appendix A
Search and optimization libraries in Python

This appendix covers setting up the Python environment, along with essential libraries for mathematical programming, graph visualization, metaheuristic optimization, and machine learning.

A.1 Setting up the Python environment

This book assumes that you already have Python 3.6 or a newer version installed on your system. For installation instructions specific to your operating system, see this Beginner's Guide: https://wiki.python.org/moin/BeginnersGuide/.

For Windows, you can follow these steps to install Python:

1. Go to the official website: www.python.org/downloads/.
2. Select the version of Python to install.
3. Download the Python executable installer.
4. Run the executable installer. Make sure you check the Install Launcher for all users and Add Python 3.8 to PATH checkboxes.
5. Verify that Python was successfully installed by typing `python -V` in a command prompt.
6. Verify that pip was installed by typing `pip -V` in a command prompt.
7. Install `virtualenv` by typing `pip install virtualenv` in a command prompt.

If you are a Linux user, execute the following commands in the terminal:

```
$ sudo apt update
$ sudo apt install python3-pip
```

Install venv and create a Python virtual environment:

```
$ sudo apt install python3.8-venv
$ mkdir <new directory for venv>
$ python -m venv <path to venv directory>
```

Make sure that you replace `python3.8` with the version of Python you are using.

You can now access your virtual environment using the following command:

```
$ source <path to venv>/bin/activate
```

In the case of macOS, Python is already preinstalled, but if you need to upgrade or install a specific version, you can use the macOS terminal as follows:

```
$ python -m ensurepip --upgrade
```

`venv` is included with python 3.8+. You can run the following command to create a virtual environment:

```
$ mkdir <new directory>
$ python -m venv <path to venv directory>
```

You can now access your virtual environment using the following command:

```
$ source <path to venv>/bin/activate
```

A better option is to install a Python distribution as explained in the next subsection.

A.1.1 Using a Python distribution

Python distributions, such as Anaconda or Miniconda, come with a package manager called `conda` that allows you to install a wide range of Python packages and manage different Python environments. Install `conda` for your OS using the guide found here: https://conda.io/projects/conda/en/latest/user-guide/install/index.html.

Conda environments are used to manage multiple installations of different versions of Python packages and their dependencies. You can create a `conda` environment with this command:

```
$ conda create --name <name of env> python=<your version of python>
```

Access the newly created environment like this:

```
$ conda activate <your env name>
```

This command allows you to switch or move between environments.

A.1.2 Installing Jupyter Notebook and JupyterLab

Jupyter is a multi-language, open-source web-based platform for interactive programming (https://jupyter.org/). The name "Jupyter" is a loose acronym meaning Julia, Python, and R. All of the code in this book is stored in Jupyter notebooks (.ipynb files), which can be opened and edited using JupyterLab or Jupyter Notebook. Jupyter Notebook feels more standalone, but JupyterLab feels more like an IDE.

You can install JupyterLab using `pip` as follows:

```
$ pip install jupyterlab
$ pip install notebook
```

Or using conda as follows:

```
$ conda install -c conda-forge jupyterlab
$ conda install -c conda-forge notebook
```

You can install the Python ipywidgets package to automatically configure classic Jupyter Notebook and JupyterLab 3.0 to display ipywidgets using pip or conda as follows:

```
$ pip install ipywidgets
$ conda install -c conda-forge ipywidgets
```

If you have an old version of Jupyter Notebook installed, you may need to manually enable the ipywidgets notebook extension with these commands:

```
$ jupyter nbextension install --user --py widgetsnbextension
$ jupyter nbextension enable --user --py widgetsnbextension
```

Google Colaboratory (Colab) can also be used. This cloud-based tool allows you to write, execute, and share Python code through the browser. It also provides free access to GPU and TPU for increased computational power. You can access Colab here: https://colab.research.google.com/.

A.1.3 Cloning the book's repository

You can clone this book's code repository as follows:

```
$git clone https://github.com/Optimization-Algorithms-Book/Code-Listings.git
```

Many of the operations in this book are long and burdensome to code from scratch. Often, they're highly standardized and can benefit from having a helper function take care of the various intricacies. optalgotools is a Python package developed for this purpose.

You can use these supporting tools locally without installing this package. To do so, you will need to download optalgotools in a local folder and add this folder to the system path. If you're using Jupyter notebook or Jupyter lab, you can do so as follows:

```
import sys
sys.path.insert(0, '../')
```

If you're using Colab, you can mount your Google Drive with these commands:

```
from google.colab import drive
drive.mount('/content/drive')
```

Then copy the optalgotools folder to your Google Drive.

This package is also available on the Python Package Index (PyPI) repository here: https://pypi.org/project/optalgotools/. You can install it as follows:

```
$pip install optalgotools
```

You can then use the import command to use these tools. Here is an example:

```
from optalgotools.problems import TSP
from optalgotools.algorithms import SimulatedAnnealing
```

The first line imports the TSP instance from the problems module, and the second line imports a simulated annealing solver from the algorithms module.

A.2 Mathematical programming solvers

Mathematical programming, also known as *mathematical optimization*, is the process of finding the best solution to a problem that can be represented in mathematical terms. It involves formulating a mathematical model of a problem, determining the parameters of the model, and using mathematical and computational techniques to find a solution that maximizes or minimizes a particular objective function or set of objective functions subject to a set of constraints. Linear programming (LP), mixed-integer linear programming (MILP), and nonlinear programming (NLP) are examples of mathematical optimization problems. Several Python libraries can be used for solving mathematical optimization problems.

Let's consider the following production planning example from Guerte el al.'s *Linear Programming* [1]. A small woodworking shop produces two sizes of boxwood chess sets. Crafting the smaller set involves 3 hours of lathe machining, while the larger set requires 2 hours. With four skilled operators each working a 40-hour week, the shop has a total of 160 lathe-hours available weekly. The smaller chess set consumes 1 kg of boxwood, while the larger set requires 3 kg. However, due to scarcity, only 200 kg of boxwood can be obtained per week. Upon sale, each large chess set generates a profit of $12, while each small set yields $5 in profit. The objective is to determine the optimal weekly production quantities for each set to maximize profit.

Let's assume that x_1 and x_2 are decision variables that represent the number of small and large chess sets respectively to make. The total profit is the sum of the individual profits from making and selling the x_1 small sets and the x_2 large sets: profit = $5x_1 + 12x_2$. However, this profit is subject to the following constraints:

- The total number of hours of machine time we will use is $3x_1 + 2x_2$. This time shouldn't exceed the maximum of 160 hours of machine time available per week. This means that $3x_1 + 2x_2 \leq 160$ (lathe-hours).
- Only 200 kg of boxwood is available each week. Since small sets use 1 kg for every set made, against 3 kg needed to make a large set, $x_1 + 3x_2 \leq 200$ (kg of boxwood).
- The joinery cannot produce a negative number of chess sets, so we have two further non-negativity constraints: x_1 and $x_2 >= 0$.

This linear programming problem can be summarized as follows. Find x_1 and x_2 that maximize $5x_1 + 12x_2$, subject to

- Machining time constraint: $3x_1 + 2x_2 \leq 160$
- Weight constraint: $x_1 + 3\ x_2 \leq 200$
- Non-negativity constraints: x_1 and $x_2 \geq 0$

Let's see how we can solve this linear programming problem using different solvers.

A.2.1 SciPy

SciPy is an open source scientific computing Python library that provides tools for optimization, linear algebra, and statistics. SciPy optimize is a submodule of the SciPy library, which includes solvers for nonlinear problems (with support for both local and global optimization algorithms), linear programing, constrained and nonlinear least-squares, root finding, and curve fitting.

To use SciPy, you will need to install it and its dependencies. You can install SciPy using the pip package manager:

```
$pip install scipy
```

Alternatively, you can use a Python distribution, such as Anaconda or Miniconda, that comes with SciPy and other scientific libraries pre-installed.

Listing A.1 shows the steps to solve the car manufacturing problem using SciPy. The code defines the coefficient vector `c` and the left-hand side (`lhs`) and right-hand side (`rhs`) of the constraint equations. The objective function represents the profit to be maximized. Since many optimization algorithms in SciPy are designed for minimization, the problem of profit maximization is typically converted into a minimization problem by minimizing the negative of the profit function. Moreover, constraints using the greater-than-or-equal-to sign cannot be defined directly. Less-than-or-equal-to must be used instead.

Listing A.1 Solving the chess set problem using SciPy

```
import numpy as np
import scipy
from scipy.optimize import linprog

c = -np.array([5,12])

lhs_constraints=([3,2],
                 [1,3])

rhs_constraints=([160,
                  200])

bounds = [(0, scipy.inf), (0, scipy.inf)]

results = linprog(c=c, A_ub=lhs_constraints, b_ub=rhs_constraints,
➥ bounds=bounds, method='highs-ds')


print('LP Solution:')
print(f'Profit: = {-round(results.fun,2)} $')
print(f'Make {round(results.x[0],0)} small sets, and make
{round(results.x[1],0)} large sets')
```

Declare coefficients of the objective function.

Left-hand side of the machining time constraint

Left-hand side of the weight constraint

Right-hand side of the machining time constraint

Right-hand side of the weight constraint

Bounds of the decision variables

Solve the linear programming problem.

Print the solution.

Running this code gives the following results:

```
LP Solution:
Profit: = 811.43 $
Make 11.0 small sets, and make 63.0 large sets
```

The `linprog()` function used in the preceding code returns a data structure with several attributes such as `x` (the current solution vector), `fun` (the current value of the objective function), and `success` (`true` when the algorithm has completed successfully).

A.2.2 PuLP

PuLP is a linear programming library in Python that allows you to define and solve linear optimization problems. There are two main classes in PuLP: `LpProblem` and `LpVariable`. PuLP variables can be declared individually or as “dictionaries” (variables indexed on another set).

You can install PuLP using pip as follows:

```
$pip install pulp
```

The following code (a continuation of listing A.1) shows how to use PuLP to solve the chess set problem:

```
#!pip install pulp
from pulp import LpMaximize, LpProblem, LpVariable, lpSum, LpStatus

model = LpProblem(name='ChessSet', sense=LpMaximize)

x1 = LpVariable('SmallSet', lowBound = 0, upBound =  None, cat='Integer')
x2 = LpVariable('LargeSet', lowBound = 0, upBound =  None, cat='Integer')

model += (3*x1 + 2*x2 <=160, 'Machining time constraint')
model += (  x1 + 3*x2 <= 200, 'Weight constraint')

profit= 5*x1 + 12*x2
model.setObjective(profit)

model.solve()

print('LP Solution:') #F
print(f'Profit: = {model.objective.value()} $') #F
print(f'Make {x1.value()} small sets, and make {x2.value()} large sets') #F
```

Define the model.

Define the decision variables.

Add constraints.

Set the profit as the objective function.

Solve the optimization problem.

Print the solution.

PuLP implements several algorithms for solving linear programming (LP) and mixed-integer linear programming (MILP) problems. Examples include COIN-OR (computational infrastructure for operations research), CLP (COIN-OR linear programming), Cbc (COIN-OR branch and cut), CPLEX (Cplex), GLPK (GNU linear programming lit), SCIP (solving constraint integer programs), HiGHS (highly scalable global solver), Gurobi LP/MIP solver, Mosek optimizer, and the XPRESS LP solver.

A.2.3 Other mathematical programming solvers

There are several other libraries in Python for solving mathematical optimization problems. This is a non-exhaustive list of other available libraries:

- OR-Tools—This is an open source software suite for optimization and constraint programming developed by Google. It includes a variety of algorithms and tools for solving problems in areas such as operations research, transportation, scheduling, and logistics. OR-Tools can be used to model and solve linear and integer programming problems, as well as constraint programming problems. Examples of OR-Tools solvers include GLOP (Google linear programming), Cbc (COIN-OR branch and cut), CP-SAT (constraint programming-satisfiability) solver, max flow and min cost flow solvers, the shortest path solver, and the BOP (binary optimization problem) solver. It is written in C++ and includes interfaces for several programming languages, including Python, C#, and Java. See section A.4.5 for more details and an example.
- Gurobi—This is commercial optimization software that offers state-of-the-art solvers for linear programming, quadratic programming, and mixed integer programming. It has a Python interface that can be used to define and solve optimization problems.
- CasADi—This is an open source tool for nonlinear optimization and algorithmic differentiation.
- Python-MIP—This is a Python library for solving mixed-integer programming problems. It is built on top of the Cbc open source optimization library and allows users to express optimization models in a high-level, mathematical programming language.
- Pyomo—This is an open source optimization modeling language that can be used to define and solve mathematical optimization problems in Python. It supports a wide range of optimization solvers, including for linear programming, mixed integer programming, and nonlinear optimization.
- GEKKO—This is a Python package for machine learning and the optimization of mixed-integer and differential algebraic equations.
- CVXPY—This is an open source Python-embedded modeling language for convex optimization problems. It lets you express your problem in a natural way that follows the math, rather than in the restrictive standard form required by solvers.
- PyMathProg—This is a mathematical programming environment for Python that allows for modeling, solving, analyzing, modifying, and manipulating linear programming problems.

- Optlang—This is a Python library for modeling and solving mathematical optimization problems. It provides a common interface to a series of optimization tools so that different solver backends can be changed in a transparent way. It is compatible with most of the popular optimization solvers like Gurobi, Cplex, and Ipopt (interior point optimizer).
- Python interface to conic optimization solvers (PICOS)—This is a Python library for modeling and solving optimization problems. It can handle complex problems with multiple objectives, and it supports both local and global optimization methods. PICOS has interfaces to different solvers such as Gurobi, CPLEX, SCS (splitting conic solver), ECOS (embedded cone solver), and MOSEK.
- CyLP—This is a Python interface to COIN-OR's linear and mixed-integer program solvers (CLP, Cbc, and CGL). COIN-OR (computational infrastructure for operations research) is a collection of open source software packages for operations research and computational optimization. It includes libraries for linear and integer programming, constraint programming, and other optimization techniques.
- SymPy—This is a Python library for symbolic mathematics. It can be used to solve equations, handle combinatorics, plot in 2D/3D, or work on polynomials, calculus, discrete math, matrices, geometry, parsing, physics, statistics, and cryptography.
- *Other libraries*—These include but are not limited to, MOSEK, CVXOPT, DOcplex, DRAKE, PySCIPOpt, PyOptim, PyMathProg, and NLPy.

Jupyter notebook "Listing A.1_Mathematical_programming_solvers.ipynb," included in the GitHub repo for the book shows how to use some of these solvers to solve the chess set problem.

A.3 Graph and mapping libraries

The following Python libraries are used in this book to process and visualize graphs, networks, and geospatial data.

A.3.1 NetworkX

NetworkX is a library for working with graphs and networks in Python. It provides tools for creating, manipulating, and analyzing graph data, as well as for visualizing graph structures. NetworkX also contains approximations of graph properties and heuristic methods for optimization. You can install NetworkX as follows:

```
$pip install networkx
```

Let's consider the traveling salesman problem (TSP). Listing A.2 shows the steps for creating a random undirected graph for this problem. Each randomly scattered node represents a city to be visited by the salesman, and the weight of each edge connecting the cities is calculated based on the Euclidian distance between the nodes using the `hypot` function, which calculates the square root of the sum of the squares. Christofides algorithm is used to solve this TSP instance—this algorithm provides a 3/2-approximation of TSP. This means that its solutions will be within a factor of 1.5 of the optimal solution length.

Listing A.2 Solving TSP using NetworkX

```
import matplotlib.pyplot as plt
import networkx as nx
import networkx.algorithms.approximation as nx_app
import math

plt.figure(figsize=(10, 7))

G = nx.random_geometric_graph(20, radius=0.4, seed=4)
pos = nx.get_node_attributes(G, "pos")

pos[0] = (0.5, 0.5)

H = G.copy()

for i in range(len(pos)):
    for j in range(i + 1, len(pos)):
        dist = math.hypot(pos[i][0] - pos[j][0], pos[i][1] - pos[j][1])
        dist = dist
        G.add_edge(i, j, weight=dist)

cycle = nx_app.christofides(G, weight="weight")
edge_list = list(nx.utils.pairwise(cycle))

nx.draw_networkx_edges(H, pos, edge_color="blue", width=0.5)

nx.draw_networkx(
    G,
    pos,
    with_labels=True,
    edgelist=edge_list,
    edge_color="red",
    node_size=200,
    width=3,
)

print("The route of the salesman is:", cycle)
plt.show()
```

Create a random geometric graph with 20 nodes.

Set (0,0) as the home city/depot.

Create an independent shallow copy of the graph and attributes.

Calculate the distances between the nodes as the edge's weight.

Solve TSP using Christofides algorithm.

Highlight the closest edges on each node only.

Draw the route.

Print the route.

Figure A.1 shows the solution for this TSP.

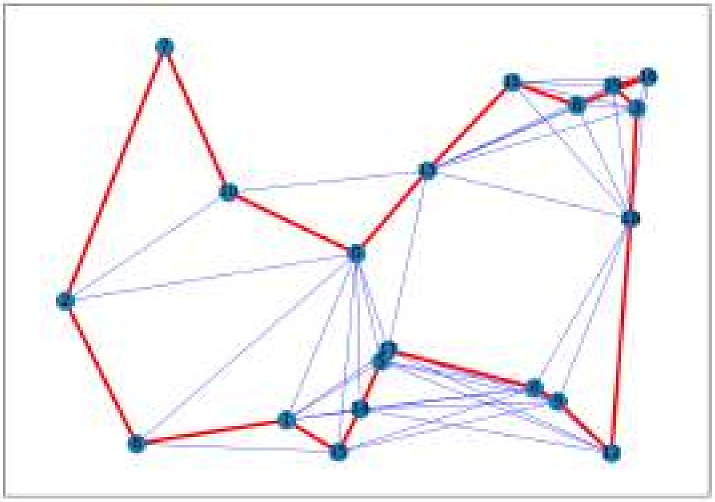

Figure A.1 Solving TSP using Christofides algorithm implemented in NetworkX. The found route is 0, 10, 7, 2, 6, 1, 15, 14, 5, 17, 4, 9, 12, 18, 3, 19, 16, 8, 11, 13, 0.

NetworkX supports a variety of graph search algorithms and can perform network analyses using packages within the geospatial Python ecosystem.

A.3.2 OSMnx

OSMnx is a Python library developed to ease the process of retrieving and manipulating data from OpenStreetMap (OSM). It offers the ability to download the data (filtered) from OSM and returns the network as a NetworkX graph data structure. It is a free and open source geographic data for the world.

You can install OSMnx with conda:

```
$ conda config --prepend channels conda-forge
$ conda create -n ox --strict-channel-priority osmnx
$ conda activate ox
```

OSMnx can be used to convert a text descriptor of a place into a NetworkX graph. Let's use Times Square in New York City as an example in the following continuation of listing A.2.

```
import osmnx as ox

place_name = "Times Square, NY"

graph = ox.graph_from_address(place_name, network_type='drive')

ox.plot_graph(graph,figsize=(10,10))
```

Name of the place or point of interest

NetworkX graph of the named place

Plot the graph.

Figure A.2 shows the graph of Times Square based on driving mode.

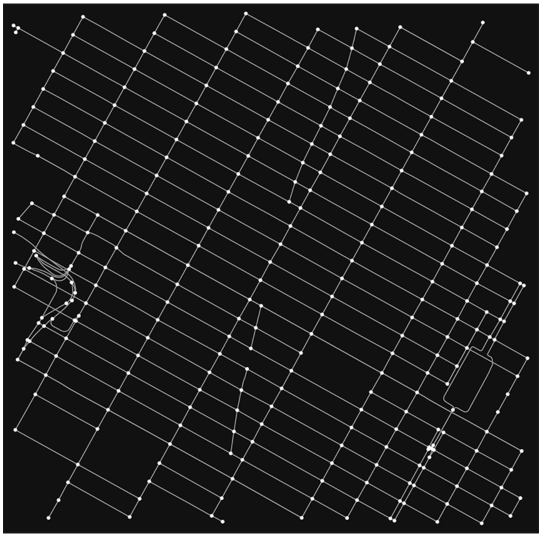

Figure A.2 Times Square graph with drivable streets

`network_type` allows you to select the type of street network based on the mobility mode: `all_private`, `all`, `bike`, `drive`, `drive_service`, or `walk`. You can highlight all the one-way edges in the Times Square street network using these two lines of code:

```
ec = ['y' if data['oneway'] else 'w' for u, v, key, data in graph.
edges(keys=True, data=True)]
fig, ax = ox.plot_graph(graph, figsize=(10,10), node_size=0, edge_color=ec,
edge_linewidth=1.5, edge_alpha=0.7)
```

Various properties of the graph can be examined, such as the graph type, edge (road) types, CRS projection, etc. For example, you can print the graph type using `type (graph)` and extract the nodes and edges of the graph as separate structures as follows:

```
nodes, edges = o.graph_to_gdfs(graph)
nodes.head(5)
```

We can further drill down to examine each individual node or edge.

```
list(graph.nodes(data=True))[1]
list(graph.edges(data=True))[0]
```

You can also retrieve the street types for the graph:

```
print(edges['highway'].value_counts())
```

Running the preceding code line gives the following statistics about the Times Square road network:

```
secondary                     236
residential                   120
primary                        83
unclassified                   16
motorway_link                  12
```

```
tertiary                          10
motorway                           7
living_street                      3
[unclassified, residential]        1
[motorway_link, primary]           1
trunk                              1
```

More statistics can be generated using `osmnx.basic_stats(graph)`.

GeoDataFrames can be easily converted to MultiDiGraphs by using `osmnx.graph_from_gdfs` as follows:

```
new_graph = ox.graph_from_gdfs(nodes,edges)
ox.plot_graph(new_graph,figsize=(10,10))
```

This results in the same road network shown in figure A.2. You can also save the street network in different formats as follows:

Save the street network as a PNG.

Save the graph to disk as an .osm XML file.

```
ox.plot_graph(graph, figsize=(10,10), show=False, save=True, close=True,
filepath='./data/TimesSquare.png')
ox.plot_graph(graph, figsize=(10,10), show=False, save=True, close=True,
filepath='./data/TimesSquare.svg')
ox.save_graph_xml(graph, filepath='./data/TimesSquare.osm')
ox.save_graph_geopackage(graph, filepath='./data/TimesSquare.gpkg')
ox.save_graphml(graph, filepath='./data/TimesSquare.graphml')
ox.save_graph_shapefile(graph, filepath='./data/TimesSquare')
```

Save the street network as an SVG.

Save the street network as a GraphML file for OSMnx, NetworkX, or Gephi.

Save the graph as a shapefile.

Save the street network as a GeoPackage file for GIS.

A.3.3 GeoPandas

GeoPandas is an extension to Pandas that handles geospatial data by extending the datatypes of Pandas and the ability to query and manipulate spatial data. It provides tools for reading, writing, and manipulating geospatial data, as well as for visualizing and mapping data on a map. You can install GeoPandas using pip or conda as follows:

```
$conda install geopandas or $pip install geopandas
```

GeoPandas can handle different geospatial data formats such as shapefiles (.shp), CSV (comma separated values), GeoJSON, ESRI JSON, GeoPackage (.gpkg), GML, GPX (GPS exchange format), and KML (Keyhole Markup Language). For example, let's assume we want to read Ontario's health region data based on a shapefile that can be downloaded from the Ontario data catalogue included in the book's GitHub repo (in the appendix B data folder). A shapefile is a popular geospatial data format for storing vector data (such as points, lines, and polygons). It is a widely used format for storing GIS data, and it's supported by many GIS software packages, including ArcGIS and QGIS. A shapefile is actually a collection of several files with different extensions, including the following:

- *.shp*—The main file, which contains the geospatial data (points, lines, or polygons)
- *.shx*—The index file, which allows for faster access to the data in the .shp file
- *.dbf*—The attribute file, which contains the attribute data (non-geographic information) for each feature in the .shp file
- *.prj*—The projection file, which defines the coordinate system and projection information for the data in the .shp file
- *.sbx*—A spatial index of the features

The following continuation of listing A.2 shows how to read this geospatial data from https://data.ontario.ca/dataset/ontario-s-health-region-geographic-data, which is stored in the book's GitHub repo:

```
import geopandas as gpd
import requests
import os

base_url = "https://raw.githubusercontent.com/Optimization-Algorithms-
➥Book/Code-Listings/05766c64c5e83dcd6788cc4415b462e2f82e0ccf/
➥Appendix%20B/data/OntarioHealth/"

files = ["Ontario_Health_Regions.shp", "Ontario_Health_Regions.shx",
➥ "Ontario_Health_Regions.dbf", "Ontario_Health_Regions.prj"]

for file in files:
    response = requests.get(base_url + file)
    with open(file, 'wb') as f:
        f.write(response.content)

ontario =  gpd.read_file("Ontario_Health_Regions.shp")

for file in files:
    os.remove(file)

print(ontario.head())
```

Files associated with the shapefile.

Define the base URL for the raw files.

Temporarily download the files from the specified URL.

Cleanup/remove the downloaded files.

Read the shapefile with geopandas.

Print the first n rows.

The complete version of listing A.2 is available in the book's GitHub repo.

A.3.4 contextily

contextily is a Python library for adding contextual basemaps to plots created with libraries such as Matplotlib, Plotly, and others. For example, contextily can be used to add context when we render the Ontario health region data, as follows:

```
#!pip install contextily
import contextily as ctx
ax=ontario.plot(cmap='jet', edgecolor='black', column='REGION', alpha=0.5,
➥ legend=True, figsize=(10,10))
ax.set_title("EPSG:4326, WGS 84")
ctx.add_basemap(ax, source=ctx.providers.OpenStreetMap.Mapnik,
➥ crs=ontario.crs.to_string())
```

contextily supports several different sources for basemaps, including the following commonly used sources:

- *OpenStreetMap (OSM)*—This is the default source for contextily. It is a free and open source map service that provides a variety of different styles, including the default Mapnik style and others such as Humanitarian and Cycle.
- *Stamen*—This source provides a variety of different map styles, including Toner, Terrain, and Watercolor.
- *Mapbox*—This source provides a variety of different map styles, including Streets, Outdoors, and Satellite. It requires an API key to use.
- *MapQuest*—This source provides a variety of different map styles, including OSM and Aerial. It requires an API key to use.
- *Here*—This source provides a variety of different map styles, including Normal Day and Normal Night. It requires an API key to use.
- *Google Maps*—This source provides a variety of different map styles, including Roadmap, Satellite, and Terrain. It requires an API key to use.

A.3.5 Folium

Folium is a library for creating interactive maps in Python using the Leaflet.js library. It provides tools for reading, writing, and manipulating geospatial data, as well as for visualizing and mapping data on a map. Folium can be used to create static or dynamic maps, as well as to customize the appearance and behavior of the map. The following continuation of listing A.2 shows how to use Folium to visualize Ontario Health regions on a map:

```
#!pip install folium
import folium

ontario = ontario.to_crs(epsg=4326)

m = folium.Map(location=[43.67621,-79.40530],zoom_start=7,
➥ tiles='cartodbpositron', scrollWheelZoom=False, dragging=True)

for index, row in ontario.iterrows():
    sim_geo = gpd.GeoSeries(row['geometry']).simplify(tolerance=0.001)
    geo_j = sim_geo.to_json()
    geo_j = folium.GeoJson(data=geo_j,
➥ name=row['REGION'],style_function=lambda x: {'fillColor': 'black'})
    folium.Popup(row['REGION']).add_to(geo_j)
    geo_j.add_to(m)

m
```

Transform geometries to a new coordinate reference system (CRS).

Set starting location, initial zoom, and base layer source.

Simplify each region's polygon, as intricate details are unnecessary.

Render the map.

The "Listing A.2_Graph_libraries.ipynb" notebook available in the book's GitHub repo provides examples of different ways of visualizing geospatial data, such as chloropleth map, cartogram map, bubble map, hexagonal binning, heat map, and cluster map.

A.3.6 Other libraries and tools

The following is a non-exhaustive list of other useful libraries and tools for working on geospatial data, graphs, and networks:

- Pyrosm—This is another Python library for reading OpenStreetMap from Protocolbuffer Binary Format files (*.osm.pbf). It can be used to download and read OpenStreetMap data, extract features such as roads, buildings, and points of interest, and analyze and visualize the data. The main difference between Pyrosm and OSMnx is that OSMnx reads the data using an OverPass API, whereas Pyrosm reads the data from local OSM data dumps that are downloaded from the Protocolbuffer Binary Format file (*.osm.pbf) data providers (Geofabrik, BBBike) and converts it into GeoPandas GeoDataFrames. This makes it possible to parse OSM data faster and makes it more feasible to extract data covering large regions.
- Pandana—This is a Python library for network analysis that uses contraction hierarchies to calculate super-fast travel accessibility metrics and shortest paths.
- GeoPy—This is a Python client for several popular geocoding web services.
- Graphviz—This is a library for creating visualizations of graphs and tree structures in Python. It provides tools for defining the structure of a graph, as well as for rendering the graph in various formats, such as PNG, PDF, and SVG. Graphviz is a useful tool for visualizing algorithms that operate on graphs, such as graph search algorithms and graph traversal algorithms.
- Gephi—This is a tool for visualizing and analyzing graphs and networks. It provides a graphical user interface for defining and customizing the appearance of graphs and diagrams, as well as for visualizing algorithms and data structures. Gephi can be used to visualize algorithms that operate on graph data, such as graph search algorithms and shortest path algorithms.
- Cytoscape—This is an open-source software platform for visualizing complex networks and integrating these with any type of attribute data.
- ipyleaflet—This is an interactive widgets library that is based on ipywidgets. ipywidgets, also known as jupyter-widgets or simply widgets, are interactive HTML widgets for Jupyter notebooks and the IPython kernel. Ipyleaflet brings mapping capabilities to the notebook and JupyterLab.
- hvPlot— This is a Python library that provides a high-level plotting API built on top of the HoloViews library. It can be used with GeoPandas to create interactive visualizations of geospatial data.
- mplleaflet—This is another `leaflet`-based library, but it plays nicely with `matplotlib`.
- Cartopy—Cartopy is a library for creating maps and geospatial plots in Python.
- geoplotlib—geoplotlib is a library for creating maps and visualizations in Python. It provides tools for styling and customizing map elements, as well as for overlaying data on top of maps. geoplotlib can be used to create static or interactive maps and supports a variety of map projections and coordinate systems.
- Shapely—This is an open source Python library for performing geometric operations on objects in the Cartesian plane.

- deck.gl—This is an open source JavaScript library for WebGL-powered large dataset visualization.
- kepler.gl—This is a powerful open source geospatial analysis tool for large-scale data sets.

A.4 Metaheuristics optimization libraries

There are several libraries in Python that provide implementations of different metaheuristic optimization algorithms. The following subsections cover some commonly used libraries.

A.4.1 PySwarms

PySwarms is a library for implementing swarm intelligence algorithms in Python. It provides tools for defining, training, and evaluating swarm intelligence models, as well as for visualizing the optimization process. PySwarms supports a variety of swarm intelligence algorithms, including particle swarm optimization (PSO) and ant colony optimization (ACO). The next listing shows the steps for solving a function optimization problem using PSO implemented in PySwarms.

Listing A.3 Solving function optimization using PSO implemented in PySwarms

```
#!pip install pyswarms
import pyswarms as ps
from pyswarms.utils.functions import single_obj as fx
from pyswarms.utils.plotters import plot_cost_history, plot_contour,
➥ plot_surface
from pyswarms.utils.plotters.formatters import Mesher, Designer
import matplotlib.pyplot as plt
from IPython.display import Image

options = {'c1':0.5, 'c2':0.3, 'w':0.9}
optimizer = ps.single.GlobalBestPSO(n_particles=50, dimensions=2,
➥ options=options)

optimizer.optimize(fx.sphere, iters=100)

plot_cost_history(optimizer.cost_history)
plt.show()

m = Mesher(func=fx.sphere, limits=[(-1,1), (-1,1)])
d = Designer(limits=[(-1,1), (-1,1), (-0.1,1)], label=['x-axis', 'y-axis',
➥ 'z-axis'])

animation = plot_contour(pos_history=optimizer.pos_history, mesher=m,
➥ designer=d, mark=(0,0))
animation.save('solution.gif', writer='imagemagick', fps=10)
Image(url='solution.gif')
```

Import the Image class to enable the display of images within the notebook environment.

Set up PSO as an optimizer with 50 particles and predefined parameters.

Solve the function optimization problem using PSO.

Set up the sphere unimodal function to be optimized using PSO, and set up the number of iterations.

Plot the cost.

Plot the sphere function's mesh for better plots.

Adjust the figure limits.

Generate animation for the solution history on a contour.

Render the animation.

A.4.2 Scikit-opt

Scikit-opt is an optimization library that provides a simple and flexible interface for defining and running optimization problems with various metaheuristics, such as genetic algorithms, particle swarm optimization, simulated annealing, ant colony algorithm, immune algorithm, and artificial fish swarm algorithm. Scikit-opt can be used to solve both continuous and discrete problems. The following continuation of listing A.3 shows the steps for solving a function optimization problem using scikit-opt:

```
#!pip install scikit-opt
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sko.SA import SA

obj_func = lambda x: np.sin(x[0]) * np.cos(x[1])

sa = SA(func=obj_func, x0=np.array([-3, -3]), T_max=1, T_min=1e-9, L=300,
➥ max_stay_counter=150)
best_x, best_y = sa.run()
print('best_x:', best_x, 'best_y', best_y)

plt.plot(pd.DataFrame(sa.best_y_history).cummin(axis=0))
plt.show()
```

Define a multimodal function.

Solve the problem using simulated annealing (SA).

Print the result.

Let's consider the TSP instance shown in figure A.3. In this TSP, a travelling salesman must visit 20 major US cities starting from a specific city.

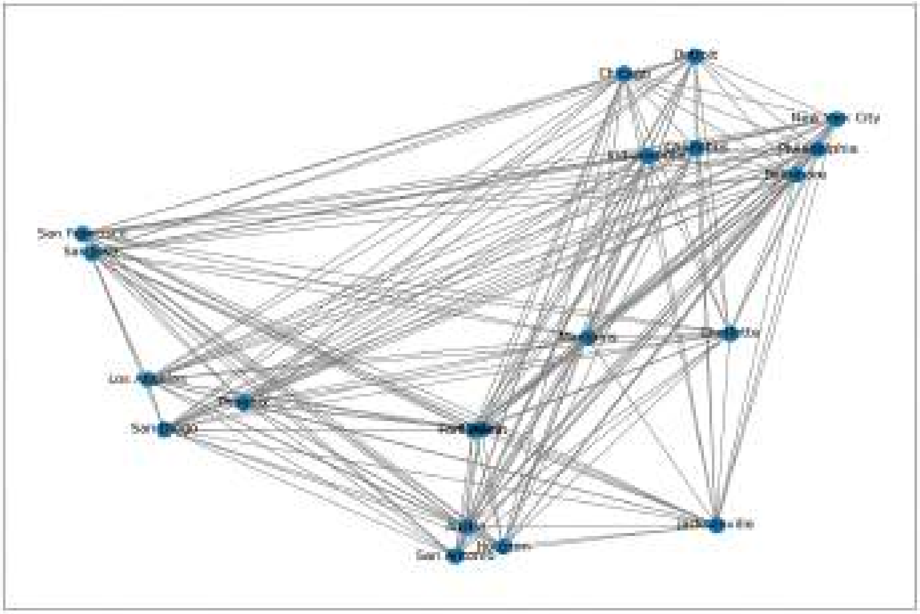

Figure A.3 Travelling salesman problem (TSP) for 20 major US cities

The following continuation of listing A.3 shows the steps for solving this problem using scikit-opt:

```
import numpy as np
import matplotlib.pyplot as plt
from sko.PSO import PSO_TSP

num_points = len(city_names)
points_coordinate = city_names
pairwise_distances = distances
```

Define the TSP problem.

```
def cal_total_distance(routine):
    num_points, = routine.shape
    return sum([pairwise_distances[routine[i % num_points], routine[(i + 1)
➥ % num_points]] for i in range(num_points)])
```

Calculate the total distance.

```
pso_tsp = PSO_TSP(func=cal_total_distance, n_dim=num_points, size_pop=200,
➥ max_iter=800, w=0.8, c1=0.1, c2=0.1)
best_points, best_distance = pso_tsp.run()
best_points_ = np.concatenate([best_points, [best_points[0]]])
```

Solve the problem using PSO.

```
print('best_distance', best_distance)
print('route', best_points_)
```

Print the solution.

A.4.3 NetworkX

NetworkX, introduced in the previous section, provides approximations of graph properties and heuristic methods for optimization. An example of these heuristics algorithms is simulated annealing. The following continuation of listing A.3 shows the steps for solving TSP using simulated annealing implemented in NetworkX:

```
#!pip install networkx
import matplotlib.pyplot as plt
import networkx as nx
from networkx.algorithms import approximation as approx

G=nx.Graph()
```

Create a graph.

```
for i in range(len(city_names)):
    for j in range(1,len(city_names)):
        G.add_weighted_edges_from({(city_names[i], city_names[j],
➥ distances[i][j])})
        G.remove_edges_from(nx.selfloop_edges(G))
```

Add weighted edges to the graph, and remove selfloop edges.

```
pos = nx.spring_layout(G)
```

Define pos as a dictionary of positions using the Fruchterman-Reingold force-directed algorithm.

```
cycle = approx.simulated_annealing_tsp(G, "greedy", source=city_names[0])
edge_list = list(nx.utils.pairwise(cycle))
cost = sum(G[n][nbr]["weight"] for n, nbr in nx.utils.pairwise(cycle))
```

Solve TSP using simulated annealing.

```
print("The route of the salesman is:", cycle, "with cost of ", cost)
```

Print the route and the cost.

A.4.4 Distributed evolutionary algorithms in Python (DEAP)

DEAP is a library for implementing genetic algorithms in Python. It provides tools for defining, training, and evaluating genetic algorithm models, as well as for visualizing the optimization process. DEAP supports a variety of genetic algorithm techniques, including selection, crossover, and mutation. The following continuation of listing A.3 shows the steps for solving TSP using simulated annealing implemented in DEAP:

```
#!pip install deap
from deap import base, creator, tools, algorithms
import random
import numpy as np

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox = base.Toolbox()
toolbox.register("permutation", random.sample, range(len(city_names)),
➥ len(city_names))
toolbox.register("individual", tools.initIterate, creator.Individual,
➥ toolbox.permutation)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

def eval_tsp(individual):
    total_distance = 0
    for i in range(len(individual)):
        city_1 = individual[i]
        city_2 = individual[(i + 1) % len(individual)]
        total_distance += distances[city_1][city_2]
    return total_distance,

toolbox.register("evaluate", eval_tsp)
toolbox.register("mate", tools.cxOrdered)
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=0.05)
toolbox.register("select", tools.selTournament, tournsize=3)

pop = toolbox.population(n=50)
hof = tools.HallOfFame(1)
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("avg", np.mean)
stats.register("min", np.min)
stats.register("max", np.max)

pop, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=50,
➥ stats=stats, halloffame=hof,verbose=True)

best_individual = hof[0]
print("Best solution:")
print("  - Fitness: ", eval_tsp(best_individual))
print("  - Route: ", [city_names[i] for i in best_individual])
```

Create a fitness function that minimizes the total distance.

Create the genetic operator functions.

Calculate the route length.

Set an ordered crossover.

Select the best individual among three randomly chosen individuals.

Set the evaluation function.

Set the shuffle mutation with probability 0.05.

Set the population size.

Set the hall of fame to select the best individual that ever lived in the population during the evolution.

Solve the problem using a simple evolutionary algorithm.

Print the solution.

DEAP includes several built-in algorithms such as genetic algorithm (GA), evolutionary strategy (ES), genetic programming (GA), estimation of distribution algorithms (EDA), and particle swarm optimization (PSO).

A.4.5 OR-Tools

As previously mentioned, OR-Tools (Operations Research Tools) is an open source library for optimization and constraint programming developed by Google. The following continuation of listing A.3 shows the steps for solving TSP using tabu search implemented in OR-Tools:

```
#!pip install --upgrade --user ortools
import numpy as np
import matplotlib.pyplot as plt
from ortools.constraint_solver import pywrapcp
from ortools.constraint_solver import routing_enums_pb2

distances2=np.asarray(distances, dtype = 'int')
```

Convert a float array into an integer array for OR_Tools.

```
data = {}
data['distance_matrix'] = distances
data['num_vehicles'] = 1
data['depot'] = 0
```

Define the problem data.

```
manager = pywrapcp.RoutingIndexManager(len(data['distance_matrix']),
➥ data['num_vehicles'], data['depot'])
routing = pywrapcp.RoutingModel(manager)
```

Define a constraint programming solver.

```
def distance_callback(from_index, to_index):
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    return data['distance_matrix'][from_node][to_node]
```

Get the distance between the cities.

```
transit_callback_index = routing.RegisterTransitCallback(distance_callback)

routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

search_parameters = pywrapcp.DefaultRoutingSearchParameters()
search_parameters.local_search_metaheuristic = (
    routing_enums_pb2.LocalSearchMetaheuristic.TABU_SEARCH)
search_parameters.time_limit.seconds = 30
search_parameters.log_search = True
```

Set up tabu search as a local search metaheuristic.

Print the solution.

```
def print_solution(manager, routing, solution):
    print('Objective: {} meters'.format(solution.ObjectiveValue()))
    index = routing.Start(0)
    plan_output = 'Route for vehicle 0:\n'
    route_distance = 0
    while not routing.IsEnd(index):
        plan_output += ' {} ->'.format(manager.IndexToNode(index))
        previous_index = index
        index = solution.Value(routing.NextVar(index))
        route_distance += routing.GetArcCostForVehicle(previous_index,
➥ index, 0)
```

```
    plan_output += ' {}\n'.format(manager.IndexToNode(index))
    print(plan_output)
    plan_output += 'Route distance: {}meters\n'.format(route_distance)

solution = routing.SolveWithParameters(search_parameters)
if solution:
    print_solution(manager, routing, solution)
```

The OR-Tools library also implements some metaheuristics but not as many as dedicated metaheuristics frameworks like DEAP. Examples include simulated annealing (SA), tabu search (TS), and guided local search (GLS).

A.4.6 Other libraries

The following non-exhaustive list identifies other useful libraries and tools for solving optimization problems using metaheuristics:

- simanneal—This is an open source Python module for simulated annealing. Listing A.3 shows the steps for solving TSP using simulated annealing implemented in simanneal.
- Non-dominated Sorting Genetic Algorithm (NSGA-II)—This is a solid multi-objective algorithm, widely used in many real-world applications. The algorithm is designed to find a set of solutions, called the Pareto front, which represents the trade-off between multiple conflicting objectives. NSGA-II implementations are available in pymoo and DEAP.
- Python genetic algorithm (PyGAD)—This is a library for implementing genetic algorithms and differential evolution in Python. It provides tools for defining, training, and evaluating genetic algorithm models, as well as for visualizing the optimization process. PyGAD supports a variety of genetic algorithm techniques, including selection, crossover, and mutation.
- Library for Evolutionary Algorithms in Python (LEAP)—This is a general-purpose evolutionary algorithm (EA) package that is simple and easy to use. It provides a high-level abstraction for defining and running EAs.
- Pyevolve—This is a Python library that provides a simple and flexible API for implementing and running genetic algorithms.
- Genetic algorithms made easy (EasyGA)—This is another Python library for genetic algorithms with several built-in genetic operators, such as selection, crossover, and mutation. EasyGA and Pyevolve are simple libraries with less functionality and fewer predefined problems than other libraries such as DEAP and Pymoo.
- MEAPLY—This is a Python library for population metaheuristic algorithms.
- swarmlib—This library implements several swarm optimization algorithms and visualizes their (intermediate) solutions.
- Hive—This is a swarm-based optimization algorithm based on the intelligent foraging behavior of honeybees. Hive implements the artificial bee colony (ABC) algorithm.

- Pants—This is a Python3 implementation of the ant colony optimization (ACO) metaheuristics.
- mlrose—This library provides implementations of hill climb, random hill climb, simulated annealing, and genetic algorithm.
- Mixed integer distributed ant colony optimization (MIDACO)—This is a numerical high-performance library based on ACO for solving single- and multi-objective optimization problems.
- cuOpt: This is a Python SDK and cloud service provided by NVIDIA that provides access to GPU-accelerated logistics solvers relying on metaheuristics to calculate complex vehicle routing problems with a wide range of constraints.

"Listing A.3_Metaheuristics_libraries.ipynb," included in the book's GitHub repo, shows how to use some of these metaheuristics libraries.

A.5 Machine learning libraries

Machine learning can be used to solve discrete optimization problems where an ML model is trained to output solutions directly from the input, usually represented as a graph. To train the model, the problem graph needs to be turned first into a feature vector using graph embedding/representation learning methods. Several Python libraries can be used for graph embedding and for solving optimization problems. The following subsections shed some light on the commonly used libraries.

A.5.1 node2vec

node2vec is an algorithmic framework for learning low-dimensional representations of nodes in a graph. Given any graph, it can learn continuous feature representations for the nodes, which can then be used for various downstream ML tasks.

To install node2vec, use the following command:

```
$ pip install node2vec
```

Alternatively, you can install node2vec by cloning the repository from GitHub and running the setup.py file:

```
$ git clone https://github.com/aditya-grover/node2vec
$ cd node2vec
$ pip install -e .
```

The following code illustrates how to use node2vec to learn low-dimensional representations of nodes in a graph based on Zachary's karate club dataset. This is a graph-based dataset commonly used in network analysis and graph-based machine learning algorithms. It represents a social network that contains information about the relationships between 34 individuals in a karate club. It was created and first described by Wayne W. Zachary in his paper "An Information Flow Model for Conflict and Fission in Small Groups" in 1977, and it has since become a popular benchmark dataset for evaluating graph-based machine learning algorithms.

Listing A.4 A node2vec example

```
import networkx as nx
from node2vec import Node2Vec

G = nx.karate_club_graph()          Create a sample graph.

node2vec = Node2Vec(G, dimensions=64, walk_length=30, num_walks=200,
➥ workers=4)          Create an instance of the Node2Vec class.      Learn the
                                                                         representations.
model = node2vec.fit(window=10, min_count=1, batch_words=4)

representations_all = model.wv.vectors          Get the representations of all nodes.

representations_specific = model.wv['1']          Get the representations of a specific node.

print(representations_specific)          Print the representations of a specific node.
```

You can visualize the generated low-dimensional representations using a dimensionality reduction technique such as t-SNE to project the representations onto a 2D or 3D space, and then use a visualization library such as Matplotlib to plot the nodes in this space. t-distributed stochastic neighbor embedding (t-SNE) is a statistical method for visualizing high-dimensional data by giving each data point a location in a 2D or 3D map. Here is an example:

```
from sklearn.manifold import TSNE          Import the required libraries.
import matplotlib.pyplot as plt

Perform t-SNE dimensionality reduction.
tsne = TSNE(n_components=2, learning_rate='auto', init='random',
➥ perplexity=3)
reduced_representations = tsne.fit_transform(representations_all)

Plot the nodes.
plt.scatter(reduced_representations[:, 0], reduced_representations[:, 1])
plt.show()
```

Running this code gives the visualization in figure A.4.

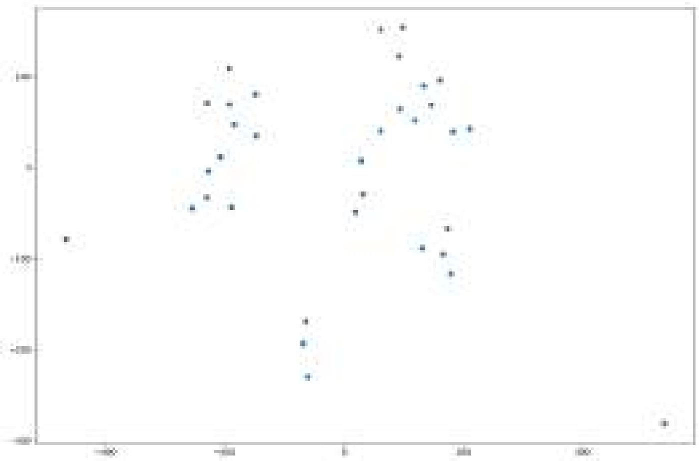

Figure A.4 t-SNE-based visualization for the low-dimensional representations generated by node2vec

A.5.2 DeepWalk

DeepWalk is a random walk-based method for graph embedding based on representation learning. For this example, we'll use a DeepWalk module provided by the Karate Club library. This library is an unsupervised machine learning extension for NetworkX. To use DeepWalk, you can install Karate Club as follows:

```
$ pip install karateclub
```

The following continuation of listing A.4 illustrates how to use DeepWalk:

```
from karateclub import DeepWalk, Node2Vec
from sklearn.decomposition import PCA
import networkx as nx
import matplotlib.pyplot as plt
G=nx.karate_club_graph()
```

Import DeepWalk. node2vec is also available.

Import principal component analysis (PCA).

Create the Karate Club graph.

```
model=DeepWalk(dimensions=128, walk_length=100)
model.fit(G)
```

Define the DeepWalk mode, and fit the graph.

```
embedding=model.get_embedding()
```

Graph embedding.

```
officer=[]
mr=[]
for i in G.nodes:
  t=G.nodes[i]['club']
  officer.append(True if t=='Officer' else False)
  mr.append(False if t=='Officer' else True)
```

Retrieve the club membership attribute of each node.

```
nodes=list(range(len(G)))
X=embedding[nodes]

pca=PCA(n_components=2)
pca_out=pca.fit_transform(X)
```

Define the DeepWalk mode, and fit the graph.

```
plt.figure(figsize=(15, 10))
plt.scatter(pca_out[:,0][officer],pca_out[:,1][officer])
plt.scatter(pca_out[:,0][mr],pca_out[:,1][mr])
plt.show()
```

Visualize the embedding.

A.5.3 PyG

PyG (PyTorch Geometric) is a library for implementing graph neural networks in Python using the PyTorch deep learning framework. It provides tools for defining, training, and evaluating graph neural network (GNN) models, as well as for visualizing the optimization process. PyG supports a variety of GNN architectures, including graph convolutional network (GCN) and graph attention networks (GATs).

You can install PyG as follows:

```
$pip install torch-scatter torch-sparse torch-cluster torch-spline-conv
torch-geometric -f https://data.pyg.org/whl/torch-1.13.0+cpu.html
```

The following continuation of listing A.4 shows how you can use PyG to generate Karate Club graph embedding using GCN:

```
import networkx as nx
import matplotlib.pyplot as plt
import torch
from torch_geometric.datasets import KarateClub
from torch_geometric.utils import to_networkx
from torch.nn import Linear
from torch_geometric.nn import GCNConv

dataset = KarateClub()
data = dataset[0]

class GCN(torch.nn.Module):
    def __init__(self):
        super().__init__()
        torch.manual_seed(1234)
        self.conv1 = GCNConv(dataset.num_features, 4)
        self.conv2 = GCNConv(4, 4)
        self.conv3 = GCNConv(4, 2)
        self.classifier = Linear(2, dataset.num_classes)

    def forward(self, x, edge_index):
        h = self.conv1(x, edge_index)
        h = h.tanh()
        h = self.conv2(h, edge_index)
        h = h.tanh()
        h = self.conv3(h, edge_index)
        h = h.tanh()
        out = self.classifier(h)

        return out, h

model = GCN()

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
optimizer.zero_grad()

for epoch in range(401):
    out, h = model(data.x, data.edge_index)

    loss = criterion(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()

h = h.detach().cpu().numpy()

plt.figure(figsize=(15, 10))
plt.scatter(h[:, 0], h[:, 1], s=140, c=data.y, cmap="Set2")
```

Use the Karate Club dataset.

Graph Convolutional Network class

Apply a final (linear) classifier.

Final GNN embedding space

Define the model.

Define the loss criterion.

Define an optimizer and clear gradient.

Perform a single forward pass.

Compute the loss solely based on the training nodes.

Derive the gradients.

Update the parameters based on the gradients.

Convert 'h' from tenser format to the numpy format for visualizing.

Visualize the embedding.

PyG is a well-supported library that provides several features, such as common benchmark datasets (e.g., KarateClub, CoraFull, Amazon, Reddit, Actor), data handling of graphs, mini-batches, data transforms, and learning methods on graphs (e.g., node2vec, MLP, GCN, GAT, GraphSAGE, GraphUNet, DeepGCNLayer, GroupAddRev, and MetaLayer).

A.5.4 OpenAI Gym

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It gives you access to variety of environments, such as

- *Classic control*—A variety of classic control tasks
- *Box2d*—A 2D physics engine
- *MuJoCo*—A physics engine that can do detailed, efficient simulations with contacts
- *Algorithmic*—A variety of algorithmic tasks, such as learning to copy a sequence
- *Atari*—A variety of Atari video games
- *gym-maze*—A simple 2D maze environment where an agent finds its way from the start position to the goal

Gymnasium is a maintained fork of OpenAI's Gym library. The following continuation of listing A.4 illustrates how to use OpenAI Gym:

```
#!pip install gym[all]
import gym
env = gym.make('MountainCar-v0')
```

Install all included environments.

Create an environment.

In this simple example, `MountainCar-v0` has discrete actions. You can also use `MountainCarCoutinous-V0`, which has continuous actions corresponding to the force with which the car is pushed. The complete code listing is available in the book's GitHub repo, in "Listing A.4_ML_libraries.ipynb".

A.5.5 Flow

Flow is a deep reinforcement learning framework for mixed autonomy traffic. It allows you to run deep RL-based control experiments for traffic microsimulation.

You can install Flow as follows:

```
$git clone https://github.com/flow-project/flow.git
$cd flow
$conda env create -f environment.yml
$conda activate flow
$python setup.py develop
```

Install Flow within the environment:

```
$pip install -e .
```

Flow enables studying complex, large-scale, and realistic multirobot control scenarios. It can be used to develop controllers that optimize the system-level velocity or other objectives in the presence of different types of vehicles, model noise, and road networks such as single-lane circular tracks, multi-lane circular tracks, figure-eight networks, loops with merge networks, and intersections.

A.5.6 Other libraries

The following is a non-exhaustive list of other useful ML libraries:

- Deep Graph Library (DGL)—A library for implementing graph neural networks in Python. It provides tools for defining, training, and evaluating GNN models, as well as for visualizing the optimization process. DGL supports a variety of GNN architectures, including GCN and GAT.
- Stanford Network Analysis Platform (SNAP)—A general-purpose, high-performance system for the analysis and manipulation of large, complex networks. SNAP includes a number of algorithms for network analysis, such as centrality measures, community detection, and graph generation. It is particularly well-suited for large-scale network analysis and is used in a variety of fields, including computer science, physics, biology, and social science.
- Spektral—An open-source Python library for graph neural networks (GNNs) built on top of TensorFlow and Keras. It is designed to make it easy to implement GNNs in research and production. It provides a high-level, user-friendly API for building GNNs, as well as a number of prebuilt layers and models for common tasks in graph deep learning. The library also includes utilities for loading and preprocessing graph data and for visualizing and evaluating the performance of GNNs.
- Jraph (pronounced "giraffe")—A lightweight library for working with graph neural networks that also provides lightweight data structure for working with graphs. You can easily work with this library to construct and visualize your graph.
- GraphNets—A library for implementing GNNs in Python. It provides tools for defining, training, and evaluating GNN models, as well as for visualizing the optimization process. GraphNets supports a variety of GNN architectures, including GCN and GAT.
- Stable-Baselines3 (SB3)—A set of reliable implementations of reinforcement learning algorithms in PyTorch.
- Vowpal Wabbit (VW)—An open source ML library specifically designed for large-scale online learning. Developed originally at Yahoo Research, and currently at Microsoft Research, it is widely used for solving reductions and contextual bandits problems.
- Ray RLlib—An open source library for RL, offering support for production-level, highly distributed multi-agent RL workloads while maintaining unified and simple APIs for a large variety of industry applications.
- TF-Agents—A library for developing RL algorithms in TensorFlow, which includes a collection of environments, algorithms, and tools for training RL agents.
- Keras-RL—A deep reinforcement learning library built on top of Keras. It provides an easy-to-use interface for developing and testing RL algorithms. Keras-RL supports a variety of RL algorithms such as deep Q-networks (DQN) and actor-critic methods. There are also built-in environments for testing RL algorithms.

- pyqlearning—A Python library to implement reinforcement learning and deep reinforcement learning, especially for Q-learning, deep Q-network, and multi-agent deep Q-network, which can be optimized by annealing models such as simulated annealing, adaptive simulated annealing, and quantum Monte Carlo method.
- Python Optimal Transport (POT)—An open source Python library providing several solvers for optimization problems related to optimal transport for signals, image processing, and machine learning.

The "Listing A.4_ML_libraries.ipynb" notebook available in the book's GitHub repo provides examples of how to install and use some of these libraries.

A.6 Projects

My course "ECE1724H: Bio-inspired Algorithms for Smart Mobility," at the University of Toronto, features a collection of exemplary projects available in the "AI for Smart Mobility (AI4SM)" publication hub (https://medium.com/ai4sm) with Python code implementations. These projects encompass a broad spectrum of optimization algorithms covered in this book and their practical applications in the smart mobility domain. Among the topics tackled are

- Employing large language models for reinforcement learning in intelligent driving
- Forecasting traffic flows
- Strategizing the placement of EV charging stations
- Optimizing food delivery services
- Predicting estimated times of arrival
- Improving the deployment of traffic sensors
- Customizing cycling routes
- Reorganizing urban fire districts
- Pricing strategies for ride-sharing
- Refining school bus routes
- Availability predictions for bikes
- Districting for efficient waste collection
- Enhancing bus stop layouts
- Implementing dynamic pricing for public transportation
- Optimizing student transportation and boarding
- Integrating hotel recommendations with route planning
- Determining ideal locations for public parcel lockers
- Predictive responses and accessibility scoring for healthcare facilities