AI techniques for design, planning, and control problems

Optimization Algorithms

Alaa Khamis

MEAP

MANNING

MEAP Edition
Manning Early Access Program

Optimization Algorithms

AI techniques for design, planning, and control problems

Version 2

welcome

Thanks for purchasing the MEAP for *Optimization Algorithms: AI techniques for design, planning, and control problems*.

Optimization problems are ubiquitous in different aspects of life. This book is written for practitioners interested in solving ill-structured search and optimization problems using modern derivative-free algorithms. This book will get you up to speed with the core concepts of search and optimization and endow you with the ability to deal with practical design, planning and control problems.

Without assuming any prior knowledge of search and optimization and with an intermediate knowledge of data structures and Python, this book has been written to take most anyone from never solving search and optimization problems to being a well-rounded search and optimization practitioner able to select, implement and adapt the right solver for the right problem.

This book grew out of several courses related to search and optimization taught by me at different universities and training centers in industry. My 25 years working as an AI and Robotics professor in the academia and as a technical leader in industry have given me a wealth of experiences to share with you through this book.

By the end of the book, you should understand:

- How to deal with discrete and continuous ill-structured optimization problems using deterministic and stochastic derivative-free search and optimization techniques
- How to apply deterministic graph search algorithms to solve graph problems
- How to apply nature-inspired algorithms to find optimal or near-optimal solutions for practical design, planning and control problems
- How to use machine learning methods to solve search and optimization problems
- How to solve the search dilemma by achieving trade-off between search space exploration and exploitation using algorithm parameter adaptation
- How to use state-of-the-art Python libraries related to search and optimization

The book is divided into five parts. Part 1 covers deterministic graph search algorithms. Part 2 will focus on trajectory-based optimization algorithms giving simulated annealing and tabu search as examples. Moving forward, Part 3 introduces evolutionary computing algorithms followed by presenting swarm-intelligence algorithms in Part 4. The last part of the book shows how machine learning-based methods can be used to solve complex search and optimization problems. Throughout this book, a wealth of examples and in-depth case studies are provided for both novices and experts.

I do believe that learning is the common ground between the author and the reader. I hope that what you'll get access to will be of immediate use to you. I also look forward to learning from your valuable feedback to develop the best book possible. Please let me know your thoughts in the liveBook Discussion forum on what's been written so far and what you'd like to see in the rest of the book.

Thanks again for your interest and for purchasing the MEAP!

—Dr. Alaa Khamis

brief contents

C Solutions to Exercises

1 Introduction to Search and Optimization

This chapter covers

- **What are search and optimization?**
- **Why care about search and optimization?**
- **Going from "toy problems" to real-world solutions**
- **Defining an optimization problem**
- **Introducing well-structured problems and ill-structured problems**
- **Search algorithms and the search dilemma**

As human beings, we constantly strive to optimize our everyday lives. Whether it's getting to work faster (so you can sleep in a little bit longer), balancing school and leisure time, or even budgeting for personal spending, we try to maximize the benefits and minimize the costs. Likewise, corporations maximize profits by increasing efficiency and eliminating waste. For example, logistics giants like FedEx, UPS, and Amazon spend millions of dollars each year researching new ways to trim the cost of delivering packages. Similarly, telecommunications agencies seek to determine the optimal placement of crucial infrastructure, like cellular towers, to service the maximum number of customers while investing in the minimum level of equipment.

This sort of optimization behavior is not unique to humans; nature likewise tends towards efficiency and optimality. Bacterial colonies, comprised of clusters of between 10 million and 10 billion individual organisms, form an adaptable, complex system that can perform many complicated tasks such as foraging, pathfinding, and learning based on external stimuli. Insects like ants and honeybees have developed their own unique optimization methods, from navigating the shortest path to an existing food source to discovering new food sources in an unknown external environment. Honeybee colonies focus their foraging efforts on only the most profitable patches and build their honeycombs with a shape that economizes labor and wax. Fish swimming in schools or cruising in the same direction minimize total energy usage by exploiting pressure fields created by the other fish. At the same time, migratory birds utilize separation, alignment, and cohesion-based optimization to avoid mid-air collisions and increase flight endurance. Non-biological phenomena also tend towards efficiency. For example, light traveling between two different media will refract along an angle that minimizes the travel time.

As technology has developed, computer-based optimization is now an inescapable reality of the digital era. Transportation network companies (TNCs) like Uber, Lyft, and DiDi route drivers efficiently during passenger trips and direct drivers to ride-hailing hotspots during idle periods to minimize passenger wait time. As urbanization intensifies worldwide, local emergency services depend on efficient dispatching and routing platforms to select and route the appropriate vehicles, equipment, and personnel to respond to incidents across increasingly complex metropolitan road networks. Airliners need to solve several optimization problems such as flight planning, fleet assignment, crew scheduling, aircraft routing and aircraft maintenance planning. Healthcare systems also handle optimization problems such as hospital resource planning, emergency procedure management, patient admission scheduling, surgery scheduling and pandemic containment. Industry 4.0, a major customer of optimization technology, deals with complex optimization problems such as smart scheduling/rescheduling, assembly line balancing, supply-chain optimization, and operational efficiency. Smart cities deal with large-scale optimization problems such as stationary asset optimal assignments, mobile asset deployment, energy optimization, water control, pollution reduction, waste management and bushfire containment. These examples show how ubiquitous and important optimization is as a way to maximize operational efficiency in different domains.

1.1 Why care about search and optimization?

Search is the systematic examination of states, starting from an initial state, and ending (hopefully) at the goal state. Optimization techniques are in reality search methods, where the goal is to find an optimal or a near-optimal state within the feasible search space. The feasible search space is a subset of the optimization problem space where all the problem's constraints are satisfied. It's hard to come up with a single industry that doesn't already use some form of search or optimization methods, software, or algorithms. It's highly likely that somehow in your workplace or industry, you deal with optimization daily; it's just that you aren't aware of it. While search and optimization are undoubtedly ubiquitous in almost all industries, it may not always be practical to use complicated algorithms to optimize processes. For example, consider a small pizzeria that offers a food delivery service to its local customers. Let's assume that the restaurant processes around ten deliveries on an average weeknight. While efficiency-improving strategies (such as avoiding left turns in left-driving countries or right turns in right-driving countries, avoiding major intersections, avoiding school zones during drop-off and pick-up times, avoiding bridges during lift times and favoring downhill roads) may theoretically shorten delivery times and reduce costs, the scale of the problem is so tiny that implementing these kinds of changes may not lead to noticeable impact.

In larger scale problems such as fleet assignment and dispatching, multi-criteria stochastic vehicle routing, resource allocation, crew scheduling, applying search and optimization techniques to a problem must be a qualified decision. Some firms or industries may not benefit from excessive process changes due to a lack of expertise or resources to implement those changes. There may also be the concern of a potential lack of follow-through from stakeholders. Implementing the changes could also cost more than the savings obtained through the optimization process. Later in this book, we will see how these costs can be accounted for when developing search and optimization algorithms.

Without assuming any prior knowledge of search and optimization and with an intermediate knowledge of data structures and Python, this book has been written to take most anyone from never solving search and optimization problems to being a well-rounded search and optimization practitioner able to select, implement and adapt the right solver for the right problem. For managers or professionals involved in the high-level technological decisions at their workplace, these skills can be critical in understanding software-based approaches, their opportunities, and limitations when discussing process improvement. In contrast, IT professionals will find these skills more directly applicable when considering options for developing or selecting new software suites and technologies for in-house use. The following subsection describes the methodology we will follow throughout this book.

1.2 Going from toy problem to the real world

When discussing algorithms, many books and references will often present them as a formal definition and then apply them to so-called "toy problems." These trivial problems are helpful because they often deal with smaller datasets and search spaces while being solvable by hand iteration. This book seeks to follow a similar approach but takes it one step further by presenting real-world data implementations. Whenever possible, resources such as real-world datasets and values are used to illustrate the direct applicability and practical drawbacks of the algorithms discussed. Initially, the scaled-down toy problems help the reader appreciate the step-by-step operations involved in the various algorithms. Later, the real-world Python implementations teach the reader how to use multiple datasets and Python libraries to address the increased complexity and scope of actual data.

As illustrated in Figure 1.1, source of inspiration of each search or optimization algorithm is highlighted followed by presenting the algorithm pseudocode, algorithm parameters and heuristics used. Algorithm pros and cons and adaptation methods are then described. The book contains many examples that allow the learners to fully understand how each algorithm works by carrying out iterations by hand on a scaled-down version of the problem. It also includes many programming exercises in a special problem/solution/discussion format to understand how a scaled-up version of the problem previously solved by hand can be solved using Python. Through programming, learners can optimally tune the algorithm and study its performance and scalability.

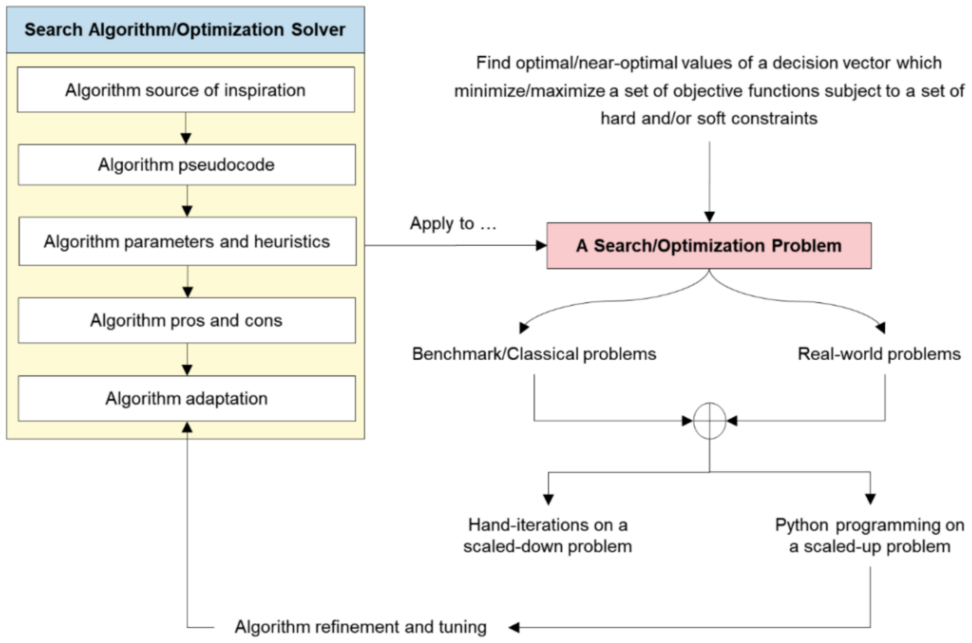

Search Algorithm/Optimization Solver
Algorithm source of inspiration
Algorithm pseudocode
Algorithm parameters and heuristics
Algorithm pros and cons
Algorithm adaptation
Apply to …
Find optimal/near-optimal values of a decision vector which minimize/maximize a set of objective functions subject to a set of hard and/or soft constraints
A Search/Optimization Problem
Benchmark/Classical problems
Real-world problems
Hand-iterations on a scaled-down problem
Python programming on a scaled-up problem
Algorithm refinement and tuning

Figure 1.1 Book methodology. Each algorithm will be introduced following a structure that goes from explanation to application.

Throughout this book, several classical and real-world optimization problems are considered to show how to use search and optimization algorithms discussed in the book. Figure 1.2 shows examples of these optimization/search problems.

Real-world design problems or strategic functions deal with situations when time is not as important as the quality of the solution and users are willing to wait (sometimes even a few days) to get optimal solutions. Planning problems or tactical functions need to be solved in a time span between a few seconds to a few minutes. While control problems or operational functions need to be solved repetitively and quickly, in a time span between few milliseconds to a few seconds. In order to find a solution during such a short period of time, optimality is usually traded in for speed gains. In the next chapter, more thorough discussion of these problem types is provided.

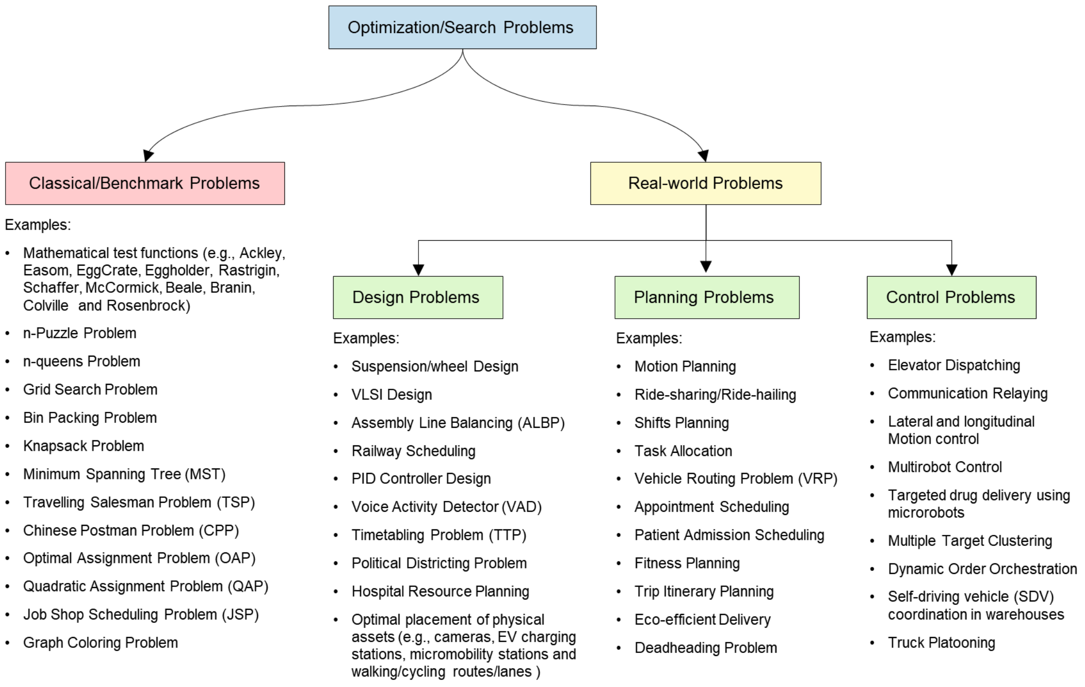

Optimization/Search Problems
Classical/Benchmark Problems
Examples:
Mathematical test functions (e.g., Ackley, Easom, EggCrate, Eggholder, Rastrigin, Schaffer, McCormick, Beale, Branin, Colville and Rosenbrock)
n-Puzzle Problem
n-queens Problem
Grid Search Problem
Bin Packing Problem
Knapsack Problem
Minimum Spanning Tree (MST)
Travelling Salesman Problem (TSP)
Chinese Postman Problem (CPP)
Optimal Assignment Problem (OAP)
Quadratic Assignment Problem (QAP)
Job Shop Scheduling Problem (JSP)
Graph Coloring Problem
Real-world Problems
Design Problems
Examples:
Suspension/wheel Design
VLSI Design
Assembly Line Balancing (ALBP)
Railway Scheduling
PID Controller Design
Voice Activity Detector (VAD)
Timetabling Problem (TTP)
Political Districting Problem
Hospital Resource Planning
Optimal placement of physical assets (e.g., cameras, EV charging stations, micromobility stations and walking/cycling routes/lanes)
Planning Problems
Examples:
Motion Planning
Ride-sharing/Ride-hailing
Shifts Planning
Task Allocation
Vehicle Routing Problem (VRP)
Appointment Scheduling
Patient Admission Scheduling
Fitness Planning
Trip Itinerary Planning
Eco-efficient Delivery
Deadheading Problem
Control Problems
Examples:
Elevator Dispatching
Communication Relaying
Lateral and longitudinal Motion control
Multirobot Control
Targeted drug delivery using microrobots
Multiple Target Clustering
Dynamic Order Orchestration
Self-driving vehicle (SDV) coordination in warehouses
Truck Platooning

Figure 1.2 Examples of classical optimization and real-world optimization/search problems.

It is highly recommended that you first perform the necessary hand iterations for the examples following each algorithm and then try to recreate the Python implementations yourself. Feel free to play around with the parameters and problem scale in the code; one of the advantages of running optimization algorithms through software is the ability to tune for optimality.

1.3 Basic ingredients of optimization problems

Optimization refers to the practice of finding the "best" solutions to a given problem, where "best" usually means satisfactory or acceptable, possibly subject to a given set of constraints. The solutions can be classified into feasible, optimal, and near-optimal solutions.

- **Feasible solutions** are solutions that satisfy all the given constraints.
- **Optimal solutions** are both feasible and provide the best objective function value
- **Near-optimal solutions** are feasible solutions that provide a superior objective function value but are not necessarily the best.

A global optimum, or a global minimum in case of minimization problems, is the best of a set of candidate solutions. A search space may combine multiple global minima, strong local minima, and weak local minima as illustrated in Figure 1.3.

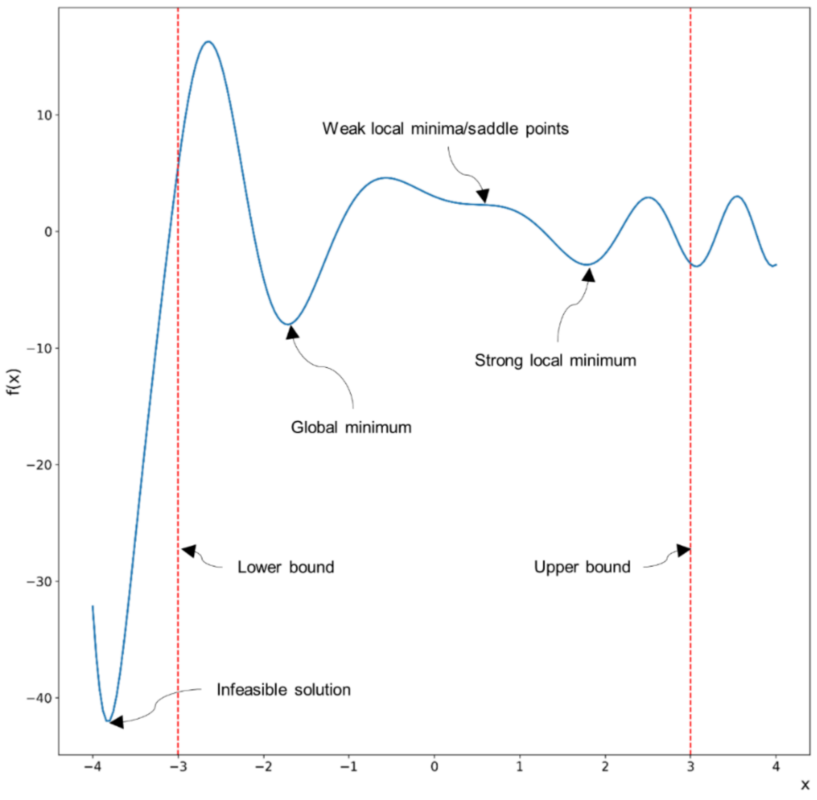


Figure 1.3 Feasible/acceptable solutions fall within the constraints of the problem. Search spaces may display a combination of global, strong local, and weak local minima.

These optimum seeking methods, also known as optimization techniques, are generally studied as a part of operations research (OR). OR, also referred to as decision or management science, is a field that originated at the beginning of World War II due to the urgent need for assignment of scarce resources in military operations. It is a branch of mathematics concerned with applying advanced scientific analytical methods to decision-making and management problems to find the best or optimal solutions.

Optimization problems can generally be stated as follows:

Find X which optimizes f

Subject to a possible set of equality and inequality constraints:

$$g_i(X)=0,\ i=1,2,\ldots,m$$
$$h_j(X)<=0,\ j=1,2,\ldots,p$$

Equation 1.1

where

- $X=(x_1, x_2,\ldots,x_n)^T$ is the vector representing the decision variables
- $f(X)=(f_1(X), f_2(X),\ldots, f_M(X))$ is the vector of objectives to be optimized
- $g_i(X)$ is a set of equality constraints
- $h_j(X)$ is a set of inequality constraints

The following subsections describe three main components of optimization problems: decision variables, objective functions, and constraints.

1.3.1 Decision Variables

Decision variables represents a set of unknowns or variables that affect the objective function's value. If X represents the unknowns, also referred to as the independent variables, then *f*(X) quantifies the quality of the candidate solution or feasible solution.

Example: Assume that an event organizer is planning a conference on Search and Optimization Algorithms. The organizer plans to pay *a* for fixed costs (venue rental, security, and guest speaking fees) and *b* for variable costs (pamphlet, lanyard, ID badge, catered lunch) that depend on the number of participants. Based on past conferences, the organizer predicts that demand for tickets will be as follows:

$$Q=5{,}000-20x$$

Equation 1.2

Where *x* is the ticket price and Q is the expected number of tickets to be sold. Thus, the company expects the following scenarios:

- if the company charges nothing or x=0, the company will give away 5,000 tickets for free;
- If the ticket price is x=\$250, the company will get no attendees as the expected number of tickets to be sold will be zero and
- If the ticket price x<\$250, the company will sell some number of tickets 0<=Q<=5,000.

The profit *f*(x) the event organizer can expect to earn can be calculated according to the following:

$$\text{Profit}=\text{Revenue}-\text{Costs}$$

Equation 1.3

where Revenue=Qx and costs=a+Qb. Altogether, the profit (or objective) function looks like this:

$$f(x)=-20x^2+ (5{,}000+20b)x -5{,}000b-a, x_{LB}<=x<=x_{UB}$$

Equation 1.4

In this problem, there is a single decision variable x, which is the price of the ticket. The predefined parameters include fixed costs *a* and variable costs *b*. The ticket price lower bound x_{LB} and upper bound x_{UB} are considered boundary constraints. Solving this optimization problem focuses on finding the best value of x that maximizes the profit $f(x)$.

1.3.2 Objective Functions

An objective function $f(x)$, also known as the criterion or merit function or utility function or cost function, stands for the quantity to be optimized. Without loss of generality, optimization can be interpreted as the minimization of a value since the maximization of a primal function $f(x)$ can be just the minimization of a dual problem generated after applying mathematical operations on $f(x)$. This means that if the primal is a minimization problem then the dual is a maximization problem (and vice versa). According to this duality aspect of optimization problems, a solution x' which is the minimum for the primal minimization problem is also, at the same time, the maximum for the dual maximization problem as illustrated in Figure 1.4. Moreover, simple mathematical operations like addition, subtraction, multiplication, or division do not change the value of the optimal point. For example, multiplying or divide $f(x)$ by a positive constant *c* or adding or subtracting a positive constant *c* to or from $f(x)$ does not change the optimal value of the decision variable as illustrated in Figure 1.4.

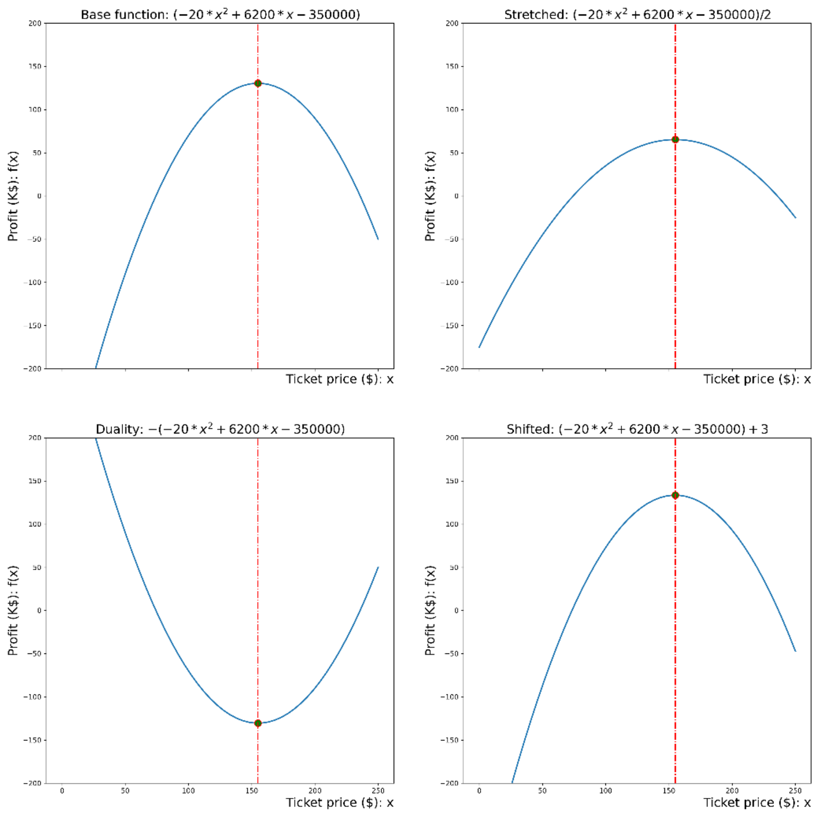


Figure 1.4 Duality principle and mathematical operations on an optimization problem.

Example: In the earlier ticket pricing problem, assume that: *a=50,000*, *b=60*, x_{LB} *=0* and x_{UB}*=250*. Using these values, we have a profit function: $f(x)=-20x^2+6,200x-350,000$. Following derivative-based approach, we can simply derive the function to find its maximum: $df/dx=-40x+6,200=0$ or $40x=6,200$. Thus, the optimal number of tickets to sell is 155, which yields a net profit of $130,500 as shown in Figure 1.5.

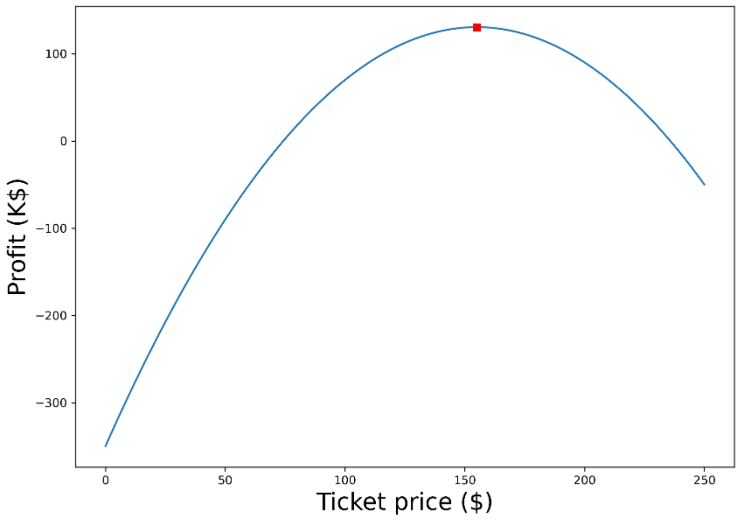


Figure 1.5 Ticket pricing problem. The optimal pricing which maximizes profits is $155 per ticket.

In ticket pricing problem, we have a single objective function to be optimized, which is the profit. In this case, the problem is called mono-objective optimization problem. An optimization problem involving multiple objective functions is known as a multi-objective optimization problem. For example, assume that we want to design an electric vehicle (EV). This design problem’s objective functions can be minimizing acceleration time and maximizing EPA driving range. The acceleration time is the time in seconds the EV takes to accelerate from 0 to 60 mph. EPA range is the approximate number of miles that a vehicle can travel in combined city and highway driving (using a mix of 55% highway and 45% city driving) before needing to be recharged according to the Environmental Protection Agency (EPA)'s testing methodology. Decision variables can include the size of the wheel, the power of the electric motor, and the battery's capacity. A bigger battery is needed to extend the driving range of the EV, which adds extra weight, and therefore the acceleration time increases. In this example, the two objectives are conflicting as we need to minimize acceleration time and maximize the EPA range as shown in Figure 1.6.

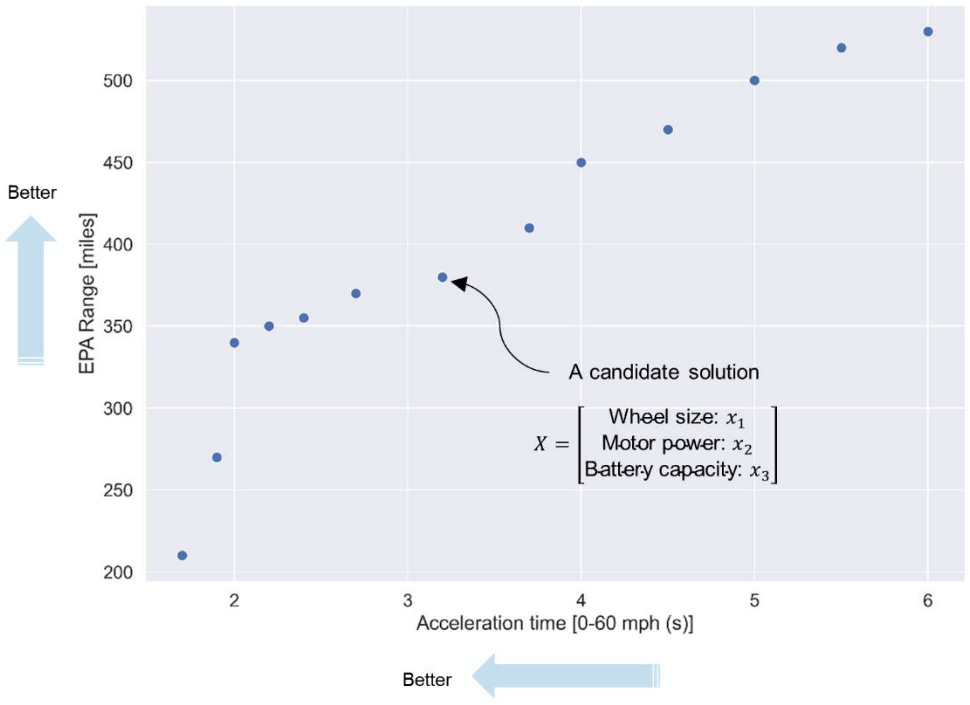


Figure 1.6 Electric vehicle design problem for maximizing EPA range and minimizing acceleration time.

This multi-objective optimization problem can be handled using a preference-based multi-objective optimization procedure or by using a Pareto optimization approach. In the former approach, duality principle is applied first to transform all the conflicting objectives for maximization (e.g., maximizing the EPA range and the inverse of the acceleration time) or minimization (e.g., minimizing the acceleration time and the inverse of the EPA range) and then combine these multiple objectives into a single or overall objective function by using a relative preference vector or a weighting scheme to scalarize the multiple objectives. For example, you may give more weight for EPA range over the acceleration time. However, finding this preference vector or weights is subjective and sometimes is not straightforward. The latter approach relies on finding multiple trade-off optimal solutions and chooses one using higher-level information. This procedure tries to find the best tradeoff by reducing the number of alternatives to an optimal set of non-dominated solutions known as the Pareto Frontier, which can be used to take strategic decisions in multi-objective space. Multi-objective optimization will be discussed in later chapters.

Constraint-satisfaction problems (CSP) do not define an explicit objective function. Instead, the goal is to find a solution that satisfies a given set of constraints. The *N*-queen problem is an example of CSP. In this problem, the aim is to put *n* queens on an *n* x *n* board with no two queens on the same row, column, or diagonal. This 4x4 Queen CSP problem has

256 possible solutions (4^4) and two optimal solutions. Neither of the two optimal solutions is inherently or objectively better than the other. The only requirement of the problem is to satisfy the given constraints.

1.3.3 Constraints

Constrained optimization problems have a set of equality and/or inequality constraints $g_i(X)$, $l_j(X)$ that restrict the values assigned to the unknowns. In addition, most problems have a set of boundary constraints, which define the domain of values for each variable. Furthermore, constraints can be hard (must be satisfied) or soft (desirable to satisfy). Consider the following examples from a school timetabling problem:

- Not having multiple lectures in the same room at the same time is a **hard constraint**
- Not having a teacher give multiple lectures at the same time is also a **hard constraint**
- Guaranteeing a minimum of three teaching days for every teacher may be a **soft constraint**
- Locating back-to-back lectures in nearby rooms may be a **soft constraint**
- Avoiding scheduling very early or very late lectures may also be a **soft constraint**

As another example of hard and soft constraints, navigation apps such as Google Maps, Apple Maps, Waze, or HERE WeGo may allow users to set preferences for routing.

- Avoiding ferries, toll roads, and highways would be **hard constraints**
- Avoiding busy intersections, highways during rush hour, or school zones during drop-off and pick-up times might be **soft constraints**

Soft constraints can be modeled by incorporating a reward/penalty function as part of the objective function. The function rewards solutions that satisfy the soft constraints and penalize those that do not.

Example: Assume that there are 10 parcels to be loaded in the cargo bike from Figure 1.7. 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. This is a classic example of a combinatorial problem.

$$f_1 = \sum_{i=0}^{n} E_i$$

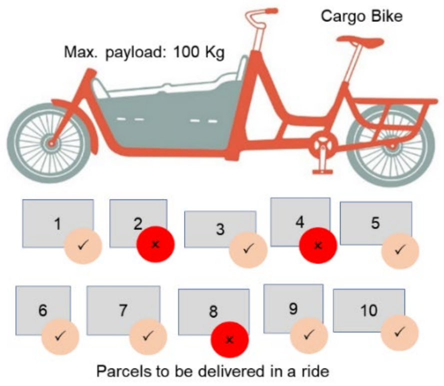
Equation 1.5

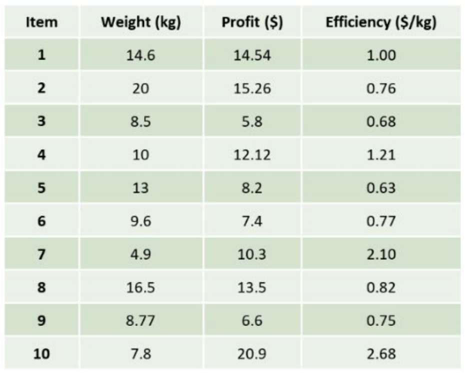
where n is the total number of packages and E_i is the efficiency of package *i*

$$f_2 = \left|\sum_{i=0}^{n} w_i - C\right|, \quad 50 \text{ is added } iff \left[\sum_{i=0}^{n} w_i > C\right]$$

Equation 1.6

where ωᵢ is the weight of package *i* and 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.





Item	Weight (kg)	Profit ($)	Efficiency ($/kg)
1	14.6	14.54	1.00
2	20	15.26	0.76
3	8.5	5.8	0.68
4	10	12.12	1.21
5	13	8.2	0.63
6	9.6	7.4	0.77
7	4.9	10.3	2.10
8	16.5	13.5	0.82
9	8.77	6.6	0.75
10	7.8	20.9	2.68

Figure 1.7 The cargo bike loading problem is an example of a problem with a soft constraint. While the weight of the packages can exceed the bike's capacity, a penalty will be applied when the bike is overweight.

Soft constraints can also be used to make the search algorithm more adaptive. For example, the severity of the penalty can be dynamically changed as the algorithm progresses, imposing less strict penalties at first to encourage exploration, while becoming more severe near the end to generate a result largely bound by the constraint.

1.4 Well-structured problems vs. Ill-structured problems

We can further classify optimization problems based on their structure, and the procedure that exists (or doesn't exist) for solving them. The following subsections introduced well-structured and ill-structured problems.

1.4.1 Well-structured problems (WSP)

In "The Structure of Ill Structured Problems" (*Artificial Intelligence, 1973*), Herbert Simon describes the six main characteristics of WSPs:

- There is a definite criterion for testing any proposed solution and a mechanizable process for applying the criterion.
- There is at least one problem space in which the initial problem state, the goal state, and all other states that may be reached, or considered, in the course of attempting a solution of the problem can be represented.
- Attainable state changes (legal moves) can be represented in a problem space, as transitions from given states to the states directly attainable from them. But considerable moves, whether legal or not, can also be represented—that is, all transitions from one considerable state to another.

- Any knowledge that the problem solver can acquire about the problem can be represented in one or more problem spaces.
- If the actual problem involves acting upon the external world, then the definition of state changes and of the effects upon the state of applying any operator reflects with complete accuracy in one or more problem spaces the laws (laws of nature) that govern the external world.
- All these conditions hold in the strong sense that the basic processes postulated require only practicable amounts of computation and the information postulated is effectively available to the processes—that is, available with the help of only practicable amounts of search.

Assume that we are planning a robotic pick-and-place task in an inspection system. In this scenario, the robot waits until receiving a signal from a presence sensor, which indicates the existence of a defected workpiece over the conveyer belt. The robot stops the conveyer belt and picks the defected piece and deposits it in a waste box. The robot reactivates the movement of the conveyer belt after depositing the defected piece. After the operation, the robot returns to its initial position and the cycle repeats itself again. As illustrated in Figure 1.8, this problem has the following well-structured components:

- **Feasible States:** position and speed of the robot arm and orientation and status (open or close and orientation) of its end-effector (gripper)
- **Operator (successor):** robot arm motion control command to move from one point to another following a certain singularity-free trajectory (positions or joint angles in space and motion speed) and end-effector control (orientation and open or close).
- **Goal:** pick and place a defected workpiece regardless its orientation
- **Solution/Path:** optimal sequence through state space for fastest pick-and-place operation
- **Stopping Criteria:** defected workpiece is picked from the conveyer belt and is placed in the waste box and robot returns to home position
- **Evaluation Criteria:** pick-and-place duration and/or success rate of pick-and-place process.

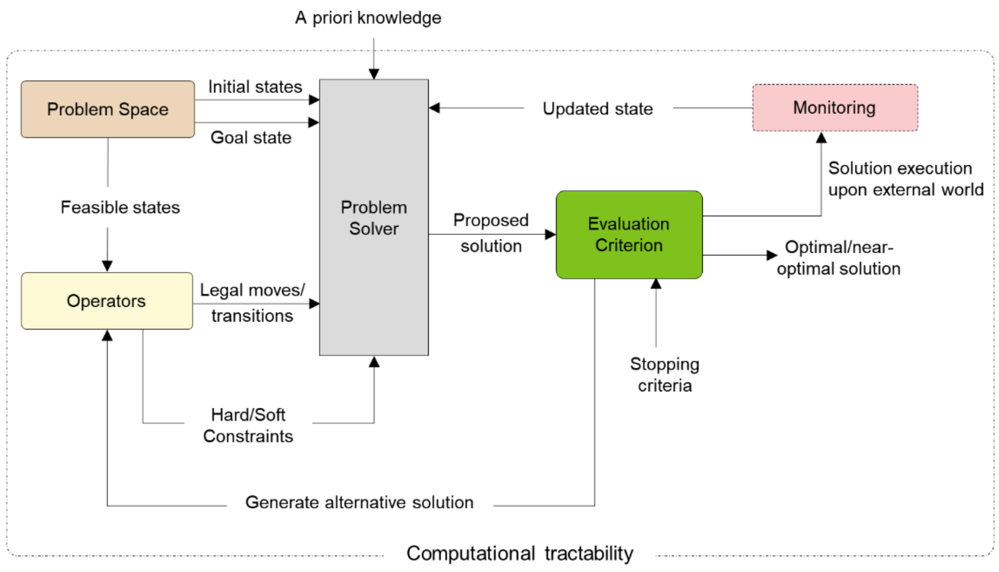


Figure 1.8 A well-structured problem will have a defined problem space, operators, and evaluation criteria.

As you can see, the work environment is highly structured, static and fully observable. An optimal pick-and-place plan can be generated and executed with high level of certainty. This pick-and-place problem can be considered as WSP.

1.4.2 Ill-structured Problems (ISP)

Ill-structured problems (ISPs) are complex discrete/continuous problems without algorithmic solutions/general problem solvers. Ill-structured problems are characterized by one or more of the following characteristics:

- Problem space with different views of problems, unclear goal, multimodality, and dynamic nature.
- Lack of exact differentiable mathematical models and/or well-proven algorithmic solutions
- Solutions are contradictory, consequence difficult to predict and risk difficult or impossible to calculate resulting in lack of clear evaluation criteria
- Considerable data imperfection in terms of uncertainty, partial observability, vagueness, incomplete information, ambiguity and/or unpredictability makes monitoring the execution of the solutions difficult and sometimes impossible.
- Computational intractability

Example: Assume that we need to find the optimal dispatching of four elevators to serve users between ten floors as illustrated in Figure 1.9. This is a classic example of a problem too large to solve using traditional means.

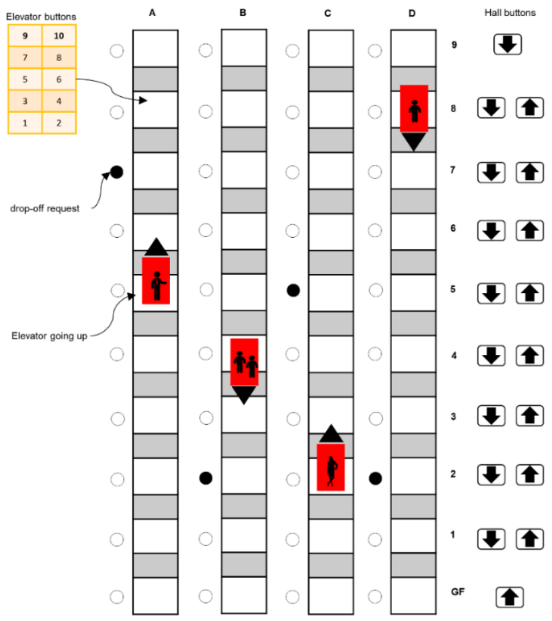


Figure 1.9 Elevator dispatching problem. With four elevator cars and ten floors, this problem has 10^{22} possible states.

In this problem, the following objective functions can be considered in this optimal dispatching problem:

- Minimizing the average waiting time: how long the user waits before getting on an elevator
- Minimizing the average system time: how long the user waits before being dropped off at the destination floor
- Minimizing the percentage of the users whose waiting time exceeds 60 seconds.
- Ensuring fairness in serving all the users of the elevators.

This optimal dispatching problem is an example of ISP as the problem space has a dynamic nature and partial observability; it is impossible to predict the user calls and destinations. Moreover, the search space is huge due to the extremely high number of possible states. There are 10^{22} possible states:

- 2^{18} possible combinations of the 18 hall call buttons, i.e., up, and down buttons, at each floor except the first and the last floor.
- 2^{40} possible combinations of the 40 elevator buttons if each elevator has 10 buttons.
- 18^{4} possible combinations of the position and directions of the elevators.

The total number of states for a problem as simple as dispatching this small building's elevators is more than the number of stars in the universe!

1.4.3 WSP but practically ISP

The Traveling Salesman Problem (TSP) is an example of a problem that may be well-structured in principle, but in practice becomes ill-structured. This is because of the impractical amount of computational power required to solve this problem in real-time.

Example: Assume that there is a traveling salesman assigned to visit a list of *n* cities. The salesman would like to make sales calls to all these cities in the minimum amount of time, as salespeople are generally paid by commission rather than hourly. Furthermore, the tour of the cities may be asymmetric; the time it takes to go from city A to city B may not be the same as the reverse, due to infrastructure, traffic patterns, and one-way streets. For example, with 13 cities to visit, the problem may seem trivial at first. However, upon closer examination it is revealed that the search space for this TSP results in *13!=6,227,020,800* different possible routes to be examined in case of using naive algorithms! However, dynamic programming algorithms enable reduced complexity as we will see in the next chapter.

This book largely focuses on ISPs and WSPs but practically ISPs kinds of problems, for a few reasons:

- Well-structured problems tend to have well-known solving algorithms that often provide a trivial, step-by-step procedure for solving WSPs. As such, there often already exist very efficient and well-known solutions to these kinds of problems. Moreover, several WSPs can be also solved using derivative-based generic solvers.
- The amount of computational power needed to solve well-structured problems is often negligible or very manageable at worst. Especially with the continued improvement of consumer-grade computers, not to mention the vast resources available through cloud computing and distributed processing, well-structured problems often do not have to settle for "near-optimal" solutions due to computational bottlenecks.
- Most problems in the real world are ill-structured problems, as the problem scope, state, and environment are dynamic and partially observable with certain degrees of uncertainties. Solutions or algorithms for ill-structured problems therefore have much more applicability to real-world scenarios, and there is a greater incentive to finding solutions to these problems.

Most of the algorithms explored in this book are derivative-free and stochastic; they use randomness in their parameters and decision processes. These are often well-suited to solving ill-structured problems, as the randomness of their initial states and operators allow the algorithms to escape local minima and find optimal or near-optimal solutions. On the other hand, deterministic algorithms use well-defined and procedural paths to reach solutions, and generally are not well suited for ISPs as they either cannot work in unknown search spaces or are unable to return solutions in a reasonable amount of time.

1.5 Search Algorithms and the Search Dilemma

The goal of any optimization method is to assign values to decision variables so that the objective function is optimized. To achieve this, optimization algorithms search the solution space for candidate solutions. Constraints are simply limitations on specific regions in the search space. Thus, all optimization techniques are in reality just search methods, where the goal is to find feasible solutions to satisfy constraints and maximizes (or minimizes) the objective functions. We define “search” as the systematic examination of feasible states, starting from the initial state, and ending (hopefully) at the goal state. However, while we explore the feasible search space, the question is if we find a few reasonably good neighboring solutions, should we exploit this region or should we explore more looking for better solutions in other regions of the feasible search space?

This exploration-exploitation or diversification-intensification dilemma is one of the most important problems in search and optimization and in life as well. We apply exploration-exploitation tactics in our life. When we move to a new city, we start by exploring different stores and restaurants and then focus on shortlisted options around us. During midlife crisis, some middle-aged individuals start to feel bored getting stuck in daily routine and lifestyle without clear satisfactory accomplishment and tend to take more explorative actions. US immigration system tries to avoid exploiting specific segments of applicants (e.g., family, skilled-workers, refugees and asylees) and enables more diversity through computer-generated lottery. In social insects like honeybees, foraging for food sources is performed by two different worker groups, recruits and scouts (5-25% of the foragers). Recruits focus on a specific food source while scouts are novelty seekers who keep scouting around for rich nectar. In search and optimization, exploration-exploitation dilemma represents the tradeoff between exploring new unvisited states/solutions in the search space and exploiting the elite solutions found in a certain neighborhood in the search space (Figure 1.10).

Exploration (or diversification) is the process of investigating new regions in the feasible search space with the hope of finding other promising solutions. On the other hand, exploitation (or intensification) is the process of directing the search agent to focus on an attractive region of the search space where good solutions have been already found.

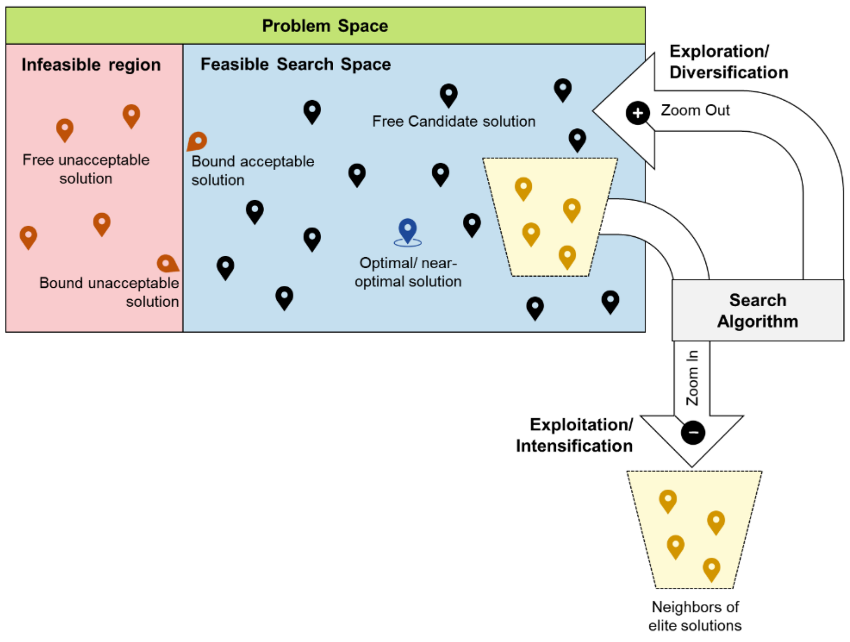

Problem Space
Infeasible region
Feasible Search Space
Free unacceptable solution
Bound unacceptable solution
Bound acceptable solution
Free Candidate solution
Optimal/ near-optimal solution
Exploration/ Diversification
Zoom Out
Search Algorithm
Zoom In
Exploitation/ Intensification
Neighbors of elite solutions

Figure 1.10 Search Dilemma. There is always a tradeoff between branching out to new areas of the search space or focusing on an area with known "good" or elite solutions.

Local search algorithms are exploitation/intensification algorithms that can be easily trapped in local optima if the search landscape is multimodal. On the other extreme, random search algorithms keep exploring the search space with a high chance of reaching global optima at the cost of the impractical search time. Generally speaking, explorative algorithms can find global optima at the cost of processing time, while exploitative algorithms risk getting stuck at local minima.

1.6 Summary

- Optimization is a search process for finding the "best" solutions to a problem that provide the best objective function values, possibly subject to a given set of hard (must be satisfied) and soft (desirable to satisfy) constraints.
- Ill-structured problems are complex, discrete, or continuous problems without exact differentiable mathematical models and/or algorithmic solutions or general problem solvers. They usually have dynamic and/or partially observable large search spaces that cannot be handled by classical optimization methods.
- Stochastic algorithms are non-greedy algorithms that explicitly use randomness to find the global optimal or near-optimal solution in a reasonably practical time. In many real-life applications, quickly finding a near-optimal solution is better than spending a large amount of time in search for an optimal solution.
- Two key concepts you'll see frequently in future chapters are exploration/diversification and exploitation/intensification. Handling this search dilemma by achieving a trade-off between exploration and exploitation will allow the algorithm to find optimal or near-optimal solutions without getting trapped in local optima in an attractive region of the search space and without spending large amount of time.

2 A Deeper Look at Search and Optimization

This chapter covers

- **Classifying optimization problems based on different criteria.**
- **Classifying search and optimization algorithms based on the way the search space is explored and how deterministic the algorithm is.**
- **Introducing heuristics, meta-heuristics, and heuristic search strategies.**
- **A first look at nature-inspired search and optimization algorithms.**

Before we dive into the problems and algorithms that we hinted at in Chapter 1, it useful to first understand how we "talk" about these problems and algorithms. Classifying a problem allows us to group similar problems together, and potentially exploit already-existing solutions. For example, a Travelling Salesman Problem(TSP) problem involving geographic values (i.e., cities and roads) may be used as a model to find the minimum length of the wires connecting the pins in very large-scale integration (VLSI) design. The same can be said for classifying the algorithms themselves, as grouping algorithms with similar properties allows us to easily identify the right algorithm to solve the problem and meeting expectation in terms of quality of the solution and permissible search time for example. Throughout this chapter, we discuss common classifications of optimization problems and algorithms. Heuristics and meta-heuristics are also introduced as general algorithmic framework or high-level strategies that guide the search process. Many of these strategies are inspired from nature so we shed some lights on nature inspired algorithms. Let's start by discussing how we can classify optimization problems based on different criteria in the first section.

2.1 Optimization Problem Classification

Optimization is everywhere! In everyday life, you may face different kinds of optimization problems. For example, you may like to set the AC thermostat at a certain degree to stay comfortable and at the same time to save energy. You may select light fixtures and adjust the light level to reduce energy costs. When you start driving your electric vehicle (EV), you search for the fastest route and/or most energy-efficient route that takes you to your destination. Before arriving to your destination, you may look for a parking spot that is affordable and provides shortest walking distance to the destination and offers EV charging and preferably is underground. These optimization problems have different levels of complexities that mainly depend on the type of the problem. As mentioned in the previous chapter, the process of optimization selects decision variables from a given feasible search space in such a way as to optimize (minimize or maximize) a given objective function, or in some cases, multiple objective functions. Search and optimization problems are characterized by three main components, namely, decision variables or design vector, objective functions or criteria to be optimized, and a set of hard and soft constraints to be satisfied. The nature of these three components, the permissible time to solve the problem and the expected quality of the solutions lead to different types of optimization problems, as shown in Figure 2.1.

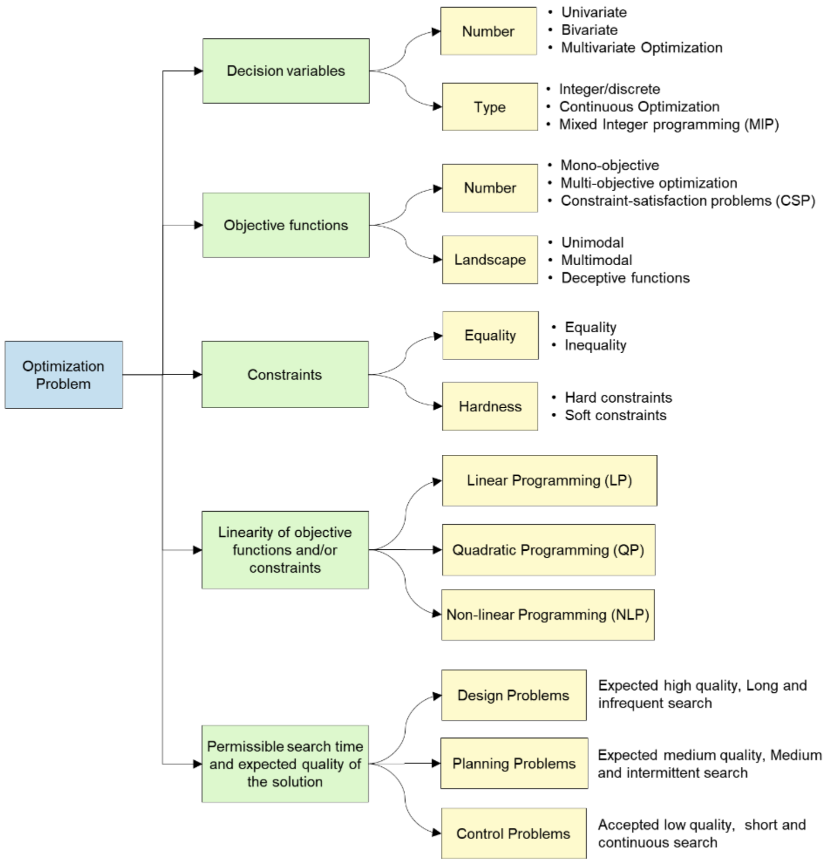

Optimization Problem
Decision variables
Number
• Univariate
• Bivariate
• Multivariate Optimization
Type
• Integer/discrete
• Continuous Optimization
• Mixed Integer programming (MIP)
Objective functions
Number
• Mono-objective
• Multi-objective optimization
• Constraint-satisfaction problems (CSP)
Landscape
• Unimodal
• Multimodal
• Deceptive functions
Constraints
Equality
• Equality
• Inequality
Hardness
• Hard constraints
• Soft constraints
Linearity of objective functions and/or constraints
Linear Programming (LP)
Quadratic Programming (QP)
Non-linear Programming (NLP)
Permissible search time and expected quality of the solution
Design Problems
Expected high quality, Long and infrequent search
Planning Problems
Expected medium quality, Medium and intermittent search
Control Problems
Accepted low quality, short and continuous search

Figure 2.1 Optimization problem classification. An optimization problem can be broken down into its constituent parts, which form the basis of the classification of such problems.

The following subsections explain these types in greater detail and provide examples for each type of optimization problem.

2.1.1 Number and Type of Decision Variables

According to the number of decision variables, optimization problems can be broadly grouped into univariate (single variable) or multivariate problems (multiple decision variables). For example, vehicle speed, acceleration, and tire pressure are among the parameters that impact a vehicle’s fuel economy, where fuel economy refers to how far a vehicle can travel on a specific amount of fuel. According to the US Department of Energy, controlling the speed and acceleration of a vehicle can improve its fuel economy by 15%–30% at highway speeds, and 10%–40% in stop-and-go traffic. A study by the National Highway Traffic Safety Administration (NHTSA) in the USA found that a 1% decrease in tire pressure correlated to a

0.3% reduction in fuel economy. If we are only looking for the optimal vehicle speed for maximum fuel economy, the problem is a univariate optimization problem. Finding the optimal speed and optimal acceleration for maximum fuel economy is a bivariate optimization problem, while finding optimal speed, acceleration, and tire pressure is a multivariate problem.

Problem classification also varies according to the type of the decision variables. A continuous problem involves continuous-valued variables where $x_j \in R$. In contrast, if $x_j \in Z$, the problem is an integer, or discrete optimization problem. A mixed-integer problem has both continuous-valued and integer-valued variables. For example, optimizing elevator speed and acceleration (continuous variables) and the sequence of picking up passengers (discrete variable) is a mixed-integer problem. Problems where the solutions are sets, combinations, or permutations of integer-valued variables are referred to as combinatorial optimization problems.

Combination vs. Permutation

Combinatorics is the branch of mathematics studying both combination and permutation of a set of elements. The main difference between combination and permutation is the order. If the order of the elements doesn't matter, it is a combination, and if the order does matter, it is a permutation. Thus, permutations are ordered combinations. Depending on whether repetition of these elements is allowed or not, we can have different forms of combinations and permutations as shown in Figure 2.2.

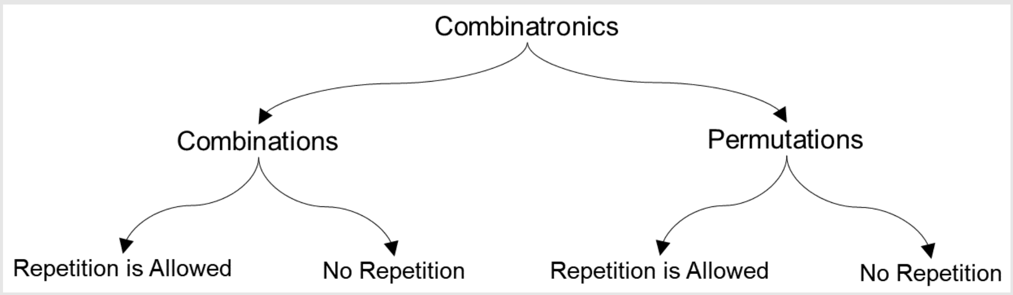


Figure 2.2 Combinations and permutations. Both combinations and permutations have variants with and without repetition. The difference lies in the fact that permutations respect order and are thus ordered combinations.

For example, assume that we are designing a fitness plan to include multiple fitness activities. Five types of fitness exercises are available to be included in the fitness plan, namely, jogging, swimming, biking, yoga, and aerobics. In a weekly plan, if we are to choose only 3 out of these 5 exercises and the repetition is allowed, the number of possible combinations will be: (n+r-1)!/r!(n-1)! = (5+3-1)!/3!(5-1)! = 7!/(3!×4!) =35. This means that we can generate 35 different fitness plans by selecting 3 out of the available 5 exercises and by allowing repetition.

However, if repetition is not allowed, the number of possible combinations will be: C(n,r)=n!/r!(n-r)!=5!/(3!×2!)=10. This formula is often called "n choose r" (such as "5 choose 3') and is also known as the "Binomial Coefficient". This means that we can generate only 10 plans if we don't want to repeat any of the exercises.

In both cases of combination with and without repetition, the fitness plan doesn't include the order of performing the included exercises. If we are to respect specific order, the plan in this case will take the form of a permutation. If repeating exercises is allowed, the number of possible permutations to select 3 exercises out of the 5 available exercises will be: $n^r=5^3=125$. However, if repetition is not allowed, the number of possible permutations will be: P(n,r)=n!/(n-r)!=5!/(5-3)!=60.

Combinatorics can be implemented fairly easily in Python using code from scratch, but there are excellent resources available in the form of libraries, such as *sympy*. For example, the Binomial Coefficient can be calculated in *sympy* using the following simple code:

```
from sympy import binomial
print(binomial(5,3))
```

See Appendix-A and the documentation for *sympy* for more on implementing combinatorics in Python.

The Traveling Salesman Problem (TSP) is a common example of combinational problems whose solution is a permutation, or a sequence of cities to be visited. In TSP, given n cities, a traveling salesman must visit the n cities and then return home, making a loop (round trip). The salesman would like to travel in the most efficient way (fastest, cheapest, shortest distance, or some other criterion). The search space in TSP is very large. For example, let's assume the traveling salesman is to visit the 13 major cities in the Greater Toronto Area (GTA) as illustrated in Figure 2.3.

TSP can be classified into symmetric TSP (STSP) and asymmetric (ATSP). In the STSP, the distance between two cities is the same in each opposite direction, forming an undirected graph. This symmetry halves the number of possible solutions. ATSP is a strict generalization of the symmetric version. In the ATSP, paths may not exist in both directions or the distances might be different, forming a directed graph. Traffic collisions, one-way streets, bridges and airfares for cities with different departure and arrival fees are examples of how this symmetry could break down.

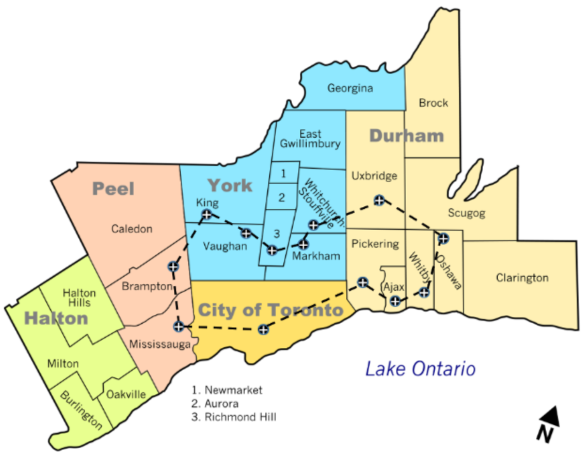


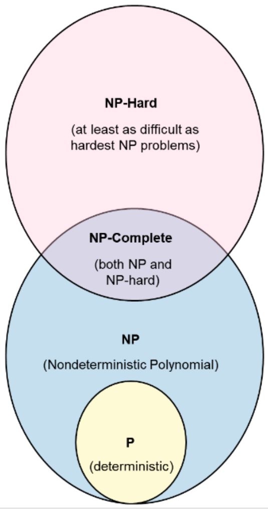
Figure 2.3 TSP in Greater Toronto Area (GTA). The traveling salesman must visit all the 13 cities, and wishes to select the "best" path, whether that be distance, time, or some other criterion.

The naive solution's complexity is O(n!). These means that there are *n!=13!= 6,227,020,800* possible tours in case of ATSP. This makes both STSP and ATSP, NP-hard problems. However, dynamic programming (DP) algorithms enable reduced complexity. Dynamic programming is a method of solving optimization problems by breaking them down into smaller subproblems and solving each subproblem independently. For example, the complexity of Bellman-Held-Karp algorithm [1] is of $O(2^n \times n^2)$. Different solvers and algorithms with different level of computational complexities and approximation ratios such as Concorde TSP Solver, 2-opt/3-opt algorithm, branch and bound algorithms, Christofides algorithm or Christofides–Serdyukov algorithm, Bellman–Held–Karp algorithm, Lin-Kernighan algorithm, metaheuristics-based algorithms, graph neural networks or deep reinforcement learning methods. For example, Christofides algorithm [2] is a polynomial-time approximation algorithm that produces a solution to the TSP that is guaranteed to be no more than 50% longer than the optimal solution with a time complexity of O(n^3). See Appendix-A for solution of TSP using Christofides algorithm implemented in *networkx*. We will discuss how to solve TSP using a number of these algorithms throughout this book.

TSP is used as a platform for the study of general methods that can be applied to a wide range of discrete optimization problems. These problems include, but are not limited to, microchips manufacturing, permutation flow shop scheduling, arranging school bus routes to pick up children in a school district, assignment of routes for airplanes, transportation of farming equipment, scheduling of service calls, meal delivery, and the routing of trucks for parcel delivery and pickup. For example, the Capacitated Vehicle Routing Problem (CVRP) is a generalization of the TSP where one has to serve a set of customers using a fleet of homogeneous vehicles based at a common depot. Each customer has a certain demand for goods which are initially located at the depot. The task is to design vehicle routes starting and ending at the depot such that all customer demands are fulfilled. Later in this book, several examples are given to show how to solve TSP and its variants using stochastic approaches.

Problem Types

Decision problems are commonly classified based on their levels of complexities. These classes can be also applied to optimization problems given the fact that optimization problems can be converted into decision-making problems. For example, the optimization problem whose objective is to find an optimal or near-optimal solution within a feasible search space can be paraphrased as a decision-making problem, which answers the question: "Is there is an optimal or a near-optimal solution within the feasible search space?" The answer will be {"yes" or "no"} or {"True" or "False"}. A generally-accepted notion of algorithm's efficiency is that its running time is polynomial. Problems that can be solved in polynomial time are known as tractable. Figure 2.4 shows different types of problems and gives examples of commonly used benchmark/toy and real-life application of each type.


NP-Hard
(at least as difficult as hardest NP problems)
NP-Complete
(both NP and NP-hard)
NP
(Nondeterministic Polynomial)
P
(deterministic)

Problem Class	Description	Examples of Benchmark/Toy Problems	Examples of Real-world Problems
NP-Hard	A problem is NP-hard iff a polynomial-time algorithm for it implies a polynomial-time algorithm for every problem in NP. NP-hard problems are at least as hard as any in NP	• Travelling salesman problem (TSP) • Graph coloring • Set covering and partitioning	• Vehicle routing • Microchips manufacturing • Mobile Radio Frequency Assignment • Map Coloring • Register Allocation • Political/school/healthcare districting
NP-Complete	NP-complete problems are NP-hard problems but solutions can be verified in polynomial time.	• Bin packing problem (BBP) • Knapsack problem • Subset Sum Problem (SSP) • Hamiltonian circuit • Vertex cover problem • N-puzzle • Boolean Satisfiability (SAT)	• Container/truck loading • Telecommunication traffic overload control • Design of queuing network models in manufacturing • Combinational Equivalence Checking • Job Shop Scheduling
NP/co-NP	Decision problems whose "YES" answer (or "No" in case of co-NP) can be verified in polynomial time, if we already have the proof (or witness)	• Integer Factorization • Graph Isomorphism • Maximum flow problem	• Image matching • Direct or indirect interaction in social media • Protein structure analysis
P	P problems are deterministic in nature and can be solved in polynomial time	• Minimum spanning tree (MST) • Shortest path problem	• Network designs • Some single machine scheduling problems

Figure 2.4 Problem classes based on hardness and completeness. Problems can be categorized into NP-hard, NP-complete, NP, or P.

For example, a complexity class P represents all decision problems that can be solved in polynomial time by deterministic algorithms (i.e., algorithms that do not guess at a solution). The NP or Nondeterministic Polynomial problems are problems whose solutions are hard to find but easy to verify and are solved by a non-deterministic algorithm in polynomial time. NP-complete problems are problems that are both NP-hard and verifiable in polynomial time. Finally, a problem is NP-hard if it is at least as hard as the hardest problem in NP-complete. NP-hard problems are usually solved by approximation or heuristic solvers as it is hard to find efficient exact algorithms to solve such kind of problems.

Clustering is an example of combinatorial problems whose solution takes the form of a combination where the order doesn't matter. In clustering, given *n* objects, we need to group them in *k* groups (clusters) in such a way that all objects in a single group or cluster have a "natural"' relation to one another, and objects not in the same group are somehow different. This means that the objects will be grouped based on some similarity or dissimilarity metric. The following formula is known as a Stirling number of the second kind (or Stirling partition number) and gives the number of ways to partition a set of n objects into k non-empty subsets:

$$S(n,k) = \left\{ {n \atop k} \right\} = \frac{1}{k!}\sum_{i=0}^{k}(-1)^i \binom{k}{i}(k-i)^n$$

Equation 2.1

Let's consider smart cart clustering as an example. Shopping and luggage carts are commonly found in shopping malls and large airports. Shoppers or travelers pick up these carts at designated points and leave them in arbitrary places. It is a considerable task to re-collect them, and it is therefore beneficial if a “smarter” version of these carts could draw themselves together automatically to the nearest assembly points as illustrated in Figure 2.5.

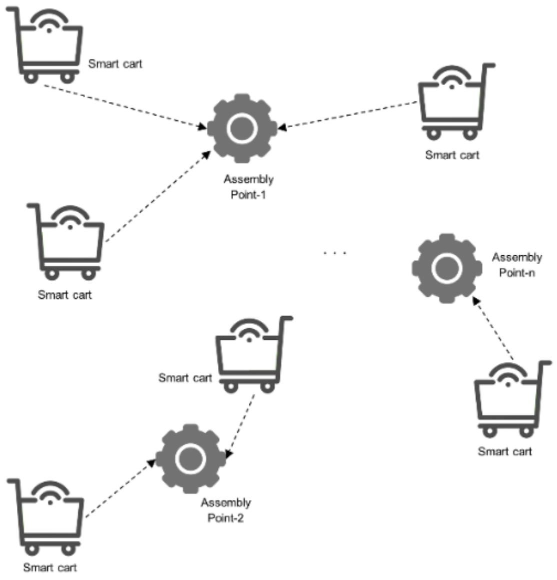


Figure 2.5 Smart Cart Clustering. Unused shopping or luggage carts congregate near designated assembly points to make collection and redistribution easier.

In practice, this problem is considered an NP-hard problem, as the search space can be very large based on the number of available carts and number of assembly points. In order to cluster these carts effectively, the centers of clustering (centroids) must be found. The carts in each cluster will then be directed to the assembly point closest to the centroids. For example, assume that there are 50 carts to be clustered around 4 assembly points. This means that n=50 and c=4. Stirling numbers can be generated using the *sympy* library. To do so, simply call the *stirling* function on two numbers n and k.

```
from sympy.functions.combinatorial.numbers import stirling
print(stirling(50,4))
print(stirling(100,4))
```

The number is 5.3×10^{28} and if n is increased to 100, the number becomes 6.7×10^{58}. Enumerating all possible partitions for large problems is practically not feasible.

2.1.2 Landscape and Number of Objective Functions

Objective function's landscape represents the distribution of values of the objective function in the feasible search space. When you walk over this landscape, you find the optimal solution or the global minima in the lowest valley of the landscape assuming that you are dealing with a minimization problem or in the highest peak of the landscape in case of a maximization problem. According to the landscape of the objective function, if there exists only one clear global optimal solution, the problem is unimodal (e.g., convex and concave functions). On the other hand, in a multimodal problem, more than one optimum exists. The objective function is called deceptive when the global minimum lies in a very narrow valley and at the same time there exists a strong local minimum with a wide basin of attraction such that the value of objective function is close to the value of objective function at global minimum [3]. Figure 2.6 is a 3D visualization of the landscapes of unimodal, multimodal, and deceptive functions generated using Python generated by Listing 2.1. Complete listing is available in the GitHub repo of the book.

Listing 2.1 Examples of Objective Functions

```
import numpy as np
import math
import matplotlib.pyplot as plt

def objective_unimodal(x, y):    #A
    return x**2.0 + y**2.0

def objective_multimodal(x, y):    #B
    return np.sin(x) * np.cos(y)

def objective_deceptive(x, y):    #C
    return (1-(abs((np.sin(math.pi*(x-2))*np.sin(math.pi*(y-2)))/(math.pi*math.pi*(x-2)*(y-
       2))))**5)*(2+(x-7)**2+2*(y-7)**2)
```

#A Unimodal function
#B Multimodal function
#C Deceptive function [3]

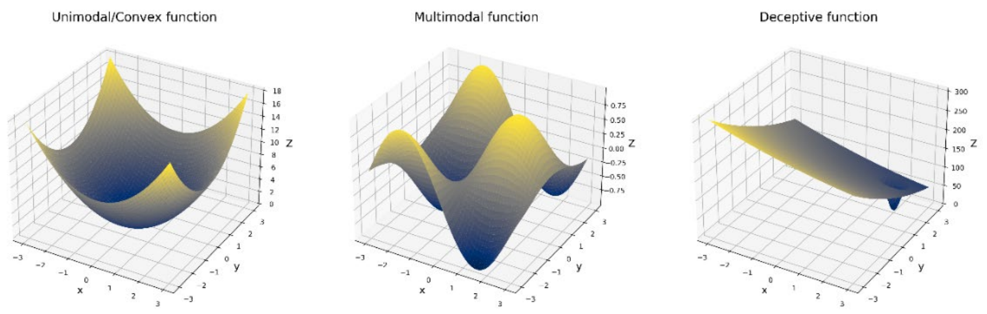


Figure 2.6 Unimodal, multimodal, and deceptive functions. Unimodal functions have one global optimum, while multimodal functions can have many. Deceptive functions contain false optima close to the value of objective function at global minimum, which can cause some algorithms to get stuck.

If the quantity to be optimized is expressed using only one objective function, the problem is referred to as a mono-objective or single-objective optimization problem (such as convex or concave functions). A multi-objective problem specifies multiple objectives to be simultaneously optimized. Problems without an explicit objective function are called Constraint-Satisfaction Problems (CSP). The goal in this case is to find a solution that satisfies a given set of constraints. The N-queen problem is an example of CSP. In this problem, the aim is to put *n* queens on an *n*×*n* board with no two queens on the same row, column, or diagonal, as illustrated in Figure 2.7. In this 4-queen problem, there are 5 conflicts in the first state ({Q1,Q2}, {Q1,Q3},{Q2,Q3},{Q2,Q4} and {Q3,Q4}). After moving Q4, the number of conflicts reduces by 2 and after moving Q3, the number of conflicts is only 1, which is between Q1 and Q2.

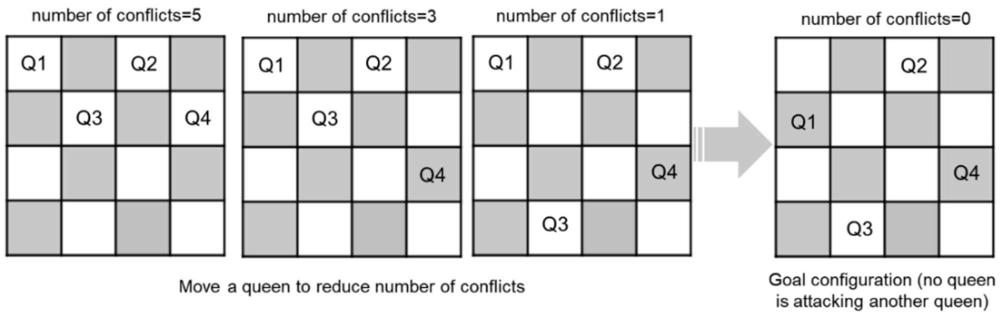

number of conflicts=5
number of conflicts=3
number of conflicts=1
number of conflicts=0
Q1
Q2
Q3
Q4
Move a queen to reduce number of conflicts
Goal configuration (no queen is attacking another queen)

Figure 2.7 N-Queen Problem. This problem has no objective function, rather, only a set of constraints that must be satisfied.

By keep moving or placing the pieces, we can reach to the goal state where number of conflicts is zero, which means that there is no any queen that could attach any other queen horizontally, vertically or diagonally. Listing 2.2 is a Python implementation of the 4-Queen problem.

Listing 2.2 N-Queen CSP

```
from copy import deepcopy
import math
import matplotlib.pyplot as plt
import numpy as np

board_size = 4
board = np.full((board_size, board_size), False)    #A

def can_attack(board, row, col):
    if any(board[row]):    #B
        return True    #B

    offset = col - row    #C
    if any(np.diagonal(board, offset)):    #C
        return True    #C
    offset = (len(board) - 1 - col) - row    #C
    if any(np.diagonal(np.fliplr(board), offset)):    #C
        return True    #C

    return False

board[0][0] = True    #D
col = 1
states = [deepcopy(board)]
while col < board_size:
    row = 0
    while row < board_size:
        if not can_attack(board, row, col):    #E
            board[row][col] = True
            col += 1
            states.append(deepcopy(board))
            break
        row += 1
        if row == board_size:    #F
            board = np.delete(board, 0, 1)
            new_col = [[False]] * board_size
            board = np.append(board, new_col, 1)
            states.append(deepcopy(board))
            col -= 1
            continue
```

#A Create an *nxn* board
#B Check existing on the same row
#C Check diagonals
#D First column is trivial
#E The piece can be placed in this column
#F The piece cannot be placed in this column

We define a function called *can_attack* to detect if a newly placed piece can attack a previously placed piece. A piece can attack another piece if it is in the same row, column, or diagonal.

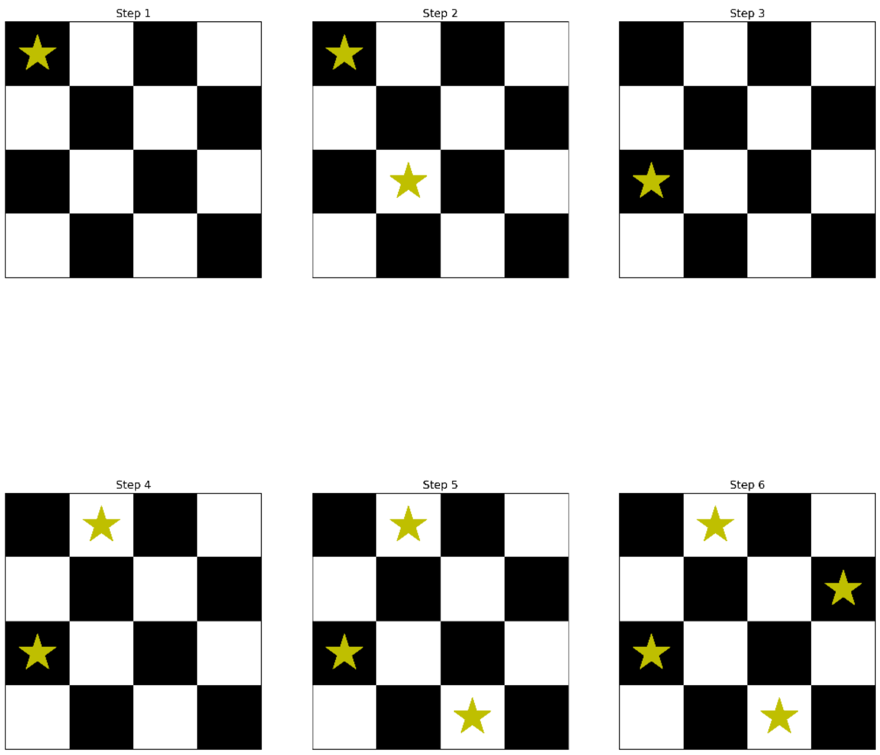


Figure 2.8 N-Queens Solution. The first piece is trivially placed in the first position. The second piece must be placed either in the 3rd position or the 4th position, as the first two can be attacked. By placing it in the 3rd position however, the 3rd piece cannot be placed. Thus, the first piece is removed (the board is "slid" one column over), and we reattempt. This continues until a solution is found.

The full code for this problem, including the code used to generate visualizations, can be found in the code file for Listing 2.2. The solution algorithm is as follows:

1. Moving from top to bottom in a column, the algorithm attempts to place the piece while avoiding conflicts. For the first column, this will default to Q1 = 0.
2. Moving to the next column, if a piece cannot be placed at row 0, it will be placed at row 1, and so on.
3. When a piece has been placed, the algorithm moves to the next column.
4. If it is impossible to place a piece in a given column, the first column of the entire board is removed, and the current column is reattempted.

Constraint programming solvers available in Google OR-Tools can be also used to solve this *n*×*n* queen problem. Listing 2.3 shows the steps of the solution using OR-Tools.

Listing 2.3 Solving N-Queen problem using OR-Tools

```
import numpy as np
import matplotlib.pyplot as plt
import math
from ortools.sat.python import cp_model #A

board_size = 4 #B

model = cp_model.CpModel() #C

queens = [model.NewIntVar(0, board_size - 1, 'x%i' % i) for i in range(board_size)] #D

model.AddAllDifferent(queens) #E

model.AddAllDifferent(queens[i] + i for i in range(board_size))
model.AddAllDifferent(queens[i] - i for i in range(board_size))

solver = cp_model.CpSolver() #G
solver.parameters.enumerate_all_solutions = True #G
solver.Solve(model) #G

all_queens = range(board_size)    #F
state=[]
for i in all_queens:
    for j in all_queens:
        if solver.Value(queens[j]) == i:
            # There is a queen in column j, row i.
            state.append(True)
        else:
            state.append(None)

states=np.array(state).reshape(-1, board_size)
fig = plt.figure(figsize=(5,5)) #H
markers = [
    x.tolist().index(True) if True in x.tolist() else None
    for x in np.transpose(states)
] #H
res = np.add.outer(range(board_size), range(board_size)) % 2 #H
plt.imshow(res, cmap="binary_r") #H
plt.xticks([]) #H
plt.yticks([]) #H
plt.plot(markers, marker="*", linestyle="None", markersize=100/board_size, color="y") #H
```

#A imports a constraint programming solver that uses SAT (satisfiability) methods.
#B Set board size for nxn Queeen problem.
#C Define a solver
#D Define the variables. The array index represents the column, and the value is the row.
#E Define the constraint: all rows must be different
#F Define the constraint: no two queens can be on the same diagonal.
#G Solve the model.
#H Visualize the solution

Running this code produces the following output:

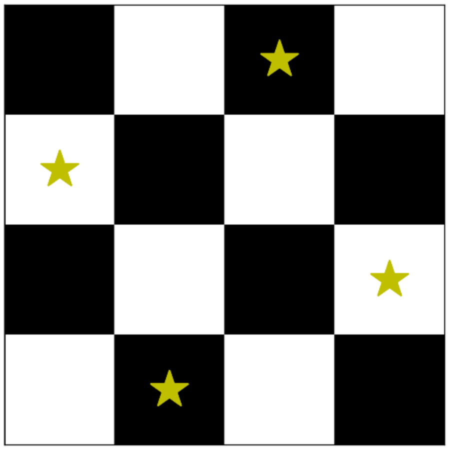


Figure 2.9 N-Queens solution using OR-Tools.

More information about Google OR-Tools is available in Appendix A.

2.1.3 Constraints

Constrained problems have equality and/or inequality hard or soft constraints. Hard constraints must be satisfied, while soft constraints are nice to satisfy (but are not mandatory). If there are no constraints to be considered aside from the boundary constraints, the problem is an unconstrained optimization problem. Let’s revisit the ticket pricing problem introduced in subsection 1.3.1 of the previous chapter. There a wide range of derivative-based solvers in Python that can handle such kind of differentiable mathematical optimization problem (see Appendix A). Listing 2.4 shows how to solve this simple ticket pricing problem using *SciPy*. *SciPy* is a library containing a collection of valuable tools for all things computation.

Listing 2.4 Optimal Ticket Pricing

```
import numpy as np
import scipy.optimize as opt
import matplotlib.pyplot as plt

def f(x): #A
    return -(-20*x**2+6200*x-350000)/1000

res=opt.minimize_scalar(f, method='bounded', bounds=[0, 250]) #B

print("Optimal Ticket Price ($): %.2f" % res.x)
print("Profit f(x) in K$: %.2f" % -res.fun)
```

#A The objective function, which is required by ‘minimize_scalar’ to be a minimization function
#B ‘bounded’ method is the constrained minimization procedure that finds the solution

Running this code produces the following output:

```
Optimal Ticket Price ($): 155.00
Profit f(x) in K$: 130.50
```

This code finds the optimal ticket price in the range between $0 and $250 that maximizes the profit. As you may have noticed the profit formula is converted into a minimization problem by adding a negative sign in the objective function to match with the *minimize* function in *scipy.optimize*. A minus sign was added in *print* function to convert it back into profit.

What if we imposed an equality constraint on this problem? Let's assume that due to incredible international demand for our event, we now consider using a different event planning company and opening up virtual attendance for our conference, so that international guests can also participate. Interested participants can now choose between attending the event in person or joining via live stream. All participants, whether in-person or virtual, will still receive a physical welcome package, which is limited to 10,000 units. Thus, in order to ensure a "full" event, we must either sell 10,000 in person tickets, 10,000 virtual tickets, or some combination thereof. This new event company is charging us a $1,000,000 flat rate for the event, and thus we want to sell as many tickets as possible (exactly 10,000). Below are the equations associated with this problem:

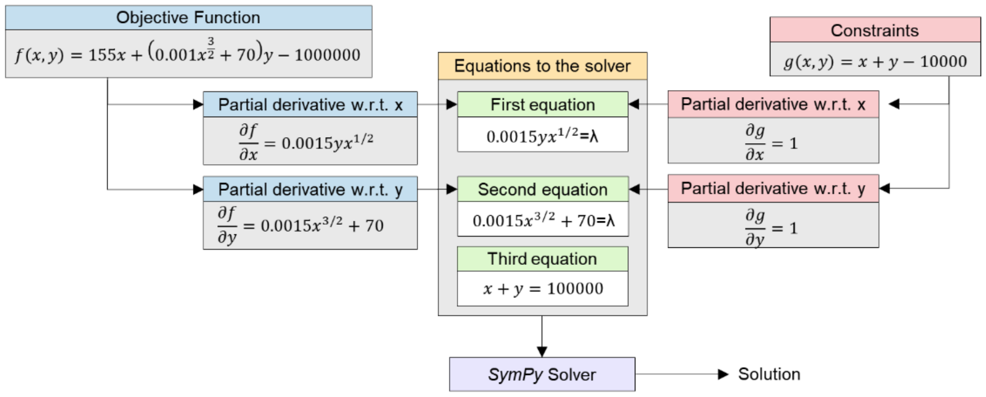
Let x be the number of physical ticket sales, and y be the number of virtual ticket sales. Additionally, let f(x,y) be the function for profits generated from the event, where

$$f(x,y)=155x+(0.001x^{3/2}+70)y-1000000$$

Equation 2.2

Essentially, we earn $155 profit on in-person attendance, and the profit for online attendance is $70, but increases by some amount the more physical attendance we have (let's say that as the event looks "more crowded", we can charge more for online attendees).

Assuming that we're adding the constraint function: $x+y\leq10000$, which shows that the combined ticket sales cannot exceed 10,000. The problem is now a bivariate mono-objective constrained optimization problem. It is possible to convert this constrained optimization problem to unconstrained optimization using Lagrange multiplier λ. We can use *sympy* to implement Lagrange multipliers to solve for the optimal mix of virtual and physical ticket sales. The idea is to take the partial derivatives of the objective functions and the constraints with respect to the decision variables x and y to form the unconstrained optimization equations to be used by the *SymPy* solver as illustrated in Figure 2.10.



Objective Function
$f(x,y) = 155x + \left(0.001x^{\frac{3}{2}} + 70\right)y - 1000000$
Partial derivative w.r.t. x
$\frac{\partial f}{\partial x} = 0.0015yx^{1/2}$
Partial derivative w.r.t. y
$\frac{\partial f}{\partial y} = 0.0015x^{3/2} + 70$
Equations to the solver
First equation
$0.0015yx^{1/2}$=λ
Second equation
$0.0015x^{3/2} + 70$=λ
Third equation
$x + y = 100000$
Constraints
$g(x,y) = x + y - 10000$
Partial derivative w.r.t. x
$\frac{\partial g}{\partial x} = 1$
Partial derivative w.r.t. y
$\frac{\partial g}{\partial y} = 1$
SymPy Solver
Solution

Figure 2.10 Steps of solving ticket pricing problem using Lagrange method

Listing 2.5 shows the Python implementation using *SymPy*. *SymPy* is an open-source Python library for symbolic mathematics. Capabilities include, but are not limited to, statistics, physics, geometry, calculus, equation solving, combinatorics, discrete math, cryptograph and parsing. See Appendix-A for more information about *SymPy*.

Listing 2.5 Maximizing Profits using Lagrange Multipliers

```
import numpy as np
import sympy as sym

x,y=sym.var('x, y', positive=True) #A

f=155*x+(0.001*x**sym.Rational(3,2)+70)*y-1000000 #B

g=x+y-10000 #C

lamda=sym.symbols('lambda') #D
Lagr=f-lamda*g

grad_Lagr=[sym.diff(Lagr,var) for var in [x,y]] #E
eqs=grad_Lagr+[g]

sol=sym.solve(eqs,[x,y,lamda], dict=True) #F

def getValueOf(k, L):
    for d in L:
        if k in d:
            return d[k]

# print the value of the objective function
profit=[f.subs(p) for p in sol]

print("optimal number of physical ticket sales: x = %.0f" % getValueOf(x, sol))
print("optimal number of online ticket sales: y = %.0f" % getValueOf(y, sol))
print("Expected profil: f(x,y) = $%.4f" % profit[0])
```

#A define decision variables
#B define the ticket pricing objective function
#C define equality constraint
#D Lagrange multiplier
#E taking the partial derivatives
#F solving these 3 equations in 3 variables (x,y,lambda) using sympy

By solving the above three equations, we get *x* and *y* values that correspond to the optimized quantities for virtual and physical ticket sales. From Listing 2.5, we can see that the best way to sell tickets is to sell 6,424 in-person tickets and 3,576 online ticket profits. This results in a maximum profit of $2,087,260.533.

2.1.4 Linearity of Objective Functions and Constraints

If all the objective functions and all associated constraint conditions are linear, the optimization problem is categorized as a linear optimization problem or Linear Programming Problem (LPP or LP), where the goal is to find the optimal value of a linear function subject to linear constraints. Blending problems are a typical application of mixed integer linear programming (MILP) where a number of ingredients are to be blended or mixed to obtain a product with certain characteristics or properties. In the animal feed mix problem described in [4], it is required to determine the optimum amounts of three ingredients to include in an animal feed mix. The possible ingredients, their nutritive contents (in kilograms of nutrient per kilograms of ingredient) and the unit cost are shown in Table 2.1.

Table 2.1 Animal Feed Mix Problem

Ingredients	Nutritive content and price of ingredients			
	Calcium (kg/kg)	Protein (kg/kg)	Fiber (kg/kg)	Unit cost (cents/kg)
Corn	0.001	0.09	0.02	30.5
Limestone	0.38	0.0	0.0	10.0
Soybean meal	0.002	0.50	0.08	90.0

The mixture must meet the following restrictions:

- Calcium — at least 0.8% but not more than 1.2%.
- Protein — at least 22%.
- Fiber — at most 5%.

The problem is to find the mixture that satisfies these constraints while minimizing cost. The decision variables are x_1, x_2 and x_3, which are proportions of Limestone, Corn and Soybean meal in the mixture respectively.

Objective Function: minimize $f = 30.5x_1 + 10x_2 + 90x_3$

Subject to the following constraints:

- Calcium limits: $0.008 <= 0.001x_1 + 0.38x_2 + 0.002x_3 <= 0.012$
- Protein constraint: $0.09x_1 + 0.5x_3 >= 0.22$
- Fiber constraint: $0.02x_1 + 0.08x_3 <= 0.05$
- Non-negativity restriction: $x_1, x_2, x_3 >= 0$

- Conservation: $x_1+x_2+x_3=1$

In this problem, both the objective function and the constraints are linear so we call it linear programming problem. There are several Python libraries that can be used for solving mathematical optimization problems (see Appendix A). Let’s consider solving the animal feed mix problem using PuLP. PuLP is a Python linear programming library. It allows users to define linear programming problems and to solve them using optimization algorithms such as COIN-OR's linear and integer programming solvers. See Appendix A for more information about PuLP and other mathematical programming solvers. Listing 2.6 shows the steps of solving the animal feed mix problem using PuLP.

Listing 2.6 Solving Linear Programming Problem using PuLP

```
from pulp import *

model = LpProblem("Animal_Feed_Mix_Problem", LpMinimize) #A

x1 = LpVariable('Corn', lowBound = 0, upBound = 1, cat='Continous') #B
x2 = LpVariable('Limestone', lowBound = 0, upBound = 1, cat='Continous') #B
x3 = LpVariable('Soybean meal', lowBound = 0, upBound = 1, cat='Continous') #B

model += 30.5*x1 + 10.0*x2 + 90*x3, 'Cost' #C

model +=0.008 <= 0.001*x1 + 0.38*x2 + 0.002*x3 <= 0.012, 'Calcium limits' #D
model += 0.09*x1 + 0.5*x3 >=0.22, 'Minimum protein' #D
model += 0.02*x1 + 0.08*x3 <=0.05, 'Maximum fiber' #D
model += x1+x2+x3 == 1, 'Conservation' #D

model.solve() #E

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

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

#A Create a linear programming model
#B Define 3 variables that represents the percentage of each ingredients Corn, Limestone and Soybean meal in the mixture.
#C Define total cost as the objective function to be minimize
#D Add the constraints
#E Solve the problem using PuLP's choice of Solver
#F Print the results (the optimal percentages of the ingredients and the cost of the mixture per kg)

As you can see in this listing, we start by importing PuLP and creating a model as linear programming problem. We then define LP Variables with the specified associated parameters such as name, lower bound and upper bound on each variable’s range and type of the variable (e.g., Integer, Binary or Continuous). A solver is then used to solve the problem. PuLP supports several solvers such as GLPK, GUROBI, CPLEX and MOSEK. The default solver in PuLP is Cbc (Coin-or branch and cut). Running this code gives the following output:

```
Corn = 65.0%
Limestone = 3.0%
Soybean_meal = 32.0%
Total cost of the mixture per kg = 49.16$
```

If one of the objective functions or at least one of the constraints is non-linear, the problem is considered a non-linear optimization problem or nonlinear programming problem (NLP) and it's harder to solve compared to a linear problem. A special case of NLP is called Quadratic Programming (QP) when the objective function is quadratic. For example, the Plant Layout Problem (PLP) or Facility Location Problem (FLP) is a quadratic assignment problem (QAP) that aims at assigning different facilities (departments) *F* to different locations *L* in order to minimize a given function cost such as the total material handling cost as shown in Figure 2.11.

Assume that ω_{ij} is the frequency of interaction or the flow of products between these facilities and $d_{f(i)f(j)}$ is the distance between facilities *i* and *j* and. The material handling cost is:

$$MHC_{ij}=\text{flow}\times\text{distance}=\omega_{ij}\times d_{f(i)f(j)}$$

Equation 2.3

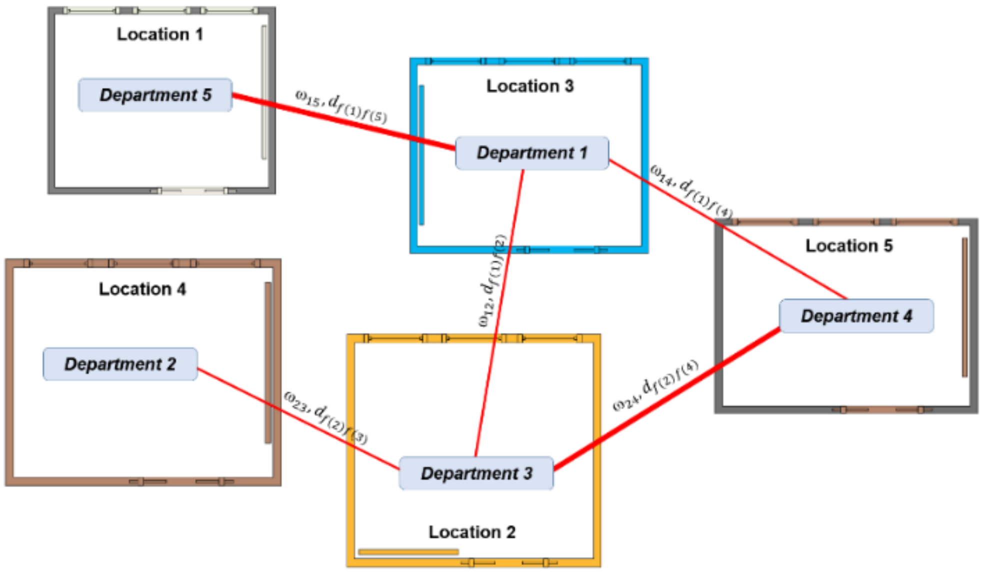

Location 1
Department 5
$\omega_{15}, d_{f(1)f(5)}$
Location 3
Department 1
$\omega_{14}, d_{f(1)f(4)}$
Location 5
Department 4
$\omega_{12}, d_{f(1)f(2)}$
Location 4
Department 2
$\omega_{23}, d_{f(2)f(3)}$
$\omega_{24}, d_{f(2)f(4)}$
Department 3
Location 2

Figure 2.11 Plant Layout Problem. What is the optimal location for each department that minimizes the overall material handling costs?

and total material handling cost (TMHC) is the summation of all the material handling costs inside the material handling cost matrix. In matrix notation, the problem can be formulated as:

Find X which minimizes $trace(WXDX^T)$

Where X represents the assignment vector, W is the flow matrix and D is the distance matrix. Trace is the sum of elements on the main diagonal (from the upper left to the lower right) of the resultant material handling cost matrix.

In a more general case, NLP includes non-linear objective functions and/or at least nonlinear constraints of any form. For example, imagine you're designing a landmine detection and disposal Unmanned Ground Vehicle (UGV) [5]. In outdoor applications like humanitarian demining, UGVs should be able to navigate through rough terrain. Sandy soils or rocky terrain with obstacles, some steep inclines, ditches, and culverts can be difficult to negotiate by the vehicle. This urges the need to carefully design the locomotion system of the vehicles to guarantee motion fluidity. Assume that you are in charge of finding optimal values for the wheel parameters (e.g., wheel diameter, breadth width and wheel loading) that will lead to:

- minimize the wheel sinkage, which is the maximum sinkage of the wheel in the soil that it is moving on;
- minimize motion resistance, which is the overall resistance faced by the UGV unit due to the different components of resistance (compaction, gravitational, etc.);
- minimize drive torque, which is the required driving torque from the actuating motors per each wheel;
- minimize drive power, which is the required driving power from the actuating motors per each wheel and
- maximize the slope negotiability that represents the maximum slope that can be climbed by the UGV unit considering its weight, as well as the soil parameters.

Due to availability in the markets and/or manufacturability concerns and costs, the wheel diameter should be in the range of 4-8.2 inches, breadth width in the range of 3-5 inches, and wheel loading in the range of 22-24 pounds per wheel. This wheel design problem (Figure 2.12) can be stated as following:

Find X which optimizes f, subject to a possible set of boundary constraints. Where X is a vector that is composed of a number of design/decision variables such as

x1=wheel diameter, x1 ∈[4.8,2]

x2=breadth width, x2∈ [3,5]

x3=wheel loading, x3∈ [22,24]

We can also consider the objective functions $f=\{f_1, f_2,\ldots\}$. For example, the function for wheel sinkage might look like this:

$$f_1 = \left(\frac{3x_3}{(3-n)(k_c+x_2 k_\varphi \sqrt{x_1})}\right)^{\frac{2}{(2n+1)}}$$

Equation 2.4

where n is the exponent of sinkage, k_c is the cohesive modulus of soil deformation, and k_φ is frictional modulus of soil deformation. This problem is considered to be non-linear since the objective function is non-linear.

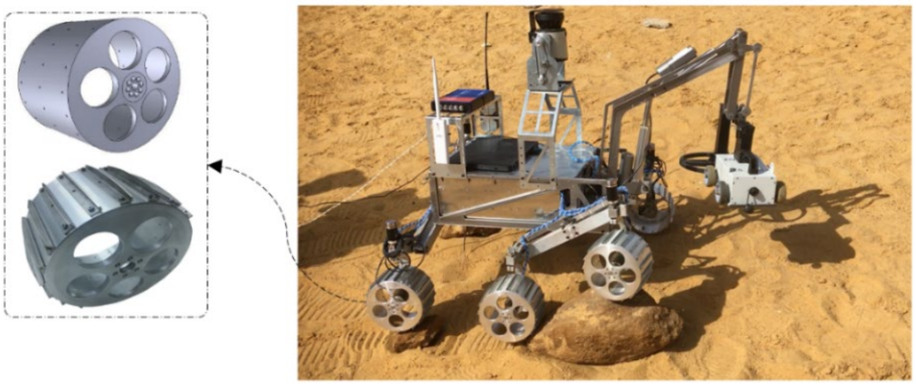


Figure 2.12 MineProbe wheel design problem [5].

The catenary problem [6] is another example of a non-linear optimization problem. A catenary is a flexible hanging object composed of multiple parts, such as a chain or telephone cable (Figure 2.13). In this problem, we are provided with n+1 homogenous beams with lengths d_1, d_2 ,…, $d_{n+1}>0$, and masses $m_1,m_2,\ldots,m_{n+1}>0$, which are connected by joints $G_1,G_2,\ldots,G_{n+1}$.The location of each joint is represented by the Cartesian coordinates (x_i,y_i,z_i). The ends of the catenary are G_1 and G_{n+1}, which have the same y and z values (they are at the same height and line with each other).

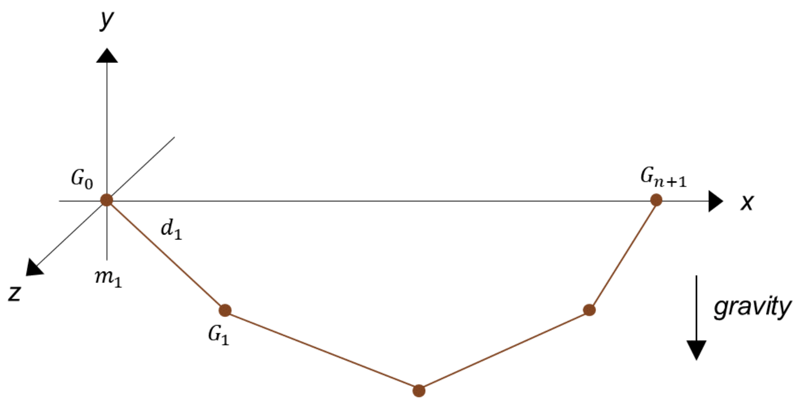


Figure 2.13 Finite Catenary Problem. The catenary (or chain) is suspended from two points G_1 and G_{n+1} [6].

Assuming that beam lengths and masses are predefined parameters, the goal is to look for stable equilibrium positions in the field of gravity, i.e., those where the potential energy is minimized. The problem can be stated as follows:

$$\text{Find } Y \text{ that minimizes } \sum_{i=1}^{n+1} m_i \gamma \frac{y_i + y_{i-1}}{2}, \quad \gamma > 0$$

Equation 2.5

subject to the following constraints:

$$g_i=(x_i-x_{i-1})^2+(y_i-y_{i-1})^2+(z_i-z_{i-1})^2-d_i^2=0,\ i=1,2,\ldots,n+1$$

Equation 2.6

Where γ is the gravitational constant. The non-linearity of the constraints makes this problem non-linear despite having a linear objective function.

2.1.5 Expected Quality and Permissible Time of the Solution

Optimization problems can also be categorized according to the expected quality of the solutions and the time given to find the solutions. Figure 2.14 shows three main types of problems, namely, design problems/strategic functions, planning problems/tactical functions and control problems/operational functions.

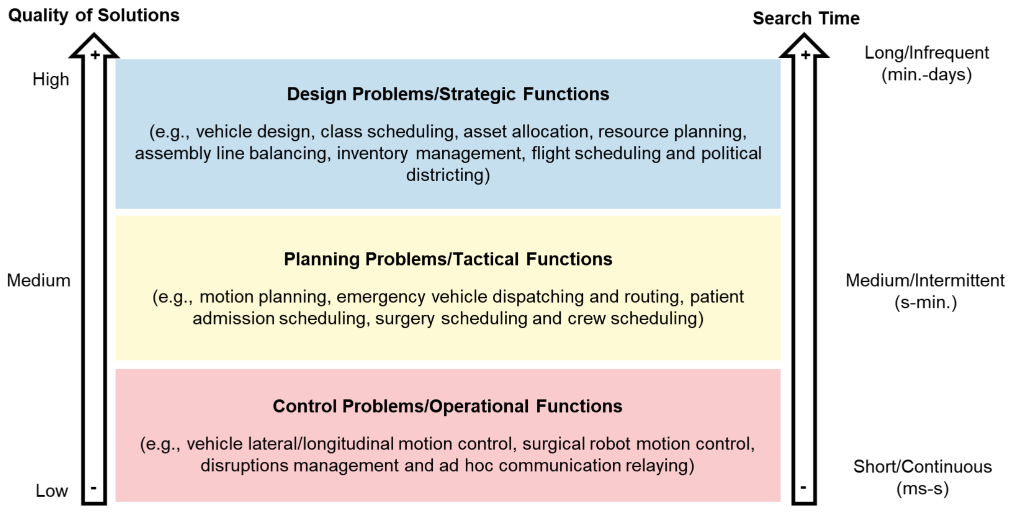

Quality of Solutions
Search Time
High
Medium
Low
+
-
Long/Infrequent (min.-days)
Medium/Intermittent (s-min.)
Short/Continuous (ms-s)
Design Problems/Strategic Functions
(e.g., vehicle design, class scheduling, asset allocation, resource planning, assembly line balancing, inventory management, flight scheduling and political districting)
Planning Problems/Tactical Functions
(e.g., motion planning, emergency vehicle dispatching and routing, patient admission scheduling, surgery scheduling and crew scheduling)
Control Problems/Operational Functions
(e.g., vehicle lateral/longitudinal motion control, surgical robot motion control, disruptions management and ad hoc communication relaying)

Figure 2.14 Qualities of solutions vs. Search Time. Some types of problems require fast computations but do not require incredibly accurate results, while others (such as design problems) allow more processing time in return for higher accuracy.

In design problems, time is not as important as the quality of the solution and users are willing to wait (sometimes even a few days) to get an optimal, or near-optimal result. These problems can be solved offline, and the optimization process is usually carried out only once in a long time. Examples of design problems include political districting, vehicle design, class scheduling, asset allocation, resource planning, assembly line balancing, inventory management, flight scheduling and political districting.

Let’s discuss political districting as a design problem in more details. Districting is the problem of grouping small geographic areas, called basic units, into larger geographic clusters, called districts, in a way that the latter are acceptable according to relevant planning criteria [7]. Typical examples for basic units are customers, streets, or zip code areas. Examples of these criteria may include:

- Balance or equity in terms of demographic background, equitable size, balanced workload, equal sales potentials, or number of customers, for example.
- Contiguity to enable travelling between the basic units of the district without having to leave the district.
- Compactness to enable having round-shaped or square shaped undistorted districts without holes.
- Respect of boundaries such as administrative boundaries, railroads, rivers, mountains, etc.
- Socio-economic heterogeneity to have a better representation of residents with different income revenues, ethnicity, concerns, or views.

Political districting, school districting, districting for health services, districting for EV charging stations, districting for micro mobility (e.g., e-bikes and e-scooters) stations and districting for sales or delivery are examples of districting problems. Political districting is an issue that has plagued societies since the advent of representative democracy in the Roman Republic. In a representative democracy, officials are nominated and elected to power to represent the interests of the people who elected them. In order to have a greater say when deciding on matters that concern the entire state, the party system came about, which defines a political platform on which each nominee will use to differentiate themselves from competitors. Manipulating the shape of electoral districts to determine the outcome of an election is called gerrymandering (named after early 19th Massachusetts governor Elbridge Gerry who redrew the map of the Senate’s districts in 1810 in order to weaken the opposing federalist party). Figure 2.15 shows how manipulating the shape of the districts can sway the vote in favor of a decision that otherwise wouldn’t have won.

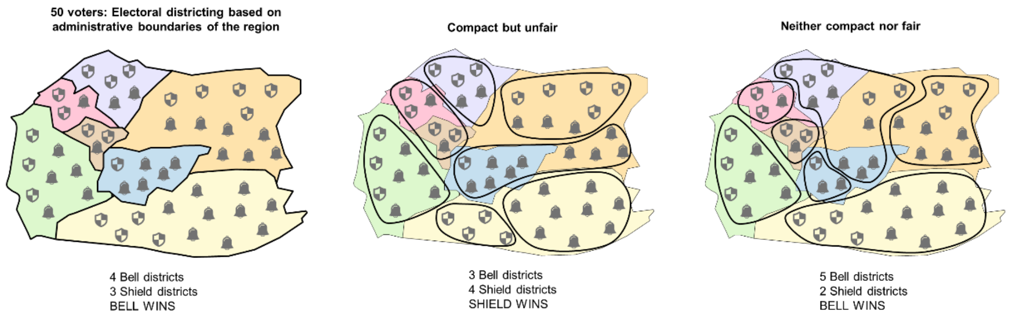


Figure 2.15 Example of Gerrymandering. The two major political parties, Shield and Bell try to gain an advantage by manipulating the district boundaries to suppress undesired interests, and to promote their own.

To avoid gerrymandering, an effective and transparent political districting strategy is needed to generate a solution that preserves the integrity of individual sub-districts and divides the population into almost equal voting populations in a reproducible way. In many countries, electoral districts are reviewed from time to time to reflect changes and movements in the country's population. For example, the Constitution of Canada requires that federal electoral districts be reviewed after each decennial (10-year) census.

Political districting is defined as aggregating *n* sub-regions of a territory into *m* electoral districts subject to the following constraints:

- The districts should have near-equal voting population,
- The socio-economic homogeneity inside each district as well as the integrity of different communities should be maximized,
- The districts have to be compact, and the sub-regions of each district have to be contiguous, and
- Sub-regions should be considered as indivisible political units and their boundaries should be respected.

The problem can be formulated as an optimization problem in which a function that quantifies the above factors is maximized. An example of this function is:

$$F(x)= \alpha_{pop} f_{pop}(x)+ \alpha_{comp} f_{comp}(x)+ \alpha_{soc} f_{soc}(x)+ \alpha_{int} f_{int}(x)+ \alpha_{sim} f_{sim}(x)$$

Equation 2.7

Where *x* is a solution to the problem or the electoral districts, α_i are user-specified multipliers $0 \le \alpha_i \le 1$, f_{pop}, f_{comp}, f_{soc}, f_{int} and f_{sim} are functions that quantify the population equality, the compactness of districts, the socio-economic homogeneity, the integrity of different communities, and the similarity to existing districts, respectively. In the upcoming chapters, we will show how to handle optimal multi-criteria assignment design problems using offline optimization algorithms.

Planning problems need to be solved relatively faster compared to design problems, in a time span from a few seconds to a few minutes. To find a solution during such a short period of time, optimality is usually traded for speed gains. Examples of planning problems include vehicle motion planning, emergency vehicle dispatching and routing, patient admission scheduling, surgery scheduling and crew scheduling. Let's consider ridesharing problem as an example of planning problem. Ridesharing involves a fleet of pay-per-use vehicles and a set of passengers with predefined pick up and drop off points (Figure 2.16). For such a set of passengers and drivers, the dispatch service of the application will have to assign a set of passengers in specific orders to each driver to maximize the total revenue earned by the drivers and minimize the number of trips missed. 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/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 driver's earning
- Minimizing passenger's waiting time
- Minimizing the total number of drivers required

For the ride-sharing problem, both the search time and the quality of the solutions are important. On many popular ridesharing platforms, dozens if not hundreds of users may be simultaneously searching for rides at the same place in a given district. Overly costly and time-consuming solutions will lead to higher operating costs (i.e., employing more drivers than necessary, calling in drivers from other districts) as well as the potential for lost business (bad user experiences may dissuade passengers from using the platform a second time), and high driver turnover rate.

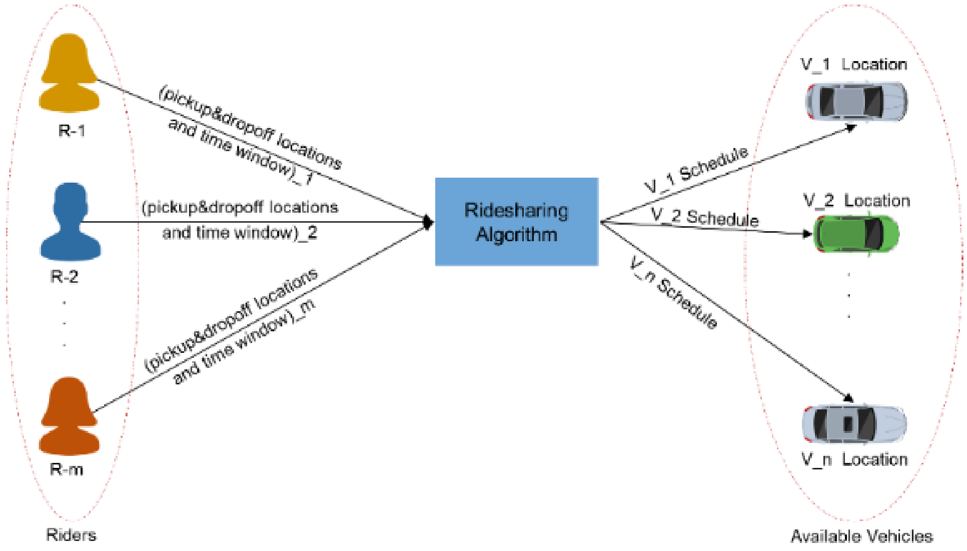

R-1
R-2
R-m
Riders
(pickup&dropoff locations and time window)_1
(pickup&dropoff locations and time window)_2
(pickup&dropoff locations and time window)_m
Ridesharing Algorithm
V_1 Schedule
V_2 Schedule
V_n Schedule
V_1 Location
V_2 Location
V_n Location
Available Vehicles

Figure 2.16 Ridesharing Problem. This planning problem needs to be solved in a shorter amount of time, as delays could mean lost trips and a bad user experience.

In practice, the assignment of drivers to passengers goes well beyond distance between passenger and driver, but may also include factors such as driver reliability, passenger rating, vehicle type, as well as pickup location and destination location types. For example, a customer going to the airport may request a larger vehicle to accommodate luggage. In the upcoming chapters, we will discuss how to solve planning problems using different search and optimization algorithms.

Control problems require very fast solutions in real-time. In most of the cases, this means in a time span between a few milliseconds or less, to a few seconds. Vehicle lateral/longitudinal motion control, surgical robot motion control, disruptions management and ad hoc communication relaying are examples of control problems. Online optimization algorithms are required to handle such kinds of problems. Optimization tasks in both planning and control problems are often carried out repetitively – new orders will, for instance, continuously arrive in a production facility and need to be scheduled to machines in a way that minimizes the waiting time of all jobs. Communication replaying can be considered as a control problem. Imagine a real-world situation where a swarm of Unmanned Aerial Vehicles (UAVs) or Micro Aerial Vehicles (MAVs) is deployed to search victims trapped on untraversable terrain after a natural disaster like an earthquake, avalanche, tsunami, tornado, wildfire, etc. The mission consists of two phases: a search phase and a relay phase. During the search phase, MAVs will conduct search according to the deployment algorithm. When a target is found, the swarm of MAVs self-organizes to utilize their range-limited communication capabilities for setting up an ad hoc communication relay network between the victim and the base station as illustrated in Figure 2.17.

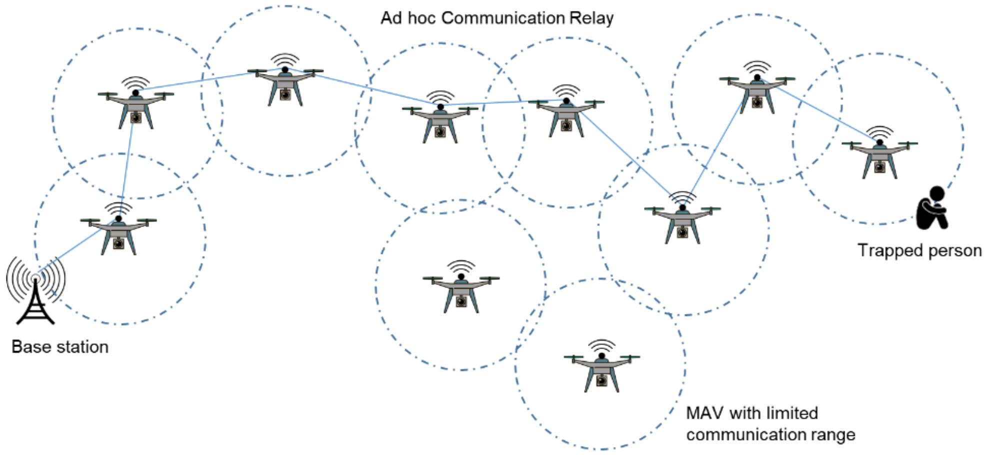

Ad hoc Communication Relay
Trapped person
Base station
MAV with limited communication range

Figure 2.17 Communications Relaying Problem. A swarm of MAVs must form an ad hoc communication replay between a base station and a trapped victim. The movement of MAVs is a control problem that must be solved repeatedly, multiple times per second. In this case, speed is more important than accuracy, as minor errors can be immediately corrected during the next cycle.

During the search space, MAVs can be deployed in such a way that maximizes the area coverage. After detecting a victim, MAVs can be repositioned to maximize the victim's visibility. The ad hoc communication relay network is then established based to maximize the radio coverage in the swarm and to find the shortest path between the MAV that detected the victim and the base station considering the following assumptions:

- MAVs are capable of situational awareness by combining data from three noise-prone sensors: magnetic compass for direction, speedometer for speed, and altimeter for altitude.
- MAVs are capable of communicating via a standard protocol such as IEEE 802.11b with a limited range of 100m communication radius.
- MAVs are capable of relaying ground signals as well as control signals sent amongst MAVs.
- MAVs have enough onboard power to sustain 30 minutes of continuous flight, at which point it must return to the base to recharge. However, the sustainment of flight is variable with respect to the amount of signaling completed during flight.
- MAVs are capable of quickly accelerating to a constant flight speed of 10m/s.
- MAVs are not capable of hovering and have a minimum turn radius of approximately 10m.

For control problems such as MAV repositioning, search time is of paramount importance. As the MAVs cannot hover and thus must remain in constant motion, delayed decisions may lead to unexpected situations, such as mid-air collisions or loss of signal. As instructions are sent (or repeated) every few milliseconds, the MAV must be able to decide its next move within that span of time. MAVs must account only for its current position, target position, and velocity, but also consider obstacles, communications signal strength, as well as wind and other environmental effects. Minor errors are acceptable, as they can be corrected in subsequent searches. In the upcoming chapters, we will discuss how to solve control problems.

This book will largely focus on complex, ill-structured problems that cannot be handled by traditional mathematical optimization or derivative-based solvers, and will give examples of different design, planning and control problems in various domains. Next, we'll take a look at how search and optimization algorithms are classified.

2.2 Search and Optimization Algorithm Classification

When we search, we try to examine different states to find a path from the start/initial state to the goal state. Very often, an optimization algorithm searches for an optimum solution by iteratively transforming a current state or a candidate solution into a new, hopefully better, solution. Search algorithms can be classified based on the way the search space is explored.

- **Local Search** uses only local information of the search space surrounding the current solution to produce new solutions. Since only local information is used, local search algorithms (also known as local optimizers) locate local optima (which may or may not be a global optima).
- **Global Search** uses more information about the search space to locate global optima. In other words, global search algorithms explore the entire search space, while local search algorithms only exploit neighborhoods.

Yet another classification includes deterministic and stochastic algorithms as illustrated in Figure 2.18.

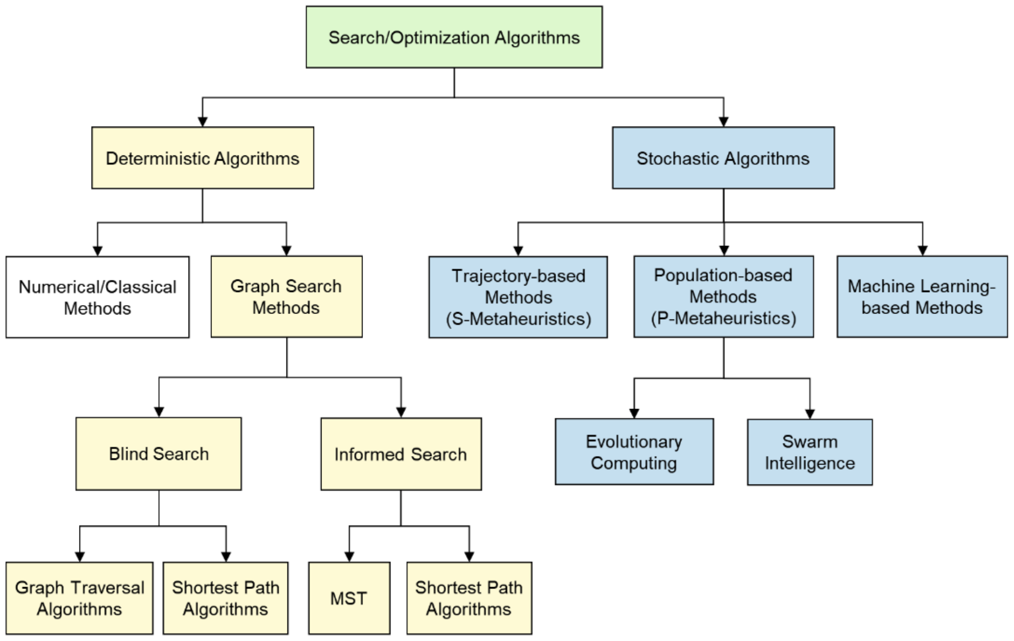

Search/Optimization Algorithms
Deterministic Algorithms
Stochastic Algorithms
Numerical/Classical Methods
Graph Search Methods
Trajectory-based Methods (S-Metaheuristics)
Population-based Methods (P-Metaheuristics)
Machine Learning-based Methods
Blind Search
Informed Search
Evolutionary Computing
Swarm Intelligence
Graph Traversal Algorithms
Shortest Path Algorithms
MST
Shortest Path Algorithms

Figure 2.18 Deterministic vs. stochastic Algorithms. Deterministic algorithms follow a set procedure, and the results are repeatable, while stochastic searches have elements of randomness built into the algorithms.

- **Deterministic Algorithms** follow a rigorous procedure in its path, and both the values of its design variables and its functions are repeatable. For the same starting point, they will follow the same path whether you run the program today or tomorrow. Examples include, but are not limited to graphical methods, gradient and Hessian based methods, penalty methods, gradient projection methods, and graph search methods. Graph search methods can be further subdivided into blind search methods (depth-first, breadth-first, Dijkstra) and informed search methods (hill climbing, beam search, Best-first, A*, contraction hierarchies). Deterministic methods are covered in Part-I of this book.
- **Stochastic Algorithms** explicitly use randomness in their parameters and/or decision-making process. For example, genetic algorithms use some pseudo-random numbers, resulting in individual paths that are not exactly repeatable. With stochastic algorithms, the time taken to obtain an optimal solution cannot be accurately foretold. Solutions do not always get better, and stochastic algorithms sometimes miss the opportunity to find optimal solutions. This behavior can be advantageous, however, to avoid becoming trapped in local optima. Examples of stochastic algorithms include tabu search, simulated annealing, genetic algorithms, differential evolution algorithms, particle swarm optimization, ant colony optimization, artificial bee colony, firefly algorithm, etc. Most statistical machine learning algorithms are stochastic because they make use of randomness during learning stage and they make

predictions during inference stage with a certain level of uncertainty. Moreover, some machine learning models are like people unpredictable. Models trained using human behavior-based data as independent variables are more likely unpredictable compared to those trained using independent variables that strictly follow physics laws. For example, human intent recognition model is less predictable compared to a model that predict the stress-stain curve of a material. So, due to the uncertainty associated with machine learning predictions, machine learning-based algorithms used in solving optimization problems can be considered as stochastic methods. Stochastic algorithms are covered in parts II, III, IV and V of this book.

Treasure Hunting Mission

The search for an optimal solution in a given search space can be likened to a treasure hunting mission. Imagine you and a group of friends decided to visit an island looking for a pirates' treasure.

All the areas on the island (except the active volcano area) correspond to the feasible search space of the optimization problem. The treasure corresponds to the optimal solution to be found in this feasible space. You and your friends are the "search agents" launched to search for the solution, each following different search approaches. If you don't have any information that can guide you while searching, you are following a blind/uninformed search approach, which is usually inefficient and time consuming. If you know that the pirates used to hide the treasure in elevated spots, you can then directly climb up to the steepest cliff and try to reach the highest peak. This scenario corresponds to the classical hill-climbing technique (informed search). You can also follow a trial-and-error approach looking for some hints and keep moving from one place to another plausible place until you find the treasure (trajectory-based search).

If you do not want to take the risk of getting nothing and decide to collaborate and share information with your friends instead of doing the treasure-hunting alone, you will be following a population-based search approach. While working in a team, you may notice that some treasure hunters show better performance than others. In this case, only good performing hunters can be kept, and new ones can be recruited to replace the less performing hunters. This is akin to evolutionary algorithms such as genetic algorithms where the fittest hunters survive. Alternatively, you and other friends can try to emulate the success of the outperforming hunters in each area of the treasure island without getting rid of any team members and without recruiting new ones. This scenario uses the so-called swarm intelligence and corresponds to the population-based optimization algorithms such as ant colony optimization and particle swarm optimization.

You alone or with the help of your friends can build a mental model based on historical data of previous and similar treasure hunting missions or train a reward predictor based on try-&-error interaction with the treasure island (search space) taking the strength of the signal from the metal detector as a reward indicator. After a few iterations, you will learn to maximize the reward from the predictor and improve your behavior until you fulfill the desired goal and find the treasure. This corresponds to a machine learning-based approach to be explained in Part-4 of this book.

2.3 Heuristics and Meta-heuristics

The word heuristic comes from the Greek word *heuriskein*, which means "to find or discover". The past tense of this verb, *eureka,* was used by the Greek mathematician, physicist, engineer, astronomer, and inventor Archimedes who was so excited with the discovery of the Buoyancy Law that he jumped out of his bath shouting loudly "Eureka! Eureka!" (I have found it! I have found it!). Heuristics (also known as "mental shortcuts" or "rules of thumb") are solution strategies, seeking methods, or rules that can facilitate finding acceptable (optimal or near-optimal) solutions to a complex problem in a reasonably practical time. Despite the fact that heuristics can seek near-optimal solutions at a reasonable computational cost, they cannot guarantee either feasibility or degree of optimality.

The term "metaheuristic" comes from the composition of two Greek words: *meta* which means "beyond, on a higher level", and heuristics. It's a term coined by Fred Glover, inventor of Tabu Search (see Chapter 6) to refer to high-level strategies used to guide and modify other heuristics to enhance their performance. Their goal is to efficiently explore the search space in order to find optimal or near-optimal solutions. Metaheuristics may incorporate mechanisms to achieve tradeoff between exploration/diversification and exploitation/intensification of the search space to avoid getting trapped in confined areas of the search space, at the same time finding the optimal or near-optimal solutions in reasonable time. Finding this proper balance of diversification and intensification is a crucial issue in heuristics as discussed in section 1.5. Metaheuristic algorithms are often global optimizers that can be applied to different linear and non-linear optimization problems with relatively few modifications to make them adapted to a specific problem. These algorithms are often robust to problem size, problem instance and random variables.

Let's assume that we have 6 objects with different sizes (2,4,3,6,5 and 1) and we need to pack them into a minimum number of bins. Each bin has a limited size of 7 so the total size of the object in the bin should be 7 or less. If we have n objects, there are *n!* possible ways of packing the objects. The minimum number of bins we need is called lower bound. To calculate this lower bound, you need to find the total number of object sizes (2+4+3+6+5+1=21). Then the lower bound is 21/7=3 bins. This means that we need at least 3 bins to pack these objects. Figure 2.19 illustrates two heuristics that can be used to solve this bin packing problem.

First-fit heuristics packs the objects following their order without taking into consideration the size. This results in the need for 4 bins that are not fully utilized as there are 7 spaces left in 3 of these bins. If we apply first-fit decreasing heuristic, we will order the objects based on their sizes and pack following this order. This heuristic allows us to pack all the objects in only 3 fully utilized bins, which is the lower bound.

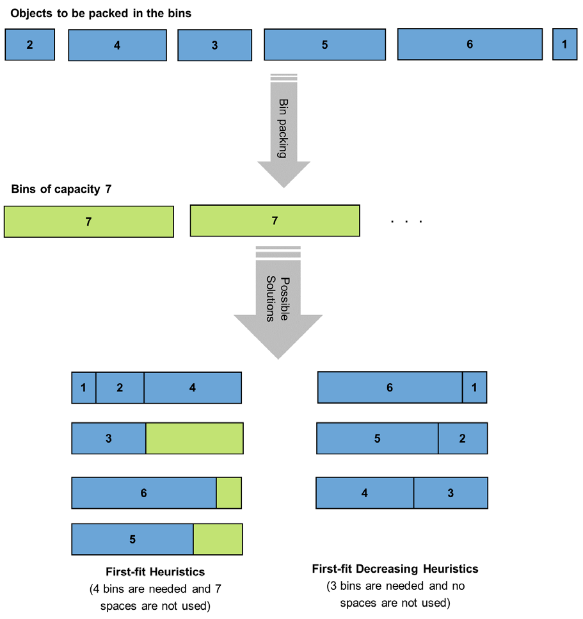

Objects to be packed in the bins
2
4
3
5
6
1
Bin packing
Bins of capacity 7
7
7
. . .
Possible Solutions
1
2
4
3
6
5
6
1
5
2
4
3
First-fit Heuristics
(4 bins are needed and 7 spaces are not used)
First-fit Decreasing Heuristics
(3 bins are needed and no spaces are not used)

Figure 2.19 Handling bin packing problem using first-first and first-fit decreasing heuristics.

In the previous example, all the objects have the same height. However, in more generalized version, let's consider objects with different width and height as illustrated in Figure 2.20.

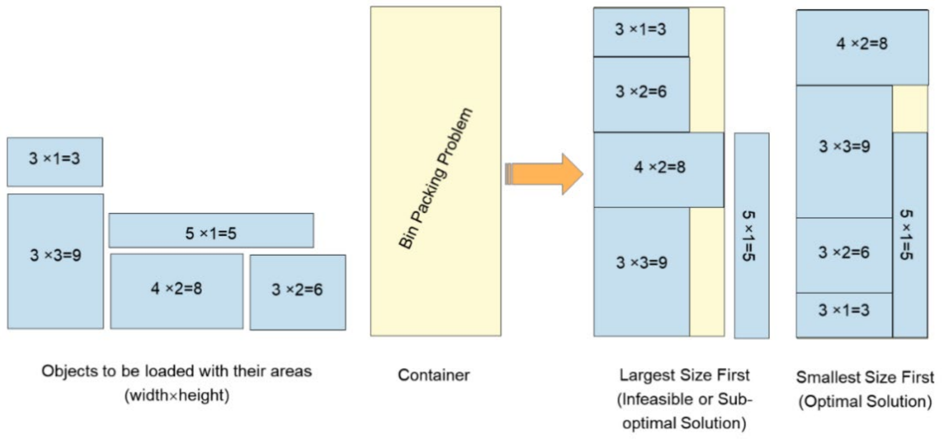

3 ×1=3
3 ×3=9
5 ×1=5
4 ×2=8
3 ×2=6
Objects to be loaded with their areas (width×height)
Bin Packing Problem
Container
3 ×1=3
3 ×2=6
4 ×2=8
3 ×3=9
5 ×1=5
Largest Size First (Infeasible or Sub-optimal Solution)
4 ×2=8
3 ×3=9
3 ×2=6
3 ×1=3
5 ×1=5
Smallest Size First (Optimal Solution)

Figure 2.20 Bin packing problem. Using heuristics allows us to solve the problem much faster than with a brute-force approach. However, some heuristic functions may result in infeasible or suboptimal solutions and they do not guarantee optimality.

Applying heuristics such as smallest-first can allow us to load the container much faster, as illustrated in Figure 2.20. Some heuristics do not guarantee optimality (i.e., heuristics like largest-first gives a sub-optimal solution, as one object is left out). This can be considered as infeasible solution if we need to load all the objects into the container or sub-optimal solution if the objective is to load as many objects as we can.

To solve this problem in Python, let's first define the objects, the containers, and what it means to place an object inside of a container. For the sake of simplicity, Listing 2.7 avoids custom classes and uses `numpy` arrays instead.

Listing 2.7 Bin Packing Problem

```
import numpy
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.colors import rgb2hex

width = 4    #A
height = 8    #A
container = numpy.full((height,width), 0)    #A

objects = [[3,1],[3,3],[5,1],[4,2],[3,2]]    #B

def fit(container, object, obj_index, rotate=True):    #C
    obj_w = object[0]    #C
    obj_h = object[1]    #C
    for i in range(height - obj_h + 1):    #C
        for j in range(width - obj_w + 1):    #C
            placement = container[i : i + obj_h, j : j + obj_w]    #C
            if placement.sum() == 0:    #C
                container[i : i + obj_h, j : j + obj_w] = obj_index    #C
                return True    #C
        return fit(container, object[::-1], obj_index, rotate=False)    #C
```

#A Define the dimensions of the container and initialize the numpy array to 0s.
#B Represent objects to be placed as [width, height].
#C The fit function places objects into the container, either through direct placement, shifting, or rotation.

The *fit* function attempts to write a value to a 2D slice of the container, provided that there are no values in that slice already (the sum is 0). If that fails, it shifts along the container from top to bottom, left to right and tries again. As a last resort, it tries the same but with the object rotated by 90 degrees.

The first heuristic prioritizes fitting by object area, descending:

```
def largest_first(container, objects):
    excluded = []
    assigned = []
    objects.sort(key=lambda obj: obj[0] * obj[1], reverse=True)    #A
    for obj in objects:
        if not fit(container, obj, objects.index(obj) + 1):
            excluded.append(objects.index(obj) + 1)    #B
        else:
            assigned.append(objects.index(obj) + 1)
    if excluded: print(f"Items excluded: {len(excluded)}")
    visualize(numpy.flip(container, axis=0), assigned)    #C
```

#A Sort elements by area, descending
#B Some objects may not fit; we can keep track of them using a list
#C Visualize the filled container

The code for visualizing this result can be found in the full code files for Listing 2.7 available on the GitHub repo of the book.

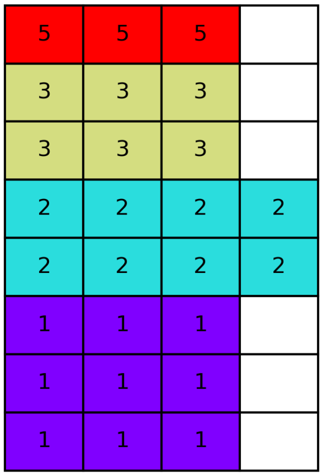


Figure 2.21 Bin Packing using the Largest-first heuristic. One object has been excluded, as it does not fit into the remaining space.

The second heuristic sorts first by width, and then by total area, ascending:

```
def smallest_width_first(container, objects):
    excluded = []
    assigned = []
    objects.sort(key=lambda obj: (obj[0], obj[0] * obj[1]))    #A
    for obj in objects:
        if not fit(container, obj, objects.index(obj) + 1):
            excluded.append(objects.index(obj) + 1)
        else:
            assigned.append(objects.index(obj) + 1)
    if excluded: print(f"Items excluded: {len(excluded)}")
    visualize(numpy.flip(container, axis=0), assigned)    #B
```

#A Sort by width as primary key, and then by area, ascending
#B Visualize the solution

The *smallest_width_first* heuristic manages to successfully fit all the objects into the container, as shown in Figure 2.22.

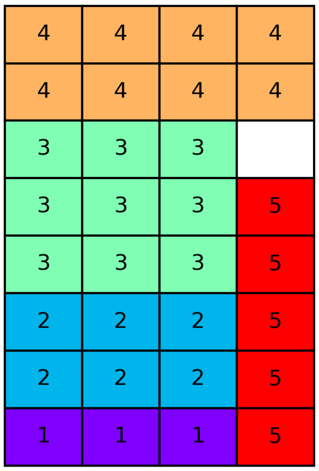


Figure 2.22 Bin Packing Problem using Smallest-first. All five objects have been successfully placed into the container.

Different heuristic search strategies can be used to generate candidate solutions. These strategies include, but are not limited to, search by repeated solution construction (e.g., graph search, ant colony optimization), search by repeated solution modification (e.g., tabu search, simulated annealing, genetic algorithm, and particle swarm optimization) and search by repeated solution recombination (e.g., genetic algorithm and differential evolution).

Considering the cargo bike load problem discussed in subsection 1.3.3 of the previous chapter, we can order the items to be delivered based on their efficiency (profit per kg) as shown in Table 2.2.

Table 2.2 Packages ranked by efficiency. The efficiency of a package is defined as the profit per kilogram.

Item	Weight (kg)	Profit ($)	Efficiency ($/kg)
10	7.8	20.9	2.68
7	4.9	10.3	2.10
4	10	12.12	1.21
1	14.6	14.54	1
8	16.5	13.5	0.82
6	9.6	7.4	0.77
2	20	15.26	0.76
9	8.77	6.6	0.75
3	8.5	5.8	0.68
5	13	8.2	0.63

Following a repeated solution construction-based heuristic search strategy, we can start by applying a greedy principle and pick items based on its efficiency until we reach the maximum payload of the cargo bike (100 kg) as a hard constraint. The steps for this are shown in Table 2.3.

Table 2.3 Repeated Solution Construction. Packages are added to the bike until the maximum capacity is reached.

Step	Item	Add?	Total Weight (kg)	Total Profit ($)
1	10	Yes	7.8	20.9
2	7	Yes	12.7	31.2
3	4	Yes	22.7	43.32
4	1	Yes	37.3	57.86
5	8	Yes	53.8	71.36
6	6	Yes	63.4	78.76
7	2	Yes	83.4	94.02
8	9	Yes	92.17	100.62
9	3	No	(100.67)	-
10	5	No	(113.67)	-

We finally obtain the following subset of items: 10, 7, 4, 1, 8, 6, 2 and 9. This can also be written as: (1,1,0,1,0,1,1,1,1,1), which when read from left to right shows that we include items 1, 2, 4, 6, 7, 8, 9 and 10 (and exclude items 3 and 5). This results in a total profit of $100.62 and weight of 92.17kg. We can generate more solutions by repeating the process of adding objects starting from an empty container.

In the case of repeated solution modification-based heuristic search strategy, instead of creating one or more solutions completely from scratch, one could also think about ways of modifying an already available feasible solution. Consider the previous solution generated for the cargo-bike problem: (1,1,0,1,0,1,1,1,1,1). We know that this feasible solution is not optimal, but how can we improve it? This can be achieved for instance by removing item 9 from the cargo bike and adding item 5. This process of removing and adding results in a new solution: (1,1,0,1,1,1,1,1,0,1) with a total profit of $102.22 and a weight of 96.4 kg.

Following search by repeated solution recombination, existing solutions are combined as a way to generate new solutions to progress in the search space. Suppose the following two solutions are given:

S_1= (1,1,1,1,1,0,0,1,0,1) with weight 75.8 kg. and a profit of $75.78 and

S_2= (0,1,0,1,1,0,1,1,1,1) with weight 80.97 kg and a profit of $86.88.

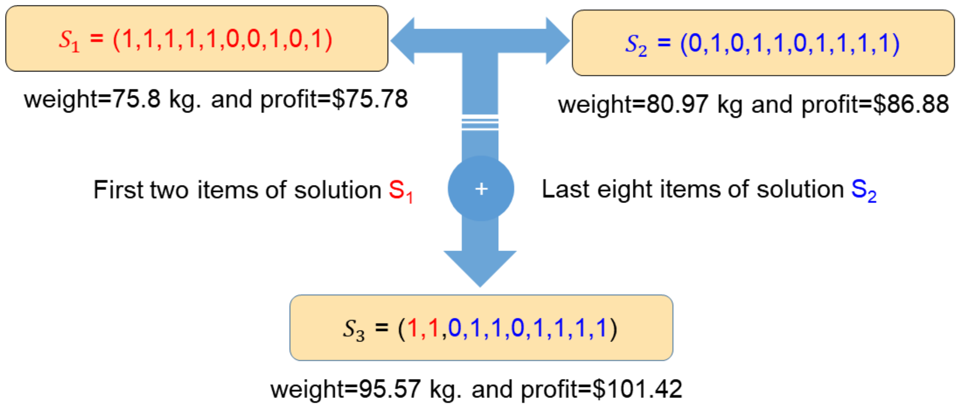

S_1 = (1,1,1,1,1,0,0,1,0,1)
weight=75.8 kg. and profit=$75.78
S_2 = (0,1,0,1,1,0,1,1,1,1)
weight=80.97 kg and profit=$86.88
First two items of solution S_1
+
Last eight items of solution S_2
S_3 = (1,1,0,1,1,0,1,1,1,1)
weight=95.57 kg. and profit=$101.42

Figure 2.23 Repeated solution recombination. Taking the first two elements of S_1 and adding it to the last eight elements of S_2 yields a new, better solution.

As illustrated in Figure 2.23, to get a new solution we can take the configuration of the first two items of S_1 and the last eight items of S_2. This means that we include items 1, 2, 4, 5,7, 8, 9 and 10 in the new solution and exclude items 3 and 6. This yields a new solution: S_3= (1,1,0,1,1,0,1,1,1,1) with a weight of 95.57 kg and a higher profit of $101.42.

2.4 Nature-inspired Algorithms

Nature is the ultimate source of inspiration. Problems in nature are usually ill-structured, dynamic, partially observable, non-linear, multimodal, multi-objectives with hard and soft constraints and with no or limited access to global information. Nature-inspired algorithms are computational models that mimic or reverse engineer the intelligent behaviors observed in nature. Examples include molecular dynamics, cooperative foraging, division of labor, self-replication, immunity, biological evolution, learning, flocking, schooling, and self-organization, just to name just a few.

Molecular dynamics (the science of simulating the motions of a system of particles applies) and thermal annealing inspired scientists to create optimization algorithm called simulated annealing to be discussed in chapter 5 of this book. Evolutionary computation algorithms such as Genetic Algorithm (GA), Genetic Programming (GP), Evolutionary Programming (EP), Evolutionary Strategies (ES), Differential Evolution (DE), Cultural Algorithms (CA) and Co-evolution (CoE) are created by inspiration from evolutionary biology (the study of the evolutionary processes) and biological evolution. Part-III of the books covered a number of evolutionary computing algorithms.

Ethology (the study of animal behavior) is the main source of inspiration of swarm intelligence algorithms such as Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), Artificial bee colony (ABC), Bat algorithm (BA), Social Spider Optimization (SSO), Firefly algorithm (FA), Butterfly Optimization Algorithm (BOA), Dragonfly Algorithm (DA), Krill Herd (KH), Shuffled Frog Leaping Algorithm (SFLA), Fish School Search (FSS), Dolphin

Partner Optimization (DPO), Dolphin Swarm Optimization Algorithm (DSOA), Cat Swarm Optimization (CSO), Monkey Search Lion Optimization Algorithm (LOA), Cuckoo Search (CS), Cuckoo Optimization Algorithm (COA), Wolf Search Algorithm (WSA) and Grey Wolf Optimizer (GWO). Swarm intelligence-based search and optimization algorithms are covered in Part-IV of the book.

Neural networks are simplified brain models that can be used to solve search and optimization problem as described in Part-V of the book. Tabu Search explained in Chapter 6 is based on evolving memory (adaptive memory and responsive exploration) which is studied in behavioral psychology (the science of behavior and mind). Reinforcement learning, another branch of machine learning inspired by behaviorist psychology, is used to solve search and optimization problem as discussed in Part-V.

Other nature-inspired search and optimization algorithms include, but are not limited to, Bacterial foraging optimization algorithm (BFO), Bacterial Swarming Algorithm (BSA), Biogeography-based optimization (BBO), Invasive Weed Optimization (IWO), Flower Pollination Algorithm (FPA), Forest Optimization Algorithm (FOA), Water Flow-like Algorithm (WFA), Water Cycle Algorithm (WCA), Brainstorm Optimization Algorithm (BSO), Stochastic diffusion search (SDS), Alliance Algorithm (AA) , Black-Hole algorithm (BH), Black Hole Mechanics Optimization (BHMO), Adaptive Black Hole Algorithm, Improved Black Hole algorithm (IBH), Levy Flight Black Hole (LBH), Multiple population levy black hole (MLBH), Spiral Galaxy-Based Search Algorithm (GbSA), Galaxy-based Search Algorithm (GSA), Big-Bang Big-Crunch (BBBC), Ray Optimization (RO), Quantum Annealing (QA), Quantum-Inspired Genetic Algorithm (QGA), Quantum-Inspired Evolutionary Algorithm (QEA), Quantum Swarm Evolutionary Algorithm (QSE) and Quantum-Inspired Particle Swarm Optimization (QPSO). For a comprehensive list of metaheuristic algorithms, the reader is referred to [8]. This is really a long list of algorithms to be covered in one book. There are a lot of similarities between these algorithms though. The following algorithms and their variants will be covered in this book:

- Graph Search Methods (blind/uninformed search and informed search algorithms)
- Simulated Annealing (SA)
- Tab Search (TS)
- Genetic algorithm (GA)
- Particle Swarm Optimization (PSO)
- Ant Colony Optimization (ACO)
- Graph Neural Networks (GNN)
- Attention Mechanisms
- (Deep) Reinforcement learning

Throughout this book, several real-world problems will be discussed to show how to apply these algorithms in real life applications.

2.5 Exercises

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

1.1. __________ is the class of decision problems that can be solved by non-deterministic 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); linearity (Linear Programming (LP), Quadratic Programming (QP), Non-linear Programming (NLP)).
   a) Minimize $y+\cos(x^2)$, $\sin(x)-xy$ and $1/(x+y)^2$
   b) Maximize $2-\exp(1-x^3)$ subject to $-3\leq x<10$
   c) Maximize $3*x-y/5$ subject to $-2\leq x<3$, $0<y\leq 3$ and $x+y=4$
   d) The knapsack problem is an example of combinatorial problem whose solution takes the form of a combination where the order doesn't matter. Let's assume a number of items, each with a utility and a weight to be packed into a bag of limited capacity as illustrated in Figure 2.24.

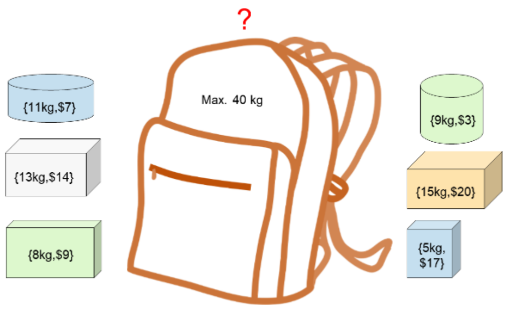


Figure 2.24. Each item has a utility and a weight, and we want to maximize the utility of the contents of the knapsack. The problem is constrained by the capacity of the bag.

   e) The school districting problem consists in determining the groups of students attending each school of a school board located over a given territory in a way that maximizes 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 as a soft constraint in this problem.

3. For the following optimization problem, state the type of the problem based on the permissible time to solve the problem and the expected quality of the solutions (design, planning or control problem). Suggest the appropriate algorithm required to handle the stated optimization problem (offline versus offline).
   a) Finding the optimal wind park design where number and types of wind turbines need to be chosen and placed considering the given wind conditions and wind park area.

b) Finding multiple vehicle routes starting and ending at different depots such that all customer demands are fulfilled.

c) Creating a fitness assistant for runners and cyclists that seamlessly automates the multiple tasks involved in planning fitness activities. The planner assesses an athlete's current fitness level and individual training goals in order to create a fitness plan. The planner also generates and recommends 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 are continuously adapting based on each user's progress within their fitness goals, thus keeping the athlete challenged and motivated.

d) Given demand and revenues for every origin-destination pair over time-of-the-day and day-of-the-week, route information distances, times, operating restrictions, aircraft characteristics and operating costs and operational and managerial constraints. We need to find a set of flights with departure and arrival times and aircraft assignment which maximize profits.

e) Finding the optimal schedule for delivery cargo-bikes, semi- and fully autonomous last-mile delivery trucks, self-driving delivery robots/droids, delivery drones, e-Palette, postal delivery, driverless deliveries, and privately owned AV to maximize customer satisfaction and minimize delivery cost taking into consideration the capacity of the vehicle, type of delivery service (couple of days delivery, next-day delivery, or same-day delivery with some extra surcharge), delivery time, drop-off locations, and so on.

f) Planning 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) Finding 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 estimated time of arrival and the consumed energy.

h) Planning a trip itinerary is perhaps the most challenging and time-consuming task when it comes to traveling. It is always ideal to optimize the visitor's time by visiting the best attractions available; however, choosing from a large pool of highly rated sites significantly makes the decision-making process more difficult. This problem can be addressed by the development of a trip planner that minimizes total commute time, maximizes the average ratings of attractions contained in a solution, and maximizes the duration spent at each of these attractions and effectively minimizes idle time when someone visits a city.

i) Finding school bus loading patterns/schedules such that the number of routes is minimized, the total distance traveled by all buses is kept at minimum, no bus is overloaded, and the time required to traverse any route does not exceed a maximum allowed policy.

   j) Minimizing deadheading for shared mobility companies (minimize the miles driven with no passenger) or for delivery services providers (minimize the models driven without cargo)
   k) Planning/replanning of transport corridors and city streets to accommodate more pedestrians, cyclists, and riders in shared transportation and less cars
   l) Finding the optimal placement of 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.

2.6 Summary

- Search and optimization problems can be classified based on number of decision variables (univariate, multivariate problems), type of decision variables (continuous, discrete, mixed-integer), number of objective functions (mono-objective, multi-objective, constraint-satisfaction problems), landscape of objective function (unimodal, multimodal, deceptive), number of constraints (unconstrained and constrained problems) and linearity of objective functions and constraints (linear problems and nonlinear problems).
- Based on expected quality of solutions and the permissible search time to find the solutions, optimization problems can also be categorized into design problems/strategic functions, planning/tactical problems, and control problems/operational functions.
- Search and optimization algorithms can be classified based on the way the search space is explored (local versus global search), corresponding optimization speeds (online versus offline search/optimization), determinism of the algorithm (deterministic versus stochastic).
- Heuristics (also known as "mental shortcuts" or "rules of thumb") facilitate finding acceptable (optimal or near-optimal) solutions to a complex problem in a reasonably practical time.
- Metaheuristics are high-level strategies used to guide and modify other heuristics to enhance their performance.
- Nature-inspired algorithms are computational models that mimic or reverse engineer the intelligent behaviors observed in nature to solve complex ill-structured problems.

[1] M. Held and R. M. Karp, "A dynamic programming approach to sequencing problems," Journal of the Society for Industrial and Applied mathematics, 10(1), 196-210, 1962.

[2] M. T. Goodrich and R. Tamassia, "The christofides approximation algorithm," Algorithm Design and Applications. Wiley, 513-514, 2015.

[3] Damavandi, N., & Safavi-Naeini, S. (2005). A hybrid evolutionary programming method for circuit optimization. IEEE Transactions on Circuits and Systems I: Regular Papers, 52(5), 902-910.

[4] Paul A. Jensen. Operations Research Models and Methods. University of Texas at Austin, 2004.

[5] Alaa Khamis and Mohammed Ashraf, "A differential evolution-based approach to design all-terrain ground vehicle wheels," 2017 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC). IEEE, 2017.

[6] K. Veselić, "Finite catenary and the method of Lagrange," SIAM review 37.2 (1995): 224-229.

[7] J. Kalcsics and Z. Ríos-Mercado Roger, “Districting problems,” Location science. Springer, Cham, 2019. 705-743.

[8] S. M. Almufti, “Historical survey on metaheuristics algorithms,” International Journal of Scientific World, 7(1), 1, 2019.

3 Blind Search Algorithms

This chapter covers

- **Applying different graph types for practical use**
- **Graph search algorithms**
- **Using graph traversal algorithms to explore the structure of a tree or a graph and to find a path between two nodes**
- **Using blind search algorithms to find the shortest path between two nodes in a graph**
- **Solving a real-world routing problem using graph search algorithms**

You have already been introduced to deterministic and stochastic algorithms in chapter 2. In this chapter, we will focus on deterministic algorithms, specifically blind search algorithms, and their applications in exploring tree or graph structures and finding the shortest path between nodes. Using these algorithms, you can explore a maze from an initial state to a goal state, solve N-puzzle problems, figure out the distance between you and any other person on a social media graph, search a family tree to determine the exact relationship between any two people who are related or find the shortest path between any origin (e.g., your home) and any destination (e.g., your workplace or any other point of interest). Blind search algorithms are important as they are often more efficient and reasonable to use when dealing with simple, well-defined problems.

3.1 Introduction to Graphs

A graph is a non-linear data structure that consists of three sets: vertices (sometimes called nodes), edges (sometimes called links), and a set representing relations between vertices and edges. A graph can be represented mathematically by G where G=(V,E). For a graph G, V represents the set of nodes or vertices, and E represents the set of edges or links. Various attributes can also be added as components to the edge tuple, such as edge length, capacity, or any other unique properties (i.e., road material). Graphs can be classified into undirected, directed, multi-graph, acyclic, and hypergraphs.

An undirected graph is one where a set of nodes are connected using bidirectional edges. This means that the order of two connected nodes is not essential. A tree, as a specialized case of a graph, is a connected graph with no circuits and no self-loops. In graph theory, a self-loop, also called a circuit, is an edge in a graph that connects a vertex/node to itself. In other words, it is an edge that has the same starting vertex and ending vertex. Trees are usually considered as undirected graphs. *networkx* is a commonly used Python library for creating, manipulating, and studying the structure, dynamics, and functions of graphs and complex networks (see Appendix A). The following example shows how to use *networkx* to create an undirected graph.

Listing 3.1 Creating an undirected graph using `networkx`

```
import networkx as nx

graph = nx.Graph()

nodes = list(range(5))    #A
graph.add_nodes_from(nodes)

edges = [(0,1),(1,2), (1,3), (2,3),(3,4)]    #B
graph.add_edges_from(edges)

nx.draw_networkx(graph, font_color="white")
```

#A Generate a list of nodes from 0 to 4
#B Define a list of edges

The output of the code is shown in Figure 3.1.

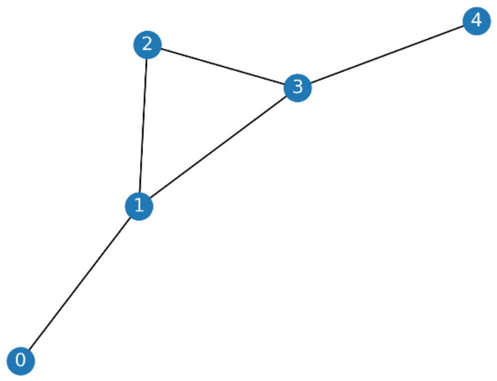


Figure 3.1 Undirected graph.

A directed graph is a graph in which a set of nodes are connected using directional edges. Directed graphs have many applications. For example, directed graphs are used to represent flow constraints (e.g., one-way streets) relations (e.g., causal relationships) and dependencies (e.g., some tasks depend on the completion of other tasks). This example shows how to use *networkx* to create a directed graph.

Listing 3.2 Creating a directed graph using networkx

```
import networkx as nx

graph = nx.DiGraph()    #A
nodes = list(range(5))
edges = [(0,1),(1,2), (1,3), (2,3),(3,4)]
graph.add_edges_from(edges)
graph.add_nodes_from(nodes)
nx.draw_networkx(graph, font_color="white")
```

#A DiGraph allows for directed edges

Code output is shown in Figure 3.2. Note the arrows indicating edge direction.

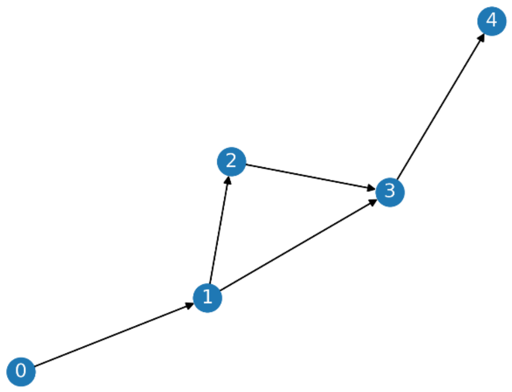


Figure 3.2 A directed graph.

A multi-graph is a graph in which multiple edges may connect the same pair of vertices. These edges are called parallel edges. Multi-graphs can be used to represent complex relationships between nodes such as multiple parallel roads between two locations in traffic routing, multiple capacities and demands in resource allocation problems and multiple relationships between individuals in social networks to name just a few. Unfortunately, *networkx* is not particularly good at visualizing multi-graphs with parallel edges. This example shows how to use *networkx* in conjunction with matplotlib to create a multi-graph.

Listing 3.3 Creating a multi-graph using networkx

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

graph = nx.MultiGraph()
nodes = list(range(5))
edges = [(0,1),(0,1),(4,3),(1,2), (1,3), (2,3),(3,4),(0,1)]
graph.add_nodes_from(nodes)
graph.add_edges_from(edges)

pos = nx.kamada_kawai_layout(graph)     #A
ax = plt.gca()

for e in graph.edges:      #B
     ax.annotate("",xy=pos[e[0]], xycoords='data', xytext=pos[e[1]], textcoords='data',
       arrowprops=dict(arrowstyle="-", connectionstyle=f"arc3, rad={0.3*e[2]}"),zorder=1)
       #B

nx.draw_networkx_nodes(graph, pos)     #C
nx.draw_networkx_labels(graph,pos, font_color='w')     #C

plt.show()
```

#A Node positions are generated using Kamada-Kawai path-length cost-function.
#B Draw each edge one at a time, modifying the curvature of the edge based on its index (i.e. the 2nd edge between nodes 0 and 1)
#C Draw nodes and node labels

It is worth noting that *kamada_kawai_layout* attempts to position nodes on the space so that the geometric (Euclidean) distance between them is as close as possible to the graph-theoretic (path) distance between them. Figure 3.3 shows an example of a multi-graph generated by the code.

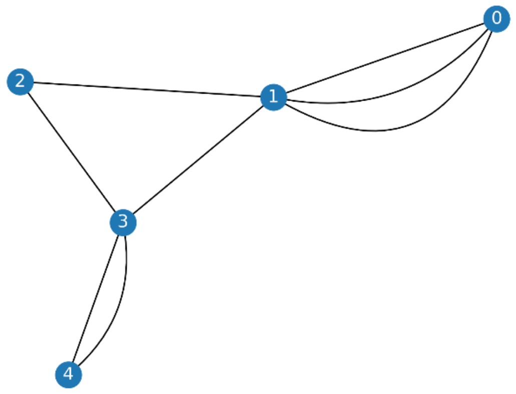


Figure 3.3 Example of a multi-graph. Notice the three parallel edges connecting nodes 0 and 1, as well as the two edges connecting nodes 3 and 4.

As the name implies, an acyclic graph is a graph without cycles. A cycle in a graph is a path that starts and ends at the same node and traverses through at least one other node. In task scheduling, acyclic graphs can be used to represent the relationships between tasks where each node represents a task and each directed edge represents a precedence constraint. This constraint means that the task represented by the end node cannot start until the task represented by the start node is completed. Assembly line balancing problem is discussed in Chapter 6 as an example of scheduling problems. Listing 3.4 shows how to use *networkx* to create and verify an acyclic graph.

Listing 3.4 Creating an acyclic graph using networkx

```
import networkx as nx

graph = nx.DiGraph()
nodes = list(range(5))
edges = [(0,1), (0,2),(4,1),(1,2),(2,3)]
graph.add_nodes_from(nodes)
graph.add_edges_from(edges)
nx.draw_kamada_kawai(graph, with_labels=True, font_color='w')

nx.is_directed_acyclic_graph(graph)     #A
```

#A Check if the graph is acyclic

An example of an acyclic graph is shown in Figure 3.4

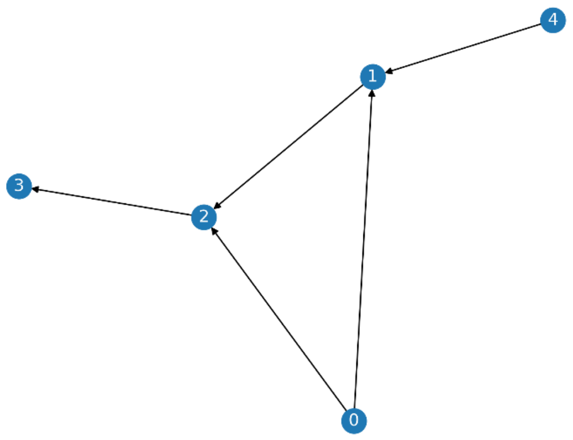


Figure 3.4 An acyclic graph. There is no path that cycles back to any starting node.

A hypergraph is a generalization of a graph in which the generalized edges (called hyperedges) can join any number of nodes. Hypergraphs are used to represent complex networks to capture higher-order many-to-many relationships in several domains such as social media, information systems, computational geometry, computational pathology, and neuroscience. For example, a group of people working on a project can be represented by a hypergraph. Each person is represented by a node and the project is represented by a hyperedge. The hyperedge connects all the people working on the project, regardless of how many people are working on it. The hyperedge can also contain other attributes such as the project's name, the start and end date, the budget, etc.

The following example shows how to use *HyperNetX (HNX)* to create a hypergraph. *HNX* is a Python library that enables modeling the entities and relationships found in complex networks as hypergraphs.

Listing 3.5 Creating a hypergraph using hypernetx

```
import hypernetx as hnx

data = {
     0: ("A","B","G"),
     1: ("A","C","D","E","F"),
     2: ("B","F"),
     3: ("A","B","D","E","F","G")
}    #A
H = hnx.Hypergraph(data) #B
hnx.draw(H)   #C
```

#A The data for the hypergraph comes as a key-value pair of hyperedge name-hyperedge node groups
#B creates a hypergraph for the provided data
#C visualize the hypergraph

Figure 3.5 shows an example of a hypergraph.

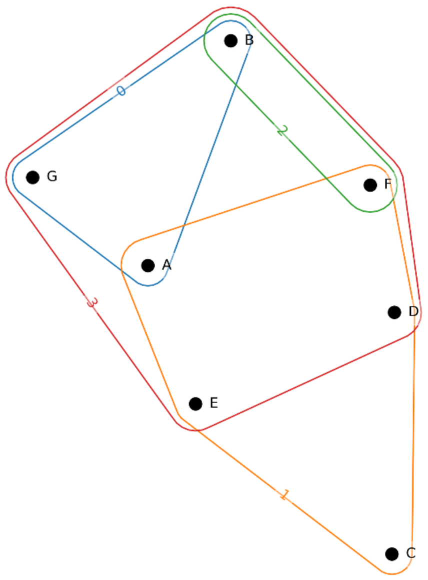


Figure 3.5 An example of a hypergraph. Hyperedges can connect more than two nodes, such as hyperedge 0, which links nodes A, B, and G.

Graphs can also be weighted or unweighted. In a weighted graph, a weight is assigned to each edge. For example, in the case of road networks, the edges can be weighted, meaning that they can have a value that represents the cost of traversing the road. This value can be distance, time, or any other metric. In telecommunications networks, the weight might represent the cost of utilizing that edge or the strength of the connections between the communication devices. Listing 3.6 shows how to create and visualize a weighted graph between telecommunication devices. The weights in this example represent the speed of connected between the devices in Mbps.

Listing 3.6 Creating a weighted graph using networkx

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

G = nx.Graph() #A

G.add_node("Device1", pos=(0,0)) #B
G.add_node("Device2", pos=(0,2)) #B
G.add_node("Device3", pos=(2,0)) #B
G.add_node("Device4", pos=(2,2)) #B

G.add_weighted_edges_from([("Device1", "Device2", 45.69),
                           ("Device1", "Device3", 56.34),
                           ("Device2", "Device4", 18.5)]) #C

pos = nx.get_node_attributes(G, 'pos') #D
nx.draw(G, pos, with_labels=True) #E
nx.draw_networkx_edge_labels(G, pos, edge_labels={(u, v): d['weight'] for u, v, d in
       G.edges(data=True)}) #E
plt.show() #E
```

#A Create an empty weighted graph
#B Add nodes to the graph (representing devices)
#C Add weighted edges to the graph (representing connections)
#D Get node position attributes from graph
#E Draw the graph

Running this code generated the weighted graph shown in Figure 3.6.

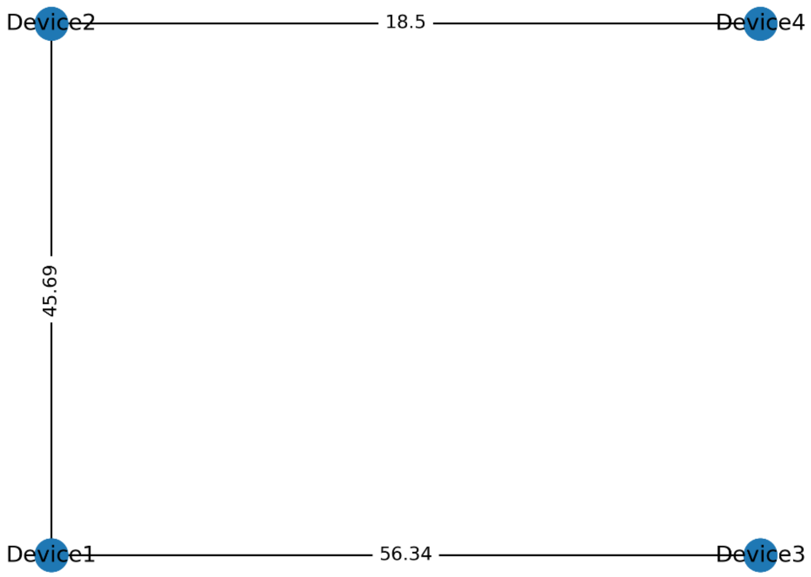


Figure 3.6 Example of a weighted graph.

Graphs are everywhere. Search engines like Google see the internet as a giant graph where each web page is a node, and two pages are joined by an edge if there is a link from one page to the other. A social media platform like Facebook treats each user profile as a node on a social graph, and two nodes are said to be connected if they are each other's friends or have social ties. The concept of "following" a user, such as on a platform like Twitter, can be represented by a directional edge, where user *A* can follow user *B*, but the reverse is not necessarily true. Table 3.1 shows the meanings of nodes and edges on different platforms like Facebook, Twitter, LinkedIn, Instagram and TikTok.

Table 3.1 Examples of graphs in the context of social media

Social Media Platform	Nodes	Edges	Type of Edge
Facebook	Users, groups, posts and events	Friendship, group membership, messages, creation of posts and reactions on posts	Undirected: like or react or a comment Directed: friend request
Twitter	Users, groups, unregistered persons and posts	Following, group membership, messages, creation of posts and reactions on posts	Undirected: a mention or a retweet Directed: following relationship (when you follow a person, he/she does not automatically follow you back)
LinkedIn	Users, groups, unregistered persons, posts, skills and jobs	Connections, group membership, posting, reactions on posts, messages, endorsement, invitations, recommending job	Undirected: an endorsement or a recommendation Directed: connection
Instagram	Users, comment, container for publishing a post, hashtag, media (e.g., photo, video, story, or album), page (Facebook page)	Following, occurrences of two hashtags in the same post.	Undirected: like or a comment Directed: follow relationship
TikTok	Users, hashtags, locations, and keywords	Relationships between users such as following, liking, and commenting	Undirected: like or a comment Directed: follow relationship

In a road network, graph nodes represent landmarks such as intersections and points of interest (POI), and the edges represent the roads. In road network graph, most of the edges are directed, meaning that they have a specific direction, and they may have additional information such as length, speed limit, capacity, etc. Each edge is a two-endpoint connection between two nodes, where the direction of the edge represents the direction of traffic flow. A route is a sequence of edges connecting the origin node to the destination node. *osmnx* is a Python library developed to simplify the retrieving and manipulating of data from *OpenStreetMap (OSM)*. OpenStreetMap is a crowdsourced geographic database of the world (see Appendix B). osmnx offers the ability to download filtered data from OSM and returns the network as a *networkx* graph data structure. It can also convert a text descriptor of a place into a *networkx* graph (see Appendix A). Let's use the University of Toronto as an example as you can see in Lisitng 3.7.

Listing 3.7 University of Toronto example

```
import osmnx

place_name = "University of Toronto"

graph = osmnx.graph_from_address(place_name) #A
osmnx.plot_graph(graph,figsize=(10,10))
```

#A graph_from_address can also take city names and mailing addresses as input

Figure 3.7 shows open street map of the area around St. George Campus of University of Toronto.

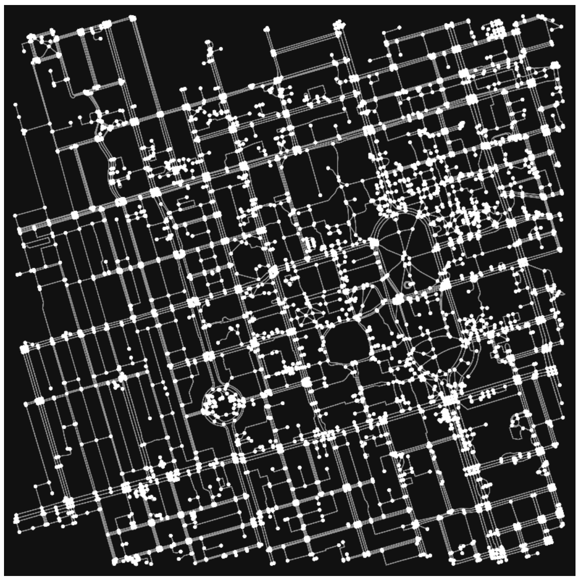


Figure 3.7 St. George Campus - University of Toronto

The graph shows the edges and nodes of the road network surrounding the University of Toronto's St. George campus in downtown Toronto. While it may look visually interesting, it lacks the context of surrounding geographic features. Let's use a *folium* map (see Appendix A) as a base layer to provide street names, neighborhood names, and even building footprints.

```
graph = osmnx.graph_from_address(place_name)
osmnx.folium.plot_graph_folium(graph)
```

Figure 3.8 shows road network surrounding the University of Toronto's St. George campus.

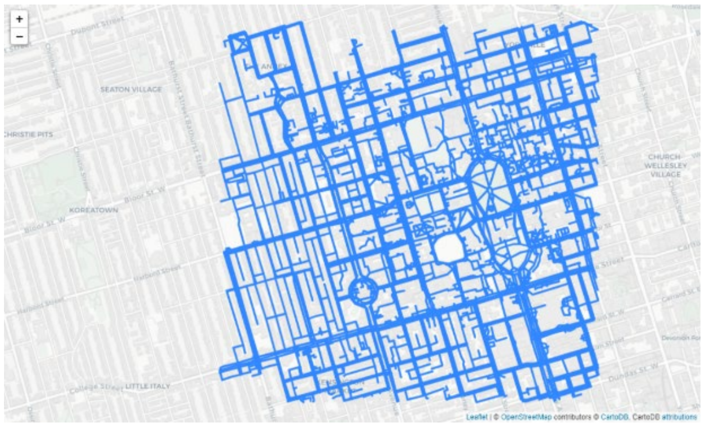


Figure 3.8 Road network around St. George Campus - University of Toronto

Suppose you want to get from one location to another on this campus. For example, starting at the King Edward VII Equestrian Statue near Queen's Park, you need to cross the campus to attend a lecture at the Bahen Centre for Information Technology. Later in this chapter, you will see how you can calculate the shortest path between these two points. For now, let's just plot these two locations on the map using *folium* (see Appendix A):

Listing 3.8 plot with folium

```
import folium

center=(43.662643, -79.395689) #A
source_point = (43.664527, -79.392442) #B
destination_point = (43.659659, -79.397669) #C

m = folium.Map(location=center, zoom_start=15) #D
folium.Marker(location=source_point,icon=folium.Icon(color='red',icon='camera',
       prefix='fa')).add_to(m) #E
folium.Marker(location=center,icon=folium.Icon(color='blue',icon='graduation-cap',
       prefix='fa')).add_to(m) #E
folium.Marker(location=destination_point,icon=folium.Icon(color='green',icon='university',
       prefix='fa')).add_to(m) #E

m
```

#A The GPS coordinates (latitude and longitude) of University of Toronto
#B The GPS coordinates of the Equestrian Statue as a source point
#C The GPS coordinates of the Bahen Centre for Information Technology as destination
#D Create a map centered around a specified point
#E Add markers with icons

Figure 3.9 shows folium map and markers.

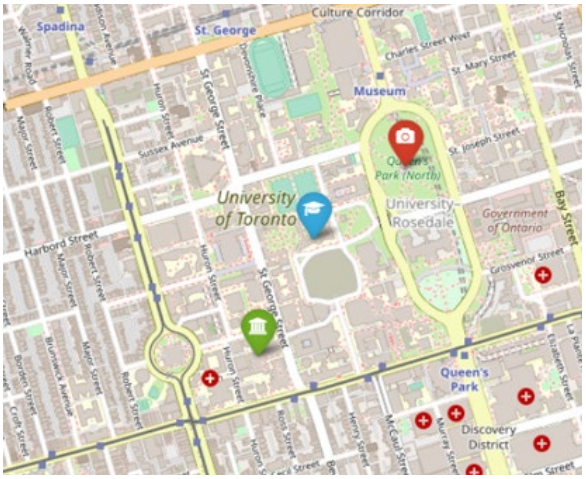


Figure 3.9 Visualizing points of interest using folium markers

The actual output of the code is interactive, and allows for features such as zooming, panning, and even layer filtering (when enabled). Appendix A provides more details about map visualization libraries in Python.

3.2 Graph Search

As previously mentioned in Chapter 2, search algorithms can be broadly classified into deterministic and stochastic algorithms. In deterministic search, the search algorithm follows a rigorous procedure and its path and values of both design variables and the functions are repeatable. The algorithm will follow the same path for the same starting point whether you run the program, whether it's today or ten years in the future. On the other hand, in stochastic search, the algorithm always has some randomness, and the solution is not exactly repeatable. This means that each time, you run the algorithm, you may get slightly different results.

Based on the availability of information about the search space or domain knowledge (e.g., the distance from the current state to the goal), deterministic search algorithms can be broadly classified into blind (or uninformed) and informed search, as illustrated in Figure 3.10. Some of these algorithms, such as Kruskal's MST algorithm, will be covered in later chapters.

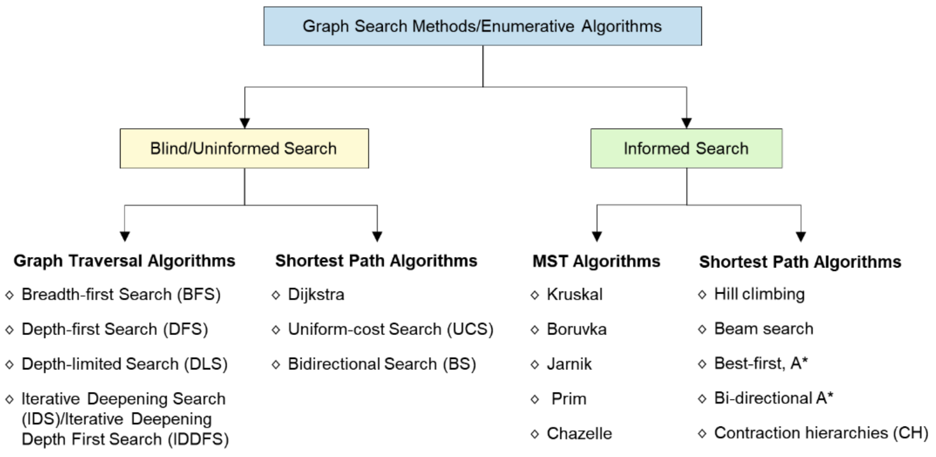

Graph Search Methods/Enumerative Algorithms
Blind/Uninformed Search
Informed Search
Graph Traversal Algorithms
Breadth-first Search (BFS)
Depth-first Search (DFS)
Depth-limited Search (DLS)
Iterative Deepening Search (IDS)/Iterative Deepening Depth First Search (IDDFS)
Shortest Path Algorithms
Dijkstra
Uniform-cost Search (UCS)
Bidirectional Search (BS)
MST Algorithms
Kruskal
Boruvka
Jarnik
Prim
Chazelle
Shortest Path Algorithms
Hill climbing
Beam search
Best-first, A*
Bi-directional A*
Contraction hierarchies (CH)

Figure 3.10 Graph Search Methods.

This chapter focuses on blind search algorithms. Blind search (also known as uninformed search) is a search approach where no information about the search space is needed. The blind search terminates once the first solution is found. However, the search space may contain numerous valid but non-optimal solutions. Thus, a blind search may return a solution that meets all the requirements but does so in a non-optimal way. An optimal solution can be found by running the blind search following an exhaustive search or brute-force strategy to find all the feasible solutions, which can then be compared to select the best one. This is similar to applying British Museum algorithm, which follows the human problem-solving strategy by checking all possibilities one by one. Given the fact that blind search treats every node in the graph/tree equally, this search approach is often referred to as uniform search.

Breadth-first Search (BFS), Depth-first Search (DFS), Depth-limited Search (DLS), Iterative Deepening Search (IDS) or Iterative Deepening Depth First Search (IDDFS), Dijkstra algorithm, Uniform-cost Search (UCS), and Bidirectional Search (BS) are examples of blind search algorithm. BFS is a graph traversal algorithm that builds the search tree by levels.

- DFS is a graph traversal algorithm that first explores nodes going through one adjacent of the root, then next adjacent until it finds a solution or until it reaches a dead end.
- DLS is a DFS with a predetermined depth limit.
- IDDFS combines DFS's space efficiency and BFS's fast search by incrementing the depth limit until the goal is reached.
- Dijkstra's algorithm solves the single-source shortest path problem for a weighted graph with non-negative edge costs.
- UCS is a variant of Dijkstra's algorithm that uses the lowest cumulative cost to find a path from the source to the destination. It is equivalent to the BFS algorithm if the path cost of all edges is the same.

- BS is a combination of forward and backward search. It searches forward from the start and backward from the goal simultaneously.

The following sections discuss graph traversal algorithms and shortest path algorithms, focusing on BFS, DFS, Dijkstra algorithm, UCS, and BS as examples of blind search approaches.

3.3 Graph Traversal Algorithms

Graph traversal is the process of exploring the structure of a tree or a graph by visiting the nodes following a specific, well-defined rule. This category of graph search algorithms only seeks to find a path between two nodes without optimizing for the length of the final route.

3.3.1 Breadth-first Search (BFS)

BFS is an algorithm where the traversal starts at a specified node (i.e., the source or starting node) and continues along with the graph layerwise, thus exploring all of the current node's neighboring nodes (those directly connected to the current node). The algorithm then searches the next-level neighbor nodes if a result is not found. This algorithm finds a solution if one exists, assuming that a finite number of successors/branches always follow any node. Algorithm 3.1 shows the BFS steps.

Algorithm 3.1 Breadth-first Search (BFS) Algorithm

```
BREADTH-FIRST-SEARCH(source,destination) return a route

queue ← a FIFO initialized with source node
explored ← empty
found ← False

while queue is not empty and found is False do
    node ← queue.dequeue()
    add node to explored
    for child in node.expand() do
    if child is not in explored and child is not in queue then
        if child is destination then
            route ← child route()
            found ← True
        add child to queue
return route
```

BFS uses the queue as a data structure to maintain the states to be explored. A queue is a First-In-First-Out (FIFO) data structure, where the node that has been sitting on the queue for the longest time is the next node to be expanded. BFS dequeue a state off the queue, then enqueue its successors back on the queue.

Let's consider the 8-puzzle (sometimes called sliding-block, tile-puzzle) problem. The puzzle consists of an area divided into a 3x3 grid. Tiles are numbered 1 through 8, except for an empty (or blank) tile. The blank tile can be moved by swapping its position with any tile directly adjacent (up, down, left, right). The puzzle's goal is to place the tiles so that they are arranged in order. Variations of the puzzle allow the empty tile to end up either at the first or last position. This problem is an example of a well-structured problem (WSP) with the following well-defined components:

- States: location of blank and location of the eight tiles
- Operator (successor): blank moves left, right, up, and down
- Goal: match the state given by the Goal state
- Solution/Path: sequence through state space
- Stopping Criteria: ordered puzzle (reached Goal state)
- Evaluation Criteria: number of steps or path cost (the path length).

Figure 3.11 illustrates the BFS steps to solve the 8-puzzle problem and the search tree traversal order. In this figure, the state represents the physical configuration of the 8-puzzle and each node in the search tree is a data structure that includes information about parent, children, depth and cost of path from initial state to this node. Level-1 nodes are generated from left to right by moving the blank title left, up and right respectively. Moving forward, level 2 nodes are generated by expanding the previously generated nodes in level 1 avoiding the previously explored nodes. We keep repeating this procedure to traverse all the possible nodes or until we hit the goal (the blue grid). The number of steps to reach the goal will depend mainly on the initial state of the 8-puzzle board. The highlighted numbers show the order of traverse. As you may notice, BFS progresses horizontally before we proceed vertically.

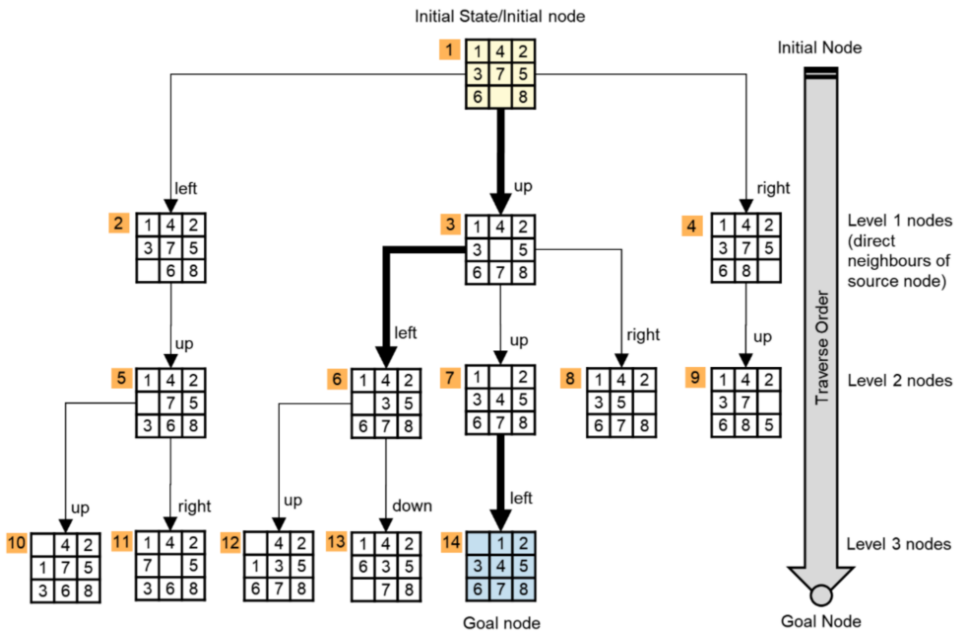


Figure 3.11 Using BFS to solve 8-puzzle problem

The following example utilizes a generic BFS algorithm developed for this book, which can be found in the *search-optimization-tools* python package (see Appendix A for installation instructions). The algorithm takes a starting and goal state as inputs and returns a *Solution* object. The *Solution* object contains the actual result and some performance metrics, such as processing time, maximum space used, and the number of solution states explored. We also import the *State* class and `visualize` function from the *eight_puzzle_problem.py* file included in the code folder for this chapter available at the GitHub repo of the book. This file helps manage some data structures and utility functions and allows us to reuse this problem's structure later on with different algorithms.

Listing 3.9 Solving 8-puzzle problem using BFS

```
from optimization_algorithms_tools.algorithms.graph_search import BFS    #A

init_state = [[1,4,2], [3,7,5], [6,0,8]]

goal_state = [[0,1,2], [3,4,5], [6,7,8]]


init_state = State(init_state)  #B
goal_state = State(goal_state)


if not init_state.is_solvable():    #C
     print(“This puzzle is not solvable.”)
else:

    solution = BFS(init_state, goal_state)
    print(f"Process time: {solution.time} s")
    print(f"Space required: {solution.space} bytes")
    print(f"Explored states: {solution.explored}")
    visualize(solution.result) #D
```

#A The BFS algorithm used here is imported from a library called search-optimization-tools (see Appendix A for installation instructions).
#B See State class in the complete listing
#C Some boards are not solvable
#D See visualization function in the complete listing

The following is an example solution, given the above inputs:

```
Process time: 0.015625 s
Space required: 624 bytes
Explored states: 7
```

Figure 3.12 shows the state changes following the BFS algorithm.

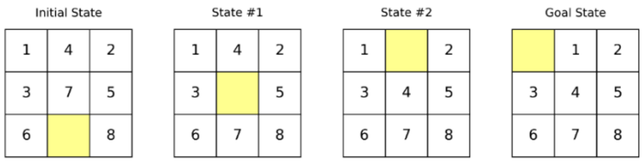


Figure 3.12 Step-by-step BFS solution using Python. BFS searches for a solution but does not consider optimality.

To really understand how BFS works, let's consider a simple path planning problem. This problem addresses finding a collision-free path for a mobile robot or an autonomous vehicle from a start position to a given destination amidst a collection of obstacles.

- Step-1: Adding source node to the queue (Figure 3.13)

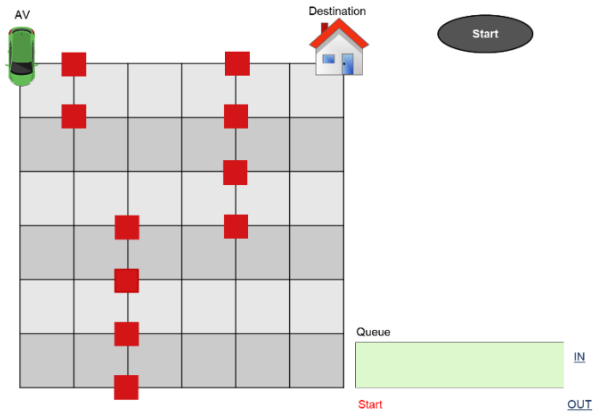


Figure 3.13 Solving path planning problem using BFS – Step 1

- Step-2: Exploring S node as E and SE nodes are obstructive (Figure 3.14)

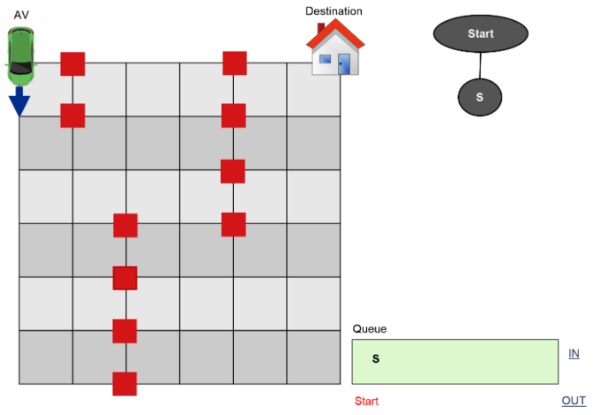


Figure 3.14 Solving path planning problem using BFS – Step 2

- Step-3: Taking S out (First-In-First-Out) and exploring its neighboring nodes (S and SE) as E is an obstructive node (Figure 3.15).

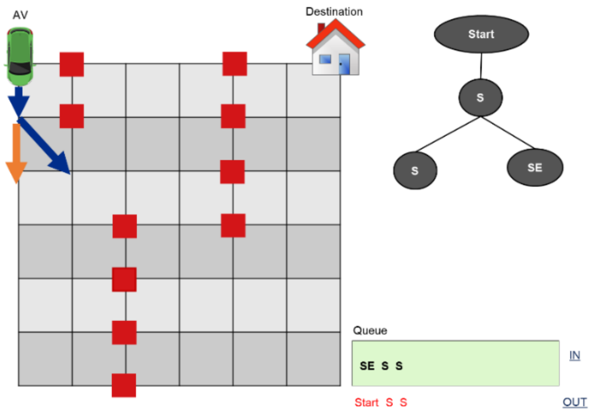


Figure 3.15 Solving path planning problem using BFS – Step 3

- Step-4: Taking S out (First-In-First-Out) and exploring its neighboring nodes (S and SE) (Figure 3.16).

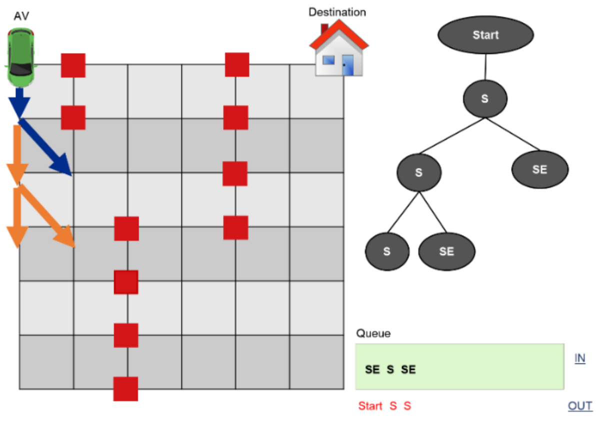


Figure 3.16 Solving path planning problem using BFS – Step 4

- Step-5: Taking SE out (First-In-First-Out) and exploring its neighboring nodes (E and NE) (Figure 3.17).

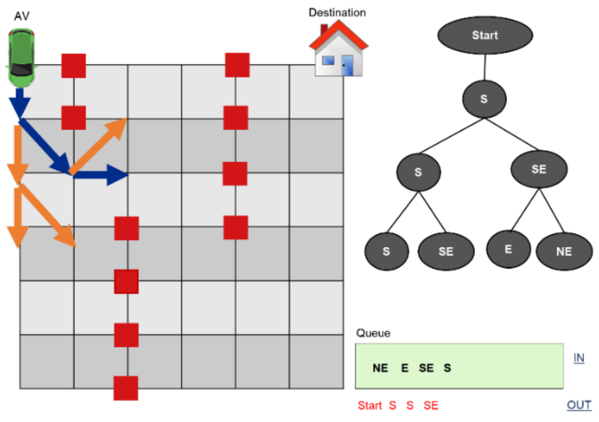


Figure 3.17 Solving path planning problem using BFS – Step 5

- Step-6: The FIFO queue continues until the goal node is found (Figure 3.18). Assuming the goal node is E, we can trace back up the tree to find the path from the source node to the goal, which will be Start-S-SE-E.

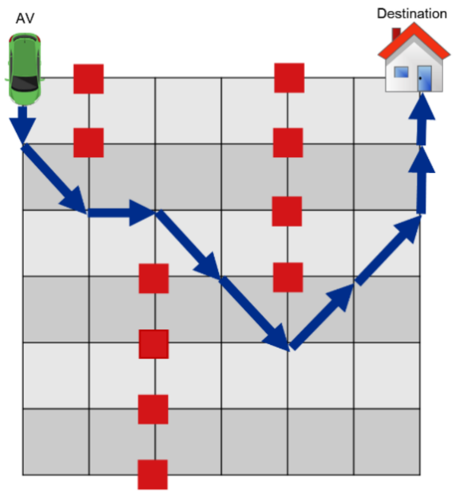


Figure 3.18 Solving path planning problem using BFS – Final route

In BFS, every node generated must remain in memory. The number of nodes generated is at most: O(b^d), where b represents the maximum branching factor for each node (i.e., the number of children the node has) and d is the depth one must expand to reach the goal. In the previous example, with E as a goal node (b=2, d=3), the total number of traversed nodes is 2^3=8, including the start node.

Aside from the ability to solve the problem at hand, algorithm efficiency is evaluated based on run-time (time complexity), memory requirements, and the number of primitive operations to solve the problem in the worst case. Examples of these primitive operations include, but are not limited to, expression evaluation, variable value assignment, array indexing, and method/function calls. Table 3.2 shows examples of algorithm complexities.

Table 3.2 Algorithm complexity

Notation	Name	Effectiveness	Description	Examples
O(1)	Constant	Excellent	Running time does not depend on the input size. As the input size grows, the number of operations does not get impacted.	Variable declaration Accessing an array element Retrieving information from a hash-table lookup Insertion and removal from a queue pushing and popping on a stack
O(log n)	Logarithmic	High	As the input size grows, the number of operations grows very slowly. Whenever n doubles or tripled, etc., the running time increases by a constant.	Binary search
O(n^c), 0<c<1	Fractional power/ sublinear	High	As the input size grows, the number of operations is replicated in multiplication	Testing Graph Connectedness Approximating the number of connected components in graph Approximating the weight of the minimum spanning tree (MST)
O(n)	Linear	Medium	As the input size grows, the number of operations increases linearly. Whenever n doubles, the running time doubles.	Print out an array's elements Simple search Kadane's Algorithm
O(n log n)=O(log n!)	Linearithmic, loglinear or quasilinear	Medium	As the input size grows, the number of operations increases slightly faster than linear.	Merge sort Heapsort Timsort
O(n^c), c>1	Polynomial or algebraic	Low	As the input size grows, the number of operations increases as the exponent increases	Minimum spanning tree (MST) Matrix determinant

$O(n^2)$	Quadratic	Low	Whenever n doubles, the running time increases fourfold. Quadratic function is practical for use only on small problems	Selection sort Bubble sort Insertion sort
$O(n^3)$	Cubic	Low	Whenever n doubles, the running time increases eightfold. Cubic function is practical for use only on small problems	Matrix multiplication
$O(c^n)$, $c>1$	Exponential	Very Low	As the input size grows, the number of operations increases exponentially. It is slow and usually not appropriate for practical use.	Power Set Towers of the Hanoi Password cracking Brute force search
$O(n!)$	Factorial	Extremely Low	Extremely slow as all possible permutations of the input data need to be checked. Factorial algorithm is even worse than the exponential function.	Travelling salesman problem Permutations of a string

Big O notation

Big O notation describes the performance or complexity of an algorithm, usually under the worst-case scenario. Big O notation helps us answer the question: "Will the algorithm scale?". To obtain the big O notation for a function f(x), if f(x) is a sum of several terms, the one with the largest growth rate is kept, and all others omitted. Moreover, if f(x) is a product of several factors, any constants (terms in the product that do not depend on x) are omitted. For example, recalling the ticket pricing problem presented in Chapter 1: $f(x)=-20x^2+6200x-350000$. This function is the sum of three terms: $-20x^2+6200x-350000$. Of these three terms, the one with the highest growth rate is the one with the largest exponent as a function of x, namely $-20x^2$. We now apply the second rule: $-20x^2$ is a product of -20 and x^2 in which the first factor does not depend on x. Dropping this factor results in the simplified form x^2. Thus, we say that f(x) is a big-oh of x^2 or mathematically, we can write $f(x)\in O(x^2)$ (pronounced "Order n squared" or "O of n squared"), which represents a quadratic complexity.

Figure 3.19 shows examples of big O notations:

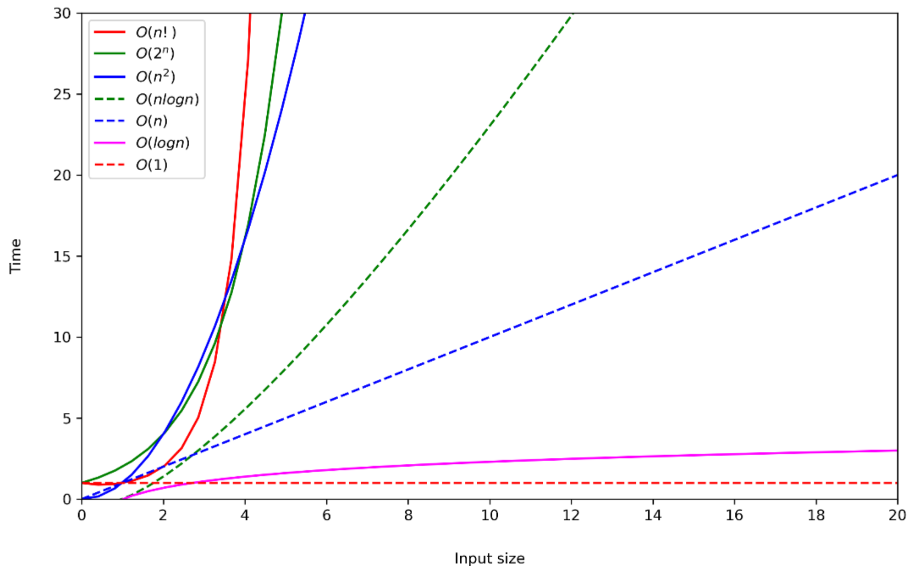


Figure 3.19 Examples of big O notations

Assume a computer with processor speed of one million operations per second is used to handle a problem of size n = 20,000. Table 3.3 shows the running time according to the big-O notation of the algorithm used to solve this problem.

Table 3.3 Algorithm complexity and the running time

Big-O	Running time
O(1)	10^{-6} seconds
O(log n)	14×10^{-6} seconds
O(n)	0.02 seconds
O(n log n)=O(log n!)	0.028 seconds
O(n^2)	6.66 minutes
O(n^3)	92.6 days
O(c^n), c=2	1262.137e+6015 years
O(n!)	5768.665 e+77331 years

For a huge workspace where the goal is deep, the number of nodes could expand exponentially and demand a large memory requirement. In terms of time complexity, For a graph G=(V,E), BFS has a running time of O(|V|+|E|), since each vertex is enqueued at most once and each edge is checked either once (for directed graph) or at most twice (for undirected graph). Time and space complexity of BFS are also defined in terms of branching factor *b* and depth of the shallowest goal *d*. Time complexity is O(b^d) and space complexity is

O(b^d) as well. Let's take a graph with a constant branching factor b=5, nodes of size 1 Kilobyte, and a limit of 1,000 nodes scanned per second. The total number of nodes N is given by the following equation:

$$N=(b(d+1)-1)/(b-1)$$

Equation 3.1

Table 3.4 shows the time and memory requirements to traverse this graph using BFS.

Table 3.4 BFS time and space complexity

Depth *d*	Nodes *N*	Time	Memory
2	31	31 ms	31 KB
4	781	0.781 second	0.78 megabytes
6	19531	5.43 hours	19.5 megabytes
8	488281	56.5 days	488 megabytes
10	12207031	3.87 years	12.2 gigabytes
12	305175781	96.77 years	305 gigabytes
14	7629394531	2419.26 years	7.63 terabytes

Next, we'll take a look at the counterpart to the BFS algorithm, which searches deep into a graph first, rather than breadth-wise.

3.3.2 Depth-first Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going as deep as possible into the graph. Then, when it reaches the last layer with no result (i.e., dead-end is reached), it "backtracks" up a layer and continues the search. In DFS, the deepest nodes are expanded first and nodes of equal depth are ordered arbitrarily. Algorithm 3.2 shows the DFS steps.

Algorithm 3.2 Depth-first Search (DFS) Algorithm

```
DEPTH-FIRST-SEARCH (source,destination) return a route

Stack ← a LIFO initialized with sourcenode
Explored ← empty
Found ← False
while stack is not empty and found is False do
    node ← stack.pop()
    add node to explored
    for child in node.expand() do
        if child is not in explored and child is not in stack then
            if child is destination then
                route ← child.route()
                found ← True
            add child to stack
return route
```

As you may have noticed, the only difference between DFS and BFS is in how the data structure works. Rather than working down layer by layer (FIFO), DFS drills down to the bottom-most layer and moves its way back to the starting node, using a Last-In-First-Out (LIFO) data structure known as a stack. The stack contains the list of discovered nodes. The most recently discovered node is put (pushed) on top of the LIFO stack. The next node to be expanded is then taken (popped) from the top of the stack, and all of its successors are added to the stack. Revisiting the 8-puzzle problem, Figure 3.20 shows the DFS solution for this problem. As you can see, when the algorithm reaches a dead-end or terminal node (node #7), it goes back to the last decision point (node #3) and proceeds with another alternative (node #8 and so on). In this example, a depth bound of 5 is placed to constrain the node expansion. This depth bound makes nodes #6, 7, 10, 11, 13, 14, 16, 17, 22, 23, 26 and 27 terminal nodes in the search tree (i.e., there have no successors).

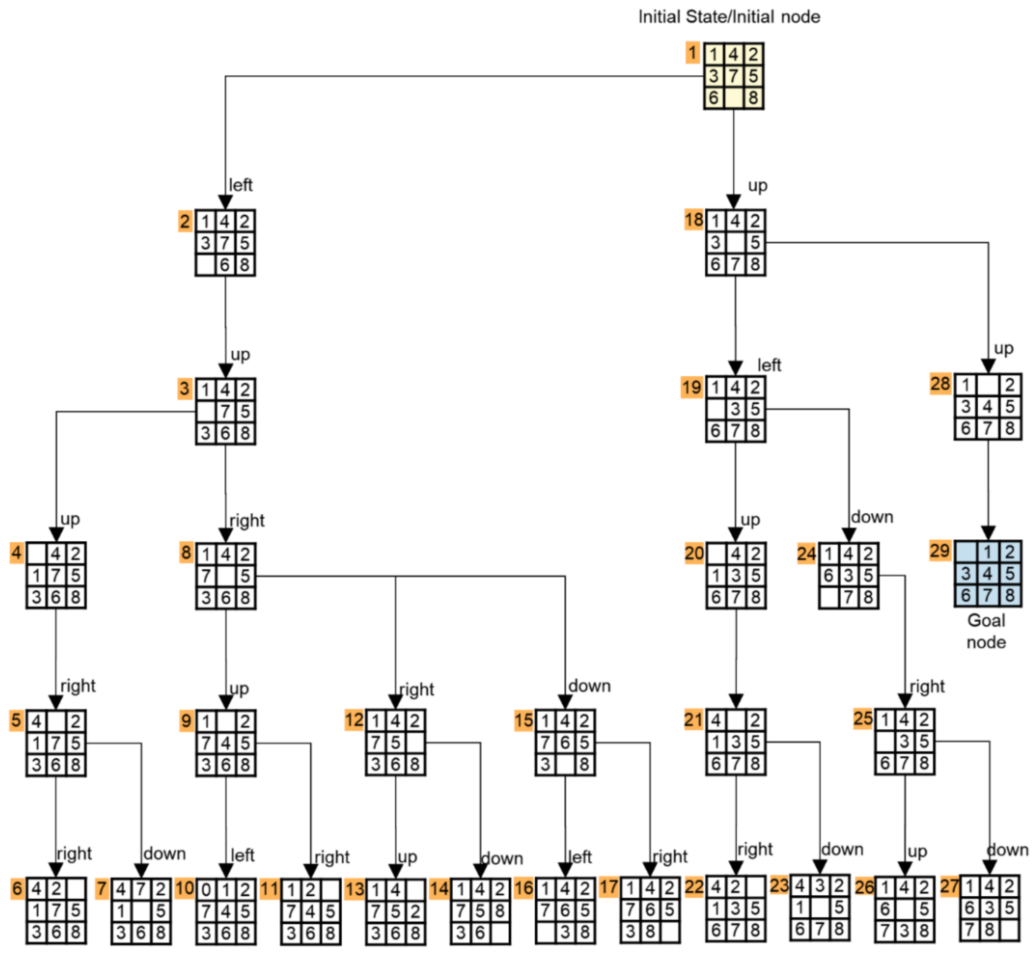


Figure 3.20 Using DFS to solve 8-puzzle problem

As you can see below, we only need to change the algorithm used (DFS). We've also omitted the solution visualization, the reason for which you'll see shortly.

Listing 3.10 Solving 8-puzzle problem using DFS

```
#!pip install optimization_algorithms_tools
from optimization_algorithms_tools.algorithms.graph_search import DFS

init_state = [[1,4,2],[3,7,5],[6,0,8]]
goal_state = [[0,1,2],[3,4,5],[6,7,8]]

init_state = State(init_state)
goal_state = State(goal_state)

if not init_state.is_solvable():    #A
     print("This puzzle is not solvable.")
else:
     solution = DFS(init_state, goal_state)    #B
     print(f"Process time: {solution.time} s")
     print(f"Space required: {solution.space} bytes")
     print(f"Explored states: {solution.explored}")
     print(f"Number of steps: {len(solution.result)}")
```

#A Some puzzles are not solvable
#B The inputs for DFS are the same as for BFS

Here’s an example code run with the above inputs:

```
Process time: 0.5247 s
Space required: 624 bytes
Explored states: 29
Number of steps: 30
```

As you can see, DFS is not great when dealing with very deep graphs, where the solution may be located closer to the top. You can also see why we opted not to visualize the final solution: there are a lot more steps in the solution than we had in BFS! Because the solution to this problem was closer to the root node, the solution generated by DFS is a lot more convoluted (30 steps) than with BFS.

Revisiting the path planning problem, DFS can be used to generate a free of obstacle path from the start location to the destination as follows:

- Step-1: Adding source node to the stack (Figure 3.21).

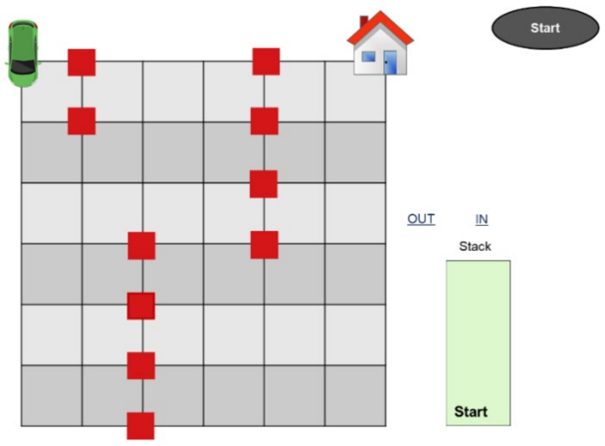


Figure 3.21 Solving path planning problem using DFS – Step 1

- Step-2: Exploring S node as E and SE nodes are obstructive (Figure 3.22).

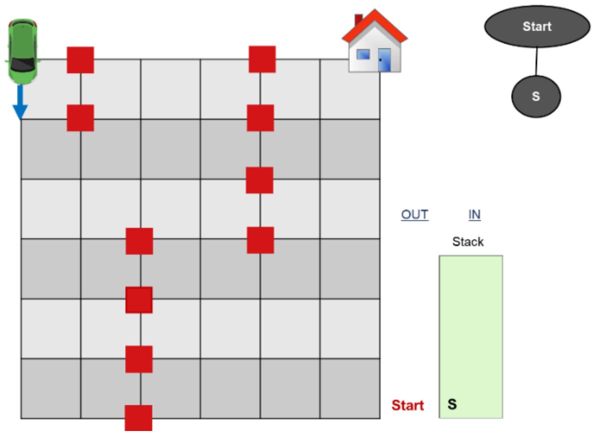


Figure 3.22 Solving path planning problem using DFS – Step 2

- Step-3: Taking S out (Last-In-First-Out) and exploring its neighboring nodes (S and SE) as E is an obstructive node (Figure 3.22).

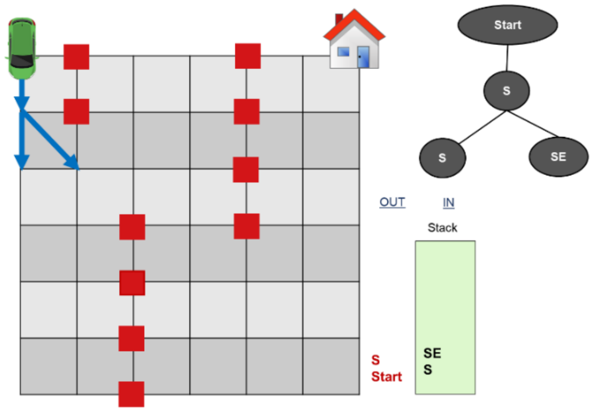


Figure 3.23 Solving path planning problem using DFS – Step 3

- Step-4: Taking SE out (Last-In-First-Out) and exploring its neighboring nodes (SW, S, E and NE) (Figure 3.24).

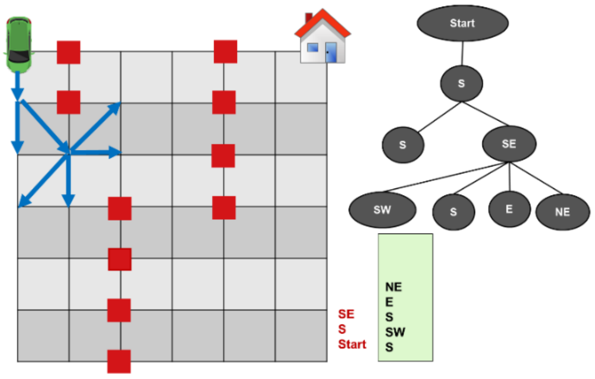


Figure 3.24 Solving path planning problem using DFS – Step 4

- Setp-5: The next node to be expanded would be NE and its successors would be added to the stack and this loop continues until the goal is found. The LIFO stack continues until the goal node is found. Once the goal is found, you can then trace back through the tree to obtain the path for the vehicle to follow.

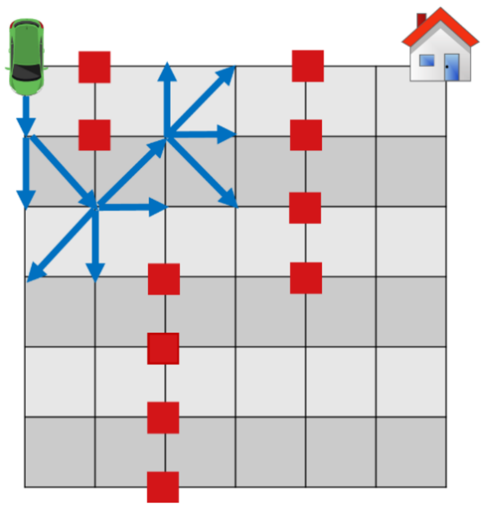


Figure 3.25 Solving path planning problem using DFS

DFS usually requires a considerably less amount of memory compared to BFS. This is mainly because DFS does not always expand out every single node at each depth. However, DFS could continue down an unbounded branch forever even if the goal is not located on that branch in case of search tree with infinite depth. One way to handle this problem is to use constrained depth-first search where the search stops after reaching a certain depth. Time Complexity is O(b^d) where *b* is the branching factor and *d* is the maximum depth of the search tree. This is terrible if *d* is much larger than *b*, but if solutions are found deeply in the tree, it may be much faster than BFS. The space complexity of DFS is O(*bd*), i.e., linear space! This space complexity represents the maximum number of nodes to be stored in the memory. Table 3.5 summarizes a comparison between BFS and DFS.

Table 3.5 BFS versus DFS

	Breadth-first Search (BFS)	Depth-first Search (DFS)
Space complexity	More expensive	Less expensive. Requires only O(*d*) space irrespective of number of children per node.
Time complexity	More time efficient. A vertex at lower level (closer to the root) is visited first before visiting a vertex that is at higher level (far away from the root)	Less time efficient
When it is preferred	• If the tree is very deep • If the branching factor is not excessive • If solution appear at a relatively shallow level (i.e, solution/target is near to the starting point/source/apex node in the tree) • Example: search the royal family tree for someone who is dead a long time ago as he/she would be closer to the top of the tree (e.g., King George VI).	• If the graph/tree is very wide with too many adjacent nodes • If no path is excessively deep • If solutions occur deeply in the tree (i.e., the target is far from the source) • Example: search the royal family tree for someone who is still alive as he/she would be on the bottom of the tree (e.g., Prince William)

In applications where the weight of edges in a graph are all equal (e.g., all length 1), BFS and DFS algorithms outperform shortest path algorithms like Dijkstra's in terms of time. Shortest path algorithms are explained in the following section.

3.4 Shortest Path Algorithms

Dijkstra algorithm, Uniform-Cost Search (UCS) and Bi-directional Dijkstra Search are discussed here as examples of blind search algorithms that try to find the shortest path between a source node and a destination node. Suppose that you were looking for the quickest way to go from home to work. Graph traversal algorithms like BFS and DFS may eventually get you to your destination, but they certainly do not optimize for distance travelled. The following three algorithms generate provably shortest paths between two nodes in a graph.

3.4.1 Dijkstra Search

Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a fully connected graph with non-negative edge path costs, producing a shortest-path tree. Dijkstra's algorithm was published in 1959 and is named after Dutch computer scientist Edsger Dijkstra. This algorithm is the base of several other graph search algorithms commonly used to solve routing problems in popular navigation apps, as illustrated in Figure 3.26. The algorithm follows dynamic programming approaches where the problem is recursively divided into simple sub-problems. Dijkstra's algorithm is uninformed, meaning it does not need to know the target node beforehand and doesn't use heuristic information.

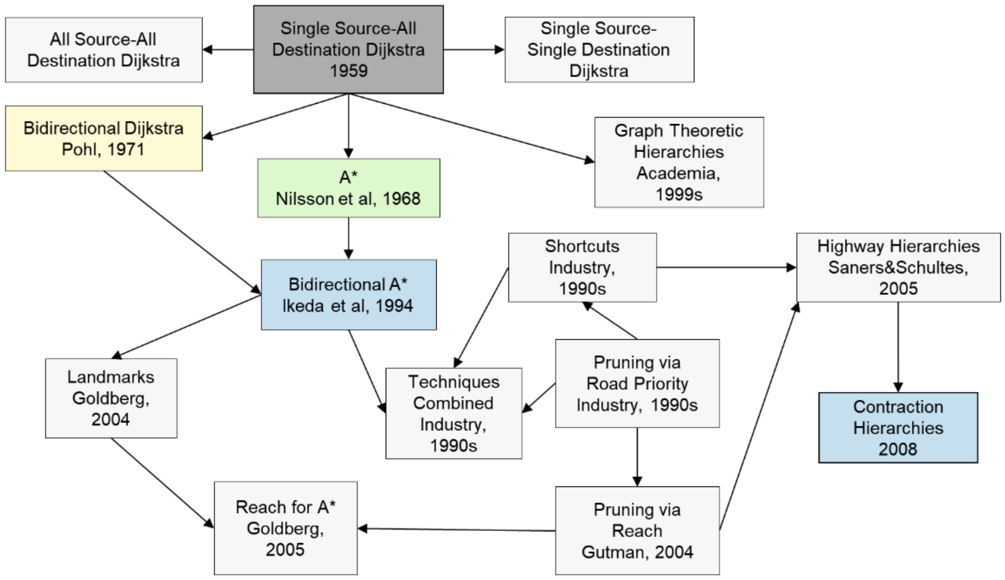

All Source-All Destination Dijkstra
Single Source-All Destination Dijkstra 1959
Single Source-Single Destination Dijkstra
Bidirectional Dijkstra Pohl, 1971
Graph Theoretic Hierarchies Academia, 1999s
A* Nilsson et al, 1968
Shortcuts Industry, 1990s
Highway Hierarchies Saners&Schultes, 2005
Bidirectional A* Ikeda et al, 1994
Pruning via Road Priority Industry, 1990s
Landmarks Goldberg, 2004
Techniques Combined Industry, 1990s
Contraction Hierarchies 2008
Reach for A* Goldberg, 2005
Pruning via Reach Gutman, 2004

Figure 3.26 Dijkstra's algorithm and its variants

Algorithm 3.3 shows the steps of the original version of Dijkstra's algorithm to find the shortest path between a known single source node to all the other nodes in the graph/tree.

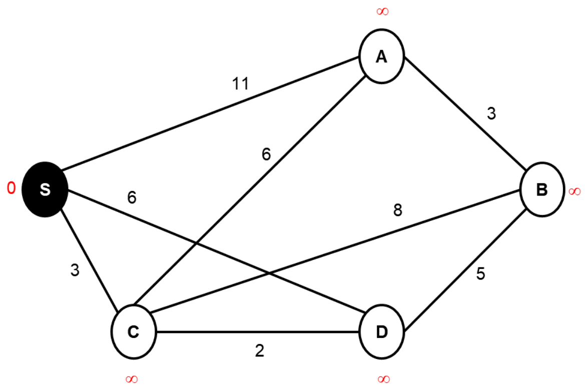
Algorithm 3.3 Dijkstra's Algorithm

```
DIJKSTRA-SEARCH(graph,source) return a list of distances

shortest_dist ← empty
unrelaxed_nodes ← empty
seen ← empty

for node in graph
    shortest_dist[node] = Infinity
    add node to unrelaxed_nodes

    shortest_dist[source] ← 0

while unrelaxed_nodes is not empty do
    node ← unrelaxed_nodes.pop()
    add node to seen
    for child in node.expand() do
        if child in seen then skip
        distance ← shortest_dist[node] + length of edge to child
        if distance < shortest_dist[child] then
            shortest_dist[child] ← distance
            child.parent ← node
return shortest_dist
```

The Dijkstra algorithm and its variants presented in the code for this book are all modified to require a target node. This improves processing time when working with large graphs (i.e., road networks).

Let's illustrate how Dijkstra's algorithm finds the shortest path between any two nodes in a graph. Priority queue is used to pop the element of the queue with the highest priority according to some ordering function (shortest distance between the node and the source node in this case).

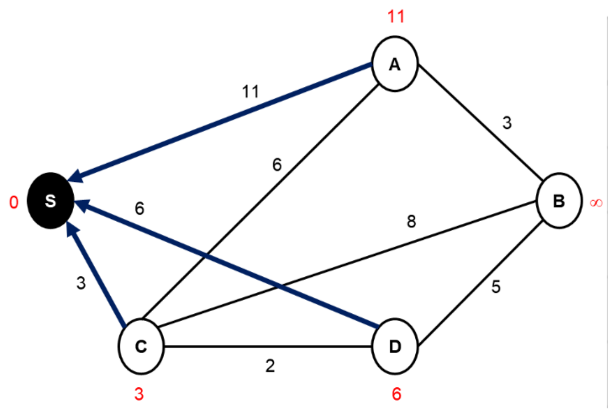
- Step 0: Initial list, no predecessors: Priority Queue = {} (Figure 3.27).



From S to:	Distance	Predecessor
S	0	-
A	∞	-
B	∞	-
C	∞	-
D	∞	-

Figure 3.27 Finding shortest path using Dijkstra's algorithm – Step 0

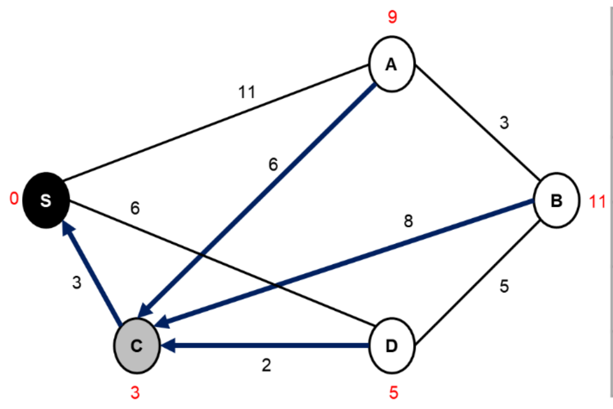
- Step 1: The closest node to the source node is S, add to Priority Queue. Update distances and predecessors for A, C and D. Priority Queue = {S} (Figure 3.28).



From S to:	Distance	Predecessor
S	0	-
A	11	S
B	∞	-
C	3	S
D	6	S

Figure 3.28 Finding shortest path using Dijkstra's algorithm – Step 1

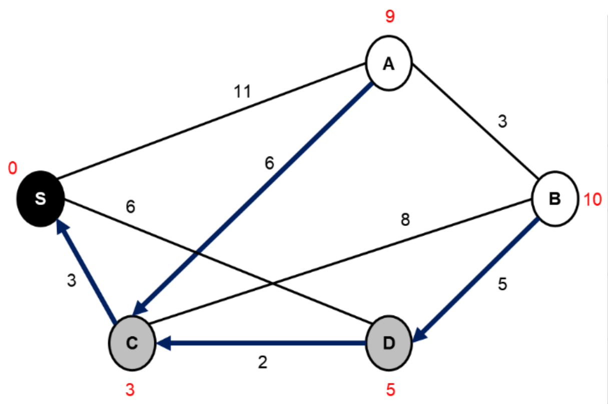
- Step 2: Next closest node is C, add to Priority Queue. Update distances and predecessors for A and D. Priority Queue = {S, C} (Figure 3.29).



From S to:	Distance	Predecessor
S	0	-
A	~~11~~ 9	C
B	~~∞~~ 11	C
C	3	S
D	~~6~~ 5	C

Figure 3.29 Finding shortest path using Dijkstra's algorithm – Step 2

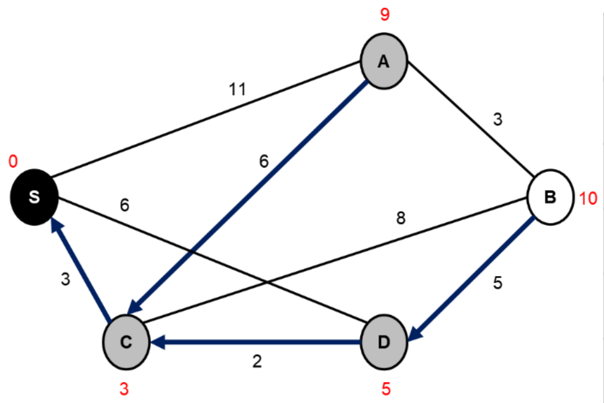
- Step 3: Next closest node is D, add to Priority Queue. Update distances and predecessor B. Priority Queue = {S, C, D} (Figure 3.30).



From S to:	Distance	Predecessor
S	0	-
A	9	C
B	~~11~~ 10	D
C	3	S
D	5	C

Figure 3.30 Finding shortest path using Dijkstra's algorithm – Step 3

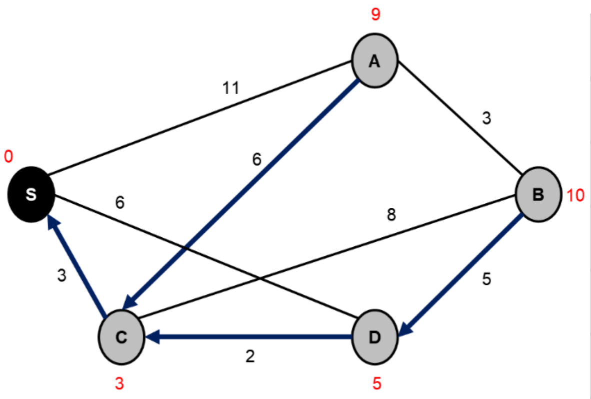
- Step 4: Next closest node is A, add to Priority Queue. Priority Queue = {S, C, D, A}. All nodes are now added (Figure 3.31).



From S to:	Distance	Predecessor
S	0	-
A	9	C
B	10	D
C	3	S
D	5	C

Figure 3.31 Finding shortest path using Dijkstra's algorithm – Step 4

- Step 5: Next step is to add the remaining node B to complete the search (Figure 3.32).



From S to:	Distance	Predecessor
S	0	-
A	9	C
B	10	D
C	3	S
D	5	C

Figure 3.32 Finding shortest path using Dijkstra's algorithm – Step 5

Once the search is complete, you can choose the goal node and find the shortest path from the table. For example, if the goal node is A, the shortest path between S and A is S-C-A with length 9. Likewise, if the goal node is B, the shortest path between S and B is S-C-D-B with a distance of 10.

Note that we can't use Dijkstra search on our 8-puzzle piece problem: it requires knowledge of the entire problem space beforehand. While the problem has a finite number of possible states (exactly 9!/2), the scale of that solution space makes the Dijkstra search not very feasible.

3.4.2 Uniform-Cost Search (UCS)

Uniform-Cost Search (UCS) algorithm is a blind search algorithm that uses the lowest cumulative cost to find a path from the origin to the destination. Essentially, the algorithm organizes nodes to be explored by their cost (with the lowest cost as the highest priority) for minimization problems or by their utility (with the highest utility as the highest priority) in the case of maximization problems. As nodes are popped from the queue, the node's children are added to the queue. If a child already exists in the priority queue, the priorities of both copies of the child are compared, and the lowest cost (highest priority) is accepted. This ensures that the path to each child is the shortest one available. We also maintain a visited list to avoid revisiting nodes that have already been popped from the queue. Algorithm 3.4 shows the steps of the UCS algorithm.

Algorithm 3.4 Uniform-Cost Search (UCS) Algorithm

```
UNIFORM-COST-SEARCH(graph,source,destination) return a route

priority_queue ← source
found ← False
seen ← source

while priority_queue is not empty and found is False do
    node ← priority_queue.pop()
    seen ← node
    node_cost ← cumulative distance from source
    if node is destination then
        route ← node.route()
        found ← True
    for child in node.expand() do
        if child in priority_queue then
            if child.priority < priority_queue[child].priority then
                priority_queue[child].priority = child.priority
        else
            priority_queue ← child
        priority_queue[child].priority ← node_cost
return route
```

UCS is a variant of Dijkstra's algorithm that is useful for large graphs as it is less time-consuming and has fewer space requirements. Whereas Dijkstra adds all nodes to the queue at the start with an infinite cost, UCS fills the priority queue gradually. For example, consider the problem of finding the shortest path between every node pair in a graph. As a graph's size and complexity grow, it quickly becomes apparent that UCS is more efficient, as it does not require knowing the entire graph beforehand. In Table 3.6, you can see the difference in processing time between Dijkstra and UCS on graphs of different sizes. These numbers were collected using the code from Listing 3.11 (available on the GitHub repo of the book) on an Intel Core i9-9900K at 3.60 GHz without multiprocessing or multithreading.

Table 3.6 UCS versus Dijkstra

Graph size= |V| + |E|	Dijkstra time	Uniform-Cost Search (UCS) time
108	0.25 s	0.14 s
628	84.61 s	58.23 s
1514	2,082.97 s	1,360.98 s

Note that running UCS on our 8-puzzle piece problem requires a distance property for each State (this defaults to 1) and overall generates decent results (around 6.2 kB of space used, 789 states explored). It is important to note that because the edge lengths are all equal, UCS cannot prioritize new nodes to explore. Thus, the solution loses the advantage of shortest path algorithms, namely, the ability to optimize for a more compact solution. In the next chapter, you'll see ways of calculating artificial "distances" between these states, ultimately generating solutions found quickly and minimizing the number of steps required.

3.4.3 Bi-directional Dijkstra Search

Bidirectional search simultaneously applies forward search and backward search. As illustrated in Figure 3.33, it runs a search forward from the source/initial state S→G and backward from the goal/final state G→S until they meet.

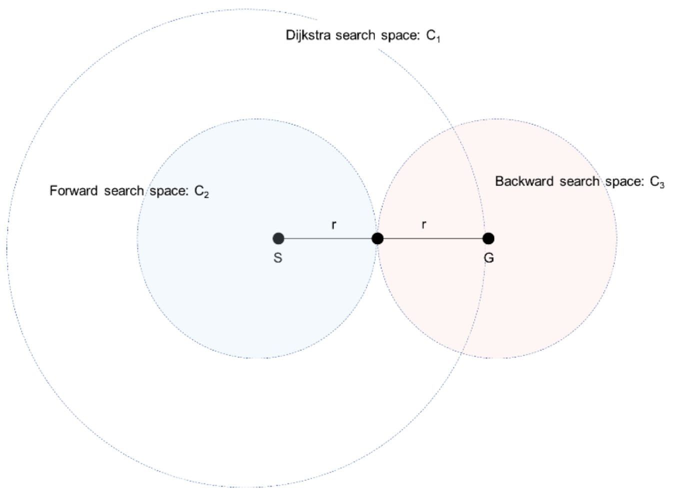

Dijkstra search space: C_1
Forward search space: C_2
Backward search space: C_3
r
r
S
G

Figure 3.33 Bidirectional Dijkstra

As shown in Figure 3.33, the Dijkstra search space is $C_1=4\pi r^2$ and the bidirectional Dijkstra search space is represented by $C_2+C_3=2\pi r^2$. This means that we reduce the search space by a factor of around two. Algorithm 3.5 shows the steps of the bidirectional Dijkstra algorithm.

Algorithm 3.5 Bidirectional Dijkstra Algorithm

```
BI-DIRECTIONAL-SEARCH(source,destination) return a route

frontier_f ← initialized with source
frontier_b ← initialized with destination
explored_f ← empty
explored_b ← empty
found ← False
collide ← False
altr_expand ← False

while frontier_f is not empty and frontier_b is not empty and not collide and not found do
    if altr_expand then
        node ← frontier_f.pop()
        add node to explored_f
        for child in node.expand() do
            if child in explored_f then continue
            if child is destination then
                route ← child.route()
                found ← True
            if child in explored_b then
                route ← child.route() + reverse(overlapped.route())
                collide ← True
            add child to frontier_f
        altr_expand ← not altr_expand
    else
        node ← frontier_b.pop()
        add node to explored_b
        for child in node.expand() do
            if child in explored_b then continue
            if child is origin then
                route ← child.route()
                found ← True
            if child in explored_f then
                route ← reverse(child.route()) + overlapped.route()
                collide ← True
            add child to frontier_b
        altr_expand ← not altr_expand
return route
```

This approach is more efficient because of the time complexities involved. For example, a BFS search with a constant branching factor b and depth d would have an overall O(b^d) space complexity. However, by running two BFS searches in opposite directions with only half the depth (d/2), the space complexity becomes O($b^{d/2}+b^{d/2}$) or simply O($b^{d/2}$), which is significantly lower.

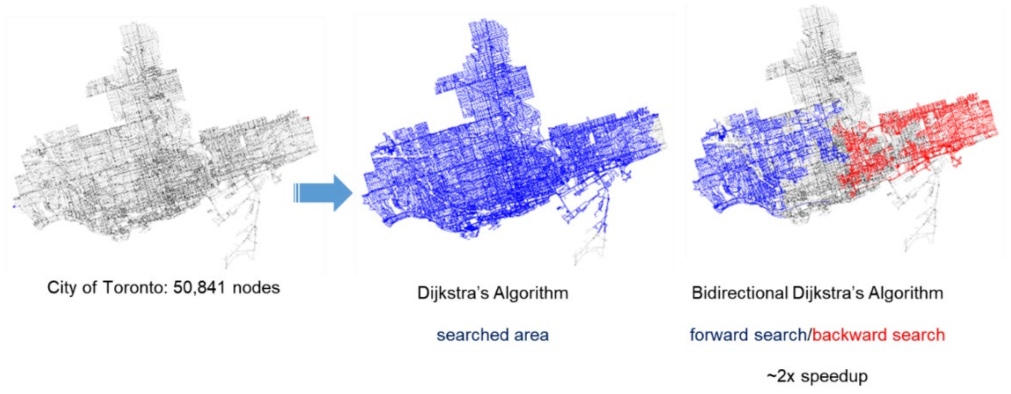

City of Toronto: 50,841 nodes
Dijkstra's Algorithm
searched area
Bidirectional Dijkstra's Algorithm
forward search/backward search
~2x speedup

Figure 3.34 Dijkstra vs Bidirectional Dijkstra. Blue represents the forward exploration, while red shows the backwards exploration.

Figure 3.34 shows the difference between Dijkstra and bidirectional Dijkstra algorithms in exploring 50,841 nodes in the City of Toronto.

3.5 Applying Blind Search to Routing Problem

Puzzle games and simple grid routing problems are nice to understand how an algorithm works. However, it's about time we show some real-world examples and outcomes from using these algorithms. For example, imagine that you are visiting the King Edward VII Equestrian statue at Queen's Park in Toronto when you suddenly remember you have a meeting at the Bahen Centre for Information Technology at the University of Toronto. If you recall, this problem was initially presented to you when we first introduced road network graphs at the beginning of this chapter. There are a couple of assumptions we make when offering this problem:

- You aren't able to open up a navigation app or call a friend for help as your phone is out of battery.
- You know your destination is somewhere in Toronto, but you have no clue where it is with reference to your starting location. In later chapters, you'll learn how knowing your destination's direction helps generate near-optimal solutions in a very short amount of time.
- Once you start using a rule for routing to your destination, you stick to that rule.

Let's take a look at how we might be able to simulate your pathfinding skills using BFS, DFS, Dijkstra, UCS, and Bidirectional Dijkstra. The code for this example is located in Listing 3.11. The following figures show the generated routes by these blind search algorithms.

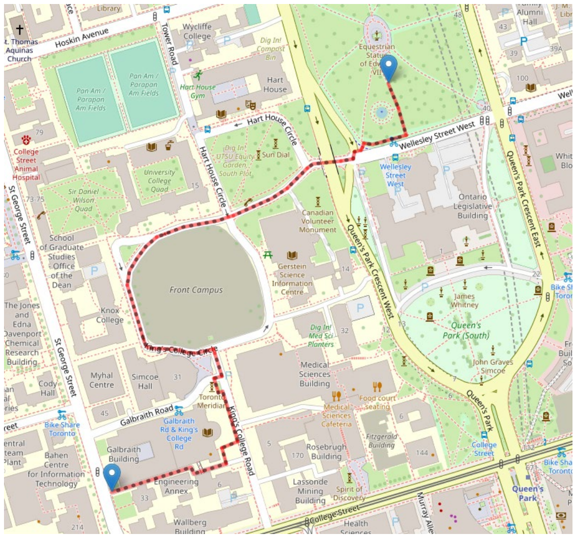


Figure 3.35 BFS routing using Python. BFS searches each layer first, before moving to the next. This works best for graphs that are not very broad, and has a solution located near the root node.

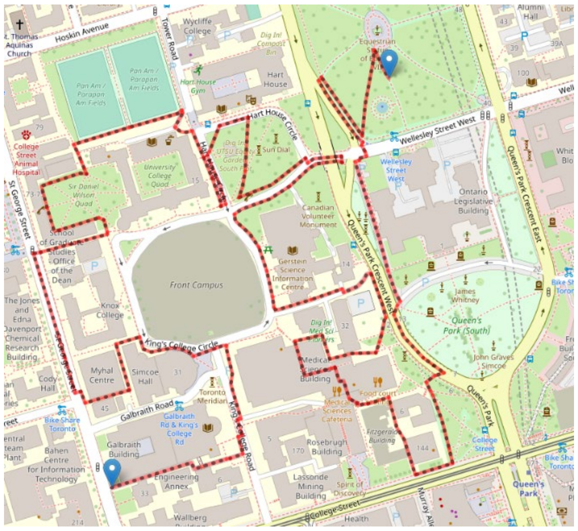


Figure 3.36 DFS routing using Python. DFS searches as deep as possible in the graph before “backtracking”. This works best when the graph is not very deep, and solutions are located further away from the root node.

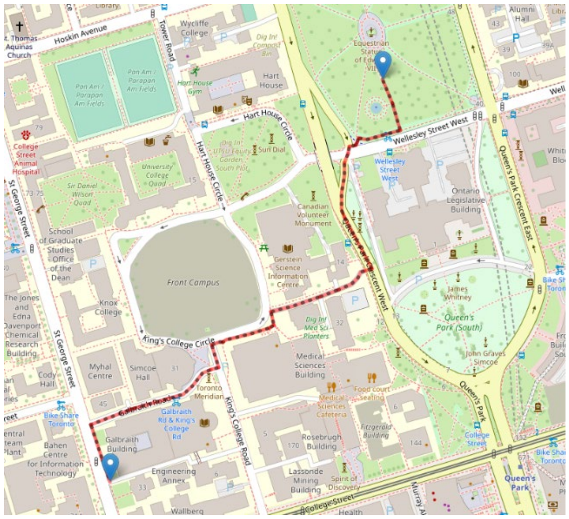


Figure 3.37 Dijkstra’s, UCS, and Bidirectional Dijkstra’s routing. All three of these algorithms will produce the same solution (optimal routing), but handle memory usage and node exploration differently.

It is worth noting that *dijkstra_path* function in *networkx* uses Dijkstra’s method to compute the shortest weighted path between two nodes in a graph. Our *optimization_algorithms_tools* package also provide implementation for different graph search algorithm such as BFS, DFS, Dijkstra, UCS, Bidirectional_Dijkstra. The implementation of Dijkstra's algorithm in *optimization_algorithms_tools* has been modified to work with our OSM data. This is because graphs generated from maps will naturally have self-loops and parallel edges. Parallel edges may result in a route that is not the shortest available, as the route length depends heavily on which parallel edge was chosen when a particular path was generated. In Figure 3.38, the shortest path from 0 to 3 may be returned as having a length of 7 if the first edge connecting 0 and 1 is chosen when calculating that path, versus a length of 3 when selecting the second edge.

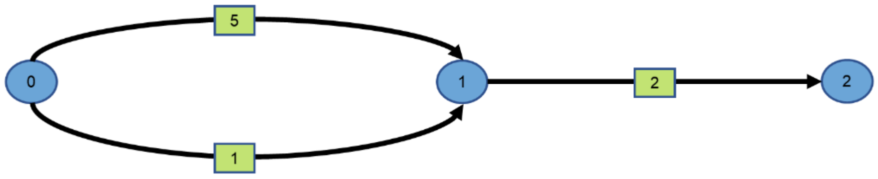


Figure 3.38 Parallel edges may be problematic as finding the shortest path depends on which parallel edge was selected during graph exploration.

Self-loops also cause trouble for the original Dijkstra algorithm. If a graph contains a self-loop, it may be the case that the shortest path to a node comes from itself. At that point, we would be unable to generate a route.

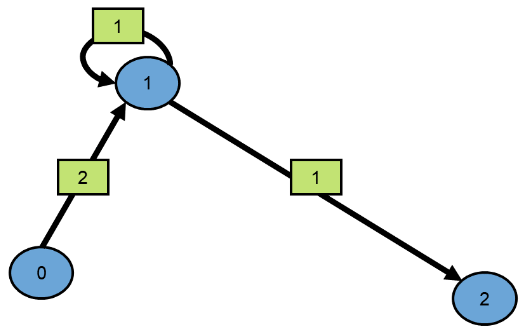


Figure 3.39 Self-loops may disrupt the chain of “parent-child nodes”, which prevents retracing the route after a solution has been found.

These two issues are generally easy but non-trivial to avoid. For parallel edges, we select the edge with the lowest weight (shortest length) and discard any other parallel edge. With self-loops, we can ignore the loop entirely as negative-weight loops do not exist in most routing problems (a road cannot have a negative length), and positive-weight loops cannot be part of the shortest path. Additionally, the version of Dijkstra’s used in this book terminates upon finding the target node, as opposed to the traditional implementation of Dijkstra’s, which ends only when the shortest path from the root node to all other nodes is found. Table 3.7 compares BFS, DFS, Dijkstra, and UCS with regards to path length, process time, space required, and the number of explored nodes.

Table 3.7 Comparison between BFS, DFS, Dijkstra and UCS where *b* is the branching factor, *m* is the maximum depth of search tree, *d* is the shallowest graph depth, E is number of edges and V is number of vertices.

Algorithm	Cost (m)	Process time (s)	Space (bytes)	Explored nodes	Worst-case time	Worst-case space	Optimality
BFS	955.962	0.015625	1152	278	$O(b^d$ $O(b^d)$	$O(b^d)$	No
DFS	3347.482	0.015625	1152	153	$O(b^m)$	$O(bm)$	No
Dijkstra's	806.892	0.0625	3752	393	$O(\|E\| + \|V\| \log \|V\|)$	$O(\|V\|)$	Yes
UCS		0.03125	592	393	$O((b + \|E\|) * d)$	$O(b^d)$	Yes
Bidirectional Dijkstra's		0.046875	3752	282	$O(b^{d/2})$	$O(b^{d/2})$	Yes

As you can see from the above results, Dijkstra's, UCS, and Bidirectional Dijkstra's algorithms produce optimal results, with varying degrees of time and space cost. While both BFS and DFS find feasible solutions in the shortest time, the solutions delivered are not optimal and, in the case of DFS, not even plausible. On the other hand, DFS requires knowing the entire graph beforehand, which is costly and sometimes not very practical. Much of selecting an appropriate search algorithm for a specific problem involves determining the ideal balance between processing time and space requirements. In later chapters, we'll look at algorithms that produce near-optimal solutions, often used when optimal solutions are either impossible or impractical to find. Note that all the above solutions are feasible; they all produce a valid (if sometimes convoluted) path from point A to point B.

In the next chapter, we will show how search can be optimized if we utilize domain-specific knowledge instead of search blindly. We'll dive right into informed search methods showing how to use these algorithms to solve minimum spanning tree and shortest path problems.

3.6 Exercises

1. MCQs and T/F: 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 towards a particular value or infinity, usually in terms of simpler functions. What is the big-O of this expression: nlog(n)+log(2n)

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

1.2. Which blind search algorithm implements 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 mentioned

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

a. True
b. False

1.4. For very large workspace where the goal is deep within the workspace, the number of nodes could expand exponentially and 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 weight of edges in a graph are all equal

a. True

b. False

1.8. In Dijkstra’s algorithm, the priority queue is filled gradually

a. True

b. False

1.9. When is breadth-first search is optimal?

a. When there is less number of nodes

b. When all step costs are equal

c. When all step costs are unequal

d. None of the mentioned

1.10 A blind search algorithm that 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 mentioned

1.11 The name of O(nlogn) is

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 mentioned

2. Consider the following simplified map shown in Figure 3.40, where edges are labeled with actual distances between the cities. State the path to go from city A to city M produced by BFS and the path produced by DFS.

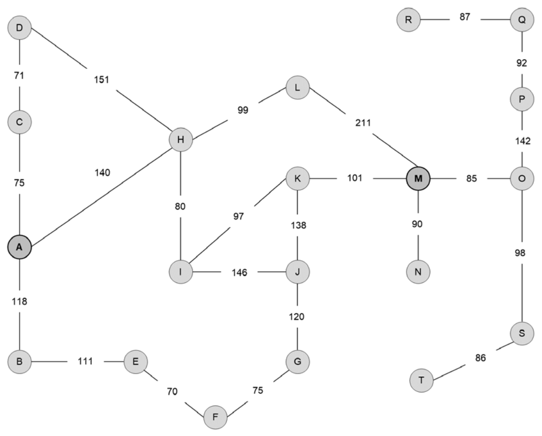


Figure 3.40 Simplified map.

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

a. 10n+nlog(n)

b. 4+n/5

c. n^5-20n^3+170n+208

d. n+10log(n)

4. Consider the search space below, where S is the start node and G1 and G2 are goal nodes. Arcs 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 depth-first search (DFS) and breadth-first search (BFS) strategies:

a. indicate which goal state is reached first (if any) and

b. list in order, all the states that are popped off the OPEN list.

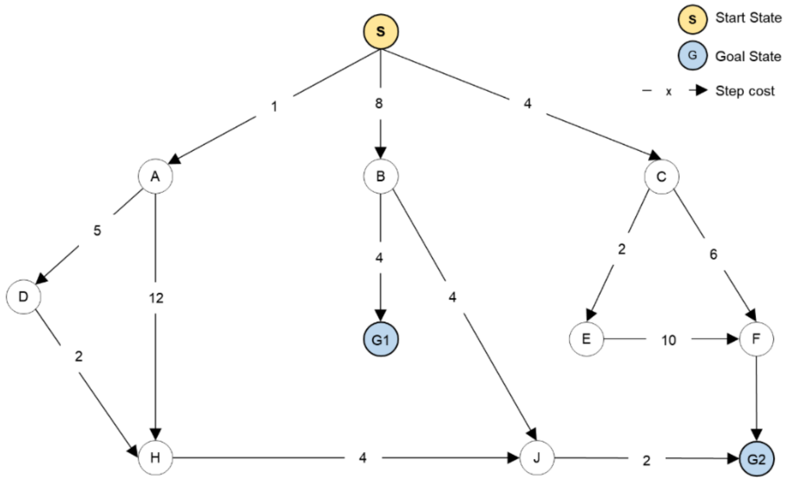
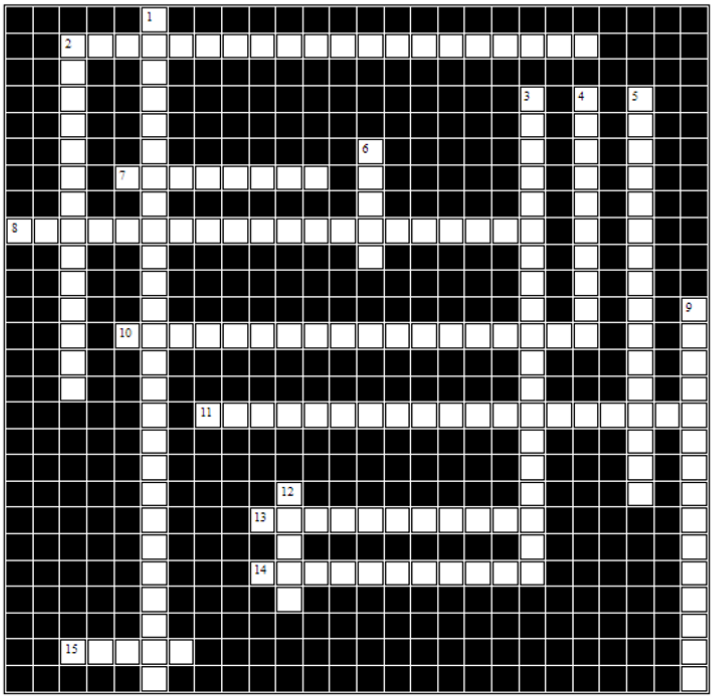


Figure 3.41.A graph search exercise

5. Solve the following 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 of the root, then 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. 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 and dashes MUST be used if the answer consists of two or more words.

3.7 Summary

- Conventional graph search algorithms (blind and informed search algorithms) are deterministic search algorithms that explore a graph either for general discovery or for explicit search.
- Graphs are non-linear data structures that consist of 3 sets: vertices/nodes, edges and a set representing relations between vertices and edges.
- Blind/uninformed search is a search approach where no information about the search space is used.
- Breadth-first Search (BFS) is a graph traversal algorithm that examines all the nodes in a search tree on one level before considering any of the nodes on the next level.
- Depth-first Search (DFS) is a graph traversal algorithm that first explores nodes going through one adjacent of the root, then next adjacent until it finds a solution or until it reaches a dead-end.
- Depth-limited Search (DLS) is a constrained DFS with a predetermined depth limit to prevent exploring of paths that are too long in search tree with infinite depth.
- Iterative Deepening Search (IDS) or Iterative Deepening Depth First Search (IDDFS) combines DFS's space efficiency and BFS's fast search by incrementing the depth limit until the goal is reached.
- Dijkstra's algorithm solves the single-source shortest path problem for a weighted graph with non-negative edge costs.
- Uniform-cost Search (UCS) is a variant of Dijkstra's algorithm that uses the lowest cumulative cost to find a path from the source to the destination. It is equivalent to the BFS algorithm if the path cost of all edges is the same.
- Bidirectional Search (BS) is a combination of forward and backward search. It searches forward from the start and backward from the goal simultaneously.
- Selecting a search algorithm involves determining the target balance between time complexity, space complexity, prior knowledge about the search space, among other factors.
- As you progress to Chapter 4, you will start to learn about informed search algorithms where additional information or heuristic rules are used during the search.

A Search and Optimization Libraries in Python

This appendix covers

- **Setting up the Python environment**
- **Mathematical programming solvers**
- **Graph and mapping Libraries**
- **Metaheuristics Optimization Libraries**
- **Machine Learning Libraries**

A.1 Setting up the Python environment

This book assumes that you already have Python 3.6 or newer installed on your system. For installation instructions specific to your operating system, see this Beginner's Guide. For Windows, you can follow these steps to install Python:

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

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

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

Install venv and create a python virtual environment:

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

Make sure that you replace python3.8 with the version of Python you are using. You can now access your virtual environment using the following command:

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

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

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

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

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

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

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

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

A.1.1 Using a Python distribution

Python distribution, such as Anaconda or Miniconda come with a package manager called `conda` that allows you to install a wide range of Python packages and manage different Python environments.

Install `conda` for your OS using the guide found here. Conda environments are used to manage multiple installations of different versions of Python packages and their dependencies. You can create a `conda` environment with this command:

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

To access the newly-created environment:

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

This command allows you to switch or move between environments.

A.1.2 Installing Jupyter Notebook and JupyterLab

Jupyter is a multi-language, open-source web-based platform for interactive programming. The word "Jupyter" is a loose acronym meaning Julia, Python, and R. All of the code in this book is stored in Jupyter notebooks (.ipynb files) can be opened and edited using jupyterlab or Jupyter notebook. Jupyter notebook feels more standalone but JupyterLab feels more like an IDE. You can install JupyterLab using `pip` as follows:

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

Or using conda as follows:

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

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

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

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

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

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

A.1.3 Cloning book repository

You can clone the book code repository as follows:

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

Many of the operations in this book are long and burdensome to code from scratch. Oftentimes, they are highly standardized and can benefit from having a helper function take care of the various intricacies. `optimization_algorithms_tools` is a python package developed for this purpose. You can use these supporting tools locally without installing this package. In this case, you will need to download `optimization_algorithms_tools` in a local folder and add this folder to the system path in case of using Jupyter notebook or Jupyter lab as follows:

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

In case of Colab, you can mount your Google Drive with:

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

Copy `optimization_algorithms_tools` folder to your Google Drive.

This package is also available on Python Package Index (PyPI) repository here: https://pypi.org/project/optimization-algorithms-tools/0.0.1/. You can it install as following:

```
$pip install optimization_algorithms_tools
```

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

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

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

A.2 Mathematical Programming Solvers

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

Let's consider the following production planning example from [C. Guerte el al, Applications of Optimization, Chapter 8: Planning problems, 2000]. A small joinery makes two different sizes of boxwood chess sets. The small set requires 3 hours of machining on a lathe, and the large set requires 2 hours. There are four lathes with skilled operators who each work a 40-hour week, so we have 160 lathe-hours per week. The small chess set requires 1 kg of boxwood, and the large set requires 3 kg. Unfortunately, boxwood is scarce and only 200 kg per week can be obtained. When sold, each of the large chess sets yields a profit of $12, and one of the small chess set has a profit of $5. The problem is to decide how many sets of each kind should be made each week so as to maximize profit. Let's assume that x_1 and x_2 are decision variables that represent the number of small and large chess sets respectively to make.

The total profit is the sum of the individual profits from making and selling the x_1 small sets and the x_2 large sets, i.e. Profit = $5x_1 + 12x_2$. However, this profit is subject to the following constraints:

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

This linear programming problem can be summarized as follows:

find x_1 and x_2 that maximize $5x_1 + 12x_2$

subject to

$3x_1 + 2x_2 <= 160$ (machining time constraint)

$x_1 + 3x_2 <= 200$ (weight constraint)

x_1 and $x_2 >= 0$ (non-negativity constraints)

Let's now see how to solve this linear programming using different solvers.

A.2.1 SciPy

SciPy is an open-source scientific computing Python library that provides tools for optimization, linear algebra, and statistics. *SciPy optimize* includes solvers for nonlinear problems (with support for both local and global optimization algorithms), linear programing, constrained and nonlinear least-squares, root finding, and curve fitting. To use SciPy, you will need to install it along with its dependencies. You can install SciPy using the pip package manager or by using a Python distribution, such as Anaconda or Miniconda, which come with SciPy and other scientific libraries pre-installed.

```
$pip install scipy
```

Listing A.1 shows the steps to solve the car manufacturing problem using SciPy. The code defines the coefficient vector c, left hand side (lhs) and right hand side (rhs) of the constraint equations.

Listing A.1 Solving Chess Set Problem using SciPy

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

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

lhs_constraints=([3,2], #B
                 [1,3]) #C

rhs_constraints=([160, #D
                  200]) #E

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

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

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

#A Declare coefficients of the objective function/profit A profit maximization is converted into minimization problem as per SciPy requirement
#B Left hand side of machining time constraint
#C Left hand side of weight constraint
#D Right hand side of machining time constraint
#E Right hand side of weight constraint
#F Bounds of the decision variables
#G Solve linear programming pro
#H Print the solution

Running this code give the following results:

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

The used *linprog()* function returns a data structure with several attributes such as x: the current solution vector, fun: the current value of the objective function, success: True when the algorithm has completed successfully. *SciPy* cannot handle maximization problems. The problem must be converted to minimization problem. Moreover, constraints using the greater-than-or-equal-to sign cannot be defined directly. Less-than-or-equal-to must be used instead.

A.2.2 PuLP

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

```
$pip install pulp
```

The following code (part of Listing A.1) shows how to use PuLP to solve the chess set problem.

Listing A.1 Solving Production Planning Problem using PuLP

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

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

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

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

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

model.solve() #E

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

#A Define the model
#B Ddefine the decision variables
#C Aadd constraints
#D Set the profit as the objective function
#E Solve the optimization problem
#F Print the solution

PuLP implements several algorithms for solving linear programming (LP) and mixed-integer linear programming (MILP) problems. Examples of these algorithms include COIN-OR (Computational Infrastructure for Operations Research), CLP (Coin-or Linear Programming), CBC (Coin-or Branch-and-Cut), CPLEX (Cplex for short), GLPK (GNU Linear Programming Kit), SCIP (Solving Constraint Integer Programs), HiGHS (Highly Scalable Global Solver), Gurobi LP/MIP solver, Mosek Optimizer and XPRESS LP solver.

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

- OR-Tools: is an open-source software suite for optimization and constraint programming developed by Google. It includes a variety of algorithms and tools for solving problems in areas such as operations research, transportation, scheduling, and logistics. OR-Tools can be used to model and solve linear and integer programming problems, as well as constraint programming problems. Examples of OR-Tools solvers include GLOP (Google Linear Programming), CBC (Coin-Or Branch and Cut), CP-SAT (Constraint Programming - SATisfiability) solver, Max Flow and Min Cost Flow solvers, and the Shortest Path solver, BOP (Binary Optimization Problem) solver. It is written in C++ and includes interfaces for several programming languages, including Python, C#, and Java. See Section A.4 for more details and an example.
- Gurobi: is a commercial optimization software that offers state-of-the-art solvers for linear programming, quadratic programming, and mixed integer programming. It has a Python interface that can be used to define and solve optimization problems.
- PYTHON-MIP: is a Python library for solving mixed-integer programming problems. It is built on top of the open-source optimization library CBC and allows users to express optimization models in a high-level, mathematical programming language.
- Pyomo: is an open-source optimization modeling language that can be used to define and solve mathematical optimization problems in Python. It supports a wide range of optimization solvers, including linear programming, mixed integer programming, and nonlinear optimization.
- GEKKO: is a Python package for machine learning and optimization of mixed-integer and differential algebraic equations.
- CVXPY: is an open source Python-embedded modeling language for convex optimization problems. It lets you express your problem in a natural way that follows the math, rather than in the restrictive standard form required by solvers.
- PyMathProg: is a mathematical programming environment for Python that enables modelling, solving, analyzing, modifying and manipulating linear programming problems.

- Optlang: is a Python library for modeling and solving mathematical optimization problems. It provides a common interface to a series of optimization tools, so different solver backends can be changed in a transparent way. It is compatible with most of the popular optimization solvers like Gurobi, Cplex, and Ipopt (Interior Point OPTimizer).
- Python interface to conic optimization solvers (PICOS): is a Python library for modeling and solving optimization problems. It can handle complex problems with multiple objectives, and it supports both local and global optimization methods. PICOS has interfaces to different solvers such as Gurobi, CPLEX, SCS (Splitting Conic Solver), ECOS (Embedded Cone Solver), and MOSEK.
- CyLP: is is a Python interface to COIN-OR's Linear and mixed-integer program solvers (CLP, CBC, and CGL). COIN-OR (COmputational INfrastructure for Operations Research) is a collection of open-source software packages for operations research and computational optimization. It includes libraries for linear and integer programming, constraint programming, and other optimization techniques.
- SymPy: is a Python library for symbolic mathematics. It can be used to deal to solve equations, handling combinatorics, plotting in 2d/3d, work on polynomials, calculus, discrete math, matrices, geometry, parsing, physics, statistics and cryptography.
- Other libraries: include but are not limited to, MOSEK, CVXOPT, DOcplex, DRAKE, Pyscipopt, PyOptim, PyMathProg and NLPy.

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

A.3 Graph and Mapping Libraries

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

A.3.1 networkx

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

```
$pip install networkx
```

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

Listing A.2 Solving TSP using networkx

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

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

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

pos[0] = (0.5, 0.5) #B

H = G.copy() #C

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

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

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

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

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

#A Create a random geometric graph with 20 nodes
#B Set (0,0) as the home city/depot
#C Create an independent shallow copy of the graph and attributes
#D Calculating the distances between the nodes as edge's weight.
#E Solve tsp using Christofides algorithm
#F Highlight the closest edges on each node only
#G Draw the route
#H Print the route

Figure A.1 shows the solution of this TSP.

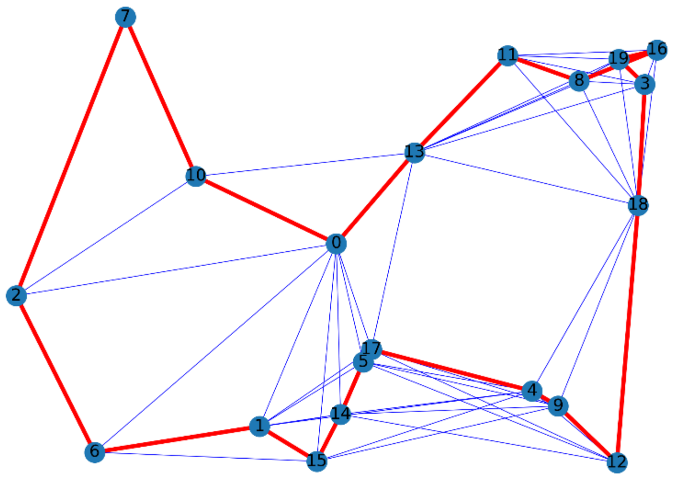


Figure A.1 Solving TSP using Christofides algorithm implemented in networkx. The found route is: [0, 10, 7, 2, 6, 1, 15, 14, 5, 17, 4, 9, 12, 18, 3, 19, 16, 8, 11, 13, 0]

networkx supports a variety of graph search algorithms and enables performing network analyses using packages within the geospatial Python ecosystem.

A.3.2 osmnx

osmnx is a Python library developed to ease the process of retrieving and manipulating the data from OpenStreetMap (OSM). It offers the ability to download the data (filtered) from OSM and returns the network as networkx graph data structure. It is a free and open-source geographic data of the world. You can install OSMnx with conda:

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

osmnx can be used to convert a text descriptor of a place into a networkx graph. Let’s use the Times Square in New York City as an example.

Listing A.2 Creating a networkx graph for Times Square using osmnx

```
import osmnx as ox

place_name = "Times Square, NY" #A

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

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

#A Name of the place or point of interest
#B networkx graph of the named place
#C Plot the graph

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

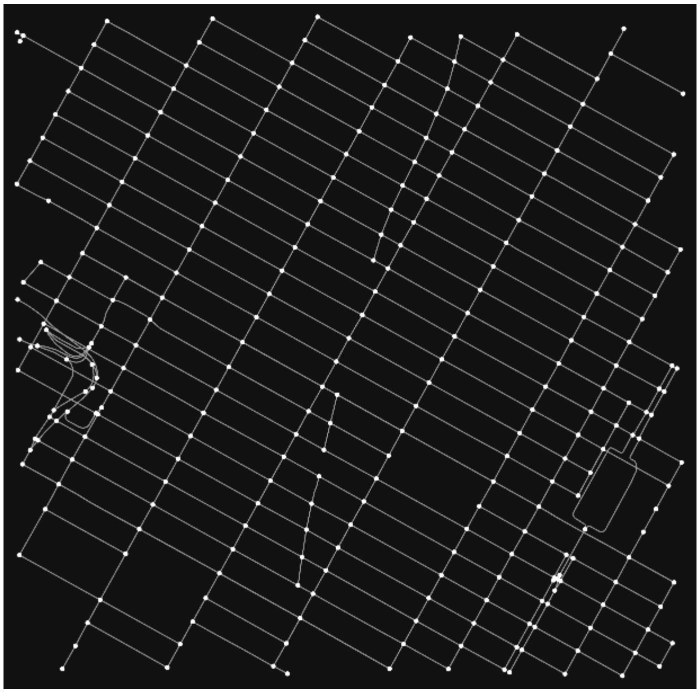


Figure A.2 Times Square Graph wit drivable streets

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

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

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

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

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

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

You can also retrieve the street types for the graph:

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

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

```
secondary                      236
residential                    120
primary                         83
unclassified                    16
motorway_link                   12
tertiary                        10
motorway                         7
living_street                    3
[unclassified, residential]      1
[motorway_link, primary]         1
trunk                            1
```

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

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

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

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

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

#A Save street network as PNG
#B Save street network as SVG
#C Save graph to disk as .osm xml file
#D Save street network as GeoPackage file for GIS
#E Save street network as GraphML file for OSMnx or networkx or gephi
#F Save graph as a shapefile

A.3.3 GeoPandas

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

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

GeoPandas can handle different geospatial data formats such as Shapefiles (.shp), CSV (Comma Separated Values), GeoJSON, ESRI JSON, GeoPackage (.gpkg), GML, GPX (GPS eXchange Format) and KML (Keyhole Markup Language). For example, let’s assume we want to read Ontario’s health region data based on a Shapefile than can be downloaed from Ontario data catalogue. A Shapefile is a popular geospatial data format for storing vector data (such as points, lines, and polygons). It is a widely-used format for storing GIS data and is supported by many GIS software packages, including ArcGIS and QGIS. A Shapefile is actually a collection of several files with different extensions, including:

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

Listing A.2 shows how to read this geospatial data downlaoded from https://data.ontario.ca/dataset/ontario-s-health-region-geographic-data and stored in a local folder.

Listing A.2 Reading geospatial data using GeoPandas

```
import geopandas as gpd
ontario = gpd.read_file(r"../Appendix B/data/OntarioHealth/Ontario_Health_Regions.shp")
```

Complete listing is available on the book GitHub repo.

A.3.4 contextily

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

Listing A.2 Rendering data using contextily

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

Contextily supports several different sources for basemaps. Some of the most commonly used sources are:

- OpenStreetMap (OSM): is the default source for Contextily. It is a free and open-source map service that provides a variety of different styles, including the default "Mapnik" style and other styles such as "Humanitarian" and "Cycle".
- Stamen: provides a variety of different map styles, including "Toner", "Terrain", and "Watercolor".
- Mapbox: provides a variety of different map styles, including "Streets", "Outdoors", and "Satellite". It requires an API key to use.

- MapQuest: provides a variety of different map styles, including "OSM" and "Aerial". It requires an API key to use.
- Here: provides a variety of different map styles, including "Normal Day" and "Normal Night". It requires an API key to use.
- Google Maps: provides a variety of different map styles, including "Roadmap", "Satellite", and "Terrain". It requires an API key to use.

A.3.5 folium

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

Listing A.2 Visualizing geospatial data on a map using folium

```
#!pip install folium
import folium

ontario = ontario.to_crs(epsg=4326) #A

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

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

m #D
```

#A Transform geometries to a new coordinate reference system (CRS)
#B Set starting location, initial zoom, and base layer source
#C Simplify each region's polygon as intricate details are unnecessary
#D Render the map

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

A.3.6 Pyrosm

Pyrosm is another Python library for reading OpenStreetMap from Protocolbuffer Binary Format -files (*.osm.pbf). It can be used to download and read OpenStreetMap data, extract features such as roads, buildings, and points of interest, and analyze and visualize the data. The main difference between pyrosm and OSMnx is that OSMnx reads the data using an OverPass API, whereas pyrosm reads the data from local OSM data dumps that are downloaded from the Protocolbuffer Binary Format -files (*.osm.pbf) data providers

(Geofabrik, BBBike) and convert it into Geopandas GeoDataFrames. This makes it possible to parse OSM data faster and make it more feasible to extract data covering large regions. Listing A.2 shows how to use Pyrosm to retrieve OpenStreetMap data about specific region of interest.

Listing A.2 Retrieving OpenStreetData using Pyrosm

```
#! pip install pyrosm
import pyrosm
place_name = 'Toronto' #A
file_path = pyrosm.get_data(place_name) #B
osm = pyrosm.OSM(file_path) #C
```

#A Specify the place of interest
#B Downloaded data from data providers such as Geofabrik, BBBike and store it into a local file
#C Initialize the OSM object that parses the generated .osm.pbf local file

Listing A.2_Graph_libraries.ipynb notebook available on the book's GitHub repo provides more examples about how to use Pyrosm.

A.3.7 Other libraries and tools

The following list is a non-exhaustive list of other useful libraries and tools to work on geospatial data, graphs and networks.

- Pandana: is a Python library for network analysis that uses contraction hierarchies to calculate super-fast travel accessibility metrics and shortest paths.
- GeoPy: is a Python client for several popular geocoding web services.
- Graphviz: is a library for creating visualizations of graphs and tree structures in Python. It provides tools for defining the structure of a graph, as well as for rendering the graph in various formats, such as PNG, PDF, and SVG. Graphviz is a useful tool for visualizing algorithms that operate on graphs, such as graph search algorithms and graph traversal algorithms.
- Gephi: is a tool for visualizing and analyzing graphs and networks. It provides a graphical user interface for defining and customizing the appearance of graphs and diagrams, as well as for visualizing algorithms and data structures. Gephi can be used to visualize algorithms that operate on graph data, such as graph search algorithms and shortest path algorithms.
- Cytoscape: is an open source software platform for visualizing complex networks and integrating these with any type of attribute data.
- ipyleaflet: is an interactive widgets library that is based on ipywidgets. ipywidgets, also known as jupyter-widgets or simply widgets, are interactive HTML widgets for Jupyter notebooks and the IPython kernel. Ipyleaflet brings mapping capabilities to the notebook and JupyterLab.
- hvplot: if you want to get going through your analysis with geopandas and dataframes and all that. You should be aware of the significance of working with vanilla GeoPandas, and that `osmnx` supports that and yields two dataframes: one for all your nodes and one for all the edges.

- mplleaflet: is another `leaflet`-based library, but it plays really nicely with `matplotlib`.
- Cartopy: Cartopy is a library for creating maps and geospatial plots in Python.
- Geoplotlib: Geoplotlib is a library for creating maps and visualizations in Python. It provides tools for styling and customizing map elements, as well as for overlaying data on top of maps. Geoplotlib can be used to create static or interactive maps, and supports a variety of map projections and coordinate systems.
- Shapely: is an open source Python library for performing geometric operation on objects in the Cartesian plane.
- deck.gl: is an open-source JavaScript library for WebGL-powered large dataset visualization.
- kepler.gl: is a powerful open source geospatial analysis tool for large-scale data sets.

A.4 Metaheuristics Optimization Libraries

There are several metaheuristics optimization libraries in Python that provide implementations of different metaheuristic optimization algorithms. Here are examples of some commonly used libraries.

A.4.1 PySwarms

PySwarms is a library for implementing swarm intelligence algorithms in Python. It provides tools for defining, training, and evaluating swarm intelligence models, as well as for visualizing the optimization process. PySwarms supports a variety of swarm intelligence algorithms, including particle swarm optimization (PSO) and ant colony optimization (ACO). A.3 shows the steps to solve a function optimization problem using POS implemented in PySwarms.

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

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

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

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

plot_cost_history(optimizer.cost_history) #E
plt.show()

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

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

#A To view animation in a Jupyter notebook
#B Set-up POS as an optimizer with 50 particles and predfined parameters
#C Solve the function optimization problem using PSO
#D Set-up Sphere unimodal function to be optimized using PSO and set up the number of iteration
#E Plot the cost
#F Plot the sphere function's mesh for better plots
#G Adjust figure limits
#H Generate animation for the solution history on a contour
#I Rendering the animation

A.4.2 Sckit-opt

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

Listing A.3 Solving function optimization using simulated annealing in scikit-opt

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

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

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

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

#A Define a multimodal function
#B Solve using simulated annealing (SA)
#C Print the result

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

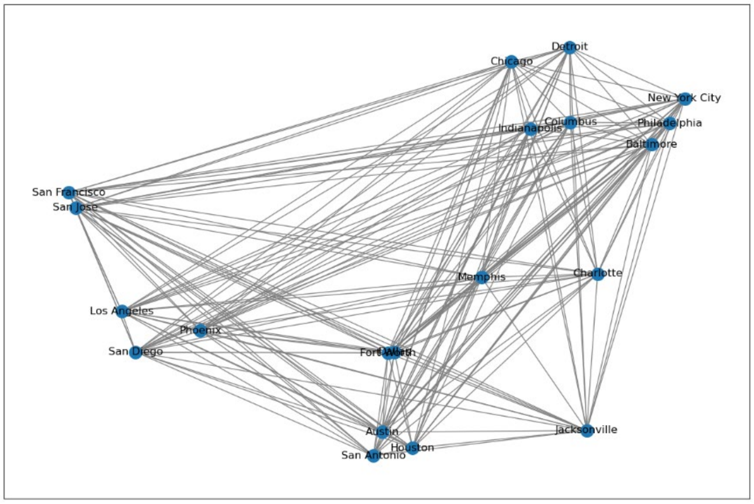


Figure A.3 Travelling Salesman Problem (TSP) for 20 largest US cities.

Listing A.3 shows the steps to solve this problem using scikit-opt.

Listing A.3 Solving TSP using scikit-opt

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

num_points = len(city_names) #A
points_coordinate = city_names #A
pairwise_distances = distances #A

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

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

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

#A Define the TSP problem
#B The objective function. input routine, return total distance.
#C Solving the problem using particle swarm optimization (PSO)
#D Print the solution

A.4.3 networkx

networkx introduced in the previous section provides approximations of graph properties and heuristic methods for optimization. Example of these heuristics algorithms is simulated annealing. Listing A.3 shows the steps to solve TSP using simulated annealing implemented in networkx.

Listing A.3 Solving TSP using simulated annealing networkx

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

G=nx.Graph() #A

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

pos = nx.spring_layout(G) #C

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

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

#A Create a graph
#B Add weighted edges to the graph and remove selfloop edges
#C Define pos is a dictionary of positions using using Fruchterman-Reingold force-directed algorithm
#D Solve TSP using simulated annealing
#E Print the route and the cost

A.4.4 Distributed Evolutionary Algorithms in Python (DEAP)

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

Listing A.3 Solving TSP using DEAP

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

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

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

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

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

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

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

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

#A Create a fitness function that minimizes the total distance
#B Create the genetic operator functions
#C Calculate route length
#D Set evaluation function
#E Set an ordered crossover
#F Set shuffle mutation with probability 0.05
#G Select the best individual among 3 randomly chosen individuals
#H Set population size
#I Set hall of fame to select the best individual that ever lived in the population during the evolution
#J Solve the problem using simple evolutionary algorithm
#K Print solution

DEAP includes several built-in algorithms such as Genetic Algorithm (GA), Evolutionary Strategy (ES), Genetic Programming (GA), Estimation of Distribution Algorithms (EDA) and Particle Swarm Optimization (PSO).

A.4.5 OR-Tools

As previously mentioned, OR-Tools (Operations Research Tools) is an open-source library for optimization and constraint programming developed by Google. Listing A.3 shows the steps to solve TSP using Tabu Search implemented in OR-Tools.

Listing A.3 Solving TSP using OR-Tools

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

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

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

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

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

transit_callback_index = routing.RegisterTransitCallback(distance_callback)

routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)

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

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

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

#A Convert float array into integer array for OR_Tools
#B Define problem data
#C Define a constraint programming solver
#D Get distance between the cities
#E Set up tabu search as a local search metaherutsic.
#F Print the solution

OR-Tools library also provides some metaheuristics implemented but not as many as dedicated metaheuristics frameworks like DEAP. Examples include Simulated Annealing (SA), Tabu Search (TS) and Guided Local Search (GLS).

A.4.6 Other Libraries

The following list is a non-exhaustive list of other useful libraries and tools to solve optimization problems using metaheuristics.

- simanneal is an open-source Python module for simulated annealing. Listing A.3 shows the steps to solve TSP using simulated annealing implemented in simanneal.
- Non-dominated Sorting Genetic Algorithm (NSGA-II): is a solid multi-objective algorithm, widely used in many real-world applications. The algorithm is designed to find a set of solutions, called the Pareto front, which represents the trade-off between multiple conflicting objectives. NSGA-II implementations are available in pymoo and DEAP.
- Python Genetic Algorithms & Differential Evolution (PyGAD): is a library for implementing genetic algorithms and differential evolution in Python. It provides tools for defining, training, and evaluating genetic algorithm models, as well as for visualizing the optimization process. PyGAD supports a variety of genetic algorithm techniques, including selection, crossover, and mutation.
- Library for Evolutionary Algorithms in Python (LEAP): is a general purpose Evolutionary algorithms (EAs) package that is simple and easy-to-use. It provides a high-level abstraction for defining and running EAs.
- Pyevolve: is a Python library for implementing and running genetic algorithms that provides a simple and flexible API for defining and running genetic algorithms.
- Genetic Algorithms made Easy (EasyGA): is another Python library for genetic algorithms with several built-in genetic operators such as selection, crossover, and mutation. It's worth noting that EasyGA and Pyevolve are simple libraries with less functionalities and pre-defined problems than other libraries such as DEAP and Pymoo.
- MEAPLY: is a Python library for population meta-heuristic algorithms.
- swarmlib: implements several swarm optimization algorithms and visualizes their (intermediate) solutions.
- Hive: is a swarm-based optimization algorithm based on the intelligent foraging behavior of honey bees. Hive implements the so-called Artificial Bee Colony (ABC)
- Pants: is a Python3 implementation of the Ant Colony Optimization Meta-Heuristics.

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

A.5 Machine Learning Libraries

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

A.5.1 Node2vec

node2vec is an algorithmic framework for learning low-dimensional representations of nodes in a graph. Given any graph, it can learn continuous feature representations for the nodes, which can then be used for various downstream machine learning tasks. To install *node2vec*, use the following command:

```
$ pip install node2vec
```

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

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

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

Listing A.4 node2vec Example

```
import networkx as nx
from node2vec import Node2Vec

G = nx.karate_club_graph() #A

node2vec = Node2Vec(G, dimensions=64, walk_length=30, num_walks=200, workers=4) #B

model = node2vec.fit(window=10, min_count=1, batch_words=4) #C

representations_all = model.wv.vectors #D

representations_specific = model.wv['1'] #E

print(representations_specific) #F
```

#A create a sample graph
#B create an instance of the Node2Vec class
#C learn the representations
#D get the representations of all nodes
#E get the representations of a specific node
#F print the representations of a specific node

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

```
from sklearn.manifold import TSNE #A
import matplotlib.pyplot as plt

tsne = TSNE(n_components=2, learning_rate='auto', init='random', perplexity=3) #B
reduced_representations = tsne.fit_transform(representations_all) #B

plt.scatter(reduced_representations[:, 0], reduced_representations[:, 1]) #C
plt.show() #C
```

#A import the requried libraries
#B perform t-SNE dimensionality reduction
#C plot the nodes

Running this code give the following visualization shown in Figure A.4.

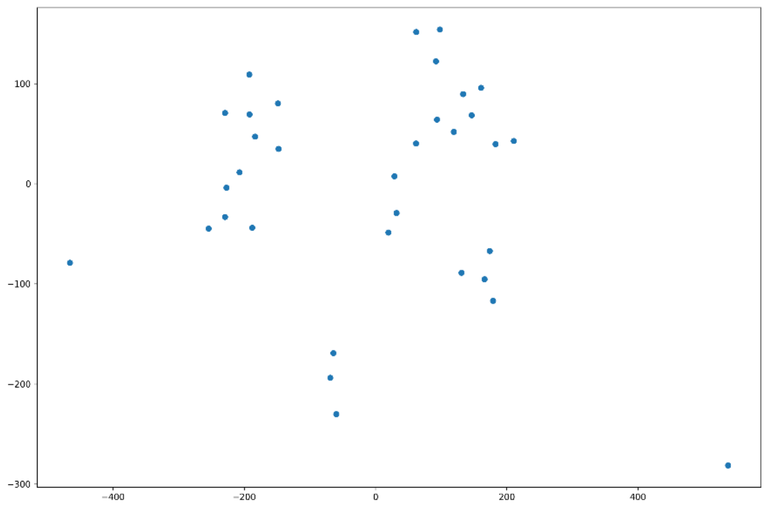


Figure A.4 t-SNE-based visualization for the low-dimensional representations generated by node2vec.

A.5.2 DeepWalk

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

```
$ pip install karateclub
```

The following code illustrates how to use DeepWalk.

Listing A.4 DeepWalk Example

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


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

embedding=model.get_embedding() #E

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

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

pca=PCA(n_components=2) #G
pca_out=pca.fit_transform(X) #G

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

#A Import DeepWalk. Noe2Vec is also available
#B Import Principal Component Analysis (PCA)
#C Create karate club graph
#D Define DeepWalk mode and fit the graph
#E Graph embedding
#F In KarateClub dataset, member represented by each node belongs can be 'Mr. Hi' or 'Officer'
#G Dimensionality reduction using Principal Component Analysis (PCA)
#H Visualize the embedding

A.5.3 PyG

PyG (PyTorch Geometric) is a library for implementing graph neural networks in Python using the PyTorch deep learning framework. It provides tools for defining, training, and evaluating Graph Neural Network (GNN) models, as well as for visualizing the optimization process. PyG supports a variety of GNN architectures, including Graph Convolutional Network (GCN) and Graph Attention Networks (GATs). You can install PyG as follows:

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

The following code shows how to use PyG to generate Karate Club graph embedding using GCN.

Listing A.4 PyG Example

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

dataset = KarateClub() #A
data = dataset[0]

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

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

        return out, h

model = GCN() #E

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

for epoch in range(401):
    out, h = model(data.x, data.edge_index) #H
    loss = criterion(out[data.train_mask], data.y[data.train_mask]) #I
    loss.backward()  #J
    optimizer.step() #K

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

#A Use karate club dataset
#B Graph Convolutional Network class
#C Apply a final (linear) classifier
#D Final GNN embedding space
#E Define the model
#F Define loss criterion
#G Define optimizer and clear gradient
#H Perform a single forward pass
#I Compute the loss solely based on the training nodes

#J Derive gradients
#K Update parameters based on gradients
#L Convert 'h' from tenser format to numpy format for visualize
#M Visualize the embedding

PyG is well-supported library that provides the several features such as common benchmark datasets (e.g., KarateClub, CoraFull, Amazon, Reddit, Actor), data handling of graphs, mini-batches, data transforms and learning methods on graphs (e.g., Node2Vec, MLP, GCN, GAT, GraphSAGE, GraphUNet, DeepGCNLayer, GroupAddRev and MetaLayer).

A.5.4 OpenAI Gym

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

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

Gymnasium is a maintained fork of OpenAI's Gym library. The following code illustrate how to use OpenAI Gym.

Listing A.4 OpenAI Gym Example

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

#A Install all included environments
#B Create an environment

In this simple example, MountainCar-v0 has discrete actions. You can also use MountainCarCoutinous-V0 that has continuous actions corresponding to the force that the car is pushed. Complete listing is available in *Listing A.4_ML_libraries.ipynb*.

A.5.5 Flow

Flow is a deep reinforcement learning framework for mixed autonomy traffic. It allows you to run deep RL-based control experiments for traffic microsimulation. You can install Flow as follows:

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

Install flow within the environment

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

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

A.5.6 Other libraries

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

- Deep Graph Library (DGL) is a library for implementing graph neural networks in Python. It provides tools for defining, training, and evaluating GNN models, as well as for visualizing the optimization process. DGL supports a variety of GNN architectures, including GCN and GAT.
- Stanford Network Analysis Platform (SNAP) is a general-purpose, high-performance system for analysis and manipulation of large complex networks. SNAP includes a number of algorithms for network analysis, such as centrality measures, community detection, and graph generation. It is particularly well-suited for large-scale network analysis and is used in a variety of fields, including computer science, physics, biology, and social science.
- Spektral is an open-source Python library for graph neural networks (GNNs) built on top of TensorFlow and Keras. It is designed to make it easy to implement GNNs in research and production. It provides a high-level, user-friendly API for building GNNs, as well as a number of pre-built layers and models for common tasks in graph deep learning. The library also includes utilities for loading and preprocessing graph data, and for visualizing and evaluating the performance of GNNs.
- Jraph (pronounced "giraffe") is a lightweight library for working with graph neural networks. Jraph (pronounced "giraffe") is a lightweight library for working with graph neural networks and it provide lightweight data structure for working with graphs. You can easily work with this library to construct and visualize your graph.
- TF-Agents is a library for developing RL algorithms in TensorFlow, which includes a collection of environments, algorithms, and tools for training RL agents.
- Keras-RL is a deep reinforcement learning library build in top of Keras. It provides an easy-to-use interface for developing and testing RL algorithms. Keras-RL supports a variety of RL algorithms such as deep Q-networks (DQN) and actor-critic methods. There are also build in environments for testing RL algorithms.
- pyqlearning is Python library to implement Reinforcement Learning and Deep Reinforcement Learning, especially for Q-Learning, Deep Q-Network, and Multi-agent Deep Q-Network which can be optimized by annealing models such as Simulated Annealing, Adaptive Simulated Annealing, and Quantum Monte Carlo Method.
- GraphNets is a library for implementing GNNs in Python. It provides tools for defining, training, and evaluating GNN models, as well as for visualizing the optimization process. GraphNet supports a variety of GNN architectures, including GCN and GAT.

Listing A.4_ML_libraries.ipynb notebook available on the book’s GitHub repo provides examples about how to install and use some of these libraries.