints

The core constraints are the basic constraints in the model which establish the connections between decision variables.

1) Connecting $route_i$ with $interval_{ir}$

$\forall\ i \in I,$

$$PerformedOnRoutes_i = \{interval_{ir}.performed \mid r \in \{0,1\}\} \tag{5.8}$$

$$MapDomain(route_i, PerformedOnRoutes_i) \tag{5.9}$$

The interval variable $interval_{ir}$ represents the time interval of performing node i on route r. The Boolean variable $interval_{ir}.performed$ in (5.8) indicates if $interval_{ir}$ is performed or not.

(5.9) builds a connection between the variable $route_i$ and the Boolean variable array $PerformedOnRoutes_i$ such that if $interval_{ir}$ is performed, the domain of $route_i$ contains the value of r.

The *MapDomain* constraint [54] is introduced in section 3.4.1.

2) Connecting $startTime_i$ and $departureTime_i$ with $interval_{ir}$

$\forall\ i \in I,$

$$departureTime_i = interval_{i,route_i}.end \tag{5.10}$$

$$startTime_i = interval_{i,route_i}.start \tag{5.11}$$

The variable $route_i$ indicates which arm performs node i. The time of starting the process of node i is equal to the start time of the interval variable $interval_{i,route_i}$ and the arm immediately leaves the node after finishing the process, as seen in (5.10) and (5.11).

3) Connecting $next_i$ with $prev_i$

$\forall\ i \in I$, since $next_i = j \Leftrightarrow prev_j = i$, we get the following element constraints:

$$next_{prev_j} = j \tag{5.12}$$

$$prev_{next_i} = i \tag{5.13}$$

(5.12) and (5.13) make a connection between the *next* and *prev* variables to ensure the consistency of the route. The two constraints are redundant to each other. However, to post both can strengthen the propagation.

4) Connecting a node with its successor and predecessor

$\forall\ i \in I,$

$$route_i = route_{next_i} = route_{prev_i} \tag{5.14}$$

Apparently for each node, its successor and predecessor should be on the same route with this node. (5.14) connects a node with its successor and predecessor using element constraint.

5) Connecting $location_i$ and $nextLocation_i$ with $next_i$

$\forall\ i \in S \cup T,$

$$nextLocation_i = location_{next_i} \tag{5.15}$$

$\forall\ i \in E,$

$$nextLocation_i = Location_{i-NumberNodes-NumberRoutes} \tag{5.16}$$

(5.15) and (5.16) are element constraints. For a start node or task node, (5.15) connects the $nextLocation$ of this node to the location of its successor. Since an end node does not have a real successor, its $nextLocation$ is equal to the location of the start node on the same route, as seen in (5.16).

6) Connecting $startTime_i$, $departureTime_i$, and $arrivalTime_i$

$\forall\ i, j \in T,$

$$next_i = j \Leftrightarrow arrivalTime_j = departureTime_i + T_{ij,route_i} \tag{5.17}$$

(5.17) post the constraint that the arrival time at the next node is equal to the departure time of current node plus the travel time between current node and next node.

$\forall\ i \in S, j \in T \cup E,$

$$next_i = j \Leftrightarrow arrivalTime_j = departureTime_i \tag{5.18}$$

$$arrivalTime_i = startTime_i = departureTime_i = 0 \tag{5.19}$$

(5.18) is posted because the depot is a dummy location, the travel time from a start node to any other node is always 0. Since the start nodes do not contain any assembly task, the durations of start nodes are 0, and the start time is the time 0, as seen in (5.19).

$\forall\ i \in E,$

$$arrivalTime_i = departureTime_i. \tag{5.20}$$

(5.20) post the constraint that the durations of end nodes are 0.

$\forall\ i \in T,$

$$arrivalTime_i \leq startTime_i \tag{5.21}$$

(5.21) post the fact that an arm might arrive early at a node and have to wait at this node for starting the process.

5.2.4.2 One route per node constraint

$\forall\ i \in I,$

$$\sum_{r=0}^{NumberRoutes-1} interval_{ir}.performed = 1. \tag{5.22}$$

(5.22) post the constraint that each node should be performed by only one arm.

5.2.4.3 Disjunctive constraint

$\forall\ r \in \{0,1\}$, $Interval_r = \{interval_{ir} \mid i \in I\}$ denotes the interval variables on route r.

$$Disjuctive(Interval_r), r \in \{0,1\} \tag{5.23}$$

(5.23) post disjunctive constraint on the intervals of each route such that no two intervals on the same route overlap.

$\forall\ i \in I,$

$$next_i = routeSequence_{route_i}.next(i) \tag{5.24}$$

The $routeSequence$ is a sequence variable derived from the disjunctive constraint posted in (5.23) [55]. The sequence variable is introduced in section 3.3. (5.24) cast the $next$ variables from the $routeSequence$ variable with element constraint. The $route_i$ indicates on which route node i is performed and the method $next(i)$ returns the successor of node i on this route.

The disjunctive constraint and the sequence variable ensure that all successors of the nodes on a route take distinct values, as well as implementing a No-Cycle constraint for the *next* variables.

5.2.4.4 Conjunctive constraint

$\forall\ i \in T, j = node_i.parentNode.id,$

$$interval_{i,0} \prec interval_{j,0} \tag{5.25}$$

$$interval_{i,1} \prec interval_{j,1} \tag{5.26}$$

$\forall\ i \in \{k \in T \mid node_k.fixture > 0\ or\ node_k.isOutput\}, j = node_i.parentNode.id,$

$$interval_{i,0} \prec interval_{j,1} \tag{5.27}$$

$$interval_{i,1} \prec interval_{j,0} \tag{5.28}$$

(5.25) - (5.28) post the conjunctive constraint enforcing that each node in the assembly tree should be performed before its parent node. (5.25) and (5.26) post constraint on each single route. (5.27) and (5.28) post constraint on both routes. Since the pick-up nodes, camera nodes and air-gun nodes are always on the same route with their parent nodes, (5.27) and (5.28) are only needed for nodes performed on fixtures and the output node.

Since (5.27) and (5.28) enforce that each node is always performed before its parent node no matter on which route, this also implicitly posts a disjunctive constraint on each fixture which guarantees that the two arms cannot access the same fixture simultaneously.

5.2.4.5 Casting sequence for each route

$\forall\ i \in T, r \in \{0,1\},$

$$sequenceOnRoute_{ir} = next_{sequenceOnRoute_{i-1,r}}$$

$$sequenceOnRoute_{0r} = next_r \tag{5.29}$$

(5.29) cast the $sequenceOnRoute$ array for each route from the *next* variables with element constraint.

The $sequenceOnRoute$ array is described in section 5.2.2. This variable array contains the possible assembly sequences on a route. Regular expression constraints are posted on this array to check the validity of the sequences.

5.2.4.6 Regular expression constraints

The regular expression constraints include capacity constraints for gripper load and suction load, component order constraints, and fixture precedence order constraints.

The $sequenceOnRoute$ array for each route is derived from the next variables, as shown in (5.29).

For $r \in \{0,1\}$, $sequenceOnRoute\ _r = (seq\ _{0r}, seq\ _{1r}, \dots, seq\ _{NumberNodes,r})$ is a sequence indicating which nodes are performed on route r and in what order.

The regular expression constraints are posted on the $sequenceOnRoute$ array for both routes using Deterministic Finite Automaton (DFA). For simplicity, the problem case shown in Figure 5.1 is used to illustrate the DFAs.

Note that the DFAs introduced below are posted on each arm respectively.

The alphabet set of the DFA is defined as

$$\Sigma = I \cup \{dummyNode\}. \tag{5.30}$$

1) Component order constraint

As discussed in section 2.3.3, if an arm picks up a component, the arm should also drop off the component, as well as performing the intermediate nodes between pick-up and drop-off in a precedence order. The component orders are shown in Figure 2.5 with the bold lines.

Take component 3 as an example, the nodes handling on this component are 22, 18, 14 and 10, i.e. pick-up, camera, air-gun and drop-off. Once node 22 appears in $sequenceOnRoute_r$, the sequence (22, 18, 14, 10) should also be part of this sequence in the following precedence order:

$$sequenceOnRoute_r = (r, \dots, 22, \dots, 18, \dots, 14, \dots, 10, \dots, 29), r \in \{0,1\}.$$

Figure 5.3 shows the state diagram of the component order DFA on component 3. The start state and accept state is S_0 and the transition set $A = \Sigma / \{10,14,18,22\}$.

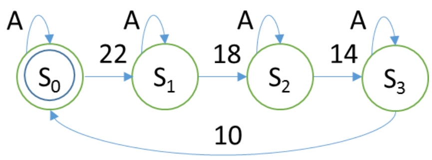


Figure 5.3 Component order DFA of component 3

The component order DFA is posted on component 3 (node 22, 18, 14 and 10) and component 5 (node 16, 12 and 9). For component 1, component 2 and component 4, the component order constraint is taken charged by the fixture precedence order DFA described in the later section.

This constraint is redundant because conjunctive constraint also takes the charge of these orders. However, this constraint is posted for efficiency, as the times for intervals are set at last but the regular expression constraints does propagation earlier in the model.

2) Suction capacity constraint

In this thesis, each arm of the robot contains two suction tools. The suction capacity constraint enforces that when both of the suctions are occupied on one arm, this arm cannot pick up any more components that require suction tool until the suction load is released. Figure 5.4 shows the state diagram of the suction capacity DFA.

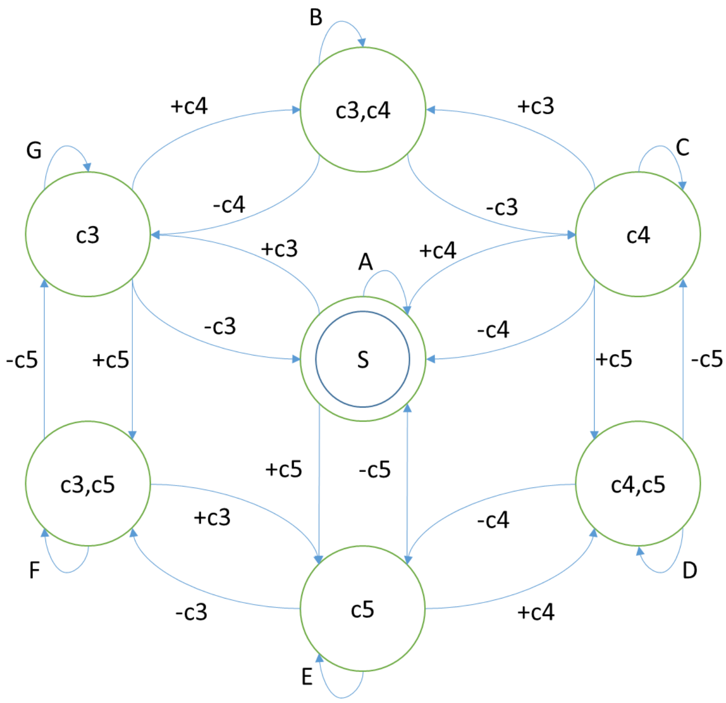


Figure 5.4 Suction capacity DFA

+ci/-ci denotes the operation of picking-up/dropping-off component *i*. Each state indicates which component(s) the arm is holding at this moment. For instance, the state of (c3, c5) represents the arm is currently holding component 3 and component 5. The start state and accept state are the same state where both suction tools of the arm are empty. The self-pointing transition sets are easy to follow according to the assembly scenario. For example, set *A* contains the nodes that do not require any suction tool.

3) Fixture precedence order and gripper capacity constraints

Since each arm only has one gripper, when the gripper is occupied, the arm can neither pick up any more component using gripper, nor perform operations such as tapping and peeling (fixture operation) with gripper. This is enforced by the gripper capacity constraint.

The fixture precedence order constraint enforces that the components on the same fixture should be assembled in the specific precedence order. For example, as shown in Figure 5.1, the precedence order of fixture 1 is (node 24, 20, 17, 13, 10, 8, 6 and 5).

Figure 5.5 shows the state diagram of the DFA implementing the fixture precedence order of fixture 1, the component order on component 1 (node 26, 24) and component 2 (node 21, 17), and the gripper capacity constraint (node 26, 21).

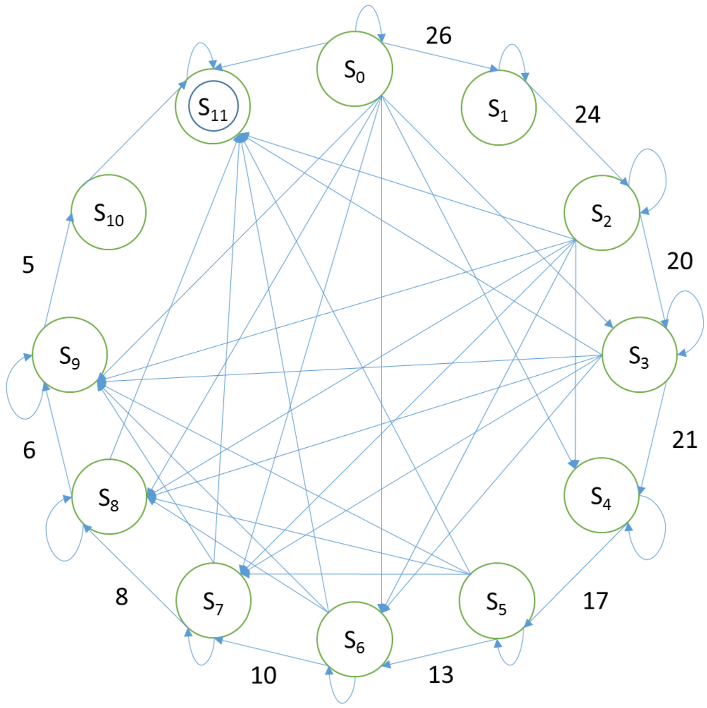


Figure 5.5 The DFA implementing fixture precedence order on fixture 1, component order on component 1 and 2, and gripper capacity constraint

In Figure 5.5, the key transition arcs on the main path are labeled. S_0 is the start state. S_3 represents the state that all tasks on component 1 (node 26, 24, 20) have been accomplished. S_{11} is the accept state in which all nodes assigned to this arm are performed. The transition $S_0 \rightarrow S_{11}$ (transition arcs include 2, 3, 4, 27 and 28) indicates that this arm is not assigned with any tasks preceding node 5 (sub-assembly). Table 5.1 shows the transition arcs from state S_0 to other states in the DFA.

Table 5.1 Transition arcs from S_0 to other states

From state	*To state*	*Transition arcs*
S_0	S_0	7, 9, 11, 12, 14, 15, 16, 18, 19, 22, 23, 25
S_0	S_1	26
S_0	S_2	
S_0	S_3	20
S_0	S_4	21
S_0	S_5	
S_0	S_6	13
S_0	S_7	10
S_0	S_8	8
S_0	S_9	6
S_0	S_{10}	5
S_0	S_{11}	2, 3, 4, 27, 28

For the right sub-tree of Figure 5.1, since there is no component requiring gripper tool, the DFA only implements the fixture precedence order on fixture 2 together with the component order for component 4, which is (node 25, 23, 19, 15, 11, 9, 7, 5, 4, 3 and 2).

There are two reasons to integrate the component order DFA and the gripper capacity DFA into the fixture precedence order DFA. The first reason is that a larger DFA has less redundant arcs than three individual DFAs, hence brings a better performance. The experiment result has confirmed this conclusion. The second reason is that the complexity of integrating the DFAs for the problem case of this thesis is acceptable.

5.2.4.7 Distinct tray location constraint

$$TrayLocations = \{location_i \mid i \in T, node_i.isTray = 1\}$$

$$AllDifferent(TrayLocations) \tag{5.31}$$

(5.31) post the constraint that all components should be placed on distinct trays such that no two components have the same tray location.

5.2.4.8 Avoiding collision constraint

(5.27) and (5.28) implicitly post a disjunctive constraint on each fixture such that two arms cannot access the same fixture location simultaneously. However, it would still be possible that two arms visit the same camera at the same time. This is disallowed.

$\forall\ i, j \in T, i \neq j, node_i.isCamera$ and $node_j.isCamera$,

$$\begin{cases} route_i \neq route_j \leftrightarrow b1 \\ location_i = location_j \leftrightarrow b2 \\ startTime_i < departureTime_j \leftrightarrow b3 \\ startTime_j < departureTime_i \leftrightarrow b4 \\ b5 = b3 * b4 \end{cases}$$

$$b1 + b2 + b5 < 3 \tag{5.32}$$

(5.32) use reification to post the constraint that the two arms cannot visit the same camera location simultaneously. In the reification, $b5$ reflects the validity of the constraint that the intervals of node i and node j overlap, $b1$ reflects the constraint that node i and node j are performed by distinct arms, and $b2$ reflects the constraint that node i and node j have the same location. To avoid collision, these three constraints are not allowed to hold simultaneously.

There is another location collision constraint that if one arm is visiting tray i, the other arm is not allowed to visit camera i at the same time. This is because camera i is used to monitor the action of picking up components from tray i. If the camera is occupied, the task of tray picking-up cannot be performed correctly.

The $TrayIndex(location)$ and $CameraIndex(location)$ denote the tray index and camera index of the given location respectively.

$\forall\ i, j \in T, i \neq j, node_i.isTray$ and $node_j.isCamera$,

$$\begin{cases} route_i \neq route_j \leftrightarrow b1 \\ TrayIndex(location_i) = CameraIndex(location_j) \leftrightarrow b2 \\ startTime_i < departureTime_j \leftrightarrow b3 \\ startTime_j < departureTime_i \leftrightarrow b4 \\ b5 = b3 * b4 \end{cases}$$

$$b1 + b2 + b5 < 3 \tag{5.33}$$

(5.33) post the constraint to avoid the two arms visiting the tray and camera with same index simultaneously.

5.2.4.9 Compact-fixtures constraint

In some real-world assembly cases, the two fixtures in the assembly cell could be too close to each other such that the two arms are not allowed to visit the fixture area simultaneously. In this situation, if one arm is visiting a fixture, the other arm cannot access any fixtures until the former arm leaves the fixture area.

$$Intervals = \{interval_{ir} \mid node_i.fixture > 0, r \in \{0,1\}\}$$

$$Disjunctive(Intervals) \tag{5.34}$$

(5.34) post disjunctive constraint on the interval variables of fixture nodes on both routes to enforce that only one arm can access the fixture area at a time. This is an optional constraint which is only posted in some special situations.

When generating the travel time matrix between locations, we deliberately design the data so that each arm has some unreachable locations. In this way, the situation that two arms cross with each other during assembly is avoided.

5.2.5 Objective function

The objective function is to minimize the makespan of a single assembly. As shown in Figure 5.6, the two arms both start from time 0. The makespan is equal to the departure time of the last node in the assembly sequence.

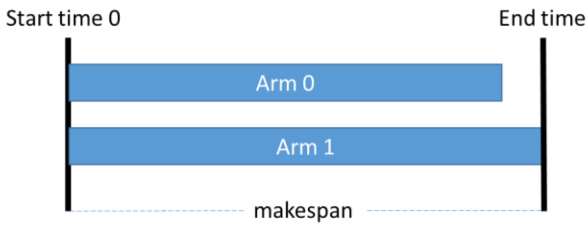


Figure 5.6 Makespan of a single assembly

The objective variable is defined as follows,

$$makespan = \max(\{departureTime_i \mid i \in I\}). \tag{5.35}$$

5.3 Search strategy

In this thesis, we are interested in finding and proving the global optimal solution for the problem case. This is because in the lean production environment, even a little improvement in the cycle time can significantly increase the total throughput of the assembly line. In addition, the computational time for proving the optimality to the problem case in this thesis is less than an hour and completely acceptable.

The search strategy is based on systematic tree search. As discussed in section 3.6, systematic search is a good choice to find the provably optimal solution to a constraint optimization problem. The decision variables to branch on are *Route*, *Next*, *Location*, and *makespan*.

5.3.1 Structure of Search Tree

The first variable to branch on is the *Route* variables. The reason to put *Route* in the first place is because for Vehicle Routing Problem, if the customers are assigned to the vehicles, the problem becomes to build a route for each vehicle. This is basically a Travelling Salesman Problem and comparatively easier than VRP problem.

The second variable to branch on is the *Next* variables, and the *Location* variables are branched on after *Next*. This is because according to the experiment, branching on *Location* before *Next* brings a worse performance.

The last variable to branch on is *makespan* which is the objective variable. Recall from (5.35) that *makespan* is equals to the maximum value of the departure time of all nodes. At this stage, the sequence and locations are set. The setting of *makespan* will give the best possible resolving of the remaining time-window-conflicts, which is basically equivalent to solving a constrained Job Shop Problem.

The problem in this thesis is a Constraint Optimization Problem with the aim of finding the optimal solution and proving its optimality, thus requires a traversal of the entire search tree. The idea of the search strategy is to find the optimal solution as soon as possible, and then prune the rest branches of which the objective value bound is worse than the objective value of optimal solution, as a result reducing the average depth of branches in the search tree.

5.3.2 Built-in Search vs. Customized Search

Two types of search strategy are used in this thesis: the built-in search strategy of the support CP library, and the customized search strategy specific to the problem case.

The built-in search strategy is applicable to general problems and handy to use. Users do not need to consider the implementation details. However, when we talk about performance and stability, the built-in strategy is not adequate. For a certain problem case, different parameter settings usually make the performance of built-in search strategy differ a lot. Due to this reason, the users have to frequently change the parameter setting for different problem cases in order to get the best search performance.

An important advantage of customized search strategy is that there is no need to change the parameter settings from case to case. Customized strategy is based on heuristics of given problems and usually provides a better performance than built-in search strategy.

5.3.3 Built-in Search Strategy

Two built-in variable selection strategies are used in this thesis:

CHOOSE_FIRST_UNBOUND

This strategy selects the first unbound variable. Variables are ordered by the indices of the variable collection passed to the selector.

CHOOSE_PATH

This is a dedicated strategy for building simple paths (paths with no repeated vertices). It is described as follows [55]:

1. Try to extend an existing path. The strategy seeks the existing path by looking for an unbound variable to which some other variable is pointing.
2. If there is no existing path, try to find a start node of a path to begin. To find a start node, the strategy looks for an unbound variable to which no other variable points.
3. If both rule 1 and rule 2 fail, pick the first unbound variable.

Here is an example to illustrate how the CHOOSE_PATH strategy works. Suppose the variables to be ordered are $next_0, next_1, \dots, next_5$, and a path exists among the variables, as shown in Figure 5.7.

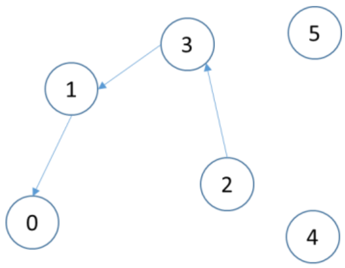


Figure 5.7 An example demonstrates CHOOSE_PATH strategy

The variable array $next = (-,0,3,1,-,-)$ where the symbol "-" means the variable at this position is unassigned. The existing path in this figure is $2 \rightarrow 3 \rightarrow 1 \rightarrow 0$. Based on CHOOSE_PATH strategy, the next variable to select is $next_0$ and the domain of this variable is {2, 4, 5}. If $next_0$ is assigned with the value 2, the path is closed, and the next variable to select will be another unbound variable which is 4 or 5.

The built-in value selection strategies used in this thesis are ASSIGN_MIN_VALUE and ASSIGN_MAX_VALUE.

ASSIGN_MIN_VALUE

This strategy selects the minimum value in the domain of the selected variable.

ASSIGN_MAX_VALUE

This strategy selects the maximum value in the domain of the selected variable.

5.3.4 Customized Search Strategy

5.3.4.1 Branching Route

Route is the most important variable among the four variables to branch on because it is on the top of the entire search tree. The search under an assignment of *Route* consists of the sub search tree of *Next*, *Location* and *makespan*, which is a large search space. If we are able to find a good or even optimal assignment to *Route* at the early stage, the best objective value under this assignment will propagate the rest branches with worse bounds of objective value, thus significantly reduce the entire search space.

Variable selection

The *Route* variables are ordered by the length of the *InvalidPrev* list of each node.

Let $variableOrdering(Route_i)$ denotes the index of $Route_i$ in the variable ordering,

$$node_i.InvalidPrev.length > node_j.Invalid.length \Rightarrow variableOrdering(Route_i) < variableOrdering(Route_j) \quad (5.36)$$

$$node_i.InvalidPrev.length = node_j.Invalid.length, and\ i < j \Rightarrow variableOrdering(Route_i) < variableOrdering(Route_j) \quad (5.37)$$

As shown in (5.36) and (5.37), if a node has a longer *InvalidPrev* list, its route variable will be branched on earlier. If two nodes have the same length of *InvalidPrev* list, the node with smaller index will be branched on first.

(5.36) and (5.37) reflect a general order of performing the assembly sequence. As seen in Figure 5.1, the node with longer *InvalidPrev* list is closer to the leaf of the assembly tree, and generally should be performed earlier in the sequence. To branch on the *Route* variables in this order conforms to the assembly scenario.

Value selection

The value selecting strategy for Route is to balance the total durations of both routes. In addition, a heuristics named "*Fixture Succession*" is used.

Suppose $Route_i$ is unbound and to be assigned. The procedure is described as follows:

1. If node *i* is a fixture operation node, perform *Fixture Succession* heuristics.

   The *Fixture Succession* heuristics works like this. Let *prevNode* denote a child of node *i* which is drop-off or fixture operation or sub-assembly. If the route of *prevNode* is bound, assign $Route_i$ with the same route as *prevNode*.

2. If rule 1 is not applicable, perform the balance heuristics.

In rule 1, the *prevNode* of node i could be drop-off or fixture operation or sub-assembly. For example, as seen in Figure 5.1, the *prevNode* of node 20 is 24, the *prevNode* of node 11 is 15, and the *prevNode* of node 4 is 5. Assign a fixture operation node with the same route as its *prevNode* is because the two nodes are at the same location (on the same fixture), to perform the fixture operation node immediately after its *prevNode* could save travel time.

Let $NodesOnRoute_r$ denote the set of nodes that have already been assigned to route r so far, $accDurOnRoute_r$ denodes the current cumulative durations on route r. The balance heuristics is described as follows:

1. Calculate the current cumulative durations on each route:

   $NodesOnRoute_r = \{node_i \mid i \in T, node_i.route = r\}$

   $accDurOnRoute_r = \sum_{node \in NodesOnRoute_r} node.durationRoute_r$

2. If $accDurOnRoute_0 \neq accDurOnRoute_1$, assign $Route_i$ with route r of which the $accDurOnRoute_r$ is smaller.

3. If $accDurOnRoute_0 = accDurOnRoute_1$, assign node i to the arm which performs this node in a shorter duration, i.e.

   if $node_i.durationRoute_0 \leq node_i.durationRoute_1$, assign 0 to $Route_i$,

   else if $node_i.durationRoute_0 > Nodes_i.durationRoute_1$, assign 1 to $Route_i$.

Balancing is an important feature of the optimal solution. With the effort of evenly allocating workload to both arms, the total makespan could be reduced. In balance heuristics, the *Route* variables are always assigned values according to the run-time cumulative durations on both arms.

5.3.4.2 Branching Next

After the *Route* variables are bound, branching on *Next* is basically to create a simple path for each route.

Variable selection

The variable selection strategy for *Next* is CHOOSE_PATH. As discussed in section 5.3.3, it is a dedicated strategy for building simple path.

Value selection

The values in the domain of $Next_i$ are the indices of the candidate successors of node i. Basically, the value ordering for a variable is to give a priority to each value in its domain. The value with higher priority will be set to the variable earlier in the search.

The value selection strategy for *Next* is based on two heuristics: the length of the *InvalidPrev* list of the candidate successors and a similar *Fixture Succession* heuristics as that of the *Route* variables.

Suppose $Next_i$ is unbound and to be assigned, the value ordering heuristic is described as follows:

1. If node i is drop-off or sub-assembly,
   - $succNode = node_i.parentNode$ is fixture operation node, as shown in Figure 5.1. Give the index of $succNode$ the highest priority among all values in the domain of $Next_i$.
   - For the rest values in the domain, give higher priority to the index of the node with longer *InvalidPrev* list.
2. If node i is neither drop-off nor sub-assembly, give higher priority to the index of the node with longer *InvalidPrev* list.

The idea of rule 1 is similar to the *Fixture Succession* heuristics of *Route*. Since a drop-off node and its *succNode* are at the same location (on the same fixture), to process the *succNode* immediately after finishing this drop-off node will save travel time. The reason of giving higher priority to the node with longer *InvalidPrev* list is because according to the assembly tree in Figure 5.1, the general trend of the assembly is from leaf to root. The node with longer *InvalidPrev* list is closer to the leaf of the assembly tree, thus should be performed earlier in the sequence in general.

5.3.4.3 Branching Location

At this stage, the *Route* and *Next* variables are assigned values. The next step is to determine the locations of tray nodes and camera nodes. The idea of branching on the *Location* variables is based on greedy strategy, as well as keeping the balancing feature for both routes.

Variable selection

To keep balance on both routes, the tray locations and camera locations are branched on both routes simultaneously. The variable ordering strategy is described as follows:

1. Calculate current cumulative time on both routes. The cumulative time includes the total durations and the total travel time between the nodes with bound locations (fixture, air-gun, output, as well as tray and camera nodes that are assigned locations). For the route with longer cumulative time, select the first unbound location on this route.
2. Select the first unbound location on the other route other than in step 1.

3. Repeat step 1 and step 2 until one route is exhausted. Then exhaust the rest route.

Value selection

The value ordering for *Location* is based on greedy strategy [56]. The greedy strategy means always selecting the location value that gives the shortest distance for each step along the route.

Suppose $Location_i$ is unbound and to be assigned. Let *prevLocation* denote the location of the predecessor of node *i*. Since we start from the first unbound location on a route and the prevLocation is always fixed or assigned along this route. The strategy is described as follows:

1. If the predecessor of node *i* is a start node, calculate the minimum distance between the possible locations of node *i* and the possible location(s) of its successor (the location of the successor could be bound or unbound). Assign $Location_i$ with the location value that leads to the minimum distance.
2. If the predecessor of node *i* is not a start node, calculate the minimum distance between *prevLocation* and the possible locations of node *i*. Assign $Location_i$ with the location value that leads to the minimum distance.

The reason of rule 1 is because the distance between a start node and any other node is always 0. To avoid blind selection, the minimum distance between current node and its successor is considered.

5.3.4.4 Branching makespan

The branching strategy for *makespan* is quite simple. Since there is only one variable, no variable selection strategy is needed. The value selection strategy is to assign the variable with the minimum value in its domain. Recall from (5.35) that makespan is equal to the departure time of the last node in the assembly sequence. Assigning *makespan* with the minimum value will tighten the entire sequence.

5.4 Implementation

The constraint programming support library used in this thesis is Google OR-Tools [55] with repository of version 3756. Google OR-Tools is an open source project developed by Google under the Apache License 2.0. The library contains a constraint programming solver, an interface to several linear programming and mixed integer programming solvers, as well as a set of dedicated algorithms such as Knapsack and graph algorithms.

The library is written in C++ but is also available in Python, Java, and .NET through SWIG [57] which is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. In this thesis, we use the .NET version of the library.

Google OR-Tools contains a set of global constraints including the constraints described in section 3.5. The library also implements a couple of search algorithms, such as systematic tree search, local search, and Large Neighborhood Search. The library allows combining sequential searches in the order defined by users. At each leaf of the current search tree, the next search tree is created.

In addition to the implemented constraints and search strategies, Google OR-Tools offers full support for implementing customized constraints and search strategies.

6 Results

In this chapter, the experiment settings will be introduced first. Then the results for 5 test cases will be presented.

6.1 Experiment Settings

The problem case in this thesis is the same as that in the previous thesis work [8] regarding the task durations and travel time matrix, but with a slightly updated assembly tree which makes the problem case a little larger. The detail of the problem case is described in Chapter 2. The task durations and travel time matrix are collected by simulating each operation and movement with both arms in Robot Studio.

The experiment is run on a computer with Windows 7 64-bit operating system, Intel Core i7 2.7 GHz CPU, and 8 GB RAM. The CP library is Google OR-Tools of version 3756 for C#. In each test, the program is executed to solve a problem instance and all solutions are recorded including the run-time of finding each solution. In addition, the information of finishing the entire search tree is recorded at the end of each test, including the run-time, branches and failures.

To compare the efficiency of the built-in search strategy and the customized search strategy, we run the programs based on each search strategy under the same problem instance. The run-time of finding the optimal solution and run-time of finishing the whole search, i.e. proving the optimality are two major indicator for comparison.

The evaluation is based on four general test cases and a special case. The only difference between the general case and the special case is that the special case is under the additional compact-fixture constraint described in Section 5.2.4.9. Under this constraint, the two arms are not allowed to access the fixture area simultaneously. Case 1 to case 4 share the same task durations and travel time matrix, but with different assembly trees. The assembly tree of case 2, case 3 and case 4 are shown in Appendix.

Table 6.1 shows the settings of each test case. Case 1 to case 4 is general case and case 5 is the special case.

Table 6.1 Settings of test cases

	Fixtures compact	*Number of components*	*Tool usage for components*
Case 1	no	5	2 grippers, 3 suctions
Case 2	no	4	1 gripper, 3 suctions
Case 3	no	4	2 grippers, 2 suctions
Case 4	no	3	1 gripper, 2 suctions
Case 5	yes	5	2 grippers, 3 suctions

For the built-in search strategy, we test all possible combination of parameter settings to find the configuration with best performance. The variable selection strategy for *Route* is FIRST_UNBOUND and the value selection strategies are ASSIGN_MAX_VALUE and ASSIGN_MIN_VALUE.

For the *Next* variables, the variable selection strategy is CHOOSE_PATH. The value selection strategy is ASSIGN_MAX_VALUE. The reason not to consider ASSIGN_MIN_VALUE is because when building a route regarding the assembly tree, the general trend of the route is always from the leaf (nodes with larger indices) to the root (nodes with smaller indices) in the assembly tree. Apparently, ASSIGN_MIN_VALUE is not an appropriate strategy.

The variable selection strategy for *Location* is FIRST_UNBOUND and the value selection strategies are ASSIGN_MAX_VALUE and ASSIGN_MIN_VALUE. For *makespan* no variable selection strategy is needed, and the value selection strategy is ASSIGN_MIN_VALUE.

Table 6.2 shows the four settings of value selection strategy for the *Route* and *Location* variables.

Table 6.2 Value selection settings for *Route* and *Location*

	Route	***Location***
Setting 1	MAX	MAX
Setting 2	MAX	MIN
Setting 3	MIN	MAX
Setting 4	MIN	MIN

As shown in the table, MAX denotes the strategy of ASSIGN_MAX_VALUE and MIN denotes ASSIGN_MIN_VALUE.

6.2 Experiment Results

The experiment results showed that the CP model in this thesis has made a great improvement over the model in the previous work [8]. For the same problem instance and under similar hardware and software environment to that of the previous work, the model and implementation in this thesis can find the optimal solution in less than 20 minutes and prove the optimality in around 40 minutes. However, the previous work can only find an approximately optimal solution but not able to prove the optimality.

In the following, we will compare the performance of the customized search strategy with the built-in search strategy under five test cases listed in Table 6.1.

6.2.1 Test on case 1

Case 1 is the general problem case with 5 components which is described in Chapter 2. The assembly tree of case 1 is shown in Figure 2.5 and Figure 5.1. Table 6.3 shows the run-time, branches and failures for finding optimal solution with the built-in strategy settings and the customized strategy.

Table 6.3 Time, branches and failures of finding the optimal solution for case 1

	Time (s)	*Branches*	*Failures*
Setting 1	1085	1557675	778844
Setting 2	1080	1562427	781223
Setting 3	9856	9003944	4501990
Setting 4	9569	9002333	4501224
Customized	243	391237	195613

Table 6.4 shows the run-time, branches and failures for proving the optimality i.e. completing the entire search.

Table 6.4 Time, branches and failures of proving the optimality for case 1

	Time (s)	*Branches*	*Failures*
Setting 1	2470	3611374	1805704
Setting 2	2466	3617326	1808683
Setting 3	11296	11174880	5587468
Setting 4	10991	11175249	5587692
Customized	1957	3036286	1518150

Figure 6.1 shows the performance comparison of each search strategy on case 1. The vertical coordinate represents run-time. Grey rectangle denotes the run-time for finding optimal solution, and black rectangle denotes the run-time for proving optimality.

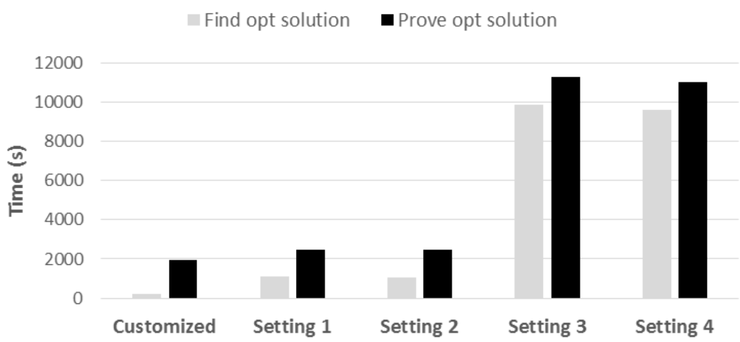


Figure 6.1 Performance comparison on case 1

As shown in Figure 6.1, the best setting for built-in strategy is setting 2. The customized strategy has a better performance than setting 2. Specifically, the customized strategy only took 22.22% of the time for finding optimal solution and 79.33% of time for proving the optimality compared to setting 2.

The diagram also shows that the performance of setting 1 and setting 2 are close to each other, and the same with setting 3 and setting 4. In another word, difference search strategies for *Location* do not bring a considerable influence on the overall performance. This is because the search on *Location* is the last part of the whole search, and its search space is fairly small compared with *Next* and *Route*. Therefore in the following test, we only consider setting 1 and setting 3 for the built-in strategy.

Figure 6.2 shows the track of finding optimal solution with customized strategy and with the best built-in strategy.

Figure 6.2 The track of finding optimal solution with customized and best built-in strategy

In this diagram, each point labeled on the curves represents a solution. The value of horizontal coordinate is the elapsed search time of finding the solution, and the value of the vertical coordinate is the objective value of this solution.

In Figure 6.3, the horizontal coordinate is displayed on logarithmic scale so that we can see more details in the beginning stage of the search.

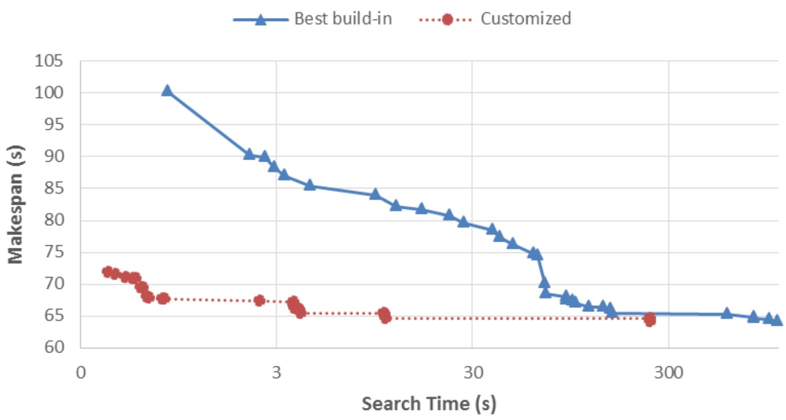


Figure 6.3 Logarithmic scale of horizontal coordinate in Figure 6.2

Figure 6.2 and Figure 6.3 show that the customized strategy initially finds solutions with much better objective values, thus reach the optimal solution faster.

6.2.2 Test on case 2 and 3

Case 2 and case 3 are both 4 component case but with different tool usage and assembly tree. In case 2, one component requires gripper tool and three components require suction tool. In this situation, the gripper capacity constraint and the suction capacity constraint both have effects. On the contrary, case 3 contains two components for gripper and two components for suction, and in this case the suction capacity constraint has no effect but the gripper capacity constraint has a big effect.

Test on case 2

Table 6.5 shows the performance of the customized strategy and the built-in strategy with setting 1 and setting 3 on case 2.

Table 6.5 Performance comparison on case 2

	Time for finding opt solution (s)	*Time for proving opt solution (s)*
Customized	51	65
Setting 1	56	77
Setting 3	2791	2808

The table shows that the customized strategy and setting 1 have much better performance than setting 3. Furthermore, customized strategy performs slightly better than setting 1.

Test on case 3

Table 6.6 shows the performance of the search strategies on case 3. For built-in strategy with setting 3, we ran the program for more than 5 hours but did not find any solution. Therefore, here gives the lower bound of the run-time.

Table 6.6 Performance comparison on case 3

	Time for finding opt solution (s)	*Time for proving opt solution (s)*
Customized	0.6	6.8
Setting 1	3191	3197
Setting 3	> 18000	> 18000

Table 6.6 shows that customized strategy has much better performance than built-in strategy under case 3. There are two reasons for the significant advantage. The first reason is because the constraint on the objective value has an extreme effect on this case. After reaching the optimal solution, the objective value of the optimal solution can propagate the rest of the search space in a very short time. As seen in table 6.6, both customized strategy and built-in strategy with setting 1 proved the optimality in 6 seconds after finding the optimal solution. The second reason is that the solutions clustered very intensively in the search space. We notice that setting 1 found the first solution after 3178 seconds and setting 3 found no solution over 5 hours, as a result, no objective value is collected to prune any branches in the search tree. On the contrary, the customized strategy happened to find the optimal solution at beginning with some luck.

6.2.3 Test on case 4

Case 4 contains one gripper picking-up component and two suction picking-up components. Apparently, neither of the gripper nor suction capacity constraint has effect on this case.

Table 6.7 shows the performance of the search strategies on case 4.

Table 6.7 Performance comparison on case 4

	Time for finding opt solution (s)	*Time for proving opt solution (s)*
Customized	3.23	3.72
Setting 1	2.56	4.05
Setting 3	21.62	23.46

From this table we can see that customized strategy and setting 1 have better performance than setting 3. Setting 1 took shorter time for finding the optimal solution, and customized strategy proved the optimality faster.

6.2.4 Test on case 5

Case 5 is the special 5 component case in which the two arms are not allowed to access the fixture area simultaneously. This is achieved by posting an additional constraint described in Section 5.2.4.9. Since this constraint affects the concurrency of the operations on two fixtures, the objective value of the optimal solution to case 5 is a little larger than that of case 1. The performance of each strategy is shown in Figure 6.4.

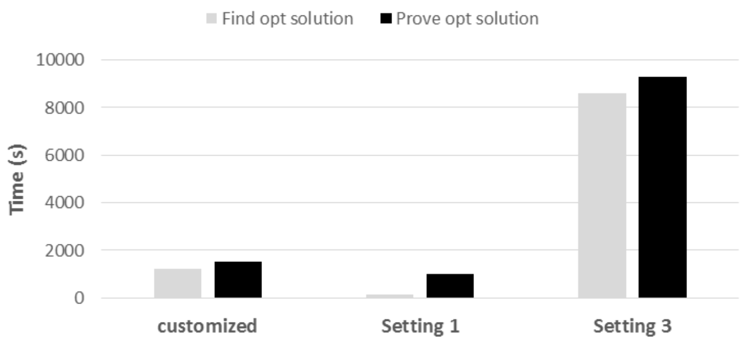


Figure 6.4 Performance comparison on case 5

In this test, setting 1 has the best performance in both finding optimal solution and proving the optimality. Although the customized strategy did not win out, its performance is much better than setting 3.

6.3 Summary

It needs to be stressed that setting 1 is not necessarily the best configuration for built-in strategy in all problem cases. In some other cases with different task durations, travel time matrix, and/or assembly trees, setting 3 would give the best performance for built-in strategy.

Actually, the only difference between setting 1 and setting 3 is the value selection strategy for *Route*. For setting 1, all nodes are assigned to arm 1 at beginning, and then the route of each node alters iteratively until finding the optimal solution. For setting 3, all nodes are initially assigned to arm 0. Basically, setting 1 and setting 3 define two symmetrical directions in the search space of *Route*. Hence, setting 3 should be the best setting for about half of the problem cases.

In practice, the parameter setting for built-in strategy always needs to change from case to case in order to get good performance. However, since the search strategy for

Route in customized strategy is based on task durations and assembly tree which are intrinsic properties of the problem case, customized strategy performs consistently for all problem cases.

7 Conclusion

In this thesis a real-world assembly case for dual arm robot is introduced. A constraint programming model and a customized search strategy for optimizing the assembly sequence and assembly cell layout of the problem case is proposed. The model and search strategy are implemented using Google OR-Tools, an open source constraint programming support library. The implementation is tested under a set of problem instances sharing the same task durations and travel time matrix but with different assembly tree and tool usage of components.

The results show that the model and search strategy are effective and efficient on the problem case studied in this thesis. For the general 5 component test case in the experiment, the implementation can find optimal solution with 4 minutes and prove the optimality with 32 minutes. The results also show that with the increase of the problem size, the run-time grows fast. However, since the common size of the real-world assembly case is similar to the 5 component case, the time for solving the problem is acceptable.

A performance comparison between the customized search strategy and the built-in search strategy of Google OR-Tools is done. The results show that the customized strategy has a better performance than the built-in strategy in many cases. For the general 5 component case, the customized strategy only takes 22.22% of the time for finding optimal solution and 79.33% of the time for proving optimality compared to built-in strategy with the best setting. For the general test cases with 4 components and 3 components, customized strategy also shows a performance advantage over built-in strategy.

The results also indicate that the parameter setting for built-in strategy is highly relevant to the problem case, therefore needs to change frequently according to different cases. However, the customized strategy performs consistently in all problem cases, thus no need to make changes.

8 Discussion & Future Work

In this chapter, a further discussion on the CP model and the customized search strategy will be conducted, and some future work is suggested.

An important part in the model is to post regular expression constraints on each route to eliminate invalid assembly sequences. The regular expression constraints are very efficient in propagation. When building these constraints, the integration of smaller DFAs to a larger DFA often brings improvements in performance, but meanwhile increases the implementation complexity. Therefore, a good balance between performance and complexity need to be determined.

Currently, the information of assembly tree and task durations is used for the customized search strategy. The strategy of balancing the total durations on both routes plays an important role. Since the makespan on each route consists of total durations and total travel time, it is advisable to find a way to utilize the travel time matrix in search heuristics.

Instead of manually generating travel time matrix from Robot Studio, a program can be developed to automatically generate the travel time from dynamic case models.

The travel time used now is the average travel time between two locations. To get more accurate travel time, more configurations for the position of an arm could be added into the model. With the increase of freedom in the model, searching for optimal solution might become infeasible. Then the non-systematic search methods such as local search and large neighborhood search are needed to get the approximately optimal solutions.

References

[1] ABB. Introduction of ABB. [Online]. http://new.abb.com. Accessed on: 2015-07-06.

[2] Robotics Industries Association. Robotic Assembly: Shrinking Footprint, Expanding. [Online]. http://www.robotics.org/content-detail.cfm/Industrial-Robotics-Industry-Insights/Robot. Accessed on: 2015-07-06.

[3] ABB. ABB Robotics. [Online]. http://www.abb.com/product/ap/seitp327/973e9073ef5d2787c12570b9003563c7.aspx. Accessed on: 2015-07-06.

[4] ABB. YuMi Robot. [Online]. http://new.abb.com/products/robotics/yumi. Accessed on: 2015-07-06.

[5] Matthias Holweg, "The Genealogy of Lean Production," *Journal of Operations Management*, vol. 25, no. 2, pp. 420-437, 2007.

[6] Xinhuanet. Foxconn to replace workers with 1 million robots in 3 years. [Online]. http://news.xinhuanet.com/english2010/china/2011-07/30/c_131018764.htm. Accessed on: 2015-07-06.

[7] ISO 8373: 1994, Robots and robotic devices - Vocabulary.

[8] Joakim Ejenstam, "Implementing A Time Optimal Task Sequence For Robot Assembly Using Constraint Programming," Uppsala University, Uppsala, Master Thesis ISSN 1401-5749, 2014.

[9] ABB. ABB Robot Studio. [Online]. http://www.abb.com/product/seitp327/30450ba8a4430bcfc125727d004987be.aspx. Accessed on: 2015-07-06.

[10] Eugene C. Freuder, "In pursuit of the holy grail," *Constraints*, vol. 2, no. 1, pp. 57-61, 1997.

[11] Francesca Rossi, Peter Van Beek, and Toby Walsh, Eds., *Handbook of Constraint Programming*.: Elsevier Science, 2006.

[12] Krzysztof Apt., *Principles of Constraint Programming*. New York, USA: Cambridge University Press, 2003.

[13] Nicolas Beldiceanu, Mats Carlsson, and Sophie Demassey. Global Constraint Catalogue. [Online]. http://sofdem.github.io/gccat/gccat/index.html.

Accessed on: 2015-07-06.

[14] J. L. Lauriere, "A Language and a Program for Stating and Solving Combinatorial Problems," *Artificial Intelligence*, vol. 1, no. 10, pp. 29-127, 1978.

[15] P. Van Hentenryck and J. P. Carillon, "Generality vs. Specificity: an Experience with AI and OR Techniques," in *National Conference on Artificial Intelligence (AAAI-88)*, 1988.

[16] J. Carlier, "The one-machine sequencing problem," *European Journal of Operational Research*, vol. 11, pp. 42-47, 1982.

[17] P. Refalo, "Linear Formulation of Constraint Programming Models and Hybrid Solvers," in *Principles and Practice of Constraint Programming (CP'2000) LNCS 1894*.: Springer-Verlag, 2000.

[18] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling and Programming with Gecode. [Online]. http://www.gecode.org/doc-latest/MPG.pdf. Accessed on: 2015-07-06.

[19] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages,and Computation*, 2nd ed.: Addison Wesley, 2001.

[20] Ruslan Mitkov, *The Oxford Handbook of Computational Linguistics*.: Oxford University Press, 2003.

[21] G. B. Dantzig and J. H. Ramer, "The Truck Dispathcing Problem," *Management Science*, vol. 6, no. 1, pp. 80-91, Oct. 1959.

[22] G. Gutin and A. Punnen, *The Traveling Salesman Problems and its Variations*. Dordrecht, Netherlands: Kluwer Academic Publishers, 2002.

[23] M. Junger, G. Reinelt, and G. Rinaldi, "The traveling salesman problem," in *Annotated Bibliographies in Combinatorial Optimization (Wiley Interscience Series in Discrete Mathematics)*.: John Wiley & Sons, 1997.

[24] G. Laporte, "The traveling salesman problem: an overview of exact and approximate algorithms," *European Journal of Operational Research*, vol. 59, pp. 231-247, 1992.

[25] Yves Caseau and Francois Laburthe, "Solving small TSPs with constraints," in *Proceedings of the 14th International Conference on Logic Programming*, 1997, pp. 316-330.

[26] Paolo Toth and Daniele Vigo, Eds., *The Vehicle Routing Problem*.: Discrete Mathematics and Applications, 2002.

[27] The blog of Logistics at MGEPS at UPV. Vehicle Routing Problem: It's There A Perfect Solution? [Online]. https://logisticsmgepsupv.wordpress.com/2014/04/10/vehicle-routing-problem-its-there-a-perfect-solution/comment-page-1/. Accessed on: 2015-07-06.

[28] Q. Lu and M. Dessouky, "An exact algorithm for the multiple vehicle pickup and delivery problem," *Transportation Science*, vol. 38, no. 4, p. 503, 2004.

[29] Sophie N. Parragh, Karl F. Doerner, and Richard F. Hartl, "A survey on pickup and delivery problems," *Journal für Betriebswirtschaft*, vol. 58, no. 2, pp. 81-117, 2008.

[30] O. Braysy and M. Gendreau, "Vehicle routing problem with time windows, part Ⅰ: Route construction and local search algorithms," *Transportation Science*, vol. 39, no. 1, pp. 104-118, 2005.

[31] O. Braysy and M. Gendreau, "Vehicle routing problem with time windows, part Ⅱ: Metaheuristics," *Transportation Science*, vol. 39, no. 1, pp. 104-118, 2005.

[32] M. Savelsbergh, "Local search in routing problems with time windows," *Annals of Operations Research*, vol. 4, pp. 285-305, 1985.

[33] B. L. Golden, S. Raghavan, and E. A. Wasil, Eds., *The Vehicle Routing Problem: Latest Advances and New Challenges*.: Springer-Verlag, 2008.

[34] Philip Kilby and Paul Shaw, "Vehicle Routing," in *Handbook of Constraint Programming*, F. Rossi, P. V. Beek, and T. Walsh, Eds.: Elsevier Science, 2006, pp. 799-834.

[35] Bruno De Backer, Vincent Furnon, Paul Shaw, Philip Kilby, and Patrick Prosser, "Solving vehicle routing problems using constraint programming and metaheuristics," *Journal of Heuristics*, vol. 6, no. 4, pp. 501-523, 2000.

[36] Judea Peal, *Heuristics: intelligent search strategies for computer problem solving*.: Addison Wesley, 1984.

[37] G. Clarke and J. Wright, "Scheduling of vehicles from a central depot to a number of delivery points," *Operations Research*, vol. 12, pp. 568-581, 1964.

[38] B. E. Gillett and L. R. Miller, "A heuristic algorithm for the vehicle dispatch

problem," *Operations Research*, vol. 22, pp. 340-349.

[39] Philip Kilby, Patrick Prosser, and Paul Shaw, "Guided local search for the vehicle routing problem with time windows," in *Meta-Heuristics*.: Springer US, 1999, pp. 473-486.

[40] Bruno De Backer, Vincent Furnon, P. Kilby, P. Prosser, and Paul Shaw, "Local search in constraint programming: Application to the vehicle routing problem," in *Proc. CP-97 Workshop Indust. Constraint-Directed Scheduling*.: Schloss Hagenberg, 1997, pp. 1-15.

[41] E. Aarts, J. H. M. Korst, and P.J.M. Van Laarhoven, "Simulated annealing," in *Local Search in Combinatorial Optimization*. Chichester, UK: John Wiley & Sons, 1997, pp. 91-120.

[42] F. Glover, "Tabu search, part Ⅰ," *ORSA Journal on Computing*, vol. 1, no. 3, pp. 190-206, 1989.

[43] F. Glover, "Tabu search, part Ⅱ," *ORSA Journal on Computing*, vol. 2, no. 1, pp. 4-32, 1990.

[44] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*.: Addison Wesley, 1989.

[45] P. Hansen and N. Mladenovic, "Variable neighborhood search: Principles and applications," *European Journal of Operational Research*, vol. 130, no. 3, pp. 449-467, 2001.

[46] Pascal Van Hentenryck and Laurent Michel, *Constraint-Based Local Search*.: MIT Press, 2009.

[47] Philip kilby and Andrew Verden, "Flexible routing combing constraint programming, large neighbourhood search, and feature-based insertion," in *Principles and Practice of Constraint Programming – CP96*, 2011, pp. 43-49.

[48] Paul Shaw, "Using constraint programming and local search methods to solve vehicle routing problems," in *Fourth International Conference on Principles and Practice of Constraint Programming - CP98*.: Springer-Verlag, 1998.

[49] David Pisinger and Stefan Ropke, "Large Neighborhood Search," in *Handbook of Metaheuristics*.: Springer-Verlag, 2010, pp. 399-419.

[50] J. C. Beck, T. K. Feng, and J. P. Watson, "Combining Constraint Programming and Local Search for Job-Shop Scheduling," *INFORMS Journal on Computing*,

vol. 23, no. 1, pp. 1-14, 2011.

[51] Manuel Loth, Michele Sebag, Youssef Hamadi, Marc Schoenauer, and Christian Schulte, "Hybridizing Constraint Programming and Monte-Carlo Tree Search: Application to the Job Shop Problem," in *LION7 - Learning and Intelligent Optimization*.: Springer-Verlag, 2013, pp. 315-320.

[52] J. Beck, P. Prosser, and E. Selensky, "Vehicle routing and job shop scheduling: What's the difference?," in *13th International Conference on Automated Planning and Scheduling ICAPS'03*, 2003.

[53] B. Kallehauge, J. Larsen, and O. B. Madsen, "Lagrangean duality applied on vehicle routing with time windows," Technical University of Denmark, Lyngby – Denmark, Technical Report IMM-TR-2001-9 2001.

[54] Google. Google OR-Tools reference manual. [Online]. http://or-tools.googlecode.com/svn/trunk/documentation/reference_manual/or-tools/. Accessed on: 2015-07-06.

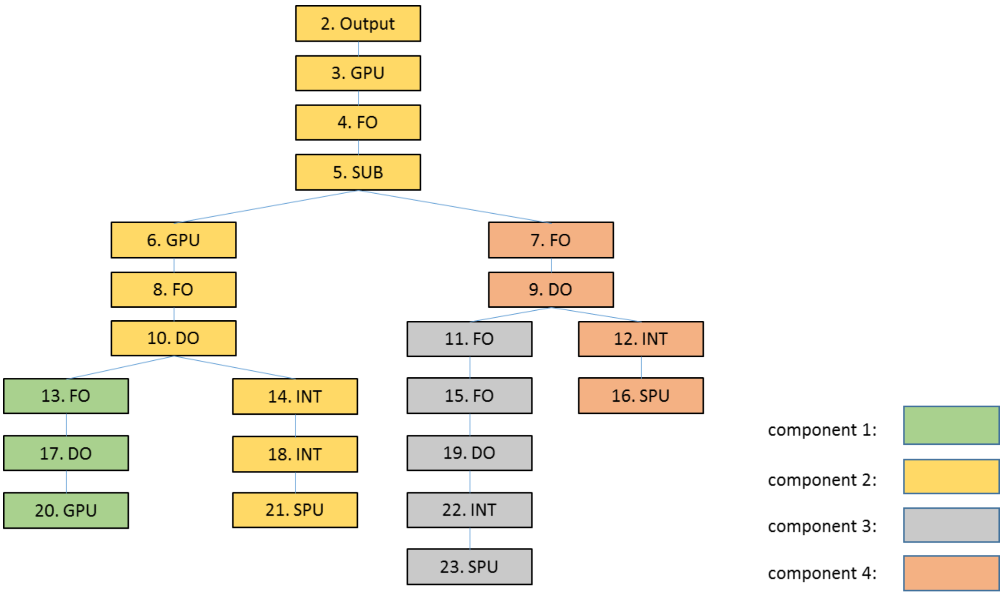
[55] Nikolaj Van Omme, Laurent Perron, and Vincent Furnon. Google OR-Tools User's Manual. [Online]. http://or-tools.googlecode.com/svn/trunk/documentation/user_manual/index.html. Accessed on: 2015-07-06.

[56] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*.: MIT Press & McGraw-Hill, 1990.

[57] SWIG. Introduction to SWIG. [Online]. http://www.swig.org/index.php. Accessed on: 2015-07-06.

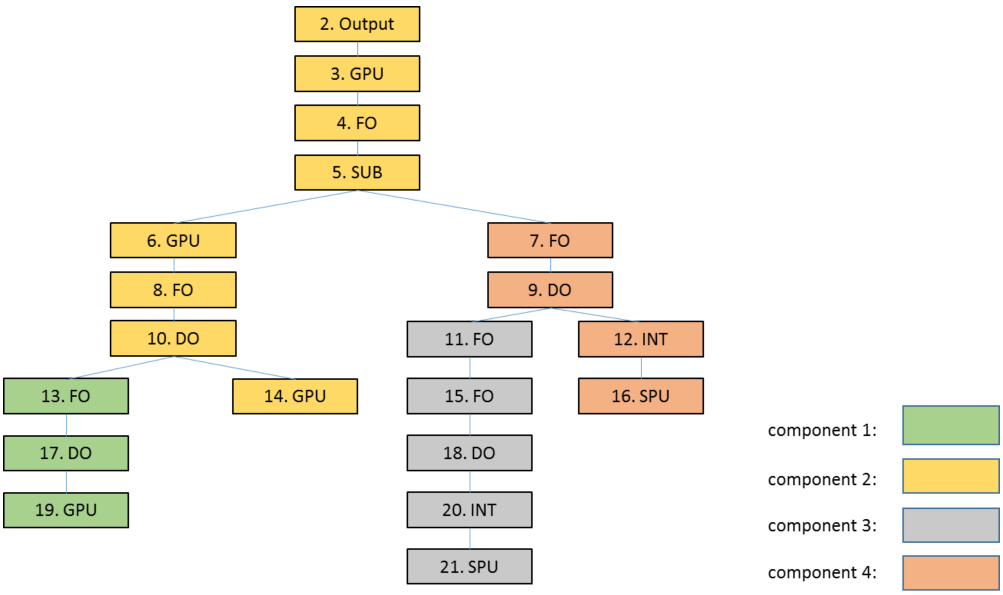
Appendix

A1. Assembly tree of case 2



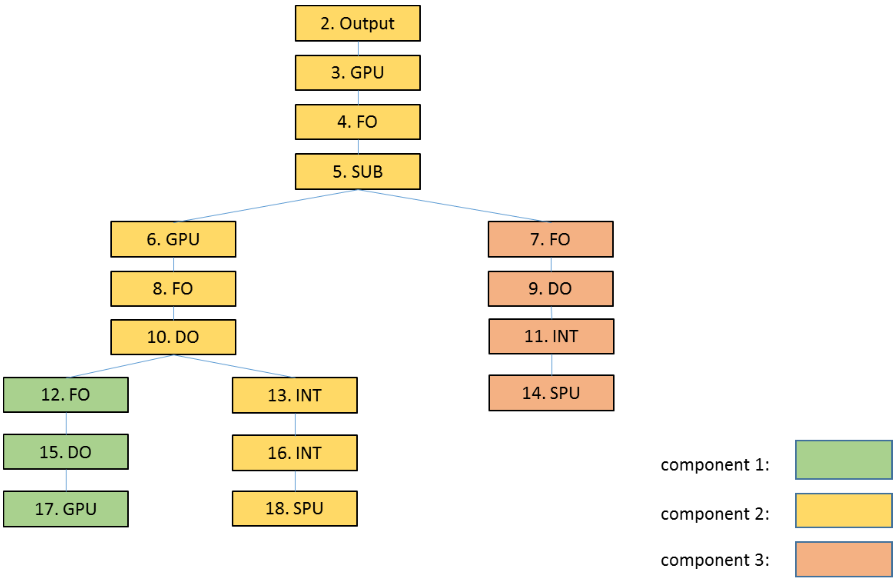
Assembly tree of case 2 (1 GPU, 3 SPU)

A2. Assembly tree of case 3



Assembly tree of case 3 (2 GPU, 2 SPU)

A3. Assembly tree of case 4



Assembly tree of case 4 (1 GPU, 2 SPU)

TRITA TRITA-ICT-EX-2015:201