Hands-on Optimization with OR-Tools in Python

From Beginning to Giving Up

Kunlei Lian

2023-02-18

Table of contents

For the Impatient

This book is still under active development.

If you have experience with other optimization tools and you are just looking for some OR-Tools code examples, the example formulation and implementation below should give you a jump start. The formulation is devoid of any practical meanings.

$$
\begin{aligned}
\text{min.} \quad & \sum_{s \in \mathcal{S}} \sum_{d \in \mathcal{D}} c_{sd} x_{sd} && (0.1)\\
\text{s.t.} \quad & \sum_{d \in \mathcal{D}} x_{sd} = p_s, \ \forall s \in \mathcal{S} && (0.2)\\
& \sum_{s \in \mathcal{S}} x_{sd} = m_d, \ \forall d \in \mathcal{D} && (0.3)\\
& x_{sd} \geq 0, \ \forall s \in \mathcal{S}, d \in \mathcal{D} && (0.4)
\end{aligned}
$$

The implementation in OR-Tools of the above model aims to convey a few information:

- A `solver` object has to be instantiated first in order to create variables, objective and constraints.
- Numerical variables can be created using the `solver.NumVar()` method.
- Objective is added using the `solver.Minimize()` (`solver.Maximize()`) function.
- Constraints are added using the `solver.Add()` function.
- Objective and constraint expressions can be built by first putting their elements into a list and using the `solver.Sum()` method to aggregate them.

```python
from ortools.linear_solver import pywraplp

# gather data
num_sources = 4
num_destinations = 5
supplies = [58, 55, 64, 71]
demands = [44, 28, 36, 52, 88]
costs = [[8, 5, 13, 12, 12],
         [8, 7, 18, 6, 5],
         [11, 12, 5, 11, 18],
```

```
        [19, 13, 5, 10, 18]]

# create solver
solver = pywraplp.Solver.CreateSolver("GLOP")

# create decision variables
var_flow = []
for src_idx in range(num_sources):
    vars = [
        solver.NumVar(0, solver.Infinity(),
                    name=f"var_{src_idx}, {dest_idx}")
        for dest_idx in range(num_destinations)
    ]
    var_flow.append(vars)

# create constraints
for src_idx in range(num_sources):
    expr = [var_flow[src_idx][dest_idx]
            for dest_idx in range(num_destinations)]
    solver.Add(solver.Sum(expr) == supplies[src_idx])

for dest_idx in range(num_destinations):
    expr = [var_flow[src_idx][dest_idx]
            for src_idx in range(num_sources)]
    solver.Add(solver.Sum(expr) == demands[dest_idx])

# create objective function
obj_expr = []
for src_idx in range(num_sources):
    for dest_idx in range(num_destinations):
        obj_expr.append(var_flow[src_idx][dest_idx] * costs[src_idx][dest_idx])
solver.Minimize(solver.Sum(obj_expr))

status = solver.Solve()

opt_flow = []
if status == pywraplp.Solver.OPTIMAL:
    print(f"optimal obj = {solver.Objective().Value()}")
    for src_idx in range(num_sources):
        opt_vals = [var_flow[src_idx][dest_idx].solution_value()
                    for dest_idx in range(num_destinations)]
```

```
        opt_flow.append(opt_vals)
```

Preface

Dear Reader,

If you're reading this preface, then congratulations! You've either stumbled upon this book accidentally, or you're one of the select few who shares my passion for optimization problems and solving them with Google OR-Tools. Either way, welcome!

First things first, let me start by saying that writing a book is hard. Like, really hard. There were times when I thought I would never finish this darn thing. But then I reminded myself of why I started in the first place: I wanted to become an expert in Google OR-Tools and share that expertise with a broader audience. Plus, it was a great excuse to avoid doing laundry for weeks on end.

Now, I'm not going to lie to you. If you're not a fan of math, algorithms, or Python, then this book may not be for you. But if you're anything like me, and you get a thrill out of finding the optimal solution to a complex problem, then buckle up! We're about to embark on an adventure of optimization, constraints, and fancy algorithms.

Throughout this book, we'll cover everything from basic linear programming to more advanced metaheuristics. And trust me, we'll have fun along the way. There will be laughter, tears (mostly from debugging errors), and hopefully a few "aha!" moments.

In all seriousness, I wrote this book because I believe that Google OR-Tools is an incredibly powerful tool for solving real-world optimization problems, and I want to share that knowledge with you. So, grab a cup of coffee (or tea, if you're fancy like that) and let's dive in!

Yours in optimization,

Kunlei Lian

1 Introduction

This book covers the usage of Google OR-Tools to solve optimization problems in Python. There are several major chapters in this book:

In Chapter 2, we explain the steps needed to setup OR-Tools in a Python environment.

In Chapter 3, we use an example to illustrate the modeling capability of OR-Tools to solve linear programming problems.

In Chapter 4, we go through the modeling techniques made available in OR-Tools.

2 Environment Setup

In this chapter, we explain the steps needed to set up Python and Google OR-Tools. All the steps below are based on MacBook Air with M1 chip and macOS Ventura 13.1.

2.1 Install Homebrew

The first tool we need is Homebrew, 'the Missing Package Manager for macOS (or Linux)', and it can be accessed at https://brew.sh/. To install Homebrew, just copy the command below and run it in the Terminal.

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install
```

We can then use the `brew --version` command to check the installed version. On my system, it shows the info below.

```
  ~/ brew --version
Homebrew 3.6.20
Homebrew/homebrew-core (git revision 5f1582e4d55; last commit 2023-02-05)
Homebrew/homebrew-cask (git revision fa3b8a669d; last commit 2023-02-05)
```

2.2 Install Anaconda

Since there are several Python versions available for our use and we may end up having multiple Python versions installed on our machine, it is important to use a consistent environment to work on our project in. Anaconda is a package and environment manager for Python and it provides easy-to-use tools to facilitate our data science needs. To install Anaconda, run the below command in the Terminal.

```
  ~/ brew install anaconda
```

After the installation is done, we can use `conda --version` to verify whether it is available on our machine or not.

```
  ~/ conda --version
conda 23.1.0
```

2.3 Create a Conda Environment

Now we will create a Conda environment named 'ortools'. Execute the below command in the Terminal, which effectively creates the required environment with Python version 3.10.

```
  ~/ conda create -n ortools python=3.10
Retrieving notices: ...working... done
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /opt/homebrew/anaconda3/envs/test

  added / updated specs:
    - python=3.10


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    setuptools-67.4.0          |     pyhd8ed1ab_0         567 KB  conda-forge
    ------------------------------------------------------------
                                           Total:         567 KB

The following NEW packages will be INSTALLED:

  bzip2              conda-forge/osx-arm64::bzip2-1.0.8-h3422bc3_4
  ca-certificates    conda-forge/osx-arm64::ca-certificates-2022.12.7-h4653dfc_0
  libffi             conda-forge/osx-arm64::libffi-3.4.2-h3422bc3_5
  libsqlite          conda-forge/osx-arm64::libsqlite-3.40.0-h76d750c_0
  libzlib            conda-forge/osx-arm64::libzlib-1.2.13-h03a7124_4
  ncurses            conda-forge/osx-arm64::ncurses-6.3-h07bb92c_1
  openssl            conda-forge/osx-arm64::openssl-3.0.8-h03a7124_0
  pip                conda-forge/noarch::pip-23.0.1-pyhd8ed1ab_0
  python             conda-forge/osx-arm64::python-3.10.9-h3ba56d0_0_cpython
```

```
  readline          conda-forge/osx-arm64::readline-8.1.2-h46ed386_0
  setuptools        conda-forge/noarch::setuptools-67.4.0-pyhd8ed1ab_0
  tk                conda-forge/osx-arm64::tk-8.6.12-he1e0b03_0
  tzdata            conda-forge/noarch::tzdata-2022g-h191b570_0
  wheel             conda-forge/noarch::wheel-0.38.4-pyhd8ed1ab_0
  xz                conda-forge/osx-arm64::xz-5.2.6-h57fd34a_0


Proceed ([y]/n)?
```

Type 'y' to proceed and Conda will create the environment for us. We can use `cnoda env list` to show all the created environments on our machine:

```
  ~/ conda env list
# conda environments:
#
base                     /opt/homebrew/anaconda3
ortools                   /opt/homebrew/anaconda3/envs/ortools
```

Note that we need to manually activate an environemnt in order to use it: `conda activate ortools`. On my machine, the activated environment `ortools` will appear in the beginning of my prompt.

```
  ~/ conda activate ortools
(ortools)  ~/
```

2.4 Install Google OR-Tools

As of this writing, the latest version of Google OR-Tools is 9.5.2237, and we can install it in our newly created environment using the command `pip install ortools==9.5.2237`. We can use `conda list` to verify whether it is available in our environment.

```
(ortools)  ~/ conda list
# packages in environment at /opt/homebrew/anaconda3/envs/ortools:
#
# Name                    Version                   Build  Channel
absl-py                   1.4.0                    pypi_0    pypi
bzip2                     1.0.8                h3422bc3_4    conda-forge
ca-certificates           2022.12.7            h4653dfc_0    conda-forge
libffi                    3.4.2                h3422bc3_5    conda-forge
```

```
libsqlite                 3.40.0                h76d750c_0    conda-forge
libzlib                   1.2.13                h03a7124_4    conda-forge
ncurses                   6.3                   h07bb92c_1    conda-forge
numpy                     1.24.2                    pypi_0    pypi
openssl                   3.0.8                 h03a7124_0    conda-forge
ortools                   9.5.2237                  pypi_0    pypi
pip                       23.0.1              pyhd8ed1ab_0    conda-forge
protobuf                  4.22.0                    pypi_0    pypi
python                    3.10.9          h3ba56d0_0_cpython    conda-forge
readline                  8.1.2                 h46ed386_0    conda-forge
setuptools                67.4.0              pyhd8ed1ab_0    conda-forge
tk                        8.6.12                he1e0b03_0    conda-forge
tzdata                    2022g                 h191b570_0    conda-forge
wheel                     0.38.4              pyhd8ed1ab_0    conda-forge
xz                        5.2.6                 h57fd34a_0    conda-forge
```

Now we have Python and Google OR-Tools ready, we can start our next journey.

Part I

Mathematical Programming

3 Linear Programming

In this chapter, we first go through the modeling capabilities provided by Google OR-Tools to solve linear programming problems. Then we get our hands dirty by solving some linear programming problems.

3.1 Modeling Capabilities

There are three components in a mathematical model, namely, decision varaibles, constraints and objective, for which we will go over in the following sections.

3.1.1 Solver

In Google OR-Tools, a `Solver` instance must be created first so that variables, constraints and objective can be added to it. The `Solver` class is defined in the `ortools.linear_solver.pywraplp` module and it requires a solver id to instantiate an object. In the code snippet below, the required module is imported first and a `solver` object is created with `GLOP`, Google's own optimization engine for solving linear programming problems. It is good practice to verify whether the desired solver is indeed created successfully or not.

```python
from ortools.linear_solver import pywraplp

solver = pywraplp.Solver.CreateSolver("GLOP")

if solver:
    print("solver creation success!")
else:
    print("solver creation failure!")
```

```
solver creation success!
```

3.1.2 Decision Variables

The `Solver` class defines a number of ways to create decision variables:

1. `Var(lb, ub, integer, name)`
2. `NumVar(lb, ub, name)`
3. `IntVar(lb, ub, name)`
4. `BoolVar(name)`

- Function `Var()`

The `Var()` method is the most flexible way to define variables, as it can be used to create numerical, integral and boolean variables. In the following code, a numerical variable named 'var1' is created with bound (0.0, 1.0). Note that the parameter `integer` is set to `False` in the call to function `Var()`.

```
var1 = solver.Var(lb=0, ub=1.0, integer=False, name="var1")
```

We could create an integer variable using the same function:

```
var2 = solver.Var(lb=0, ub=1.0, integer=True, name="var2")
```

- Function `NumVar()`

`var1` could be creatd alternatively using the specialized function `NumVar()`:

```
var1 = solver.NumVar(lb=0, ub=1.0, name="var1")
```

- Function `IntVar()`

Similarly, `var2` could be created alternatively using the specialized function `IntVar()`:

```
var2 = solver.IntVar(lb=0, ub=1.0, name="var2")
```

- Function `BoolVar()`

A boolean variable could be created using the `BoolVar()` function:

```
var3 = solver.BoolVar(name="var3")
```

3.1.3 Constraints

Constraints limit the solution space of an optimization problem, and there are two ways to define constraints in Google OR-Tools. In the first approach, we could use the `Add()` function to create a constraint and automatically add it to the model at the same time, as the below code snippet illustrates.

```
cons1 = solver.Add(constraint=var1 + var2 <= 1, name="cons1")

type(cons1)
```

```
ortools.linear_solver.pywraplp.Constraint
```

Note that the `Add()` function returns an object of the `Constraint` class defined in the `pywraplp` module, as shown in the code output. It is a good practice to retain the reference of the newly created constraint, as we might want to query its information later on.

The second approach works in a slightly different way. It starts with an empty constraint, with potential lower bound and upper bounds provided, and add components of the constraint gradually. The code snippet below shows an example of adding a second constraint to the model. In this approach, we must retain the reference to the constraint, as it is needed to add decision variables to the constraint in following steps.

```
cons2 = solver.Constraint(-solver.infinity(), 10.0, "cons2")
cons2.SetCoefficient(var1, 2)
cons2.SetCoefficient(var2, 3)
cons2.SetCoefficient(var3, 4)
type(cons2)
```

```
ortools.linear_solver.pywraplp.Constraint
```

3.1.4 Objective

Similar to constraints, there are two ways to define the objective in Google OR-Tools. In the first approach, we directly add an objective to the model by using the `Maximize()` or `Minimize()` function. Below is an example:

```
solver.Minimize(var1 + var2 + var3)
```

Note that the function itself does not return a reference to the newly created objective function, but we could use a dedicated function to retrive it:

```
obj = solver.Objective()
print(obj)
```

```
<ortools.linear_solver.pywraplp.Objective; proxy of <Swig Object of type 'operations_researc
```

In the second approach, we build the objective incrementally, just as in the second approach of creating constraints. Specifically, we start with an empty objective function, and gradually add components to it. In the end, we specify the optimization sense - whether we want to maximize or minimnize the objective.

```
obj = solver.Objective()
obj.SetCoefficient(var1, 1.0)
obj.SetCoefficient(var2, 1.0)
obj.SetCoefficient(var3, 1.0)
obj.SetMinimization()
print(obj)
```

```
<ortools.linear_solver.pywraplp.Objective; proxy of <Swig Object of type 'operations_researc
```

3.1.5 Objective and Constraint Expressions

When we build constraints or objective functions, sometimes they comprise of complex expressions that we would like to build incrementally, possibly within loops. For example, we might have a mathematical expression of the form $expr = 2x_1 + 3x_2 + 4x_3 + x_4$, which could be part of the objective function or any constraints. In this case, we can either use the aforementioned `SetCoefficient()` function to add each element of the expression to the constraint or objective, or we could build an expression first and add it once in the end. The code snippet below shows an example.

```
infinity = solver.Infinity()
x1 = solver.NumVar(0, infinity, name="x1")
x2 = solver.NumVar(0, infinity, name="x2")
x3 = solver.NumVar(0, infinity, name="x3")
x4 = solver.NumVar(0, infinity, name="x4")

expr = []
expr.append(2 * x1)
expr.append(3 * x2)
expr.append(4 * x3)
```

```
expr.append(x4)

constr = solver.Add(solver.Sum(expr) <= 10)
print(constr)

solver.Minimize(solver.Sum(expr))
```

```
<ortools.linear_solver.pywraplp.Constraint; proxy of <Swig Object of type 'operations_researc
```

Of course, it is not obvious here that the retitive calls to the `append()` method are any more convenient than the `SetCoefficient()` method. Let's say that we have a slightly more complex expression of the form $\sum_{0 \leq i < 4} w_i \cdot x_i$, now we could build the expression using a loop:

```
w = [2, 3, 4, 1]
x = [x1, x2, x3, x4]
expr = []
for i in range(4):
    expr.append(w[i] * x[i])

constr = solver.Add(solver.Sum(expr) <= 10)
```

3.1.6 Query the Model

After we build the model, we can query it using some helper functions. For example, to get the total number of constraints, we use the `NumVariables()` function. In a similar fashion, we can retrieve the total number of constraints with the `NumConstraints()` function.

```
num_vars = solver.NumVariables()
print(f"there are a total of {num_vars} variables in the model")

num_constr = solver.NumConstraints()
print(f"there are a total of {num_constr} constraints in the model")
```

```
there are a total of 9 variables in the model
there are a total of 4 constraints in the model
```

3.2 Applications

In this section, we use some examples to showcase the modeling capability of Google OR-Tools.

3.2.1 Trivial Problem

We now consider an simple linear programming problem with two decision variables x and y. The formal mathematical model is defined as below:

$$\begin{aligned} \text{max.} \quad & x + 2y && (3.1) \\ \text{s.t.} \quad & x + y \leq 10 && (3.2) \\ & x >= 1 && (3.3) \\ & y >= 1 && (3.4) \end{aligned}$$

Figure 3.1 shows the three defining constraints represented in blue lines and the feasible space depicted by the orange shaded area. The objective function is indicated by the red dashed lines. It can be seen from the figure that the point in green circle gives the maximal objective value of 19.

Let's now use Google OR-Tools to model and solve this problem. The code snippet below shows the complete program.

```
# import Google OR-Tools library
from ortools.linear_solver import pywraplp

# create a solver
solver = pywraplp.Solver.CreateSolver("GLOP")

# create decision variables
x = solver.NumVar(1.0, solver.Infinity(), "x")
y = solver.NumVar(1.0, solver.Infinity(), "y")

# create constraints
constr = solver.Add(x + y <= 10)

# create objective
solver.Maximize(x + 2 * y)

# solve the problem
```

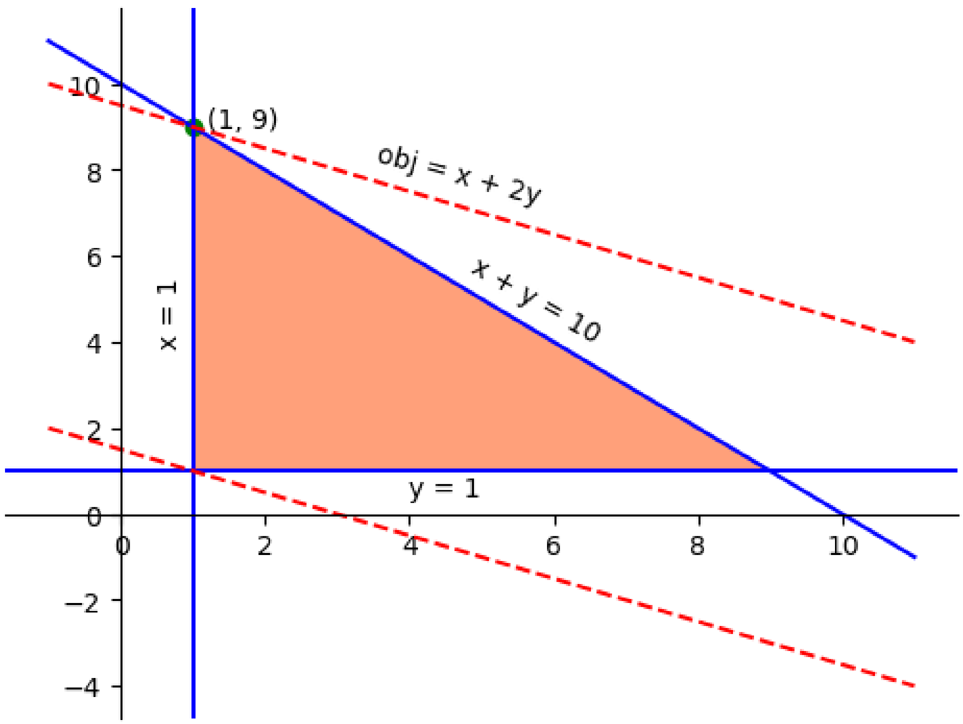

Figure 3.1: A simple LP example

```python
status = solver.Solve()

if status == pywraplp.Solver.OPTIMAL or status == pywraplp.Solver.FEASIBLE:
    print(f"obj = {solver.Objective().Value()}")
    print(f"x = {x.solution_value()}, reduced cost = {x.reduced_cost()}")
    print(f"y = {y.solution_value()}, reduced cost = {y.reduced_cost()}")
    print(f"constr dual value = {constr.dual_value()}")
```

```
obj = 19.0
x = 1.0, reduced cost = -1.0
y = 9.0, reduced cost = 0.0
constr dual value = 2.0
```

We can see from the output that the optimal solution is $x = 1.0$ and $y = 9.0$, and the optimal objective is 19.0. This can also be validated from Figure 3.1 that the optimal solution is exactly the green point that sits at the intersection of the three lines $x = 1$, $x + y = 10$ and $x + 2y = 19$.

Figure 3.1 also shows that the point (1, 1) should give us the minimal value of the objective function. To validate this, we can actually change the optimization sense of the objective function from maximization to minimization using the function `SetOptimizationDirection()`, as shown in the code below:

```python
solver.Objective().SetOptimizationDirection(maximize=False)

solver.Solve()

print(f"obj = {solver.Objective().Value()}")
print(f"x = {x.solution_value()}, reduced cost = {x.reduced_cost()}")
print(f"y = {y.solution_value()}, reduced cost = {y.reduced_cost()}")
print(f"constr dual value = {constr.dual_value()}")
```

```
obj = 3.0
x = 1.0, reduced cost = 1.0
y = 1.0, reduced cost = 2.0
constr dual value = 0.0
```

3.2.2 Transportation Problem

The transportation problem involves moving goods from its sources $\mathcal{S}$ to destinations $\mathcal{D}$. Each source $s \in \mathcal{S}$ has a total amount of goods p_s it could supply, and each destination $s \in \mathcal{D}$ has a certain amount of demands m_d. There is a transportation cost, denoted by c_{sd}, to move one unit of goods from a source to a destination. The problem is to find the best set of goods to move from each source to each destination such that all the destination demands are met with the lowest transportation costs.

To model this transportation problem, we define the decision variable x_{sd} to be the amount of goods moving from source s to destinaiton d. Then we could state the problem mathematically as below.

$$\text{min.} \quad \sum_{s \in \mathcal{S}} \sum_{d \in \mathcal{D}} c_{sd} x_{sd} \tag{3.5}$$

$$\text{s.t.} \quad \sum_{d \in \mathcal{D}} x_{sd} = p_s, \ \forall s \in \mathcal{S} \tag{3.6}$$

$$\sum_{s \in \mathcal{S}} x_{sd} = m_d, \ \forall d \in \mathcal{D} \tag{3.7}$$

$$x_{sd} \geq 0, \ \forall s \in \mathcal{S}, d \in \mathcal{D} \tag{3.8}$$

The objective function (3.5) aims to minimize the total transportation costs going from all sources to all destinations. Constraints (3.6) make sure that the sum of goods leaving a source node s must equal to its available supply p_s. Constraints (3.7) require that the sum of goods gonig to a destination node d must equal to its demand m_d. Constraints (3.8) state that the flow variables from sources to destination can only be nonnegative values.

Table 3.1 shows an instance of the transportation problem in which there are four sources and five destinations. Entries in the last row give the corresponding demand from each destination, and the last column list the available supply at each source. The entries in the middle of the table show the transportation cost associated with moving from a specific source to a specific destination. For example, it costs \$18 to move one unit of good from source S2 to D3.

Table 3.1: A transportation problem

	D1	D2	D3	D4	D5	Supply
S1	8	5	13	12	12	58
S2	8	7	18	6	5	55
S3	11	12	5	11	18	64
S4	19	13	5	10	18	71
Demand	44	28	36	52	88	248

We show here two modeling flavors of using OR-Tools to solve this problem. In the first approach, decision variables are created using the `NumVar()` function, cnostraints are defined using the `Add()` function and the objective function is added using the `Minimize()` function.

Note that both constraints and the objective function are generated with the help of of `Sum()` function that creates an expression.

```
from ortools.linear_solver import pywraplp

# gather data
num_sources = 4
num_destinations = 5
supplies = [58, 55, 64, 71]
demands = [44, 28, 36, 52, 88]
costs = [[8, 5, 13, 12, 12],
         [8, 7, 18, 6, 5],
         [11, 12, 5, 11, 18],
         [19, 13, 5, 10, 18]]

# create solver
solver = pywraplp.Solver.CreateSolver("GLOP")

# create decision variables
var_flow = []
for src_idx in range(num_sources):
    vars = [
        solver.NumVar(0, solver.Infinity(),
                    name=f"var_{src_idx}, {dest_idx}")
        for dest_idx in range(num_destinations)
    ]
    var_flow.append(vars)

# create constraints
for src_idx in range(num_sources):
    expr = [var_flow[src_idx][dest_idx]
            for dest_idx in range(num_destinations)]
    solver.Add(solver.Sum(expr) == supplies[src_idx])

for dest_idx in range(num_destinations):
    expr = [var_flow[src_idx][dest_idx]
            for src_idx in range(num_sources)]
    solver.Add(solver.Sum(expr) == demands[dest_idx])

# create objective function
obj_expr = []
for src_idx in range(num_sources):
```

```
    for dest_idx in range(num_destinations):
        obj_expr.append(var_flow[src_idx][dest_idx] * costs[src_idx][dest_idx])
solver.Minimize(solver.Sum(obj_expr))

status = solver.Solve()

opt_flow = []
if status == pywraplp.Solver.OPTIMAL:
    print(f"optimal obj = {solver.Objective().Value()}")
    for src_idx in range(num_sources):
        opt_vals = [var_flow[src_idx][dest_idx].solution_value()
                    for dest_idx in range(num_destinations)]
        opt_flow.append(opt_vals)
```

```
optimal obj = 2013.0
```

The optimal solution is shown in Table 3.2.

Table 3.2: The optimal solution

	D1	D2	D3	D4	D5	Supply
S1	0	28	0	0	30	58
S2	0	0	0	0	55	55
S3	44	0	20	0	0	64
S4	0	0	16	52	3	71
Demand	44	28	36	52	88	248

In the second approach shown in the code snippet below, decision variables are created with the `Var(integer=False)` method instead of the `NumVar()` method. In addition, both constraints and the objective function are created using the `SetCoefficient()` method. In the case of constraints, a lower bound and upper bound are used to generated an empty constraint, and variables are then added to the constraint one by one with their corresponding coefficient. In the case of the objective function, an empty objective is first initialized and variables are then added to it sequentially. Note that the optimization sense is set using the `SetMinimization()` function.

```
from ortools.linear_solver import pywraplp

# gather data
```

```python
num_sources = 4
num_destinations = 5
supplies = [58, 55, 64, 71]
demands = [44, 28, 36, 52, 88]
costs = [[8, 5, 13, 12, 12],
        [8, 7, 18, 6, 5],
        [11, 12, 5, 11, 18],
        [19, 13, 5, 10, 18]]

# create solver
solver = pywraplp.Solver.CreateSolver("GLOP")

# create decision variables
var_flow = []
for src_idx in range(num_sources):
    vars = [
        solver.Var(
            0, solver.Infinity(), integer=False,
            name=f"var_{src_idx}, {dest_idx}"
        )
        for dest_idx in range(num_destinations)
    ]
    var_flow.append(vars)

# create constraints
for src_idx in range(num_sources):
    constr = solver.Constraint(supplies[src_idx], supplies[src_idx])
    for dest_idx in range(num_destinations):
        constr.SetCoefficient(var_flow[src_idx][dest_idx], 1.0)

for dest_idx in range(num_destinations):
    constr = solver.Constraint(demands[dest_idx], demands[dest_idx])
    for src_idx in range(num_sources):
        constr.SetCoefficient(var_flow[src_idx][dest_idx], 1.0)

# create objective function
obj = solver.Objective()
for src_idx in range(num_sources):
    for dest_idx in range(num_destinations):
        obj.SetCoefficient(var_flow[src_idx][dest_idx], costs[src_idx][dest_idx])
obj.SetMinimization()
```

```python
status = solver.Solve()

opt_flow = []
if status == pywraplp.Solver.OPTIMAL:
    print(f"optimal obj = {solver.Objective().Value()}")
    for src_idx in range(num_sources):
        opt_vals = [var_flow[src_idx][dest_idx].solution_value()
                    for dest_idx in range(num_destinations)]
        opt_flow.append(opt_vals)
```

```
optimal obj = 2013.0
```

To validate the results, Table 3.3 shows the optimal solution produced by the second modeling approach, which is the same as in the previous approach.

Table 3.3: The optimal solution

	D1	D2	D3	D4	D5	Supply
S1	0	28	0	0	30	58
S2	0	0	0	0	55	55
S3	44	0	20	0	0	64
S4	0	0	16	52	3	71
Demand	44	28	36	52	88	248

3.2.3 Resource Allocation Problem

The resource allocation problems involves distributing scarce resources among alternative activities. The resources could be machines in a manufacturing facility, money available to spend, or CPU runtime. The activities could be anything that brings profit at the cost of consuming resources. The objective of this problem is therefore to allocate the available resources to activities such that the total profit is maximized.

Here, we give a general resource allocation model devoid of any practical meanings. To this end, we define a few input parameters to this problem:

- $\mathcal{A}$: the set of candidate activities
- $\mathcal{R}$: the set of avaiable resources
- p_a: the profit of performing one unit of activity $a \in \mathcal{A}$
- c_{ar}: the amount of resource $r \in \mathcal{R}$ required by one unit of activity $a \in \mathcal{A}$
- b_r: the total amount of available quantities for resource $r \in \mathcal{R}$

The decision variable x_a represents the amount of activity $a \in \mathcal{A}$ we select to perform, and the mathematical mode is defined below:

$$\text{max.} \quad \sum_{a \in \mathcal{A}} p_a x_a \tag{3.9}$$

$$\text{s.t.} \quad \sum_{a \in \mathcal{A}} c_{ar} \leq b_r, \ \forall r \in \mathcal{R} \tag{3.10}$$

$$x_a \geq 0, \ a \in \mathcal{A} \tag{3.11}$$

Table 3.4 shows an instance of the resource allocation problem, in which there are three type of resources and five candidate activities. The last row gives the profit of performing each unit of an activity, while the last column shows the available amount of resources. The remaining entries in the table refer to the resource consumption for each activity. For example, selecting one unit of activity 1 (A1) requires 90, 64 and 55 units of resources R1, R2 and R3, respectively.

Table 3.4: A resource allocation problem

	A1	A2	A3	A4	A5	Available
R1	90	57	51	97	67	2001
R2	64	58	97	56	93	2616
R3	55	87	77	52	51	1691
Profit	1223	1238	1517	1616	1027	

In the code snippet below, we use Google OR-Tools to solve this problem instance. Again, we start with initializing a solver object, follwed by creation of five decision variables, one for each activity. Both the constraints and objective function are created using the first modeling approach demonstrated previously. The optimal solution is outputed in the end.

```
from ortools.linear_solver import pywraplp

# gather instance data
num_resources = 3
num_activities = 5
profits = [1223, 1238, 1517, 1616, 1027]
available_resources = [2001, 2616, 1691]
costs = [[90, 57, 51, 97, 67],
         [64, 58, 97, 56, 93],
         [55, 87, 77, 52, 51]]
```

```python
# initialize a solver object
solver = pywraplp.Solver.CreateSolver("GLOP")

infinity = solver.Infinity()
# create decision variables
var_x = [solver.NumVar(0, infinity, name=f"x_P{a}")
        for a in range(num_activities)]

# create objective function
solver.Maximize(solver.Sum([profits[a] * var_x[a]
                            for a in range(num_activities)]))

# create constraints
for r_idx in range(num_resources):
    cons = solver.Add(
            solver.Sum([costs[r_idx][a_idx] * var_x[a_idx]
                        for a_idx in range(num_activities)])
                      <= available_resources[r_idx])

status = solver.Solve()
if status != pywraplp.Solver.OPTIMAL:
    print("solver failure!")

print("solve complete!")
opt_obj = solver.Objective().Value()
print(f"optimal obj = {opt_obj:.2f}")

opt_sol = [var_x[a_idx].solution_value()
           for a_idx in range(num_activities)]
for a_idx in range(num_activities):
    print(f"opt_x[{a_idx + 1}] = {opt_sol[a_idx]:.2f}")
```

```
solve complete!
optimal obj = 41645.23
opt_x[1] = 0.00
opt_x[2] = 0.00
opt_x[3] = 12.45
opt_x[4] = 14.08
opt_x[5] = 0.00
```

3.2.4 Workforce Planning Problem

In the workforce planning problem, there are a number of time periods and each period has a workforce requirement that must be satisfied. In addition, there are a set of available work patterns to assign workers to and each pattern cover one or more time periods. Note that assignment of workers to a particular pattern incurs a certain cost. The problem is then to identify the number of workers assigned to each pattern such that the total cost is minimized.

Table 3.5 shows a contrived workforce planning problem instance. In this problem, there are a total of 10 time periods and there are four patterns available to assign workers to. The last row gives the work requirement in each time period and the last column shows the cost of assigning a worker to a pattern.

Table 3.5: A workforce planning problem instance

Coverage	1	2	3	4	5	6	7	8	9	10	Cost
Pattern 1	x	x	x	x							10
Pattern 2			x	x	x						30
Pattern 3				x	x	x	x				20
Pattern 4							x	x	x	x	40
Requirement	3	4	3	1	5	7	2	4	5	1	

To model this problem, we use $\mathcal{T}$ and $\mathcal{P}$ to denote the set of time periods and patterns, respectively. The parameter m_{pt} indicates whether a pattern $p \in \mathcal{P}$ covers a certain time period $t \in \mathcal{T}$. The work requirement of each time period and the cost of assigning a pattern is represented as r_t and c_p, respectively.

Now we are ready to define the variable x_p as the number of workers that are assigned to pattern p, and the mathematical model can be stated as below.

$$\text{min.} \quad \sum_{p \in \mathcal{P}} c_p x_p \tag{3.12}$$

$$\text{s.t.} \quad \sum_{p \in \mathcal{P}} m_{pt} x_p \geq r_t, \ \forall t \in \mathcal{T} \tag{3.13}$$

$$x_p \geq 0, \ \forall p \in \mathcal{P} \tag{3.14}$$

The code snippet below gives the Python code to solve this problem using Google OR-Tools.

```python
from ortools.linear_solver import pywraplp

# import instance data
```

```python
num_periods = 10
num_patterns = 4
requirements = [3, 4, 3, 1, 5, 7, 2, 4, 5, 1]
costs = [10, 30, 20, 40]
patterns = [set([1, 2, 3, 4]),
            set([3, 4, 5]),
            set([4, 5, 6, 7]),
            set([7, 8, 9, 10])]

# create solver object
solver = pywraplp.Solver.CreateSolver('GLOP')

infinity = solver.Infinity()
# create decision variables
var_p = [solver.NumVar(0, infinity, name=f"x_{p}")
         for p in range(num_patterns)]

# create objective function
solver.Minimize(
    solver.Sum([costs[p] * var_p[p]
                for p in range(num_patterns)])
)

# create constraints
for t in range(num_periods):
    solver.Add(
        solver.Sum([var_p[p]
                    for p in range(num_patterns)
                    if (t + 1) in patterns[p]])
        >= requirements[t])

# solve the problem and retrieve optimal solution
status = solver.Solve()
if status == pywraplp.Solver.OPTIMAL:
    print(f"obj = {solver.Objective().Value()}")
    for p in range(num_patterns):
        print(f"var_{p + 1} = {var_p[p].solution_value()}")
```

```
obj = 380.0
var_1 = 4.0
var_2 = 0.0
var_3 = 7.0
```

```
var_4 = 5.0
```

3.2.5 Sudoku Problem

In a Sudoku problem, a grid of 9x9 is given and the task is to fill all the cells with numbers 1-9. At the beginning, some of the cells are already gilled with numbers and the requirements are that the remaining cells must be filled so that each row, each column, and each of the 9 3x3 sub-grids contain all the numbers from 1 to 9 without any repitition. The difficulty level of Sudoku problems depends on the number of cells that are already filled in the grid at the beginning of the game. Problems with fewer initial digits filled are considered more challenging. Figure 3.2 illustrate a sample Sudoku problem.

To model this problem, we define set $S = (1, 2, 3, 4, 5, 6, 7, 8, 9)$ and use $i, j \in S$ to index the row and column respectively. In addition, we use $M = \{(i, j, k) | i, j, k \in S\}$ to represent all the known numbers in the grid.

To formulate this problem, we define 9 binary variables for each cell in the 9x9 grid. Each of the 9 variables corresponds to one of the numbers in set S. Formally, x_{ijk} represents whether the value k shows up in cell (i, j) of the grid. Note that $i, j, k \in S$. The mathematical formulation can be stated as below.

$$\begin{aligned}
\text{min.} \quad & 0 && (3.15)\\
\text{s.t.} \quad & \sum_{j \in S} x_{ijk} = 1, \ \forall i, k \in S && (3.16)\\
& \sum_{i \in S} x_{ijk} = 1, \ \forall j, k \in S && (3.17)\\
& \sum_{k \in S} x_{ijk} = 1, \ \forall i, j \in S && (3.18)\\
& \sum_{(i-1)\times 3+1}^{(i-1)\times 3+3} \ \sum_{(j-1)\times 3+1}^{(j-1)\times 3+3} x_{ijk} = 1, \ \forall i, j \in \{1, 2, 3\}, k \in S && (3.19)\\
& x_{ijk} = 1, \ \forall (i, j, k) \in M && (3.20)\\
& x_{ijk} \in \{0, 1\}, \ \forall i, j, k \in S && (3.21)
\end{aligned}$$

Since no feasible soution is more preferable than another, we use a constant value as the objective function, meaning any feasible soution is an optimal solution to this problem. Constraints (3.16) require that the number $k \in S$ shows up once and only once in each row of the grid. Similarly, (3.17) make sure that the number $k \in S$ shows up once and only once in each column of the grid. For each cell in the grid, only one of the numbers in S can appear, which is guaranteed by constraints (3.18). Constraints (3.19) ensure that the numbers in set S show

		6						
		3						
5						3	7	9
2	1	4	9					
					5	4		
3	5	8				9		
4								2
		5						
8	2							

Figure 3.2: A Sudoku problem

up once and only once in each of the sub-grids. Constraints (3.20) make sure that the existing numbers in the grid stay the same in the optimal solution.

We can then solve the problem using Google OR-Tools and the code snippet is given below.

```
import numpy as np
from ortools.linear_solver import pywraplp

# import data
grid_size = 9
subgrid_size = 3
M = [[(1, 3, 6)],
     [(2, 3, 3)],
     [(3, 1, 5), (3, 7, 3), (3, 8, 7), (3, 9, 9)],
     [(4, 1, 2), (4, 2, 1), (4, 3, 4), (4, 4, 0)],
     [(5, 6, 5), (5, 7, 4)],
     [(6, 1, 3), (6, 2, 5), (6, 3, 8), (6, 7, 9)],
     [(7, 1, 4), (7, 9, 2)],
     [(8, 3, 5)],
     [(9, 1, 8), (9, 2, 2)]]

# create solver
solver = pywraplp.Solver.CreateSolver("SCIP")

# # create decision variables
sudoku_vars = np.empty((grid_size, grid_size, grid_size), dtype=object)
for row in range(grid_size):
     for col in range(grid_size):
          for num in range(grid_size):
               sudoku_vars[row][col][num] = solver.Var(0,
                                        1,
                                        integer=True,
                                        name=f"x_{row, col, num}")

# create objective
solver.Minimize(0)

# create constraints
for row in range(grid_size):
     for num in range(grid_size):
          solver.Add(
               solver.Sum([sudoku_vars[row][col][num]
```

```
                              for col in range(grid_size)
                         ]) == 1
                    )

for col in range(grid_size):
     for num in range(grid_size):
          solver.Add(
               solver.Sum([sudoku_vars[row][col][num]
                              for row in range(grid_size)
                         ]) == 1
                    )

for row in range(grid_size):
     for col in range(grid_size):
          solver.Add(
               solver.Sum([sudoku_vars[row][col][num]
                              for num in range(grid_size)
                         ]) == 1
                    )

for row in range(grid_size):
     known_values = M[row]
     for value in known_values:
          row, col, num = value
          solver.Add(
               sudoku_vars[row - 1][col - 1][num - 1] == 1
          )

# solve the problem
status = solver.Solve()

if status == pywraplp.Solver.OPTIMAL or status == pywraplp.Solver.FEASIBLE:
     sudoku_sol = np.zeros((grid_size, grid_size), dtype=int)
     for row in range(grid_size):
          for col in range(grid_size):
               for num in range(grid_size):
                    if sudoku_vars[row][col][num].solution_value() == 1:
                         sudoku_sol[row][col] = num + 1
```

Figure 3.3 shows one solution to the example problem.

7	8	6	1	2	3	5	9	4
1	4	3	2	5	9	6	8	7
5	6	2	4	1	8	3	7	9
2	1	4	9	3	6	7	5	8
6	3	7	8	9	5	4	2	1
3	5	8	7	4	2	9	1	6
4	9	1	5	6	7	8	3	2
9	7	5	6	8	1	2	4	3
8	2	9	3	7	4	1	6	5

Figure 3.3: One solution to the Sudoku problem

4 Integer Programming

Integer programming has a wide range of applications across various industries and domains. Some of the classical applications of integer programming include:

- Production Planning and Scheduling: Integer programming is widely used in production planning and scheduling to optimize the allocation of resources, such as machines, workers, and raw materials. It helps to minimize costs and maximize efficiency by determining the optimal production schedule.
- Network Optimization: Integer programming is used in network optimization problems such as routing, scheduling, and allocation of resources in transportation networks, telecommunication networks, and supply chain management.
- Facility Location: Integer programming is used in facility location problems, which involve determining the optimal location for a facility based on various factors such as demand, supply, and transportation costs. It is commonly used in logistics, transportation, and distribution industries.
- Portfolio Optimization: Integer programming is used in finance to optimize investment portfolios, where the goal is to maximize the returns on the investment while minimizing risk.
- Cutting Stock and Bin Packing: Integer programming is used in cutting stock and bin packing problems where items of varying sizes must be packed into containers or cut from a stock. This is commonly used in the packaging and manufacturing industries.
- Crew Scheduling: Integer programming is used in crew scheduling problems, where the goal is to optimize the allocation of crew members to different shifts, duties, or activities. It is commonly used in industries such as airlines, railways, and public transportation.
- Timetabling: Integer programming is used in timetabling problems such as scheduling classes, exams, and events in academic institutions. It helps to minimize scheduling conflicts and maximize resource utilization.

This chapter explores the various methods that Google OR-Tools provides for modeling and solving (mixed) integer linear programming problems. The first step is to review the additional conditions that arise when modeling integer variables and solving integer programs. Following that, we use specific instances to demonstrate how these techniques are applied.

4.1 Modeling Capabilities

When modeling integer programs, there are two main tasks that require attention. The first is declaring integer variables, and the second is selecting a solver that is capable of solving integer programs.

4.1.1 Declaring Integer Variables

As reviewed in Chapter 3, Google OR-Tools provides two options to create integer variables:

- The `Var(lb, ub, integer: bool, name)` function
- The `IntVar(lb, ub, name)` function
- The `Variable.SetInteger(integer: bool)` function

In the code snippet below, we create three integer variables using all the aforementioned approaches:

```
from ortools.linear_solver import pywraplp

solver = pywraplp.Solver.CreateSolver('SCIP')

# option 1
x = solver.Var(lb=0, ub=10, integer=True, name='x')

# option 2
y = solver.IntVar(lb=10, ub=20, name='y')

# option 3
z = solver.NumVar(lb=0, ub=5.5, name='z')
z.SetInteger(integer=True)
```

We can verify the types of variables x, y, z:

```
print(f"x is integer? {x.integer()}")
print(f"y is integer? {y.integer()}")
print(f"z is integer? {z.integer()}")
```

```
x is integer? True
y is integer? True
z is integer? True
```

4.1.2 Selecting an Integer Solver

There are several solvers available for solving integer programs, and some options include:

- CBC_MIXED_INTEGER_PROGRAMMING or CBC
- BOP_INTEGER_PROGRAMMING or BOP
- SAT_INTEGER_PROGRAMMING or SAT or CP_SAT
- SCIP_MIXED_INTEGER_PROGRAMMING or SCIP
- GUROBI_MIXED_INTEGER_PROGRAMMING or GUROBI or GUROBI_MIP
- CPLEX_MIXED_INTEGER_PROGRAMMING or CPLEX or CPLEX_MIP
- XPRESS_MIXED_INTEGER_PROGRAMMING or XPRESS or XPRESS_MIP
- GLPK_MIXED_INTEGER_PROGRAMMING or GLPK or GLPK_MIP

It's important to note that some of these solvers are open-source, while others require a commercial license. The code block above demonstrates how to create an instance of an integer solver. To do so, we simply need to specify the name of the solver in the Solver.CreateSolver() function.

```
solver = pywraplp.Solver.CreateSolver('CBC')
```

5 Job Shop Scheduling

In this section, we use Google OR-Tools to solve some of the classical integer programming problems.

To test the modeling of JSSP, we use a benchmarking instance from the OR-Library (Beasley (1990)), shown in the box below. The two numbers in the first line represent the number of jobs and the number of machines, respectively. Each remaining line contains the operations, processing machine and processing time, for each job. Note that the machines are numbered starting from 0.

```
# Instance ft06 from OR-Library
# 6 6
# 2 1   0   3   1   6   3   7   5   3   4   6
# 1 8   2   5   4   10  5   10  0   10  3   4
# 2 5   3   4   5   8   0   9   1   1   4   7
# 1 5   0   5   2   5   3   3   4   8   5   9
# 2 9   1   3   4   5   5   4   0   3   3   1
# 1 3   3   3   5   9   0   10  4   4   2   1
```

Suppose this instance data is saved in a file named *ft06.txt* and the code below defines an utility function to read and parse the instance for later use.

```
def read_jssp_instance(filename: str):
    with open(filename) as f:
        num_jobs, num_machines = [int(x) for x in next(f).split()]
        operations = []
        processing_times = {}
        job_idx = 0
        for line in f:
            info = [int(x) for x in line.split()]
            arr = [info[2 * m] for m in range(num_machines)]
            times = {info[2 * m]: info[2 * m + 1]
                    for m in range(num_machines)}
            operations.append(arr)
            processing_times[job_idx] = times
            job_idx += 1
```

```
    return num_jobs, num_machines, operations, processing_times
```

We present here three classical formulations of the JSSP from the literature and implement them using Google OR-Tools.

5.1 Disjunctive model

This model is taken from Ku and Beck (2016) and Manne (1960). The decision variables are defined as follows:

- x_{ij}: the processing starting time of job j on machine i
- z_{ijk}: a binary variable that equals 1 if job j precedes job k on machine i

The disjunctive model can then be stated as below.

$$
\begin{aligned}
\text{min.} \quad & C_{max} && (5.1)\\
\text{s.t.} \quad & x_{ij} \geq 0,\ \forall j \in \mathcal{J}, i \in \mathcal{M} && (5.2)\\
& x_{o_h^j,j} \geq x_{o_{h-1}^j,j} + p_{o_{h-1}^j,j},\ \forall j \in \mathcal{J}, h = 2, \cdots, m && (5.3)\\
& x_{ij} \geq x_{ik} + p_{ik} - V \cdot z_{ijk},\ \forall i \in \mathcal{M}, j, k \in \mathcal{J}, j < k && (5.4)\\
& x_{ik} \geq x_{ij} + p_{ij} - V \cdot (1 - z_{ijk}),\ \forall i \in \mathcal{M}, j, k \in \mathcal{J}, j < k && (5.5)\\
& C_{max} \geq x_{o_m^j,j} + p_{o_m^j,j},\ \forall j \in \mathcal{J} && (5.6)\\
& z_{ijk} \in \{0, 1\},\ \forall i \in \mathcal{M}, j, k \in \mathcal{J} && (5.7)
\end{aligned}
$$

The objective (5.1) aims to minimize the maximal completion time of any job $j \in \mathcal{J}$. Constraints (5.2) require that all the job processing starting time must not be negative values. Constraints (5.3) enforce the sequencing order among operations for every job, which state that the h-th operation of job j, o_h^j, cannot start unless its preceeding operation o_{h-1}^j finishes. Constraints (5.4) and (5.5) together make sure that at most one job can be processed on a machine at any time. To be specific, in case of job j preceding job k on machine i, z_{ijk} takes the value of 1 and constraints (5.5) ensure that job k won't start processing on machine i unless job i completes processing; Otherwise, z_{ijk} takes the value of 0 and constraints (5.4) require that job j starts processing after job k. Note that both constraints are needed when we require $j < k$; Otherwise, only one of them is needed if we create a constraint for every pair of j and k on a machine. Constraints (5.6) derive C_{max} across all jobs. The last constraints (5.7) state the variable type of z_{ijk}.

The disjunctive formulation code is presented entirely in the following lines. The data related to the specific case are read between lines 5 to 8, and a solver object is created in line 11. The variable x_{ij} is introduced in lines 16 to 23, followed by the introduction of variable z_{ijk} in lines

25 to 34. The variable C_{max} is defined in lines 36 to 38. The objective of the model is set in line 41, and the constraints are established in lines 44 to 80. The instance is solved, and the optimal solution is obtained from lines 82 to 93.

```
1  from typing import List, Dict
2  from ortools.linear_solver import pywraplp
3  
4  # read and parse the data
5  filename = './data/jssp/ft06.txt'
6  num_jobs, num_machines, \
7  operations, processing_times = \
8      read_jssp_instance(filename)
9  
10 # create solver
11 solver = pywraplp.Solver.CreateSolver('SCIP')
12 
13 # create variables
14 infinity = solver.Infinity()
15 var_time: List[List] = []
16 for machine in range(num_machines):
17     arr = [
18         solver.NumVar(0,
19                       infinity,
20                       name=f'x_{machine, job}')
21         for job in range(num_jobs)
22         ]
23     var_time.append(arr)
24 
25 var_prec: Dict = []
26 for machine in range(num_machines):
27     mac_dict = {}
28     for job_j in range(num_jobs - 1):
29         for job_k in range(job_j + 1, num_jobs):
30             mac_dict[(job_j, job_k)] = \
31             solver.BoolVar(
32                 name=f'z_{machine, job_j, job_k}'
33             )
34     var_prec.append(mac_dict)
35 
36 var_makespan = solver.NumVar(0,
37                              infinity,
38                              name='C_max')
```

```python

# create objective
solver.Minimize(var_makespan)

# create constraints
for job, job_operations in enumerate(operations):
    for h in range(1, num_machines):
        curr_machine = job_operations[h]
        prev_machine = job_operations[h - 1]
        prev_time = processing_times[job][prev_machine]
        solver.Add(
            var_time[curr_machine][job] >=
                var_time[prev_machine][job] +
                prev_time
            )

V = 0
for job in processing_times:
    V += sum(processing_times[job].values())
for machine in range(num_machines):
    for job_j in range(num_jobs - 1):
        for job_k in range(job_j + 1, num_jobs):
            solver.Add(
                var_time[machine][job_j] >=
                var_time[machine][job_k] +
                processing_times[job_k][machine] -
                V * var_prec[machine][(job_j, job_k)]
            )
            solver.Add(
                var_time[machine][job_k] >=
                var_time[machine][job_j] +
                processing_times[job_j][machine] -
                V * (1 - var_prec[machine][(job_j, job_k)])
            )

for job in range(num_jobs):
    last_oper_machine = operations[job][-1]
    solver.Add(
        var_makespan >=
        var_time[last_oper_machine][job] +
        processing_times[job][last_oper_machine]
    )
```

```
status = solver.Solve()

if status == solver.OPTIMAL:
    print(f"min. makespan = {solver.Objective().Value():.2f}")

    opt_time = []
    for machine in range(num_machines):
        arr = [
            int(var_time[machine][job].solution_value())
            for job in range(num_jobs)
        ]
        opt_time.append(arr)
```

```
min. makespan = 55.00
```

The output of the model indicates that the lowest possible time needed to complete all tasks in this instance is 55. To illustrate the most efficient solution, we have created a function called `show_schedule()` that displays a Gantt chart of the tasks needed to process all jobs. Figure 5.1 displays the optimal solution for this instance.

```
import matplotlib as mpl
import matplotlib.pyplot as plt

def show_schedule(num_jobs, operations, processing_times, opt_time):
    colors = mpl.colormaps["Set1"].colors

    fig, ax = plt.subplots(figsize=[7, 3], dpi=100)

    for idx, job in enumerate(range(num_jobs)):
        machines = operations[job]
        job_start_times = [opt_time[machine][job]
                           for machine in machines]
        job_processing_times = [processing_times[job][machine]
                                for machine in operations[job]]

        if idx >= len(colors):
            idx = idx % len(colors)
        color = colors[idx]

        bars = ax.barh(machines,
```

```
                    width=job_processing_times,
                    left=job_start_times,
                    label=f'Job {job + 1}',
                    color=color)

            ax.bar_label(bars,
                        fmt=f'{job + 1}',
                        label_type='center')

    ax.set_yticks(machines)
    ax.set_xlabel("Time")
    ax.set_ylabel("Machine")
    fig.tight_layout()
    plt.show()

show_schedule(num_jobs, operations, processing_times, opt_time)
```

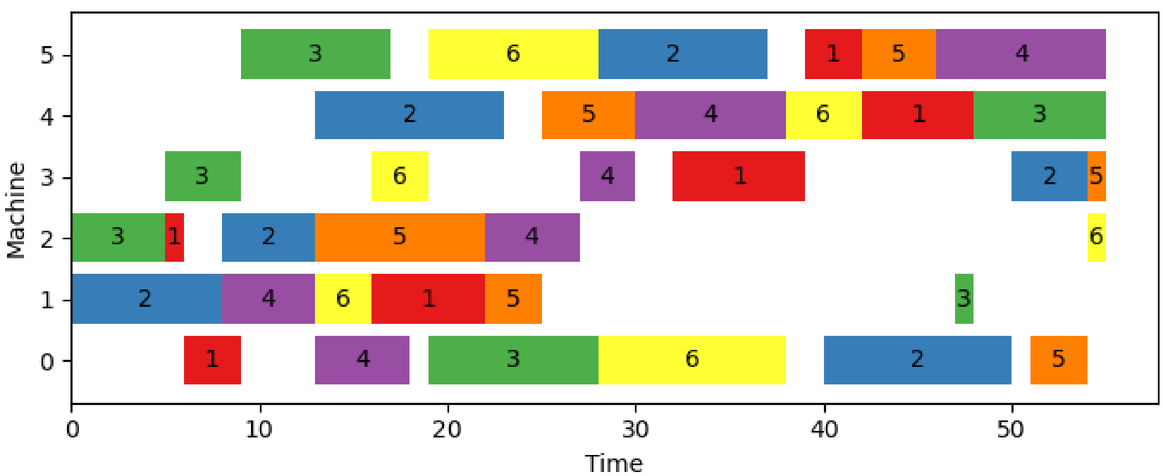

Figure 5.1: Optimal solution of the ft06 instance using the disjunctive formulation

5.2 Time-indexed model

The time-indexed formulation, proposed by Kondili, Pantelides, and Sargent (1988) and Ku and Beck (2016), involves the use of a binary variable x_{ijt} that takes the value of 1 if job j starts at time t on machine i. The model can be expressed as follows.

$$
\begin{aligned}
\text{min.} \quad & C_{max} && (5.8)\\
\text{s.t.} \quad & \sum_{t \in H} x_{ijt} = 1, \;\; \forall j \in \mathcal{J}, i \in \mathcal{M} && (5.9)\\
& \sum_{t \in H} (t + p_{ij}) \cdot x_{ijt} \leq C_{max}, \;\; \forall j \in \mathcal{J}, i \in \mathcal{M} && (5.10)\\
& \sum_{j \in \mathcal{J}} \sum_{t' \in T_{ijt}} x_{ijt'} \leq 1, \;\; \forall i \in \mathcal{M}, t \in H, T_{ijt} = \{t - p_{ij} + 1, \cdots, t\} && (5.11)\\
& \sum_{t \in H} (t + p_{o^j_{h-1}}, j) \cdot x_{o^j_{h-1}, jt} \leq \sum_{t \in H} t \cdot x_{o^j_h, jt}, \;\; \forall j \in \mathcal{J}, h = 2, \cdots, m && (5.12)\\
& x_{ijt} \in \{0, 1\}, \;\; \forall j \in \mathcal{J}, i \in \mathcal{M}, t \in H && (5.13)
\end{aligned}
$$

In this formulation, the first set of constraints, referred to as (5.9), state that each job j must start at one specific time within the scheduling horizon H, which is determined as the sum of processing times for all jobs - $H = \sum_{i \in \mathcal{I}, j \in \mathcal{J}} p_{ij}$. Constraints (5.10) are used to calculate the value of C_{max}, while constraints (5.11) ensure that only one job can be processed by a machine at any given time. It's important to note that a job will remain on a machine for its full processing time and cannot be interrupted. Constraints (5.12) make sure that the order of processing jobs is followed, and constraints (5.13) define the variable types used in the formulation.

The following code provides a program that uses the time-indexed formulation to solve the `ft06` instance, with the value of H being the sum of all job processing times plus one. The two variables, x_{ijt} and C_{max}, are created in lines 18 - 27. The constraints are created in lines 33 - 57. The optimal solution is retrieved in lines 61 - 71. It can be seen from the output that the same optimal objective, 55, is obtained using this formulation.

```
1  from typing import List, Dict
2  from ortools.linear_solver import pywraplp
3  import numpy as np
4
5  # read and parse the data
6  filename = './data/jssp/ft06.txt'
7  num_jobs, num_machines, \
8  operations, processing_times = \
9      read_jssp_instance(filename)
10
11 # create solver
12 solver = pywraplp.Solver.CreateSolver('SCIP')
13
14 # create variables
```

```
H = 1
for job in processing_times:
    H += sum(processing_times[job].values())
var_x = np.empty((num_machines, num_jobs, H), dtype=object)
for machine in range(num_machines):
    for job in range(num_jobs):
        for t in range(H):
            var_x[machine][job][t] = solver.BoolVar(name=f'x_{machine, job, t}')

infinity = solver.Infinity()
var_makespan = solver.NumVar(0,
                            infinity,
                            name='C_max')

# create objective
solver.Minimize(var_makespan)

# create constraints
for machine in range(num_machines):
    for job in range(num_jobs):
        solver.Add(solver.Sum([var_x[machine][job][t] for t in range(H)]) == 1)

for machine in range(num_machines):
    for job in range(num_jobs):
        arr = [var_x[machine][job][t] * (t + processing_times[job][machine]) for t in rang
        solver.Add(solver.Sum(arr) <= var_makespan)

for machine in range(num_machines):
    for t in range(H):
        arr = [var_x[machine][job][tt]
                    for job in range(num_jobs)
                    for tt in range(t - processing_times[job][machine] + 1, t + 1)]
        solver.Add(solver.Sum(arr) <= 1)

for job in range(num_jobs):
    for oper in range(1, num_machines):
        prev_machine = operations[job][oper - 1]
        curr_machine = operations[job][oper]
        expr_prev = [(t + processing_times[job][prev_machine]) * var_x[prev_machine][job][
                    for t in range(H)]
        expr_curr = [t * var_x[curr_machine][job][t]
```

```
                    for t in range(H)]
        solver.Add(solver.Sum(expr_prev) <= solver.Sum(expr_curr))

status = solver.Solve()

if status == solver.OPTIMAL:
    print(f"min. makespan = {solver.Objective().Value():.2f}")

    opt_time = []
    for machine in range(num_machines):
        arr = []
        for job in range(num_jobs):
            for t in range(t):
                if int(var_x[machine][job][t].solution_value()) == 1:
                    arr.append(t)
        opt_time.append(arr)
```

```
min. makespan = 55.00
```

The optimal solution can be seen in Figure 5.2. Even though both formulations achieve the same optimal objective value, there are some minor discrepancies between the optimal solutions. For instance, in Figure 5.1, there is a gap after job 4 completes processing on machine 3, which is not present in Figure 5.2.

```
show_schedule(num_jobs, operations, processing_times, opt_time)
```

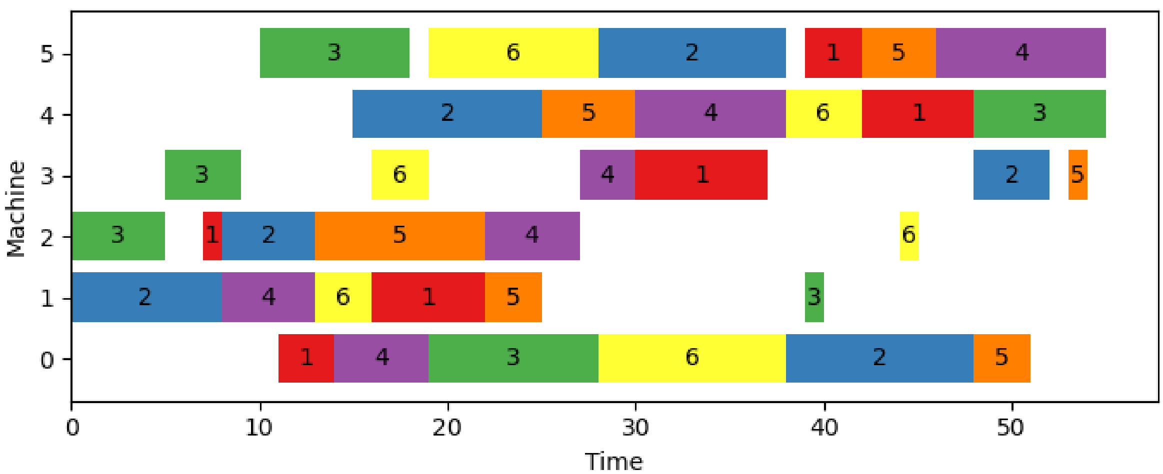

Figure 5.2: Optimal solution of the ft06 instance using the time-indexed formulation

5.3 Rank-based model

The rank-based model is due to Wagner (1959) and taken from Ku and Beck (2016). There are three decision variables in this formulation:

- x_{ijk}: a binary variable that equals 1 if job j is scheduled at the k-th position on machine i
- h_{ik}: a numerical variable that represents the start time of job at the k-th position of machine i.
- C_{max}: the makespan to be minimized

The complete model is given below.

$$
\begin{aligned}
\text{min.} \quad & C_{max} && (5.14)\\
\text{s.t.} \quad & \sum_{j \in \mathcal{J}} x_{ijk} = 1, \ \forall i \in \mathcal{M}, k = 1, \cdots, n && (5.15)\\
& \sum_{k=1}^{n} x_{ijk} = 1, \ \forall i \in \mathcal{M}, j \in \mathcal{J} && (5.16)\\
& h_{ik} + \sum_{j \in \mathcal{J}} p_{ij} x_{ijk} \leq h_{i,k+1}, \ \forall i \in \mathcal{M}, k = 1, \cdots, n-1 && (5.17)\\
& \sum_{i \in \mathcal{M}} r_{ijl} h_{ik} + \sum_{i \in \mathcal{M}} r_{ijl} p_{ij} \leq V \cdot (1 - \sum_{i \in \mathcal{M}} r_{ijl} x_{ijk}) + \\
& V \cdot (1 - \sum_{i \in \mathcal{M}} r_{ij,l+1} x_{ijk'}) + \sum_{i \in \mathcal{M}} r_{ij,l+1} h_{ik'}, \\
& \forall j \in \mathcal{J}, k, k' = 1, \cdots, n, l = 1, \cdots, m-1 && (5.18)\\
& h_{in} + \sum_{j \in \mathcal{J}} p_{ij} x_{ijk} \leq C_{max}, \ \forall i \in \mathcal{M} && (5.19)\\
& h_{ik} \geq 0, \ \forall i \in \mathcal{M}, k = 1, \cdots, n && (5.20)\\
& x_{ijk} \in \{0, 1\}, \ \forall j \in \mathcal{J}, i \in \mathcal{M}, k = 1, \cdots, n && (5.21)\\
& C_{max} \geq 0 && (5.22)
\end{aligned}
$$

In this formulation, constraints (5.15) make sure that there is only one job assigned to a particular rank k on machine i. Constraints (5.16) ensure that any job j is assigned to one and only one rank on a machine i. Constraints (5.17) require that a machine can only process at most one job at any point of time. Constraints (5.18) guarantee that the processing order of a job is respected. Constraints (5.19) computes the makespan. The remaining constraints (5.20), (5.21) and (5.22) indicate the variable types.

We now solve the same problem instance using this rank-based formulation in Google OR-Tools, for which the complete code is shown below.

```
from typing import List
from itertools import product
import numpy as np
from ortools.linear_solver import pywraplp

# read and parse the data
filename = './data/jssp/ft06.txt'
num_jobs, num_machines, \
operations, processing_times = \
    read_jssp_instance(filename)

# create solver
solver = pywraplp.Solver.CreateSolver('SCIP')

# create variables
var_x = np.empty(shape=(num_machines, num_jobs, num_jobs), dtype=object)
for machine, job, rank in product(range(num_machines),
                                  range(num_jobs),
                                  range(num_jobs)):
    var_x[machine][job][rank] = solver.BoolVar(name=f'x_{machine, job, rank}')


infinity = solver.Infinity()
var_h = np.empty(shape=(num_machines, num_jobs), dtype=object)
for machine, rank in product(range(num_machines), range(num_jobs)):
    var_h[machine][rank] = solver.NumVar(0, infinity, name=f'h_{machine, rank}')

var_makespan = solver.NumVar(0, infinity, name=f'makesplan')

# create objective
solver.Minimize(var_makespan)

# create constraints
for machine, rank in product(range(num_machines), range(num_jobs)):
    expr = [var_x[machine][job][rank] for job in range(num_jobs)]
    solver.Add(solver.Sum(expr) == 1)

for machine, job in product(range(num_machines), range(num_jobs)):
    expr = [var_x[machine][job][rank] for rank in range(num_jobs)]
```

```
    solver.Add(solver.Sum(expr) == 1)

for machine, rank in product(range(num_machines), range(num_jobs - 1)):
    expr = [var_x[machine][job][rank] * processing_times[job][machine]
            for job in range(num_jobs)]
    solver.Add(var_h[machine][rank] + solver.Sum(expr) <= var_h[machine][rank + 1])

r = np.zeros((num_machines, num_jobs, num_machines))
for job in range(num_jobs):
    job_operations: List = operations[job]
    for o_idx, o_machine in enumerate(job_operations):
        r[o_machine][job][o_idx] = 1
V = 0
for job in processing_times:
    V += sum(processing_times[job].values())

for job, k, kk, l in product(range(num_jobs),
                             range(num_jobs),
                             range(num_jobs),
                             range(num_machines - 1)):
    expr_1 = [r[machine][job][l] * var_h[machine][k]
            for machine in range(num_machines)]
    expr_2 = [r[machine][job][l] * processing_times[job][machine]
            for machine in range(num_machines)]
    expr_3 = [r[machine][job][l] * var_x[machine][job][k]
            for machine in range(num_machines)]
    expr_4 = [r[machine][job][l + 1] * var_x[machine][job][kk]
            for machine in range(num_machines)]
    expr_5 = [r[machine][job][l + 1] * var_h[machine][kk]
            for machine in range(num_machines)]
    solver.Add(solver.Sum(expr_1) + solver.Sum(expr_2) <= V * (1 - solver.Sum(expr_3)) + V

for machine in range(num_machines):
    expr = [var_x[machine][job][num_jobs - 1] * processing_times[job][machine] for job in
    solver.Add(var_h[machine][num_jobs - 1] + solver.Sum(expr) <= var_makespan)

status = solver.Solve()

if status == pywraplp.Solver.OPTIMAL:
    print(f"opt_obj = {solver.Objective().Value():.4f}")
    opt_time = []
```

```
    for machine in range(num_machines):
        arr = []
        for job in range(num_jobs):
            for rank in range(num_jobs):
                if int(var_x[machine][job][rank].solution_value()) == 1:
                    arr.append(var_h[machine][rank].solution_value())
        opt_time.append(arr)
```

```
opt_obj = 55.0000
```

Figure 5.3 displays the optimal solution obtained by utilizing the rank-based model. Upon careful examination, it is slightly distinct from the optimal solutions produced by the disjunctive model and the time-indexed model. Nevertheless, all three models achieve the same objective value of 55.

```
show_schedule(num_jobs, operations, processing_times, opt_time)
```

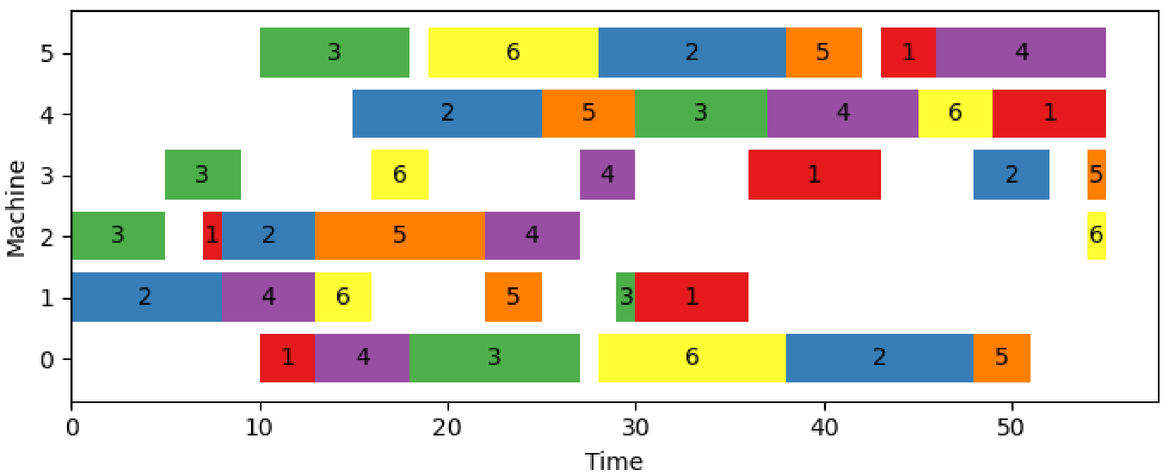

Figure 5.3: Optional solution found by the rank-based model for instance ft06

While this book does not aim to compare the performance of the three modeling approaches, Table 5.1 presents the computational times required by each formulation to discover the optimal solutions. The table indicates that the disjunctive model is the most efficient of the three, followed by the time-indexed model, while the rank-based model requires the longest time to converge. It should be noted that making a conclusion about the performance of these models based on one experimental run on a single instance is insufficient.

Table 5.1: Computational time comparison of the three formulations

Instance	Disjunctive Model	Time-indexed Model	Rank-based Model
ft06	1.7s	1m34.9s	11m38.2s

6 Traveling Salesman Problem

The traveling salesman problem (TSP) is a classic optimization problem in computer science and operations research. The problem can be stated as follows: given a set of cities and the distances between them, what is the shortest possible route that visits each city exactly once and returns to the starting city?

The TSP has many real-world applications, including logistics and transportation planning, circuit board drilling, and DNA sequencing. However, it is a well-known NP-hard problem, meaning that finding an optimal solution is computationally difficult for large instances of the problem. As a result, many heuristic and approximation algorithms have been developed to find suboptimal solutions that are still very good in practice. In this chapter, we present several mathematical formulations of the TSP existing in the literature and implement them using OR-Tools.

6.1 TSP Instances

Before discussing the mathematical models of the TSP, we first provide an introduction to the instances that will be utilized to test various formulations and illustrate the resulting TSP solutions. The TSP is a widely recognized optimization problem that has been studied for several decades. Due to its significance, many benchmarking problem instances of varying sizes are available in literature. In this chapter, we do not aim to solve the most challenging TSP instances, but instead, our objective is to demonstrate how to apply different formulations of the TSP using OR-Tools.

To achieve this objective, we will focus on presenting some of the small-sized instances that can be solved effectively using OR-Tools. These instances are well-documented, which makes them easy to understand and implement in practice. Additionally, they help illustrate the optimization techniques used to solve the TSP, such as branch-and-bound and cutting plane methods. Moreover, small-sized instances allow for quicker computation, making it easier to observe the behavior of different algorithms and identify which formulations are most efficient.

By presenting a range of examples, we aim to provide a clear understanding of how to implement different TSP formulations using OR-Tools, which can be applied to real-world problems in various domains, such as transportation planning and logistics, network design, and circuit board drilling. Additionally, we aim to demonstrate the advantages and limitations of different TSP formulations and algorithms, highlighting which techniques perform well under specific circumstances. By doing so, readers can gain insight into how to apply TSP optimization techniques to their own problems effectively.

6.1.1 TSPLIB

The TSP instances used in this section are sourced from TSPLIB95, a library of TSP benchmark instances. To make it easier to work with these instances, we utilize the `tsplib95` Python library, which can be installed using the `pip install tsplib95` command.

In the code snippet below, we demonstrate how to use the `tsplib95` package to load the *ulysses22.tsp* problem from a data file downloaded from TSPLIB95. The loaded data can be used to formulate and solve TSP instances using OR-Tools or other optimization tools. The full instance data is provided at the end for reference. By leveraging the `tsplib95` package, we can quickly and easily access TSP instances for experimentation and analysis, and focus our efforts on the formulation and optimization aspects of the problem.

```
import tsplib95

# load problem
problem = tsplib95.load('./data/tsp/ulysses22.tsp')

# show instance
problem.as_name_dict()
```

```
{'name': 'ulysses22.tsp',
 'comment': 'Odyssey of Ulysses (Groetschel/Padberg)',
 'type': 'TSP',
 'dimension': 22,
 'edge_weight_type': 'GEO',
 'display_data_type': 'COORD_DISPLAY',
 'node_coords': {1: [38.24, 20.42],
  2: [39.57, 26.15],
  3: [40.56, 25.32],
  4: [36.26, 23.12],
  5: [33.48, 10.54],
  6: [37.56, 12.19],
  7: [38.42, 13.11],
  8: [37.52, 20.44],
  9: [41.23, 9.1],
  10: [41.17, 13.05],
  11: [36.08, -5.21],
  12: [38.47, 15.13],
```

```
  13: [38.15, 15.35],
  14: [37.51, 15.17],
  15: [35.49, 14.32],
  16: [39.36, 19.56],
  17: [38.09, 24.36],
  18: [36.09, 23.0],
  19: [40.44, 13.57],
  20: [40.33, 14.15],
  21: [40.37, 14.23],
  22: [37.57, 22.56]}}
```

The list of nodes can be retrieved using the `get_nodes()` function, as shown below.

```
list(problem.get_nodes())
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]
```

To get the distance between any pair of nodes, we use the `get_weight()` function.

```
print(f'distance between node 1 and 2 = {problem.get_weight(1, 2)}')
```

```
distance between node 1 and 2 = 509
```

6.1.2 Visualize TSP Solution

In this section, our goal is to gain a better understanding of the TSP problem by visualizing the optimal solution found for instances provided by TSPLIB95. To achieve this, we define a class called `TspVisualizer` in our code that is responsible for displaying the route that connects all nodes in a TSP solution. The `TspVisualizer` class contains a single function, called `show(locations, edges)`, which accepts two input parameters: *locations* and *edges*.

The *locations* parameter is a dictionary that contains the mapping between location ID and its corresponding coordinates. The *edges* parameter is a list of edges that form the TSP tour. By calling the show function with the appropriate input parameters, we can visualize the TSP tour and gain an intuitive understanding of what the TSP problem is trying to accomplish. This visualization can be a helpful tool in understanding how the different TSP formulations and algorithms work, and can aid in identifying potential improvements to the solution. The use of the `TspVisualizer` class allows for easy visualization of the TSP solution and makes it possible to explore and analyze TSP instances in a more meaningful way.

```python
import networkx as nx
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

class TspVisualizer:
    """visualize a TSP tour
    """

    @staticmethod
    def show(locations, edges):
        """draw TSP tour
        adapted from https://stackoverflow.com/a/50682819

        examples:
        locations = {
            0: (5, 5),
            1: (4, 9),
            2: (6, 4),
        }

        edges = [
            (0, 1),
            (1, 2),
            (2, 0),
        ]

        Args:
            locations (dict): location id -> (lat, lon)
            edges (list): list of edges
        """
        G = nx.DiGraph()
        G.add_edges_from(edges)
        plt.figure(figsize=(15,10))

        colors = mpl.colormaps["Set1"].colors
        color_idx = 1
        color = np.array([colors[color_idx]])

        nx.draw_networkx_nodes(G,
                               locations,
```

```
                                nodelist=[x[0]
                                        for x in edges],
                                node_color=color)
        nx.draw_networkx_edges(G,
                                locations,
                                edgelist=edges,
                                width=4,
                                edge_color=color,
                                style='dashed')

        # labels
        nx.draw_networkx_labels(G, locations,
                                font_color='w',
                                font_size=12,
                                font_family='sans-serif')

        #print out the graph
        plt.axis('off')
        plt.show()
```

Now let's load the optimal solution for the aforementioned instance and show its content below.

```
solution = tsplib95.load('./data/tsp/ulysses22.opt.tour')
solution.as_name_dict()
```

```
{'name': 'ulysses22.opt.tour',
 'comment': 'Optimal solution of ulysses22 (7013)',
 'type': 'TOUR',
 'dimension': 22,
 'tours': [[1,
   14,
   13,
   12,
   7,
   6,
   15,
   5,
   11,
   9,
   10,
   19,
```

```
20,
21,
16,
3,
2,
17,
22,
4,
18,
8]]}
```


The code snippet below plots the optimal tour, which is shown in Figure 6.1.

```
locations = problem.node_coords
tour = solution.tours[0]
edges = []
for i in range(len(tour) - 1):
    edges.append((tour[i], tour[i + 1]))
edges.append((tour[-1], tour[0]))
edges = []
for i in range(len(tour) - 1):
    edges.append((tour[i], tour[i + 1]))
edges.append((tour[-1], tour[0]))
TspVisualizer.show(locations, edges)
```

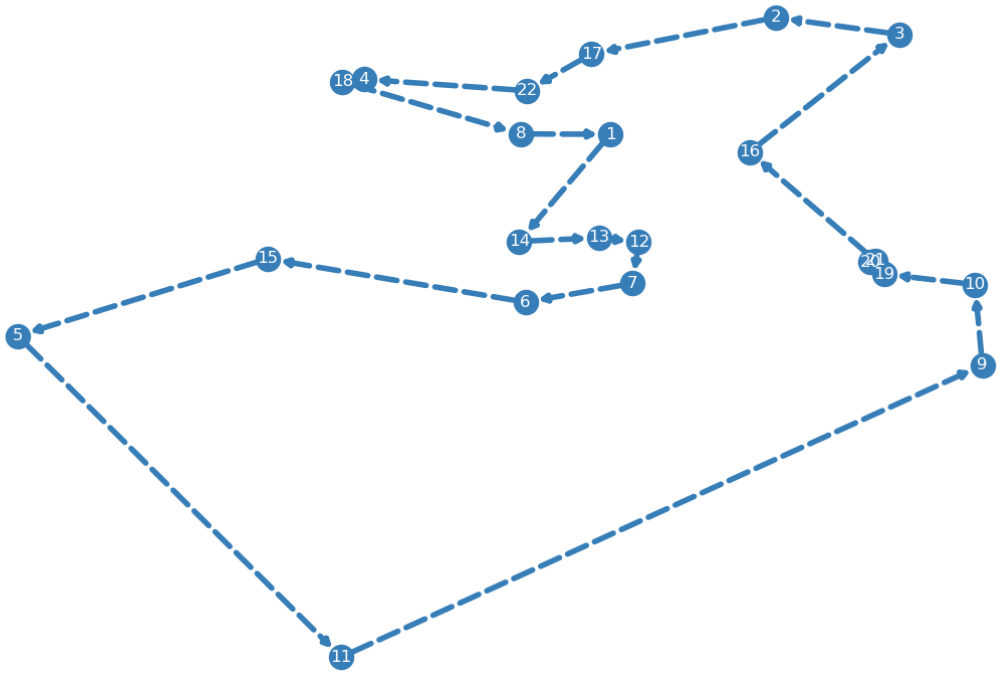

Figure 6.1: Optimal tour of the *ulysses22* instance

Let's put this visualization procedure into a dedicated function, as is given below.

```
import tsplib95

def visualize_tsp(instance_name: str):
    # load problem
    problem = tsplib95.load(f'./data/tsp/{instance_name}.tsp')
    solution = tsplib95.load(f'./data/tsp/{instance_name}.opt.tour')

    locations = problem.node_coords
    tour = solution.tours[0]
    edges = []
    for i in range(len(tour) - 1):
        edges.append((tour[i], tour[i + 1]))
    edges.append((tour[-1], tour[0]))
    edges = []
    for i in range(len(tour) - 1):
```

```
        edges.append((tour[i], tour[i + 1]))
    edges.append((tour[-1], tour[0]))
    TspVisualizer.show(locations, edges)
```

Figure 6.2 and Figure 6.3 show the optimal tours for the *berlin52* and *pr76* instances, respectively.

```
visualize_tsp('berlin52')
```

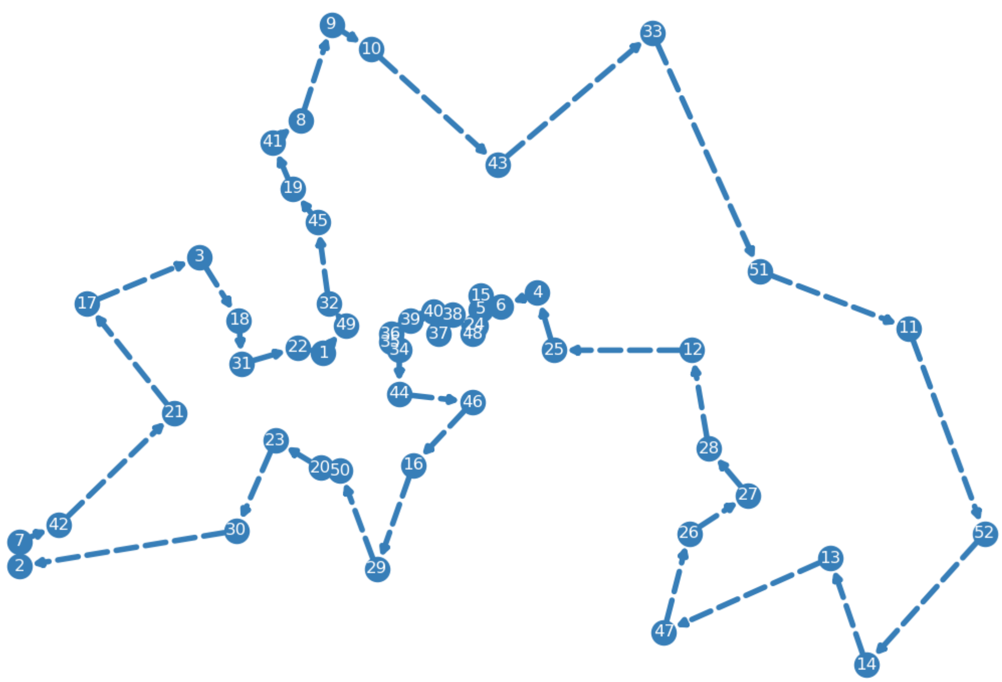

Figure 6.2: Optimal tour of the *berlin52* instance

```
visualize_tsp('pr76')
```

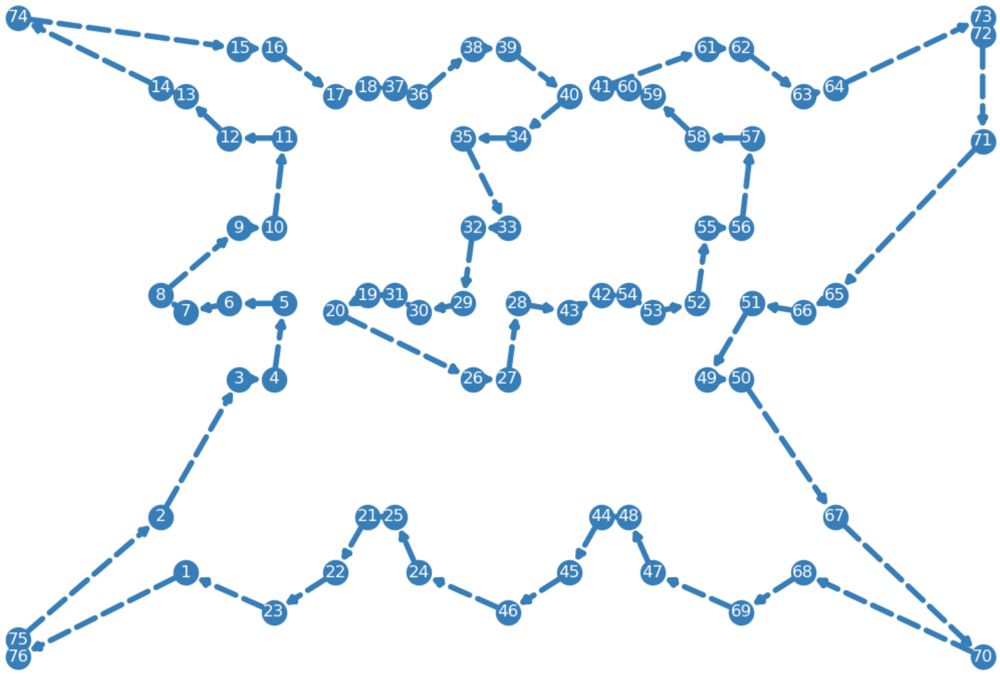

Figure 6.3: Optimal tour of the *pr76* instance

6.2 Problem Description

Let $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ be an undirected complete graph, where $V = \{1, 2, \cdots, n\}$ represents a set of n cities or vertices, and $\mathcal{A} = \{(i, j) \mid i, j \in \mathcal{V}, i \neq j\}$ represents the set of edges connecting these cities. The edges in $\mathcal{A}$ have weights or distances associated with them, c_{ij}, representing the distances or costs to travel between pairs of cities.

The objective of the TSP is to find the shortest possible closed tour that visits each city in $\mathcal{V}$ exactly once and returns to the starting city, while obeying the following constraints:

- Each city must be visited exactly once: The tour must include all the cities in $\mathcal{V}$, and each city must be visited exactly once during the tour.
- The tour must be closed: The last city visited in the tour must be the same as the starting city, forming a closed loop.

6.3 Model 1 - DFJ

The first formulation was proposed by Dantzig, Fulkerson, and Johnson (1954). It uses the following decision variables:

- x_{ij}: a binary variable that equals 1 if arc $(i, j) \in \mathcal{A}$ shows up in the optimal solution, 0 otherwise

We can state the model as follows:

$$\text{min.} \quad \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} \tag{6.1}$$

$$\text{s.t.} \quad \sum_{j\in\mathcal{V},\ j\neq i} x_{ij} = 1,\ \forall i \in \mathcal{V} \tag{6.2}$$

$$\sum_{i\in\mathcal{V},\ i\neq j} x_{ij} = 1,\ \forall j \in \mathcal{V} \tag{6.3}$$

$$\sum_{i,j\in S,\ (i,j)\in\mathcal{A}} x_{ij} \leq |S| - 1, \quad \forall S \subset \mathcal{V},\ 2 \leq |S| \leq n-2 \tag{6.4}$$

$$x_{ij} \in \{0, 1\},\ \forall (i, j) \in \mathcal{A} \tag{6.5}$$

In this formulation, the objective (6.1) aims to minimize the total distance of the optimal tour. Constraints (6.2) and (6.3) make sure that each node is entered and left once and only once, and they are also called *degree constraints*. Constraints (6.4) are the so-called *subtour elimination constraints*.

Figure 6.4 and Figure 6.5 show two subtours with 2 and 3 nodes, respectively. Note that the number of arcs (edges) in a subtour equals to the number of nodes. Therefore,

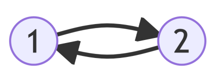

Figure 6.4: Subtour with 2 nodes

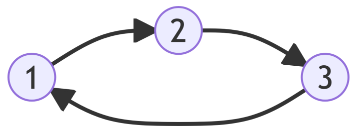

Figure 6.5: Subtour with 3 nodes

6.4 Model 2 - MTZ

6.5 Model 3 - Single Commodity Flow

6.6 Model 4 - Two Commodity Flow

6.7 Model 5 - Multi-Commodity Flow

7 Capacitated Vehicle Routing Problem

The Capacitated Vehicle Routing Problem (CVRP) is a classical combinatorial optimization problem that involves finding the optimal set of routes for a fleet of vehicles to deliver goods or services to a set of customers. In the CVRP, a set of customers is given, each with a known demand and location. A fleet of vehicles, each with a limited capacity, is available to serve these customers. The problem is to find a set of routes visited by the vehicles such that each customer is visited once and only once and the total traveling distance is minimized.

We use the notation provided in Toth and Vigo (2014) to facilitate the presentation of different CVRP models. The depot, denoted as 0, serves as the starting point for transporting goods to customers in $\mathcal{N} = 1, 2, \cdots, n$ using a homogeneous fleet $\mathcal{K} = 1, 2, \cdots, |\mathcal{K}|$. Each customer in $\mathcal{N}$ has a demand of $q_i \geq 0$, and each vehicle has a positive capacity of $Q > 0$. The cost of transportation, denoted by c_{ij}, is incurred when a vehicle travels between i and j. A vehicle's route begins at the depot, visits some or all the customers in $\mathcal{N}$, and then returns to the depot. The objective is to determine the optimal set of routes for the fleet to minimize the total cost of transportation.

The CVRP could be defined on a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, where $\mathcal{V} = \{0\} \cup \{1, 2, \cdots, n\} = \{0, 1, \cdots, n\}$, and $\mathcal{A} = \{(i, j) | i, j \in \mathcal{V}, i \neq j\}$. Let S be a subset of $\mathcal{V}$, that is, $S \subseteq \mathcal{V}$. The in-arcs and out-arcs of S are defined as follows:

- $\delta^-(S) = \{(i, j) \in \mathcal{A} | i \notin S, j \in S\}$
- $\delta^+(S) = \{(i, j) \in \mathcal{A} | i \in S, j \notin S\}$

In addition, we use $\mathcal{A}(S) = \{(i, j) \in \mathcal{A} | i \in S, j \in S\}$ to indicate all the arcs that connect nodes within S.

7.0.0.1 CVRP Instances

We use the instances taken from CVRPLIB (2014) to illustrate the modeling and solving process with Google OR-Tools. CVRPLIB (2014) contains many benchmarking instances for CVRP and we use the python package `vrplib` to load the instance `P-n16-k8.vrp` and its optimal solution `P-n16-k8.sol`.

```
# install vrplib with command: pip install vrplib
import vrplib
```

```python
# Read VRPLIB formatted instances (default)
instance = vrplib.read_instance("./data/cvrp/P-n16-k8.vrp")
solution = vrplib.read_solution("./data/cvrp/P-n16-k8.sol")
```

Let's first create a function to visualize vehicle routes, as given below.

```python
import networkx as nx
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

def show_vehicle_routes(locations, edges):
    """draw vehicles routings
    adapted from https://stackoverflow.com/a/50682819

    examples:
    locations = {
        0: (5, 5),
        1: (4, 9),
        2: (6, 4),
        3: (2, 6),
    }

    edges = [
        (0, 1, {'vehicle': '0'}),
        (1, 2, {'vehicle': '0'}),
        (2, 0, {'vehicle': '0'}),
        (0, 3, {'vehicle': '1'}),
        (3, 0, {'vehicle': '1'}),
    ]

    Args:
        locations (dict): location id -> (lat, lon)
        edges (list): list of edges
    """
    G = nx.DiGraph()
    G.add_edges_from(edges)
    plt.figure(figsize=(15,10))

    vehicles = set([e[2]['vehicle'] for e in edges])
    num_vehicles = len(vehicles)
```

```
    colors = mpl.colormaps["Set1"].colors
    for v in range(num_vehicles):
        temp = [e for e in edges if e[2]['vehicle'] == str(v)]

        color_idx = v
        if color_idx >= len(colors):
            color_idx = color_idx % len(colors)
        color = np.array([colors[color_idx]])

        nx.draw_networkx_nodes(G,
                               locations,
                               nodelist=[x[0] for x in temp],
                               node_color=color)
        nx.draw_networkx_edges(G,
                               locations,
                               edgelist=temp,
                               width=4,
                               edge_color=color,
                               style='dashed')

    #let's color the node 0 in black
    nx.draw_networkx_nodes(G, locations,
                           nodelist=[0],
                           node_color='k')

    # labels
    nx.draw_networkx_labels(G, locations,
                            font_color='w',
                            font_size=12,
                            font_family='sans-serif')

    #print out the graph
    plt.axis('off')
    plt.show()
```

Figure 7.1 shows the optimal vehicle routes for the instance P-n16-k8.vrp.

```
# visualize the optimal solution
node_coords = instance['node_coord']
locations = {}
for idx, coord in enumerate(node_coords):
```

```
    locations[idx] = (coord[0], coord[1])

routes = solution['routes']
vehicle_idx = 0
edges = []
for route in routes:
    r_temp = route.copy()
    r_temp.insert(0, 0)
    r_temp.insert(len(r_temp), 0)
    for i in range(len(r_temp) - 1):
        edges.append((r_temp[i], r_temp[i + 1], {'vehicle': str(vehicle_idx)}))

    vehicle_idx += 1

show_vehicle_routes(locations, edges)
```

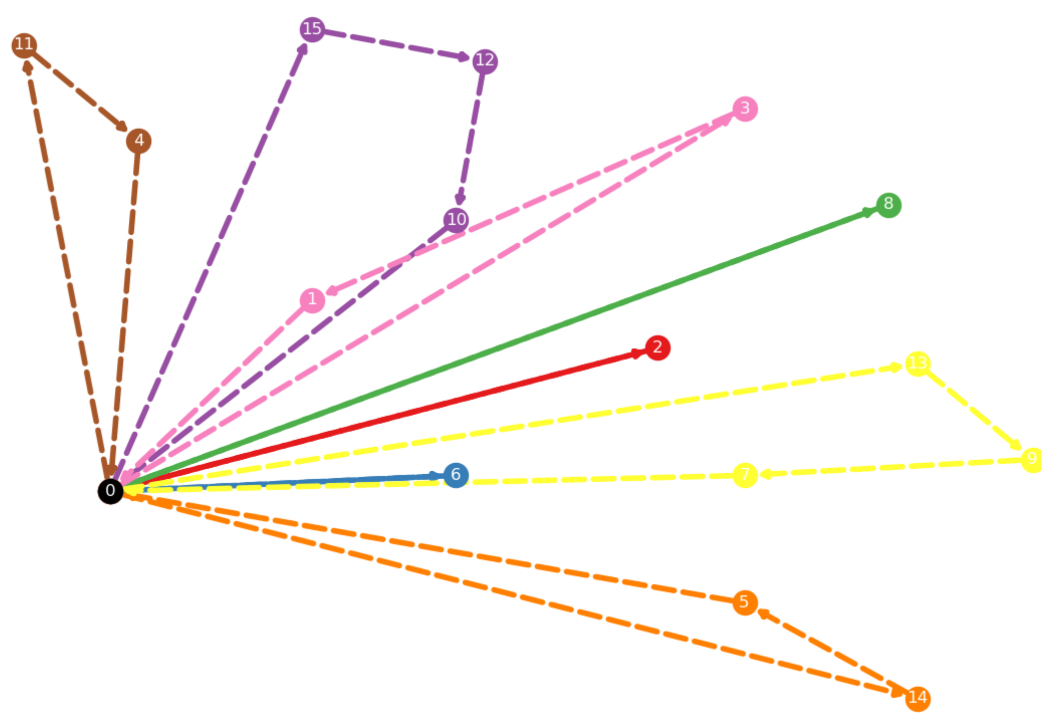

Figure 7.1: Optimal routes for instance P-n16-k8.vrp

7.0.0.2 Subtour Elimination Constraints (SECs)

7.0.0.3 Two-index Formulation - 1

This formulation was proposed by Laporte, Mercure, and Nobert (1986) and we present the formulation given in Toth and Vigo (2014). In this formulation, we define the variable x_{ij} that equals 1 if the arc (i, j) is traversed by a vehicle. The complete model is given below.

$$\text{min.} \quad \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} \tag{7.1}$$

$$\text{s.t.} \quad \sum_{j\in\delta^+(i)} x_{ij} = 1, \ \forall i \in \mathcal{N} \tag{7.2}$$

$$\sum_{i\in\delta^-(j)} x_{ij} = 1, \ \forall j \in \mathcal{N} \tag{7.3}$$

$$\sum_{j\in\delta^+(0)} x_{oj} = |\mathcal{K}| \tag{7.4}$$

$$\sum_{(i,j)\in\delta^+(S)} x_{ij} \geq r(S), \ \forall S \subseteq \mathcal{N}, S \neq \emptyset \tag{7.5}$$

$$x_{ij} \in \{0, 1\}, \ \forall (i, j) \in \mathcal{A} \tag{7.6}$$

The objective function represented by equation (7.1) is to minimize the overall transportation costs. The constraints expressed in equations (7.2) and (7.3) work together to guarantee that each customer is visited only once, with one incoming and outgoing arc. Constraints (7.4) ensure that all available vehicles are utilized to serve the customers. Constraints (7.5) prevent the formation of sub-tours. Finally, the variable types are defined by the last set of constraints, which are presented in equation (7.6).

To solve the aforementioned instance, we'll first prepare the data for our following implementation use. Let's first define two classes, `Node` and `Vehicle`, to represent a node in the network and the vechiles, respectively.

```
class Node:
    """a node is either a depot (0) or a customer
    """

    def __init__(self, id, x_coord, y_coord, demand):
        self._id = id
        self._x_coord = x_coord
        self._y_coord = y_coord
        self._demand = demand
```

```
    @property
    def id(self): return self._id

    @property
    def x_coord(self): return self._x_coord

    @property
    def y_coord(self): return self._y_coord

    @property
    def demand(self): return self._demand

    def __str__(self):
        return f"id: {self._id}, x_coord: {self._x_coord}, y_coord: {self._y_coord}, deman
```

```
class Vehicle:
    """a vehicle
    """

    def __init__(self, id, capacity):
        self._id = id
        self._capacity = capacity

    @property
    def id(self): return self._id

    @property
    def capacity(self): return self._capacity

    def __str__(self):
        return f"id: {self._id}, capacity: {self._capacity}"
```

Now let's define a class `CvrpDataCenter` to hold all the information we need later. It also has a helper function to read and parse an CVRP instance.

```
from typing import List
import vrplib
import re
from itertools import combinations

class CvrpDataCenter:
```

```python
    """this class manages all the data for a CVRP instance
    """

    def __init__(self):
        self._nodes: List = None
        self._vehicles: List = None
        self._distance: List[List] = None

    def read_cvrp_instance(self, instance_file):
        """read a given cvrp instance

        Args:
            instance_file (str): instance file
        """
        instance = vrplib.read_instance(instance_file)

        # gather nodes
        nodes = []
        idx = 0
        for node, demand in zip(instance['node_coord'],
                                instance['demand']):
            node = Node(id=idx,
                        x_coord=node[0],
                        y_coord=node[1],
                        demand=demand)
            idx += 1
            nodes.append(node)

        # gather vehicles
        comment = instance['comment']
        num_vehicles = int(re.search(r'\d', comment).group())
        vehicles = []
        for v in range(num_vehicles):
            vehicle = Vehicle(v, int(instance['capacity']))
            vehicles.append(vehicle)

        # gather distance matrix
        distance = instance['edge_weight']

        self._nodes = nodes
        self._vehicles = vehicles
```

```
        self._distance = distance

    @property
    def nodes(self): return self._nodes

    @property
    def vehicles(self): return self._vehicles

    @property
    def num_nodes(self): return len(self._nodes)

    @property
    def num_vehicles(self): return len(self._vehicles)

    @property
    def vehicle_capacity(self):
        return self._vehicles[0].capacity

    def distance(self, i, j, integer=False):
        return round(self._distance[i][j]) \
            if integer else self._distance[i][j]

    def get_all_combinations(self, numbers):
        combs = []
        for i in range(1, len(numbers) + 1):
            combs.extend(list(combinations(numbers, i)))
        return combs
```

To implement this formulation using Google OR-Tools, we first create a `CvrpDataCenter` object and read in the instance `P-n16-k8.vrp`. Then we create a `solver` object with solver option `SCIP` to solve mixed integer programming problems.

```
from ortools.linear_solver import pywraplp
import numpy as np
from itertools import product
import math

# prepare instance
cvrp_data_center = CvrpDataCenter()
cvrp_data_center.read_cvrp_instance("./data/cvrp/P-n16-k8.vrp")

# instantiate solver
```

```python
solver = pywraplp.Solver.CreateSolver('SCIP')
```

Now let's create the decision variable x_{ij}. Note that we don't need to create variables when $i = j$ since there is no arc pointing to itself in the graph $\mathcal{G}$ we defined earlier.

```python
# create decision variables
num_nodes = cvrp_data_center.num_nodes
num_vehicles = cvrp_data_center.num_vehicles
var_x = np.empty((num_nodes, num_nodes), dtype=object)
for i, j in product(range(num_nodes), range(num_nodes)):
    if i == j: continue
    var_x[i][j] = solver.BoolVar(name="x_{i, j}")
```

Then we create the objective function.

```python
# define objective function
obj_expr = [
    cvrp_data_center.distance(i, j, integer=True) * var_x[i][j]
    for i, j in product(range(num_nodes), range(num_nodes))
    if i != j
]
solver.Minimize(solver.Sum(obj_expr))
```

And we create the constraints (7.2) and (7.3).

```python
# create incoming and outgoing arc constraints
for i in range(1, num_nodes):
    out_arcs = [var_x[i][j] for j in range(num_nodes) if j != i]
    in_arcs = [var_x[j][i] for j in range(num_nodes) if j != i]
    solver.Add(solver.Sum(out_arcs) == 1)
    solver.Add(solver.Sum(in_arcs) == 1)
```

Constraints (7.4) are created as follows.

```python
# create fleet size constraint
expr = [var_x[0][i] for i in range(1, num_nodes)]
solver.Add(solver.Sum(expr) == num_vehicles)
```

```
<ortools.linear_solver.pywraplp.Constraint; proxy of <Swig Object of type 'operations_resear
```

To create the subtour elimination constraints (7.5), we first need to enumerate all the non-empty subset of $\mathcal{N}$, for which we define a helper function named `get_all_combinations()`

in the `CvrpDataCenter` class. In the code snippet below, we define a separate constraint for every nonempty customer set S, and the right-hand side $r(S)$ is defined as $\lceil q(S)/Q \rceil$.

```
# create subtour elimination constraint
nodes = cvrp_data_center.nodes
vehicle_capacity = cvrp_data_center.vehicle_capacity
customer_ids = [node.id for node in nodes if node.id > 0]
node_ids = [node.id for node in nodes]
nonempty_customer_sets = cvrp_data_center.get_all_combinations(customer_ids)
for customer_set in nonempty_customer_sets:
    others = set(node_ids).difference(customer_set)
    expr = [var_x[i][j]
            for i in customer_set
            for j in others]
    total_demand = sum([node.demand
                        for node in nodes
                        if node.id in set(customer_set)])
    rhs = math.ceil(total_demand / vehicle_capacity)
    solver.Add(solver.Sum(expr) >= rhs)
```

Putting it all together, we have the complete program below. It can be seen from the output that the optimal solution is 450 and there are 8 routes in the identified solution.

```
from ortools.linear_solver import pywraplp
import numpy as np
from itertools import product
import math

# prepare instance
cvrp_data_center = CvrpDataCenter()
cvrp_data_center.read_cvrp_instance("./data/cvrp/P-n16-k8.vrp")

# instantiate solver
solver = pywraplp.Solver.CreateSolver('SCIP')

# create decision variables
num_nodes = cvrp_data_center.num_nodes
num_vehicles = cvrp_data_center.num_vehicles
var_x = np.empty((num_nodes, num_nodes), dtype=object)
for i, j in product(range(num_nodes), range(num_nodes)):
    if i == j: continue
    var_x[i][j] = solver.BoolVar(name="x_{i, j}")
```

```python
# define objective function
obj_expr = [
    cvrp_data_center.distance(i, j, integer=True) * var_x[i][j]
    for i, j in product(range(num_nodes), range(num_nodes))
    if i != j
]
solver.Minimize(solver.Sum(obj_expr))

# create incoming and outgoing arc constraints
for i in range(1, num_nodes):
    out_arcs = [var_x[i][j] for j in range(num_nodes) if j != i]
    in_arcs = [var_x[j][i] for j in range(num_nodes) if j != i]
    solver.Add(solver.Sum(out_arcs) == 1)
    solver.Add(solver.Sum(in_arcs) == 1)

# create fleet size constraint
expr = [var_x[0][i] for i in range(1, num_nodes)]
solver.Add(solver.Sum(expr) == num_vehicles)

# create subtour elimination constraint
nodes = cvrp_data_center.nodes
vehicle_capacity = cvrp_data_center.vehicle_capacity
customer_ids = [node.id for node in nodes if node.id > 0]
node_ids = [node.id for node in nodes]
nonempty_customer_sets = cvrp_data_center.get_all_combinations(customer_ids)
for customer_set in nonempty_customer_sets:
    others = set(node_ids).difference(customer_set)
    expr = [var_x[i][j]
            for i in customer_set
            for j in others]
    total_demand = sum([node.demand
                        for node in nodes
                        if node.id in set(customer_set)])
    rhs = math.ceil(total_demand / vehicle_capacity)
    solver.Add(solver.Sum(expr) >= rhs)

status = solver.Solve()
if not status:
    opt_obj = solver.Objective().Value()
    print(f'optimal value: {opt_obj}')
```

```python
    opt_x = np.zeros((num_nodes, num_nodes))
    for i, j in product(range(num_nodes), range(num_nodes)):
        if i == j: continue
        opt_x[i][j] = int(var_x[i][j].solution_value())

    routes = []
    for i in range(1, num_nodes):
        if opt_x[0][i] == 0: continue
        # new route found
        route = []

        route_length = 0
        # add the first arc
        arc_start = 0
        arc_end = i
        route.append((arc_start, arc_end))
        route_length += cvrp_data_center.distance(arc_start,
                                                  arc_end,
                                                  integer=True)

        # add remaining arcs on the route
        arc_start = arc_end
        while True:
            for j in range(num_nodes):
                if opt_x[arc_start][j] == 1:
                    arc_end = j
                    break
            route.append((arc_start, arc_end))
            route_length += cvrp_data_center.distance(arc_start,
                                                      arc_end,
                                                      integer=True)
            if arc_end == 0: break
            arc_start = arc_end

        routes.append(route)
        print(f'route: {route}, length: {route_length}')
```

```
optimal value: 450.0
route: [(0, 1), (1, 0)], length: 28
route: [(0, 2), (2, 0)], length: 42
route: [(0, 4), (4, 11), (11, 0)], length: 57
route: [(0, 5), (5, 9), (9, 3), (3, 0)], length: 93
```

```
route: [(0, 6), (6, 0)], length: 24
route: [(0, 10), (10, 12), (12, 15), (15, 0)], length: 67
route: [(0, 13), (13, 8), (8, 0)], length: 71
route: [(0, 14), (14, 7), (7, 0)], length: 68
```

Figure 7.2 shows the routes found by the two-index formulation. Note that the routes are different from the ones in Figure 7.1 but they have the same objective value.

```
locations = {
    node.id: (node.x_coord, node.y_coord)
    for node in nodes
}

edges = []
vehicle_idx = 0
for route in routes:
    for arc in route:
        edges.append((arc[0], arc[1], {'vehicle': str(vehicle_idx)}))
    vehicle_idx += 1
edges

show_vehicle_routes(locations, edges)
```

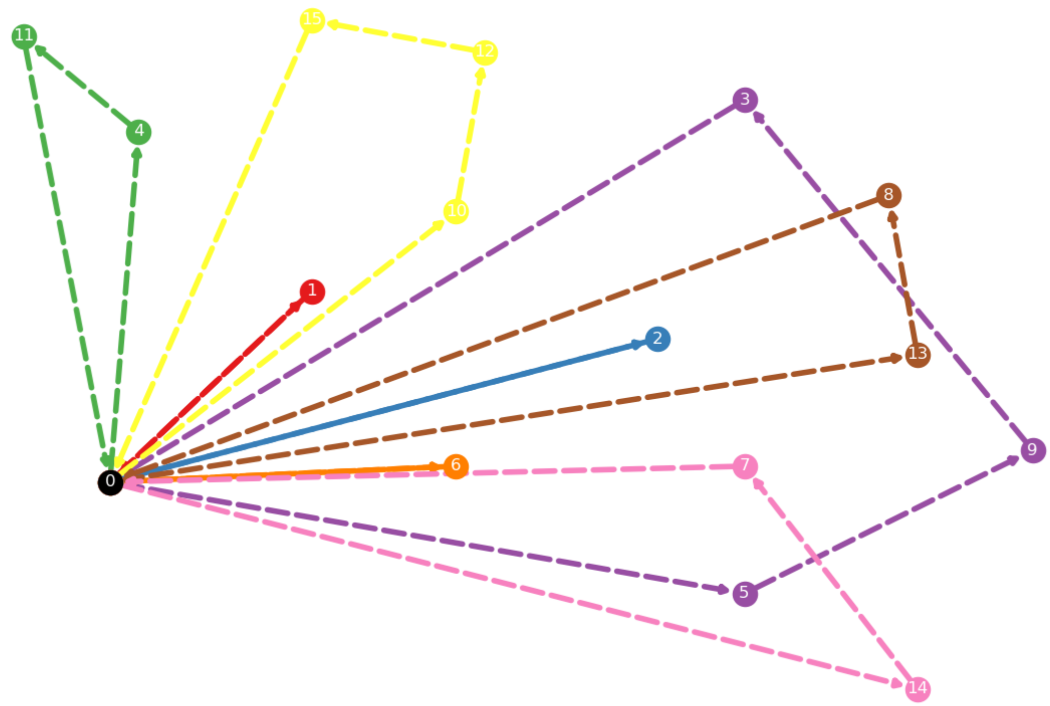

Figure 7.2: Optimal routes found by the two-index formulation

To facilitate the model comparison in following steps, we'll wrap the above program into a dedicated class `CvrpTwoIndexModel1`.

```
from ortools.linear_solver import pywraplp
from itertools import product
import numpy as np
import math

class CvrpTwoIndexModel1:
    """solve the cvrp model using the two index formulation
    """

    def __init__(self, cvrp_data_center: CvrpDataCenter):
        self._data_center: CvrpDataCenter = cvrp_data_center
        self._solver = pywraplp.Solver.CreateSolver('SCIP')
        self._var_x = None
```

```
        self._opt_obj = None
        self._opt_x = None
        self._opt_routes = None

    def read_instance(self, instance_file):
        self._data_center.read_cvrp_instance(instance_file)

    def build_model(self):
        self._create_variables()
        self._create_objective()
        self._create_constr_flow()
        self._create_constr_fleet()
        self._create_constr_subtour()

    def optimize(self):
        status = self._solver.Solve()
        if not status:
            self._retrieve_opt_solution()
            self._retrieve_opt_routes()

    def _create_variables(self):
        num_nodes = self._data_center.num_nodes
        self._var_x = np.empty((num_nodes, num_nodes), dtype=object)
        for i, j in product(range(num_nodes), range(num_nodes)):
            if i == j: continue
            self._var_x[i][j] = self._solver.BoolVar(name="x_{i, j}")

    def _create_objective(self):
        num_nodes = self._data_center.num_nodes
        obj_expr = [
            self._data_center.distance(i, j, integer=True) *
                self._var_x[i][j]
                for i, j in product(range(num_nodes), range(num_nodes))
                if i != j
        ]
        self._solver.Minimize(self._solver.Sum(obj_expr))

    def _create_constr_flow(self):
        # create incoming and outgoing arc constraints
        num_nodes = self._data_center.num_nodes
        for i in range(1, num_nodes):
```

```
            out_arcs = [self._var_x[i][j] for j in range(num_nodes) if j != i]
            in_arcs = [self._var_x[j][i] for j in range(num_nodes) if j != i]
            self._solver.Add(self._solver.Sum(out_arcs) == 1)
            self._solver.Add(self._solver.Sum(in_arcs) == 1)

    def _create_constr_fleet(self):
        # create fleet size constraint
        num_nodes = self._data_center.num_nodes
        num_vehicles = self._data_center.num_vehicles
        expr = [self._var_x[0][i] for i in range(1, num_nodes)]
        self._solver.Add(self._solver.Sum(expr) == num_vehicles)

    def _create_constr_subtour(self):
        # create subtour elimination constraint
        nodes = self._data_center.nodes
        vehicle_capacity = self._data_center.vehicle_capacity
        customer_ids = [node.id for node in nodes if node.id > 0]
        node_ids = [node.id for node in nodes]
        nonempty_customer_sets = self._data_center.get_all_combinations(customer_ids)
        for customer_set in nonempty_customer_sets:
            others = set(node_ids).difference(customer_set)
            expr = [self._var_x[i][j]
                    for i in customer_set
                    for j in others]
            total_demand = sum([node.demand
                                for node in nodes
                                if node.id in set(customer_set)])
            rhs = math.ceil(total_demand / vehicle_capacity)
            self._solver.Add(self._solver.Sum(expr) >= rhs)
        print(f"No. subtour elimination constraints: {len(nonempty_customer_sets)}")

    def show_model_summary(self):
        print(f"No. of variables: {self._solver.NumVariables()}")
        print(f"No. of constraints: {self._solver.NumConstraints()}")

    def _retrieve_opt_solution(self):
        self._opt_obj = self._solver.Objective().Value()
        print(f'optimal value: {self._opt_obj}')

        num_nodes = self._data_center.num_nodes
        self._opt_x = np.zeros((num_nodes, num_nodes))
```

```
        for i, j in product(range(num_nodes), range(num_nodes)):
            if i == j: continue
            self._opt_x[i][j] = int(self._var_x[i][j].solution_value())

    def _retrieve_opt_routes(self):
        num_nodes = self._data_center.num_nodes
        self._routes = []
        for i in range(1, num_nodes):
            if self._opt_x[0][i] == 0: continue
            # new route found
            route = []

            route_length = 0
            # add the first arc
            arc_start = 0
            arc_end = i
            route.append((arc_start, arc_end))
            route_length += self._data_center\
                    .distance(arc_start,
                            arc_end,
                            integer=True)

            # add remaining arcs on the route
            arc_start = arc_end
            while True:
                for j in range(num_nodes):
                    if self._opt_x[arc_start][j] == 1:
                        arc_end = j
                        break
                route.append((arc_start, arc_end))
                route_length += self._data_center\
                    .distance(arc_start,
                            arc_end,
                            integer=True)
                if arc_end == 0: break
                arc_start = arc_end

            self._routes.append(route)
            print(f'route: {route}, length: {route_length}')
```

The code below validates that the same optimal solution is obtained using this object-oriented approach.

```python
cvrp_two_index_model_1 = CvrpTwoIndexModel1(CvrpDataCenter())
cvrp_two_index_model_1.read_instance("./data/cvrp/P-n16-k8.vrp")
cvrp_two_index_model_1.build_model()
cvrp_two_index_model_1.show_model_summary()
cvrp_two_index_model_1.optimize()
```

```
No. subtour elimination constraints: 32767
No. of variables: 240
No. of constraints: 32798
optimal value: 450.0
route: [(0, 1), (1, 0)], length: 28
route: [(0, 2), (2, 0)], length: 42
route: [(0, 4), (4, 11), (11, 0)], length: 57
route: [(0, 5), (5, 9), (9, 3), (3, 0)], length: 93
route: [(0, 6), (6, 0)], length: 24
route: [(0, 10), (10, 12), (12, 15), (15, 0)], length: 67
route: [(0, 13), (13, 8), (8, 0)], length: 71
route: [(0, 14), (14, 7), (7, 0)], length: 68
```

It can be seen from the model output that there are a total of 32798 constraints, out of which 32767 are subtour elimination constraints, even for such a small instance with only 15 customers. In the next section, we will present another two index formulation to handle this exponential number of constraints.

7.0.0.4 Two-index Formulation - 2

7.0.0.5 Three-index Formulation

This formulation is also known as the *MTZ-formulation* as a new set of constraints initially proposed for traveling salesman problem (Miller, Tucker, and Zemlin (1960)) is used to eliminate subtours.

7.0.0.6 Commodity-flow Formulation

8 Column Generation

Column generation is a technique used in linear programming to solve problems with a large number of variables. Some classical/typical applications of column generation include:

Transportation and distribution problems: Column generation can be used to optimize transportation and distribution networks by determining the most efficient routes for goods and services.

Crew scheduling: Column generation is useful in determining optimal crew scheduling for airlines, railways, and other transportation companies.

Cutting stock problems: In the manufacturing industry, column generation can be used to optimize cutting stock problems by finding the best way to cut raw materials into smaller pieces to minimize waste.

Network design: Column generation can be applied to network design problems, such as determining the optimal location of facilities in a supply chain network.

Vehicle routing: Column generation can be used to optimize vehicle routing problems, such as determining the best routes for delivery trucks or garbage trucks.

Resource allocation: Column generation can also be applied to resource allocation problems, such as scheduling employees or assigning tasks to machines in a production facility.

Overall, column generation is a powerful technique that can be applied to a wide range of optimization problems.

Part II

Constraint Programming

Part III

Tailored Algorithms

9 Summary

In summary, this book has no content whatsoever.

References

Beasley, John E. 1990. "OR-Library: Distributing Test Problems by Electronic Mail." *Journal of the Operational Research Society* 41 (11): 1069–72.

CVRPLIB. 2014. "CVRPLIB." http://vrp.galgos.inf.puc-rio.br/index.php/en/.

Dantzig, G., R. Fulkerson, and S. Johnson. 1954. "Solution of a Large-Scale Traveling-Salesman Problem." *Journal of the Operations Research Society of America* 2 (4): 393–410. https://doi.org/10.1287/opre.2.4.393.

Kondili, E, CC Pantelides, and R WH Sargent. 1988. "A General Algorithm for Scheduling Batch Operations." In. Barton, ACT. https://search.informit.org/doi/10.3316/informit.394925233030714.

Ku, Wen-Yang, and J. Christopher Beck. 2016. "Mixed Integer Programming Models for Job Shop Scheduling: A Computational Analysis." *Computers & Operations Research* 73 (September): 165–73. https://doi.org/10.1016/j.cor.2016.04.006.

Laporte, Gilbert, Hélène Mercure, and Yves Nobert. 1986. "An Exact Algorithm for the Asymmetrical Capacitated Vehicle Routing Problem." *Networks* 16 (1): 33–46. https://doi.org/10.1002/net.3230160104.

Manne, Alan S. 1960. "On the Job-Shop Scheduling Problem." *Operations Research* 8 (2): 219–23. https://doi.org/10.1287/opre.8.2.219.

Miller, C. E., A. W. Tucker, and R. A. Zemlin. 1960. "Integer Programming Formulation of Traveling Salesman Problems." *Journal of the ACM* 7 (4): 326–29. https://doi.org/10.1145/321043.321046.

Toth, Paolo, and Daniele Vigo. 2014. *Vehicle Routing: Problems, Methods, and Applications*. SIAM.

Wagner, Harvey M. 1959. "An Integer Linear-Programming Model for Machine Scheduling." *Naval Research Logistics Quarterly* 6 (2): 131–40. https://doi.org/10.1002/nav.3800060205.