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.

$$\text{min.} \quad \sum_{s \in \mathcal{S}} \sum_{d \in \mathcal{D}} c_{sd} x_{sd} \tag{0.1}$$

$$\text{s.t.} \quad \sum_{d \in \mathcal{D}} x_{sd} = p_s, \ \forall s \in \mathcal{S} \tag{0.2}$$

$$\sum_{s \in \mathcal{S}} x_{sd} = m_d, \ \forall d \in \mathcal{D} \tag{0.3}$$

$$x_{sd} \geq 0, \ \forall s \in \mathcal{S}, d \in \mathcal{D} \tag{0.4}$$

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.

```
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)]
```

```
52            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.

```
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_researcl
```

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_researcl
```

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_resear
```

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
```

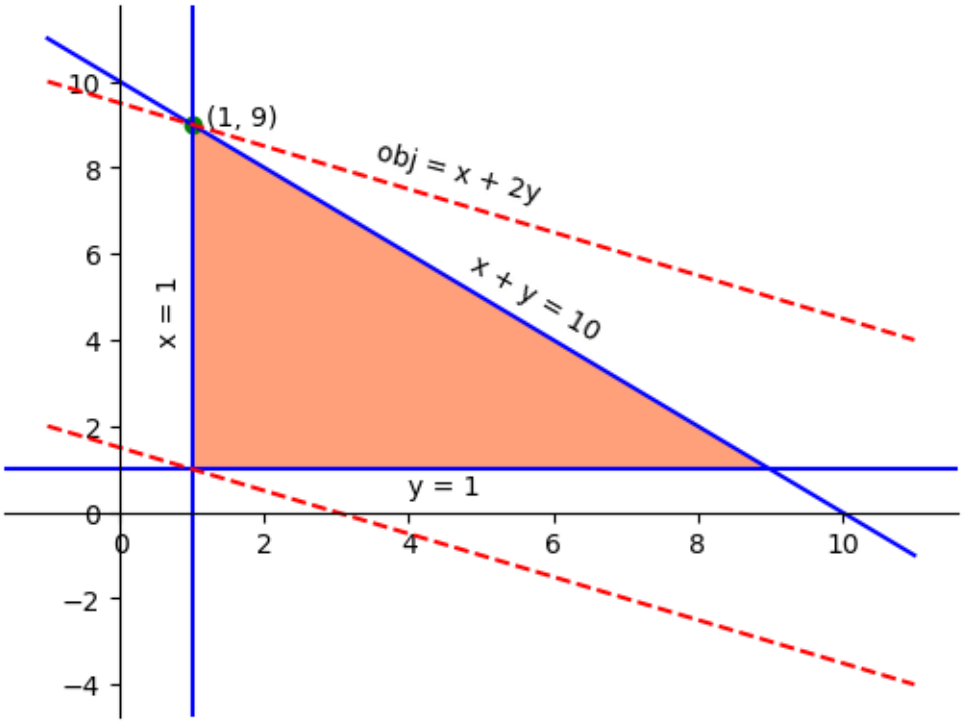


Figure 3.1: A simple LP example

```
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:

```
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
```

```
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()
```

```
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]]
```

```
# 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.

```
from ortools.linear_solver import pywraplp

# import instance data
```

```
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.

$$\text{min.} \quad 0 \tag{3.15}$$

$$\text{s.t.} \quad \sum_{j \in S} x_{ijk} = 1, \ \forall i, k \in S \tag{3.16}$$

$$\sum_{i \in S} x_{ijk} = 1, \ \forall j, k \in S \tag{3.17}$$

$$\sum_{k \in S} x_{ijk} = 1, \ \forall i, j \in S \tag{3.18}$$

$$\sum_{(i-1)\times 3+1}^{(i-1)\times 3+3} \sum_{(j-1)\times 3+1}^{(j-1)\times 3+3} x_{ijk} = 1, \ \forall i, j \in \{1, 2, 3\}, k \in S \tag{3.19}$$

$$x_{ijk} = 1, \ \forall (i, j, k) \in M \tag{3.20}$$

$$x_{ijk} \in \{0, 1\}, \ \forall i, j, k \in S \tag{3.21}$$

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.

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

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^j_h, cannot start unless its preceeding operation o^j_{h-1} 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')
```

```
39
40  # create objective
41  solver.Minimize(var_makespan)
42
43  # create constraints
44  for job, job_operations in enumerate(operations):
45      for h in range(1, num_machines):
46          curr_machine = job_operations[h]
47          prev_machine = job_operations[h - 1]
48          prev_time = processing_times[job][prev_machine]
49          solver.Add(
50              var_time[curr_machine][job] >=
51                  var_time[prev_machine][job] +
52                  prev_time
53              )
54
55  V = 0
56  for job in processing_times:
57      V += sum(processing_times[job].values())
58  for machine in range(num_machines):
59      for job_j in range(num_jobs - 1):
60          for job_k in range(job_j + 1, num_jobs):
61              solver.Add(
62                  var_time[machine][job_j] >=
63                  var_time[machine][job_k] +
64                  processing_times[job_k][machine] -
65                  V * var_prec[machine][(job_j, job_k)]
66              )
67              solver.Add(
68                  var_time[machine][job_k] >=
69                  var_time[machine][job_j] +
70                  processing_times[job_j][machine] -
71                  V * (1 - var_prec[machine][(job_j, job_k)])
72              )
73
74  for job in range(num_jobs):
75      last_oper_machine = operations[job][-1]
76      solver.Add(
77          var_makespan >=
78          var_time[last_oper_machine][job] +
79          processing_times[job][last_oper_machine]
80      )
```

```
81
82  status = solver.Solve()
83
84  if status == solver.OPTIMAL:
85      print(f"min. makespan = {solver.Objective().Value():.2f}")
86
87      opt_time = []
88      for machine in range(num_machines):
89          arr = [
90              int(var_time[machine][job].solution_value())
91              for job in range(num_jobs)
92          ]
93          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)
```

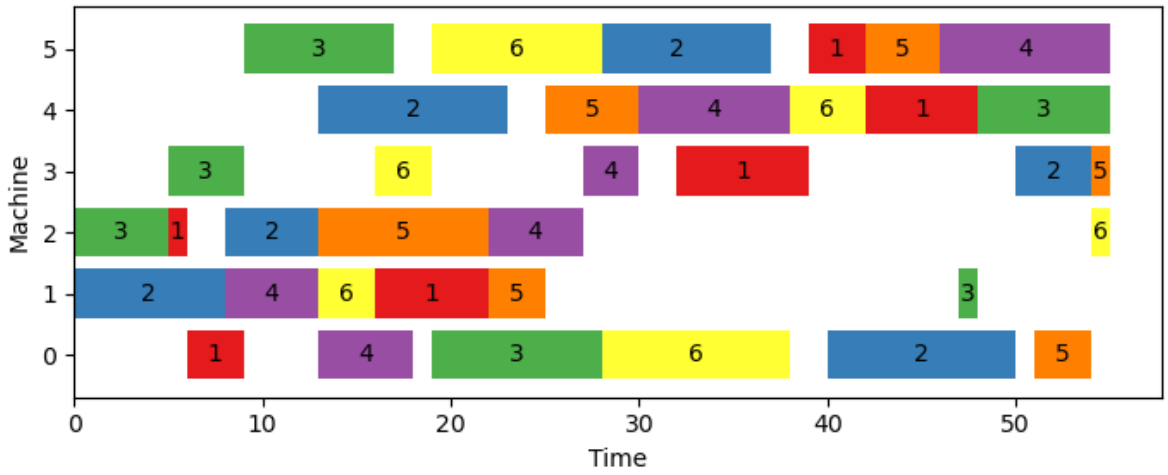


Figure 5.1: Optimal solution of the ft06 instance using the disjunctive formulation

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.

$$\text{min.} \quad C_{max} \tag{5.8}$$

$$\text{s.t.} \quad \sum_{t \in H} x_{ijt} = 1, \ \forall j \in \mathcal{J}, i \in \mathcal{M} \tag{5.9}$$

$$\sum_{t \in H} (t + p_{ij}) \cdot x_{ijt} \leq C_{max}, \ \forall j \in \mathcal{J}, i \in \mathcal{M} \tag{5.10}$$

$$\sum_{j \in \mathcal{J}} \sum_{t' \in T_{ijt}} x_{ijt'} \leq 1, \ \forall i \in \mathcal{M}, t \in H, T_{ijt} = \{t - p_{ij} + 1, \cdots, t\} \tag{5.11}$$

$$\sum_{t \in H} (t + p_{o^j_{h-1}}, j) \cdot x_{o^j_{h-1}, jt} \leq \sum_{t \in H} t \cdot x_{o^j_h, jt}, \ \forall j \in \mathcal{J}, h = 2, \cdots, m \tag{5.12}$$

$$x_{ijt} \in \{0, 1\}, \ \forall j \in \mathcal{J}, i \in \mathcal{M}, t \in H \tag{5.13}$$

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 - $H = \sum_{i \in \mathcal{J}, j \in \mathcal{J}} p_{ij}$. 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)
```

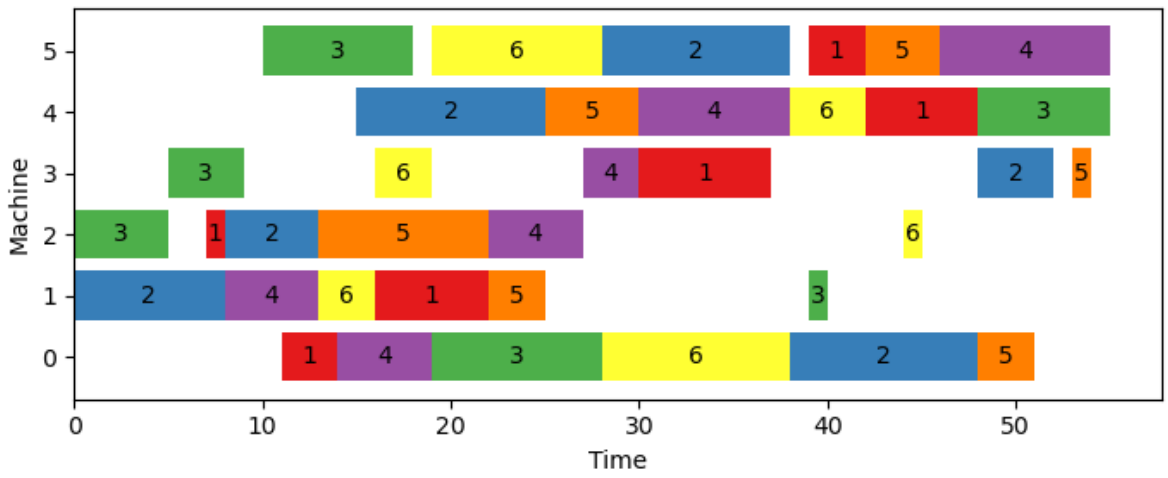


Figure 5.2: Optimal solution of the ft06 instance using the time-indexed formulation

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.

$$\text{min.} \quad C_{max} \tag{5.14}$$

$$\text{s.t.} \quad \sum_{j\in\mathcal{J}} x_{ijk} = 1, \ \forall i \in \mathcal{M}, k = 1, \cdots, n \tag{5.15}$$

$$\sum_{k=1}^{n} x_{ijk} = 1, \ \forall i \in \mathcal{M}, j \in \mathcal{J} \tag{5.16}$$

$$h_{ik} + \sum_{j\in\mathcal{J}} p_{ij} x_{ijk} \leq h_{i,k+1}, \ \forall i \in \mathcal{M}, k = 1, \cdots, n-1 \tag{5.17}$$

$$\sum_{i\in\mathcal{M}} r_{ijl} h_{ik} + \sum_{i\in\mathcal{M}} r_{ijl} p_{ij} \leq V \cdot (1 - \sum_{i\in\mathcal{M}} r_{ijl} x_{ijk}) + V \cdot (1 - \sum_{i\in\mathcal{M}} r_{ij,l+1} x_{ijk'}) + \sum_{i\in\mathcal{M}} r_{ij,l+1} h_{ik'}, \quad \forall j \in \mathcal{J}, k, k' = 1, \cdots, n, l = 1, \cdots, m-1 \tag{5.18}$$

$$h_{in} + \sum_{j\in\mathcal{J}} p_{ij} x_{ijk} \leq C_{max}, \ \forall i \in \mathcal{M} \tag{5.19}$$

$$h_{ik} \geq 0, \ \forall i \in \mathcal{M}, k = 1, \cdots, n \tag{5.20}$$

$$x_{ijk} \in \{0, 1\}, \ \forall j \in \mathcal{J}, i \in \mathcal{M}, k = 1, \cdots, n \tag{5.21}$$

$$C_{max} \geq 0 \tag{5.22}$$

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)
```

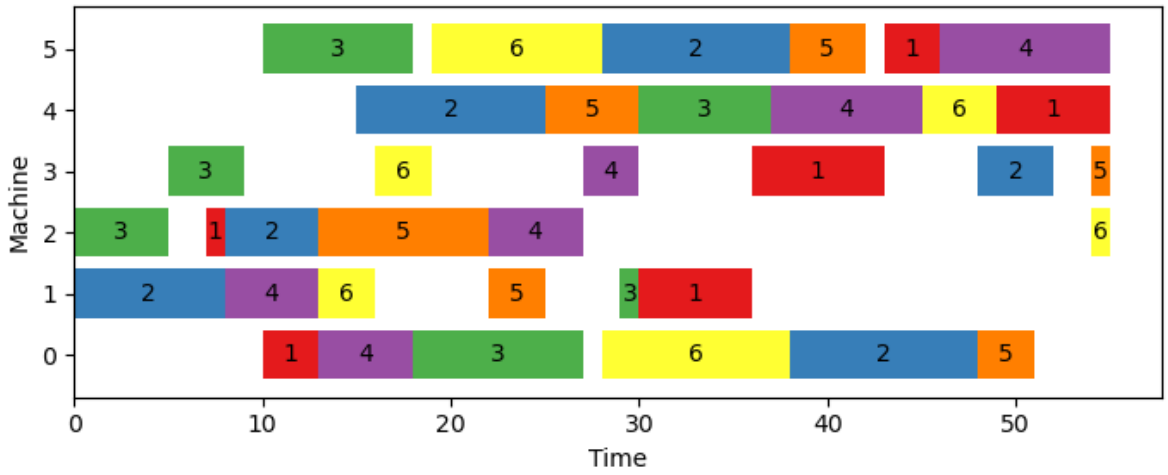


Figure 5.3: Optional solution found by the rank-based model for instance ft06

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)
```

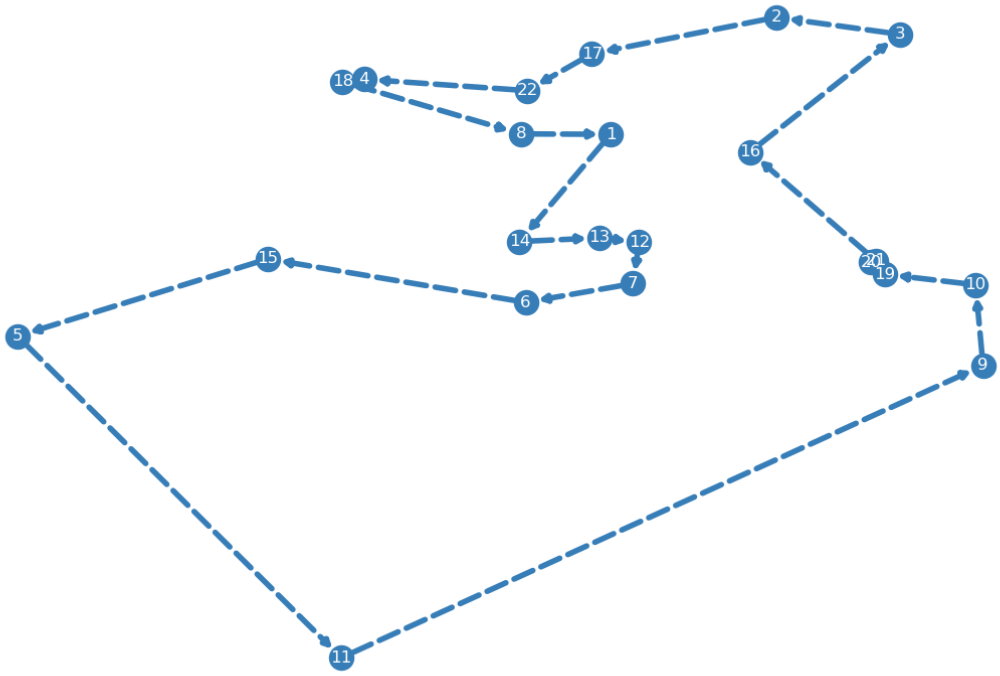


Figure 6.1: Optimal tour of the *ulysses22* instance

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')
```

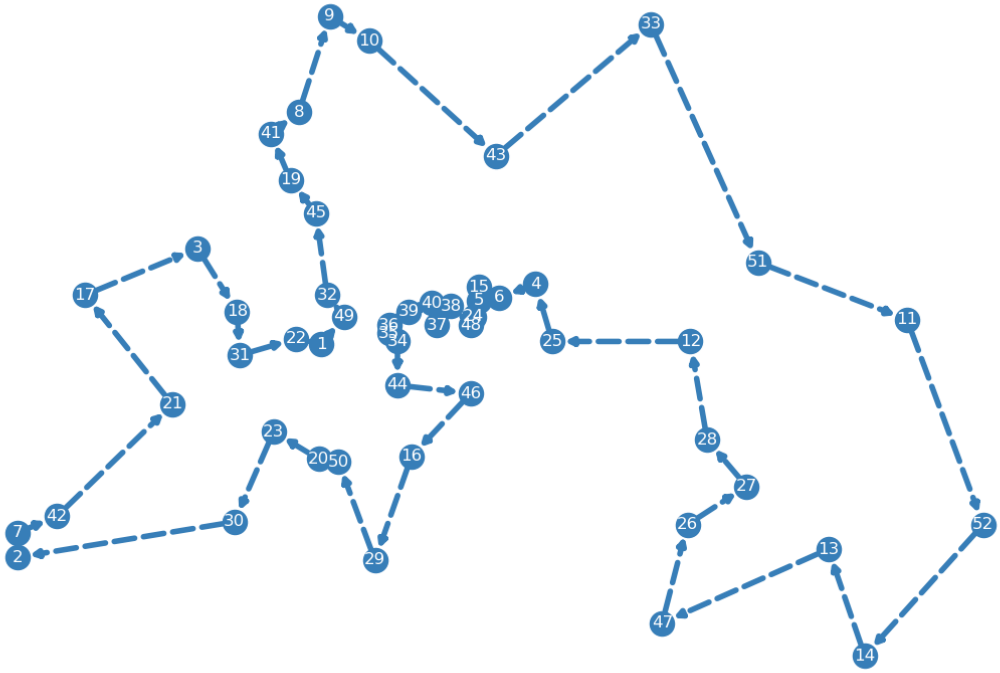


Figure 6.2: Optimal tour of the *berlin52* instance

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

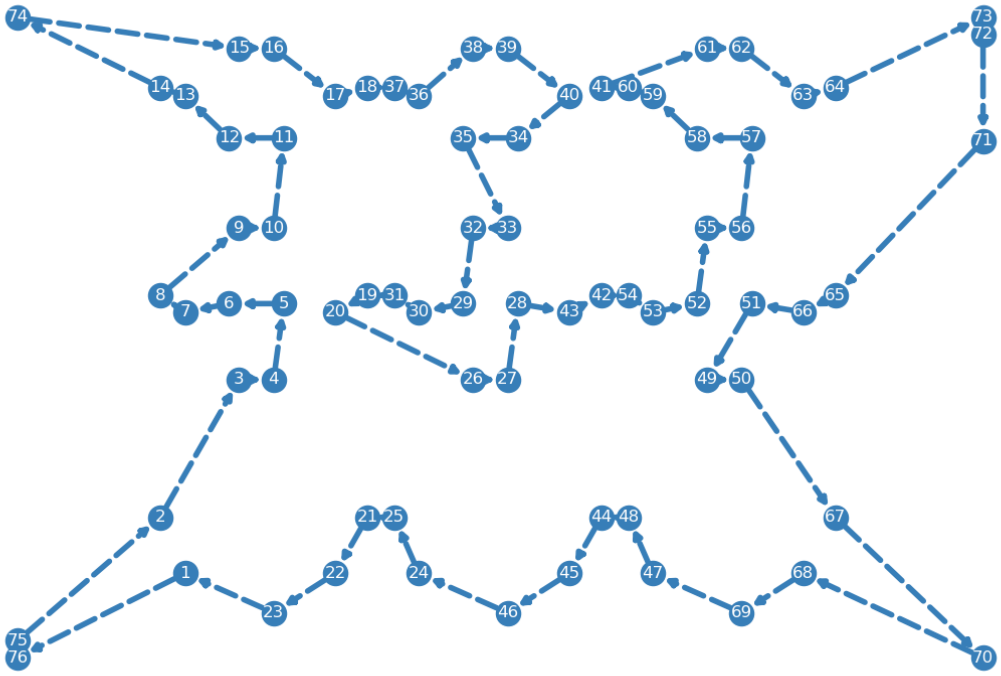


Figure 6.3: Optimal tour of the *pr76* instance

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:

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

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

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

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

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

This specific TSP formulation aims to find the optimal route with the shortest total distance. To ensure that each node is visited exactly once, constraints (6.2) and (6.3), also known as *degree constraints*, are used. Another set of constraints (6.4), called subtour elimination constraints, ensure that the solution does not contain any subtours. Subtours are smaller cycles within the larger route that violate the requirement of visiting each city exactly once. Two examples of solutions with subtours are shown in figures Figure 6.4 and Figure 6.5. While these solutions satisfy the degree constraints, they violate the subtour elimination constraints. A subtour contains the same number of edges (or arcs) as nodes, so limiting the number of edges to be less than the number of nodes can help to eliminate subtours. Furthermore, because of the presence of degree constraints, subtours with only one node cannot exist, and similarly, subtours with $n-1$ nodes are also impossible. Therefore, it is acceptable to define the subtour elimination constraints only for subtours that contain between 2 and $n-2$ nodes.

An equivalent way of constraints (6.4) is

$$\sum_{i\in S}\sum_{j\notin S} x_{ij} \geq 1,\ \forall S \subset \mathcal{V},\ 2 \leq |S| \leq n-2 \tag{6.6}$$

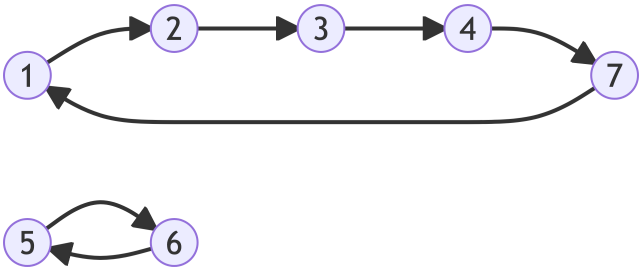


Figure 6.4: Subtour with 2 nodes

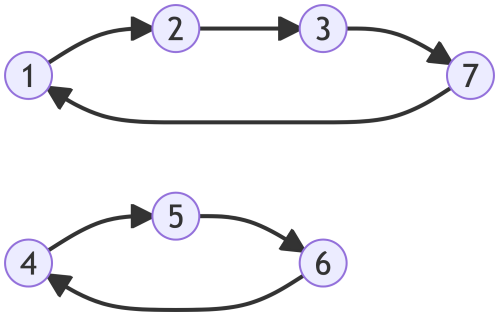


Figure 6.5: Subtour with 3 nodes

To understand this, observe that the value of $|S|$ can be calculated as the sum of two terms: the sum of the decision variables x_{ij} for all edges (i, j) that are included in the subset, and the sum of the decision variables x_{ij} for all edges that cross the boundary of the subtour, that is, $|S| = \sum_{i,j \in S,\ (i,j) \in \mathcal{A}} x_{ij} + \sum_{i \in S} \sum_{j \notin S} x_{ij}$. This means that the constraints (6.4) and (6.6) are interchangeable, as they represent the same condition in different forms.

To simplify the implementation of different TSP formulations that will be presented in the following sections, we have created a base class called `TspModel` in the code below. This class contains the instance information that needs to be solved, as well as several helper functions. Within the class definition, the attribute `_node_list` holds the list of nodes that must be visited by the TSP tour. The attribute `_node_coords` is a dictionary that stores the location information for each node. Finally, the attribute `_distance` is another dictionary that provides the distance between any pair of nodes. The `read_inputs()` function, defined between lines 15 and 31, takes a TSP problem instance and extracts the necessary information for solving the problem later. The `get_combinations()` function, defined between lines 33 and 34, generates all possible combinations of nodes with the specified size.

```
from typing import Dict, List
from itertools import combinations
import tsplib95

class TspModel:
    """base class for TSP models
    """

    def __init__(self, name: str):
        self._name: str = name
        self._node_list: List[int] = None
        self._node_coords: Dict[int, List[float, float]] = None
        self._distance: Dict[int, Dict[int, int]] = None

    def read_inputs(self, instance_file: str) -> None:
        problem = tsplib95.load(instance_file)

        node_coords = problem.node_coords
        self._node_coords = {
            id: node_coords[id]
            for id in node_coords
        }
        self._node_list = list(node_coords.keys())

        self._distance = {}
```

```
        for i in self._node_list:
            dist = {
                j: problem.get_weight(i, j)
                for j in self._node_list
            }
            self._distance[i] = dist

    def get_combinations(self, size):
        return list(combinations(self._node_list, size))

    @property
    def name(self): return self._name

    @property
    def num_nodes(self): return len(self._node_coords)
```

To implement the TSP model using OR-Tools, we define the `TspModelV1` class that inherits from the base class `TspModel`. The constructor initializes a solver object and defines attributes to store decision variables and the optimal solution. The `_create_variables()` function creates binary decision variables for every arc in the problem, while the `_create_objective()` function calculates the total traveling cost. Degree constraints are defined in the `_create_degree_constraints()` function, while the subtour elimination constraints are defined in the `_create_subtour_elimination_constraints()` function. The `build_model()` function needs to be called before optimize() to construct the model.

```
from itertools import product
from ortools.linear_solver import pywraplp

class TspModelV1(TspModel):

    def __init__(self, name='Tspmodel_v1'):
        super().__init__(name)

        self._solver = pywraplp.Solver.CreateSolver('SCIP')
        self._var_x = None
        self._opt_obj = None
        self._opt_x = None
        self._opt_route = None

    def build_model(self):
        self._create_variables()
```

```
        self._create_objective()
        self._create_degree_constraints()
        self._create_subtour_elimination_constraints()

    def optimize(self, enable_output: bool):
        if enable_output: self._solver.EnableOutput()
        status = self._solver.Solve()
        if status is pywraplp.Solver.OPTIMAL:
            self._retrieve_opt_solution()
            self._retrieve_opt_route()

    def _create_variables(self):
        self._var_x = {}
        for i in self._node_list:
            self._var_x[i] = {
                j: self._solver.BoolVar('x_{i, j}')
                for j in self._node_list
                if j != i
            }

    def _create_objective(self):
        node_list = self._node_list
        expr = [self._distance[i][j] * self._var_x[i][j]
                for i, j in product(node_list, node_list)
                if i != j]
        self._solver.Minimize(self._solver.Sum(expr))

    def _create_degree_constraints(self):
        for i in self._node_list:
            out_expr = [self._var_x[i][j]
                        for j in self._node_list
                        if j != i
                        ]
            in_expr = [self._var_x[j][i]
                       for j in self._node_list
                       if j != i
                       ]
            self._solver.Add(self._solver.Sum(out_expr) == 1)
            self._solver.Add(self._solver.Sum(in_expr) == 1)

    def _create_subtour_elimination_constraints(self):
```

```
        num_nodes = self.num_nodes
        for size in range(2, num_nodes - 1):
            combinations = self.get_combinations(size)
            for comb in combinations:
                expr = [self._var_x[i][j]
                        for i, j in product(comb, comb)
                        if i != j]
                self._solver.Add(self._solver.Sum(expr) <=
                                 len(comb) - 1)

    def _retrieve_opt_solution(self):
        self._opt_obj = float(self._solver.Objective().Value())
        self._opt_x = {}
        for i in self._node_list:
            self._opt_x[i] = {
                j: round(self._var_x[i][j].solution_value())
                for j in self._node_list
                if j != i
            }

        print(f'optimal value = {self._opt_obj:.2f}')

    def _retrieve_opt_route(self):
        self._opt_route = []
        route_start = list(self._opt_x.keys())[0]
        edge_start = route_start
        while True:
            for n in self._opt_x[edge_start]:
                if self._opt_x[edge_start][n] == 0: continue

                edge_end = n
                self._opt_route.append((edge_start, edge_end))
                break

            if edge_end == route_start: break
            edge_start = edge_end

    def show_opt_route(self):
        TspVisualizer.show(self._node_coords, self._opt_route)
```

```
    def show_model_info(self):
        print(f"Number of variables: {self._solver.NumVariables()}")
        print(f"Number of constraints: {self._solver.NumConstraints()}")
```

Now we use the `TspModelV1` class to solve the *burma14* instance and plots the optimal solution found by the OR-Tools in Figure 6.6.

```
instance = './data/tsp/burma14.tsp'

model_v1 = TspModelV1()
model_v1.read_inputs(instance_file=instance)
model_v1.build_model()
model_v1.show_model_info()
model_v1.optimize(False)
model_v1.show_opt_route()
```

```
Number of variables: 182
Number of constraints: 16382
optimal value = 3323.00
```

Figure 6.6: Optimal route for the *burma14* instance

6.4 Model 2 - MTZ

In this formulation, an alternative way of modeling the subtour elimination constraints was proposed by Miller, Tucker, and Zemlin (1960). The model uses two types of 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
- u_i: a continuous variable for $i \in \mathcal{V} \backslash \{1\}$

The complete formulation is given below.

$$\text{min.} \quad \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} \tag{6.7}$$

$$\text{s.t.} \quad \sum_{j\in\mathcal{V},\ j\neq i} x_{ij} = 1,\ \forall i \in \mathcal{V} \tag{6.8}$$

$$\sum_{i\in\mathcal{V},\ i\neq j} x_{ij} = 1,\ \forall j \in \mathcal{V} \tag{6.9}$$

$$u_i - u_j + (n-1)x_{ij} \leq n-2, \quad \forall i,j \in \{2,\cdots,n\}, i \neq j \tag{6.10}$$

$$x_{ij} \in \{0,1\},\ \forall (i,j) \in \mathcal{A} \tag{6.11}$$

$$1 \leq n_i \leq n-1, i = 2,\cdots,n \tag{6.12}$$

In this formulation, constraints (6.8) and (6.9) serve as the degree constraints requiring that there is only one arc entering and leaving a node. Constraints (6.10) are the new subtour elimination constraints. To see how constraints (6.10) effectively forbid subtours, let's examine the example below in Figure 6.7. In the figure, nodes 5 and 6 form a subtour and constraints (6.10) become:

$$u_5 - u_6 + (7-1) * 1 \leq 7-2 \tag{6.13}$$

$$u_6 - u_5 + (7-1) * 1 \leq 7-2 \tag{6.14}$$

$$\tag{6.15}$$

which translate to:

$$u_5 \leq u_6 - 1 \tag{6.16}$$

$$u_6 \leq u_5 - 1 \tag{6.17}$$

and thus:

$$u_5 \leq u_6 - 1 \tag{6.18}$$

$$\leq u_5 - 1 - 1 \tag{6.19}$$

$$\leq u_5 - 2 \tag{6.20}$$

for which, we have $0 \leq -2$, which is obviously wrong. Therefore, any subtour will violate the constraints defined in (6.10).

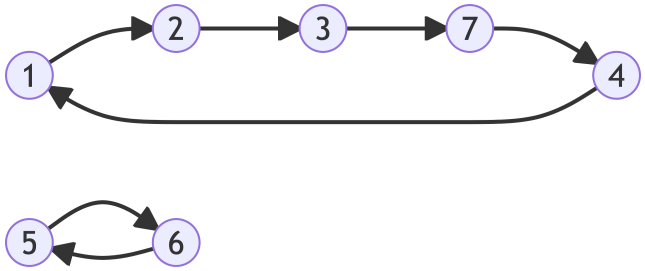


Figure 6.7: Subtour with 2 nodes

The code below gives the complete program of the formulation. The new variable u_i is defined in function `_create_variables()` and the subtour elimination constraints are updated in function `_create_subtour_elimination_constraints()`.

```
from itertools import product
from ortools.linear_solver import pywraplp

class TspModelV2(TspModel):

    def __init__(self, name='Tspmodel_v2'):
        super().__init__(name)

        self._solver = pywraplp.Solver.CreateSolver('SCIP')
        self._var_x = None
        self._var_u = None
        self._opt_obj = None
        self._opt_x = None
        self._opt_route = None

    def build_model(self):
        self._create_variables()
        self._create_objective()
        self._create_degree_constraints()
        self._create_subtour_elimination_constraints()

    def optimize(self, enable_output: bool):
        if enable_output: self._solver.EnableOutput()
        status = self._solver.Solve()
        if status is pywraplp.Solver.OPTIMAL:
```

```
        self._retrieve_opt_solution()
        self._retrieve_opt_route()

    def _create_variables(self):
        self._var_x = {}
        for i in self._node_list:
            self._var_x[i] = {
                j: self._solver.BoolVar(f'x_{i, j}')
                for j in self._node_list
                if j != i
            }

        self._var_u = {
            i: self._solver.NumVar(1, self.num_nodes, f'v_{i}')
            for i in self._node_list
            if i != 1
        }

    def _create_objective(self):
        node_list = self._node_list
        expr = [self._distance[i][j] * self._var_x[i][j]
                for i, j in product(node_list, node_list)
                if i != j]
        self._solver.Minimize(self._solver.Sum(expr))

    def _create_degree_constraints(self):
        for i in self._node_list:
            out_expr = [self._var_x[i][j]
                        for j in self._node_list
                        if j != i
                        ]
            in_expr = [self._var_x[j][i]
                       for j in self._node_list
                       if j != i
                       ]
            self._solver.Add(self._solver.Sum(out_expr) == 1)
            self._solver.Add(self._solver.Sum(in_expr) == 1)

    def _create_subtour_elimination_constraints(self):
        num_nodes = self.num_nodes
        for i, j in product(self._node_list, self._node_list):
```

```
            if i == j: continue
            if i == 1 or j == 1: continue
            self._solver.Add(self._var_u[i] -
                             self._var_u[j] +
                             (num_nodes - 1) * self._var_x[i][j] <=
                             num_nodes - 2)

    def _retrieve_opt_solution(self):
        self._opt_obj = float(self._solver.Objective().Value())
        self._opt_x = {}
        for i in self._node_list:
            self._opt_x[i] = {
                j: round(self._var_x[i][j].solution_value())
                for j in self._node_list
                if j != i
            }

        print(f'optimal value = {self._opt_obj:.2f}')

    def _retrieve_opt_route(self):
        self._opt_route = []
        route_start = list(self._opt_x.keys())[0]
        edge_start = route_start
        while True:
            for n in self._opt_x[edge_start]:
                if self._opt_x[edge_start][n] == 0: continue

                edge_end = n
                self._opt_route.append((edge_start, edge_end))
                break

            if edge_end == route_start: break
            edge_start = edge_end

    def show_opt_route(self):
        TspVisualizer.show(self._node_coords, self._opt_route)


    def show_model_info(self):
        print(f"Number of variables: {self._solver.NumVariables()}")
        print(f"Number of constraints: {self._solver.NumConstraints()}")
```

We now use this formulation to solve the *burma14* instance and show its optimal solution in Figure 6.8.

```
instance = './data/tsp/burma14.tsp'

model_v2 = TspModelV2()
model_v2.read_inputs(instance_file=instance)
model_v2.build_model()
model_v2.show_model_info()
model_v2.optimize(False)
model_v2.show_opt_route()
```

```
Number of variables: 195
Number of constraints: 184
optimal value = 3323.00
```

Figure 6.8: Optimal route for the *burma14* instance

Figure 6.9 shows the optimal solution found by the formulation for the *ulysses22* instance.

```
instance = './data/tsp/ulysses22.tsp'

model_v2 = TspModelV2()
model_v2.read_inputs(instance_file=instance)
model_v2.build_model()
model_v2.show_model_info()
model_v2.optimize(False)
model_v2.show_opt_route()
```

```
Number of variables: 483
Number of constraints: 464
optimal value = 7013.00
```

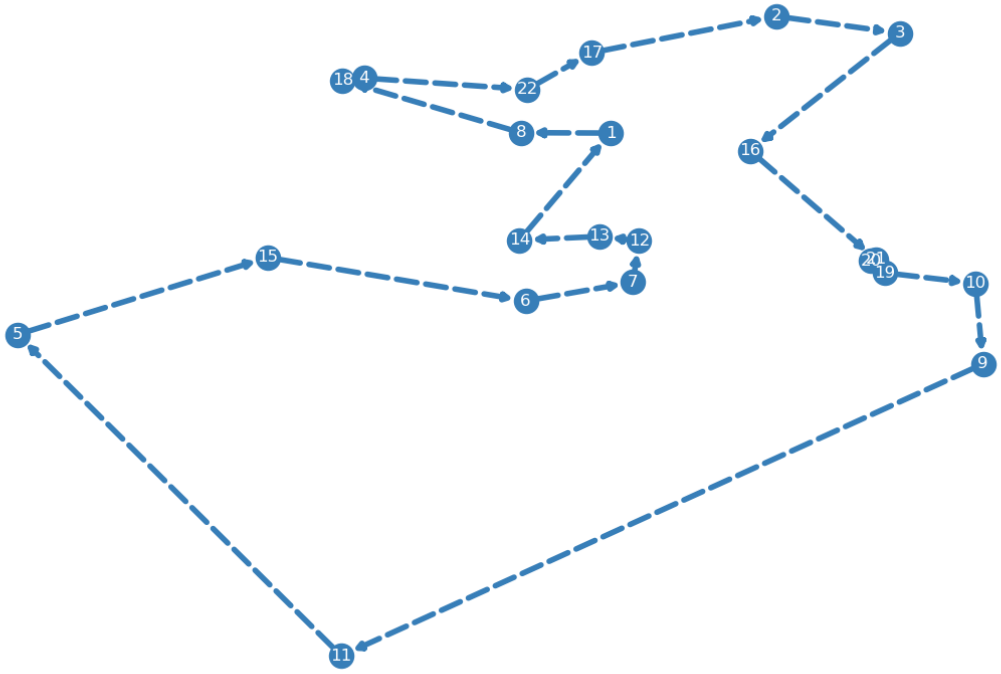


Figure 6.9: Optimal route for the *ulysses22* instance

6.5 Model 3 - Single Commodity Flow

The model uses two types of 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
- y_{ij}: a continuous variable representing the flow on arc $(i, j) \in \mathcal{A}$

The complete formulation is given below.

$$\text{min.} \quad \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} \tag{6.21}$$

$$\text{s.t.} \quad \sum_{j\in\mathcal{V},\ j\neq i} x_{ij} = 1,\ \forall i \in \mathcal{V} \tag{6.22}$$

$$\sum_{i\in\mathcal{V},\ i\neq j} x_{ij} = 1,\ \forall j \in \mathcal{V} \tag{6.23}$$

$$y_{ij} \leq (n-1)x_{ij},\ \forall (i,j) \in \mathcal{A} \tag{6.24}$$

$$\sum_{j\in\mathcal{V}\backslash\{1\}} y_{1j} = n-1, \tag{6.25}$$

$$\sum_{i\in\mathcal{V}\backslash\{j\}} y_{ij} - \sum_{k\in\mathcal{V}\backslash\{j\}} y_{jk} = 1,\ \forall j \in \mathcal{V}\backslash\{1\} \tag{6.26}$$

$$x_{ij} \in \{0, 1\},\ \forall (i,j) \in \mathcal{A} \tag{6.27}$$

$$y_{ij} \geq 0,\ \forall (i,j) \in \mathcal{A} \tag{6.28}$$

In this formulation, the constraints (6.24) - (6.26) are the subtour elimination constraints. Specifically, constraints (6.24) make sure that the amount of flow on any arc (i, j) is at most $n-1$ when the arc is active. Constraints (6.25) state that there is a total of $n-1$ flowing out of the source node 1. Constraints (6.26) require that the incoming flow is 1 unit bigger than the outgoing flow at any node other than the source node 1. To see how this prevents subtours, we'll use Figure 6.10 as an example.

According to constraints (6.25), we have the following relations:

$$v - w = 1 \tag{6.29}$$

$$w - v = 1 \tag{6.30}$$

Summing them together leads to $0 = 2$, which is clearly false.

The code below gives the complete implementation of this formulation.

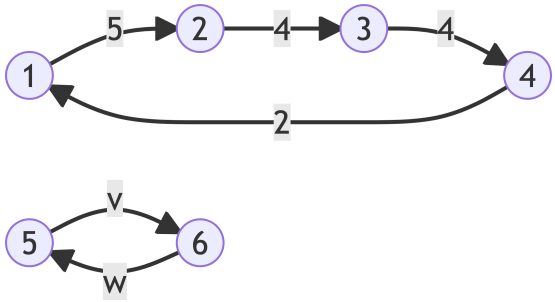


Figure 6.10: Subtour with 2 nodes

```python
from itertools import product
from ortools.linear_solver import pywraplp

class TspModelV3(TspModel):

    def __init__(self, name='Tspmodel_v3'):
        super().__init__(name)

        self._solver = pywraplp.Solver.CreateSolver('SCIP')
        self._var_x = None
        self._var_y = None
        self._opt_obj = None
        self._opt_x = None
        self._opt_route = None

    def build_model(self):
        self._create_variables()
        self._create_objective()
        self._create_degree_constraints()
        self._create_subtour_elimination_constraints()

    def optimize(self, enable_output: bool):
        if enable_output: self._solver.EnableOutput()
        status = self._solver.Solve()
        if status is pywraplp.Solver.OPTIMAL:
            self._retrieve_opt_solution()
            self._retrieve_opt_route()
```

```
    def _create_variables(self):
        self._var_x = {}
        for i in self._node_list:
            self._var_x[i] = {
                j: self._solver.BoolVar(f'x_{i, j}')
                for j in self._node_list
                if j != i
            }

        infinity = self._solver.Infinity()
        self._var_y = {}
        for i in self._node_list:
            self._var_y[i] = {
                j: self._solver.NumVar(0, infinity, f'y_{i, j}')
                for j in self._node_list
                if j != i
            }

    def _create_objective(self):
        node_list = self._node_list
        expr = [self._distance[i][j] * self._var_x[i][j]
                for i, j in product(node_list, node_list)
                if i != j]
        self._solver.Minimize(self._solver.Sum(expr))

    def _create_degree_constraints(self):
        for i in self._node_list:
            out_expr = [self._var_x[i][j]
                        for j in self._node_list
                        if j != i
                        ]
            in_expr = [self._var_x[j][i]
                       for j in self._node_list
                       if j != i
                       ]
            self._solver.Add(self._solver.Sum(out_expr) == 1)
            self._solver.Add(self._solver.Sum(in_expr) == 1)

    def _create_subtour_elimination_constraints(self):
        num_nodes = self.num_nodes
        for i, j in product(self._node_list, self._node_list):
```

```
            if i == j: continue
            self._solver.Add(
                self._var_y[i][j] <=
                (num_nodes - 1) * self._var_x[i][j])

        expr = [self._var_y[1][j] for j in self._node_list if j != 1]
        self._solver.Add(self._solver.Sum(expr) == num_nodes - 1)

        for j in self._node_list:
            if j == 1: continue
            expr1 = [self._var_y[i][j]
                    for i in self._node_list if i != j]
            expr2 = [self._var_y[j][k]
                     for k in self._node_list if k != j]
            self._solver.Add(
                self._solver.Sum(expr1) -
                self._solver.Sum(expr2) == 1)


    def _retrieve_opt_solution(self):
        self._opt_obj = float(self._solver.Objective().Value())
        self._opt_x = {}
        for i in self._node_list:
            self._opt_x[i] = {
                j: round(self._var_x[i][j].solution_value())
                for j in self._node_list
                if j != i
            }

        print(f'optimal value = {self._opt_obj:.2f}')

    def _retrieve_opt_route(self):
        self._opt_route = []
        route_start = list(self._opt_x.keys())[0]
        edge_start = route_start
        while True:
            for n in self._opt_x[edge_start]:
                if self._opt_x[edge_start][n] == 0: continue

                edge_end = n
                self._opt_route.append((edge_start, edge_end))
                break
```

```
            if edge_end == route_start: break
            edge_start = edge_end

    def show_opt_route(self):
        TspVisualizer.show(self._node_coords, self._opt_route)


    def show_model_info(self):
        print(f"Number of variables: {self._solver.NumVariables()}")
        print(f"Number of constraints: {self._solver.NumConstraints()}")
```

Figure 6.11 shows the optimal soution for instance *burma14* found by this formulation.

```
instance = './data/tsp/burma14.tsp'

model_v3 = TspModelV3()
model_v3.read_inputs(instance_file=instance)
model_v3.build_model()
model_v3.show_model_info()
model_v3.optimize(False)
model_v3.show_opt_route()
```

```
Number of variables: 364
Number of constraints: 224
optimal value = 3323.00
```

Figure 6.11: Optimal route for the *burma14* instance

Figure 6.12 shows the optimal route found by the formulation for the instance *ulysses22*.

```
instance = './data/tsp/ulysses22.tsp'

model_v3 = TspModelV3()
model_v3.read_inputs(instance_file=instance)
model_v3.build_model()
model_v3.show_model_info()
model_v3.optimize(False)
model_v3.show_opt_route()
```

```
Number of variables: 924
Number of constraints: 528
optimal value = 7013.00
```

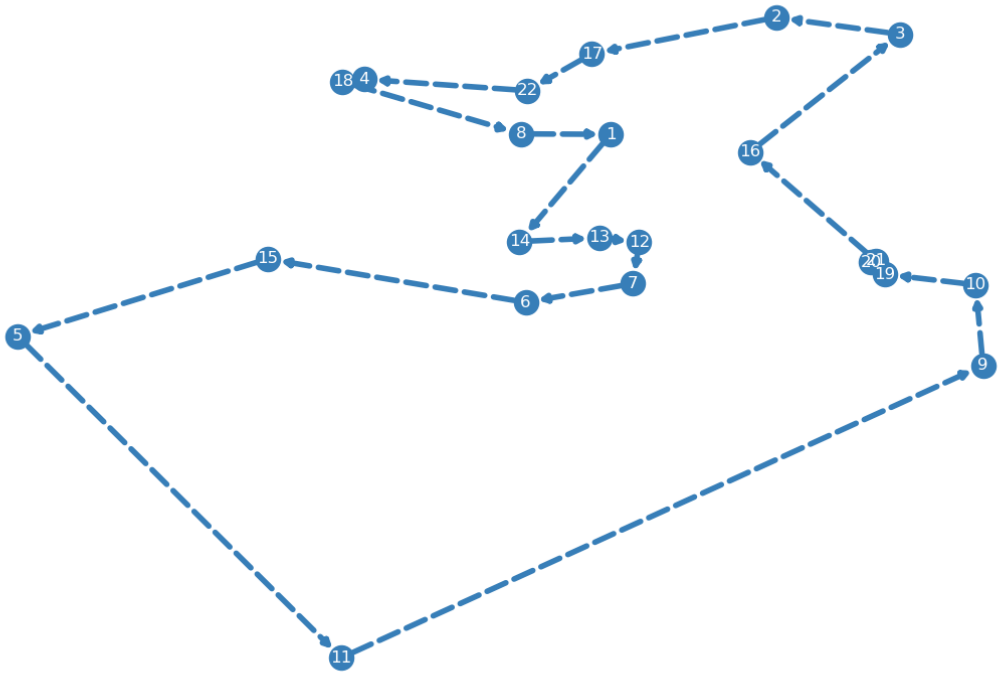


Figure 6.12: Optimal route for the *ulysses22* instance

Figure 6.13 gives the optimal solution of instance *berlin52*.

```
instance = './data/tsp/berlin52.tsp'

model_v3 = TspModelV3()
model_v3.read_inputs(instance_file=instance)
model_v3.build_model()
model_v3.show_model_info()
model_v3.optimize(False)
model_v3.show_opt_route()
```

```
Number of variables: 5304
Number of constraints: 2808
optimal value = 7542.00
```

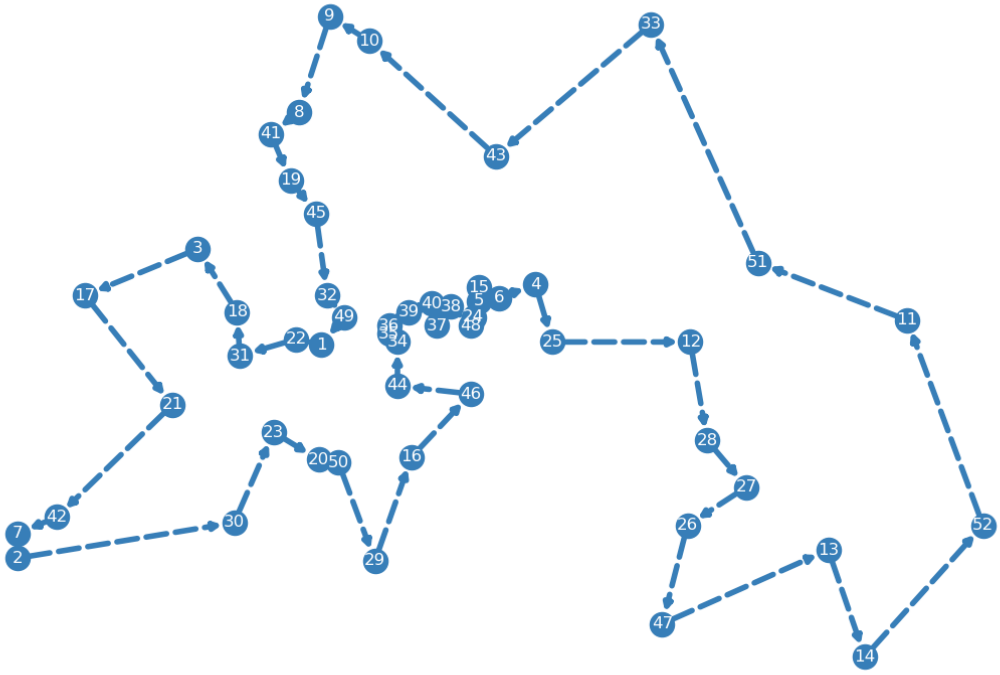


Figure 6.13: Optimal route for the *berlin52* instance

Figure 6.14 shows the optimal solution for instance *pr16*.

```
instance = './data/tsp/pr76.tsp'

model_v3 = TspModelV3()
model_v3.read_inputs(instance_file=instance)
model_v3.build_model()
model_v3.show_model_info()
model_v3.optimize(False)
model_v3.show_opt_route()
```

```
Number of variables: 11400
Number of constraints: 5928
optimal value = 108159.00
```

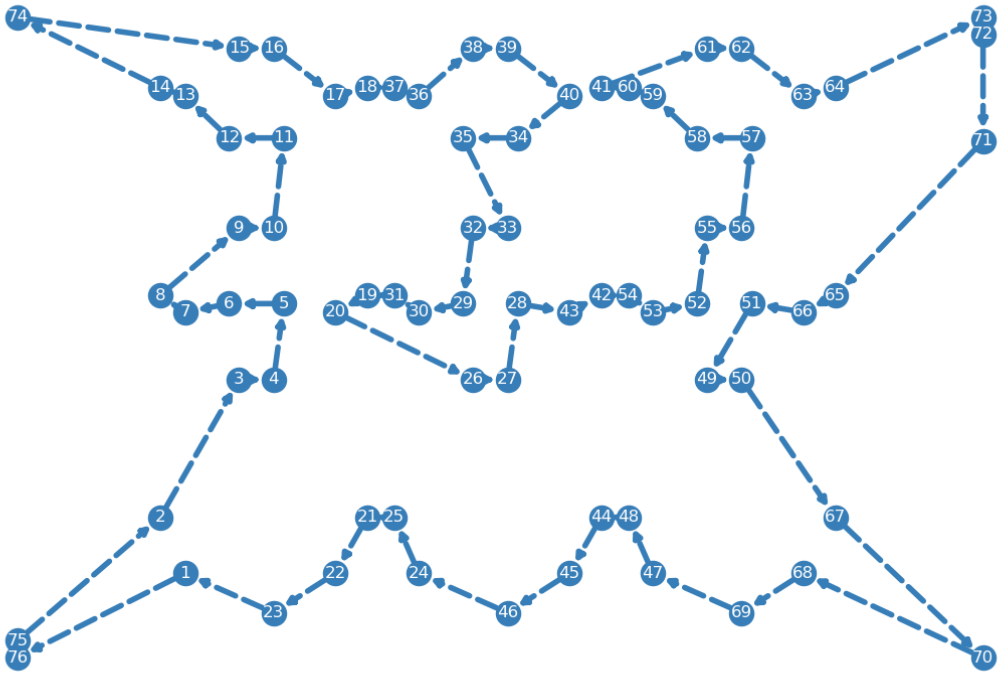


Figure 6.14: Optimal route for the *pr76* instance

6.6 Model 4 - Two Commodity Flow

The model uses three types of 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
- y_{ij}: a continuous variable representing the flow of commodity 1 on arc $(i, j) \in \mathcal{A}$
- z_{ij}: a continuous variable representing the flow of commodity 2 on arc $(i, j) \in \mathcal{A}$

The complete formulation is given below.

$$\text{min.} \quad \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} \tag{6.31}$$

$$\text{s.t.} \quad \sum_{j\in\mathcal{V},\ j\neq i} x_{ij} = 1,\ \forall i \in \mathcal{V} \tag{6.32}$$

$$\sum_{i\in\mathcal{V},\ i\neq j} x_{ij} = 1,\ \forall j \in \mathcal{V} \tag{6.33}$$

$$\sum_{j\in\mathcal{V}\backslash\{1\}} (y_{1j} - y_{j1}) = n - 1 \tag{6.34}$$

$$\sum_{j\in\mathcal{V},j\neq i} (y_{ij} - y_{ji}) = -1,\ \forall i \in \mathcal{V}\backslash\{1\} \tag{6.35}$$

$$\sum_{j\in\mathcal{V}\backslash\{1\}} (z_{1j} - z_{j1}) = -(n - 1) \tag{6.36}$$

$$\sum_{j\in\mathcal{V},j\neq i} (z_{ij} - z_{ji}) = 1,\ \forall i \in \mathcal{V}\backslash\{1\} \tag{6.37}$$

$$\sum_{j\in\mathcal{V},j\neq i} (y_{ij} + z_{ij}) = n - 1,\ \forall i \in \mathcal{V} \tag{6.38}$$

$$y_{ij} + z_{ij} = (n-1)x_{ij},\ \forall (i,j) \in \mathcal{A} \tag{6.39}$$

$$x_{ij} \in \{0,1\},\ \forall (i,j) \in \mathcal{A} \tag{6.40}$$

$$y_{ij} \geq 0,\ \forall (i,j) \in \mathcal{A} \tag{6.41}$$

$$z_{ij} \geq 0,\ \forall (i,j) \in \mathcal{A} \tag{6.42}$$

In this formulation, constraints (6.34) - (6.39) together serve as the subtour elimination constraints. To better understand the two commodity flow formulation, we'll use Figure 6.15 as an example. At the source node 1, there is $n-1$ units of commodity 1 leaving the node and there is no unit of commodity 2 leaving it. At each subsequent node, the amount of commodity 1 decreases by 1 unit while the amount of commodity 2 increases by 1 unit. On any arc in the tour, the total amount of commodities is always $n-1$.

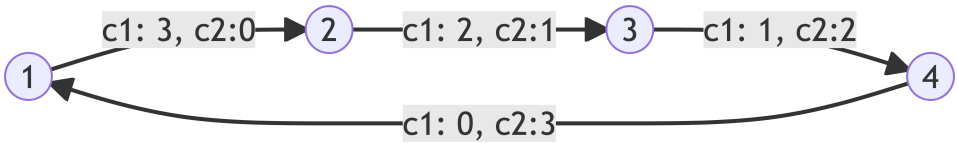


Figure 6.15: Two commodities flowing on the tour

The code below gives the complete implementation of the two commodity flow formulation.

```
from itertools import product
from ortools.linear_solver import pywraplp

class TspModelV4(TspModel):

    def __init__(self, name='Tspmodel_v4'):
        super().__init__(name)

        self._solver = pywraplp.Solver.CreateSolver('SCIP')
        self._var_x = None
        self._var_y = None
        self._var_z = None
        self._opt_obj = None
        self._opt_x = None
        self._opt_route = None

    def build_model(self):
        self._create_variables()
        self._create_objective()
        self._create_degree_constraints()
        self._create_subtour_elimination_constraints()

    def optimize(self, enable_output: bool):
        if enable_output: self._solver.EnableOutput()
        status = self._solver.Solve()
        if status is pywraplp.Solver.OPTIMAL:
            self._retrieve_opt_solution()
            self._retrieve_opt_route()

    def _create_variables(self):
        self._var_x = {}
        for i in self._node_list:
            self._var_x[i] = {
                j: self._solver.BoolVar(f'x_{i, j}')
                for j in self._node_list
                if j != i
            }

        infinity = self._solver.Infinity()
        self._var_y = {}
        for i in self._node_list:
```

```
            self._var_y[i] = {
                j: self._solver.NumVar(0, infinity, f'y_{i, j}')
                for j in self._node_list
                if j != i
            }

        self._var_z = {}
        for i in self._node_list:
            self._var_z[i] = {
                j: self._solver.NumVar(0, infinity, f'z_{i, j}')
                for j in self._node_list
                if j != i
            }

    def _create_objective(self):
        node_list = self._node_list
        expr = [self._distance[i][j] * self._var_x[i][j]
                for i, j in product(node_list, node_list)
                if i != j]
        self._solver.Minimize(self._solver.Sum(expr))

    def _create_degree_constraints(self):
        for i in self._node_list:
            out_expr = [self._var_x[i][j]
                        for j in self._node_list
                        if j != i
                        ]
            in_expr = [self._var_x[j][i]
                       for j in self._node_list
                       if j != i
                       ]
            self._solver.Add(self._solver.Sum(out_expr) == 1)
            self._solver.Add(self._solver.Sum(in_expr) == 1)

    def _create_subtour_elimination_constraints(self):
        num_nodes = self.num_nodes
        expr1 = [self._var_y[1][j]
                 for j in self._node_list
                 if j != 1]
        expr2 = [self._var_y[j][1]
                 for j in self._node_list
```

```
        if j != 1]
self._solver.Add(
    self._solver.Sum(expr1) -
    self._solver.Sum(expr2) ==
    num_nodes - 1)

for i in self._node_list:
    if i == 1: continue
    expr1 = [self._var_y[i][j]
            for j in self._node_list
            if j != i]
    expr2 = [self._var_y[j][i]
            for j in self._node_list
            if j != i]
    self._solver.Add(
        self._solver.Sum(expr1) -
        self._solver.Sum(expr2) ==
        -1)

expr1 = [self._var_z[1][j]
        for j in self._node_list
        if j != 1]
expr2 = [self._var_z[j][1]
        for j in self._node_list
        if j != 1]
self._solver.Add(
    self._solver.Sum(expr1) -
    self._solver.Sum(expr2) ==
    -num_nodes + 1)

for i in self._node_list:
    if i == 1: continue
    expr1 = [self._var_z[i][j]
            for j in self._node_list
            if j != i]
    expr2 = [self._var_z[j][i]
            for j in self._node_list
            if j != i]
    self._solver.Add(
        self._solver.Sum(expr1) -
        self._solver.Sum(expr2) ==
```

```
                1)

        for i in self._node_list:
            expr1 = [self._var_y[i][j]
                     for j in self._node_list
                     if j != i]
            expr2 = [self._var_z[i][j]
                     for j in self._node_list
                     if j != i]
            self._solver.Add(
                self._solver.Sum(expr1) +
                self._solver.Sum(expr2) ==
                num_nodes - 1)

        for i, j in product(self._node_list, self._node_list):
            if i == j: continue
            self._solver.Add(
                self._var_y[i][j] +
                self._var_z[i][j] ==
                (num_nodes - 1) *
                self._var_x[i][j])


    def _retrieve_opt_solution(self):
        self._opt_obj = float(self._solver.Objective().Value())
        self._opt_x = {}
        for i in self._node_list:
            self._opt_x[i] = {
                j: round(self._var_x[i][j].solution_value())
                for j in self._node_list
                if j != i
            }

        print(f'optimal value = {self._opt_obj:.2f}')

    def _retrieve_opt_route(self):
        self._opt_route = []
        route_start = list(self._opt_x.keys())[0]
        edge_start = route_start
        while True:
            for n in self._opt_x[edge_start]:
```

```
                if self._opt_x[edge_start][n] == 0: continue

                edge_end = n
                self._opt_route.append((edge_start, edge_end))
                break

            if edge_end == route_start: break
            edge_start = edge_end

    def show_opt_route(self):
        TspVisualizer.show(self._node_coords, self._opt_route)


    def show_model_info(self):
        print(f"Number of variables: {self._solver.NumVariables()}")
        print(f"Number of constraints: {self._solver.NumConstraints()}")
```

Figure 6.16 shows the optimals solution identified by the formulation for instance *burma14*.

```
instance = './data/tsp/burma14.tsp'

model_v4 = TspModelV4()
model_v4.read_inputs(instance_file=instance)
model_v4.build_model()
model_v4.show_model_info()
model_v4.optimize(False)
model_v4.show_opt_route()
```

```
Number of variables: 546
Number of constraints: 252
optimal value = 3323.00
```

Figure 6.16: Optimal route for the *burma14* instance

Figure 6.17 gives the optimal solution of the *ulysses22* instance.

```
instance = './data/tsp/ulysses22.tsp'

model_v4 = TspModelV4()
model_v4.read_inputs(instance_file=instance)
model_v4.build_model()
model_v4.show_model_info()
model_v4.optimize(False)
model_v4.show_opt_route()
```

```
Number of variables: 1386
Number of constraints: 572
optimal value = 7013.00
```

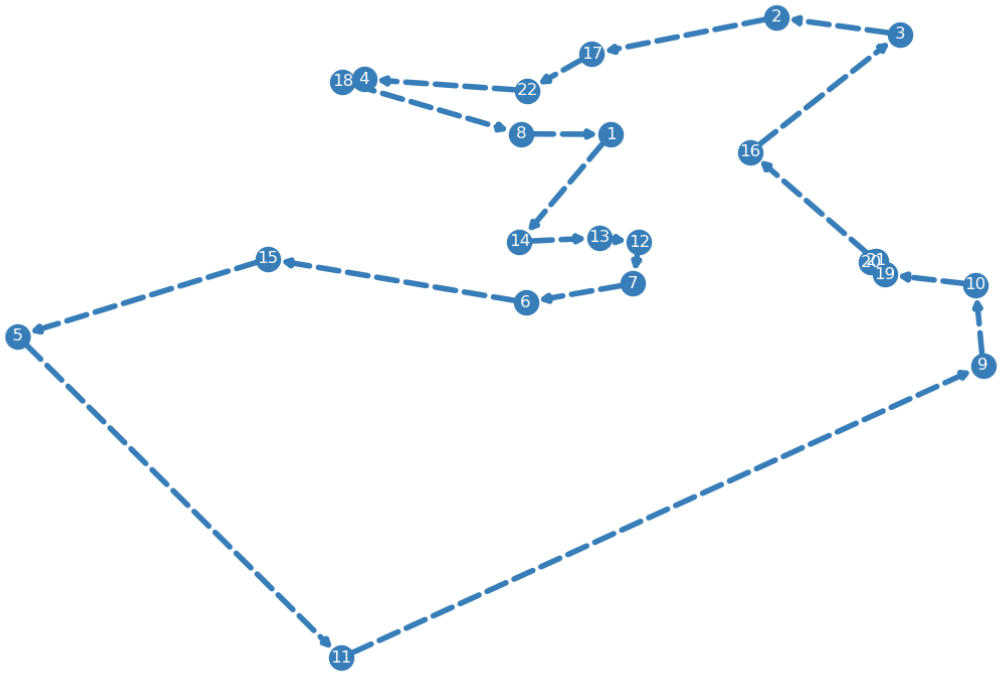


Figure 6.17: Optimal route for the *ulysses22* instance

Figure 6.18 shows the optimals solution identified by the formulation for instance *berlin52*.

```
instance = './data/tsp/berlin52.tsp'

model_v4 = TspModelV4()
model_v4.read_inputs(instance_file=instance)
model_v4.build_model()
model_v4.show_model_info()
model_v4.optimize(False)
model_v4.show_opt_route()
```

```
Number of variables: 7956
Number of constraints: 2912
optimal value = 7542.00
```

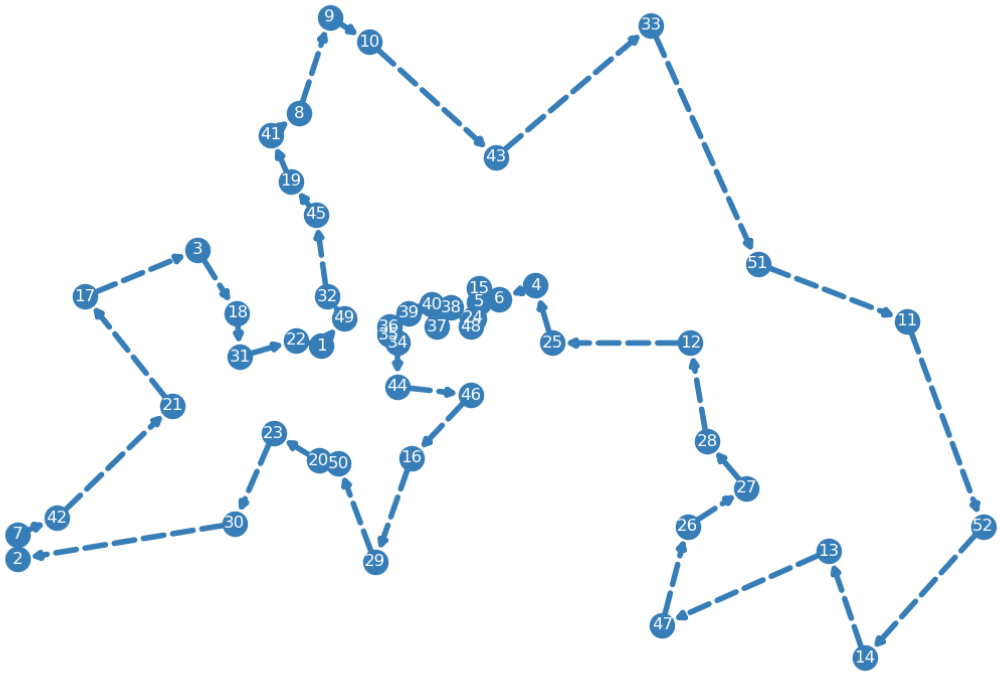


Figure 6.18: Optimal route for the *berlin52* instance

6.7 Model 5 - Multi-Commodity Flow

The model uses two types of 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
- y_{ij}^k: a continuous variable representing the flow of commodity k on arc $(i, j) \in \mathcal{A}$, $k \in \mathcal{V} \backslash \{1\}$

The complete formulation of the problem is presented below. To understand this formulation, let's imagine that node 1 is the source node in the graph, and each of the remaining nodes demands one unit of different commodities. For instance, node 2 requires one unit of commodity 2, node 3 requires one unit of commodity 3, and so on. The problem can then be expressed as finding the most cost-effective graph that can handle the flow of all $n - 1$ commodities. Constraints (6.44) and (6.45) are the same as in other formulations. Constraints (6.46) state that the arc connecting nodes (i,j) must be active to allow any commodity flow, essentially acting as a capacity constraint that limits the flow on the arc. Constraints (6.47) indicate

that exactly one unit of each commodity k flows out of the source node 1. Constraints (6.48) require that no commodity flows into the source node 1. Constraints (6.49) ensure that one unit of commodity k flows into node k, and constraints (6.50) require that there is no flow of commodity k out of node k. Finally, constraints (6.51) are flow conservation constraints that guarantee that the incoming flow is equal to the outgoing flow at all nodes except the source node for every commodity k.

$$\text{min.} \quad \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} \tag{6.43}$$

$$\text{s.t.} \quad \sum_{j\in\mathcal{V},\ j\neq i} x_{ij} = 1,\ \forall i \in \mathcal{V} \tag{6.44}$$

$$\sum_{i\in\mathcal{V},\ i\neq j} x_{ij} = 1,\ \forall j \in \mathcal{V} \tag{6.45}$$

$$y^k_{ij} \leq x_{ij},\ \forall (i,j) \in \mathcal{A}, k \in \mathcal{V}\backslash\{1\} \tag{6.46}$$

$$\sum_{j\in\mathcal{V}\backslash\{1\}} y^k_{1j} = 1,\ \forall k \in \mathcal{V}\backslash\{1\} \tag{6.47}$$

$$\sum_{j\in\mathcal{V}\backslash\{1\}} y^k_{j1} = 0,\ \forall k \in \mathcal{V}\backslash\{1\} \tag{6.48}$$

$$\sum_{j\in\mathcal{V}\backslash\{k\}} y^k_{jk} = 1,\ \forall k \in \mathcal{V}\backslash\{1\} \tag{6.49}$$

$$\sum_{j\in\mathcal{V}\backslash\{k\}} y^k_{kj} = 0,\ \forall k \in \mathcal{V}\backslash\{1\} \tag{6.50}$$

$$\sum_{i\in\mathcal{V}\backslash\{j\}} y^k_{ij} - \sum_{i\in\mathcal{V}\backslash\{j\}} y^k_{ji} = 0,\ \forall j,k \in \mathcal{V}\backslash\{1\}, j \neq k \tag{6.51}$$

$$x_{ij} = \{0,1\},\ \forall (i,j) \in \mathcal{A} \tag{6.52}$$

$$y_{ij} \geq 0,\ \forall (i,j) \in \mathcal{A}, k \in \mathcal{V}\backslash\{1\} \tag{6.53}$$

```python
from itertools import product
from ortools.linear_solver import pywraplp

class TspModelV5(TspModel):

    def __init__(self, name='Tspmodel_v5'):
        super().__init__(name)

        self._solver = pywraplp.Solver.CreateSolver('SCIP')
        self._var_x = None
        self._var_y = None
```

```
        self._opt_obj = None
        self._opt_x = None
        self._opt_route = None

    def build_model(self):
        self._create_variables()
        self._create_objective()
        self._create_degree_constraints()
        self._create_subtour_elimination_constraints()

    def optimize(self, enable_output: bool):
        if enable_output: self._solver.EnableOutput()
        status = self._solver.Solve()
        if status is pywraplp.Solver.OPTIMAL:
            self._retrieve_opt_solution()
            self._retrieve_opt_route()

    def _create_variables(self):
        self._var_x = {}
        for i in self._node_list:
            self._var_x[i] = {
                j: self._solver.BoolVar(f'x_{i, j}')
                for j in self._node_list
                if j != i
            }

        infinity = self._solver.Infinity()
        self._var_y = {}
        for k in self._node_list[1:]:
            vars = {}
            for i in self._node_list:
                vars[i] = {
                    j: self._solver.NumVar(0, infinity, f'y_{i, j}')
                    for j in self._node_list
                    if j != i
                }
            self._var_y[k] = vars

    def _create_objective(self):
        node_list = self._node_list
        expr = [self._distance[i][j] * self._var_x[i][j]
```

```
            for i, j in product(node_list, node_list)
            if i != j]
    self._solver.Minimize(self._solver.Sum(expr))

def _create_degree_constraints(self):
    for i in self._node_list:
        out_expr = [self._var_x[i][j]
                for j in self._node_list
                if j != i
                ]
        in_expr = [self._var_x[j][i]
                    for j in self._node_list
                    if j != i
                ]
        self._solver.Add(self._solver.Sum(out_expr) == 1)
        self._solver.Add(self._solver.Sum(in_expr) == 1)

def _create_subtour_elimination_constraints(self):
    for k in self._node_list[1:]:
        for i, j in product(self._node_list, self._node_list):
            if i == j: continue
            self._solver.Add(self._var_y[k][i][j] <= self._var_x[i][j])

    for k in self._node_list[1:]:
        expr = [self._var_y[k][1][j]
                for j in self._node_list
                if j != 1]
        self._solver.Add(self._solver.Sum(expr) == 1)

        expr = [self._var_y[k][j][1]
                for j in self._node_list
                if j != 1]
        self._solver.Add(self._solver.Sum(expr) == 0)

        expr = [self._var_y[k][j][k]
                for j in self._node_list
                if j != k]
        self._solver.Add(self._solver.Sum(expr) == 1)

        expr = [self._var_y[k][k][j]
                for j in self._node_list
```

```
                if j != k]
        self._solver.Add(self._solver.Sum(expr) == 0)

    for j, k in product(self._node_list[1:],
                        self._node_list[1:]):
        if j == k: continue
        expr1 = [self._var_y[k][i][j]
                for i in self._node_list
                if i != j]
        expr2 = [self._var_y[k][j][i]
                for i in self._node_list
                if i != j]
        self._solver.Add(self._solver.Sum(expr1) ==
                        self._solver.Sum(expr2))


def _retrieve_opt_solution(self):
    self._opt_obj = float(self._solver.Objective().Value())
    self._opt_x = {}
    for i in self._node_list:
        self._opt_x[i] = {
            j: round(self._var_x[i][j].solution_value())
            for j in self._node_list
            if j != i
        }

    print(f'optimal value = {self._opt_obj:.2f}')

def _retrieve_opt_route(self):
    self._opt_route = []
    route_start = list(self._opt_x.keys())[0]
    edge_start = route_start
    while True:
        for n in self._opt_x[edge_start]:
            if self._opt_x[edge_start][n] == 0: continue

            edge_end = n
            self._opt_route.append((edge_start, edge_end))
            break

        if edge_end == route_start: break
```

```
            edge_start = edge_end

    def show_opt_route(self):
        TspVisualizer.show(self._node_coords, self._opt_route)


    def show_model_info(self):
        print(f"Number of variables: {self._solver.NumVariables()}")
        print(f"Number of constraints: {self._solver.NumConstraints()}")
```

```
instance = './data/tsp/burma14.tsp'

model_v5 = TspModelV5()
model_v5.read_inputs(instance_file=instance)
model_v5.build_model()
model_v5.show_model_info()
model_v5.optimize(False)
model_v5.show_opt_route()
```

```
Number of variables: 2548
Number of constraints: 2602
optimal value = 3323.00
```

Figure 6.19: Optimal route for the *burma14* instance

```
instance = './data/tsp/ulysses22.tsp'

model_v5 = TspModelV5()
model_v5.read_inputs(instance_file=instance)
model_v5.build_model()
model_v5.show_model_info()
model_v5.optimize(False)
model_v5.show_opt_route()
```

```
Number of variables: 10164
Number of constraints: 10250
optimal value = 7013.00
```

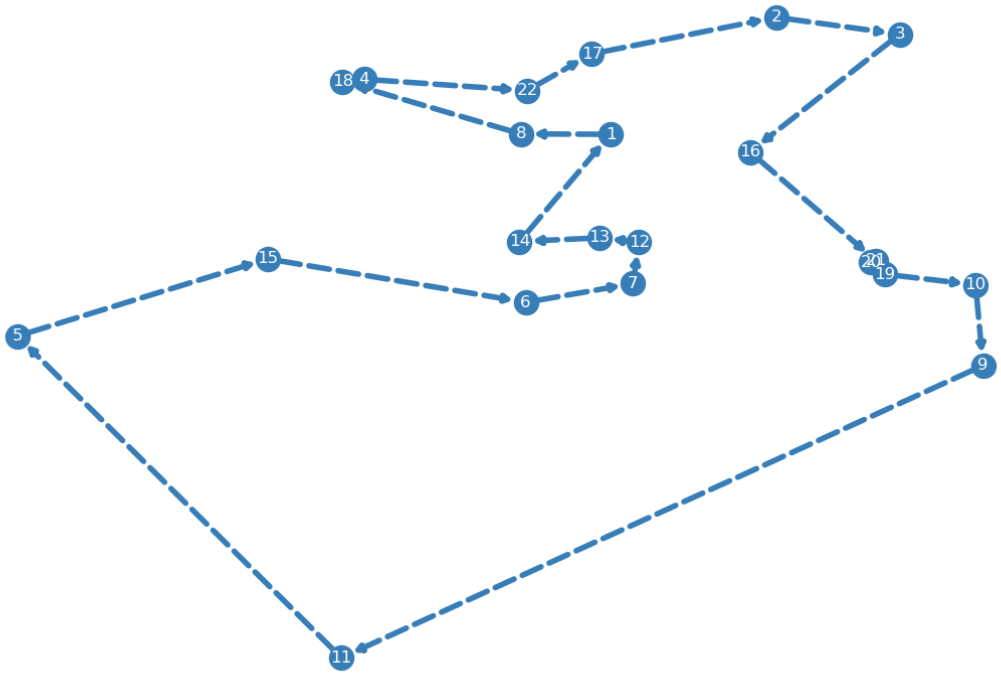


Figure 6.20: Optimal route for the *ulysses22* instance

```
instance = './data/tsp/berlin52.tsp'

model_v5 = TspModelV5()
model_v5.read_inputs(instance_file=instance)
model_v5.build_model()
model_v5.show_model_info()
model_v5.optimize(False)
model_v5.show_opt_route()
```

```
Number of variables: 137904
Number of constraints: 138110
optimal value = 7542.00
```

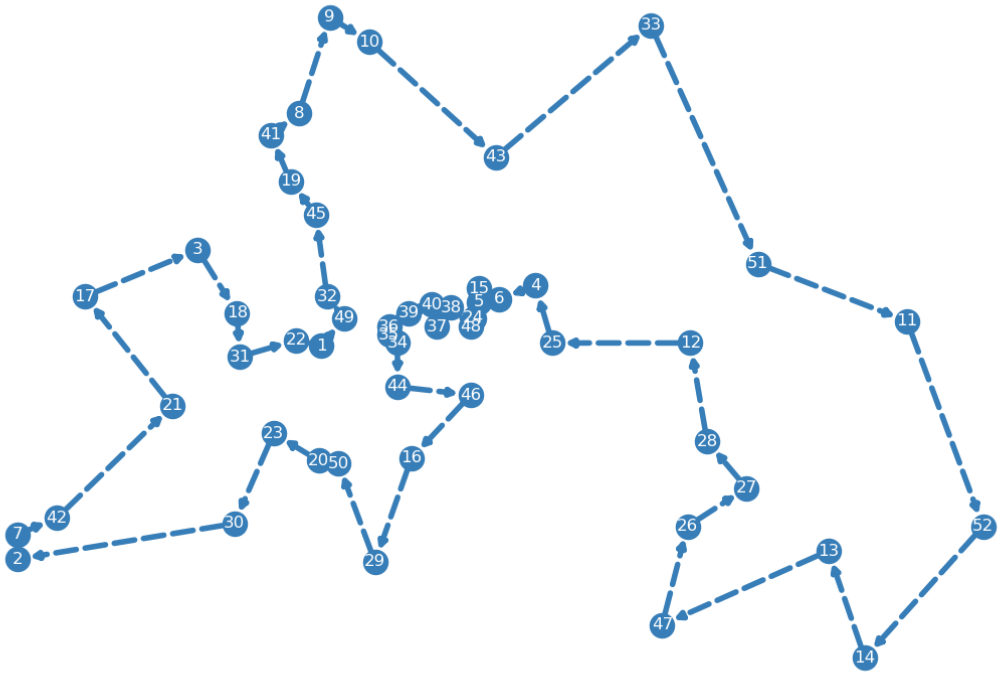


Figure 6.21: Optimal route for the *berlin52* instance

6.8 Performance Comparison

We give in Table 6.1 the computational times required to find the optimal solutions for different instances. It is by no means a thorough or comprehensive performance comparison, as it is only based on one run and on a few instances. It can be seen from the table that the single commodity flow formulation seems to perform the best among all the five formulations.

Table 6.1: Computational time comparison of the five formulations

Instance	Model 1	Model 2	Model 3	Model 4	Model 5
burma14	2.9s	0.4s	0.2s	0.6s	0.3s
ulysses22	-	7m25s	7.2s	19.8s	2.8s
berlin52	-	-	8.6s	13.8s	18m23s
pr76	-	-	10m51s	-	-

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.

7.1 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

# 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.

```
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)
```

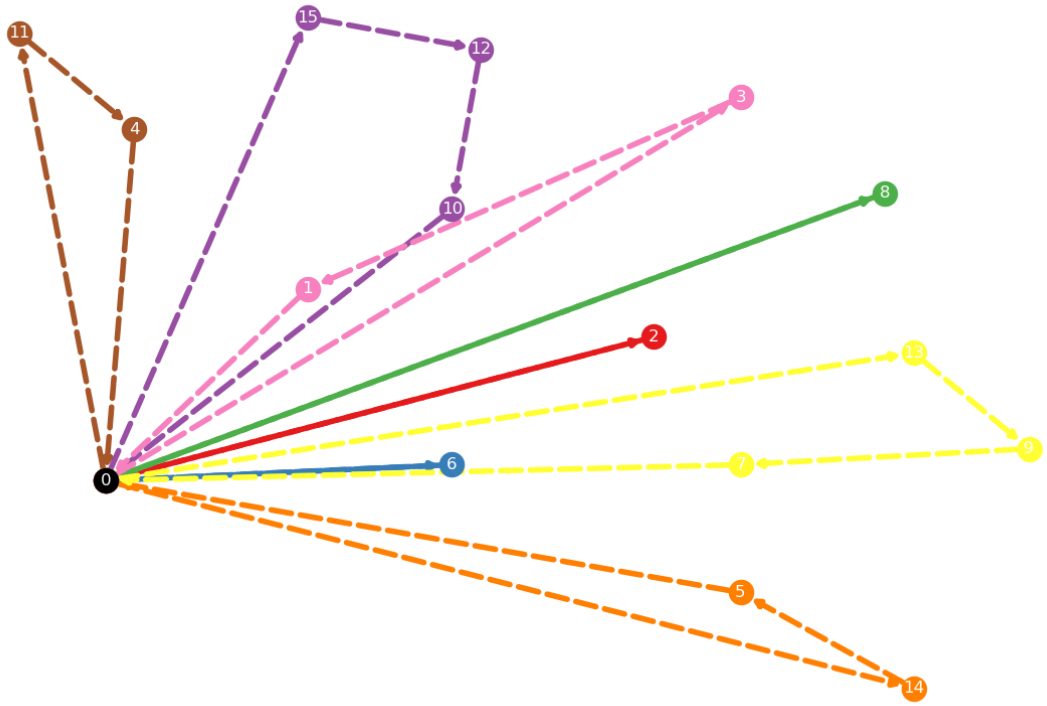


Figure 7.1: Optimal routes for instance P-n16-k8.vrp

7.2 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=0, y_coord=0, demand=0):
        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},\
            demand: {self._demand}"
```

```
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 a CVRP instance.

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

class CvrpDataCenter:
    """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
        if 'node_coord' in instance:
            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)
        else:
            for demand in instance['demand']:
                node = Node(id=idx,
                            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-mini.vrp")
```

```
# instantiate solver
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.

```
# 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.

```
# 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).

```
# 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.

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

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-mini.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}")
```

```
# 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()

    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)
```

To facilitate the model comparison in following steps, we'll wrap the above program into a dedicated class `Cvrp1`.

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

```
import math

class Cvrp1:
    """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:.2f}')

    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}')

    def show_opt_routes(self):
        nodes = self._data_center.nodes
        locations = {
            node.id: (node.x_coord, node.y_coord)
            for node in nodes
        }

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

        show_vehicle_routes(locations, edges)
```

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

```
cvrp1 = Cvrp1(CvrpDataCenter())
cvrp1.read_instance("./data/cvrp/P-n16-k8-mini.vrp")
cvrp1.build_model()
cvrp1.show_model_summary()
cvrp1.optimize()
cvrp1.show_opt_routes()
```

```
No. subtour elimination constraints: 2047
No. of variables: 132
No. of constraints: 2070
Optimal value: 347.00
route: [(0, 1), (1, 3), (3, 0)], length: 66
```

```
route: [(0, 2), (2, 0)], length: 42
route: [(0, 4), (4, 10), (10, 0)], length: 55
route: [(0, 5), (5, 9), (9, 7), (7, 0)], length: 68
route: [(0, 6), (6, 0)], length: 24
route: [(0, 8), (8, 11), (11, 0)], length: 92
```

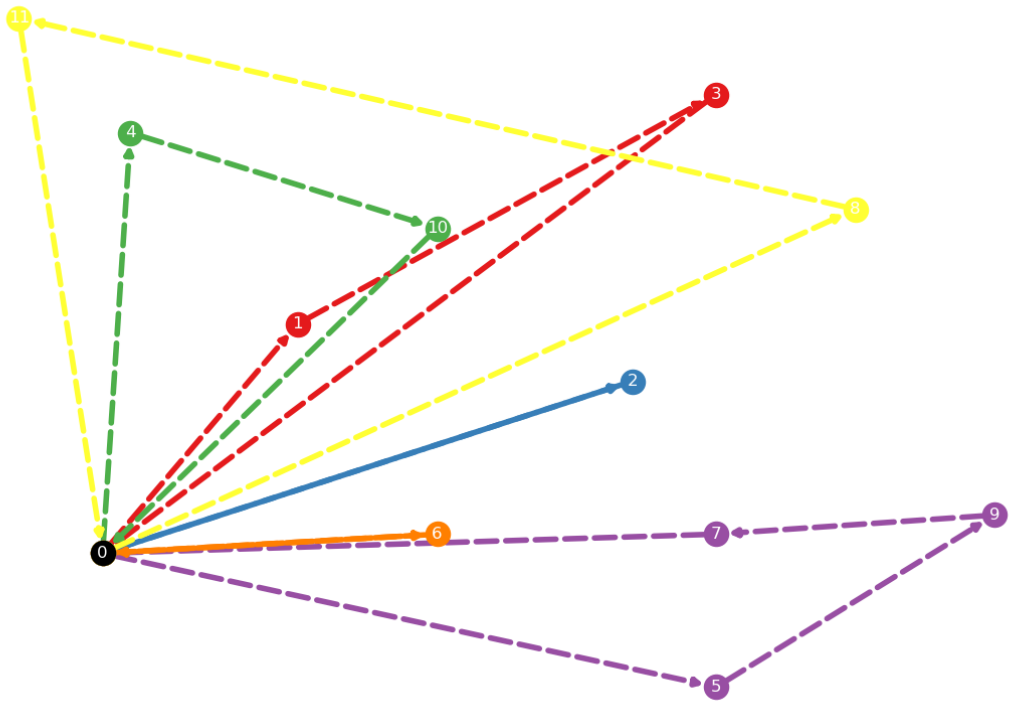


Figure 7.2: Optimal routes found by the two-index formulation

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.

7.3 Two-index Formulation - 2

This formulation is based on the MTZ-model introduced by Miller, Tucker, and Zemlin (1960) for the TSP. To eliminate subtours, a new variable u_i is defined for every node $i \in \mathcal{N}$:

- u_i: the total demands distributed by any vehicle when it arrives at node i

$$\text{min.} \quad \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} \tag{7.7}$$

$$\text{s.t.} \quad \sum_{j\in\delta^+(i)} x_{ij} = 1,\ \forall i \in \mathcal{N} \tag{7.8}$$

$$\sum_{i\in\delta^-(j)} x_{ij} = 1,\ \forall j \in \mathcal{N} \tag{7.9}$$

$$\sum_{j\in\delta^+(0)} x_{oj} = |\mathcal{K}| \tag{7.10}$$

$$u_i \le u_j - q_j + Q(1 - x_{ij}),\ \forall (i,j) \in \mathcal{A}, i, j \in \mathcal{N} \tag{7.11}$$

$$q_i \le u_i \le Q,\ \forall i \in \mathcal{N} \tag{7.12}$$

$$x_{ij} \in \{0,1\},\ \forall (i,j) \in \mathcal{A} \tag{7.13}$$

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

class Cvrp2:
    """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._var_u = 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}")

        self._var_u = np.empty(num_nodes, dtype=object)
        nodes = self._data_center.nodes
        vehicle_capacity = float(self._data_center.vehicle_capacity)
        for node in nodes:
            if node.id == 0: continue
            self._var_u[node.id] = self._solver.NumVar(
                lb=float(node.demand),
                ub=vehicle_capacity,
                name='v')

    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]
```

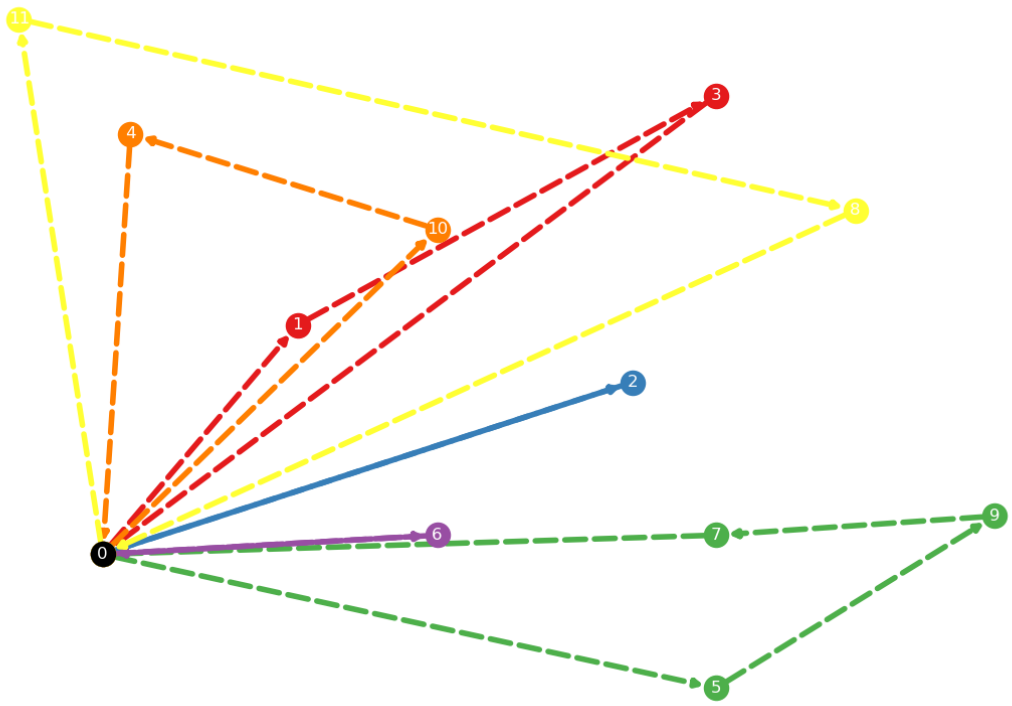
```
            in_arcs = [self._var_x[j][i] for j in range(num_nodes) if j != i]
            self._solver.Add(self._solver.Sum(out_arcs) == 1)
            self._solver.Add(self._solver.Sum(in_arcs) == 1)

    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
        constraints = []
        nodes = self._data_center.nodes
        vehicle_capacity = self._data_center.vehicle_capacity
        for ni, nj in product(nodes, nodes):
            if ni.id == 0 or nj.id == 0: continue
            if ni.id == nj.id: continue
            constr = self._solver.Add(
                self._var_u[ni.id] <=
                self._var_u[nj.id] -
                nj.demand +
                vehicle_capacity * (
                    1 - self._var_x[ni.id][nj.id]
                )
            )
            constraints.append(constr)

        print(f"No. subtour elimination constraints: {len(constraints)}")

    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:.2f}')

        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}')

    def show_opt_routes(self):
```

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

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

        show_vehicle_routes(locations, edges)
```

```
cvrp2 = Cvrp2(CvrpDataCenter())
cvrp2.read_instance("./data/cvrp/P-n16-k8-mini.vrp")
cvrp2.build_model()
cvrp2.show_model_summary()
cvrp2.optimize()
cvrp2.show_opt_routes()
```

```
No. subtour elimination constraints: 110
No. of variables: 143
No. of constraints: 133
Optimal value: 347.00
route: [(0, 1), (1, 3), (3, 0)], length: 66
route: [(0, 2), (2, 0)], length: 42
route: [(0, 5), (5, 9), (9, 7), (7, 0)], length: 68
route: [(0, 6), (6, 0)], length: 24
route: [(0, 10), (10, 4), (4, 0)], length: 55
route: [(0, 11), (11, 8), (8, 0)], length: 92
```



7.4 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.

We define the following variables in this formualation:

- x_{ijk}: a binary variable that equals 1 when the vehicle k visits arc $(i, j) \in \mathcal{A}$, 0 otherwise
- y_{ik}: a binary variable that equals 1 if node i is visited by vehicle k, 0 otherwise
- u_{ik}: a continuous variable that represents the demands delivered by vehicle k when arriving at node i

$$\text{min.} \quad \sum_{k\in\mathcal{K}}\sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ijk} \tag{7.14}$$

$$\text{s.t.} \quad x_{iik} = 0,\ \forall i \in \mathcal{V}, k \in \mathcal{K} \tag{7.15}$$

$$\sum_{j\in\mathcal{V}} x_{ijk} = \sum_{j\in\mathcal{V}} x_{jik},\ \forall i \in \mathcal{V}, k \in \mathcal{K} \tag{7.16}$$

$$\sum_{k\in\mathcal{K}} y_{ik} = 1,\ \forall i \in \mathcal{N} \tag{7.17}$$

$$y_{ik} = \sum_{j\in\mathcal{V}} x_{ijk},\ \forall i \in \mathcal{V}, k \in \mathcal{K} \tag{7.18}$$

$$y_{0k} = 1,\ \forall k \in \mathcal{K} \tag{7.19}$$

$$u_i \leq u_j - q_j + Q(1 - x_{ijk}),\ \forall i \neq j, i, j \in \mathcal{N}, k \in \mathcal{K} \tag{7.20}$$

$$q_i \leq u_i \leq Q,\ \forall i \in \mathcal{N} \tag{7.21}$$

$$x_{ijk} \in \{0, 1\},\ \forall (i, j) \in \mathcal{A}, k \in \mathcal{K} \tag{7.22}$$

$$y_{ik} \in \{0, 1\},\ \forall i \in \mathcal{V}, k \in \mathcal{K} \tag{7.23}$$

$$u_{ik} \geq 0,\ \forall i \in \mathcal{V}, k \in \mathcal{K} \tag{7.24}$$

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

class Cvrp3:
    """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._var_y = None
        self._var_u = 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_customer_must_be_visited()
        self._create_constr_subtour()

    def optimize(self):
        status = self._solver.Solve()
        print("solve complete!")
        if not status:
            self._retrieve_opt_solution()
            self._retrieve_opt_routes()
        else:
            print(f"status={status}")

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

        self._var_y = np.empty((num_vehicles, num_nodes), dtype=object)
        for k, i in product(range(num_vehicles),
                               range(num_nodes)):
            self._var_y[k][i] = self._solver.BoolVar(name=f'y_{k, i}')

        vehicle_capacity = self._data_center.vehicle_capacity
        self._var_u = np.empty(num_nodes, dtype=object)
        for node in self._data_center.nodes:
            self._var_u[node.id] = self._solver.NumVar(
                float(node.demand),
                vehicle_capacity,
                name=f'u_{node.id}')

    def _create_objective(self):
        num_vehicles = self._data_center.num_vehicles
```

```
        num_nodes = self._data_center.num_nodes
        obj_expr = [
            self._data_center.distance(i, j, integer=True) *
                self._var_x[k][i][j]
                for k, i, j in product(range(num_vehicles), range(num_nodes), range(num_no
                if i != j
        ]
        self._solver.Minimize(self._solver.Sum(obj_expr))

    def _create_constr_flow(self):
        # create incoming and outgoing arc constraints
        num_vehicles = self._data_center.num_vehicles
        num_nodes = self._data_center.num_nodes
        for k, i in product(range(num_vehicles),
                            range(num_nodes)):
            self._solver.Add(self._var_x[k][i][i] == 0)

        for k, i in product(range(num_vehicles),
                            range(num_nodes)):
            expr1 = [self._var_x[k][i][j] for j in range(num_nodes)]
            expr2 = [self._var_x[k][j][i] for j in range(num_nodes)]
            self._solver.Add(self._solver.Sum(expr1) ==
                            self._solver.Sum(expr2))

    def _create_constr_customer_must_be_visited(self):
        num_vehicles = self._data_center.num_vehicles
        num_nodes = self._data_center.num_nodes
        for i in range(1, num_nodes):
            expr = [self._var_y[k][i] for k in range(num_vehicles)]
            self._solver.Add(self._solver.Sum(expr) == 1)

        for k, i in product(range(num_vehicles),
                            range(num_nodes)):
            expr = [self._var_x[k][i][j] for j in range(num_nodes)]
            self._solver.Add(self._solver.Sum(expr) ==
                            self._var_y[k][i])

        for k in range(num_vehicles):
            self._solver.Add(self._var_y[k][0] == 1)

    def _create_constr_subtour(self):
        # create subtour elimination constraint
```

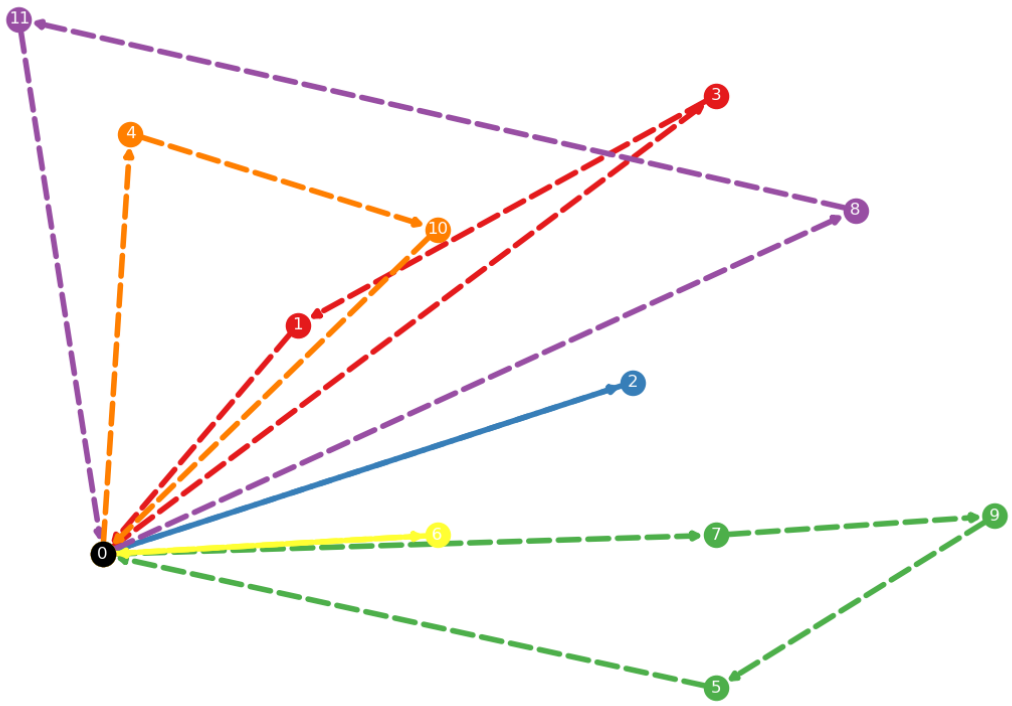
```
        constraints = []
        nodes = self._data_center.nodes
        vehicle_capacity = self._data_center.vehicle_capacity
        num_vehicles = self._data_center.num_vehicles
        for k, ni, nj in product(range(num_vehicles), nodes, nodes):
            if ni.id == 0 or nj.id == 0: continue
            if ni.id == nj.id: continue
            constr = self._solver.Add(self._var_u[ni.id] <=
                    self._var_u[nj.id] -
                    float(nj.demand) +
                    vehicle_capacity * (
                        1 - self._var_x[k][ni.id][nj.id]
                    )
                )
            constraints.append(constr)

        print(f"No. subtour elimination constraints: {len(constraints)}")

    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:.2f}')

        num_nodes = self._data_center.num_nodes
        num_vehicles = self._data_center.num_vehicles
        self._opt_x = np.zeros((num_vehicles, num_nodes, num_nodes))
        for k, i, j in product(range(num_vehicles), range(num_nodes), range(num_nodes)):
            self._opt_x[k][i][j] = int(self._var_x[k][i][j].solution_value())

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

```
                # new route found
                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[k][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}')

                break

    def show_opt_routes(self):
        nodes = self._data_center.nodes
        locations = {
            node.id: (node.x_coord, node.y_coord)
            for node in nodes
        }

        edges = []
        vehicle_idx = 0
        for route in self._routes:
            for arc in route:
```

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

        show_vehicle_routes(locations, edges)
```

```
cvrp3 = Cvrp3(CvrpDataCenter())
cvrp3.read_instance("./data/cvrp/P-n16-k8-mini.vrp")
cvrp3.build_model()
cvrp3.show_model_summary()
cvrp3.optimize()
cvrp3.show_opt_routes()
```

```
No. subtour elimination constraints: 660
No. of variables: 948
No. of constraints: 893
solve complete!
Optimal value: 347.00
route: [(0, 3), (3, 1), (1, 0)], length: 66
route: [(0, 2), (2, 0)], length: 42
route: [(0, 7), (7, 9), (9, 5), (5, 0)], length: 68
route: [(0, 8), (8, 11), (11, 0)], length: 92
route: [(0, 4), (4, 10), (10, 0)], length: 55
route: [(0, 6), (6, 0)], length: 24
```



7.5 Performance Comparison

Table 7.1 shows the runtime comparison on the same instance.

Table 7.1: Computational time comparison of the three formulations

Instance	Model 1	Model 2	Model 3
P-n12-k8-mini.vrp	0.4s	0.3s	1m5s

8 The Knapsack Problem

The knapsack problem is a classic optimization problem in the field of operations research. It involves selecting a subset of items from a given set of items to maximize the total value/profit while satisfying certain constraints. This problem is NP-hard, meaning that it is computationally difficult to find an optimal solution for large instances of the problem. Various algorithms have been developed to solve this problem, including dynamic programming, branch and bound, and heuristic methods. The knapsack problem has applications in a variety of fields, including computer science, finance, and logistics, among others. In this chapter, we'll examine two variants of the knapsack problem, namely, the single-dimensional knapsack problem and the multi-dimensional knapsack problem.

8.1 Single-dimensional Knapsack Problem

In the single-dimensional knapsack problem (SDKP), we are given a set of n items, each with a weight w_j and a value v_j, and a knapsack with a maximum weight capacity W, the goal is to select a subset of items such that the sum of their weights is less than or equal to W, and the sum of their values is maximized. The SDKP can be formally formulated as follows (Dudziński and Walukiewicz (1987)).

$$\text{max.} \quad \sum_{j=1}^{n} v_i x_j \tag{8.1}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} w_j x_j \leq W \tag{8.2}$$

$$x_j \in \{0,1\},\ j = 1, \cdots, n \tag{8.3}$$

In the model, the n represents the total amount of items that are being considered. The binary decision variable x_i has a value of 1 when item i is chosen, and 0 when it's not. The value or profit of selecting item i is indicated by v_i, and the weight of item i is represented by w_i. Finally, W specifies the maximum weight capacity of the knapsack.

To represent the list of items in the problem, we first define a class `Item` that has three attributes:

- `_index`: this is the index of an item and starts from 0
- `_profit`: this is the profit or value of selecting the item
- `_properties`: this dictionary saves an item's properties, including weight

```
class Item:
    """An item represents an object that can be placed within a knapsack
    """

    def __init__(self, index, profit):
        """constructor

        Args:
            index (int): index of the item, starting from 0
            profit (float): profit of choosing the item
        """
        self._index = index
        self._profit = profit
        self._class = 0
        self._properties = {}

    @property
    def index(self): return self._index

    @property
    def profit(self): return self._profit

    def get_class(self): return self._class

    def set_class(self, cls): self._class = cls

    def get_property(self, name):
        return self._properties[name]

    def set_property(self, name, value):
        self._properties[name] = value

    def __str__(self):
        p_str = ""
        for attr in self._properties:
            p_str += f'{attr}: {self._properties[attr]}'
        return f"index: {self._index}, profit: {self._profit}, " + \
                p_str
```

Next, we define a class `KnapsackDataCenter` to hold all the information we need to solve a knapsack problem. The class has two attributes:

- `_items`: this is the full list of items being considered for putting into the knapsack
- `_capacities`: this saves the maximal capacity of the knapsack corresponding to different properties, including weight.

The class also defines a method `read_data_set_f()` that reads and parses some benchmarking instancese available online.

```
class KnapsackDataCenter:

    def __init__(self):
        self._items = []
        self._capacities = {}

    @property
    def items(self): return self._items

    @property
    def capacities(self): return self._capacities
```

```
class SDKPDataCenter(KnapsackDataCenter):

    def __init__(self):
        super().__init__()

    def read_sdkp_dataset_f(self, data_file: str):
        """this function reads and parses data presented in
        http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/

        Args:
            data_file (str): path to the data file
            num_items (int): number of items in the file
            capacity (int): knapsack capacity
        """
        with open(data_file) as f:
            first_line = f.readline()
            num_items, capacity = first_line.split()
            self._capacities['weight'] = float(capacity)

            item_idx = 0
```

```
            rest_lines = f.readlines()
            for line in rest_lines:
                profit, weight = line.split()
                item = Item(item_idx, float(profit))
                item.set_property('weight', float(weight))
                self._items.append(item)
                item_idx += 1
                if item_idx == int(num_items): break
```

The code snippet below reads the instance `f1_l-d_kp_10_269` and shows its key information.

```
data_file = "./data/knapsack/SDKP/instances_01_KP/low-dimensional/f1_l-d_kp_10_269"

data_center = SDKPDataCenter()
data_center.read_sdkp_dataset_f(data_file)

capacities = data_center.capacities
print(f"Knapsack info: {capacities}")

items = data_center.items
print("Item info: ")
for item in items:
    print(item)
```

```
Knapsack info: {'weight': 269.0}
Item info:
index: 0, profit: 55.0, weight: 95.0
index: 1, profit: 10.0, weight: 4.0
index: 2, profit: 47.0, weight: 60.0
index: 3, profit: 5.0, weight: 32.0
index: 4, profit: 4.0, weight: 23.0
index: 5, profit: 50.0, weight: 72.0
index: 6, profit: 8.0, weight: 80.0
index: 7, profit: 61.0, weight: 62.0
index: 8, profit: 85.0, weight: 65.0
index: 9, profit: 87.0, weight: 46.0
```

Now we are ready to solve the SDKP using OR-Tools. The code below shows the completion definition of the class `SDKnapsackSolver` which has a number of attributes:

- `_data_center`: this should be an instantiated `KnapsackDataCenter` object

- `_solver`: this is the solver object used for modeling and problem solving
- `_var_x`: this is the object that holds all the decision variables used in the model
- `_opt_obj`: this is the optimal solution value found by the solver
- `_opt_x`: this is the optimal solution identified when the solving process completes

The class has two major methods:

- `build_model()`: this is responsible for instantiating the solver object, creating variables, generating constraints and defining the objective function
- `optimize()`: this is the place where the OR-Tools searches for the optimal solution

```
from ortools.linear_solver import pywraplp

class SDKnapsackSolver:

    def __init__(self, data_center):
        self._data_center = data_center

        self._solver = None
        self._var_x = None

        self._opt_obj = None
        self._opt_x = None

    def build_model(self):
        self._solver = pywraplp.Solver.CreateSolver('SCIP')

        self._create_variables()
        self._create_objective()
        self._create_constraints()

    def optimize(self):
        status = self._solver.Solve()
        if status == pywraplp.Solver.OPTIMAL:
            self._opt_obj = self._solver.Objective().Value()
            items = self._data_center.items
            self._opt_x = [
                    self._var_x[item.index].solution_value()
                    for item in items
                ]

    def _create_variables(self):
```

```
        items = self._data_center.items
        self._var_x = [self._solver.BoolVar(name=f'x_{i}')
                       for i, item in enumerate(items)]

    def _create_objective(self):
        items = self._data_center.items
        obj_expr = [
                self._var_x[item.index] * item.profit
                for item in items
                ]
        self._solver.Maximize(self._solver.Sum(obj_expr))

    def _create_constraints(self):
        items = self._data_center.items
        capacities = self._data_center.capacities
        expr = [
            self._var_x[item.index] *
                item.get_property('weight')
            for item in items
            ]
        self._solver.Add(
                self._solver.Sum(expr) <= capacities['weight']
            )

    @property
    def opt_obj(self): return self._opt_obj

    def get_num_chosen_items(self):
        return sum(self._opt_x)
```

We employ the model to tackle 10 benchmark instances that were downloaded from an online source (http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/). Table 8.1 displays the computational results.

More specifically, the first column of the table lists the names of the instances, while the second and third columns indicate the number of items and the capacity of the knapsack for each instance, respectively. The last two columns report the optimal solution's objective value and the number of selected items, respectively.

Table 8.1: Computational results of Knapsack problems

Instance	No. of Items	Capacity	Optimal Value	No. of Chosen Items
f1_l-d_kp_10_269	10	269	295	6
f2_l-d_kp_20_878	20	878	1024	17
f3_l-d_kp_4_20	4	20	35	3
f4_l-d_kp_4_11	4	11	23	2
f5_l-d_kp_15_375	15	375	481.069	9
f6_l-d_kp_10_60	10	60	52	7
f7_l-d_kp_7_50	7	50	107	2
f8_l-d_kp_23_10000	23	10000	9767	11
f9_l-d_kp_5_80	5	80	130	4
f10_l-d_kp_20_879	20	879	1025	17

8.2 Multi-Dimensional Knapsack Problem

The multi-dimensional knapsack problem is a variant of the classical knapsack problem where there are multiple candidate items and each item has multiple attributes or dimensions (Petersen (1967)). The goal is to select a subset of items that maximizes the total value or profit subject to the constraint that the sum of the attribute values of the selected items does not exceed certain limits.

Formally, let there be n items, and for each item j ($1 \leq j \leq n$), let:

- v_j be the profit of selecting it
- w_{ij} be the ith attribute value of item j
- m is the number of dimensions or attributes
- W_i is the capacity of the knapsack for dimension i

The multi-dimensional knapsack problem can then be formulated as the following optimization problem:

$$\text{max.} \quad \sum_{j=1}^{n} v_j x_j \tag{8.4}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} w_{ij} x_j \leq W_i, \ \forall i = 1, \cdots, m \tag{8.5}$$

$$x_j \in \{0, 1\}, \ j = 1, \cdots, n \tag{8.6}$$

To test the model, we take data from online source (http://people.brunel.ac.uk/~mastjjb/jeb/orlib/mknapinfo.ht and define a reading function `read_mdkp_dataset_1()` in class `MDKPDataCenter`. Note that the class inherits the `KnapsackDataCenter` class defined in the previous section, and all the parsed contents of an instance will be saved in the `_items` and `_capacities` attributes.

```
class MDKPDataCenter(KnapsackDataCenter):

    def read_mdkp_dataset_1(self, data_file: str):
        """read data from testing instance

        Args:
            data_file (str): data file
        """
        with open(data_file) as f:
            first_line = f.readline()
            num_items, \
            num_constraints, \
            opt_val = first_line.split()

            second_line = f.readline()
            profits = second_line.split()
            item_idx = 0
            for p in profits:
                item = Item(item_idx, float(p))
                self._items.append(item)
                item_idx += 1

            for i in range(int(num_constraints)):
                line = f.readline()
                weights = line.split()
                prop = f'prop_{i}'
                for idx, val in enumerate(weights):
                    self._items[idx].set_property(
                        prop,
                        float(val)
                    )

            last_line = f.readline()
            for idx, val in enumerate(last_line.split()):
                prop = f'prop_{idx}'
                self._capacities[prop] = float(val)
```

Now we define in the code below the class `MDKnapsackSolver` that performs a couple of

things:

- create variables in lines 31 - 34
- define objective in lines 36 - 42
- create constraints in line 44 - 58

```
from ortools.linear_solver import pywraplp

class MDKnapsackSolver:

    def __init__(self, data_center):
        self._data_center = data_center

        self._solver = None
        self._var_x = None

        self._opt_obj = None
        self._opt_x = None

    def build_model(self):
        self._solver = pywraplp.Solver.CreateSolver('SCIP')

        self._create_variables()
        self._create_objective()
        self._create_constraints()

    def optimize(self):
        status = self._solver.Solve()
        if status == pywraplp.Solver.OPTIMAL:
            self._opt_obj = self._solver.Objective().Value()
            items = self._data_center.items
            self._opt_x = [
                    self._var_x[item.index].solution_value()
                    for item in items
                ]

    def _create_variables(self):
        items = self._data_center.items
        self._var_x = [self._solver.BoolVar(name=f'x_{i}')
                    for i, item in enumerate(items)]

    def _create_objective(self):
```

```
        items = self._data_center.items
        obj_expr = [
                self._var_x[item.index] * item.profit
                for item in items
                ]
        self._solver.Maximize(self._solver.Sum(obj_expr))

    def _create_constraints(self):
        items = self._data_center.items
        capacities = self._data_center.capacities
        num_properties = len(capacities)
        for p_idx in range(num_properties):
            prop = f'prop_{p_idx}'
            expr = [
                self._var_x[item.index] *
                    item.get_property(prop)
                for item in items
                ]
            self._solver.Add(
                    self._solver.Sum(expr) <=
                    capacities[prop], name=f'cons_{p_idx}'
                )

    @property
    def opt_obj(self): return self._opt_obj

    def get_num_chosen_items(self):
        return sum(self._opt_x)
```

Table 8.2 shows the computational results of some testing instances.

Table 8.2: Computational results of multi-dimensional knapsack problems

Instance	No. of Items	Optimal Value	No. of Chosen Items
inst1.txt	6	3800	3
inst2.txt	10	8706.1	5
inst3.txt	15	4015	9
inst4.txt	20	6120	9
inst5.txt	28	12400	18
inst6.txt	39	10618	27
inst7.txt	50	16537	35

8.3 Multi-Choice Knapsack Problem

In the multi-choice knapsack problem, there are n candidate items to be placed in a knapsack and each item belongs to a specific class $k = 1, \cdots m$. It is required that only one item can be selected from each class. The problem can be formulated as below (Sinha and Zoltners (1979)).

$$\max. \quad \sum_{j=1}^{n} v_j x_j \tag{8.7}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} w_j x_j \leq W \tag{8.8}$$

$$\sum_{j=1}^{n} c_{jk} x_j = 1, \ \forall k = 1, \cdots m \tag{8.9}$$

$$x_j = \{0, 1\}, \ \forall j = 1, \cdots n \tag{8.10}$$

The class `MCKPDataCenter` defines a reader function to parse the instances available online (https://or-dii.unibs.it/index.php?page=tiks).

```
class MCKPDataCenter(KnapsackDataCenter):

    def __init__(self):
        super().__init__()

        self._num_classes = 0
        self._num_choices = 0

    def read_mcmdkp_dataset_1(self, data_file: str):
        with open(data_file) as f:
            f.readline()
            num_classes,\
            num_choices,\
            num_constraints = f.readline().split()
            self._num_classes = int(num_classes)
            self._num_choices = int(num_choices)

            capacities = f.readline().split()
            for p_idx in range(int(num_constraints)):
                prop = f'prop_{p_idx}'
                self._capacities[prop] = float(capacities[p_idx])
```

```
            item_idx = 0
            for i in range(int(num_classes)):
                f.readline()
                for c in range(int(num_choices)):
                    line = f.readline().split()
                    item = Item(item_idx, profit=float(line[0]))
                    for p_idx in range(int(num_constraints)):
                        prop = f'prop_{p_idx}'
                        item.set_property(prop, float(line[p_idx + 1]))
                        item.set_class(i)
                    item_idx += 1
                    self._items.append(item)

    @property
    def num_classes(self): return self._num_classes

    @property
    def num_choices(self): return self._num_choices
```

The complete code to solve the problem is given below.

```
from ortools.linear_solver import pywraplp

class MCKnapsackSolver:

    def __init__(self, data_center):
        self._data_center = data_center

        self._solver = None
        self._var_x = None

        self._opt_obj = None
        self._opt_x = None

    def build_model(self):
        self._solver = pywraplp.Solver.CreateSolver('SCIP')

        self._create_variables()
        self._create_objective()
        self._create_constraints()
```

```
    def optimize(self):
        status = self._solver.Solve()
        if status == pywraplp.Solver.OPTIMAL:
            self._opt_obj = self._solver.Objective().Value()
            items = self._data_center.items
            self._opt_x = [
                    self._var_x[item.index].solution_value()
                    for item in items
                ]

    def _create_variables(self):
        items = self._data_center.items
        self._var_x = [self._solver.BoolVar(name=f'x_{i}')
                    for i, item in enumerate(items)]

    def _create_objective(self):
        items = self._data_center.items
        obj_expr = [
                self._var_x[item.index] * item.profit
                for item in items
                ]
        self._solver.Maximize(self._solver.Sum(obj_expr))

    def _create_constraints(self):
        items = self._data_center.items
        capacities = self._data_center.capacities
        p_idx = 0
        prop = f'prop_{p_idx}'
        expr = [
            self._var_x[item.index] *
                item.get_property(prop)
            for item in items
            ]
        self._solver.Add(
                self._solver.Sum(expr) <=
                capacities[prop], name=f'cons_{p_idx}'
            )

        num_classes = self._data_center.num_classes
        for k in range(num_classes):
            expr = [self._var_x[item.index]
```

```
                for item in items
                if item.get_class() == k]
            self._solver.Add(
                self._solver.Sum(expr) == 1
            )

    @property
    def opt_obj(self): return self._opt_obj

    def get_num_chosen_items(self):
        return sum(self._opt_x)
```

Table 8.3 shows some computational experiments.

Table 8.3: Computational results of multi-choice knapsack problems

Instance	No. of Items	Optimal Value	No. of Chosen Items
INST01.txt	500	13411	50
INST02.txt	500	13953	50
INST03.txt	600	15727	60
INST04.txt	700	18928	70
INST05.txt	750	20314	75
INST06.txt	750	20277	75
INST07.txt	800	21372	80
INST08.txt	800	21556	80
INST09.txt	800	21581	80
INST10.txt	900	24232	90
INST11.txt	900	24267	90
INST12.txt	1000	26206	100
INST13.txt	3000	24382	100
INST14.txt	4500	36971	150
INST15.txt	5400	44001	180
INST16.txt	6000	48833	200
INST17.txt	7500	61056	250
INST18.txt	5600	68021	280
INST19.txt	6000	73054	300
INST20.txt	7000	84958	350

8.4 Multi-Choice Multi-Dimensional Knapsack Problem

In the multi-choice multi-dimensional knapsack problem (MCMDKP), there are n candidate items and each item belongs to a specific class $k = 1, \cdots, m$. Each item also has d attributes. The requirement is to select exactly one item from each class such that the total profit is maximized. Note that the knapsack capacity cannot be violated for any item attribute. The MCMDKP can be formulated as follows (Chen and Hao (2014)).

$$\text{max.} \quad \sum_{j=1}^{n} v_j x_j \tag{8.11}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} w_{ij} x_j \leq W_i, \ \forall i = 1, \cdots d \tag{8.12}$$

$$\sum_{j=1}^{n} c_{jk} x_j = 1, \ \forall k = 1, \cdots m \tag{8.13}$$

$$x_j = \{0, 1\}, \ \forall j = 1, \cdots n \tag{8.14}$$

The code snippet below gives the complete program to solve this problem.

```
from ortools.linear_solver import pywraplp

class MCMDKnapsackSolver:

    def __init__(self, data_center):
        self._data_center = data_center

        self._solver = None
        self._var_x = None

        self._opt_obj = None
        self._opt_x = None

    def build_model(self):
        self._solver = pywraplp.Solver.CreateSolver('SCIP')

        self._create_variables()
        self._create_objective()
        self._create_constraints()

    def optimize(self):
```

```
        status = self._solver.Solve()
        if status == pywraplp.Solver.OPTIMAL:
            self._opt_obj = self._solver.Objective().Value()
            items = self._data_center.items
            self._opt_x = [
                    self._var_x[item.index].solution_value()
                    for item in items
                ]

    def _create_variables(self):
        items = self._data_center.items
        self._var_x = [self._solver.BoolVar(name=f'x_{i}')
                    for i, item in enumerate(items)]

    def _create_objective(self):
        items = self._data_center.items
        obj_expr = [
                self._var_x[item.index] * item.profit
                for item in items
                ]
        self._solver.Maximize(self._solver.Sum(obj_expr))

    def _create_constraints(self):
        items = self._data_center.items
        capacities = self._data_center.capacities
        num_properties = len(capacities)
        for p_idx in range(num_properties):
            prop = f'prop_{p_idx}'
            expr = [
                self._var_x[item.index] *
                    item.get_property(prop)
                for item in items
                ]
            self._solver.Add(
                    self._solver.Sum(expr) <=
                    capacities[prop], name=f'cons_{p_idx}'
                )

        num_classes = self._data_center.num_classes
        for k in range(num_classes):
            expr = [self._var_x[item.index]
```

```
                for item in items
                if item.get_class() == k]
            self._solver.Add(
                self._solver.Sum(expr) == 1
            )

    @property
    def opt_obj(self): return self._opt_obj

    def get_num_chosen_items(self):
        return sum(self._opt_x)
```

Table 8.4 shows some empirical computational results.

Table 8.4: Computational results of multi-choice multi-dimensional knapsack problems

Instance	No. of Items	Optimal Value	No. of Chosen Items
INST01.txt	250	7059	25
INST02.txt	250	6998	25
INST03.txt	300	8418	30
INST04.txt	300	8518	30
INST05.txt	300	8418	30
INST06.txt	300	8418	30
INST07.txt	300	8418	30
INST08.txt	300	8418	30
INST09.txt	300	8418	30
INST10.txt	300	8418	30
INST11.txt	300	8418	30
INST12.txt	300	8418	30
INST13.txt	900	8833	30
INST14.txt	900	8841	30
INST15.txt	900	8833	30
INST16.txt	900	8788	30
INST17.txt	900	8820	30
INST18.txt	600	8664	30
INST19.txt	600	8667	30
INST20.txt	600	8714	30

9 The N-queens Problem

The n-queens problem is a classic puzzle that involves placing n queens on an $n \times n$ chessboard in such a way that no two queens threaten each other. In other words, no two queens can be placed on the same row, column, or diagonal. The problem is called the n-queens problem because it can be generalized to any size of n.

For example, the 8-queens problem involves placing 8 queens on an 8x8 chessboard, and the solution requires that no two queens share the same row, column, or diagonal.

The n-queens problem is a well-known problem in computer science and has been studied extensively because it has applications in various fields, such as optimization, artificial intelligence, and computer graphics. There are various algorithms and techniques that can be used to solve the n-queens problem, including brute-force search, backtracking, and genetic algorithms. In this chapter, we'll model this problem as an integer programming problem and solve it using OR-Tools.

To model this problem on an $n \times n$ chessboard, we define the following decision variable:

- x_{ij}: a binary variable that equals 1 if a queen is place on position (i, j), where $i, j = 0, \cdots, n-1$; and 0 otherwise

The complete model given below is based on Letavec and Ruggiero (2002).

$$\text{max.} \quad 0 \tag{9.1}$$

$$\text{s.t.} \quad \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} x_{ij} = n \tag{9.2}$$

$$\sum_{j=0}^{n-1} x_{ij} \leq 1, \ \forall i = 0, \cdots, n-1 \tag{9.3}$$

$$\sum_{i=0}^{n-1} x_{ij} \leq 1, \ \forall j = 0, \cdots, n-1 \tag{9.4}$$

$$\sum_{i=0}^{n-1}\sum_{j=0, i+j=k}^{n-1} x_{ij} \leq 1, \ \forall k = 1, \cdots, 2(n-1)-1 \tag{9.5}$$

$$\sum_{i=0}^{n-1}\sum_{j=0, i-j=k}^{n-1} x_{ij} \leq 1, \ \forall k = 2-n, \cdots, n-2 \tag{9.6}$$

$$x_{ij} \in \{0, 1\}, \ i, j = 0, \cdots, n-1 \tag{9.7}$$

In the formulation, the objective function (9.1) serves no practical use as our goal is to find any feasible solution to the puzzle. Constraints (9.2) require that there are exactly n queens be placed on a board of size $n \times n$. Constraints (9.3) make sure that there is at most one queen be placed in any row of the board. Similarly, constraints (9.4) ensure that there is at most one queen be placed in any column of the board. To understand constraints (9.5) and (9.6), let's look at a 8-queens solution in Figure 9.1.

Note that no two queens can be placed on the same diagonal anywhere on the board. Upon observing the chessboard, we can see that there are two types of diagonals:

- Diagonals that start from the upper left corner and move to the bottom right corner
  - Positions (1, 0), (0, 1) form a diagonal - sum of position coordinates = 1
  - Positions (2, 0), (1, 1), (0, 2) form a diagonal - sum of position coordinates = 2
  - ...
  - Positions (7, 6), (6, 7) form a diagonal - sum of position coordinates = 2 * (8 - 1) - 1
- Diagonals that start from the upper right corner and move to the bottom left corner
  - Positions (0, 6), (1, 7) form a diagonal - difference of position coordinates = -6
  - Positions (0, 5), (1, 6), (2, 7) form a diagonal - difference of position coordinates = -5
  - ...
  - Positions (6, 0), (7, 1) form a diagonal - difference of position coordinates = 8 - 2

The first type of diagonals can be expressed as constraints (9.5) and the second type of diagonals can be expressed as constraints (9.6). Both constraints guarantee that there cannot be any two queens showing up in the same diagonal.

The complete code to solve the problem is given below.

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

class NQueensSolver:

    def __init__(self, size: int):
        self._size = size

        self._solver = None
        self._var_x = None
        self._opt_x = None

    def build_model(self):
        # instantiate solver
        self._solver = pywraplp.Solver.CreateSolver('SCIP')

        # create decision variables
        self._var_x = np.empty((self._size, self._size),
                               dtype=object)
        for i, j in product(range(self._size),
                            range(self._size)):
            self._var_x[i][j] = \
                self._solver.BoolVar(name=f'x_{i,j}')

        # declare objective function
        self._solver.Maximize(0)

        # constraint: there must be n queens on the board
        expr = [self._var_x[i][j]
                for i in range(self._size)
                for j in range(self._size)]
        self._solver.Add(self._solver.Sum(expr) == self._size)

        # constraint: no two queens can be in the same row
        for row in range(self._size):
```

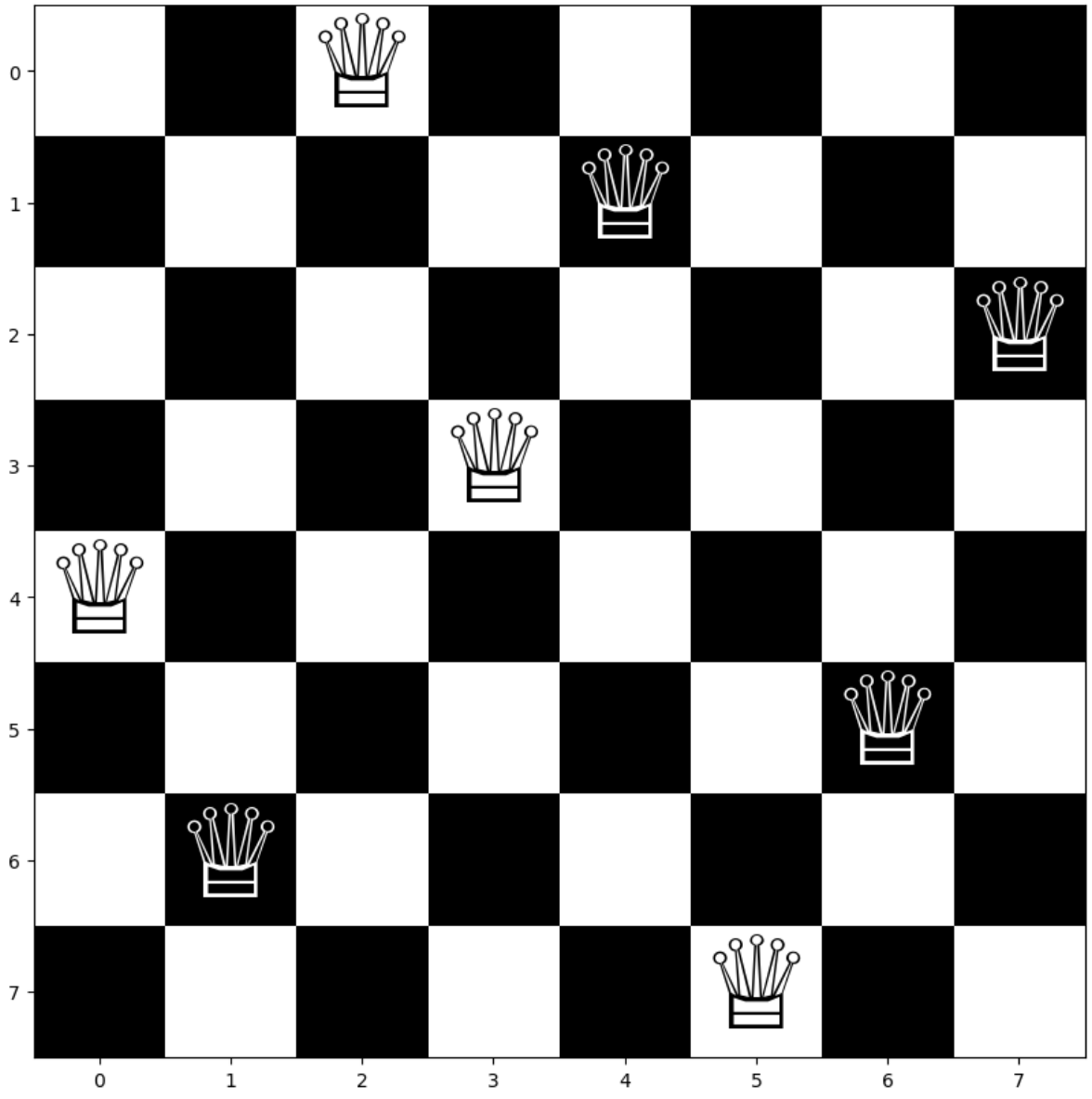


Figure 9.1: A 8-Queens solution

```
            expr = [self._var_x[row][j]
                    for j in range(self._size)]
            self._solver.Add(self._solver.Sum(expr) <= 1)

        # constraint: no two queens can be in the same column
        for col in range(self._size):
            expr = [self._var_x[i][col]
                    for i in range(self._size)]
            self._solver.Add(self._solver.Sum(expr) <= 1)

        # constraint: no two queens can be in the same diagonal
        for k in range(1, 2 * (self._size - 1) - 1):
            expr = [self._var_x[i][j]
                    for i in range(self._size)
                    for j in range(self._size)
                    if i + j == k]
            self._solver.Add(self._solver.Sum(expr) <= 1)

        for k in range(2 - self._size, self._size - 2):
            expr = [self._var_x[i][j]
                    for i in range(self._size)
                    for j in range(self._size)
                    if i - j == k]
            self._solver.Add(self._solver.Sum(expr) <= 1)

    def optimize(self):
        status = self._solver.Solve()
        if status == pywraplp.Solver.OPTIMAL:
            self._opt_x = np.zeros((self._size,
                                    self._size))
            for i, j in product(range(self._size),
                                range(self._size)):
                self._opt_x[i][j] = \
                    self._var_x[i][j].solution_value()
        else:
            print("solve failure!")
            print(f"status={status}")

    @property
    def opt_x(self): return self._opt_x
```

```
    def get_queen_coordinates(self):
        coordinates = [(i, j)
            for i in range(self._size)
            for j in range(self._size)
            if self._opt_x[i][j] == 1]
        return coordinates
```

Figure 9.2 shows a 4-Queens puzzle solution.

Figure 9.3 shows a 10-Queens puzzle solution.

Figure 9.4 shows a 20-Queens puzzle solution.

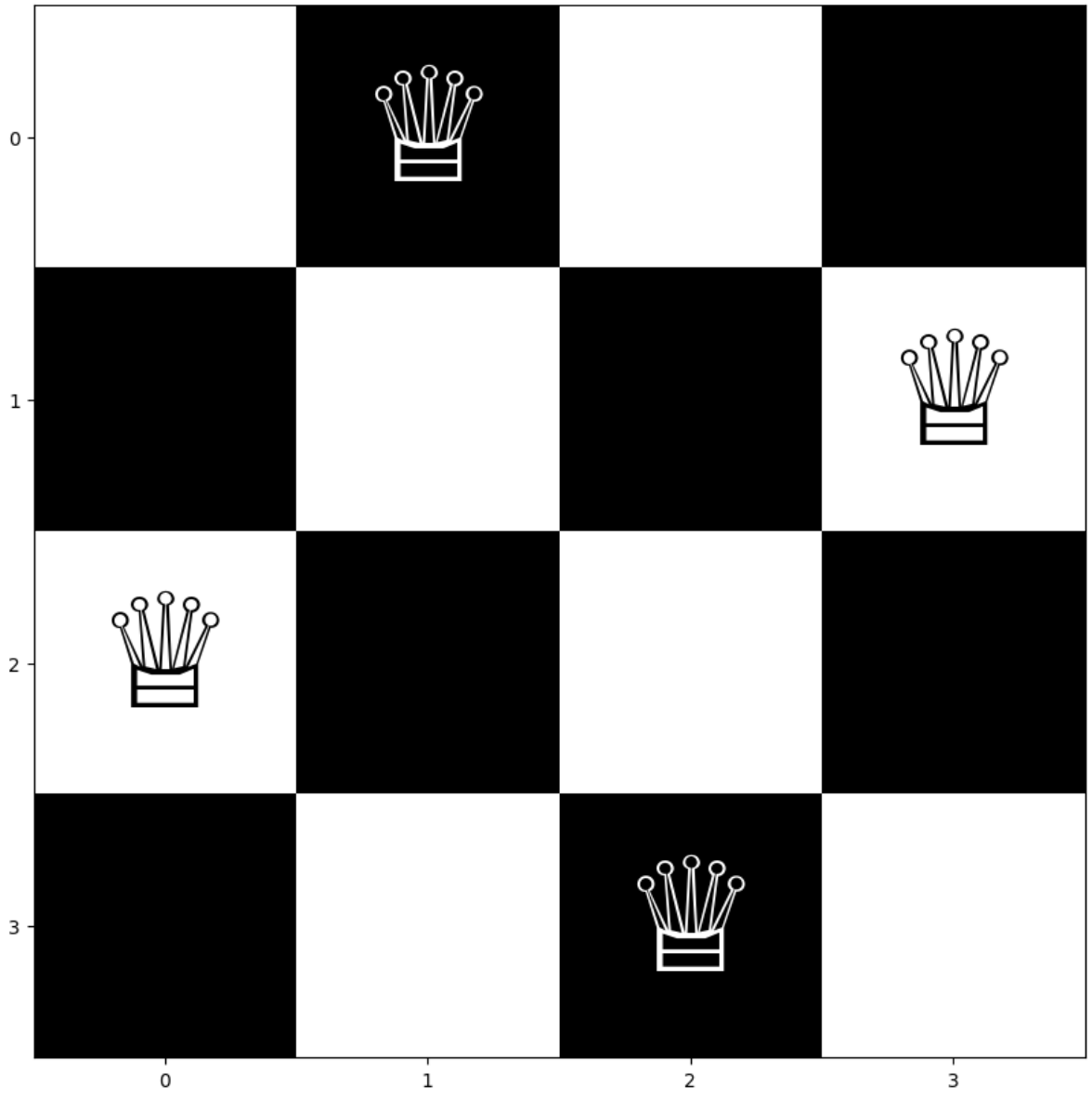


Figure 9.2: A 4-Queens solution

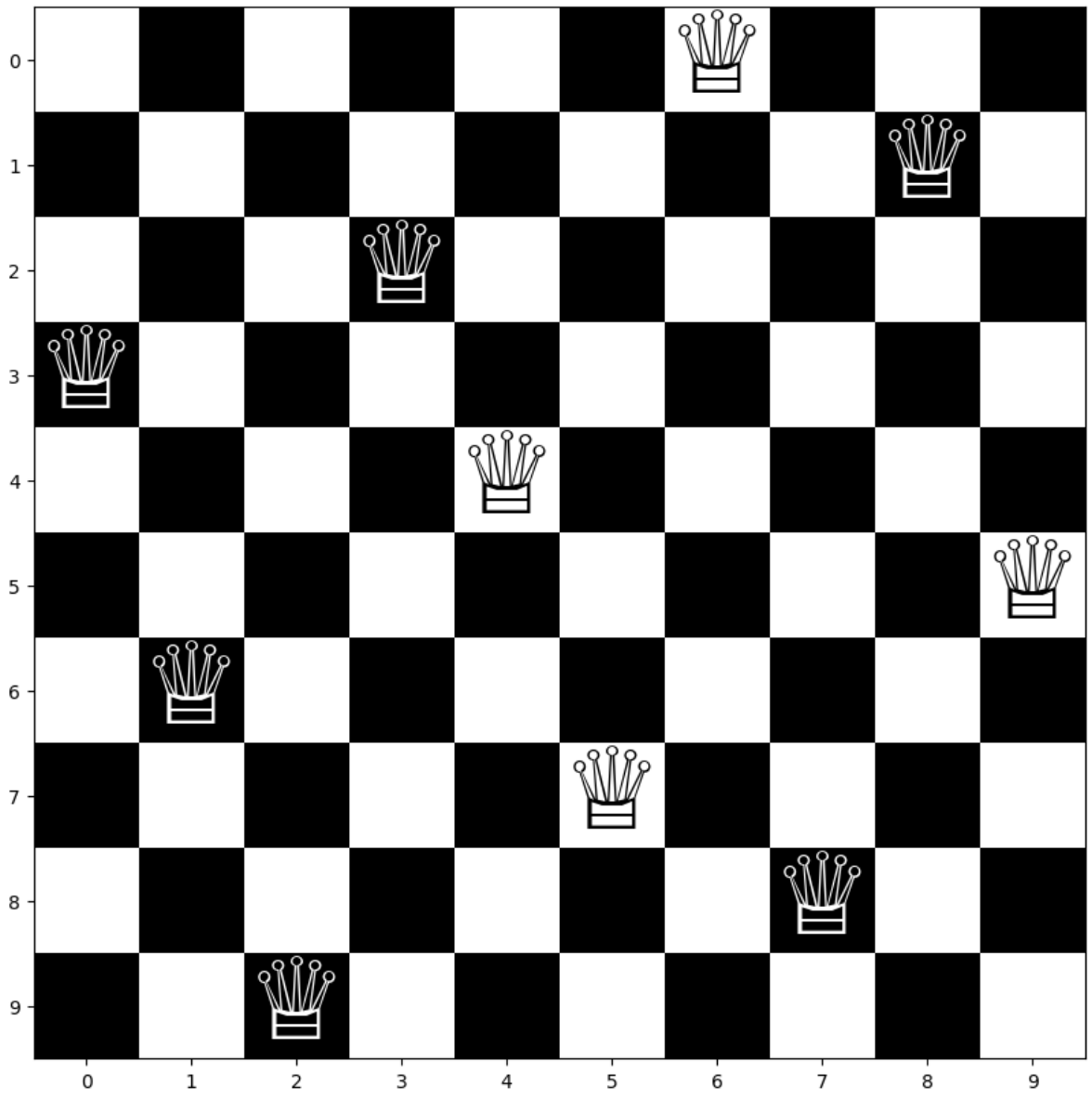


Figure 9.3: A 10-Queens solution

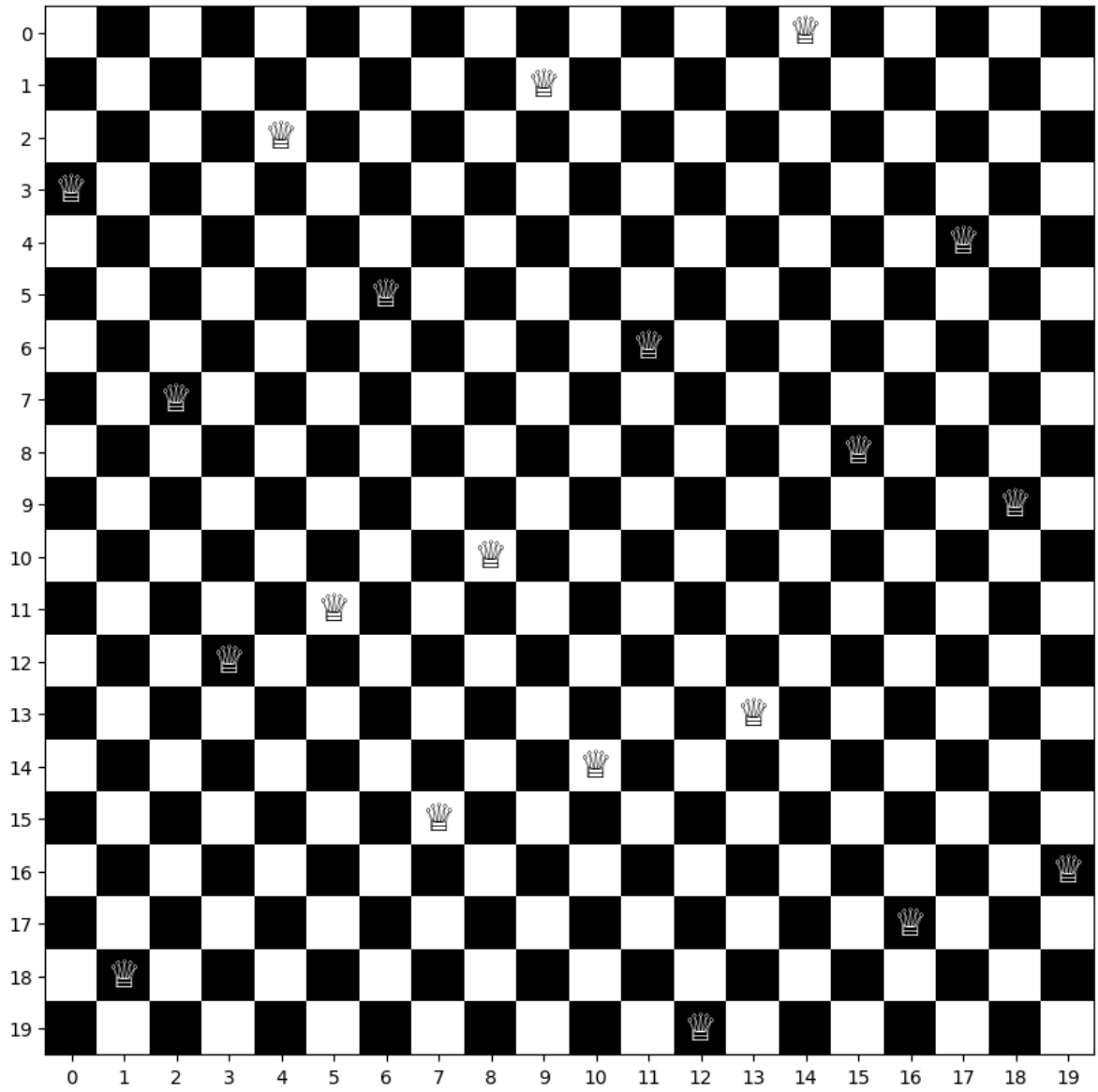


Figure 9.4: A 20-Queens solution

10 Single Row Facility Layout Problem

Single Row Facility Layout Problem (SRFLP) is a type of optimization problem in which the objective is to arrange a set of facilities in a single row in such a way as to minimize the total cost of movement between facilities. In other words, the goal is to find the best possible arrangement of facilities in a linear fashion that minimizes the total distance traveled by workers or materials between facilities. The SRFLP has a wide range of applications in various industries:

- Manufacturing Industry: In manufacturing plants, the SRFLP is used to determine the optimal placement of machines, workstations, and other facilities in a single line to minimize material handling and transportation costs. By arranging the facilities in an optimal sequence, the movement of raw materials, work-in-progress, and finished products can be minimized, reducing production time and costs.
- Warehousing and Distribution: In the warehousing and distribution industry, the SRFLP is used to optimize the layout of storage areas, packing and shipping stations, and other facilities. By minimizing the distance traveled between facilities, the time and cost of moving products within the warehouse or distribution center can be reduced.
- Retail Industry: The SRFLP can be used to optimize the layout of retail stores, such as supermarkets and department stores. By arranging product displays and checkout stations in an optimal sequence, customer flow can be improved, and waiting times can be reduced, resulting in better customer satisfaction and increased sales.
- Healthcare Industry: In hospitals and clinics, the SRFLP can be used to optimize the layout of patient rooms, laboratories, and other medical facilities. By arranging these facilities in an optimal sequence, healthcare professionals can move more efficiently, reducing patient wait times and improving the quality of care.
- Office Layout: The SRFLP can also be used to optimize the layout of office spaces, including the arrangement of desks, meeting rooms, and common areas. By minimizing the distance between facilities, the productivity and efficiency of workers can be improved.

In summary, the SRFLP has many applications in different industries, where the optimization of facility layout can result in significant cost savings, improved productivity, and increased customer satisfaction.

The SRFLP can be defined as follows: Assume there are n rooms of different lengths h_i that need to be arranged in a single row. The flow of materials between each pair of rooms, denoted

by w_{ij}, is already known. Let $\mathcal{N}$ represent the set of rooms to be arranged, which consists of integers from 0 to $n-1$. Let H be the sum of the lengths of all the rooms, and let V be a large number.

To formulate the problem mathematically, we introduce the following variables:

- x_{ij}: a binary variable that takes the value of 1 if room i is placed to the left of room j, and 0 otherwise.
- y_i: a continuous variable that represents the location of the starting point of room i on the line.
- l_{ij}: a continuous variable that represents the distance between the centroid of room i and the centroid of room j if room i is placed to the left of room j, and 0 otherwise.
- r_{ij}: a continuous variable that represents the distance between the centroid of room i and the centroid of room j if room i is placed to the right of room j, and 0 otherwise.

The SRFLP can then be formulated as follows:

$$\text{min.} \quad \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} w_{ij} \cdot (l_{ij} + r_{ij}) \tag{10.1}$$

$$\text{s.t.} \quad 0 \leq y_i \leq H, \ \forall i = 0, \cdots, n-1 \tag{10.2}$$

$$y_i \geq y_j + h_j - V \cdot x_{ij}, \ \forall i, j \in \mathcal{N}, \ i < j \tag{10.3}$$

$$y_j \geq y_i + h_i - V \cdot (1 - x_{ij}), \ \forall i, j \in \mathcal{N}, \ i < j \tag{10.4}$$

$$r_{ij} - l_{ij} = y_i - y_j + \frac{1}{2}(h_i - h_j), \ \forall i, j \in \mathcal{N}, i \neq j \tag{10.5}$$

$$x_{ij} \in \{0, 1\}, \ \forall i, j \in \mathcal{N}, i \neq j \tag{10.6}$$

$$y_i \geq 0, \ \forall i \in \mathcal{N} \tag{10.7}$$

$$l_{ij} \geq 0, \ \forall i, j \in \mathcal{N}, i \neq j \tag{10.8}$$

$$r_{ij} \geq 0, \ \forall i, j \in \mathcal{N}, i \neq j \tag{10.9}$$

$$\tag{10.10}$$

The goal of the objective function (10.1) is to minimize the total cost of moving all the rooms. The constraints (10.2) ensure that the starting point of each room is within the range of $[0, H]$. Constraints (10.3) and (10.4) are complementary disjunctive constraints that ensure that no two rooms overlap with each other. Constraints (10.5) calculate the distance between the centroids of any two rooms placed in a specific order on the line. The remaining constraints specify the type of variables used.

To evaluate the effectiveness of the formulated model, we utilize benchmarking instances obtained from an online source (https://grafo.etsii.urjc.es/optsicom/srflp.html). In the following

code, we define a class called SRFLPDataCenter. This class is responsible for parsing the instance file and storing the relevant information in its attributes.

```
import numpy as np

class SRFLPDataCenter:

    def __init__(self):
        self._num_rooms = None
        self._room_lengths = None
        self._distance_matrix = None

    def read_data(self, data_file):
        with open(data_file) as f:
            first_line = f.readline()
            self._num_rooms = int(first_line)

            second_line = f.readline().split()
            self._room_lengths = [
                float(v) for v in second_line
            ]

            self._distance_matrix = \
                np.zeros((self._num_rooms, self._num_rooms))
            for row in range(self._num_rooms):
                line = f.readline().split()
                for col in range(self._num_rooms):
                    self._distance_matrix[row][col] = \
                        float(line[col])

    @property
    def num_rooms(self): return self._num_rooms

    def get_room_length(self, room_idx):
        return self._room_lengths[room_idx]

    def get_distance(self, i, j):
        return self._distance_matrix[i][j]
```

The following code provides a comprehensive program that uses OR-Tools to solve the SRFLP.

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

class SRFLPSolver:

    def __init__(self, data_center):
        self._data_center = data_center

        self._solver = None
        self._var_x = None
        self._var_y = None
        self._var_l = None
        self._var_r = None

        self._opt_obj = None
        self._opt_y = None

    def build_model(self):
        self._solver = pywraplp.Solver.CreateSolver('SCIP')

        self._create_variables()
        self._create_objective()
        self._create_constraints()

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

        H = sum([self._data_center.get_room_length(r)
                for r in range(num_rooms)])
        self._var_y = [
                    self._solver.NumVar(0, H, name=f"y_{i}")
```

```
                for i in range(num_rooms)
            ]

        infinity = self._solver.Infinity()
        self._var_l = np.empty((num_rooms, num_rooms),
                               dtype=object)
        self._var_r = np.empty((num_rooms, num_rooms),
                               dtype=object)
        for i, j in product(range(num_rooms),
                            range(num_rooms)):
            if i == j: continue
            self._var_l[i][j] = \
                self._solver.NumVar(0,
                                    infinity,
                                    name=f"l_{i,j}")
            self._var_r[i][j] = \
                self._solver.NumVar(0,
                                    infinity,
                                    name=f"r_{i,j}")

    def _create_objective(self):
        num_rooms = self._data_center.num_rooms
        expr = [
            self._data_center.get_distance(i, j) *
            (self._var_l[i][j] + self._var_r[i][j])
            for i in range(0, num_rooms - 1)
            for j in range(i + 1, num_rooms)
        ]
        self._solver.Minimize(self._solver.Sum(expr))

    def _create_constraints(self):
        num_rooms = self._data_center.num_rooms
        H = sum([self._data_center.get_room_length(r)
                 for r in range(num_rooms)])
        for i in range(0, num_rooms):
            hi = self._data_center.get_room_length(i)
            for j in range(i + 1, num_rooms):
                hj = self._data_center.get_room_length(j)
                self._solver.Add(
                    self._var_y[i] >=
                    self._var_y[j] +
```

```
                    hj -
                    H * self._var_x[i][j]
                )

                self._solver.Add(
                    self._var_y[j] >=
                    self._var_y[i] +
                    hi -
                    H * (1 - self._var_x[i][j])
                )

        for i, j in product(range(num_rooms),
                            range(num_rooms)):
            if i == j: continue
            hi = self._data_center.get_room_length(i)
            hj = self._data_center.get_room_length(j)
            self._solver.Add(
                self._var_r[i][j] - self._var_l[i][j] ==
                self._var_y[i] - self._var_y[j] +
                0.5 * (hi - hj)
            )

    def optimize(self):
        status = self._solver.Solve()
        if status == pywraplp.Solver.OPTIMAL:
            self._opt_obj = self._solver.Objective().Value()
            print(f"obj = {self._opt_obj:.2f}")

            num_rooms = self._data_center.num_rooms
            self._opt_y = [
                self._var_y[r].solution_value()
                for r in range(num_rooms)
            ]
            for r in range(num_rooms):
                print(f"room {r} starting position: \
                    {self._opt_y[r]:.2f},\t\
                    length: \
                        {self._data_center.get_room_length(r)}")
```

To test the program since the original instance is too large, we use the first 8 rooms from the instance *AnKeVa_2005_60dept_set1* and solve it using OR-Tools. The program will output the optimal objective value and the positions of all the rooms.

```
import os

data_dir = "./data/srflp/Anjos"
data_file = "AnKeVa_2005_60dept_set1-mini.txt"

data_center = SRFLPDataCenter()
data_center.read_data(os.path.join(data_dir, data_file))

solver = SRFLPSolver(data_center)
solver.build_model()
solver.optimize()
```

```
obj = 3715.00
room 0 starting position: 217.00,                    length: 53.0
room 1 starting position: 138.00,                    length: 7.0
room 2 starting position: 53.00,                     length: 47.0
room 3 starting position: 147.00,                    length: 15.0
room 4 starting position: 100.00,                    length: 38.0
room 5 starting position: 189.00,                    length: 28.0
room 6 starting position: 162.00,                    length: 27.0
room 7 starting position: 145.00,                    length: 2.0
```

11 Warehouse Location Problem

The warehouse location problem (WLP) is a classic optimization problem in operations research that aims to find the optimal locations for warehouses in order to minimize transportation costs while meeting the demand for goods from a set of customers. The problem is particularly relevant for businesses that need to distribute their products across a large geographic region.

The problem can be formulated as follows: Given a set of stores $\mathcal{S} = \{1, \cdots, s\}$ and a set of potential warehouse locations $\mathcal{W} = \{1, \cdots, w\}$, the objective is to select a subset of warehouse locations and allocate stores to these locations such that the total transportation cost is minimized. The transportation cost typically depends on the distance between each store and the warehouse they are assigned to, as well as the quantity of goods that need to be transported.

In order to simplify our mathematical model, we utilize the following symbols to indicate the input parameters:

- f_w: the fixed cost required for initiating warehouse w
- c_{ws}: the cost of transporting goods from warehouse w to store s
- d_s: the quantity of goods demanded by store s
- N_w: the storage capacity of warehouse w

We can then define two variables:

- y_s: a binary variable that takes on a value of 1 if warehouse s is chosen, and 0 otherwise
- x_{ws}: a continuous variable that represents the fraction of store s's demand that will be met by warehouse w

The complete model of this problem is given below.

$$\text{min.} \quad \sum_{w \in \mathcal{W}} \sum_{s \in \mathcal{S}} c_{ws} d_s x_{ws} + \sum_{w \in \mathcal{W}} f_w y_w \tag{11.1}$$

$$\text{s.t.} \quad \sum_{w \in \mathcal{W}} x_{ws} = 1, \ \forall s \in \mathcal{S} \tag{11.2}$$

$$\sum_{s \in \mathcal{S}} d_s x_{ws} \leq N_w y_w, \ \forall w \in \mathcal{W} \tag{11.3}$$

$$0 \leq x_{ws} \leq 1, \ \forall w \in \mathcal{W}, s \in \mathcal{S} \tag{11.4}$$

$$y_w \in \{0, 1\}, \ \forall w \in \mathcal{W} \tag{11.5}$$

The objective function (11.1) is to minimize the total cost of transportation and the fixed cost of opening warehouses. The first term of the objective function sums up the transportation cost of moving goods from each warehouse w to each store s, multiplied by the proportion of store s's demand met by warehouse w, represented by x_{ws}. The second term of the objective function sums up the fixed cost of opening each warehouse w, represented by f_w, multiplied by a binary variable y_w that takes on a value of 1 if warehouse w is selected, and 0 otherwise.

The model is subject to four constraints:

- Constraint (11.2) ensures that the entire demand of each store s is met. The sum of x_{ws} over all warehouses must be equal to 1 for each store s.
- Constraint (11.3) ensures that the total demand of all stores served by warehouse w does not exceed its capacity N_w. The sum of $d_s x_{ws}$ over all stores s must be less than or equal to $N_w y_w$ for each warehouse w.
- Constraint (11.4) ensures that the fraction of store s's demand met by warehouse w, represented by x_{ws}, is between 0 and 1.
- Constraint (11.5) ensures that the binary variable y_w takes on a value of either 0 or 1, indicating whether or not warehouse w is selected.

To demonstrate the solution process for the Warehouse Location Problem (WLP) using OR-Tools, we will use some sample instances available online from the website https://opthub.uniud.it/problem/facility-location/wlp. Specifically, we will be using an instance file called *wlp2*, the contents of which are shown below.

```
Warehouses = 2;
Stores = 5;

Capacity = [65, 47];
FixedCost = [80, 184];
Goods = [4, 15, 17, 6, 6];
SupplyCost = [|57, 71
              |30, 59
              |43, 71
              |37, 72
              |30, 68|];
```

In this instance file, there are 2 warehouses and 5 stores. The *Capacity* array specifies the maximum capacity of each warehouse, where the first warehouse has a capacity of 65 and the second has a capacity of 47. The *FixedCost* array specifies the fixed cost of opening each warehouse, where the first warehouse has a fixed cost of 80 and the second has a fixed cost of 184. The *Goods* array specifies the demand for each store, where the first store has a demand of 4, the second store has a demand of 15, the third store has a demand of 17, the fourth store has a demand of 6, and the fifth store has a demand of 6. The *SupplyCost* matrix specifies the transportation cost of moving goods from each warehouse to each store. The entry in row i

and column j of the matrix represents the transportation cost of moving goods from warehouse j to store i. For example, the transportation cost of moving goods from the first warehouse to the first store is 57, the transportation cost of moving goods from the second warehouse to the fourth store is 71, etc.

We will create a `WlpDataCenter` class that will be responsible for reading and storing the necessary information required for solving the problem later on.

```
import re

class WlpDataCenter:

    def __init__(self):
        self._num_warehouses = None
        self._num_stores = None
        self._capacities = None
        self._fixed_costs = None
        self._demands = None
        self._transport_costs = None

    def read_data(self, data_file):
        with open(data_file) as f:
            lines = f.readlines()

        self._num_warehouses = int(re.findall(r'\d+',
                                              lines[0])[0])
        self._num_stores = int(re.findall(r'\d+',
                                          lines[1])[0])

        self._capacities = [
                    int(num)
                    for num in re.findall(r'\d+', lines[3])
                ]
        self._fixed_costs = [
                    int(num)
                    for num in re.findall(r'\d+', lines[4])
                ]
        self._demands = [
                    int(num)
                    for num in re.findall(r'\d+', lines[5])
                ]
        self._transport_costs = []
```

```
        for line in lines[6:]:
            numbers = [
                    int(num)
                    for num in re.findall(r'\d+', line)
                ]
            self._transport_costs.append(numbers)

    @property
    def num_warehouses(self): return self._num_warehouses

    @property
    def num_stores(self): return self._num_stores

    @property
    def capacities(self): return self._capacities

    @property
    def demands(self): return self._demands

    @property
    def fixed_costs(self): return self._fixed_costs

    @property
    def transport_costs(self): return self._transport_costs
```

Now we are ready to solve the WLP using OR-Tools!

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

class WlpSolver:

    def __init__(self, data_center):
        self._data_center: WlpDataCenter = data_center

        self._solver = None
        self._var_x = None
        self._var_y = None

        self._opt_obj = None
```

```
        self._opt_x = None
        self._opt_y = None

    def build_model(self):
        self._solver = pywraplp.Solver.CreateSolver('SCIP')

        self._create_variables()
        self._create_objective()
        self._create_constraints()

    def optimize(self):
        self._solver.SetTimeLimit(20000)
        status = self._solver.Solve()
        if status == pywraplp.Solver.OPTIMAL:
            num_warehouses = self._data_center.num_warehouses
            num_stores = self._data_center.num_stores
            self._opt_obj = self._solver.Objective().Value()
            self._opt_x = np.zeros((num_warehouses,
                                    num_stores))
            for w, s in product(range(num_warehouses),
                                range(num_stores)):
                self._opt_x[w][s] = \
                    self._var_x[w][s].solution_value()

            self._opt_y = [
                self._var_y[w] for w in range(num_warehouses)
            ]

    def _create_variables(self):
        num_warehouses = self._data_center.num_warehouses
        num_stores = self._data_center.num_stores
        self._var_x = np.empty((num_warehouses, num_stores),
                               dtype=object)
        for w, s in product(range(num_warehouses),
                               range(num_stores)):
            self._var_x[w][s] = \
                self._solver.NumVar(0, 1,
                                    name=f'x_{w,s}')

        self._var_y = [
            self._solver.BoolVar(name=f'y_{w}')
```

```
            for w in range(num_warehouses)
        ]

    def _create_objective(self):
        num_warehouses = self._data_center.num_warehouses
        num_stores = self._data_center.num_stores
        demands = self._data_center.demands
        transport_costs = self._data_center.transport_costs
        fixed_costs = self._data_center.fixed_costs
        expr1 = [
            transport_costs[s][w] *
            demands[s] *
            self._var_x[w][s]
            for w in range(num_warehouses)
            for s in range(num_stores)
        ]
        expr2 = [
            fixed_costs[w] *
            self._var_y[w]
            for w in range(num_warehouses)
        ]
        self._solver.Minimize(
            self._solver.Sum(expr1) +
            self._solver.Sum(expr2)
        )

    def _create_constraints(self):
        num_warehouses = self._data_center.num_warehouses
        num_stores = self._data_center.num_stores
        for s in range(num_stores):
            expr = [
                self._var_x[w][s]
                for w in range(num_warehouses)
            ]
            self._solver.Add(
                self._solver.Sum(expr) == 1
            )

        demands = self._data_center.demands
        capacities = self._data_center.capacities
        for w in range(num_warehouses):
```

```
            expr = [
                self._var_x[w][s] *
                demands[s]
                for s in range(num_stores)
            ]
            self._solver.Add(
                self._solver.Sum(expr) <=
                capacities[w] *
                self._var_y[w]
            )

    @property
    def opt_obj(self): return self._opt_obj

    @property
    def opt_x(self): return self._opt_x

    @property
    def opt_y(self): return self._opt_y
```

The program defines a class named `WlpSolver` which has methods to build the model, optimize it and retrieve the optimized solution.

The `WlpSolver` class takes an instance of `WlpDataCenter` as input, which contains the data necessary to solve the problem. The `build_model()` method creates variables, objectives, and constraints for the problem using the OR-Tools library. The `optimize()` method solves the problem and saves the optimized objective function value, `opt_obj`, the matrix `opt_x` that indicates the optimal assignment of stores to warehouses, and a list `opt_y` of binary variables that indicate whether or not each warehouse is open.

The `_create_variables()` method creates two types of variables: a matrix of continuous variables that indicates the proportion of a store's demands that's served by a warehouse and a list of binary variables that indicate whether or not each warehouse is open. The `_create_objective()` method creates the objective function of the problem, which is to minimize the total cost of opening warehouses and serving stores. The `_create_constraints()` method creates constraints that ensure that every store's demands are fully fulfilled and a warehouse's capacity is not violated.

Finally, the `opt_obj`, `opt_x`, and `opt_y` properties allow access to the results of the optimization.

Table 11.1 reports the best solutions found within the time limits of 10 seconds for some testing instances.

Table 11.1: Computational results of WLP instances

Instance	No. Warehouses	No. Stores	Best Solution
wlp1	1	3	1931
wlp2	2	5	1891
wlp3	3	7	4358
wlp4	4	10	4246
wlp5	5	12	3502
wlp6	6	15	4108
wlp7	7	17	4276
wlp8	8	20	4888
wlp9	9	22	7959
wlp10	10	25	8893
wlp12	12	30	4890
wlp15	15	37	14881
wlp20	20	50	9727
wlp30	30	75	11964
wlp50	50	120	15164
wlp100	100	220	20848

12 The Cutting Stock Problem

The cutting stock problem (CSP) is a common problem in the paper industry, where large rolls of paper must be cut into smaller rolls of various widths to meet customer demands. For example, a paper mill may have a large roll of paper that is 60 inches wide and needs to produce smaller rolls of widths 30, 24, 18, and 12 inches. The mill must determine the best way to cut the large roll of paper into the required widths while minimizing waste.

Let's say that a paper company produces rolls with a uniform width of 100 inches and receives orders for rolls with widths of 20, 30, 40, and 50 inches. Table 12.1 shows the order details.

Table 12.1: Demands of different paper rolls

Order Width	Order Quantity
20	100
30	120
40	40
50	20

To fulfill these orders, a single 100 inch roll can be cut into one or more of the order widths. This process generates scrap, which is the leftover material that cannot be used. The different combinations of cuts are called patterns. In this case, there are many possible patterns but for simplicity purpose, we'll assume only the 10 patterns listed in Table 12.2 are available.

Table 12.2: Available patterns

Pattern	Width (20)	Wdith (30)	Width (40)	Width (50)
Pattern 1	1	1	1	0
Pattern 2	1	1	0	1
Pattern 3	0	0	1	1
Pattern 4	3	0	1	0
Pattern 5	0	2	1	0
Pattern 6	0	0	0	2
Pattern 7	5	0	0	0
Pattern 8	0	3	0	0
Pattern 9	0	0	2	0

Pattern	Width (20)	Wdith (30)	Width (40)	Width (50)
Pattern 10	2	0	0	1

The objective of the cutting stock problem is to reduce the amount of material wastage while fulfilling the specified demands. The demand for each order is denoted by r_i, and the number of order size i in pattern j is represented by a_{ij}. To achieve this objective, we introduce the variable x_j which is a non-negative integer representing the frequency with which pattern j is used to fulfill the orders. The index j ranges from 1 to 10. Thus, the problem can be expressed as follows.

$$\text{min.} \quad \sum_{j=1}^{n} x_j \tag{12.1}$$

$$\text{s.t.} \quad \sum_{j=1}^{n} a_{ij} x_j \geq r_i, \ \forall i \in \mathcal{I} \tag{12.2}$$

$$x_j \in \mathbb{Z}^+ = \{1, 2, 3, ...\}, \ \forall j \in \{1, \cdots, 10\} \tag{12.3}$$

Now we solve this contrived cutting stock problem using OR-Tools. The code below gives the complete program.

```
from ortools.linear_solver import pywraplp

# prepare data
num_orders = 4
order_quantities = [100, 120, 40, 20]
num_patterns = 10
pattern_details = [
    [1, 1, 1, 0],
    [1, 1, 0, 1],
    [0, 0, 1, 1],
    [3, 0, 1, 0],
    [0, 2, 1, 0],
    [0, 0, 0, 2],
    [5, 0, 0, 0],
    [0, 3, 0, 0],
    [0, 0, 2, 0],
    [2, 0, 0, 1]
]

# instantiate solver
```

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

# create decision varaibles
infinity = solver.Infinity()
var_j = [
    solver.IntVar(0, infinity, name=f'x_{j}')
    for j in range(num_patterns)
]

# create objective function
solver.Minimize(solver.Sum(var_j))

# create constraints
for i in range(num_orders):
    expr = [
        var_j[j] * pattern_details[j][i]
        for j in range(num_patterns)
    ]
    solver.Add(solver.Sum(expr) >= order_quantities[i])

# solve the problem
status = solver.Solve()
if status == pywraplp.Solver.OPTIMAL:
    obj = solver.Objective().Value()
    print(f"optimal obj = {obj}")

    opt_j = [
        var_j[j].solution_value()
        for j in range(num_patterns)
    ]
    print(opt_j)
```

```
optimal obj = 83.0
[0.0, 20.0, 0.0, 0.0, 40.0, 0.0, 16.0, 7.0, 0.0, 0.0]
```

Table 12.3 shows the final quantities produced for every order width.

Table 12.3: Solution

Pattern	Width (20)	Wdith (30)	Width (40)	Width (50)	Optimal Quantity
Pattern 1	1	1	1	0	0
Pattern 2	1	1	0	1	20
Pattern 3	0	0	1	1	0
Pattern 4	3	0	1	0	0
Pattern 5	0	2	1	0	40
Pattern 6	0	0	0	2	0
Pattern 7	5	0	0	0	16
Pattern 8	0	3	0	0	7
Pattern 9	0	0	2	0	0
Pattern 10	2	0	0	1	0
Produced Order	100	121	40	20	83

This example problem seems trivial to solve, but in practice the CSP is challenging to solve because of the large number of possible cutting patterns that can be used and the combinatorial nature of the problem.

To efficiently solve the cutting stock problem, a column generation approach is often used. The basic idea behind column generation is to start with a small subset of the possible cutting patterns, and then iteratively generate and add new cutting patterns to the problem until an optimal solution is found.

The column generation approach involves solving a master problem and a subproblem iteratively. The master problem involves selecting the best set of cutting patterns from a larger set of potential patterns, while the subproblem involves finding the next cutting pattern(s) to add to the master problem. By solving these two problems iteratively, the algorithm can gradually add new cutting patterns to the master problem until the optimal solution is reached.

The column generation approach has several advantages over other methods for solving the cutting stock problem. First, it can handle large-scale instances of the problem more efficiently. This is because the algorithm only considers a subset of the possible cutting patterns at each iteration, which reduces the computational time and memory required. Second, the approach is flexible and can easily handle changes in the problem parameters, such as demand or material availability. Finally, the approach is guaranteed to converge to an optimal solution, provided certain conditions are met.

Overall, the cutting stock problem is challenging to solve due to its combinatorial nature and large number of possible cutting patterns. The column generation approach is a powerful technique for solving this problem efficiently and effectively, and is widely used in practice.

13 The Bin Packing Problem

The bin packing problem (BPP) is a classic optimization problem in computer science that involves packing a set of items into a minimum number of bins or containers, subject to certain constraints. The problem is also known as the container loading problem, or the bin packing optimization problem.

In the bin packing problem, we are given a set of items, each with a weight or size, and a set of bins, each with a fixed capacity. The goal is to pack the items into the bins such that the number of bins used is minimized.

The problem is NP-hard, which means that there is no known algorithm that can solve it in polynomial time. However, several heuristic algorithms have been developed that can provide good solutions in practice.

The bin packing problem has many real-world applications, such as optimizing the use of storage space in warehouses, minimizing the number of trucks needed for transportation, and optimizing the use of computer memory. It is also a fundamental problem in the study of computational complexity theory and has been extensively studied in the fields of operations research, computer science, and mathematics.

14 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.

Chen, Yuning, and Jin-Kao Hao. 2014. "A 'Reduce and Solve' Approach for the Multiple-Choice Multidimensional Knapsack Problem." *European Journal of Operational Research* 239 (2): 313–22. https://doi.org/10.1016/j.ejor.2014.05.025.

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.

Dudziński, Krzysztof, and Stanisław Walukiewicz. 1987. "Exact Methods for the Knapsack Problem and Its Generalizations." *European Journal of Operational Research* 28 (1): 3–21. https://doi.org/10.1016/0377-2217(87)90165-2.

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.

Letavec, Craig, and John Ruggiero. 2002. "The n -Queens Problem." *INFORMS Transactions on Education* 2 (3): 101–3. https://doi.org/10.1287/ited.2.3.101.

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.

Petersen, Clifford C. 1967. "Computational Experience with Variants of the Balas Algorithm Applied to the Selection of R&D Projects." *Management Science* 13 (9): 736–50. https://doi.org/10.1287/mnsc.13.9.736.

Sinha, Prabhakant, and Andris A. Zoltners. 1979. "The Multiple-Choice Knapsack Problem." *Operations Research* 27 (3): 503–15. https://www.jstor.org/stable/170213.

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.