

appendix C

Exercises and solutions

In this appendix, you will find a comprehensive set of exercises and their corresponding solutions, organized by chapter, to enhance your understanding and the application of the material presented in this book. These exercises are designed to reinforce the concepts, theories, and practical skills covered throughout the chapters.

C.1 Chapter 2: A deeper look at search and optimization

C.1.1 Exercises

1. Multiple choice: Choose the correct answer for each of the following questions.

1.1. _____ is the class of decision problems that can be solved by nondeterministic polynomial algorithms and whose solutions are hard to find but easy to verify.

- a P
- b NP
- c co-NP
- d NP-complete
- e NP-hard

1.2. Which of the following benchmark (toy) problems is not NP-complete?

- a Bin packing
- b Knapsack problem
- c Minimum spanning tree
- d Hamiltonian circuit
- e Vertex cover problem

1.3. _____ is the class of decision problems whose “No” answer can be verified in polynomial time.

- a P
- b NP
- c co-NP
- d NP-complete
- e NP-hard

1.4. Which of the following real-world problems is NP-hard?

- a Image matching
- b Single machine scheduling
- c Combinational equivalence checking
- d Capacitated vehicle routing problem (CVRP)
- e Container/truck loading

1.5. _____ is a theory that focuses on classifying computational problems according to their resource usage and relating these classes to each other.

- a Optimization complexity
- b Time complexity
- c Computational complexity
- d Operation research
- e Decision complexity

2. Describe the following search and optimization problems in terms of decision variable (univariate, bivariate, multivariate); objective functions (mono-objective, multi-objective, no objective function, or constraint-satisfaction problem); constraints; (hard constraints, soft constraints, both hard and soft constraints, unconstrained); and linearity (linear programming (LP), quadratic programming (QP), nonlinear programming (NLP)).

- a Minimize $y + \cos(x^2)$, $\sin(x) - x \times y$, and $1 / (x + y)^2$
- b Maximize $2 - e^{(1-x)}$ subject to $-3 \leq x < 10$
- c Maximize $3 \times x - y / 5$ subject to $-2 \leq x < 3$, $0 < y \leq 3$, and $x + y = 4$
- d The school districting problem consists of determining the groups of students attending each school of a school board located over a given territory in a way that maximizes the contiguity of school sectors, taking into consideration a number of hard constraints such as school capacity for each grade and class capacity. Walkability and keeping students in the same school from year to year are considered soft constraints in this problem.
- e The knapsack problem is an example of a combinatorial problem whose solution takes the form of a combination where the order doesn't matter. As illustrated in figure C.1, given a set of items, each with a utility and a weight, the task is to select a subset of items to maximize the total utility while ensuring that the total weight of the selected items does not exceed a predefined capacity. The decision to include or exclude each item is binary.

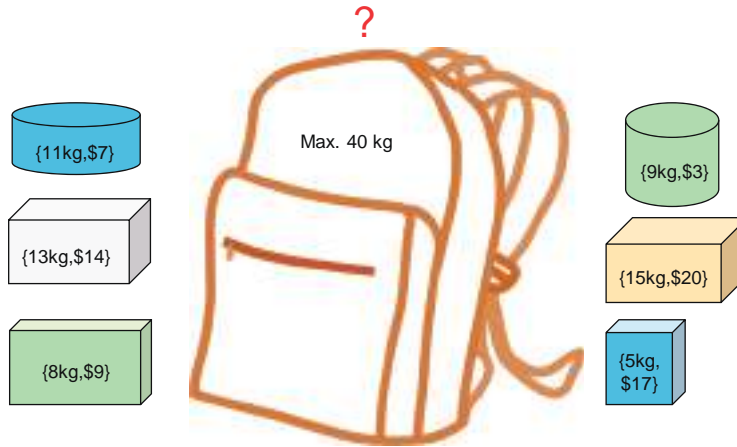


Figure C.1 Each item has a utility and a weight, and we want to maximize the utility of the contents of the knapsack. The problem is constrained by the capacity of the bag.

3. For the following optimization problems, state the type of the problem (design, planning, or control problem) based on the permissible time to solve the problem and the expected quality of the solutions. Suggest the appropriate algorithm required to handle the optimization problem (offline versus online).

- a Find the optimal wind park design where the number and types of wind turbines need to be chosen and placed based on the wind conditions and wind park area.
- b Find multiple vehicle routes starting and ending at different depots so that all customer demands are fulfilled.
- c Create a fitness assistant for runners and cyclists that seamlessly automates the multiple tasks involved in planning fitness activities. The planner will assess an athlete's current fitness level and individual training goals in order to create a fitness plan. The planner will also generate and recommend geographical routes that are both popular and customized to the user's goals, level, and scheduled time, thus reducing the challenges involved in the planning stage. The suggested fitness plans will continuously adapt based on each user's progress toward their fitness goals, thus keeping the athlete challenged and motivated.
- d Find a set of flights with departure and arrival times and aircraft assignments that maximize profits, given demand and revenues for every flight, route information distances, times, operating restrictions, aircraft characteristics and operating costs, and operational and managerial constraints.
- e Find the optimal schedule for delivery cargo bikes, semi and fully autonomous last-mile delivery trucks, self-driving delivery robots or delivery drones to maximize customer satisfaction and minimize delivery costs, taking into consideration the capacity of the vehicle, type of delivery service (a couple of days delivery, next-day delivery, or same-day delivery with some extra surcharge), delivery time, drop-off locations, and so on.

- f Plan on-demand responsive transit during pandemics to support the transportation of essential workers and essential trips to pharmacies and grocery stores for the general public, especially the elderly, taking into consideration store operating hours, capacity, and online delivery options.
 - g Find a collision-free path for a vehicle from a start position to a given goal position, amid a collection of obstacles, in such a way that minimizes the estimated time of arrival and the consumed energy.
 - h Develop a trip planner that minimizes total commute time, maximizes the average ratings of attractions, maximizes the duration spent at each of these attractions, and effectively minimizes idle time when someone visits a city.
 - i Find school bus loading patterns and schedules such that the number of routes is minimized, the total distance traveled by all buses is kept to a minimum, no bus is overloaded, and the time required to traverse any route does not exceed a maximum time policy.
 - j Minimize deadheading for shared mobility companies (minimize the miles driven with no passenger) or for delivery services providers (minimize the miles driven without cargo).
 - k Plan or replan transport corridors and city streets to accommodate more pedestrians, cyclists, and riders in shared transportation and fewer cars.
 - l Find the optimal placement for bus stops, traffic sensors, micro mobility stations, EV charging stations, air taxi takeoff and landing locations, walking routes, and cycling lanes for active mobility.
4. Modify listing 2.6 to define the animal feed mix problem data using Python dictionaries or to read the problem data from a CSV file.

C.1.2 Solutions

1. Multiple choice
 - 1.1. b) NP
 - 1.2. c) Minimum spanning tree
 - 1.3. c) co-NP
 - 1.4. d) Capacitated vehicle routing problem (CVRP)
 - 1.5. c) Computational complexity
2. Optimization problem description
 - a Bivariate, multi-objective, unconstrained, nonlinear programming
 - b Univariate, mono-objective, hard constraints, nonlinear programming
 - c Bivariate, mono-objective, hard constraints, linear programming
 - d Multivariate, mono-objective, both hard and soft constraints, linear programming (see Jacques A. Ferland and Gilles Gu  nette, “Decision support system for the school districting problem” [1])
 - e Bivariate, mono-objective, hard constraints, linear programming

3. Optimization problem and process

- a Design problem. Offline optimization.
- b Planning problem during the process of generating the route, and control problem during rerouting. Offline optimization during planning, and online optimization during rerouting.
- c Planning problem to generate the plans and control problem for adaptation. Offline optimization during the planning phase, and online during the adaptation phase.
- d Design problem to generate the flight schedule, and planning problem if adaptation is required, such as in the case of faulty aircraft or cancellation due to weather conditions. Offline optimization, and online optimization if adaptation is required.
- e Planning problem during path generation, and control problem during rerouting for adaptive motion planning. Online optimization.
- f Planning problem for scheduling, and control problem if rerouting is involved. Online optimization.
- g Planning problem to generate the plans, and control problem for adaptation. Online optimization.
- h Planning problem. Offline optimization.
- i Design problem. Offline optimization.
- j Planning problem. Online optimization.
- k Design problem. Offline optimization.
- l Design problem. Offline optimization.

4. The next listing shows the steps for defining the animal feed mix problem data using Python dictionaries or to read the problem data from a CSV file.

Listing C.1 Animal feed mix problem—defining data using dictionaries

```

import pandas as pd
from pulp import *

Ingredients = ["Limestone", "Corn", "Soybean meal"]

Price = {"Limestone": 10.0, "Corn": 30.5, "Soybean meal": 90.0,}
Calcium = {"Limestone": 0.38, "Corn": 0.001, "Soybean meal": 0.002,}
Protein = {"Limestone": 0.0, "Corn": 0.09, "Soybean meal": 0.50,}
Fiber = {"Limestone": 0.0, "Corn": 0.02, "Soybean meal": 0.08,}

model = LpProblem("Animal_Feed_Mix_Problem", LpMinimize)

ingredient_vars = LpVariable.dicts("Ingr", Ingredients, 0)

model += (lpSum([Price[i] * ingredient_vars[i] for i in
    ➔ Ingredients]), "Total Cost of Ingredients per kg",)

model += (lpSum([Calcium[i] * ingredient_vars[i] for i in Ingredients]) >=

```

Create a list of ingredients. →

Dictionary of calcium (kg/kg). →

Create a model. →

Add the objective function. →

Dictionary of protein (kg/kg). ←

Dictionary of unit cost (cents/kg). ←

Dictionary called 'ingredient_vars' is created to contain the referenced variables. ←

Add the five constraints.

```

➡ 0.008, "Minimum calcium",)
model += (lpSum([Calcium[i] * ingredient_vars[i] for i in Ingredients]) <=
➡ 0.012, "Maximum calcium",)
model += (lpSum([Protein[i] * ingredient_vars[i] for i in Ingredients])
➡ >=0.22, "Minimum protein",)
model += (lpSum([Fiber[i] * ingredient_vars[i] for i in Ingredients])
➡ <=0.05, "MMaximum fiber",)
model += lpSum([ingredient_vars[i] for i in Ingredients]) == 1,
➡ "Conservation"

```

model.solve()  Solve the problem using PuLP's choice of solver.

Print the results.

```

for v in model.variables():
    print(v.name, '=', round(v.varValue,2)*100, '%')

print('Total cost of the mixture per kg = ', round(value(model.objective),
➡ 2), '$')

```

We can also read the problem data from a CSV file as follows.

Read the CSV file.

Listing C.2 Animal feed mix problem—reading problem data from a CSV file

```
df = pd.read_csv('Blending_problem_data.csv')
```


Create a model.

```
data = df.to_dict()  Convert data frame to a dictionary.
```

```
model = LpProblem("Animal_Feed_Mix_Problem", LpMinimize)
```

Add the objective function.

```

ingredient_vars = LpVariable.dicts("Ingr", data.get('Ingredients'), 0)  A dictionary called 'ingredient_vars' is created to contain the referenced variables.

model += (lpSum([data.get('Price')[i] * ingredient_vars[i] for i in
➡ data.get('Ingredients')]), "Total Cost of Ingredients per kg",)

```

Add the five constraints.

```

model += lpSum([ingredient_vars[i] for i in data.get('Ingredients')]) == 1,
➡ "Conservation"
model += (lpSum([data.get('Calcium')[i] * ingredient_vars[i] for i in
➡ data.get('Ingredients')]) >= 0.008, "Minimum calcium",)
model += (lpSum([data.get('Calcium')[i] * ingredient_vars[i] for i in
➡ data.get('Ingredients')]) <= 0.012, "Maximum calcium",)
model += (lpSum([data.get('Protein')[i] * ingredient_vars[i] for i in
➡ data.get('Ingredients')]) >=0.22, "Minimum protein",)
model += (lpSum([data.get('Fiber')[i] * ingredient_vars[i] for i in
➡ data.get('Ingredients')]) <=0.05, "MMaximum fiber",)

```

model.solve()  Solve the problem using PuLP's choice of solver.

Print the results.

```

for v in model.variables():
    print(v.name, '=', round(v.varValue,2)*100, '%')

print('Total cost of the mixture per kg = ', round(value(model.objective),
➡ 2), '$')

```

Running listing C.2 produces the following results:

```
Ingr_Corn = 65.0 %  
Ingr_Limestone = 3.0 %  
Ingr_Soybean_meal = 32.0 %  
Total cost of the mixture per kg = 49.16 $
```

C.2 Chapter 3: Blind search algorithms

C.2.1 Exercises

1. Multiple choice and true/false: Choose the correct answer for each of the following questions.

1.1. Big O specifically describes the limiting behavior of a function (worst-case scenario) when the argument tends toward a particular value or infinity, usually in terms of simpler functions. What is the big O of this expression: $n\log(n) + \log(2n)$?

- a Linearithmic
- b Loglinear
- c Quasilinear
- d All of the above

1.2. Which blind search algorithm implements a stack operation for searching the states?

- a Breadth-first search (BFS)
- b Uniform-cost search (UCS)
- c Bidirectional search (BS)
- d Depth-first search (DFS)
- e None of the above

1.3. A tree is a connected graph with no circuits and no self-loops.

- a True
- b False

1.4. For a very large workspace where the goal is deep within the workspace, the number of nodes could expand exponentially, and a depth-first search will demand a very large memory requirement.

- a True
- b False

1.5. Best-first is a mixed-depth and breadth-first search that uses heuristic values and expands the most desirable unexpanded node.

- a True
- b False

1.6. In design problems or strategic functions, optimality is usually traded in for speed gains.

- a True
- b False

1.7. Graph traversal algorithms outperform shortest path algorithms in applications where the weights of edges in a graph are all equal.

- a True
- b False

1.8. In Dijkstra's algorithm, the priority queue is implemented using which data structure?

- a Stack
- b Queue
- c Heap
- d Array

1.9. When is breadth-first search optimal?

- a When there are fewer nodes
- b When all step costs are equal
- c When all step costs are unequal
- d None of the above

1.10 Which blind search algorithm combines DFS's space-efficiency and BFS's fast search by incrementing the depth limit until the goal is reached?

- a Depth-limited search (DLS)
- b Iterative deepening search (IDS)
- c Uniform-cost search (UCS)
- d Bidirectional search (BS)
- e None of the above

1.11 Which term describes an algorithm with a computational complexity of $O(n \log n)$?

- a Logarithmic
- b Exponential
- c Quasilinear
- d None of the above

1.12. Which search algorithm is implemented with an empty first in, first out queue?

- a Depth-first search
- b Breadth-first search
- c Bidirectional search
- d None of the above

2. Consider the simplified map shown in figure C.2, where the edges are labeled with actual distances between the cities. State the path from city A to city M that would be produced by BFS and the path produced by DFS.

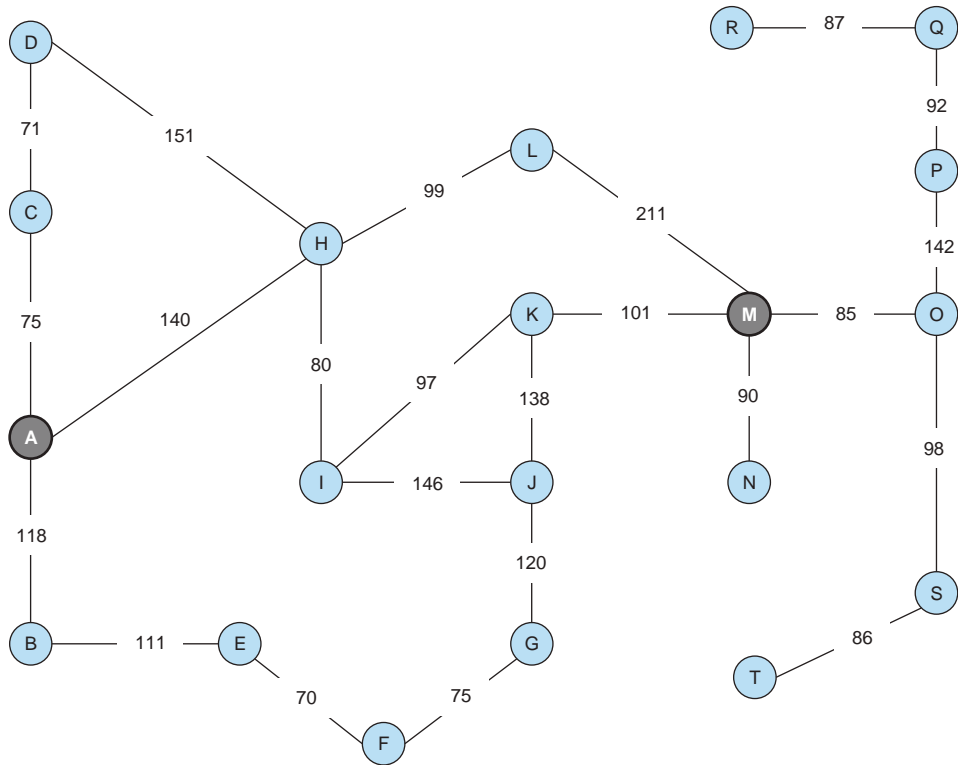


Figure C.2 A simplified map

3. Find the big O notation for the following functions:

- a $10n + n \log(n)$
- b $4 + n/5$
- c $n^5 - 20n^3 + 170n + 208$
- d $n + 10 \log(n)$

4. Consider the search space in figure C.3, where S is the start node and G1 and G2 are the goal nodes. Edges are labeled with the value of a cost function; the number gives the cost of traversing the arc. Above each node is the value of a heuristic function; the number gives the estimate of the distance to the goal. Assume that uninformed search algorithms always choose the left branch first when there is a choice. For each of the depth-first search (DFS) and breadth-first search (BFS) strategies

- a Indicate which goal state is reached first (if any).
- b List, in order, all the states that are popped off the OPEN list.

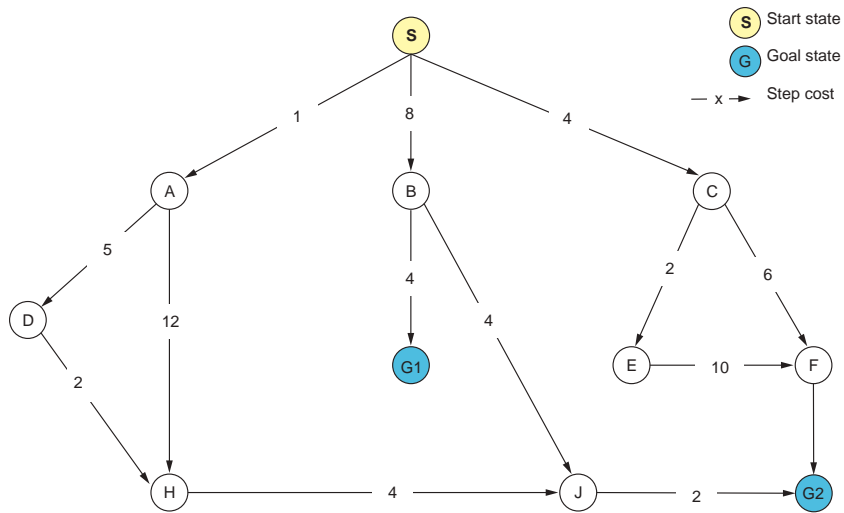


Figure C.3 A graph search exercise

5. Solve the crossword puzzle in figure C.4.

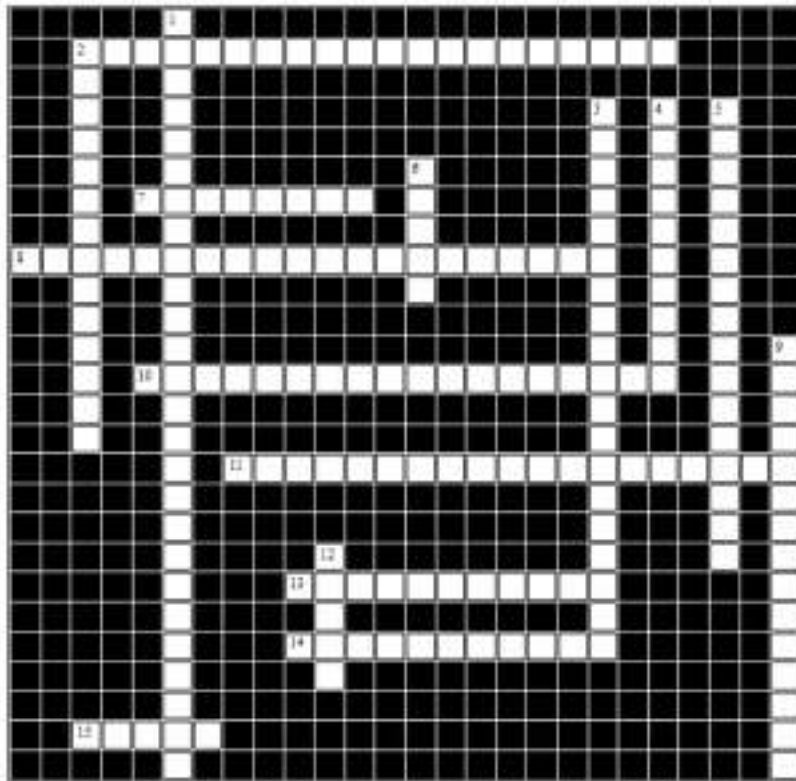


Figure C.4 Blind search crossword puzzle

Across

2. A depth-first search with a predetermined depth limit
7. A blind search algorithm that solves the single-source shortest path problem for a weighted graph with non-negative edge costs
8. A search algorithm that combines forward and backward search
10. A graph traversal algorithm that first explores nodes going through one adjacent to the root, then the next adjacent, until it finds a solution or until it reaches a dead end
11. A variant of Dijkstra's algorithm that is appropriate for large graphs
13. A function that is slightly faster than linear complexity
14. A graph in which multiple edges may connect the same pair of vertices
15. A last in, first out (LIFO) data structure

Down

1. A search algorithm that combines DFS's space-efficiency and BFS's fast search by incrementing the depth limit until the goal is reached
2. A graph used by Twitter to represent following
3. A graph traversal search algorithm that is preferred when the tree is deep
4. A generalization of a graph in which generalized edges can join any number of nodes
5. The type of graph used in LinkedIn to represent users, groups, unregistered persons, posts, skills, and jobs
6. A notation used to describe the performance or complexity of an algorithm
9. The process of exploring the structure of a tree or a graph by visiting the nodes following a certain well-defined rule
12. A first in, first out (FIFO) data structure

Hint: Spaces or dashes *must* be used if the answer consists of two or more words.

C.2.2 Solutions

1. Multiple choice and true/false
 - 1.1. d) All of the above
 - 1.2. d) Depth-first search (DFS)
 - 1.3. a) True
 - 1.4. b) False
 - 1.5. a) True
 - 1.6. b) False
 - 1.7. a) True
 - 1.8. c) Heap
 - 1.9. b) When all step costs are equal
 - 1.10. b) Iterative deepening search (IDS)
 - 1.11. c) Quasilinear
 - 1.12. b) Breadth-first search
2. The route obtained by BFS to go from A to M is
 $A \rightarrow B \rightarrow H \rightarrow C \rightarrow E \rightarrow I \rightarrow L \rightarrow D \rightarrow F \rightarrow J \rightarrow K \rightarrow M \rightarrow H \rightarrow G \rightarrow J \rightarrow K \rightarrow M$;
 tracking back, the final route is $A \rightarrow H \rightarrow L \rightarrow M$.
 The route obtained by DFS to go from A to M is $A \rightarrow B \rightarrow H \rightarrow C \rightarrow E \rightarrow F \rightarrow G \rightarrow J \rightarrow K \rightarrow I \rightarrow M$.

C.3 Chapter 4: Informed search algorithms

C.3.1 Exercises

1. Multiple choice and true/false: Choose the correct answer for each of the following questions.

1.1 How many shortcuts will we have to add to the augmented graph if we decide to contract node E in figure C.6?

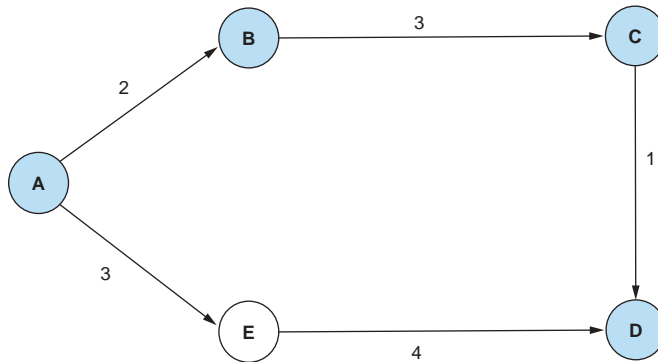


Figure C.6 Contracting node E

- a 0
- b 1
- c 2
- d 3

1.2 The A* algorithm is a special version of

- a Breadth-first search
- b Depth-first search
- c Hill climbing
- d Best-first search
- e Dijkstra's algorithm

1.3 Which of the following is *not* a variant of the hill climbing algorithm?

- a Complex hill climbing
- b Steepest-ascent hill climbing
- c Random-restart hill climbing
- d Steepest-ascent hill climbing
- e All of the above are variants of hill climbing.

1.4 If $f(n)$ is the evaluation function (cost) of a path through n to the goal for each node, and $h(n)$ is an estimated cost from n to the goal, such as the straight-line distance from n to the goal, what is the heuristic function of greedy best-first search?

- a $f(n) \neq h(n)$
- b $f(n) < h(n)$
- c $f(n) = h(n)$
- d $f(n) > h(n)$

1.5 In the directed weighted graph in figure C.7, how many shortcuts are needed if we contract the white node?

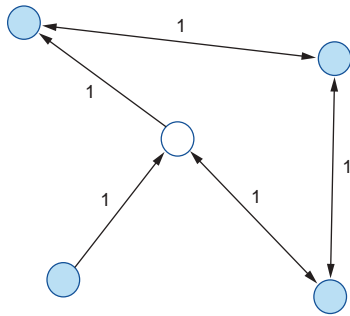


Figure C.7 Directed graph

- a 0
- b 1
- c 2
- d 3
- e 4

1.6 The search strategy that uses problem-specific knowledge is known as

- a Informed search
- b Best-first search
- c A* search
- d Heuristic search
- e All of the above

1.7 Which of the following is an algorithm used to solve the MST problem?

- a Kruskal
- b Borůvka
- c Jarník-Prim
- d Chazelle
- e All of the above

1.8 Hill climbing is an informed breadth-first search that demands little in terms of memory and computational overhead.

- a True
- b False

1.9 Node ordering methods for CH include

- a** Edge difference
- b** Iterative updates
- c** Number of contracted neighbors
- d** Shortcut cover
- e** All of the above

1.10 A* is optimal if $h(n)$ is an admissible heuristic, i.e., $h(n)$ never overestimates the cost to reach the goal.

- a** True
- b** False

1.11 The edge difference is the number of shortcuts introduced when contracting a node minus the number of incoming edges onto the node.

- a** True
- b** False

1.12 Best-first is a mixed depth- and breadth-first search that uses heuristic values and expands the most desirable unexpanded node.

- a** True
- b** False

1.13 What is the evaluation function in A* search?

- a** Estimated cost from the current node to the goal node
- b** Cost of the path through the current node to the goal node
- c** Sum of the path cost through the current node to the goal node and the estimated cost from the current node to the goal node
- d** Average of the path cost through the current node to the goal node and the estimated cost from the current node to the goal node
- e** None of the above

1.14 Which search is complete and optimal when $h(n)$ is consistent?

- a** Best-first search
- b** Depth-first search
- c** Both best-first and depth-first search
- d** A* search

1.15 In the contraction hierarchies (CH) algorithm, we contract nodes based on which of the following?

- a** Decreasing order of their importance
- b** Increasing order of their importance

1.16 The A* search algorithm tries to reduce the total number of states explored by incorporating a heuristic estimate of the cost to get the goal from a given state.

- a** True
- b** False

1.17 The hill climbing algorithm is a local greedy search algorithm that tries to improve the efficiency of breadth-first search.

- a True
- b False

1.18 In CH, the importance of a node may change during the contraction process, necessitating the recomputation of its importance.

- a True
- b False

1.19 A beam search with a beam width equal to the number of nodes in each level is the same as

- a Breadth-first search
- b Depth-first search
- c Hill climbing
- d Best-first search
- e Dijkstra's algorithm

1.20 In CH, the order of contraction does not affect the query performance.

- a True
- b False

2. Consider the search space in figure C.8, where S is the start node and G1 and G2 are the goal nodes. The edges are labeled with the value of a cost function; the number gives the cost of traversing the arc. Above each node is the value of a heuristic function; the number gives the estimate of the distance to the goal.

- a Indicate which goal state is reached first (if any)
- b List in order all the states that are popped off until one of the goal state is found

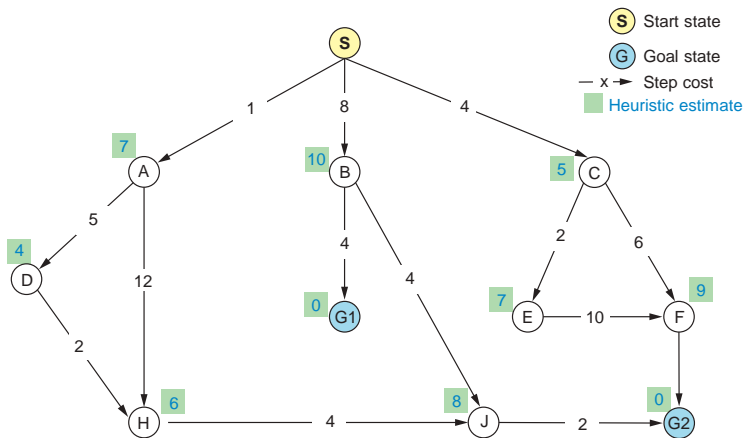


Figure C.8 Search space

3. In the word search puzzle shown in figure C.9, find the hidden terms used in this chapter. You can search horizontally (from left to right or from right to left), vertically (from top to bottom or from bottom to top), or diagonally.



Figure C.9 Informed search word-search puzzle

C.3.2 Solutions

1. Multiple choice and true/false:

- 1.1 a) 0 (no shortcut needed because a witness path exists between A and D)
- 1.2 d) Best-first search
- 1.3 a) Complex hill climbing
- 1.4 c) $f(n) = h(n)$
- 1.5 c) 2
- 1.6 e) All of the above
- 1.7 e) All of the above
- 1.8 b) False (It is a variant of depth-first search.)
- 1.9 e) All of the above
- 1.10 a) True
- 1.11 b) False (Edge difference is the number of shortcuts introduced when contracting a node minus the total degree of the node; i.e., the sum of the number of incoming edges onto the node plus the number of outgoing edges emanating from the node.)
- 1.12 a) True
- 1.13 b) Cost of the path through the current node to the goal node
- 1.14 d) A* search
- 1.15 b) Increasing order of their importance

1.16 a) True

1.17 b) False

1.18 a) True

1.19 a) Breadth-first search

1.20 b) False (The order of contraction does not affect the success of CH but will affect the preprocessing time and the query time. Some contraction ordering systems minimize the number of the shortcuts added in the augmented graph, and thus the overall running time.)

2. The order of expansion is based on the sum of edge weights $g(n)$ and the heuristic estimation $h(n)$; i.e., $f(n) = g(n) + h(n)$. For example, starting from S, $f(A) = 8$, $f(B) = 18$, and $f(C) = 9$, so the queue would look like [A,B,C] because $f(A) < f(C) < f(B)$.

At the next step, when A is popped off the queue, nodes D and H are evaluated such that $f(D) = d(S,A) + d(A,D) + h(D) = 1 + 5 + 4 = 10$, and $f(H) = d(S,A) + d(A,H) + h(H) = 19$, and pushing them onto the queue in order will result in [A,C,D,B,H].

Therefore, the next node to be expanded would be C, which adds E and F to the queue with $f(E) = 11$ and $f(F) = 19$: [A,C,D,E,B,H,F]

Hence, the next node is D, which causes an update on $f(H) = d(S,A) + d(A,D) + d(D,H) + h(H) = 1 + 5 + 2 + 6 = 14$ that subsequently repositions it in the queue: [A,C,D,E,H,B,F]. The same strategy should be followed until one of the goals is found.

a G1

b S, A, C, D, E, H, B, G1

3. The solution to the word search puzzle is shown in figure C.10.



Figure C.10 Informed search word-search puzzle solution

The word directions and start points are formatted as (direction, X , Y).

A-START (E, 7, 10)
BEAM SEARCH (E, 8, 6)
BEST-FIRST (E, 8, 9)
DEPTH-FIRST (E, 7, 12)
DOWNWARD GRAPH
(E, 3, 11)
EDGE DIFFERENCE (E, 2, 4)
HAVERSINE (E, 2, 8)
HILL CLIMBING (E, 5, 1)
INFORMED SEARCH (E, 3, 3)
KRUSKAL ALGORITHM (E, 2, 7)
WITNESS PATH (S,1, 1)

C.4 Chapter 5: Simulated annealing

C.4.1 Exercises

1. Multiple choice and true/false: Choose the correct answer for each of the following questions.

1.1 Simulated Annealing, unlike hill climbing, incorporates a probabilistic mechanism that allows it to accept downward steps, influenced by the current temperature and the quality of the move being considered.

- a True
- b False

1.2 Simulated annealing is an optimization technique that always guarantees finding the global optimum solution.

- a True
- b False

1.3 Dual annealing is an implementation of generalized hill climbing.

- a True
- b False

1.4 In totally adaptive SA, a random combination of previously accepted steps and parameters are used to estimate new steps and parameters.

- a True
- b False

1.5 The simulated annealing algorithm explores more of the search space when the temperature gets lower.

- a True
- b False

1.6. Which cooling schedule asymptotically converges toward the global minimum but requires prohibitive computing time?

- a Linear cooling schedule
- b Geometric cooling schedule
- c Logarithmic cooling schedule
- d Exponential cooling schedule
- e Nonmonotonic adaptive cooling schedule

1.7. SA uses a thermal jump to avoid getting trapped in local minima while quantum annealing relies on quantum tunneling.

- a True
- b False

2. The Rosenbrock function, often referred to as the “valley” or “banana” function, is a nonconvex function that is defined as $f(x,y) = (1-x)^2 + 100(y-x^2)^2$. This is a standard test function and quite tough for most conventional solvers.

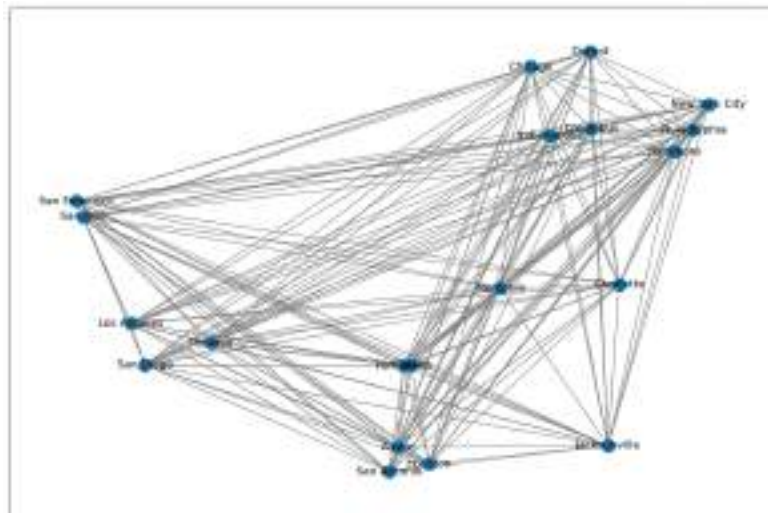
- a Use listing 5.1 or 5.2 or implement your own version of the simulated annealing algorithm from scratch or using one of the libraries mentioned in section A.4 of appendix A to find the global minimum of this function.
- b This banana function is still relatively simple, as it has a curved narrow valley. Other functions, such as the egg crate function, are strongly multimodal and highly nonlinear. Solve this egg crate function as an example of a highly nonlinear multimodal function: $f(x,y) = x^2 + y^2 + 25[\sin^2(x) + \sin^2(y)]$. Consider the domain $(x,y) \in [-5,5] \times [-5,5]$. It would take about 2,500 evaluations to get an optimal solution accurate to the third decimal place.
- c Investigate the rate of convergence of simulated annealing for different cooling schedules.
- d For standard SA, the cooling schedule is a monotonically decreasing function. There is no reason why we should not use other forms of cooling. For example, we can use $T(i) = T_0 \cos^2(i) e^{-\alpha i}$, $\alpha > 0$. Modify the code implemented in the previous step to study the behavior of various functions as a cooling schedule.

3. Modify listing 5.5, or use ASA-GS [2], or implement your own version of simulated annealing to conduct a comparative study between different TSP datasets (see listing B.2 in appendix B to see how to get access to TSP instances). Fill in table C.1 with the tour length obtained by simulated annealing.

Table C.1 SA solutions for different TSP datasets

Dataset	Best known solution	SA solution	CPU time (s)
burma14	30.8758		
ulysses22	75.6651		
oliver30	420		
att48	33,524		
eil51	426		
eil75	535		
kroA100	21,282		
d198	15,780		

4. Solve TSP for 20 major US cities as shown in figure C.11. In this TSP, a travelling salesman must visit a number US cities starting from a specific city. Assume the following cities, defined by their names and GPS latitude and longitude coordinates: New York City (40.72, -74.00); Philadelphia (39.95, -75.17); Baltimore (39.28, -76.62); Charlotte (35.23, -80.85); Memphis (35.12, -89.97); Jacksonville (30.32, -81.70); Houston (29.77, -95.38); Austin (30.27, -97.77); San Antonio (29.53, -98.47); Fort Worth (32.75, -97.33); Dallas (32.78, -96.80); San Diego (32.78, -117.15); Los Angeles (34.05, -118.25); San Jose (37.30, -121.87); San Francisco (37.78, -122.42); Indianapolis (39.78, -86.15); Phoenix (33.45, -112.07); Columbus (39.98, -82.98); Chicago (41.88, -87.63); and Detroit (42.33, -83.05). Visualize the cities and the generated solution as a NetworkX graph using the GPS coordinates of each city.

**Figure C.11 Travelling salesman problem (TSP) for 20 largest US cities**

5. Solve the crossword puzzle in figure C.12.

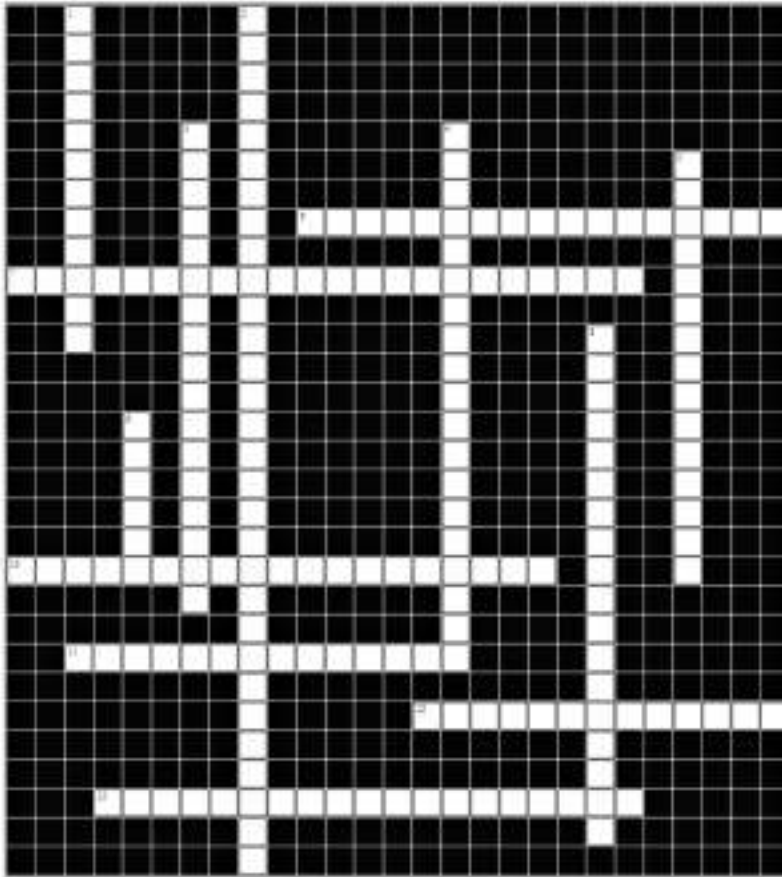


Figure C.12 Simulated annealing crossword puzzle

Across

6. An optimization process that searches an energy landscape to find the optimal or near-optimal solution by applying quantum effects
7. The probability of acceptance or rejection of neighboring solutions
10. A cooling process by which the temperature is decreased very quickly during the first iterations but the speed of the exponential decay is slowed down later
11. A cooling schedule in which the maximum number of iterations needs to be specified
12. Compared to this search algorithm, the main difference is that SA probabilistically allows downward steps controlled by current temperature and how bad a move is.
13. A cooling schedule that requires a prohibitive computing time

Down

1. A stochastic or probabilistic model that describes a sequence of possible moves in which the probability of each move depends only on the state attained in the previous move
 2. A cooling schedule that explicitly takes into consideration how the search is progressing
 3. A cooling schedule that decreases the temperature by a cooling factor
 4. An optimization process based on the physical annealing process
 5. Probability distribution used in transition probability of simulated annealing
 8. A quantum mechanical phenomenon whereby a wavefunction can propagate through a potential barrier
 9. State of the system at which no better or worse moves are being accepted
- Hint: Spaces and dashes *must* be used if the answer consists of two words.

C.4.2 Solutions

1. Multiple choice and true/false
 - 1.1 a) True
 - 1.2 b) False
 - 1.3 b) False (Dual annealing is an implementation of the generalized simulated annealing algorithm.)
 - 1.4 a) True
 - 1.5 b) False (The simulated annealing algorithm exploits the search space when the temperature gets lower.)
 - 1.6 c) Logarithmic cooling schedule
 - 1.7 a) True
2. The generic forms of `__problem_base.py` and `_sa.py`, provided in the book's GitHub repo, can be used to solve this problem. You can also modify listing 5.1 or 5.2 to solve this problem.
3. You can use listing 5.5 or the ASA-GS implementation [1] to run simulated annealing with selected parameters on different TSP instances and report your results. In listing 5.5, replace the permanent link with the link corresponding to the TSP instance. For example, click on `burma14.tsp`, click on the button with three dots at the upper-right corner, and select Copy Permalink. Consider tuning the algorithm's parameters to get close to the best tour length known (so far) for each dataset.
4. The next listing shows how to use the generic solver implemented as part of the `optalgo-tools` package to solve this problem.

Listing C.3 Solving TSP using SA

```
from optalgotools.algorithms import SimulatedAnnealing
from optalgotools.problems import TSP
pairwise_distances = distances

tsp_US = TSP(dists=pairwise_distances, gen_method='insert',
```

```

➡ init_method='greedy')
sa=SimulatedAnnealing(max_iter=10000, max_iter_per_temp=10000,
➡ initial_temp=10000000, final_temp=0.0001,
➡ cooling_schedule='geometric', cooling_alpha=0.9, debug=1)

sa.run(tsp_US)
print(sa.s_best)

```

As a continuation of listing C.3, the following code shows how to solve this problem using the `simanneal` Python library:

```

#!pip install simanneal
import math
import random
from simanneal import Annealer

class TravellingSalesmanProblem(Annealer):
    def __init__(self, state, distance_matrix):
        self.distance_matrix = distance_matrix
        super(TravellingSalesmanProblem, self).__init__(state)

    def move(self):
        initial_energy = self.energy()
        a = random.randint(0, len(self.state) - 1)
        b = random.randint(0, len(self.state) - 1)
        self.state[a], self.state[b] = self.state[b], self.state[a]
        return self.energy() - initial_energy

    def energy(self):
        e = 0
        for i in range(len(self.state)):
            e += self.distance_matrix[self.state[i-1]][self.state[i]]
        return e

init_state = list(cities)
random.shuffle(init_state)

tsp = TravellingSalesmanProblem(init_state, distance_matrix)
tsp.set_schedule(tsp.auto(minutes=0.2))
tsp.copy_strategy = "slice"
state, e = tsp.anneal()

while state[0] != 'New York City':
    state = state[1:] + state[:1]

print("%i mile route:" % e)
print(" → ".join(state))

```

Test the annealer with a TSP as per the `simanneal` module.

Swap two cities in the route.

Calculate the length of the route.

Initial state, a randomly ordered itinerary

Set New York City as the home city.

Print the route and its cost.

As a continuation, the following code can be used to visualize the problem and its solution:


```

fig, ax = plt.subplots(figsize=(15,10))
reversed_dict = {key: value[::-1] for key, value in cities.items()}
H = G.copy()
edge_list = list(nx.utils.pairwise(state))
nx.draw_networkx_edges(H, pos=reversed_dict, edge_color="gray", width=0.5)
ax=nx.draw_networkx(H, pos=reversed_dict, with_labels=True, edgelist=edge_
list, edge_color="red", node_size=200, width=3,)
plt.show()

```

Reverse the lat and long for a correct visualization.

Create an independent shallow copy of the graph and attributes.

Draw the route.

Visualize.

Draw the closest edges on each node only.

Running this code produces the visualization shown in figure C.13.

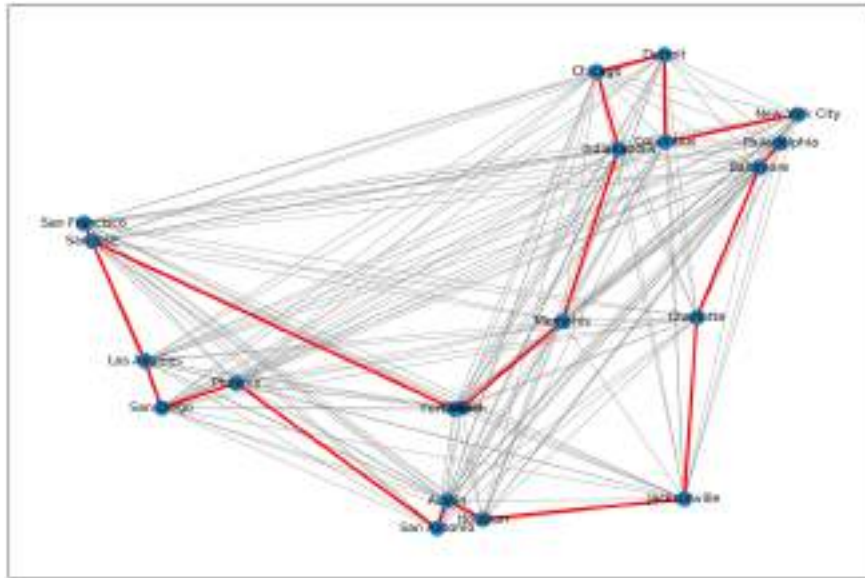


Figure C.13 Solution for the 20 major US cities TSP using simanneal

Appendix A shows examples of available Python libraries, such as scikit-opt, DEAP, OR-Tools and simanneal, that can solve this problem using simulated annealing and other metaheuristics.

5. The solution to the crossword puzzle is shown in figure C.14.

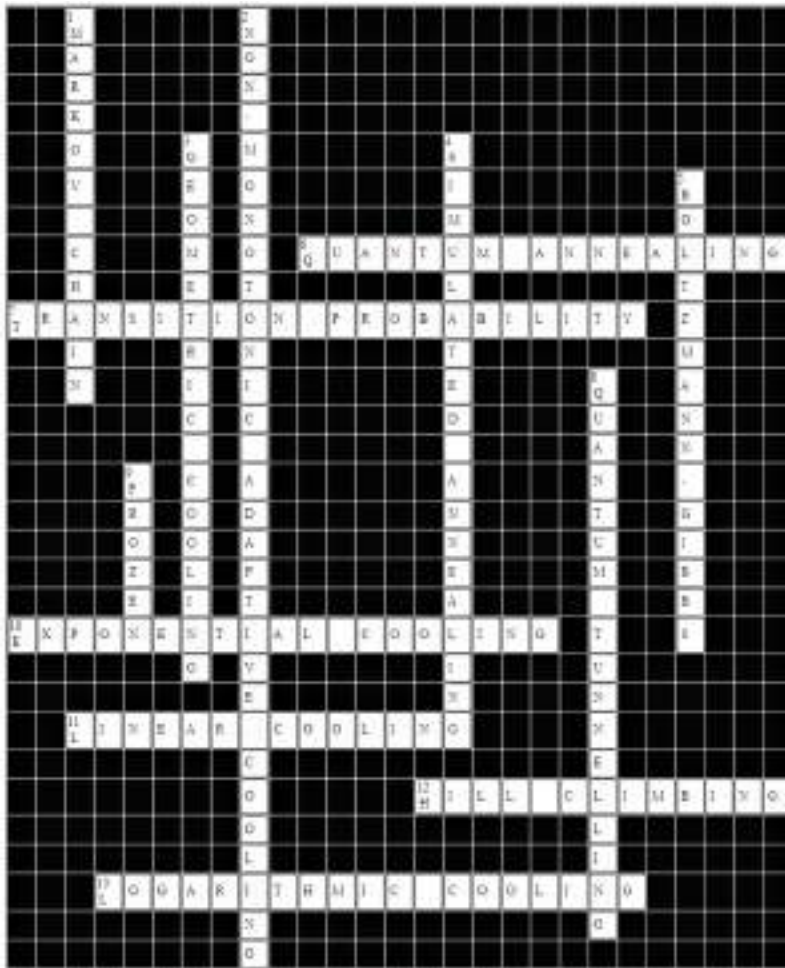


Figure C.14 Simulated annealing crossword puzzle solution

C.5 Chapter 6: Tabu search

C.5.1 Exercises

1. Multiple choice and true/false: Choose the correct answer for each of the following questions.

1.1. In TS, non-improving solutions are conditionally accepted in order to escape from a local optimum.

- a True
- b False

1.2. A frequency-based memory maintains information about how recently a search point has been visited.

- a** True
- b** False

1.3. In order to increase the efficiency for solving some problems, TS uses memory via a tabu list to avoid revisiting recently visited neighborhoods.

- a** True
- b** False

1.4. The stopping criteria to terminate TS can be

- a** Neighborhood is empty
- b** Number of iterations is larger than a specified threshold
- c** Evidence shows that an optimum solution has been obtained
- d** All of the above (a, b, and c)
- e** None of the above (a, b, or c)

1.5. Tabu-active moves are stored in a long-term memory.

- a** True
- b** False

1.6. For large and difficult problems (scheduling, quadratic assignment, and vehicle routing), tabu search obtains solutions that often represent a global optimum or near-optimum.

- a** True
- b** False

1.7. Recency memory is used to increase intensification of the search by remembering neighbors with good solutions.

- a** True
- b** False

1.8. Aspiration criteria are used to revoke tabu-active moves as a way to avoid search stagnation.

- a** True
- b** False

1.9. TS can be considered a combination of random search and memory structures.

- a** True
- b** False

1.10. When the length of the tabu list is too small, the algorithm can get trapped in cycles, and when it's too long, many moves could be prevented at each iteration, leading to stagnation.

- a** True
- b** False

2. As shown in appendix B, the Schwefel function is complex, with many local minima. Figure C.15 shows the two-dimensional form of the function.

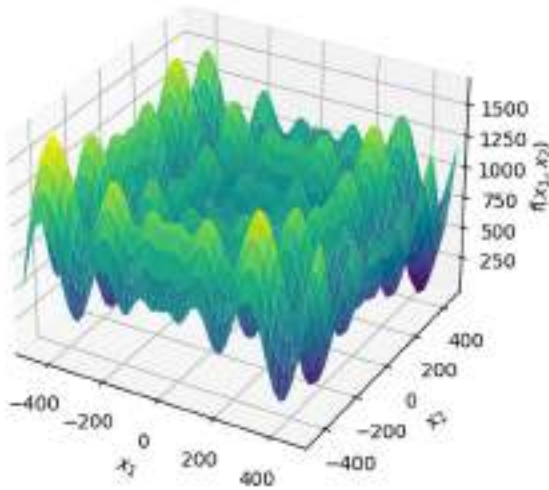


Figure C.15 The Schwefel function

Modify listing 6.1 to solve the Schwefel function using tabu search.

3. Apple filed US Patent 7790637 B2 for a composite laminate that includes seven sheets stacked one over another and a scrim layer. Seven planar sheets of fibers impregnated with a resin (e.g., a viscous liquid substance) are placed with different orientations to improve the strength of the composite laminate, as illustrated in figure C.16. The scrim layer is the cosmetic layer, which is a different material and is bonded to the outside to improve the cosmetic exterior surface and consistent appearance.

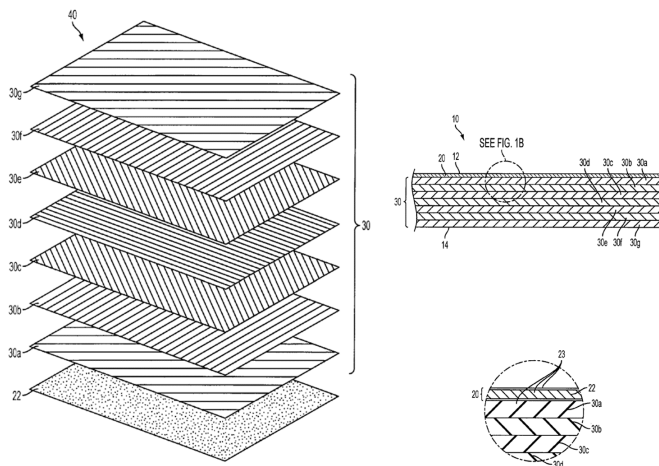


Figure C.16 Composite laminate design—Apple's US patent 7790637 B2

The way the seven layers are ordered results in different levels of strength in the composite laminate. Assume that the pairwise strength gain or loss is given by the empirical values in figure C.17. Positive values represent a strength gain in the case of contact between the two layers (up or down), and negative values represent the strength loss.

	b	c	d	e	f	g
a	+3	-1	+4	-2	0	+1
b		+2	+3	+2	-1	0
c			0	-1	+1	+2
d				+2	0	-1
e					-3	+4
f						+3

Figure C.17 Pairwise strength gain or loss

Assume that we want to find the optimal ordered combination (i.e., permutation) of the seven fiber sheets to maximize the composite laminate strength. Perform four hand iterations of tabu search to show the steps of the algorithm to solve this problem. Write Python code to solve this problem.

4. Write Python code to use tabu search to solve the simple assembly line balancing problem, type 1 (SALBP-1), with the machine and worker constraints described in Kamarudin and Rashid's paper "Modelling of Simple Assembly Line Balancing Problem Type 1 (SALBP-1) with Machine and Worker Constraints" [3].

5. In the word search puzzle in figure C.18, find the hidden terms commonly used in tabu search. You can search horizontally (from left to right or right to left), vertically (from top to bottom or bottom to top), or diagonally.



Figure C.18 TS word search puzzle

C.5.2 Solutions

1. Multiple choice and true/false

1.1. a) True

1.2. b) False (Recency-based memory maintains information about how recently a search point has been visited.)

1.3. a) True

1.4. d) All of the above (a, b, and c)

1.5. b) False (Tabu-active moves are stored in a short-term memory.)

1.6. a) True

1.7. a) True

1.8. a) True

1.9. b) False (TS can be considered a combination of combination of local search and memory structures.)

1.10. a) True

2. Listing C.4 in the book's GitHub repository shows how to solve the Schwefel function using tabu search.

3. Figure C.19 shows the TS initialization and first iteration for the composite laminate problem. The number of possible permutations without repetition is $n!$. The number of possible solutions for ordering the seven fiber sheets is $7! = 5,040$. To generate a neighboring solution, fiber sheet swapping can be used. The neighborhood is defined as any other solution that is obtained by a pairwise exchange of any two sheets in the solution. If we let the number of nodes $n = 7$, pairwise exchange $k = 2$. The number of neighbors is the number of combinations without repetition $C(n, k)$ or n -choose- k : $C(n, k) = n! / (k!(n - k)!) = 21$ neighbors. Let's assume that the tabu tenure is set at 3 iterations.

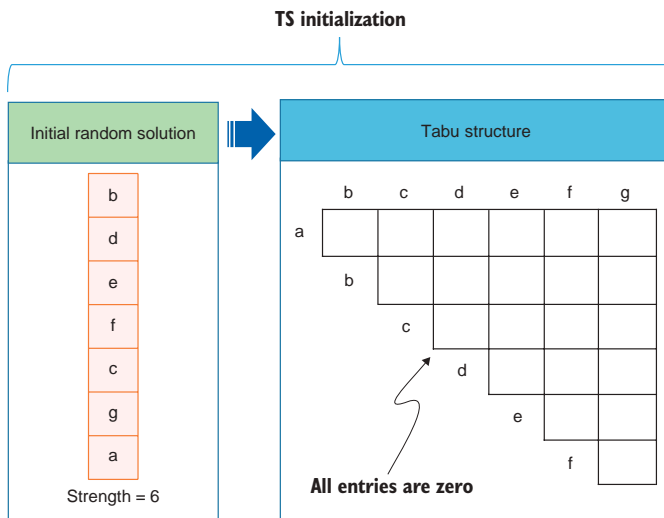


Figure C.19 TS initialization for the composite laminate design

Figures C.20 through C.22 show iterations 1, 2, and 3 respectively.

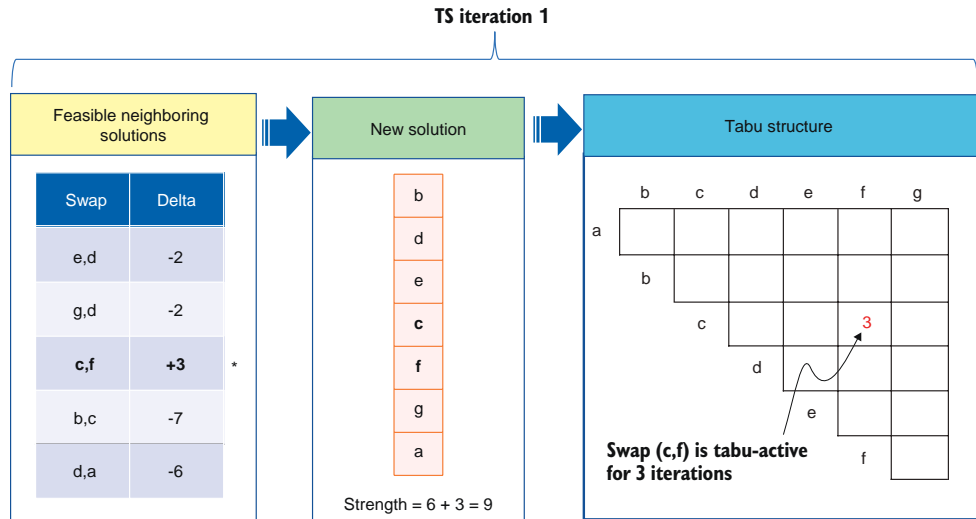


Figure C.20 Composite laminate design—iteration 1

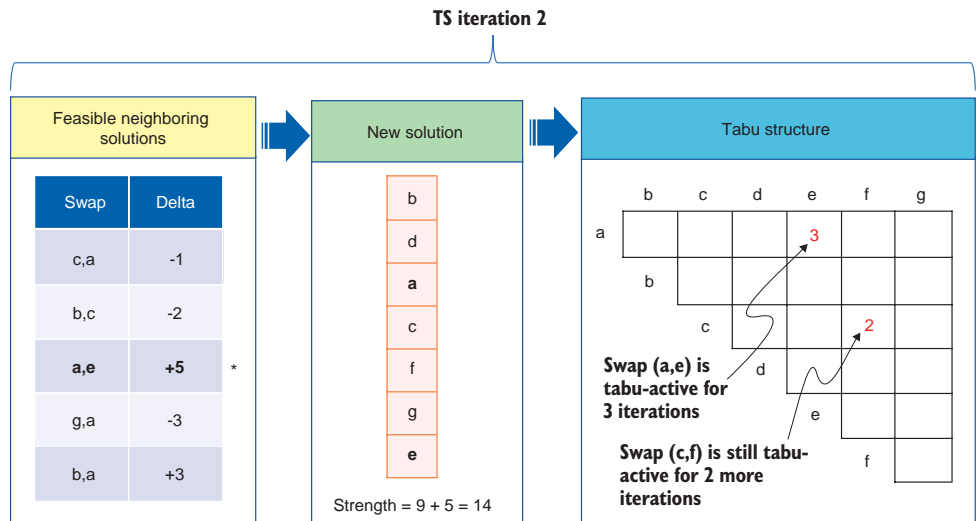


Figure C.21 Composite laminate design—iteration 2

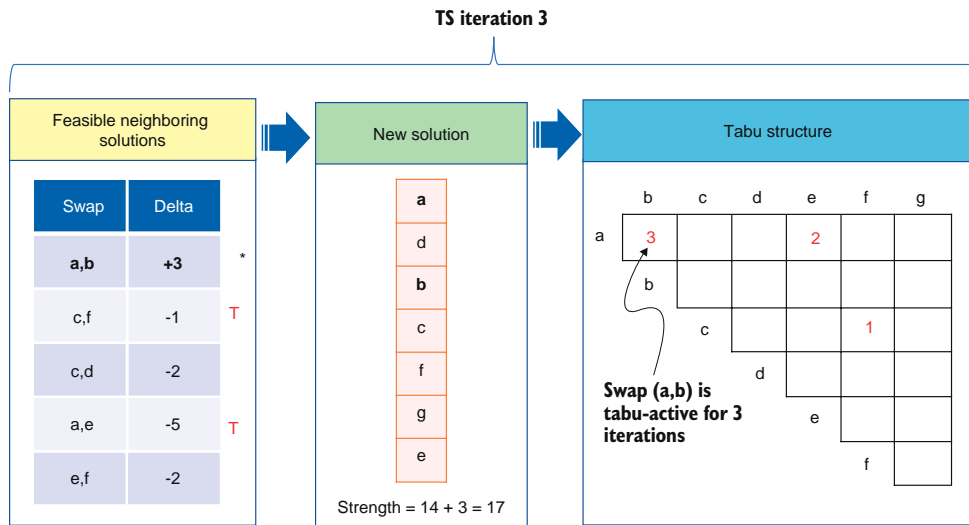


Figure C.22 Composite laminate design—iteration 3

In each iteration, we generate a number of candidate solution using layer swapping, and we select the move that results in the maximum delta value, which is the difference between the previous solution and the new solution in terms of strength gain. In the next iteration, shown in figure C.23, there is no move with a positive gain, so the best (non-tabu) move will be non-improving. The move (a,e) is selected, since it has a tabu tenure of only one iteration compared to the other tabu-active best move (a,b). This means that the tabu status of (a,e) can be overridden by applying the aspiration criteria.

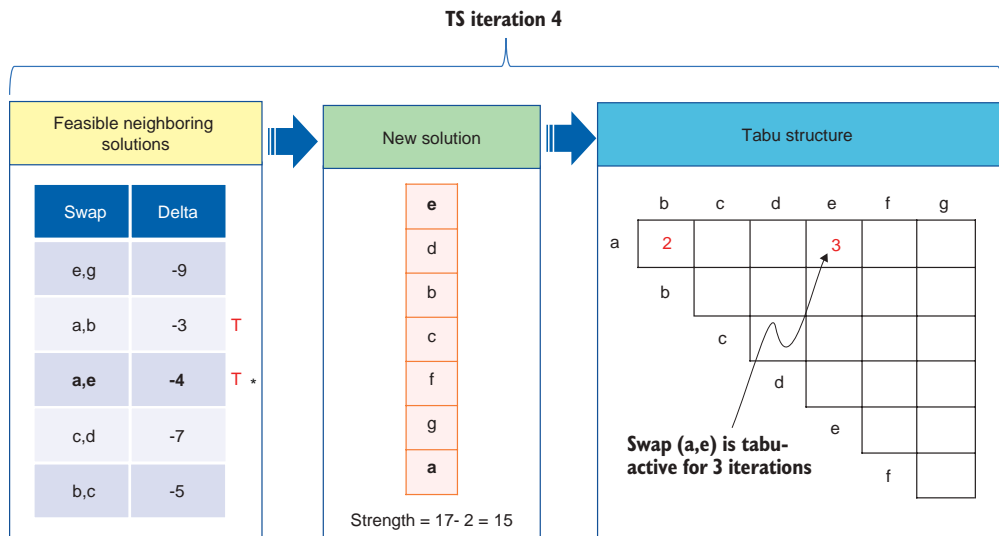


Figure C.23 Composite laminate design—TS iteration 4 and aspiration criterion

To show the diversification using frequency-based memory, assume that the solution reached after 26 iterations is [a c f b g e d], and the strength value is 23. Assume that we will penalize the solution based on its frequency in usage (a highly repetitive solution gets more penalties). The tabu structure is updated based on both recency-based (upper triangle) and frequency-based (lower triangle) memories, as shown in figure C.24a. The five top candidates after swapping are shown in figure C.24b.

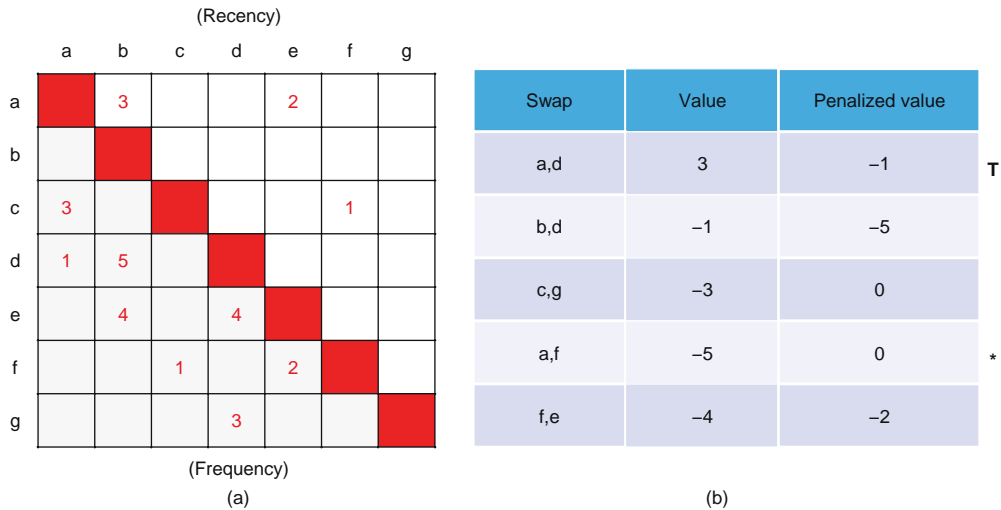


Figure C.24 a) Recency- and frequency-based memory, b) five top candidates after swapping with their penalty values

We need to diversify when no admissible improving moves exist. Non-improving moves are penalized by assigning larger penalties to more frequent swaps. According to the penalized value, swap (a,f) is chosen. Listing C.5 shows the solution of the composite laminate problem using tabu search.

Listing C.5 Solving the composite laminate problem using tabu search

```

Import the tabu search solver.
from optalgotools.problems import SHEETS
from optalgotools.algorithms import TabuSearch
import matplotlib.pyplot as plt

Import the SHEETS problem instance, which contains the composite laminate design problem description.
iphone_case = SHEETS(init_method='greedy')

Create different sheets objects. For a complete list of the supported parameters, see the SHEETS class in the sheets.py file in the optalgotools.problems module.

ts = TabuSearch(max_iter=1000, tabu_tenure=10, neighbor_size=7,
    use_aspiration=True, aspiration_limit=3, use_longterm=False, debug=1,
    maximize=True, penalize=True)

Create a TS object to help in solving the composite laminate problem.
ts.init_ts(iphone_case)

Get an initial random solution, and check its cost.
ts.val_cur

```

```
ts.run(iphone_case, repetition=1)
ts.val_allbest
```

Run TS, and evaluate
the best solution cost.

4. Listing C.6 shows a snippet of the code for solving the SALBP-1 problem using tabu search.

Listing C.6 Solving the SALBP-1 problem using tabu search

Read the
data from
appendix B
directly using
the URL.

```
tasks = pd.DataFrame(columns=['Task', 'Duration'])
tasks= pd.read_csv("https://raw.githubusercontent.com/Optimization-
Algorithms-Book/Code-Listings/main/Appendix%20B/data/ALBP/ALB_TS_DATA2.txt",
sep =",")
Prec= pd.read_csv("https://raw.githubusercontent.com/Optimization-
Algorithms-Book/Code-
Listings/main/Appendix%20B/data/ALBP/ALB_TS_PRECEDENCE2.txt", sep =",")
Prec.columns=['TASK', 'IMMEDIATE_PRECEDESSOR']
```

```
Cycle_time = 10
```

Define the cycle time.

```
tenure = 3
max_itr=100
```

Ensure the feasibility of the solution
considering the task precedence constraint.

Get an initial solution.

```
solution = Initial_Solution(len(tasks))
soln_init = Make_Solution_Feasible(solution, Prec)
```

Run the
tabu search.

```
sol_best, SI_best=tabu_search(max_itr, soln_init, SI_init, tenure, WS,
tasks, Prec_Matrix, Cycle_time)
```

```
Smoothing_index(sol_best, WS, tasks, Cycle_time, True)
```

Calculate the smoothing
index of the best solution.

Visualize
the
solution.

```
plt = Make_Solution_to_plot(sol_best, WS, tasks, Cycle_time)
plt.show()
```

Running the complete code produces the following output:

```
The Smoothing Index value for ['T2', 'T7', 'T6', 'T1', 'T3', 'T4', 'T8',
➡ 'T9', 'T5'] solution sequence is: 1.0801234497346432
The number of workstations for ['T2', 'T7', 'T6', 'T1', 'T3', 'T4', 'T8',
➡ 'T9', 'T5'] solution sequence is: 6
The workloads of workstation for ['T2', 'T7', 'T6', 'T1', 'T3', 'T4', 'T8',
➡ 'T9', 'T5'] solution sequence are: [7. 6. 5. 7. 6. 6.]
1.0801234497346432
```

Complete code is available in the book's GitHub repo.

5. The solution to the word search puzzle is shown in figure C.25.

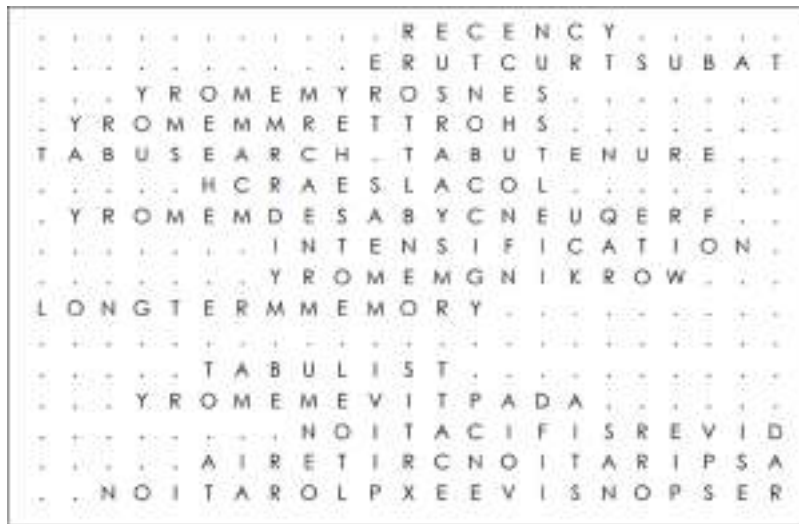


Figure C.25 TS word search puzzle solution

The word directions and start points are formatted as (Direction, X, Y)

- ADAPTIVE MEMORY (W, 17, 13)
- ASPIRATION CRITERIA (W, 23, 15)
- DIVERSIFICATION (W, 23, 14)
- FREQUENCY-BASED MEMORY (W, 21, 7)
- INTENSIFICATION (E, 8, 8)
- LOCAL SEARCH (W, 16, 6)
- LONG-TERM MEMORY (E, 1, 10)
- RECENCY (E, 12, 1)
- RESPONSIVE EXPLORATION (W, 23, 16)
- SENSORY MEMORY (W, 16, 3)
- SHORT-TERM MEMORY (W, 16, 4)
- TABU LIST (E, 6, 12)
- TABU SEARCH (E, 1, 5)
- TABU STRUCTURE (W, 23, 2)
- TABU TENURE (E, 12, 5)
- WORKING MEMORY (W, 20, 9)

C.6 Chapter 7: Genetic algorithm

C.6.1 Exercises

1. Multiple choice and true/false: Choose the correct answer for each of the following questions.

1.1. Given a binary string 1101001100101101 and another binary string yxyxyxyxyxyxy in which the values 0 and 1 are denoted by x and y, what offspring result from applying 1-point crossover on the two strings at a randomly selected recombination point?

- a yxyxyxyxy11010 and yxyx01100101101
- b 11010xyxyxyxyxy and yxyx01100101101
- c 11010xyxyxyxyxy and 01100101101xyxyx
- d None of the above

1.2. In binary genetic algorithms, how does the bitwise mutation operator work?

- a It swaps the positions of two randomly chosen bits.
- b It averages the values of two randomly chosen bits.
- c It flips a randomly chosen bit in the binary representation.
- d It reverses the order of a randomly chosen segment of bits.

1.3. The name of the position on the chromosome every gene has is called

- a Allele
- b Locus
- c Genotype
- d Phenotype
- e None of the above

1.4. In the steady-state model of genetic algorithms, how are new offspring introduced into the population?

- a By replacing the entire population
- b By replacing a small fraction of the population
- c By adding them to the existing population
- d By replacing the worst individuals in the population

1.5. Suppose you have the population shown in table C.2.

Table C.2 Given population

Individual	Individual 1	Individual 2	Individual 3	Individual 4	Individual 5
Fitness	12	25	8	53	10

Rank-based selection attempts to remove problems of fitness-proportionate selection (FPS) by basing selection probabilities on relative rather than absolute fitness. Assume that the ranking process is linear ranking, as follows:

$$p(r) = \frac{2 - SP}{N} + \frac{r(i)(SP - 1)}{N(N - 1)}$$

where N is the number of individuals in the population and r is the rank associated with each individual in this population (the least fit individual has $r = 1$, the fittest individual $r = N$). SP is the selection pressure (assume $SP = 1.5$). Which two individuals will be selected if we use this linear ranking-based selection?

- a Individuals 1 and 2
- b Individuals 1 and 3
- c Individuals 2 and 3
- d Individuals 2 and 4
- e Individuals 3 and 4
- f None of the above

1.6. Which selection method in genetic algorithms involves choosing a fixed number of individuals at random and selecting the best among them?

- a Roulette wheel selection
- b Rank selection
- c Tournament selection
- d Stochastic universal sampling (SUS)

1.7. In P-metaheuristics, the computational cost of the Latin hypercube strategy is the same as the pseudo-random initialization strategy.

- a True
- b False

1.8. In binary genetic algorithms, which type of chromosome encoding is used?

- a Real value
- b Permutation
- c Binary
- d Tree

1.9. Which of the following methods can be used to convert a minimization problem into a maximization problem in optimization?

- a Adding a constant to the objective function
- b Taking the reciprocal of the objective function
- c Negating the objective function
- d Scaling the objective function by a factor

1.10. Which of the following is an advantage of the steady-state model compared to the generational model in genetic algorithms?

- a Faster convergence
- b Better diversity preservation
- c Lower computational cost
- d More robust mutation operators

1.11. In a binary genetic algorithm, what does the mutation operator do to an individual's genes?

- a Reverses the gene value (1 to 0 or 0 to 1)
- b Randomly assigns a new gene value (0 or 1)
- c Swaps the positions of two genes
- d Combines genes from different individuals

1.12. Which of the following is a commonly used crossover method in binary genetic algorithms?

- a Single-point crossover
- b Two-point crossover
- c Uniform crossover
- d All of the above

1.13. What is the primary difference between generational and steady-state models in genetic algorithms?

- a Selection methods
- b Crossover operators
- c Mutation operators
- d Replacement strategies

1.14. Which crossover method in binary genetic algorithms involves exchanging genetic material between parent chromosomes based on a predefined probability for each gene?

- a Single-point crossover
- b Two-point crossover
- c Uniform crossover
- d Arithmetic crossover

1.15. What is a potential disadvantage of using a high mutation rate in binary genetic algorithms?

- a Loss of diversity in the population
- b Premature convergence
- c Disruption of good solutions
- d Decreased selection pressure

1.16. In the generational model of genetic algorithms, what happens to the population in each generation?

- a A small fraction of the population is replaced.
- b The entire population is replaced.
- c The population remains the same.
- d The population size is gradually reduced.

1.17. When using a transformation to convert a minimization problem into a maximization problem, which property of the optimal solution must be preserved?

- a Feasibility
- b Optimality
- c Dominance
- d Convexity

1.18. What are the primary operators used in genetic algorithms?

- a Initialization, pooling, and backpropagation
- b Selection, crossover, and mutation
- c Convolution, pooling, and activation
- d Forward propagation, backward propagation, and optimization

1.19. Which of the following is an advantage of the generational model compared to the steady-state model in genetic algorithms?

- e Improved diversity preservation
- f Faster convergence
- g Lower computational cost
- h Better handling of constraints

1.20. Suppose you need to solve the following function maximization problem using a binary genetic algorithm:

$$f(x) = V - \frac{O_i(x) \times N}{\sum_{i=1}^N O_i(x)}, \quad 0.0 \leq x \leq 31.5$$

where

- O_i is the objective function value of individual i and $O_i = -(x - 6.4)^2$.
- N is the population size.
- V is a large value to ensure non-negative fitness values.

The value of V is the maximum value of the second term of the fitness function $f(x)$ so that the fitness value corresponding to the maximum value of the objective function is 0.

0. How many bits are required to represent the solution with precision 0.1?

- a 6 bits
- b 7 bits
- c 8 bits
- d 9 bits
- e 10 bits
- f None of the above

2. The Ackley function is a nonlinear, multimodal function with a large number of local minima, making it a challenging optimization problem. The general form of the Ackley function is

$$f(x) = -a \exp \left(-b \sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2} \right) - \exp \left(\frac{1}{d} \sqrt{\sum_{i=1}^d \cos(cx_i)} \right) + a + \exp(1)$$

where

- $x = (x_1, x_2, \dots, x_d)$ is the input vector.
- a , b , and c are positive constants (usually $a = 20$, $b = 0.2$, and $c = 2 \times \pi$).
- d is the dimension of the input vector.

The function has a global minimum at the origin ($x_i = 0$), where $f(x) = 0$, and it is surrounded by several local minimam, as shown in figure C.26.

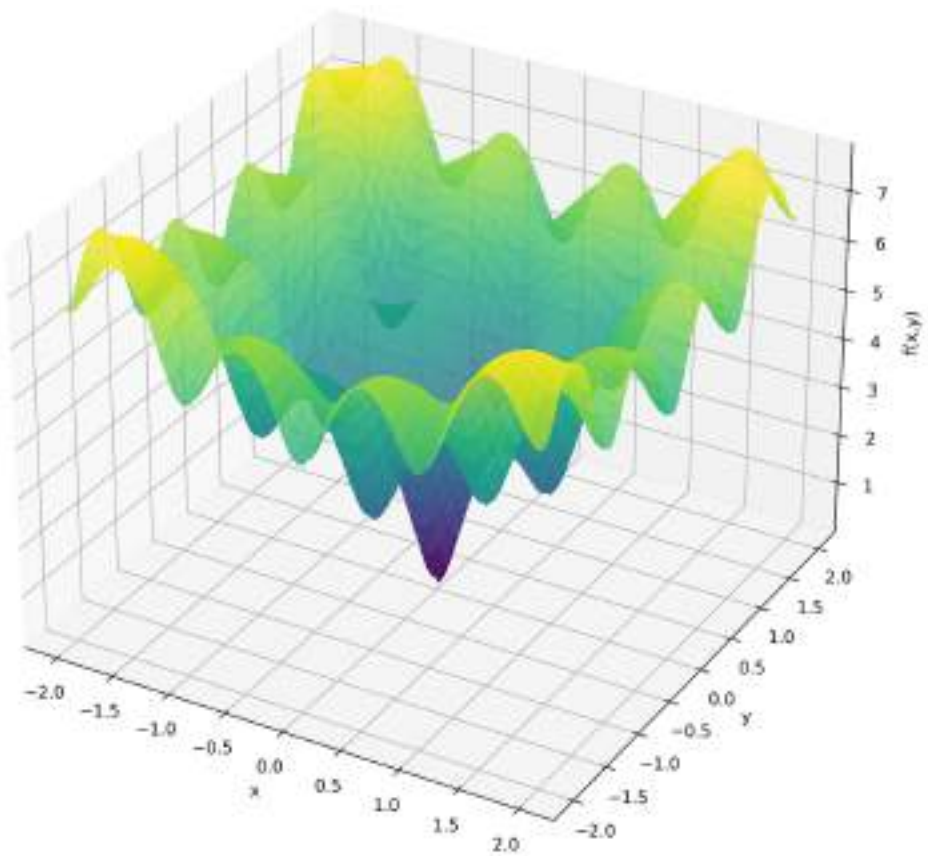


Figure C.26 The Ackley function

The presence of these local minima makes it difficult for optimization algorithms to find the global minimum, especially those that can get stuck in local minima. Write Python code to solve a 6D Ackley function.

C.6.2 Solutions

1. Multiple choice and true/false

1.1. b) 11010yxyxyxyxy and yxyx01100101101

1.2. c) It flips a randomly chosen bit in the binary representation.

1.3. b) Locus

1.4. b) By replacing a small fraction of the population

1.5. Considering the following linear ranking

$$p(r) = \frac{2 - SP}{N} + \frac{r(i)(SP - 1)}{N(N - 1)} = \frac{0.5}{5} + \frac{0.5 \times r(i)}{5 \times 4} = 0.1 + 0.025r(i)$$

Individual	Individual 1	Individual 2	Individual 3	Individual 4	Individual 5
Fitness	12	25	8	53	10
Ranking r	3	4	1	5	2
p(r)	0.175	0.2	0.125	0.225	0.15

Individuals 4 and 2 will be selected, as they have the highest probability. So the correct answer is d.

1.6. c) Tournament selection

1.7. a) True

1.8. c) Binary

1.9. b and c) Taking the reciprocal of the objective function, and negating the objective function

1.10. c) Lower computational cost

1.11. a) Reverses the gene value (1 to 0 or 0 to 1)

1.12. d) All of the above

1.13. d) Replacement strategies

1.14. c) Uniform crossover

1.15. c) Disruption of good solutions

1.16. b) The entire population is replaced.

1.17. b) Optimality

1.18. b) Selection, crossover, and mutation

1.19. a) Improved diversity preservation

1.20. d) 9 bits

Given $0.0 \leq x \leq 31.5$ with a precision of 0.1

- Calculate the range size: $(31.5 - 0) = 31.5$
- Divide the range size by the desired precision: $31.5 / 0.1 = 315$
- Round up to the nearest whole number: 315

Now you have 315 steps (values) to represent the numbers from 0.0 to 31.5 with a precision of 0.1:

$$\begin{aligned}\text{number_of_bits} &= \text{ceil}(\log_2(\text{number_of_steps})) \\ &= \text{ceil}(\log_2(315)) \approx \text{ceil}(8.29) = 9\end{aligned}$$

So you need 9 bits to represent the numbers from 0.0 to 31.5 with a precision of 0.1. Listing C.8 shows a snippet of the code to solve the Ackley function using a genetic algorithm implemented from scratch or based on the pymoo open source library. We start by defining constants for the genetic algorithm and Ackley function as follows:

Listing C.8 Solving the Ackley function using GA

```
import random
import math

POPULATION_SIZE = 100
GENOME_LENGTH = 30
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.8
GENERATIONS = 100

a = 20
b = 0.2
c = 2 * math.pi
```

We then define the decode function to convert the binary genome to real values:

```
def decode(genome):
    x = []
    for i in range(0, GENOME_LENGTH, 5):
        binary_string = "".join([str(bit) for bit in genome[i:i+5]])
        x.append(int(binary_string, 2) / 31 * 32 - 16)
    return x
```

The following function defines the fitness function (Ackley function):

```
def fitness(genome):
    x = decode(genome)
    term1 = -a * math.exp(-b * math.sqrt(sum([(xi ** 2) for xi in x]) /
    ➤ len(x)))
    term2 = -math.exp(sum([math.cos(c * xi) for xi in x]) / len(x))
    return term1 + term2 + a + math.exp(1)
```

Let's now define the crossover function:

```
def crossover(parent1, parent2):
    child1 = []
    child2 = []
    for i in range(GENOME_LENGTH):
        if random.random() < CROSSOVER_RATE:
            child1.append(parent2[i])
```

```

        child2.append(parent1[i])
    else:
        child1.append(parent1[i])
        child2.append(parent2[i])
    return child1, child2

```

The mutation function is defined as follows:

```

def mutate(genome):
    for i in range(GENOME_LENGTH):
        if random.random() < MUTATION_RATE:
            genome[i] = 1 - genome[i]

```

Now it's time to create an initial population:

```

population = [[random.randint(0, 1) for _ in range(GENOME_LENGTH)] for _ in
    range(POPULATION_SIZE)]

```

Let's now run the genetic algorithm:

```

for generation in range(GENERATIONS):
    fitness_scores = [(genome, fitness(genome)) for genome in population]
    fitness_scores.sort(key=lambda x: x[1])

    print(f"Generation {generation}: Best fitness =
    {fitness_scores[0][1]}")

    parents = [fitness_scores[i][0] for i in range(POPULATION_SIZE // 2)]

    next_generation = []
    for i in range(POPULATION_SIZE // 2):
        parent1 = random.choice(parents)
        parent2 = random.choice(parents)
        child1, child2 = crossover(parent1, parent2)
        mutate(child1)
        mutate(child2)
        next_generation.append(child1)
        next_generation.append(child2)

    population = next_generation

```

Calculate the fitness score for each genome in the population.

Print out the best fitness score for each generation.

Select the best half of the population to act as parents for the next generation.

Generate the next generation by applying crossover and mutation.

Replace the current population with the new generation of children.

You can print the best genome and its fitness as follows:

```

fitness_scores = [(genome, fitness(genome)) for genome in population]
fitness_scores.sort(key=lambda x: x[1])
print(f"Generation {GENERATIONS}: Best fitness = {fitness_scores[0][1]}")

```

You can print the best genome, decode the best genome, and then print the decision variables in real values as follows:

```

print(f"Best genome = {fitness_scores[0][0]}")
print(f"Best genome decoded = {decode(fitness_scores[0][0])}")
print(f"Decision variables in real values = {decode(fitness_scores[0][0])}")

```

You can solve this problem faster using GA implemented in pymoo as follows:

```
from pymoo.algorithms.soo.nonconvex.ga import GA
from pymoo.problems import get_problem
from pymoo.optimize import minimize

problem = get_problem("ackley", n_var=6)

algorithm = GA(
    pop_size=100,
    eliminate_duplicates=True)

res = minimize(problem,
               algorithm,
               seed=1,
               verbose=False)

print("Best solution found: \nX = %s\nF = %s" % (res.X, res.F))
```

Running this code produces the following output:

```
Best solution found:
X = [ 0.00145857 -0.00011553 0.00033902 -0.00169267 -0.0005825 -0.00547546]
F = [0.01003609]
```

C.7 Chapter 8: Genetic algorithm variants

C.7.1 Exercises

1. Assume that a company is trying to choose the optimal location for a new facility. The decision variables x_1 and x_2 represent the coordinates of the potential location for the new facility in a 2D plane. The company needs to minimize the distance squared between the new facility and an existing facility at the location (2, 1). This could represent the need to minimize transportation costs between the two facilities. Moreover, the company aims to maximize the distance squared between the new facility and a competitor's facility at the location (3, 4). This could represent the need to establish a competitive advantage by distancing the new facility from the competitor. The new facility must be located within a certain region due to zoning regulations or other restrictions. It also must adhere to environmental constraints or property boundaries. This multi-objective constrained optimization problem can be formulated as follows:

Minimize $f_1(x_1, x_2) = \sqrt{(x_1 - 2)^2 + (x_2 - 1)^2}$ (distance between the company facilities)

Maximize $f_2(x_1, x_2) = -\sqrt{(x_1 - 3)^2 + (x_2 - 4)^2}$ (distance between the new facility and a competitor's facility)

such that

$$g_1(x_1, x_2) = x_1 + 2x_2 - 6 \leq 0 \text{ (zoning regulations)}$$

$$g_2(x_1, x_2) = 2x_1 - x_2 - 2 \leq 0 \text{ (environmental constraints or property boundaries)}$$

$$0 \leq x_1 \leq 5 \text{ (boundary constraint)}$$

$$0 \leq x_2 \leq 5 \text{ (boundary constraint)}$$

Write Python code to find the optimal location for the new facility.

2. The All Ones problem, also known as the Max Ones problem, is a simple problem that aims to maximize the number of ones in a binary string of a fixed length. For a 10-bit All Ones problem, the optimal solution takes this form: [1111111111]. The problem is described by K. Sutner in his article “Linear cellular automata and the Garden-of-Eden” [4] as follows: suppose each of the squares of an $n \times n$ chessboard is equipped with an indicator light and a button. If a square’s button is pressed, the light for that square will change from off to on or vice versa; the same happens to the lights of all the edge-adjacent squares. Initially all lights are off. The goal of the All Ones problem is to find a sequence of buttons that can be pressed in such a way that all lights are on at the end. This problem itself may not have direct applications, but it is used as a test problem to compare the performance of various algorithms. Write Python code to find a solution for the 10-bit All Ones problem using a genetic algorithm.

3. Assume that there are 10 parcels to be loaded in the cargo bike in figure C.27.

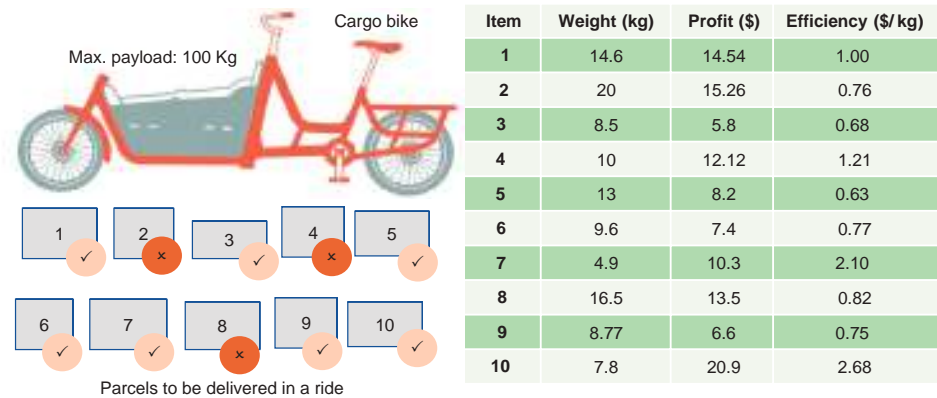


Figure C.27 Cargo bike loading problem for 10 items with given weight, profit, and efficiency (profit/weight)

Each parcel has its own weight, profit, and efficiency value (profit per kg). The goal is to select the parcels to be loaded in such a way that the utility function f_1 is maximized and the weight function f_2 is minimized.

$$f_1 = \sum E_i, i = 0, 1, \dots, n.$$

$$f_2 = |\sum w_i - C|, 50 \text{ is added if and only if } \sum w_i > C$$

where

- n is the total number of packages.
- E_i is the efficiency of package i .
- w_i is the weight of package i .
- C is the maximum capacity of the bike.

A penalty of 50 is added if and only if the total weight of the added parcels exceeds the maximum capacity. Write Python code to determine which items should be loaded for maximum efficiency.

4. In the opencast mining problem described by Guéret et al. in their book “Linear programming” [5], the opencast uranium mine is divided into blocks for exploitation. As illustrated in figure C.28, there are 18 blocks of 10,000 tons on three levels, identified based on the results of test drilling. The pit needs to be terraced to allow trucks to reach the bottom, and the pit is limited in the west by a village and in the east by a group of mountains. To extract a block, three blocks of the level above it need to be extracted due to constraints on the slope: the block immediately on top of it, and the blocks to the right and to the left. There are different costs associated with extracting blocks depending on their level. Level 1 blocks cost \$100 per ton to extract, level 2 blocks cost \$200 per ton, and level 3 blocks cost \$300 per ton. However, the hatched blocks, which are formed of a very hard rock rich in quartz, cost \$1,000 per ton to extract. The blocks that contain uranium are displayed in a gray shade: these are blocks 0, 6, 9, 11, 16, and 17. These blocks have different market values, with block 17 being rich in ore but made of the same hard rock as the other hatched blocks. The market values of blocks 0, 6, 9, 11, 16, and 17 are \$200, \$300, \$500, \$200, \$1,000, and \$1,200 per ton, respectively.

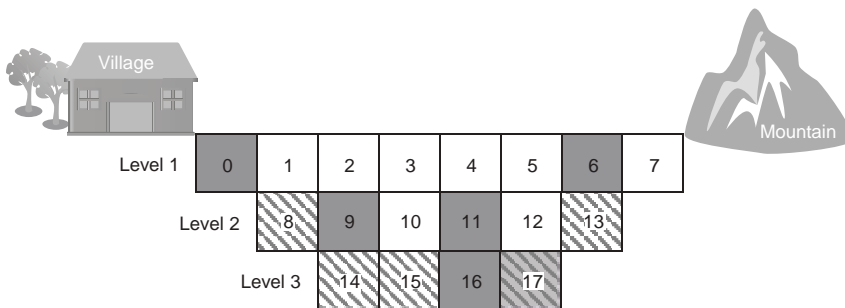


Figure C.28 Opencast mining problem

- a Write a Python code to determine which blocks to extract to maximize the total profit, given by $\sum (\text{VALUE} - \text{COST}) \times x_i$ for $x = 0, 1, \dots, 17$, where x_i is an assignment binary variable.

- b Assume that the time duration required to process the blocks is [1, 1, 1, 1, 1, 1, 1, 3, 2, 2, 2, 2, 3, 4, 4, 3, 4] depending on the block level and the hardness of the block. Write Python code to determine which blocks to extract to maximize the discounted profit and increased cost given by the following equations:

$$f_1 = \sum \frac{\text{VALUE} - \text{COST}}{(1 + \text{discount_rate})^{\text{TIME}}} \times x_i$$

$$f_2 = \sum ((\text{COST} \times (1 + 2 \times \text{TIME})) \times x_i$$

5. In the Chapter 8/Projects folder in the book's GitHub repository, the following sample research projects are provided for review and experimentation:

- *Routing*—Addresses using the genetic algorithm to find the shortest path between two points of interest in Vaughan, a municipality to the north of Toronto.
- *Bus routing*—Addresses the school bus routing problem, formulating the problem as a contained multi-objective optimization problem. The cluster-first route-second scheme, genetic algorithm, and adaptive genetic algorithm are applied to solve this problem. The performance of these algorithms is evaluated using real data from public schools in the city of Winchester, Virginia, USA.
- *Location allocation*—Tackles the placement of drone delivery stations using bio-inspired optimization algorithms. The solution framework consists of two stages: the first tackles the location planning problem of stations, while the second deals with the allocation of delivery demand to located stations.

C.7.2 Solutions

1. The next listing shows how to solve the facility allocation problem using NSGA-II.

Listing C.9 Solving the facility allocation problem using NSGA-II

```
import numpy as np
import math
import matplotlib.pyplot as plt
from pymoo.core.problem import ElementwiseProblem
from pymoo.algorithms.moo.nsga2 import NSGA2
from pymoo.operators.crossover.sbx import SBX
from pymoo.operators.mutation.pm import PolynomialMutation
from pymoo.operators.repair.rounding import RoundingRepair
from pymoo.operators.sampling.rnd import FloatRandomSampling

class FacilityProblem(ElementwiseProblem):

    def __init__(self):
        super().__init__(n_var=2,
```

```

        n_obj=2,
        n_ieq_constr=2,
        xl=np.array([0,0]),
        xu=np.array([5,5]))

    def _evaluate(self, x, out, *args, **kwargs):
        f1 = math.sqrt((x[0]-2)**2 + (x[1]-1)**2)
        f2 = -math.sqrt((x[0]-3)**2 + (x[1]-4)**2)

        g1 = x[0]+2*x[1]-6  ← Zooning constraint
        g2 = 2*x[0]-x[1]-2  ← Environment/property constraint

        out["F"] = [f1, f2]
        out["G"] = [g1, g2]

problem = FacilityProblem()

algorithm = NSGA2(
    pop_size=50,
    sampling=FloatRandomSampling(),
    crossover=SBX(prob=0.8),
    mutation = PolynomialMutation(prob=0.6, repair=RoundingRepair()),
    eliminate_duplicates=True
)

from pymoo.termination import get_termination

termination = get_termination("n_gen", 50)

from pymoo.optimize import minimize

res = minimize(problem,
                algorithm,
                termination,
                seed=1,
                save_history=True,
                verbose=False)

X = res.X
F = res.F

```

The complete listing, available in the book's GitHub repository, also shows how to perform decision making to select the best solution.

2. The All Ones problem solution is shown in the following listing.

Listing C.10 Solving the All Ones problem using GA

```

import numpy as np
from pymoo.algorithms.soo.nonconvex.ga import GA
from pymoo.operators.crossover.sbx import SBX
from pymoo.operators.mutation.pm import PolynomialMutation
from pymoo.operators.sampling.rnd import FloatRandomSampling

```



```

from pymoo.core.problem import Problem
from pymoo.optimize import minimize

class AllOnes(Problem):
    def __init__(self, n_var):
        super().__init__(n_var=n_var, n_obj=1, n_constr=0, xl=0, xu=1,
➡ vtype=int)

    def _evaluate(self, x, out, *args, **kwargs):
        out["F"] = -np.sum(x, axis=1)

problem = AllOnes(n_var=10)
algorithm = GA(
    pop_size=100,
    sampling=FloatRandomSampling(),
    crossover=SBX(prob=1.0, eta=30, n_offsprings=2),
    mutation=PolynomialMutation(prob=1.0),
    eliminate_duplicates=True
)

res = minimize(problem, algorithm, ('n_gen', 100), seed=1, verbose=True)

print("Sum of the ones in the solution:", -res.F)
print("Solution: ", res.X)

```

This code defines an `AllOnes` class that extends the `Problem` class from `pymoo`. The `_evaluate` method of the `AllOnes` class calculates the fitness of an individual by counting the number of ones in the binary string and returning the negative of that count (because `pymoo` minimizes objective functions).

3. The solution to the cargo bike loading problem is shown in the next listing.

Listing C.11 Solving the cargo bike loading problem using GA

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pymoo.algorithms.soo.nonconvex.ga import GA
from pymoo.operators.crossover.ux import UniformCrossover
from pymoo.operators.mutation.pm import PolynomialMutation
from pymoo.operators.sampling.rnd import FloatRandomSampling
from pymoo.termination.default import DefaultSingleObjectiveTermination
from pymoo.core.problem import Problem
from pymoo.optimize import minimize

class CargoBike(Problem): ➡ Define the problem.
    def __init__(self, weights, efficiency, capacity):
        super().__init__(n_var=len(weights), n_obj=1, n_constr=1, xl=0,
➡ xu=1, vtype=bool)
        self.weights = weights
        self.efficiency = efficiency
        self.capacity = capacity

    def _evaluate(self, x, out, *args, **kwargs):

```

Define the
problem
inputs.

```

x = np.round(x) # Ensure X is binary
total_weight = np.sum(self.weights * x, axis=1)
total_profit = np.sum(self. efficiency * x, axis=1)
out["F"] = -total_profit[:, None]
out["G"] = np.where(total_weight <= self.capacity, 0, total_weight
➡ - self.capacity)[:, None]

weights = np.array([14.6, 20, 8.5, 10, 13, 9.6, 4.9, 16.5, 8.77, 7.8])
profits = np.array([14.54, 15.26, 5.8, 12.12, 8.2, 7.4, 10.3, 13.5, 6.6,
➡ 20.9])
efficiency = np.array([1.00, 0.76, 0.68, 1.21, 0.63, 0.77, 2.1, 0.82, 0.75,
➡ 2.68])
capacity = 100
df=pd.DataFrame({'Weight (kg)':weights,'Profit ($)':profits,'Efficiency ($/
Kg)':efficiency})

problem = CargoBike(weights, efficiency, capacity) ➡ Create a problem instance.

algorithm = GA(
    pop_size=200,
    sampling=FloatRandomSampling(),
    crossover=UniformCrossover(prob=1.0),
    mutation=PolynomialMutation(prob=0.5),
    eliminate_duplicates=True
) ➡ Define the genetic algorithm.

termination = DefaultSingleObjectiveTermination()

res = minimize(problem, algorithm, termination, verbose=True) ➡ Run the optimization.

print(f"Best solution found: {res.X}")
print(f"Best objective value: {-res.F[0]}") ➡ Print the results.

res_bool=np.round(res.X)
selected_items = df.loc[res_bool.astype(bool), :]

fig, ax1 = plt.subplots()
ax1.bar(1+selected_items.index, selected_items['Efficiency ($/Kg)'])
ax1.set_ylabel('Efficiency ($/Kg)')
ax1.set_ylim(0, 5)
ax1.legend(['Efficiency ($/Kg)'], loc="upper left")

ax2 = ax1.twinx()
ax2.bar(1+selected_items.index, selected_items['Weight (kg)'], width=0.5,
➡ alpha=0.5, color='orange')
ax2.grid(False)
ax2.set_ylabel('Weight (kg)')
ax2.set_ylim(0, 30)
ax2.legend(['Weight (kg)'], loc="upper right")
plt.show() ➡ Visualize the solution.

```

Running this code produces the solution shown in figure C.29.

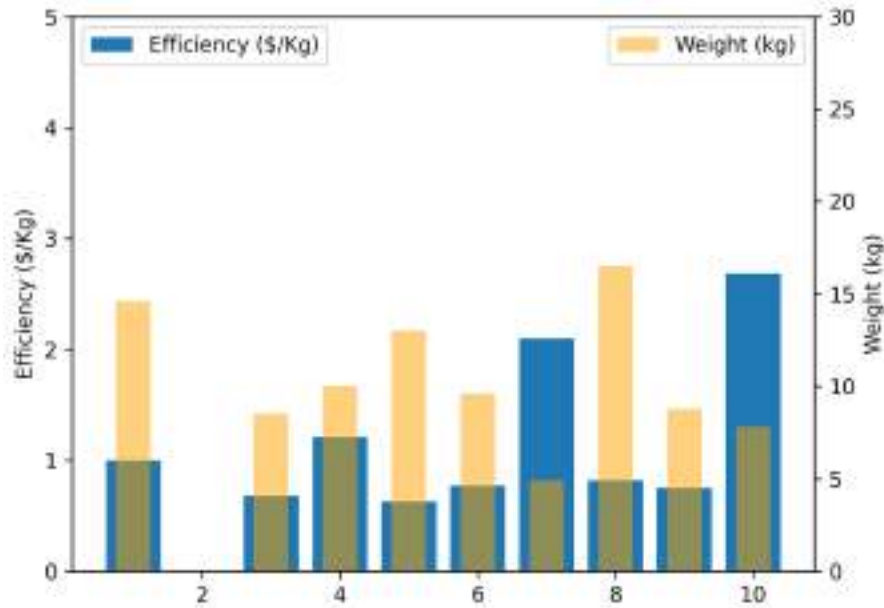


Figure C.29 Cargo bike loading solution

4. The next listing shows the steps to solve the opencast mining problem.

Listing C.12 Solving the opencast mining problem

```
import numpy as np

BLOCKS = np.arange(0, 17)
LEVEL23 = np.array([8, 9, 10, 11, 12, 13, 14, 15, 16, 17])
COST = np.array([100, 100, 100, 100, 100, 100, 100, 100, 100,
                 200, 200, 200, 200, 200, 200, 200,
                 1000, 1000, 1000, 1000])

VALUE = np.array([200, 0, 0, 0, 0, 0, 0, 300, 0,
                 0, 500, 0, 200, 0, 0,
                 0, 0, 1000, 0])

Precedence = np.array([[0, 1, 2],
                       [1, 2, 3],
                       [2, 3, 4],
                       [3, 4, 5],
                       [4, 5, 6],
                       [5, 6, 7],
                       [8, 9, 10],
                       [9, 10, 11],
                       [10, 11, 12],
                       [11, 12, 13]])
```

Annotations:

- Blocks in the mine:** Points to `BLOCKS = np.arange(0, 17)`
- Blocks in levels 2 and 3:** Points to `LEVEL23 = np.array([8, 9, 10, 11, 12, 13, 14, 15, 16, 17])`
- Cost of blocks:** Points to `COST = np.array(...)`
- Value of blocks:** Points to `VALUE = np.array(...)`
- Define the ARC for each block in LEVEL23:** Points to `Precedence = np.array(...)`

As a continuation, you can visualize the precedence of extraction of blocks in levels 2 and 3 as follows:

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.DiGraph()  # Create the directed graph.

G.add_nodes_from(BLOCKS)  # Add the nodes.

for b, arc in zip(LEVEL23, Precedence):
    for predecessor in arc:
        G.add_edge(predecessor, b)

pos = nx.spring_layout(G)  # Create a pos dictionary with a default position for all nodes.
pos.update({0: (0, 0), 1: (0, 1), 2: (0, 2), 3: (1, 0), 4: (1, 1), 5: (1, 2),
    ➡ 6: (2, 0), 7: (2, 1),
    8: (2, 2), 9: (3, 0), 10: (3, 1), 11: (3, 2), 12: (4, 0), 13:
    ➡ (4, 1),
    14: (4, 2), 15: (5, 0), 16: (5, 1), 17: (5, 2)})  # Update the positions of nodes
                                                         # that have a specific position.

plt.figure(figsize=(10, 5))
nx.draw(G, pos, with_labels=True, node_size=1500, node_color='skyblue',
    ➡ font_size=12, font_weight='bold')
plt.title("Precedence graph for extraction of blocks in Level 2 and 3")
plt.show()
```

Add precedence edges.

Draw the graph.

Figure C.30 shows the precedence graph for extracting blocks in levels 2 and 3.

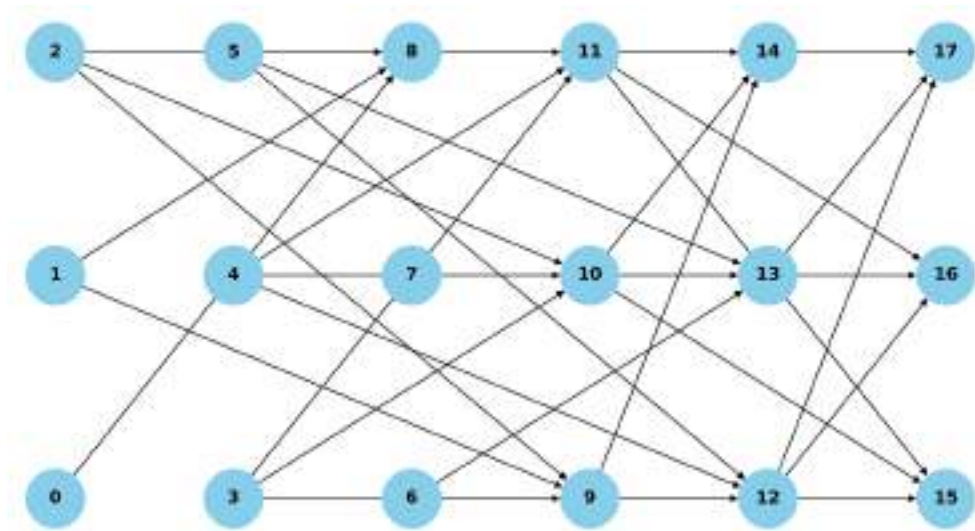


Figure C.30 Precedence graph for extracting blocks in levels 2 and 3

As a continuation of listing C.12, the following code snippet shows how to define the problem as a single objective constrained optimization problem.

```
from pymoo.algorithms.soo.nonconvex.ga import GA
from pymoo.operators.crossover.pntx import PointCrossover
from pymoo.operators.mutation.pm import PolynomialMutation
from pymoo.operators.repair.rounding import RoundingRepair
from pymoo.operators.sampling.rnd import FloatRandomSampling
from pymoo.core.problem import Problem
from pymoo.optimize import minimize

class OpencastMiningProblem(Problem):
    def __init__(self):
        super().__init__(n_var=18, n_obj=1, n_constr=len(LEVEL23), xl=0,
➡ xu=1)

    def _evaluate(self, X, out, *args, **kwargs):
        X = np.round(X)
        profits = np.sum((VALUE - COST) * X, axis=1)

        constraints = np.zeros((X.shape[0], len(LEVEL23)))
        for i in range(X.shape[0]):
            for j, b in enumerate(LEVEL23):
                constraints[i, j] = min(X[i, Precedence[j]-1]) - X[i, b-1]

        out["F"] = profits.reshape(-1, 1)
        out["G"] = constraints
```

We can now define the GA solver and apply it to solve the problem as follows:

```
problem = OpencastMiningProblem()
algorithm = GA(
    pop_size=50,
    sampling=FloatRandomSampling(),
    crossover=PointCrossover(prob=0.8, n_points=2),
    mutation = PolynomialMutation(prob=0.3, repair=RoundingRepair()),
    eliminate_duplicates=True
)

res = minimize(problem, algorithm, ('n_gen', 50), seed=1, verbose=True)

print("Best solution found:\nX =", res.X)
print("Objective value =", -res.F[0])
```

Running this code produces the following output:

```
Best solution found:
X = [0 1 1 0 1 1 0 1 1 0 1 0 1 1 1 0 1]
Objective value = 4300.0
```

The complete version of listing C.12 in the book's GitHub repository includes the Pareto optimization implementation, which treats the problem as a multi-objective optimization problem.

C.8 Chapter 9: Particle swarm optimization

C.8.1 Exercises

1. Multiple choice and true/false: Choose the correct answer for each of the following questions.

1.1. Which of the following is a drawback of the star structure in PSO?

- a It may lead to premature convergence to a suboptimal solution.
- b It may cause the algorithm to converge too slowly.
- c It may cause the algorithm to become stuck in local optima.
- d It does not allow the algorithm to explore the search space effectively.

1.2. Asynchronous PSO usually produces better results because it causes the particles to use more up-to-date information.

- a True
- b False

1.3. What is the difference between the cognitive and social components in PSO?

- a The cognitive component is based on the particle's own experience, while the social component is based on the experience of the swarm as a whole.
- b The cognitive component is based on the experience of the particle's neighbors, while the social component is based on the particle's own experience.
- c The cognitive component is based on random perturbations, while the social component is based on gradient information.
- d The cognitive and social components are the same thing.

1.4. Ring topology, or lbest PSO, has been shown to converge faster than other network structures, but with a susceptibility to be trapped in local minima.

- a True
- b False

1.5. In binary PSO (BPSO), each bit in the binary strings is updated by considering its current state, the best state it has held so far, and the best state of its neighboring bits.

- a True
- b False

1.6. In PSO, the velocity model that makes particles behave like independent hill climbers is

- a Cognitive-only model
- b Social-only model

1.7. PSO is guaranteed to find the global optimum of a function.

- a True
- b False

- 1.8. What is the role of the acceleration coefficients in the PSO algorithm?
- a To control the speed of the particles
 - b To control the exploration/exploitation trade-off
 - c To control the swarm size
 - d To control the mutation rate
- 1.9. PSO is a local search algorithm that is only able to find local optima.
- a True
 - b False
- 1.10. PSO was originally developed for continuous-valued spaces.
- a True
 - b False

2. In the restaurant game described by Martin et al. in their article “Local termination criteria for swarm intelligence: A comparison between local stochastic diffusion search and ant nest-site selection,” [6], a group of delegates attends a long conference in an unfamiliar town. Each night they have to find somewhere to dine. There is a large choice of restaurants, each of which offers a large variety of meals. The problem the group faces is to find the best restaurant, which is the restaurant where the maximum number of delegates would enjoy dining. Even a parallel exhaustive search through the restaurant and meal combinations would take too long to accomplish. If you were to solve this problem using PSO, how would you describe the three components of the velocity update in the context of this problem?

3. Ridesharing is a successful implementation of the sharing economy business model where personal vehicles are shared by their owners or drivers with individual travelers who have similar itineraries and schedules. The ridesharing problem is a multi-objective constrained optimization problem. A non-comprehensive list of optimization goals for ridesharing includes

- Minimizing the total travel distance or time of drivers’ trips
- Minimizing the total travel time of passengers’ trips
- Maximizing the number of matched (served) requests
- Minimizing the cost for the drivers’ trips
- Minimizing the cost for the passengers’ trips
- Maximizing the driver’s earnings
- Minimizing the passenger’s waiting time
- Minimizing the total number of drivers required

Consider the ridesharing problem shown in figure C.31, where the objective is to come up with a schedule for drivers that minimizes the total distances of vehicles’ trips. The statement of this multi-objective optimization is as follows:

Find s that minimizes f

$$f = \sum_{m \in V} \sum_{i,j \in P} S_{i,j}^m d_{i,j}$$

where

- f is the total distance of vehicles' trips to serve all the passengers.
- $s_{i,j}$ is 1 if location j is assigned within the schedule of the driver and 0 otherwise.
- $d_{i,j}$ is the distance between two points within the route, where $d_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.
- P is the passenger set, with known pickup and delivery points.
- V is the vehicle set, with predefined initial and final locations.

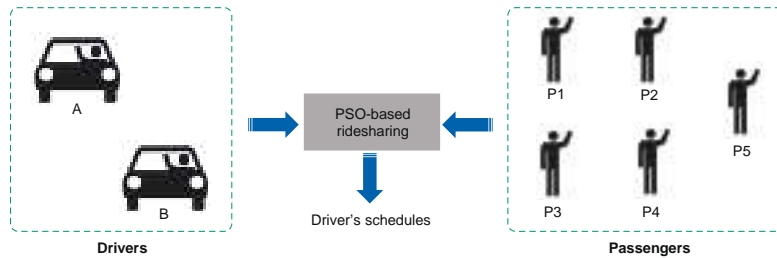


Figure C.31 The ridesharing problem

Given the pickup and drop-off locations of five passengers (table C.3) and the initial and final locations of the two drivers (table C.4), define a suitable representation for a particle (i.e., a candidate solution) that represents each driver's schedule and evaluates its fitness.

Table C.3 Pickup and drop-off locations of the passengers

Passengers	Pickup x-coordinate	Pickup y-coordinate	Drop-off x-coordinate	Drop-off y-coordinate
P1	9	9	4	2
P2	8	5	2	4
P3	6	1	8	6
P4	7	6	9	8
P5	3	5	10	3

Table C.4 Initial and final locations of the drivers

Drivers	Initial x-coordinate	Initial y-coordinate	Final x-coordinate	Final y-coordinate
A	4	1	8	10
B	1	6	9	4

4. Consider the trip itinerary planning problem where the objective is to provide an optimal travel itinerary for tourists visiting a new city, taking into consideration the quality of the places to be visited, the proximity of the attractions to one another, and how much of the day can be completely occupied with little idle time. The problem is stated as following:

Find X which optimizes f

$$f(X) = \frac{1}{1 + \frac{Z_c}{480}} \times Z_r \times \frac{Z_d}{480}$$

subject to the following constraints:

- The duration of the itinerary must not exceed total time in a given day, which is set to 480 minutes. This is expressed as the summation of the duration of each venue and total commute time to be visited in an itinerary. This means that $Z_d + Z_c \leq 480$ minutes.
- There should never be repetitions of the same venue in a solution.

where

- $X = [Z_d \ Z_c \ Z_r]^T$
- Z_d represents the duration of the itinerary (minutes).
- Z_c represents the total commute time in the itinerary.
- Z_r represents the average rating of all the venues in an itinerary. The ratings are derived from Yelp, Google reviews, or others.

Table C.5 lists the commute times between 10 attractions.

Table C.5 Commute time between attractions

Commute time (mins)		To								
From	0	1	2	3	4	5	6	7	8	9
	0	10	20	7	11	8	19	7	1	1
	1	0	12	14	1	7	12	10	22	22
	2	12	0	25	21	28	1	22	12	12
	3	14	25	0	15	15	25	4	6	6
	4	1	21	15	0	25	22	11	10	10
	5	7	28	15	25	0	28	14	28	29
	6	12	1	25	22	28	0	22	12	12
	7	10	22	4	11	14	22	0	3	2
	8	22	12	6	10	28	12	3	0	1
	9	22	12	6	10	29	12	2	1	0

The durations and ratings for the different attractions are listed in table C.6.

Table C.6 Ratings and durations for attractions

ID	Rating	Duration (mins)
0	Starting hotel	
1	2	120
2	3	60
3	3	180
4	0	180
5	5	120
6	1	60
7	4	60
8	0	60
9	2	120

Define a suitable representation of a particle (i.e., a candidate solution), and carry out two hand iterations to show how to solve this problem using PSO, assuming a swarm size of 4.

5. *Trilateration* is used to identify the location of a moving object such as a connected vehicle or a cellphone. This process uses the distance between the vehicle and three or more known cell towers to determine the location of the vehicle. By measuring the signal strength of the device's signal at each tower, the distance between the device and each tower can be calculated. The intersection of the three (or more) circles created by these distance measurements gives an estimate of the device's location. As shown in figure C.32, three cell towers advertise their coordinates and transmit a reference signal.

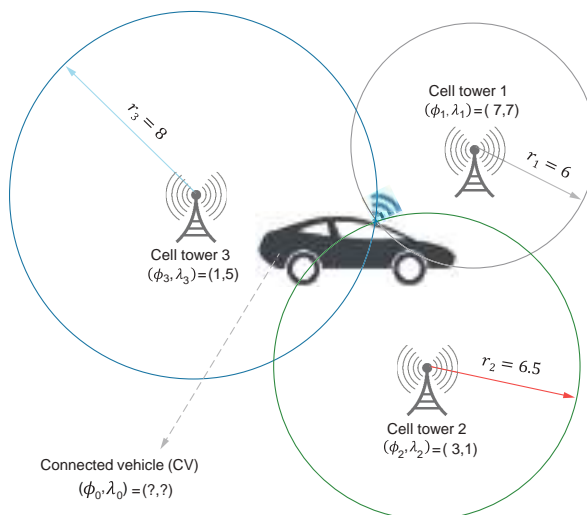


Figure C.32 Cell tower trilateration

The connected vehicle uses the reference signal to estimate distances to each of the cells r_i . These distance measurements r_i may be subject to noise. Assume that the associated error is given by the following equation:

$$f_i = r_i - \sqrt{(\phi_i - \phi_o)^2 + (\lambda_i - \lambda_o)^2}$$

The vehicle position (ϕ_o, λ_o) is the position that minimizes the following objective function:

$$f(\phi_o, \lambda_o) = \min \sum_{i=1}^3 f_i^2$$

Write Python code to find the vehicle's position using PSO.

6. A coffee shop offers two sizes of coffee: small and large. The cost of making a small coffee is \$1, and the cost of making a large coffee is \$1.50. The coffee shop sells small coffees for \$2 each and large coffees for \$3 each. The coffee shop wants to maximize its profit, but it also wants to ensure that it sells at least 50 small coffees and 75 large coffees and at most 300 and 450 respectively per day. This problem can be formulated as an optimization problem as follows:

$$\text{Maximize profit} = 2x_1 + 3x_2 - (x_1 + 1.5x_2)$$

Subject to

$$50 \leq x_1 \leq 300$$

$$75 \leq x_2 \leq 450$$

where

- x_1 is the number of small coffees to make.
- x_2 is the number of large coffees to make.

Profit is the total profit, where the first term in the equation represents the revenue from selling the coffees, and the second term represents the cost of making the coffees. Write Python code to find the optimal number of small coffees and large coffees to make every day.

7. A hospital wants to optimize the scheduling of its doctors to minimize the overall cost of labor while ensuring that enough doctors are available to meet patient demand. Each doctor has a different hourly rate, and there are different levels of patient demand at different times of the day. The goal is to find the optimal schedule that minimizes the total cost of labor while meeting the minimum doctor requirements during both peak and non-peak hours.

The hospital must schedule at least four doctors during peak hours and at least two doctors during non-peak hours. The hospital can hire part-time doctors at a lower hourly rate, but they can only work during non-peak hours. The hospital also has the option of paying overtime to full-time doctors to meet demand during peak hours, but at a higher hourly rate.

The problem can be mathematically described as follows:

$$\text{Minimize the total cost of labor: } f(x) = \sum (c_i \times x_i) + \sum (c_i^o \times x_i^o) + \sum (c_j^p \times x_j^p)$$

subject to the following constraints:

- $\sum x_i + \sum x_i^o \geq 4$: At least four doctors must be scheduled during peak hours.
- $\sum x_j^p \geq 2$: At least two doctors must be scheduled during non-peak hours.
- $x_i, x_i^o, x_j^p \geq 0$: Non-negativity constraints

where

- i is the index for full-time doctors, $i = 1, 2, \dots, m$.
- j is the index for part-time doctors, $j = 1, 2, \dots, n$.

Predefined parameters:

- c_i is the hourly rate for full-time doctor i .
- c_i^o is the overtime hourly rate for full-time doctor i .
- c_j^p is the hourly rate for part-time doctor j .

Decision variables:

- x_i is the number of hours worked by full-time doctor i during peak hours.
- x_i^o is the number of overtime hours worked by full-time doctor i during peak hours.
- x_j^p is the number of hours worked by part-time doctor j during non-peak hours.

Assume the full-time rates of full-time doctors are [30, 35, 40, 45, 50], their overtime rates are 1.5 times the full-time rates, and the part-time doctors' rates are [25, 27, 29, 31, 33]. Write Python code to solve this problem using PSO.

8. In neighborhood Y of city X , there are eight schools that collectively possess 100 microscopes for use in biology classes. These resources, however, are not uniformly distributed amongst the schools. With recent changes in student enrollment, four schools have more microscopes than needed, while the other four schools are in need of additional ones. To address this problem, Dr. Rachel Carson, who is in charge of the biology department at city X 's school board, decides to use a mathematical model. She chooses to use the transportation problem model, a strategy aimed at efficiently allocating resources while minimizing transportation costs as per R. Lovelace in his article "Open source tools for geographic analysis in transport planning" [7]. The model represents supply n and demand m as unit weights of decision variables at various points in a network, with the cost of transporting a unit from a supply point to a demand point

equivalent to the time or distance between nodes. This data is captured in an $n \times m$ cost matrix. The formal statement of this integer linear programming problem is described in Daskin's book "Network and Discrete Location: Models, Algorithms, and Applications" [8] as follows:

$$\text{Minimize } \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij}$$

subject to the following constraints

- $\sum_{j \in J} x_{ij} \leq S_i \forall i \in I$
- $\sum_{i \in I} x_{ij} \geq D_j \forall j \in J$
- $x_{ij} \geq 0 \forall i \in I \text{ and } \forall j \in J$

where

- i is each potential origin node.
- I is the complete set of potential origin nodes.
- j is each potential destination node.
- J is the complete set of potential nodes.
- x_{ij} is the amount to be shipped from $\forall i \in I$ to $\forall j \in J$.
- c_{ij} is per unit shipping costs between all i, j pairs.
- S_i is node i supply for $\forall i \in I$.
- D_j is node j demand for $\forall j \in J$.

Write Python code to solve this problem using PSO. Visualize the solution on a geospatial map.

9. In the Chapter 9/Projects folder in the book's GitHub repository, the following sample research projects are provided for review and experimentation.

- Routing—Addresses using PSO to find the shortest path between two points of interest in Toronto
- Bus stops placement—Addresses how to use PSO to find the optimal placement of bus stops in the Waterloo/Kitchener area, Ontario, Canada

C.8.2 Solutions

1. Multiple choice and true/false
 - 1.1. c) It may cause the algorithm to become stuck in local optima.
 - 1.2. a) True
 - 1.3. a) The cognitive component is based on the particle's own experience, while the social component is based on the experience of the swarm as a whole.
 - 1.4. b) False
 - 1.5. a) True

1.6. a) Cognitive-only model

1.7. b) False

1.8. b) To control the exploration/exploitation trade-off

1.9. b) False (PSO is designed to explore the search space globally and has the potential to find the global optimum.)

1.10. a) True

2. Imagine a group of delegates visiting an unfamiliar city for a conference. They are trying to find the best restaurant in town using PSO principles. The town is large, and each delegate starts at a different location. Each delegate has a preferred way of exploring restaurants, like walking along certain streets or visiting specific neighborhoods. This is similar to the inertia component in PSO, where particles maintain their current velocity and direction, ensuring they don't change their exploration pattern too abruptly.

As each delegate visits different restaurants, they remember the best one they've been to so far (their personal best). If they come across a less appealing restaurant, they're more likely to return to their favorite one, knowing it was a good choice based on their own experience. This is the cognitive component, where particles in PSO are attracted to their personal best positions, following their past experiences and individual preferences.

The delegates also communicate with each other via group chat, sharing their experiences and the locations of the best restaurants they've found. If someone discovers an outstanding restaurant, others might decide to visit that place and try it for themselves, even if it wasn't their personal favorite. This is the social component, where particles in PSO are influenced by the global best position or the collective knowledge of the swarm.

3. Due to the discrete nature of the formulated problem, permutation-based PSO should be used. A particle in this algorithm represents an ordering of the passengers to be picked up and dropped off by each driver. In this problem, we have two vehicles, A and B, and five passenger requests to match (P1–P5). For example, a candidate solution for a problem with five passengers and two drivers would have the following format:

A ⁺	P3 ⁺	P4 ⁺	P3 ⁻	P5 ⁺	P4 ⁻	P5 ⁻	A ⁻
B ⁺	P1 ⁺	P2 ⁺	P2 ⁻	P1 ⁻	B ⁻		

where

- + denotes the pickup points of passengers' requests and the vehicles' sources.
- - denotes the delivery points of passengers' requests and the vehicles' destinations

This solution can be read as follows:

- Vehicle A source point → passenger 3 pickup → passenger 4 pickup → passenger 3 drop-off → passenger 5 pickup → passenger 4 drop-off → passenger 5 drop-off → vehicle A destination.

- Vehicle B source point → passenger 1 pickup → passenger 1 pickup → passenger 2 drop-off → passenger 2 drop-off → vehicle B destination.

These two schedules can be also concatenated as follows:

A ⁺	P3 ⁺	P4 ⁺	P3 ⁻	P5 ⁺	P4 ⁻	P5 ⁻	A ⁻	B ⁺	P1 ⁺	P2 ⁺	P2 ⁻	P1 ⁻	B ⁻
----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------	----------------	----------------	-----------------	-----------------	-----------------	-----------------	----------------

We use the objective function to evaluate this solution as follows:

$$\begin{aligned}
 f &= \sum_{m \in V'} \sum_{i,j \in P} S_{i,j}^m d_{i,j} \\
 &= (d_{A^+P3^+} + d_{P3^+P4^+} + d_{P4^+P3^-} + d_{P3^-P5^+} + d_{P5^+P4^-} + d_{P4^-P5^-} + d_{P5^-A^-}) \\
 &\quad + (d_{B^+P1^+} + d_{P1^+P2^+} + d_{P2^+P2^-} + d_{P2^-P1^-} + d_{P1^-B^-})
 \end{aligned}$$

For a more comprehensive discussion of the ride-matching problem with time windows (RMPTW) and an extended version of this simplified problem, see Herbawi and Weber's article "A genetic and insertion heuristic algorithm for solving the dynamic ride-matching problem with time windows" [9].

4. Binary PSO (BPSO) is used to handle this problem. A binary string is used to describe the attractions to be visited. For example, [0 0 1 0 1 0 0 0] means visit attractions 3 and 5. The velocity is calculated using the following equation:

$$V_{k+1}^{id} = V_k^{id} + \phi_1 (pbest_k^{id} - x_k^{id}) + \phi_2 (gbest_k^{id} - x_k^{id})$$

where

- i is the particle number.
- d is the dimension or the attraction.
- v_k^{id} is the velocity at iteration k for particle i and dimension d .
- $pbest_k^{id}$ is the personal best at iteration t for particle i and dimension d .
- $gbest_k^d$ is the global best at iteration k for dimension d .
- x_k^{id} is the current position at iteration k for particle i and dimension d .
- Φ_1, Φ_2 are uniformly generated random numbers between 0 and 2.

Once the velocity vector is updated, the sigmoid value of each of the velocities is updated as follows:

$$sig(V_{k+1}^{id}) = \frac{1}{1 + e^{-V_{k+1}^{id}}}$$

A new position with the sigmoid value of the velocity is created. Next the particle position is updated as follows:

$$x_{k+1}^{id} = \begin{cases} 1, & \text{if } sig(V_{k+1}^{id}) > r \\ 0, & \text{otherwise} \end{cases}$$

where

- r is a uniformly generated random number between 0 and 1.
- $sig(v_{k+1}^{id})$ is the sigmoid value of the velocity at v_{k+1}^{id}

Table C.7 Initialization

	ϕ_1	ϕ_2	v_{01}	v_{02}	v_{03}	v_{04}	v_{05}	v_{06}	v_{07}	v_{08}	v_{09}
x_1	0	0	0	0	0	0	0	0	0	0	0
x_2	0	0	0	0	0	0	0	0	0	0	0
x_3	0	0	0	0	0	0	0	0	0	0	0
x_4	0	0	0	0	0	0	0	0	0	0	0

Let p_i be the current binary value for the position of attraction i for a particle as shown in table C.8.

Table C.8 Binary value for the positions of the 9 attractions

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9
x_1	1	1	0	0	0	0	0	0	0
x_2	0	0	1	0	0	0	0	1	0
x_3	0	1	0	0	1	0	0	0	0
x_4	1	0	0	0	0	0	1	0	0

Table C.9 Initialization: Fitness evaluation

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	Total commute time	Total places visited	Average rating	Total duration	$f(x_i)$
x_1	1	1	0	0	0	0	0	0	0	34	2	2.000	180	0.700
x_2	0	0	1	0	0	0	0	1	0	19	2	5.000	240	2.405
x_3	0	1	0	0	1	0	0	0	0	76	2	2.500	180	0.809
x_4	1	0	0	0	0	0	1	0	0	30	2	3.000	180	1.059

Table C.10 Particle's current personal best at initialization

	$pbest_{01}$	$pbest_{02}$	$pbest_{03}$	$pbest_{04}$	$pbest_{05}$	$pbest_{06}$	$pbest_{07}$	$pbest_{08}$	$pbest_{09}$	$pbestVal$
x_1	1	1	0	0	0	0	0	1	1	0.700
x_2	0	0	1	0	0	0	0	0	0	2.405
x_3	0	1	0	0	1	0	0	0	1	0.809
x_4	1	0	0	0	0	0	1	1	0	1.059

Table C.11 Iteration 1: Velocity update

	Φ_1	Φ_2	V_{11}	V_{12}	V_{13}	V_{14}	V_{15}	V_{16}	V_{17}	V_{18}	V_{19}
x_1	0.958	0.830	-0.830	-0.830	0.830	0.000	0.000	0.000	0.000	0.830	0.000
x_2	1.347	1.517	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
x_3	1.320	1.649	0.000	-1.649	1.649	0.000	-1.649	0.000	0.000	1.649	0.000
x_4	0.757	0.678	-0.678	0.000	0.678	0.000	0.000	0.000	-0.678	0.678	0.000

Table C.12 Updated sigmoid velocity values of the particle after iteration 1

	$\text{sig}(v_{11})$	$\text{sig}(v_{12})$	$\text{sig}(v_{13})$	$\text{sig}(v_{14})$	$\text{sig}(v_{15})$	$\text{sig}(v_{16})$	$\text{sig}(v_{17})$	$\text{sig}(v_{18})$	$\text{sig}(v_{19})$
x_1	0.304	0.304	0.696	0.500	0.500	0.500	0.500	0.696	0.500
x_2	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500	0.500
x_3	0.500	0.161	0.839	0.500	0.161	0.500	0.500	0.839	0.500
x_4	0.337	0.500	0.663	0.500	0.500	0.500	0.337	0.663	0.500

Table C.13 Uniformly generated random numbers for deciding particle updated after iteration 1

	r_{11}	r_{12}	r_{13}	r_{14}	r_{15}	r_{16}	r_{17}	r_{18}	r_{19}
x_1	0.477	0.724	0.875	0.654	0.088	0.089	0.853	0.925	0.528
x_2	0.673	0.530	0.438	0.785	0.218	0.763	0.838	0.749	0.590
x_3	0.534	0.086	0.301	0.763	0.653	0.754	0.809	0.974	0.763
x_4	0.218	0.697	0.875	0.854	0.116	0.941	0.678	0.742	0.965

Iteration 1: Global best is 2,405, and the best particle is $x_2 = [0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0]$.

Iteration 2: Velocity update

Table C.14 Updated particle state and fitness function after iteration 2

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	Total commute time	Total places visited	Average rating	Total duration	$f(x_i)$
x_1	0	0	0	0	1	1	0	0	0	64	2	3.500	180	1.158
x_2	0	0	1	0	1	0	0	0	0	37	2	5.000	300	2.901
x_3	0	1	1	0	0	0	0	0	0	70	2	2.500	240	1.091
x_4	1	0	0	0	1	0	0	0	0	24	2	4.500	240	2.143

Table C.15 Updated personal best of the particle after iteration 2

	pbest_{11}	pbest_{12}	pbest_{13}	pbest_{14}	pbest_{15}	pbest_{16}	pbest_{17}	pbest_{18}	pbest_{19}	pbestVal
x_1	0	0	0	0	1	1	0	0	0	1.158
x_2	0	0	1	0	1	0	0	0	0	2.901
x_3	0	1	1	0	0	0	0	0	0	1.091
x_4	1	0	0	0	1	0	0	0	0	2.143

The global best is 2.901, and the best particle is $x_2 = [0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0]$. Repeat until the stopping criteria is fulfilled.

5. The next listing shows the steps for solving the trilateration problem using PSO.

Listing C.13 Solving the trilateration problem using PSO

```
import numpy as np
from pyswarms.single import GlobalBestPSO
import pyswarms as ps

def objective_function(pos):
    r = np.array([6, 6.5, 8])
    x0 = np.array([1, 3, 7])
    y0 = np.array([5, 1, 7])

    x = pos[:, 0]
    y = pos[:, 1]

    f = np.sum(r - np.sqrt((x0 - x.reshape(-1, 1)) ** 2 + (y0 - y.reshape(-
    1, 1)) ** 2), axis=1)
    return f

options = {"c1": 0.5, "c2": 0.5, "w": 0.79}
bounds = (np.array([0, 0]), np.array([8, 8]))

optimizer = GlobalBestPSO(n_particles=100, dimensions=2, options=options,
    bounds=bounds)
best_cost, best_position = optimizer.optimize(objective_function,
    iters=1000)

print("Best position:", np.round(best_position, 2))
print("Best cost:", np.round(best_cost, 3))
```

Define the objective function.

Set up the PSO algorithm.

Initialize GlobalBestPSO, and minimize the objective function.

Print the results.

6. The next listing shows the steps for solving the coffee shop problem.

Listing C.14 Solving the coffee shop problem using PSO

```
import numpy as np
import pyswarms as ps

def fitness_function(x):
    n_particles = x.shape[0]
    profit=0
    for i in range(n_particles):
        profit+=(2*x[i][0] + 3*x[i][1] - (x[i][0] + 1.5*x[i][1]))
    return profit

num_particles = 10
num_iterations = 100

lb = np.array([50, 75])
ub = np.array([300, 450])
bounds = (lb, ub)
```

Import the required libraries.

Define the fitness function.

Set the number of particles and iterations.

Set the lower and upper bounds of the variables.

```

options={'w':0.79, 'c1': 1.5, 'c2': 1.3}  ← Set the optimizer options.

optimizer = ps.single.GlobalBestPSO(n_particles=num_particles,
➡ dimensions=2, options=options, bounds=bounds)  ← Initialize the optimizer.

best_cost, best_pos = optimizer.optimize(fitness_function,
➡ iters=num_iterations)  ← Perform the optimization.

best_pos=np.asarray(best_pos, dtype = 'int')
print('#####')
print('Total profit: ', round(-best_cost, 2), '$')
print('Optimal number of small coffees to make: ',best_pos[0])
print('Optimal number of large coffees to make: ', best_pos[1])

```

7. The next listing shows the steps for solving the doctor scheduling problem.

Listing C.15 Solving the doctor scheduling problem using PSO

```

import matplotlib.pyplot as plt
import numpy as np
import pyswarms as ps

full_time_rates = np.array([30, 35, 40, 45, 50])
overtime_rates = full_time_rates * 1.5
part_time_rates = np.array([25, 27, 29, 31, 33])  ← Set full-time hourly rates, overtime
                                                    rates, and part-time doctor rates.

def objective_function(x):
    x1 = x[:, :len(full_time_rates)]
    x2 = x[:, len(full_time_rates):2 * len(full_time_rates)]
    x3 = x[:, 2 * len(full_time_rates):]

    total_cost = np.sum(x1 * full_time_rates) + np.sum(x2 * overtime_rates)
    ➡ + np.sum(x3 * part_time_rates)  ← Define the total cost as
                                    an objective function.

    penalty_x1_x2 = np.maximum(0, 4 - np.sum(x1) - np.sum(x2)) * 10000
    penalty_x3 = np.maximum(0, 2 - np.sum(x3)) * 10000

    total_cost_with_penalty = total_cost + penalty_x1_x2 + penalty_x3

    return total_cost_with_penalty

min_bound = np.zeros(3 * len(full_time_rates))
max_bound = np.full(3 * len(full_time_rates), 10)
bounds = (min_bound, max_bound)  ← Initialize the bounds.

options = {'w': 0.74, 'c1': 2.05, 'c2': 2.05, 'k': 5, 'p': 1}  ← Set the options
                                                                    for PSO.

optimizer = ps.single.LocalBestPSO(n_particles=30, dimensions=3 *
➡ len(full_time_rates), options=options, bounds=bounds)  ← Create an instance of PSO.

cost, pos = optimizer.optimize(objective_function, iters=100, verbose=True)  ← Perform the optimization.

optimal_x1 = pos[:len(full_time_rates)]
optimal_x2 = pos[len(full_time_rates):2 * len(full_time_rates)]
optimal_x3 = pos[2 * len(full_time_rates):]

```

Define the three decision variables.

Define the constraints and add them as a penalty in the cost function.

Extract optimal values of the decision variables x1, x2, and x3.

The complete version of listing C.15, available in the book's GitHub repo, contains a function to print and visualize the results. Figure C.33 shows the hours worked for each doctor.

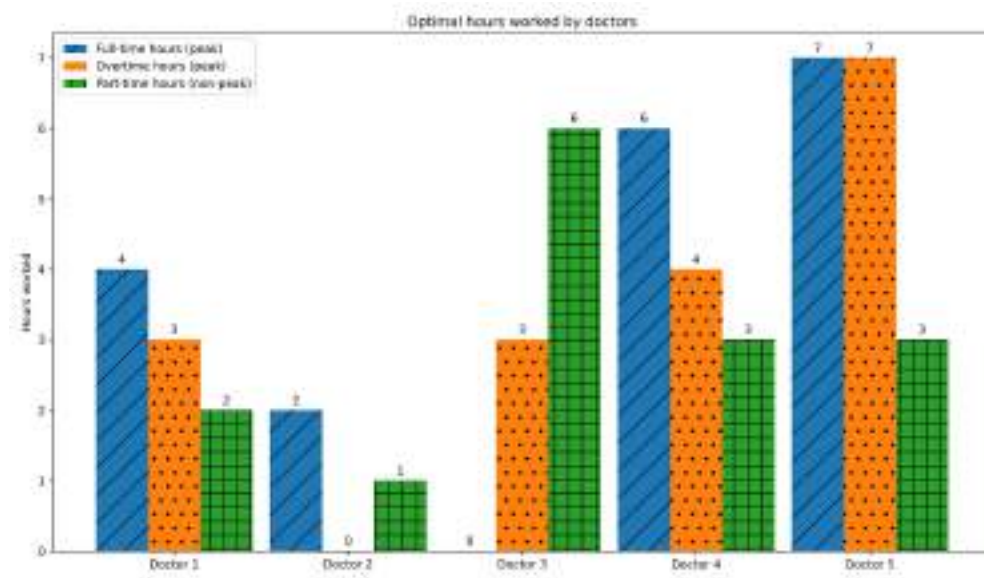


Figure C.33 Hours worked by each doctor

8. The next listing shows the steps for solving the supply chain optimization problem using PSO. We start by importing the required libraries.

Listing C.16 Solving the supply chain optimization problem using PSO

```
import numpy as np
import random
from pymoo.algorithms.soo.nonconvex.pso import PSO
from pymoo.optimize import minimize
from pymoo.algorithms.soo.nonconvex.ga import GA
from pymoo.operators.crossover.pntx import PointCrossover
from pymoo.operators.mutation.pm import PolynomialMutation
from pymoo.operators.repair.rounding import RoundingRepair
from pymoo.operators.sampling.rnd import FloatRandomSampling
from pymoo.core.problem import Problem
from geopy.geocoders import Nominatim
from geopy.distance import geodesic
import folium
```

We then define the problem data as follows:

Schools with microscopes available	→	supply_schools = [1, 6, 7, 8]	Schools with microscopes requested
		demand_schools = [2, 3, 4, 5]	
		amount_supply = [20, 30, 15, 35]	Number of microscopes available at each school

```

amount_demand = [5, 45, 10, 40]
n_var=len(supply_schools)*len(demand_schools)

```

Number of variables

Number of microscopes requested at each school

As a continuation of listing C.16, the following function generates random locations around a center point to represent the supply and demand schools in the selected city (using Toronto as an example). We calculate the distances between the schools using geopy:

```

np.random.seed(0)

def generate_random(number, center_point):
    lat, lon = center_point
    coords = [(random.uniform(lat - 0.01, lat + 0.01), random.uniform(lon -
    0.01, lon + 0.01)) for _ in range(number)]
    return coords

supply_coords = generate_random(len(supply_schools), [location.latitude,
location.longitude])
demand_coords = generate_random(len(demand_schools), [location.latitude,
location.longitude])

distances = []
for supply in supply_coords:
    row = []
    for demand in demand_coords:
        row.append(geodesic(supply, demand).kilometers)
    distances.append(row)
distances = np.array(distances)

cost_matrix=50*distances

```

For reproducibility

Set the center of the map.

Generate random GPS coordinates (lat, long) for the supply schools.

Generate random GPS coordinates (lat, long) for the demand schools.

Calculate geodesic distances between the schools in km.

The following class defines the transportation problem in a format compatible with pymoo. It defines the decision variables, constraints, and the objective function:

```

class Transportation_problem(Problem):
    def __init__(self,
                 cost_matrix,
                 amount_supply,
                 amount_demand
                 ):
        super().__init__(n_var=n_var,n_constr=1, vtype=int)
        self.cost_matrix = cost_matrix
        self.amount_supply = amount_supply
        self.amount_demand = amount_demand
        self.xl = np.array(np.zeros(n_var))
        self.xu = np.repeat(amount_supply, len(amount_demand))

    def _evaluate(self, X, out, *args, **kwargs):
        loss = np.zeros((X.shape[0], 1))
        g = np.zeros((X.shape[0], 1))
        for i in range(len(X)):
            soln = X[i].reshape(self.cost_matrix.shape)
            cost_x = X[i].reshape(self.cost_matrix.shape)
            cost = cost_x * cost_matrix.T

```

```

cost = cost.sum()
loss[i] = cost
total_supply = soln.sum(axis=1)
total_demand = soln.sum(axis=0)
print("total_supply: ", total_supply)
print("total_demand: ", total_demand)
g[i] = np.any(total_supply<self.amount_supply) or
➡ np.any(total_demand<self.amount_demand)
out["F"] = loss
out["G"] = g

```

Now we can create a problem object and a PSO solver as follows:

```

problem = Transportation_problem(cost_matrix,amount_supply,amount_demand)
algorithm = PSO(pop_size=100,repair=RoundingRepair())

```

The following line is used to run 150 iterations of the PSO solver:

```

res = minimize(problem, algorithm, ('n_gen', 150), seed=1, verbose=False)

```

The following code snippet is used to print the solution obtained by the PSO solver:

```

soln = res.X  ← Extract the solution.

supply_num=len(amount_supply)  ← Number of supply points
demand_num=len(amount_demand)  ← Number of demand points

for i in range(supply_num):  ← Print each supply point.
    print(f"Supply School({supply_schools[i]}): {' + [CA]'.
join(['soln['+str(j)+']' for j in range(i*supply_num,
➡ (i+1)*supply_num)])} <= {amount_supply[i]} or {' + '.join(map(str,
➡ soln[i*supply_num:(i+1)*supply_num])}] <= {amount_supply[i]} or
➡ {sum(soln[i*supply_num:(i+1)*supply_num])} <= {amount_supply[i]}")

for j in range(demand_num):  ← Print each demand point.
    print(f"Demand School({demand_schools[j]}): {' +
➡ '.join(['soln['+str(i*demand_num+j)+']' for i in range(demand_num)])}
➡ >= {amount_demand[j]} or {' + '.join(map(str, [soln[i*4+j] for i in
➡ range(demand_num)]))} >= {sum(soln[i*demand_num+j] for i in
➡ range(demand_num))} or {sum(soln[i*demand_num+j] for i in
➡ range(demand_num))} >= {amount_demand[j]}")

print(f"Shipping cost = {round(res.F[0], 2)} $")  ← Print the shipping cost.

```

Given the randomness included in the implementation, the code will produce output that is something like the following:

```

Supply School(1): soln[0] + soln[1] + soln[2] + soln[3] <= 20 or 1 + 0 + 14 +
5 <= 20 or 20 <= 20
Supply School(6): soln[4] + soln[5] + soln[6] + soln[7] <= 30 or 0 + 30 + 0 +
0 <= 30 or 30 <= 30

```

```

Supply School(7): soln[8] + soln[9] + soln[10] + soln[11] <= 15 or 0 + 15 + 0
+ 0 <= 15 or 15 <= 15
Supply School(8): soln[12] + soln[13] + soln[14] + soln[15] <= 35 or 4 + 0 +
0 + 35 <= 35 or 39 <= 35
Demand School(2): soln[0] + soln[4] + soln[8] + soln[12] >= 5 or 1 + 0 + 0 +
4 >= 5 or 5 >= 5
Demand School(3): soln[1] + soln[5] + soln[9] + soln[13] >= 45 or 0 + 30 + 15
+ 0 >= 45 or 45 >= 45
Demand School(4): soln[2] + soln[6] + soln[10] + soln[14] >= 10 or 14 + 0 + 0
+ 0 >= 14 or 14 >= 10
Demand School(5): soln[3] + soln[7] + soln[11] + soln[15] >= 40 or 5 + 0 + 0
+ 35 >= 40 or 40 >= 40
Shipping cost = 3053.94 $

```

The following code snippet can be used to visualize the solution of a spatial map using folium:

```

def normalize(lst):  ← Normalize function
    s = sum(lst)
    return list(map(lambda x: (x/s)*10, lst))

soln_normalized = normalize(soln)  ← Normalize soln array

colors = ['cyan', 'brown', 'orange', 'purple']  ← Define a color list.

m = folium.Map(location=[location.latitude, location.longitude],
    ↪ zoom_start=15, scrollWheelZoom=False, dragging=False)  ← Create a map
                                                             centered at
                                                             downtown
                                                             Toronto.

for i, coord in zip(supply_schools, supply_coords):
    folium.Marker(location=coord, icon=folium.Icon(icon="home",
    ↪ color='red'), popup=f'Supply School {i+1}').add_to(m)  ← Add markers for
                                                             supply schools.

for i, coord in zip(demand_schools, demand_coords):
    folium.Marker(location=coord, icon=folium.Icon(icon="flag",
    ↪ color='blue'), popup=f'Demand School {i+1}').add_to(m)

    ↪ Add markers
    ↪ for demand
    ↪ schools.

for i in range(len(supply_schools)):  ← Add lines (edges) between
    for j in range(len(demand_schools)):
        soln_value = soln[i*len(demand_schools) + j]
        folium.PolyLine(locations=[supply_coords[i], demand_coords[j]],
        ↪ color=colors[i % len(colors)],
        ↪ weight=5*soln_normalized[i*len(demand_schools) + j],
        ↪ popup=folium.Popup(f'# of microscopes: {soln_value}')).add_to(m)

m  ← Show the map.

```

Figure C.34 shows a solution that may be produced by PSO.

Print
the results.

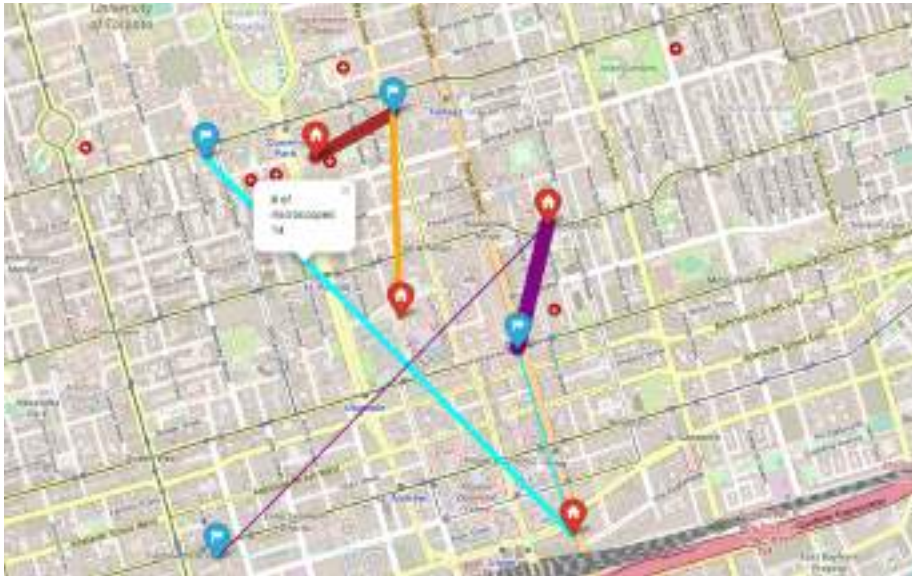


Figure C.34 School microscopes supply and demand

The complete version of listing C.16, available in the book's [GitHub repo](#), also shows how to solve this problem using the genetic algorithm.

C.9 Chapter 10: Other swarm intelligence algorithms to explore

C.9.1 Exercises

1. Match the terms and descriptions shown in table C.16.

Table C.16

Terms	Descriptions
1. Ant colony system (ACS)	a. Bees that search for new food sources after exhausting their current ones
2. Stigmergy	b. Positive feedback about food path causes that path to be followed by more and more ants
3. Scout bees	c. A pheromone update method that doesn't take the desirability of the solution into account
4. Max-min ant system (MMAS)	d. The maximum number of unsuccessful attempts made by a scout bee to find a new food source
5. Autocatalytic behavior	e. Bees that probabilistically choose food sources depending on the fitness of the solutions found by the employed bees

Table C.16 (continued)

6. Ant density model	f. Indirect communication among social insects through environmental modifications that serve as external memory
7. Onlooker bees	g. A pheromone update method that uses local information to update the pheromone concentrations
8. Ant-cycle	h. An ACO variation that uses an elitist strategy called pseudo-random proportional action rule
9. Ant System (AS)	i. An ACO variant that adds a memory capability by including a tabu list
10. Trial limit	k. An ACO variation that overcomes stagnation

2. Write Python code to find the shortest path between a source point and a destination point using the ant colony optimization algorithm. Assume that you're currently standing at the King Edward VII equestrian statue in the city of Toronto with GPS coordinates (43.664527, -79.392442). Imagine you're a student at the University of Toronto, and you're already running late for your Optimization Algorithms lecture at the Bahen Centre for Information Technology with GPS coordinates (43.659659, -79.397669). Visualize the obtained route on a map centered on King's College Circle with GPS coordinates (43.661667, -79.395) so you can reach your destination. Feel free to use the helping functions available in the `optalgotools` package, such as `Node`, `cost`, and `draw_route`. Use the code to experiment with different search spaces (different areas of interest, different origins, and destinations) and different algorithm parameter settings.

3. Reverse osmosis (RO) is a very effective and important process for desalination and water waste reclamation. Assume that you need to maximize the RO high pressure pump power. This power depends of a number of parameters according to the following equation:

$$HP = \frac{M_d^2 + 1200 \times M_d \times N_v \times \Delta\pi}{3600 \times N_v \times RR \times \eta \times \rho}$$

where

- HP is the RO high pressure pump power in kW.
- M_d is the RO productivity in m^3/d and is in the range of $41.67 < M_d < 416.67 \text{ m}^3/\text{d}$.
- N_v is number of pressure vessels and is ranged as $1 < N_v < 200$.
- $\Delta\pi$ is the net osmotic pressure across the membrane and is in the range $1400 < \Delta\pi < 2600 \text{ kPa}$.
- RR is the recovery ratio and is ranged as $1 < RR < 50\%$.
- η is the efficiency of the high pressure pump and is in the range of $0.70 < \eta < 0.85$.
- ρ is the density of water.

Write Python code to find the optimal values of the decision variables (Md , Nv , $\Delta\pi$, RR , η , and ρ) to maximize HP using the ant colony optimization (ACO) algorithm. Solve the same problem using the `optimize` solver in SciPy.

4. Solve the supply and demand problem introduced in exercise 8 of chapter 9 using the ant colony optimization (ACO) algorithm. Use mixed integer distributed ant colony optimization (MIDACO) to solve this problem. MIDACO (www.midaco-solver.com) is a numerical high-performance software for solving single- and multi-objective optimization problems. It is based on ACO with an extension for mixed integer search domains.

To install MIDACO, follow these steps:

- 1 Download MIDACO Python Gateway (`midaco.py`), and remove the `.txt` extension.
- 2 Download the appropriate library file (`midacopy.dll` or `midacopy.so`).
- 3 Store all the files in the same folder on your PC.
- 4 Import `midaco` into your Jupyter notebook.

MIDACO is licensed software with a limited free license that enables optimization with up to four variables (two supply schools and two demand schools).

Assume the following reduced problem data:

<code>supply_schools = [1, 3]</code>	←	Schools with microscopes available
<code>demand_schools = [2, 4]</code>	←	Schools with microscopes requested
<code>amount_supply = [20, 30]</code>	←	Number of microscopes available at each school
<code>amount_demand = [5, 45]</code>	←	Number of microscopes requested at each school

Write Python code to solve this problem using MIDACO. Visualize the solution on a geospatial map. For a larger number of schools, you can obtain an unlimited license or request a free academic trial of the unlimited version that can support up to 100,000 variables.

C.9.2 Solutions

1. Answers

1-h, 2-f, 3-a, 4-k, 5-b, 6-c, 7-e, 8-g, 9-i, 10-d

2. Listing C.17 shows the steps to find and visualize the shortest path between two points of interest in the city of Toronto. The code uses helping functions from `optalgotools` such as `Node`, `cost`, and `draw_route`. We start by defining and visualizing the search space as follows.

Listing C.17 Solving a routing problem using ACO

```
import osmnx as ox
from optalgotools.structures import Node
from optalgotools.routing import cost, draw_route
import random
from tqdm.notebook import tqdm
import matplotlib.pyplot as plt
```

```

reference = (43.661667, -79.395)
G = ox.graph_from_point(reference, dist=300, clean_periphery=True,
    ➔ simplify=True)
origin = (43.664527, -79.392442)
destination = (43.659659, -79.397669)

origin_id = ox.distance.nearest_nodes(G, origin[1], origin[0])
destination_id = ox.distance.nearest_nodes(G, destination[1],
    ➔ destination[0])

origin_node = Node(graph=G, osmid=origin_id)
destination_node = Node(graph=G, osmid=destination_id)

highlighted = [destination_id, origin_id]

nc = ['red' if node in highlighted else '#336699' for node in G.nodes()]
ns = [70 if node in highlighted else 8 for node in G.nodes()]
fig, ax = ox.plot_graph(G, node_size=ns, node_color=nc, node_zorder=2)

```

Set King's College Circle, Toronto, ON as the center of the map.

Create a graph.

Set the King Edward VII equestrian statue as the origin.

Set the Bahen Centre for Information Technology at University of Toronto as the destination.

Get the osmid of the nearest nodes to the origin and destination points.

Create the origin and destination nodes.

Mark both the source and destination nodes.

Visualize the search space.

AS a continuation of listing C.17, we can initialize the parameters of the ACO algorithm as follows:

```

alpha = 1
beta = 1
n = 500
Q = 1

pheremone_concentrations = dict()
known_routes = dict()

pheremone_concentrations = {(u,v):random.uniform(0,0.5) for [u,v] in
    ➔ G.edges()}

def pheremone(level, distance, alpha, beta):
    return level ** alpha * ((1/distance)) ** beta

```

Set alpha, a parameter to control the influence of pheromones.

Set beta, a parameter to control the influence of desirability of the city transition.

Randomize the pheromones.

Randomize the pheromones.

A function to calculate pheromone levels

We now implement the ACO procedure as follows:

```

for ant in tqdm(range(n)):
    frontier = [origin_node]
    explored = set()
    route = []
    found = False

    while frontier and not found:
        parent = frontier.pop(0)
        explored.add(parent)

        children = []
        children_pheremones = []

```

Place the ant at the origin node.

```

for child in parent.expand():
    if child == destination_node:
        found = True
        route = child.path()
        continue
    if child not in explored:
        children.append(child)
        children_pheromones.append(
            pheromone(
                pheromone_concentrations[(parent.osmid,
➡ child.osmid)],
                child.distance,
                alpha,
                beta,
            )
        )

    if len(children) == 0:
        continue

    transition_probability = [
        children_pheromones[i] / sum(children_pheromones)
        for i in range(len(children_pheromones))
    ]

    chosen = random.choices(children, weights=transition_probability,
➡ k=1)[0] ← Probabilistically choose a child to explore.

    children.pop(children.index(chosen))
    frontier.extend(children)
    frontier.insert(0, chosen)
    ← Add all the non-explored children in case we need to explore them later.
    ← Set the chosen child to be the next node to explore.

    for u, v in zip(route[:-1], route[1:]):
        length_of_edge = G[u][v][0]['length']
        pheromone_concentrations[(u,v)] += Q/length_of_edge
        ← Update the pheromones.

    route = tuple(route)
    if route in known_routes:
        known_routes[route] += 1
        ← If the route is newly discovered, add it to the list.
    else:
        known_routes[route] = 1

```

You can now print the best route and its cost and visualize the obtained route as follows:

```

best_route = max(known_routes, key=known_routes.get)
times_used = known_routes[best_route]
route = list(best_route)
print("Cost:", cost(G, route))
print("Times used:", times_used)
draw_route(G, route)

```

Figure C.35 shows the optimal route generated by ACO.

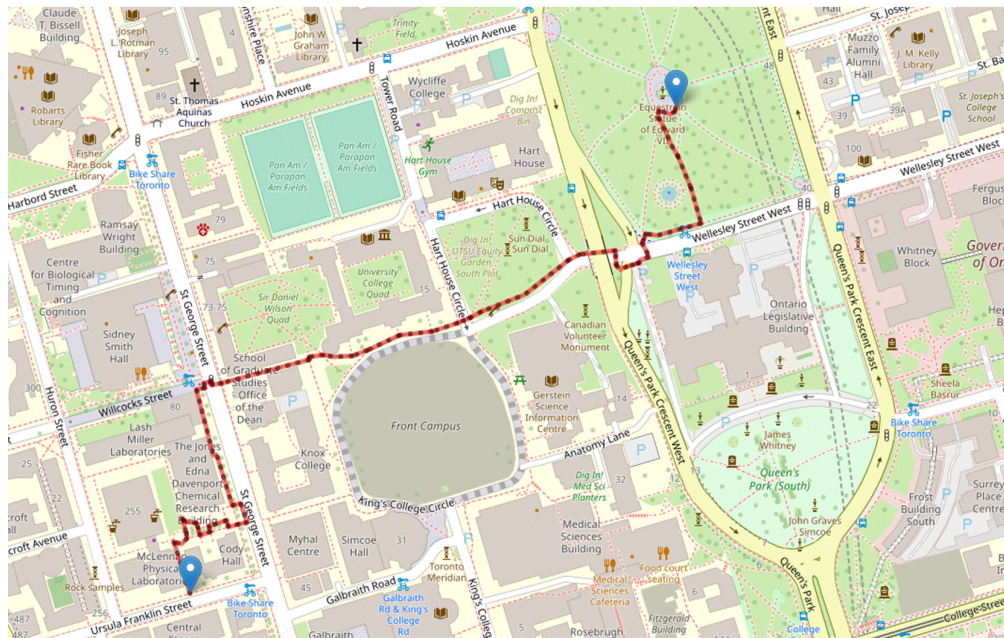


Figure C.35 Optimal route generated by ACO

3. The next listing shows the steps for solving the reverse osmosis (RO) high pressure pump power maximization problem using ACO.

Listing C.18 Solving the RO high pressure pump power maximization problem with ACO

```
import random

md_range = (41.67, 416.67)
nv_range = (1, 200)
delta_range = (1400, 2600)
rr_range = (1, 50)
eta_range = (0.70, 0.85)
rho = 1000

def ro_pump_power(X):
    md, nv, delta, rr, eta = X
    return (md ** 2 + 1200 * md * nv * delta) / (nv * rr * 3600 * eta * rho)

num_ants = 100
num_iterations = 300
evaporation_rate = 0.7
pheromone_deposit = 1
initial_pheromone = 0.25

pheromone_matrix = [[initial_pheromone] * 5 for _ in range(num_ants)]
```

Define the range for each decision variable.

Density of water in kg/m^3

Define the objective function.

Define the ACO parameters.

Initialize the pheromone matrix.

```

best_solution = None
best_power = float('-inf')

for _ in range(num_iterations):
    solutions = []
    powers = []

    for ant in range(num_ants):
        md = random.uniform(md_range[0], md_range[1])
        nv = random.uniform(nv_range[0], nv_range[1])
        delta = random.uniform(delta_range[0], delta_range[1])
        rr = random.uniform(rr_range[0], rr_range[1])
        eta = random.uniform(eta_range[0], eta_range[1])

        soln=(md, nv, delta, rr, eta)
        power = ro_pump_power(soln)

        solutions.append((md, nv, delta, rr, eta))
        powers.append(power)

        if power > best_power:
            best_solution = (md, nv, delta, rr, eta)
            best_power = power

    for ant in range(num_ants):
        for i in range(5):
            pheromone_matrix[ant][i] *= evaporation_rate
            if solutions[ant][i] == best_solution[i]:
                pheromone_matrix[ant][i] += pheromone_deposit / powers[ant]

print("Optimal Solution:")
print("Md:", format(best_solution[0], '.2f'))
print("Nv:", format(best_solution[1], '.2f'))
print("Delta:", format(best_solution[2], '.2f'))
print("RR:", format(best_solution[3], '.2f'))
print("Eta:", format(best_solution[4], '.2f'))
print("Optimal HP:", format(best_power, '.2f'))

```

Initialize the best solution and its corresponding power.

Construct solutions for each ant.

Calculate the power for the current solution.

Store the solution and its power.

Update the best solution if necessary.

Update pheromone trails based on the power of each solution.

Print the optimal values of the decision variables and the optimal HP.

An example of the generated output is show here:

```

Optimal Solution:
Md: 404.10
Nv: 7.39
Delta: 2536.93
RR: 1.05
Eta: 0.76
Optimal HP: 425.75

```

As a continuation of listing C.18, the following code snippet shows the steps for solving this problem using the optimize solver in SciPy.

```

Import the scipy optimizer. from scipy.optimize import minimize

def ro_pump_power(X):
    md, nv, delta, rr, eta=X

```

Define the objective function with a negative sign for minimization, as per scipy's requirement.

```

    return -(md ** 2 + 1200 * md * nv * delta) / (nv * rr * 3600 * eta *
    rho)

```

Set the bounds for the decision variables.

```

bounds = [md_range, nv_range, delta_range, rr_range, eta_range]
x0=[200, 100, 2000, 25, 0.75]

```

Set an initial guess.

```

result = minimize(ro_pump_power, x0, bounds=bounds, method='SLSQP')

```

Solve the optimization problem using sequential least squares programming (SLSQP).

```

print("Optimal Solution:")
print("Md:", format(result.x[0], '.2f'))
print("Nv:", format(result.x[1], '.2f'))
print("Delta:", format(result.x[2], '.2f'))
print("RR:", format(result.x[3], '.2f'))
print("Eta:", format(result.x[4], '.2f'))
print("Optimal HP:", format(-result.fun, '.2f'))

```

Print the optimal values of the decision variables and the optimal HP.

The generated output is show here:

```

Optimal Solution:
Md: 416.67
Nv: 99.98
Delta: 2600.00
RR: 1.00
Eta: 0.70
Optimal HP: 515.88

```

You can fine-tune the ACO parameters to get results comparable to those of the SciPy optimizer.

4. The next listing shows the steps for solving the supply and demand problem using the MIDACO solver. We start by importing the libraries and setting the problem data.

Listing C.19 Solving the supply and demand problem using MIDACO

```

import numpy as np
import random
import midaco
from geopy.geocoders import Nominatim
from geopy.distance import geodesic
import folium

```

Import the MIDACO solver.

```

supply_schools = [1, 3]
demand_schools = [2, 4]
amount_supply = [20, 30]
amount_demand = [5, 45]
n_var=len(supply_schools)*len(demand_schools)
n_constr=len(supply_schools)

```

Set the schools with microscopes available.

Set the schools with microscopes requested.

Set the number of microscopes available at each school.

Set the number of microscopes requested at each school.

As a continuation, we'll generate random GPS coordinates for the supply and demand schools:

```

geolocator = Nominatim(user_agent="SupplyDemand")
location = geolocator.geocode("Toronto, Ontario")
def generate_random(number, center_point):

```

Create a geolocator object.

Get the coordinates of Toronto.

Function to generate random locations around a center point

```

lat, lon = center_point
coords = [(random.uniform(lat - 0.01, lat + 0.01), random.uniform(lon -
➡ 0.01, lon + 0.01)) for _ in range(number)]
return coords

np.random.seed(0)

```

Set the random seed to ensure reproducibility of random numbers generated.

Generate random GPS coordinates (lat, long) for the supply schools.

```

supply_coords = generate_random(len(supply_schools),
➡ [location.latitude, location.longitude])
demand_coords = generate_random(len(demand_schools),
➡ [location.latitude, location.longitude])

```

Generate random GPS coordinates (lat, long) for the demand schools.

```

distances = []
for supply in supply_coords:
    row = []
    for demand in demand_coords:
        row.append(geodesic(supply, demand).kilometers)
    distances.append(row)
distances = np.array(distances)

```

Calculate geodesic distances between the schools in km.

```

cost_matrix=50*distances

```

Calculate the cost matrix.

The following function defines the main ingredients of the optimization problem, including the objective function and the constraints.

```
def problem_function(x):
```

```

    f = [0.0]*1
    g = [0.0]*n_constr
    f[0] = np.sum(np.multiply(cost_matrix.flatten(), x))
    soln=np.reshape(x, (len(supply_schools), len(demand_schools)))
    total_supply = soln.sum(axis=1)
    total_demand = soln.sum(axis=0)
    g[0] = (np.all(total_supply>=amount_supply) and np.all(total_
demand>=amount_demand)) -1
    return f,g

```

Objective functions F(X)

Initialize the array for objectives F(X).

Initialize the array for constraints G(X).

Candidate solution

Inequality constraints G(X) ≥ 0 must come second in g[me:m-1]

Return the objective function and constraint evaluation.

The first step in using MIDACO is to define the problem as follows:

Free limited license that supports up to 4 variables.

```

key = b'MIDACO_LIMITED_VERSION__[CREATIVE_COMMONS_BY-NC-ND_LICENSE]'
problem = {}
option = {}
problem['@'] = problem_function

```

Initialize the dictionary containing the problem specifications.

Initialize the dictionary containing the MIDACO options.

Handle for problem function name.

The problem dimensions are defined as follows:


```

Number of objectives → problem['o'] = 1
                      problem['n'] = n_var ← Number of variables (in total)
                      problem['ni'] = n_var ← Number of integer variables (0 ≤ ni ≤ n)
                      problem['m'] = n_constr ← Number of constraints (in total)
                      problem['me'] = 0 ← Number of equality constraints (0 ≤ me ≤ m)

```

The lower and upper bounds x_l and x_u are defined as follows:

```

problem['xl'] = [0.0]*n_var
problem['xu'] = [30.0]*n_var

```

The starting point x is set as the lower bound:

```

problem['x'] = problem['xl']

```

The stopping criteria are defined as follows:

```

option['maxeval'] = 10000 ← Maximum number of function evaluation
option['maxtime'] = 60*60*24 ← Maximum time limit in seconds (e.g., 1 day = 60*60*24)

```

We set the printing options as follows:

```

option['printeval'] = 1000 ← Print-frequency for current best solution
option['save2file'] = 1 ← Save screen and solution [0 for no and 1 for yes]

```

MIDACO offers the option to evaluate multiple solution candidates in parallel (aka co-evaluation or fine-grained parallelization). According to the MIDACO user manual, for a parallelization factor of $P=10$, the potential speed-up is around 10 times, while for a parallelization factor of $P=100$, the potential speed up is around 70 times. We can set the parallelization factor as follows:

```

option['parallel'] = 0 ← Serial: 0 or 1, Parallel: 2,3,4,5,6,7,8...

```

We can now run midaco to solve the problem:

```

solution = midaco.run( problem, option, key)

```

The following code snippet is used to print the solution obtained by the PSO solver:

```

soln = solution['x']
supply_num=len(amount_supply) ← Number of supply points
demand_num=len(amount_demand) ← Number of demand points

for i in range(supply_num): ← Print each supply point.
    print(f"Supply School({supply_schools[i]}): {' +
    ➤ '.join(['soln['+str(j)+']' for j in range(i*supply_num,
    ➤ (i+1)*supply_num)])} <= {amount_supply[i]} or {' + '.join(map(str,
    ➤ soln[i*supply_num:(i+1)*supply_num]))} <= {amount_supply[i]} or
    ➤ {sum(soln[i*supply_num:(i+1)*supply_num])} <= {amount_supply[i]}")

for j in range(demand_num): ← Print each demand point.
    print(f"Demand School({demand_schools[j]}): {' +
    ➤ '.join(['soln['+str(i*demand_num+j)+']' for i in range(demand_num)])} >=
    ➤ {amount_demand[j]} or {' + '.join(map(str, [soln[i*demand_num+j] for i
    ➤ in range(demand_num)]))} >= {sum(soln[i*demand_num+j] for i in

```

```

➡ range(demand_num))} or {sum(soln[i*demand_num+j] for i in
➡ range(demand_num))} >= {amount_demand[j]}")

print(f"Shipping cost = {solution['f']} $") ← | Print the shipping cost.

```

The code will produce output like the following:

```

Supply School(1): soln[0] + soln[1] <= 20 or 0.0 + 30.0 <= 20 or 30.0 <= 20
Supply School(3): soln[2] + soln[3] <= 30 or 15.0 + 15.0 <= 30 or 30.0 <= 30
Demand School(2): soln[0] + soln[2] >= 5 or 0.0 + 15.0 >= 15.0 or 15.0 >= 5
Demand School(4): soln[1] + soln[3] >= 45 or 30.0 + 15.0 >= 45.0 or 45.0 >=
45
Shipping cost = [1919.3192442452619] $

```

The following code snippet can be used to visualize the solution of a spatial map using folium:

```

def normalize(lst): ← | Normalize function.
    s = sum(lst)
    return list(map(lambda x: (x/s)*10, lst))

soln_normalized = normalize(soln) ← | Normalize solution array.

colors = ['cyan', 'brown', 'orange', 'purple'] ← | Define a color list.

m = folium.Map(location=[location.latitude, location.longitude],
➡ zoom_start=15, scrollWheelZoom=False, dragging=False) ← | Create a map centered
                                                         at downtown Toronto.

for i, coord in zip(supply_schools, supply_coords):
    folium.Marker(location=coord, icon=folium.Icon(icon="home",
➡ color='red'), popup=f'Supply School {i+1}').add_to(m) ← | Add markers for
                                                         supply schools.

for i, coord in zip(demand_schools, demand_coords):
    folium.Marker(location=coord, icon=folium.Icon(icon="flag",
➡ color='blue'), popup=f'Demand School {i+1}').add_to(m) ← | Add markers for
                                                         demand schools.

for i in range(len(supply_schools)): ← | Add lines (edges) between
    for j in range(len(demand_schools)): supply and demand schools.
        soln_value = soln[i*len(demand_schools) + j]
        folium.PolyLine(locations=[supply_coords[i], demand_coords[j]],
➡ color=colors[i % len(colors)],
➡ weight=5*soln_normalized[i*len(demand_schools) + j],
➡ popup=folium.Popup(f'# of microscopes: {soln_value}')).add_to(m)

m ← | Show the map.

```

Figure C.36 shows the solution obtained using MIDACO.



Figure C.36 Supply and demand problem solution obtained by MIDACO

For a higher number of schools, you can obtain an unlimited license or request a free academic trial of the unlimited version that can support up to 100,000 variables.

C.10 Chapter 11: Supervised and unsupervised learning

C.10.1 Exercises

1. Multiple choice and true/false: Choose the correct answer for each of the following questions.

1.1. What are the traditional categories of machine learning algorithms?

- a Supervised, unsupervised, unreinforced
- b Supervised, hybrid, reinforcement
- c Supervised, unsupervised, hybrid, reinforcement
- d Unsupervised, semi-supervised, hybrid

1.2. The Kohonen map is trained using supervised learning to produce a two-dimensional representation of the input space of the training samples.

- a True
- b False

1.3. What kind of tasks are common in supervised learning?

- a Clustering and data reduction
- b Classification and regression
- c Feature extraction and anomaly detection
- d Dimensionality reduction and normalization

1.4. What is the task of clustering in unsupervised learning?

- a Group objects based on certain similarity measures
- b Identify traffic signs in a self-driving car
- c Map input features to known labels or classes
- d Optimize a model's performance by feedback loop

1.5. The pointer-network (Ptr-Net) model is designed to address specific limitations of conventional sequence-to-sequence (seq2seq) models, particularly in tasks involving variable-length output sequences.

- a True
- b False

1.6. In reinforcement learning, how does a learning agent learn to make decisions?

- a By minimizing the error between predicted and actual classes
- b By identifying clusters within the input data
- c By maximizing cumulative reward through actions in an environment
- d By mapping input features to known labels or classes

1.7. What does deep learning (DL) enable in machine learning?

- a Feature representation learning at different levels of abstractions
- b Classification of different objects based on labeled data
- c Grouping of similar data points based on certain measures
- d Reward-based decision-making in an interactive environment

1.8. Graph embedding learns a mapping from a low-dimensional continuous domain to a discrete high-dimensional graph domain.

- a True
- b False

1.9. How does DL reduce the need for extensive data preprocessing?

- a By using a large amount of unlabeled data for training
- b By learning through interactions in a feedback loop
- c By approximating mapping functions between data and known labels
- d By learning discriminative features from raw data automatically

1.10. Why is graph-structured data important in the field of combinatorial optimization?

- a It helps to maximize cumulative rewards
- b It assists in mapping functions between data and labels
- c It captures and represents the relationships and constraints between elements
- d It groups objects based on similarity measures

2. Match the terms and descriptions shown in table C.17.

Table C.17

Terms	Descriptions
1. Self-organizing map (SOM)	a. A polygon that fully encompasses a given set of points with maximum area and minimum boundary or circumference of the shape.
2. Convex hull	b. A mechanism that allows the model to dynamically prioritize which parts of the input are most relevant for each output it's trying to predict, making it more effective at understanding context and reducing confusion from long input sequences
3. Pointer-network (Ptr-Net)	c. A type of artificial neural network that is trained using unsupervised learning to produce a low-dimensional (typically two-dimensional), discretized representation of the input space of the training samples
4. K-hop neighborhood	d. A set of neighboring nodes at a distance less than or equal to K
5. Attention mechanism	e. A type of neural network architecture designed to deal with variable-sized input data sequences

3. Find the shortest path to visit 20 major US cities starting from New York City using self-organizing maps. The cities are given by the names and GPS latitude and longitude coordinates as follows:

- New York City (40.72, -74.00)
- Philadelphia (39.95, -75.17)
- Baltimore (39.28, -76.62)
- Charlotte (35.23, -80.85)
- Memphis (35.12, -89.97)
- Jacksonville (30.32, -81.70)
- Houston (29.77, -95.38)
- Austin (30.27, -97.77)
- San Antonio (29.53, -98.47)
- Fort Worth (32.75, -97.33)
- Dallas (32.78, -96.80)
- San Diego (32.78, -117.15)
- Los Angeles (34.05, -118.25)
- San Jose (37.30, -121.87)
- San Francisco (37.78, -122.42)
- Indianapolis: (39.78, -86.15)
- Phoenix (33.45, -112.07)

- Columbus (39.98, -82.98)
- Chicago (41.88, -87.63)
- Detroit (42.33, -83.05)

4. Optimizing the hyperparameters can significantly improve the performance of the ML model. Tune the hyperparameters in listing 11.6, and observe the effect on the ConvexNet model's performance with different testing datasets. The hyperparameters to be tuned include

- The number of input features for the model
- The number of embedding dimensions
- The number of hidden units in the model
- The number of attention heads in the multi-head self-attention mechanism
- The number of layers in the model
- The dropout probability
- The number of training epochs
- The batch size used during training
- The learning rate for the optimizer

C.10.2 Solutions

1. Multiple choice and true/false

1.1. c) Supervised, unsupervised, hybrid, reinforcement

1.2. b) False (The Kohonen map is trained using unsupervised learning to produce a low-dimensional representation of the input space of the training samples, not supervised learning.)

1.3. b) Classification and regression

1.4. a) Group objects based on certain similarity measures

1.5. a) True

1.6. c) By maximizing cumulative reward through actions in an environment

1.7. a) Feature representation learning at different levels of abstractions

1.8. b) False (Graph embedding learns a mapping from a discrete high-dimensional graph domain to a low-dimensional continuous domain, not the other way around.)

1.9. d) By learning discriminative features from raw data automatically

1.10. c) It captures and represents the relationships and constraints between elements

2. 1-c, 2-a, 3-e, 4-d, and 5-b.

3. MiniSom is used in this listing. MiniSom is a minimalistic and Numpy-based implementation of the SOM. You can install this library using `!pip install minisom`.

The next listing shows how to solve the 20-city TSP using self-organizing maps. We start by importing the libraries and modules, defining the cities of interest, and calculating the haversine distances between pairs of cities.

Listing C.20 Solving TSP using SOM

```
import numpy as np
import pandas as pd
from collections import defaultdict
from haversine import haversine
import networkx as nx
import matplotlib.pyplot as plt
import random
from minisom import MiniSom
```

```
cities = {
    'New York City': (40.72, -74.00),
    'Philadelphia': (39.95, -75.17),
    'Baltimore': (39.28, -76.62),
    'Charlotte': (35.23, -80.85),
    'Memphis': (35.12, -89.97),
    'Jacksonville': (30.32, -81.70),
    'Houston': (29.77, -95.38),
    'Austin': (30.27, -97.77),
    'San Antonio': (29.53, -98.47),
    'Fort Worth': (32.75, -97.33),
    'Dallas': (32.78, -96.80),
    'San Diego': (32.78, -117.15),
    'Los Angeles': (34.05, -118.25),
    'San Jose': (37.30, -121.87),
    'San Francisco': (37.78, -122.42),
    'Indianapolis': (39.78, -86.15),
    'Phoenix': (33.45, -112.07),
    'Columbus': (39.98, -82.98),
    'Chicago': (41.88, -87.63),
    'Detroit': (42.33, -83.05)
}
```

← **Define latitude and longitude for twenty major US cities.**

**Create a
haversine
distance matrix
based on
latitude-
longitude
coordinates.**

```
distance_matrix = defaultdict(dict)
for ka, va in cities.items():
    for kb, vb in cities.items():
        distance_matrix[ka][kb] = 0.0 if kb == ka else haversine((va[0],
        ➔ va[1]), (vb[0], vb[1]))
```

```
distances = pd.DataFrame(distance_matrix)
city_names=list(distances.columns)
```

Convert the distance dictionary into a dataframe.

```
distances=distances.values
```

← **Get the haversine distances between pairs of cities.**

We then define a SOM to solve the TSP instance. This SOM is 1D with N neurons. The dimensionality of the input data is 2 (latitude and longitude coordinates). The sigma parameter is used for the Gaussian neighborhood function. This parameter controls the spread of the influence of neighboring neurons during training. The learning rate determines the step size of weight updates during training. The neighborhood function used during training is set to Gaussian, which means the influence of neighboring neurons decreases with distance. The seed for the random number generator is set to 50 to ensure reproducibility of the results.

```

N_neurons = city_count*2  ← Set the number of neurons (nodes) for the 1D SOM.

som = MiniSom(1, N_neurons, 2, sigma=10, learning_rate=.5,
              neighborhood_function='gaussian', random_seed=50)
Initialize the weights. → som.random_weights_init(points)
                                Create a self-organizing map
                                with 1xN_neurons grid.

```

The following code snippet generates a set of visualizations to show the progress of the SOM training:

```

plt.figure(figsize=(10, 9))
for i, iterations in enumerate(range(5, 61, 5)):
    som.train(points, iterations, verbose=False, random_order=False)
    plt.subplot(3, 4, i+1)
    plt.scatter(x,y)
    visit_order = np.argsort([som.winner(p)[1] for p in points])
    visit_order = np.concatenate((visit_order, [visit_order[0]]))
    plt.plot(points[visit_order][:,0], points[visit_order][:,1])
    plt.title("iterations: {i};\nerror: {e:.3f}".format(i=iterations,
    → e=som.quantization_error(points)))
    plt.xticks([])
    plt.yticks([])
plt.tight_layout()
plt.show()

```

Figure C.37 shows the visualized plots.

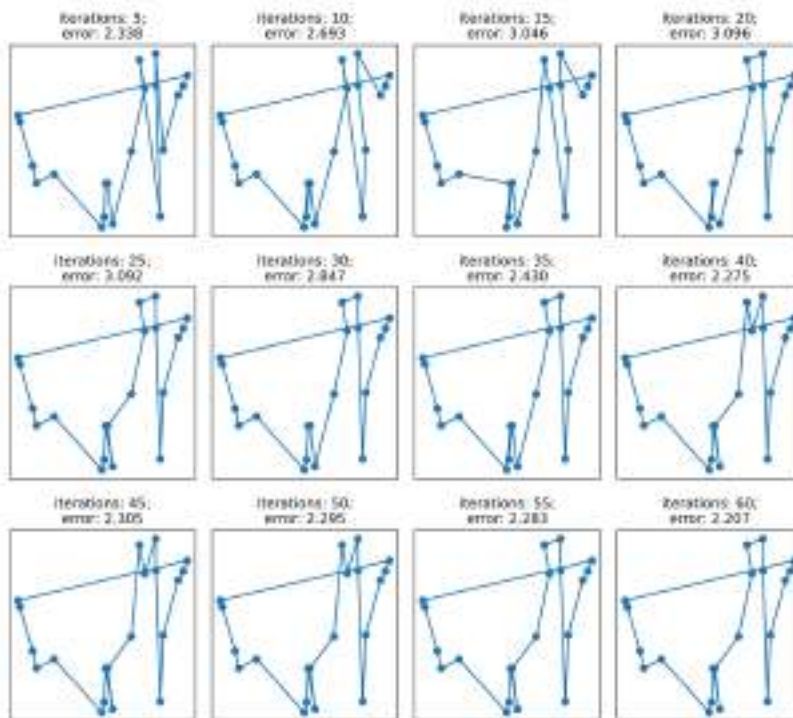


Figure C.37 Progress of SOM training with increasing numbers of iterations

C.11 Chapter 12: Reinforcement learning

C.11.1 Exercises

1. Multiple choice and true/false: Choose the correct answer for each of the following questions.

- 1.1. In reinforcement learning, what does the term “reward” represent?
 - a The penalty for performing an action
 - b The immediate feedback received from the environment
 - c The probability of taking a particular action
 - d The number of steps taken by the agent
- 1.2. The goal of reinforcement learning is to
 - a Minimize the cumulative reward
 - b Find the shortest path to the goal state
 - c Learn an optimal policy to maximize cumulative rewards
 - d Achieve a deterministic environment
- 1.3. Which of the following RL algorithms is considered model-free?
 - a Expert iteration
 - b Proximal policy optimization (PPO)
 - c Imagination-augmented agents (I2A)
 - d None of the above
- 1.4. The concept of a “discount factor” in reinforcement learning is used to
 - a Reduce the size of the state space
 - b Decrease the rewards obtained from the environment
 - c Balance the importance of immediate rewards and future rewards
 - d Encourage exploration over exploitation
- 1.5. Which of the following RL algorithms is considered an on-policy RL method?
 - a Q-learning
 - b Twin-delayed deep deterministic policy gradient (TD3)
 - c Deep deterministic policy gradient (DDPG)
 - d Proximal policy optimization (PPO)
- 1.6 PPO-clip and PPO-penalty are two variants of the policy gradient method designed to address potential instability during training.
 - a True
 - b False
- 1.7. Which multi-armed bandit strategy randomly selects a slot machine at each trial without considering past results?
 - a Exploit-only greedy strategy
 - b Epsilon-greedy strategy

- c Upper confidence bound (UCB) strategy
 - d Explore-only strategy
- 1.8.** In the ε -greedy strategy, how does the agent balance exploration and exploitation?
- a It always selects the machine with the highest estimated mean reward.
 - b It randomly selects a machine with a certain probability (epsilon) and otherwise selects the machine with the highest estimated mean reward.
 - c It explores all machines equally during each trial.
 - d It focuses on exploiting the current best machine only.
- 1.9.** What does “regret” measure in the multi-armed bandit problem?
- a The difference between the maximum possible reward and the reward obtained from each selected machine
 - b The number of times the agent chooses to explore a new machine
 - c The total time spent in the same single state
 - d The total number of arms or actions available to the agent
- 1.10.** What does the Markov decision process (MDP) represent in the context of reinforcement learning?
- a A process that involves making decisions without considering state transitions
 - b A method for supervised learning using labeled datasets
 - c A mathematical framework for planning under uncertainty, in which actions influence future states with certain probabilities
 - d A type of optimization algorithm for clustering data

2. Imagine you are a digital marketer running an online advertising campaign. You have several ad variations that you can display to users, each with its own click-through rate (CTR) or conversion rate. Click-through measures the rate at which users click on a link, while conversion measures the rate at which users complete a desired action after clicking on the link, such as making a purchase, signing up for a newsletter, or completing a form. Your goal is to maximize user engagement or conversions by selecting the most effective ad variation.

Let’s assume you have three ad variations, represented by arms A1, A2, and A3. Each ad variation has an associated probability distribution of click-through or conversion rates, denoted as Q1, Q2, and Q3. These probability distributions represent the likelihood of a user clicking on each ad variation. At each time step t , you need to choose an ad variation A to display to users. When ad variation A is displayed, users interact with it, and you observe the outcome, which can be a click or a conversion. The outcome is drawn from the probability distribution $Q(A)$, representing the likelihood of a click or conversion for ad variation A . Assume that the three probability distributions Q1, Q2, and Q3 are normal distributions with means of {7, 10, 6} and standard deviations of {0.45, 0.65, 0.35} respectively. Your objective is to maximize the cumulative number of clicks over a series of ad displays (let’s say 10,000 ad displays). Write Python code to implement an ε -greedy strategy to determine which ad variation to display at each time step based on the estimated click-through rates.

3. The taxi environment is based on the taxicab, or ride-hailing, problem, where a taxi must pick up a passenger from one location and drop them off at another specified location. The goal of the agent is to learn a policy that navigates the taxi through the grid to pick up and drop off passengers while maximizing the cumulative reward. When the episode starts, the taxi is at a random square, and the passenger is at a random location. The taxi drives to the passenger's location, picks them up, drives to the passenger's destination (another one of the four specified locations), and drops off the passenger. Once the passenger is dropped off, the episode ends. The states, actions, and rewards are as follows:

- States (observation space includes 500 discrete states): 25 taxi positions (any location within the 5×5 grid world); 5 passenger locations (0: R(ed); 1: G(reen); 2: Y(ellow); 3: B(lue); 4: in taxi) and 4 destinations (0: R(ed); 1: G(reen); 2: Y(ellow); 3: B(lue)). Thus, this taxi environment has a total of $5 \times 5 \times 5 \times 4 = 500$ possible states.
- Actions (action space includes 6 discrete actions): 0 = move south; 1 = move north; 2 = move east; 3 = move west; 4 = pick up passenger, and 5 = drop off passenger.
- Rewards: +20 (a high positive reward for a successful drop off); -10 (a penalty for executing pickup and drop-off actions illegally, such as if the agent tries to drop off a passenger in a wrong location), and -1 (a slight negative reward for not making it to the destination after every time step, to mimic the delay).

Write Python code to show how to use A2C to learn the optimal policy for this environment. Experiment with vectorized environments where multiple independent environments are stacked into a single environment. Vectorized environments enable you to run multiple instances of an environment in parallel. Instead of training an RL agent on one environment per step, it allows you to train it on n environments per step. For example, if you want to run four parallel environments, you can specify this number when you create the environment, as follows: `env = make_vec_env("Taxi-v3", n_envs=4, seed=0)`.

4. In preparing for a scheduled flight, an airline's flight operations team is tasked with selecting the best flight route and service according to the shared context. The shared context represents the type of flight (domestic or international) and the type of passengers (business class, economy class, or a mix). The flight operations team must decide the best strategy for flight route, meal service, and in-flight entertainment. The options are represented as follows:

- *Flight route*—The most direct route, a fuel-efficient route that may be longer, or a route that avoids turbulence but may require more time
- *Meal service*—Full meal with multiple options, a simple meal with fewer options, or just snacks and beverages
- *In-flight entertainment*—Movies and music, in-flight Wi-Fi service, or a combination of both

The reward is how satisfactory the chosen options are for a given flight (shared context). The reward function receives as arguments the shared context (the type of flight and passenger class) and the selected actions for each option (the chosen flight route, meal service, and in-flight entertainment). To mirror real-world scenarios and complexities, we inject normal noise in the reward value. The objective is to select the best action from the available combinatorial actions in such a way that it maximizes the total reward. Write Python code to train and test a contextual bandit for this problem.

5. Listing 11.4 shows how to solve TSP using an ML model pretrained with a supervised approach or a reinforcement learning approach. Replace the supervised learning model `sl-ar-var-20pnn-gnn-max_20200308T172931` with the pretrained RL model `rl-ar-var-20pnn-gnn-max_20200313T002243`, and report your observations.

6. The gym-electric-motor (GEM) package is a Python toolbox designed for simulating and controlling electric motors. It is built upon OpenAI Gym environments and is suitable for classical control simulations and reinforcement learning experiments. Use GEM to define a permanent magnet synchronous motor (PMSM) environment as shown in figure C.38. For more information about PMSM and GEM, see Traue et al.'s article "Toward a reinforcement learning environment toolbox for intelligent electric motor control" [10].

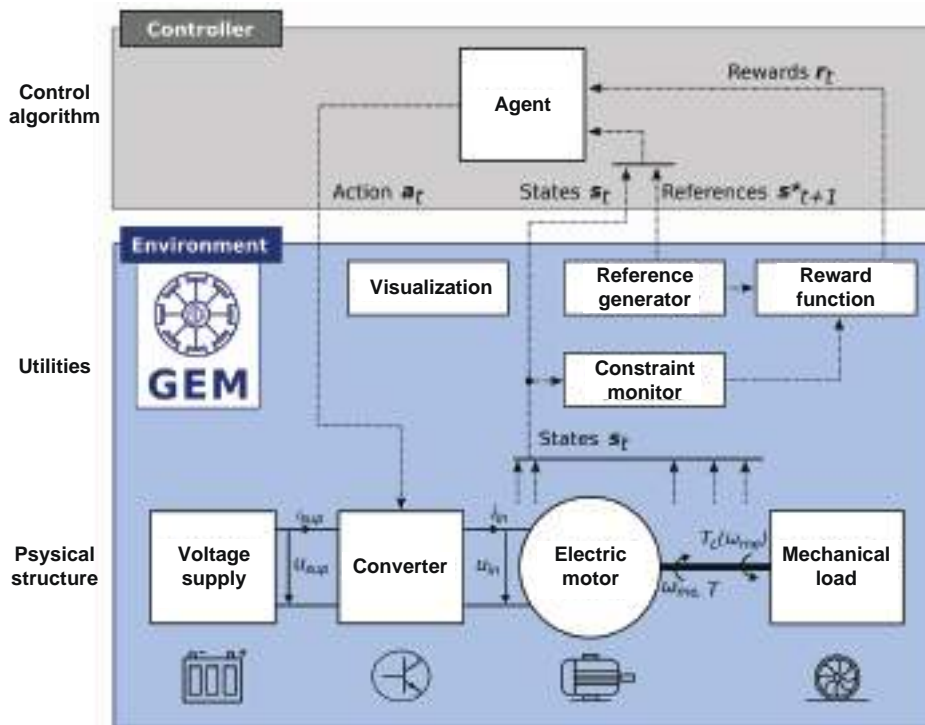


Figure C.38 The gym-electric-motor (GEM) environment described in Traue et al.'s article [9]

Listing C.24, provided in the book's GitHub repo, provides a simplified implementation of PPO for electric motor control. This code is used to train a control model (a PPO RL agent) to control the current for a permanent magnet synchronous motor. This agent mainly controls the converter that converts the supply currents to the currents flowing into the motor. Experiment with different parameters in this algorithm and consider trying other RL models available in Stable-Baselines3 (SB3).

C.11.2 Solutions

1. Multiple choice and true/false
 - 1.1. b) The immediate feedback received from the environment
 - 1.2. c) Learn an optimal policy to maximize cumulative rewards.
 - 1.3. b) Proximal policy optimization (PPO)
 - 1.4. c) Balance the importance of immediate rewards and future rewards.
 - 1.5. d) Proximal policy optimization (PPO)
 - 1.6. a) True
 - 1.7. d) Explore-only strategy
 - 1.8. b) It randomly selects a machine with a certain probability (epsilon) and otherwise selects the machine with the highest estimated mean reward.
 - 1.9. a) The difference between the maximum possible reward and the reward obtained from each selected machine
 - 1.10. c) A mathematical framework for planning under uncertainty, in which actions influence future states with certain probabilities
2. Listing C.21 shows an implementation of a ϵ -greedy strategy to determine which ad variation to display at each time step based on the estimated click-through rates. In this code snippet, 10,000 ad displays are simulated. The estimates of the click-through rates for each ad variation are updated after each display.

Listing C.21 Online advertising using MAB

```
import numpy as np

num_arms = 3  # Initialize the number of arms (actions).
num_trials = 10000  # Initialize the number of trials (ads).

mu = [7, 10, 6]
sigma = [0.45, 0.65, 0.35]  # Probability distribution of each arm

counts = np.zeros(num_arms)
rewards = np.zeros(num_arms)  # Counters for each arm

a = np.random.choice(num_arms)  # Select an initial arm.

eps = 0.1  # Set the epsilon value of the epsilon-greedy algorithm.

for t in range(num_trials):
    if np.random.rand() > eps:  # Exploitation
        a = np.argmax(rewards / (counts + 1e-5))  # Add a small constant to avoid division by zero.
```

```

Exploration else:
    a = np.random.choice(num_arms)

    reward = np.random.normal(mu[a], sigma[a])

    counts[a] += 1
    rewards[a] += reward

estimates = rewards / counts

print("Estimated click-through rates: ", estimates)

```

Calculate and print the estimated click-through rates.

In this script, `counts` keeps track of the number of times each ad variation has been displayed, and `rewards` keeps track of the total number of clicks for each ad variation. At the end of the script, the estimated click-through rates for each ad variation are calculated and printed.

3. Listing C.22 shows the steps of learning the optimal policy for the taxicab problem using A2C. This code uses Stable-Baselines3 (SB3), a library for reinforcement learning, to train an agent using A2C on the Taxi-v3 environment. The SB3 function `make_vec_env` is used to create a vectorized environment that can run multiple parallel environments in the same process. The SB3 function `evaluate_policy` is used to evaluate the learned policy of the agent.

Listing C.22 Dispatching a taxicab using A2C RL

```

from stable_baselines3.common.env_util import make_vec_env
from stable_baselines3 import A2C
from stable_baselines3.common.evaluation import evaluate_policy
import matplotlib.pyplot as plt
from IPython.display import display, clear_output

env = make_vec_env("Taxi-v3", n_envs=1, seed=0)
print('Number of states:{}'.format(env.observation_space))
print('Number of actions:{}'.format(env.action_space))

model = A2C(policy="MlpPolicy", env=env, verbose=True)
model.learn(total_timesteps=10000, progress_bar=True)

```

Print the observation and action spaces of the environment.

Create a vectorized environment with a single parallel environment (`n_envs=1`).

Create an A2C agent with MlpPolicy as the policy network.

Train the A2C agent on the Taxi-v3 environment for 10,000 timesteps.

After the training is complete, the agent will have learned an optimal policy for navigating the Taxi-v3 environment to efficiently pick up and drop off passengers at the correct locations. The following code snippet visualizes the learned policy by rendering the Taxi-v3 environment using the trained A2C agent.

```

images = []
vec_env = model.get_env()
obs = vec_env.reset()

```

Create an empty list to store the frames (images) of the rendered environment.

Retrieve the environment associated with the model.

Reset the environment and obtain the initial observation after the reset.

```

for i in tqdm(range(300)):
    action, _states = model.predict(obs, deterministic=True)
    obs, rewards, dones, info = vec_env.step(action)
    state_img = vec_env.render("rgb_array")
    fig = plt.figure()
    plt.imshow(state_img)
    plt.axis('off')
    display(fig)
    images.append(fig)
    clear_output(wait=True)
    plt.close()

```

Predict an action based on the current observation.

Render the environment as an RGB image.

Get a new observation and the reward.

Clear the output for the next image.

4. Listing C.23 shows the implementation of a contextual bandit for an airline's flight operations using the Vowpal Wabbit Python library. The shared context is defined by two lists: `flight_types` and `passenger_classes`. The possible choices, or actions, for the bandit problem are defined by `flight_routes`, `meal_services`, and `entertainment_options`. The `reward_function` calculates the reward associated with a particular combination of flight route, meal service, and entertainment option. The rewards are generated using a normal distribution with different means for different choices. The standard deviation (scale) is set to 0.05, implying that the rewards are sampled from a normal distribution with a small amount of variance.

Listing C.23 Contextual bandit for an airline's flight operations

```

import vowpalwabbit
import torch
import matplotlib.pyplot as plt
import pandas as pd
import random
import numpy as np

flight_types = ['domestic', 'international']
passenger_classes = ['business', 'economy', 'mix']

flight_routes = ['direct', 'fuel_efficient', 'turbulence_avoidance']
meal_services = ['full_meal', 'simple_meal', 'snacks_beverages']
entertainment_options = ['movies_music', 'in_flight_wifi', 'combo']

def reward_function(shared_context, flight_route, meal_service,
    entertainment_option):
    if flight_route == 'direct':
        route_reward = np.random.normal(loc=0.9, scale=0.05)
    elif flight_route == 'fuel_efficient':
        route_reward = np.random.normal(loc=0.8, scale=0.05)
    else:
        route_reward = np.random.normal(loc=0.7, scale=0.05)

    if meal_service == 'full_meal':
        meal_reward = np.random.normal(loc=0.9, scale=0.05)
    elif meal_service == 'simple_meal':
        meal_reward = np.random.normal(loc=0.8, scale=0.05)
    else:

```

Set the shared context.

Set possible choices/action options.

Calculate the reward associated with a particular combination of options.

```

meal_reward = np.random.normal(loc=0.7, scale=0.05)

if entertainment_option == 'movies_music':
    entertainment_reward = np.random.normal(loc=0.9, scale=0.05)
elif entertainment_option == 'in_flight_wifi':
    entertainment_reward = np.random.normal(loc=0.8, scale=0.05)
else:
    entertainment_reward = np.random.normal(loc=0.7, scale=0.05)

reward = (route_reward + meal_reward + entertainment_reward) / 3.0

return reward

```

As a continuation, the following two utility functions are defined. `generate_combinations` generates combinations of flight routes, meal services, and entertainment options, along with their associated descriptions. `sample_truck_pmf` performs sampling based on a probability mass function (PMF):

```

def generate_combinations(shared_context, flight_routes, meal_services,
    ➤ entertainment_options):
    examples = [f"shared |FlightType {shared_context[0]} PassClass
    ➤ {shared_context[1]}"
    descriptions = []
    for route in flight_routes:
        for meal in meal_services:
            for entertainment in entertainment_options:
                examples.append(f"|Action route={route} meal={meal}
    ➤ entertainment={entertainment}")
                descriptions.append((route, meal, entertainment))
    return examples, descriptions

def sample_truck_pmf(pmf):
    pmf_tensor = torch.tensor(pmf)
    index = torch.multinomial(pmf_tensor, 1).item()
    chosen_prob = pmf[index]

    return index, chosen_prob

```

We can now create a contextual bandit using the Vowpal Wabbit (VW) library and evaluate its performance over a specified number of iterations. This contextual bandit will make decisions (select actions) in the context of different flight types and passenger classes to maximize the expected reward:

```

cb_vw = vowpalwabbit.Workspace(
    "--cb_explore_adf --epsilon 0.2 --interactions AA AU AAU -l 0.05 -
    ➤ power_t 0",
    quiet=True,
)

```

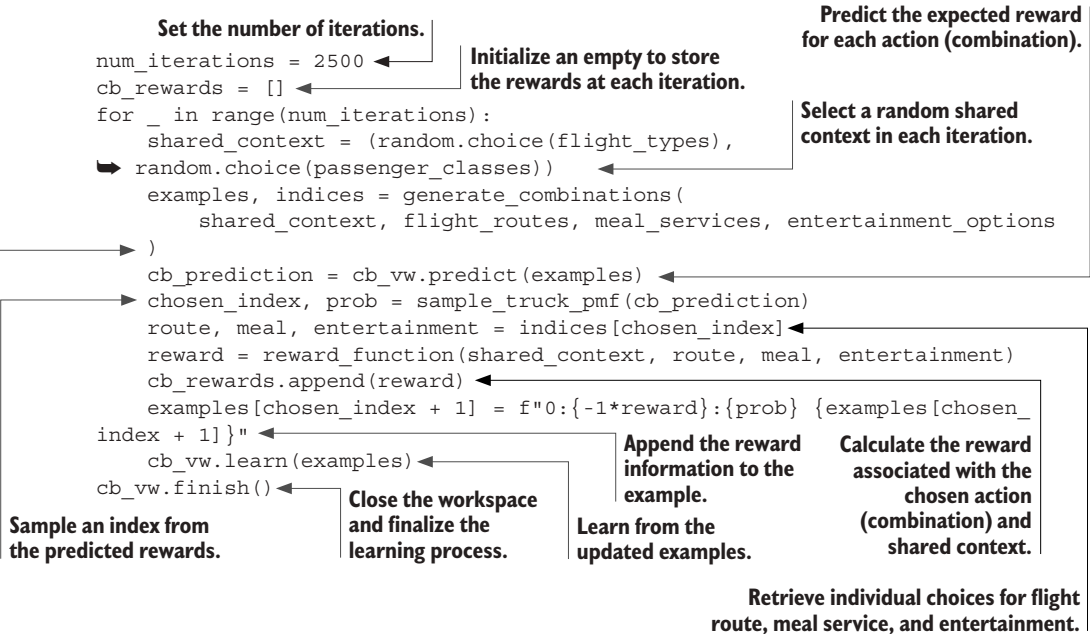
Here are the key arguments to create contextual bandit:

- `--cb_explore_adf`—Enables contextual bandit exploration with action-dependent features

- `--epsilon 0.2`—Sets the exploration rate to 0.2, meaning that the bandit will explore non-greedy actions with a probability of 0.2 (20% of the time)
- `--interactions AA AU AAU`—Specifies three-way interactions between features AA, AU, and AAU
- `-l 0.05`—Sets the learning rate to 0.05, which controls the step size in the learning process
- `--power_t 0`—Specifies that the learning rate is constant (no learning rate decay)
- `num_iterations = 2500`

The following code snippet allows us to run the created contextual bandit to make decisions:

Generate all possible combinations of flight routes, meal services, and entertainment options based on the chosen shared context.



We can print the average reward during training as follows:

```

plt.plot(pd.Series(cb_rewards).expanding().mean())
plt.xlabel("Iterations")

plt.ylabel("Average reward")
plt.show()

```

Figure C.39 shows the progress of an average reward obtained in each iteration during the learning process.

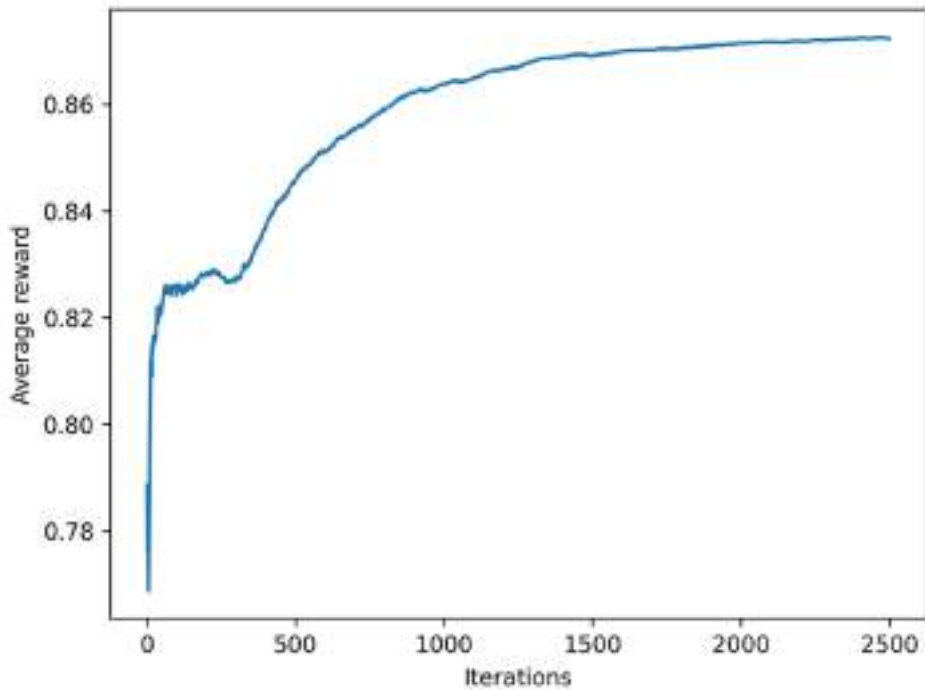


Figure C.39 Average reward in each iteration during the learning process

The complete version of listing C.23 provided in the book's GitHub repo defines a `test_model` function and then tests the contextual bandit model using a given shared context. The `test_model` function is defined to test the contextual bandit model by simulating a single decision-making instance for a given shared context. It takes four parameters—`shared_context`, `flight_routes`, `meal_services`, and `entertainment_options`:

Predict the expected reward for each action (combination) based on the provided examples.

Generate all possible combinations of flight routes, meal services, and entertainment options based on the given shared context.

```
def test_model(shared_context, flight_routes, meal_services,
    entertainment_options):
    examples, _ = generate_combinations(shared_context, flight_routes,
    meal_services, entertainment_options)
    cb_prediction = cb_vw.predict(examples)
    chosen_index, prob = sample_truck_pmf(cb_prediction)
    chosen_action = examples[chosen_index + 1]
```

Retrieve the chosen action (combination).

Sample an index from the predicted rewards.

```

route, meal, entertainment = indices[chosen_index]
expected_reward = reward_function(shared_context, route, meal,
entertainment)
print("Chosen Action:", chosen_action)
print("Expected Reward:", expected_reward)

test_shared_context = ('domestic', 'business')
test_model(test_shared_context, flight_routes, meal_services,
entertainment_options)

```

Print the chosen action and the expected reward.

Calculate the expected reward associated with the chosen action and shared context.

Set a specific shared context.

Test the contextual bandit model's decision-making process for this specific context.

Retrieve individual choices for flight route, meal service, and entertainment.

This code will generate the output like the following for the given context:

```

Chosen Action: |Action route=fuel_efficient meal=full_meal
entertainment=movies_music
Expected Reward: 0.87

```

5. Replacing the included supervised learning model, `sl-ar-var-20pnn-gnn-max_20200308T172931`, with the pretrained RL model, `rl-ar-var-20pnn-gnn-max_20200313T002243`, in listing 11.4 is done as follows:

```

model = "learning_tsp/pretrained/tsp_20-50/rl-ar-var-20pnn-gnn-
max_20200313T002243"
# model = "learning_tsp/pretrained/tspsl_20-50/sl-ar-var-20pnn-gnn-
max_20200308T172931"

```

6. Listing C.24, available in the book's GitHub repo, provides a simplified implementation of PPO for electric motor control. Experiment with the different parameters of this algorithm and consider trying other RL models available in SB3, such as advantage actor-critic (A2C), soft actor-critic (SAC), deep deterministic policy gradient (DDPG), deep Q network (DQN), hindsight experience replay (HER), and twin delayed DDPG (TD3).