Mathematical Report on NP-Hard Problems in Complexity Theory: The Knapsack Problem and the VRP as Examples

LZPMPC004M: Maths and Stats

Chenyun Fu and Vincent Cheng

May 21, 2024

Abstract

The purpose of this research is to explore the the mathematical modeling of the Vehicle Routing Problem (VRP) and to analyze the time complexity of algorithm when solving Knapsack Problem. By empirical experiments, it is observed that in dynamic programming and greedy algorithm, the the runtime growth patterns align with theoretical time complexity.

1 Introduction

Computational complexity theory is a pivotal branch of computer science that focuses on the resource requirements of algorithms, including time and space, as well as the inherent complexity of problems. In this field, researchers strive to categorize computational problems into different complexity classes based on their difficulty in the worst-case scenario. Among these complexity classes, time complexity stands out as a crucial metric, measuring the time required by algorithms to solve problems. On the other hand, the Vehicle Routing Problem (VRP), as a classic combinatorial optimization problem, has garnered widespread attention from researchers.[CC08] During the research process, abstract computing devices such as the Turing machine model are often utilized to analyze algorithm performance and explore potential approaches to solving various computational problems. Therefore, through the study of complexity theory, we can gain deeper insights into the relationship between time complexity and the VRP, thereby providing guidance and inspiration for tackling complex optimization problems in practice.

Time complexity (also named runtime complexity), the two concepts are prone to get mixed up. Time Complexity doesn't refer to the time an algorithm takes to execute, but to describe that, with the growth of the input/problem size, to what extent the executing time would expand (Geekforgeeks, 2023)[Gee23a]. While runtime directly means the actual time the code takes to execute thoroughly. Although with the growth of size of problem, both runtime and time complexity would expand, measuring runtime is not perfectly fit the time complexity for it is common sense that runtime can be significantly influenced by many factors, such as the different coding styles or hardware situations. Time complexity also is known as one of the indicators of algorithm performance. It is in line with intuition that the lower the complexity, the relatively better performance an algorithm could have.

The Vehicle Routing Problem (VRP) is a class of optimization problems aimed at determining the optimal routes for a set of vehicles that start from one or more depots, visit a group of customers, and return to the depot, minimizing the total transportation cost.[TV14] There are several variants of VRP, such as VRP with Time Windows (VRPTW) and Capacitated VRP (CVRP). Additionally, VRP is an NP-hard problem, which means that for large-scale VRP instances, it is impossible to find a polynomial-time solution algorithm. [CC08]Therefore, heuristic and metaheuristic algorithms are often used in practice to find approximate solutions.There are two main approaches in solving VRP:

1.Exact Algorithms: Methods such as branch and bound and dynamic programming, which are only suitable for small-scale problems.

2.Heuristic and Metaheuristic Algorithms: Methods such as greedy algorithms, simulated annealing, genetic algorithms, and ant colony optimization, which are suitable for large-scale problems and can find good solutions within a reasonable amount of time.

The practical significance of complexity theory lies in its ability to identify problems that are theoretically difficult to solve, thereby guiding the choice of solution strategies for practical problems.

2 Complexity Classes[HS11]

- P Problems that can be solved in polynomial time. This means there exists an algorithm that can solve these problems in polynomial time with respect to the input size n.
- NP Problems for which a solution can be verified in polynomial time. That is, given a candidate solution, its correctness can be verified within polynomial time.
- NP-Complete Problems If a polynomial-time algorithm can be found to solve any NP-complete problem, then all NP problems can be solved in polynomial time.
- NP-Hard Problems that are at least as hard as NP-complete problems. NP-hard problems are not necessarily in NP, but if an NP-hard problem can be solved in polynomial time, then all NP problems can also be solved in polynomial time.

3 NP-Hard Problems-Knapsack Problem and VRP

NP-hard problems are those to which all NP problems can be polynomial-time reduced. They represent a critical class of problems in complexity theory because they embody the limits of computational

complexity.

3.1 NP-Completeness

A problem L is called NP-complete if:

1. L belongs to the NP class.
2. Every problem in NP can be transformed to L via polynomial-time reduction.

3.2 Importance

The importance of NP-hard and NP-complete problems lies in their role in understanding the difficulty of solving computational problems. Specifically, finding a polynomial-time solution for an NP-complete problem would imply $P = NP$, one of the greatest unsolved questions in computer science.

3.3 Example Problems

- **Vehicle Routing Problem, VRP**:Is a classic problem in operations research and combinatorial optimization, widely applied in logistics, distribution, transportation, and e-commerce. The core objective of VRP is to optimize the routes of several vehicles to minimize the total travel distance or cost, thereby meeting certain demands.In logistics distribution, VRP is used to plan the optimal routes for delivery vehicles to ensure that goods can be delivered to customers in a timely and efficient manner. Considering each vehicle's capacity constraints and the customers' demands, properly arranging the routes can significantly reduce transportation costs and time.[DR59]
- **Traveling Salesman Problem (TSP)**: Given a set of cities and the distances between them, find the shortest possible route that visits each city once and returns to the origin city.
- **Hamiltonian Path Problem**: Determine if there exists a path in a graph that visits each vertex exactly once.
- **Subset Sum Problem**: Given a set of integers, determine if there is a subset whose sum equals a specific value.
- **Knapsack Problem**: Knapsack Problem is about maximizing the total value of a group of items with different weights, within the limitation of a given total weight (the capacity of the knapsack).

4 Methodology

In pervious study, Pranav, Dutta, and Chakraborty conducted an empirical and statistical analysis on an algorithm named AES-128. In the experience, they measured 5 times the runtime of the specified algorithm, got the average value and fit the data. They successfully used runtime and input size to prove and approximate the empirical time complexity of AES-128 is O(n).

Inspired by previous study (Pranav, Dutta, and Chakraborty, 2022)[PDC22], empirical analysis through quantitative research will be conducted to investigate the trend of the runtime and compare it with the given theoretical time complexity. The dataset is randomly generated by python, and the experiment will also conduct in python. The problem would be solved in the following experiments is knapsack problem. In total two algorithm will be tested: dynamic programming algorithm and greedy algorithm. The previous two algorithms will both solve a single knapsack problem. In the experiments, the python library matplotlib, scipy and numpy are used to draw diagram and to analysis.

According to Google OR-Tools [Goo24],OR-Tools is open source software for combinatorial optimization, which seeks to find the best solution to a problem out of a very large set of possible solutions. Here are some examples of problems that OR-Tools solves:

Vehicle routing: Find optimal routes for vehicle fleets that pick up and deliver packages given constraints (e.g., "this truck can't hold more than 20,000 pounds" or "all deliveries must be made

within a two-hour window"). Scheduling: Find the optimal schedule for a complex set of tasks, some of which need to be performed before others, on a fixed set of machines, or other resources. Bin packing: Pack as many objects of various sizes as possible into a fixed number of bins with maximum capacities. In most cases, problems like these have a vast number of possible solutions—too many for a computer to search them all. To overcome this, OR-Tools uses state-of-the-art algorithms to narrow down the search set, in order to find an optimal (or close to optimal) solution.

5 Findings about Knapsack problem and VRP

5.1 Knapsack Problem 2 Experiments

5.1.1 Knapsack Experiment 1

The first experiment is about using dynamic programming algorithm to solve knapsack problem. The code and the raw dataset will be attached in the appendix. The main content of the experiment is that there exists one single knapsack with capacity m and n pieces of items. Each item has different weight w and value v. The goal is to find proper number of specific items with a maximum total value, where these items can be put exactly in the knapsack (the total weight of the items picked should not be more than the capacity of the knapsack). In brief, the dynamic programming solves the problem by comparing and updating the items in its list. It loops to try put each item to the list, until find the maximum total value. The experiment fixes the capacity of the knapsack to be 50, sets the number of items from 1 to 500 and repeats the process 5 times. In each trial and each loop, the runtime is recorded. The average runtime represents an average value of 5 trials. So, there are 500 observed data points displayed in the scatter plot. In Figure 1, on the left side is a straight-forward result of the increasing trend of runtime with the growth of the input size, which is the number of items need to be placed in the container. The scatter plot shows a linear growth trend. But not all the data points fit the linear trend. On the right side is a plot of residuals. Residual means the difference between the actual value and the fitted value (Mathworks, 2024).[Mat24] The residual plot preliminarily demonstrates the extent of the fit of the observed data and the fitted data.

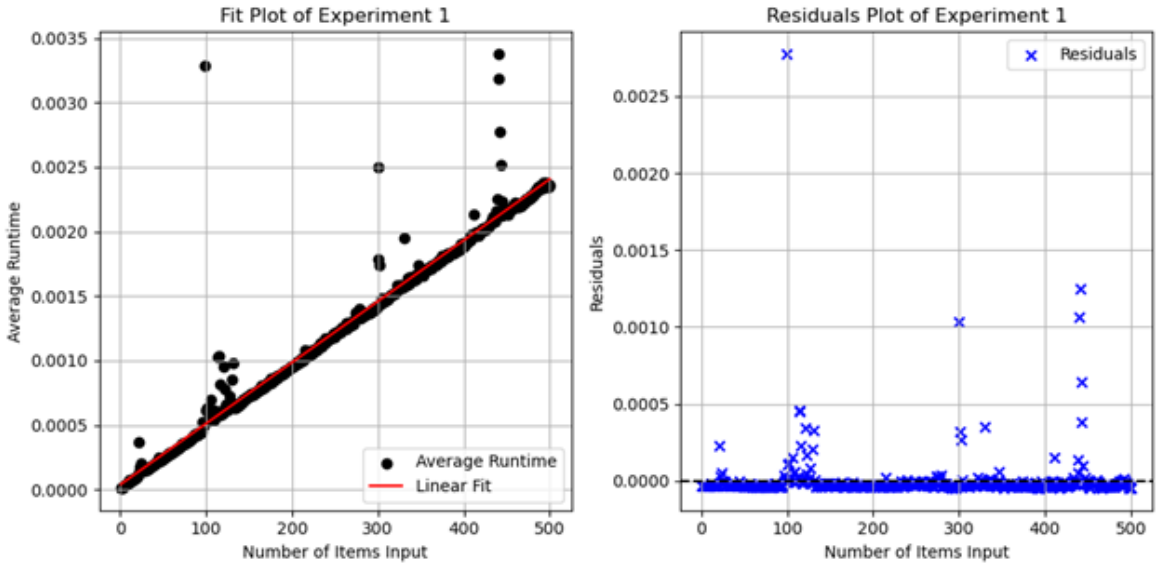

Figure 1: Fit Plot and Residuals Plot of Experiment 1

Using `numpy` function, which uses the principle of least squares, the regression equation is fitted in the form of a first-degree polynomial: $y = ax + b$; the parameters are $4.73892774 \times 10^{-6}$ and $3.67912908 \times 10^{-5}$ respectively. The value of R^2 (coefficient of determination) is 0.9447530056527242

and the value of R (Pearson correlation coefficient) is 0.9719840562749598, which are both close to 1, indicating that there is a high degree of linear correlation between the two variables.

According to the original resource of the dynamic programming algorithm, the time complexity of using it to solve the knapsack problem is $O(n \times m)$, with n items and a knapsack with capacity m (Python Algorithms, 2019).[Pyt19] In Experiment 1, m is fixed as 50. Therefore, the theoretical time complexity should be $O(n)$. The constant factor can be removed because it does not influence the linear trend of growth.

Since the fitting results of the observed data points prove that the two variables: the number of items and the average runtime have a high degree of linear correlation, and the theoretical time complexity also indicates that the running time will increase linearly with the size of the input, we can conclude that the experiment confirms the theoretical time complexity.

5.1.2 Knapsack Experiment 2

The second experiment is about using greedy algorithm to solve knapsack problem. The same dataset as in Experiment 1 will be used in this section. The content of Experiment 2 is similar to Experiment 1. But in greedy algorithm for solving knapsack problem, the item is fractionable, and the value of the divided item is calculated based on the proportion of the item's divided weight to its original weight. In general, in greedy algorithm the items will first be sorted into a list according to its value-to-weight ratio (Sharma, 2022).[Sha22] Thereafter, the item will be looped and checked if it can be added completely or partly into the knapsack.

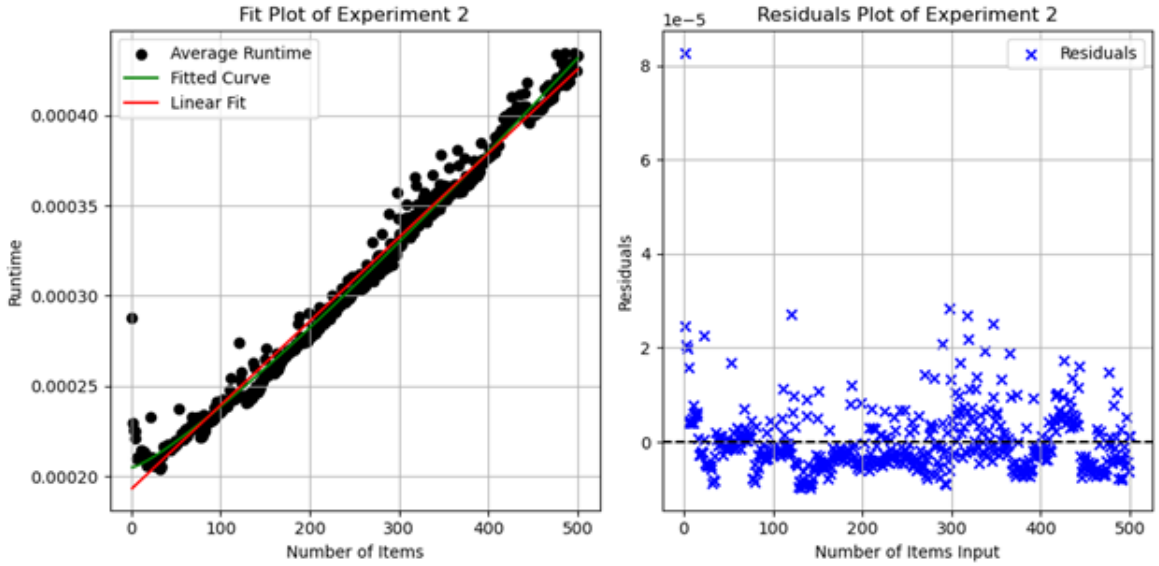

Figure 2: Fit Plot and Residuals Plot of Experiment 2

Different from the dynamic programming algorithm, the theoretical time complexity of the greedy algorithm is $O(n \log n)$ (Geeksforgeeks, 2023)[Gee23b]. The main reason is that the built-in sorting algorithm has a time complexity of $O(n \log n)$, while other parts of the greedy algorithm have relatively lower time complexity.

From the scatter plot, it is difficult to validate whether the trend fits a logarithmic curve. Therefore, both linear regression and curve fitting are used in Experiment 2 to evaluate which fit is better.

The parameters of linear regression obtained in Experiment 2 are $4.64679355 \times 10^{-7}$ and $1.93014974 \times 10^{-4}$, in the form of a first-degree polynomial: $y = ax + b$. The parameters of curve fitting are $7.28640879 \times 10^{-8}$ and $2.05093203 \times 10^{-4}$, in the form of $y = a \cdot x \cdot \log(x) + b$. The curve fitting has higher R and R^2 than the linear regression, indicating that the second function has better performance when explaining the data. We can draw a preliminary conclusion that the experiment also confirms the theoretical time complexity of $O(n \log n)$.

However, there are flaws in this approach. When n is below 500, the difference between the linear and logarithmic functions is not very apparent due to the small magnitude of the constants and parameters.

5.2 Vehicle Routing Problem Mathematical Modeling and Application

5.2.1 Mathematical Modeling Of VRP[TV14]

- Definitions
  - $G = (V, A)$: Represents a graph where V is the set of nodes and A is the set of arcs (edges).
  - $V = \{0, 1, 2, \ldots, n\}$: Represents the set of nodes, where node 0 denotes the central depot and nodes 1 to n denote the customers.
  - $A = \{(i, j) \mid i, j \in V, i \neq j\}$: Represents the set of arcs between nodes.
- Parameters
  - d_{ij}: Represents the distance or cost from node i to node j.
  - q_i: Represents the demand of customer i.
  - Q: Represents the capacity limit of each vehicle.
  - K: Represents the number of vehicles available to service the customers.
- Variables
  - x_{ij}: Binary variable, which is 1 if there is a path from node i to node j, and 0 otherwise.
  - u_i: Represents the start time of service at node i.
- Objective Function

  The objective of the VRP is to minimize the total travel distance or cost. Thus, our objective function can be expressed as:

$$\text{minimize} \sum_{i \in V} \sum_{j \in V, j \neq i} d_{ij} x_{ij}$$

- Constraints
  - Each customer must be visited exactly once

$$\sum_{j \in V, j \neq i} x_{ij} = 1, \quad \forall i \in V, i \neq 0$$

  - Each vehicle leaves the depot, visits customers, and returns to the depot

$$\sum_{j \in V, j \neq 0} x_{0j} = K$$

$$\sum_{i \in V, i \neq 0} x_{i0} = K$$

  - Capacity constraint

$$\sum_{i \in V} q_i x_{ij} \leq Q, \quad \forall j \in V$$

  - Route determination

$$u_i + q_i \leq u_j, \quad \forall i, j \in V, j \neq 0, x_{ij} = 1$$

- Solution

  We use integer programming or heuristic algorithms to solve the VRP problem to obtain optimal or near-optimal solutions.

- Conclusion

  The VRP is a complex combinatorial optimization problem with extensive applications in logistics and distribution. By employing appropriate mathematical modeling and effective solution algorithms, we can find the best vehicle routes to minimize total travel distance or cost, thereby enhancing transportation efficiency and reducing costs.[TV14]

  And by conducting empirical experiments, we can preliminary confirm that in Knapsack Problem, using dynamic programming algorithm and greedy algorithm, the pattern of the growth of runtime of the algorithm with the increase of input size could match the theoretical pattern of time complexity.

5.2.2 Practical Applications and Solutions of the VRP

Consider a scenario with 15 demand points (customers) represented by a distance matrix, and the task requires using a limited number of vehicles (2 vehicles) to deliver goods from a depot to these customers along the shortest distance routes (which also minimizes the cost) and then return to the depot.

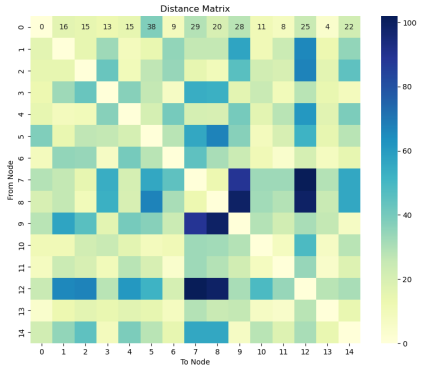

Figure 3: Heatmap of Distance Matrix

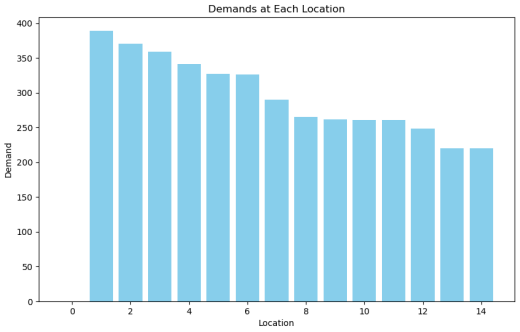

Figure 4: Bar Chart of Demands

$$\text{data} = \{\ \text{distance\_matrix} = \begin{bmatrix} 0 & 16 & 15 & 13 & 15 & 38 & 9 & 29 & 20 & 28 & 11 & 8 & 25 \\ 4 & 22 & & & & & & & & & & & \\ 16 & 0 & 15 & 33 & 9 & 14 & 35 & 26 & 26 & 57 & 11 & 24 & 66 \\ 13 & 35 & & & & & & & & & & & \\ 15 & 15 & 0 & 43 & 10 & 27 & 34 & 15 & 10 & 46 & 22 & 20 & 67 \\ 16 & 45 & & & & & & & & & & & \\ 13 & 33 & 43 & 0 & 37 & 26 & 7 & 54 & 53 & 16 & 25 & 10 & 28 \\ 14 & 9 & & & & & & & & & & & \\ 15 & 9 & 10 & 37 & 0 & 21 & 40 & 21 & 20 & 40 & 16 & 28 & 61 \\ 17 & 39 & & & & & & & & & & & \\ 38 & 14 & 27 & 26 & 21 & 0 & 28 & 56 & 67 & 37 & 9 & 20 & 52 \\ 15 & 28 & & & & & & & & & & & \\ 9 & 35 & 34 & 7 & 40 & 28 & 0 & 45 & 31 & 24 & 11 & 4 & 21 \\ 8 & 15 & & & & & & & & & & & \\ 29 & 26 & 15 & 54 & 21 & 56 & 45 & 0 & 13 & 89 & 33 & 33 & 102 \\ 29 & 56 & & & & & & & & & & & \\ 20 & 26 & 10 & 53 & 20 & 67 & 31 & 13 & 0 & 99 & 32 & 29 & 99 \\ 25 & 56 & & & & & & & & & & & \\ 28 & 57 & 46 & 16 & 40 & 37 & 24 & 89 & 99 & 0 & 29 & 22 & 31 \\ 26 & 7 & & & & & & & & & & & \\ 11 & 11 & 22 & 25 & 16 & 9 & 11 & 33 & 32 & 29 & 0 & 8 & 49 \\ 7 & 28 & & & & & & & & & & & \\ 8 & 24 & 20 & 10 & 28 & 20 & 4 & 33 & 29 & 22 & 8 & 0 & 34 \\ 5 & 18 & & & & & & & & & & & \\ 25 & 66 & 67 & 28 & 61 & 52 & 21 & 102 & 99 & 31 & 49 & 34 & 0 \\ 28 & 31 & & & & & & & & & & & \\ 4 & 13 & 16 & 14 & 17 & 15 & 8 & 29 & 25 & 26 & 7 & 5 & 28 \\ 0 & 14 & & & & & & & & & & & \\ 22 & 35 & 45 & 9 & 39 & 28 & 15 & 56 & 56 & 7 & 28 & 18 & 31 \\ 14 & 0 & & & & & & & & & & & \end{bmatrix},$$

$$\text{demands} = [0, 389, 370, 359, 341, 327, 326, 290, 265, 262, 261, 261, 248, 220, 220],$$

$$\text{vehicle\_capacities} = [2198, 2198],$$

$$\text{num\_vehicles} = 2,$$

$$\text{depot} = 0$$

- 1. Create the data model.
  - distance matrix: Distance matrix between points.As figure 3.
  - demands: Demands of each point (customer).As figure 4.
  - vehicle capacities: Capacities of each vehicle.
  - num vehicles: Number of vehicles.
  - depot: Index of the depot.
- 2. Initialize RoutingIndexManager and RoutingModel.
- 3. Define and register a distance callback function to compute the distance from one point to another.
- 4. Define and register a demand callback function to retrieve the demand for each point.
- 5. Add capacity constraints for the vehicles.
- 6. Set the search parameters, selecting the initial route strategy as "PATH CHEAPEST ARC".

- 7. Solve the problem and print the solution.As figure 5.

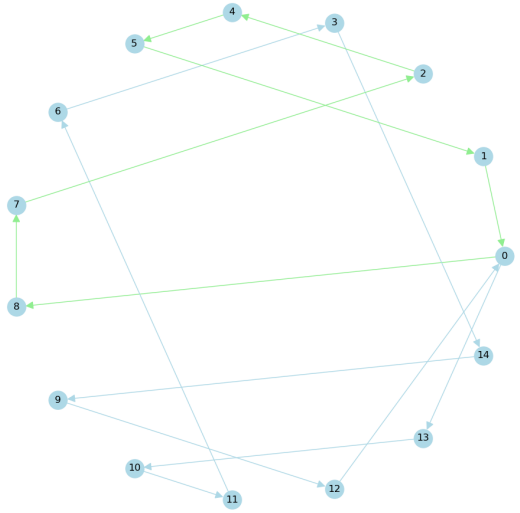

Figure 5: Visualization of Vehicle Routes

6 Conclusion

In the fields of computer science and operations research, NP-hard problems in complexity theory hold significant importance. This paper focuses on discussing two classic NP-hard problems—the Knapsack Problem and the Vehicle Routing Problem (VRP)—and conducts mathematical analysis and solutions using Python. By leveraging programming languages and their third-party libraries, quick and efficient solutions to the problems were obtained. However, due to limited time and expertise, this study has many limitations, and we welcome criticism and suggestions from readers.

References

[CC08] T. Caric and T. Caric. Vehicle routing problem. In H. Gold, editor, *Vehicle Routing Problem*. IntechOpen, 2008.

[DR59] George B. Dantzig and John H. Ramser. The truck dispatching problem. *Management Science*, 6(1):80–91, 1959.

[Gee23a] GeeksforGeeks. Fractional Knapsack Problem. https://www.geeksforgeeks.org/fractional-knapsack-problem/, 2023. Accessed 21 May 2024.

[Gee23b] GeeksforGeeks. Greedy Approach vs Dynamic Programming. https://www.geeksforgeeks.org/greedy-approach-vs-dynamic-programming/, 2023. Accessed 21 May 2024.

[Goo24] Google Developers. Introduction to Optimization. https://developers.google.com/optimization/introduction, Accessed 2024.

[HS11] S. (Steven) Homer and A. L. Selman. *Computability and Complexity Theory*. Springer US, 2011.

[Mat24] Mathwork. Evaluating Goodness of Fit. https://ww2.mathworks.cn/help/curvefit/evaluating-goodness-of-fit.html, 2024. Accessed 21 May 2024.

[PDC22] P. Pranav, S. Dutta, and S. Chakraborty. Empirical and Statistical Complexity Analysis of AES-128. https://doi.org/10.21203/rs.3.rs-1418564/v1, 2022. Accessed 21 May 2024.

[Pyt19] Python Algorithms. chapter 8: Dynamic Programming. https://high-python-ext-3-algorithms.readthedocs.io/ko/latest/chapter8.html, 2019. Accessed 21 May 2024.

[Sha22] P. Sharma. Knapsack Problem in Python. https://www.analyticsvidhya.com/blog/2022/05/knapsack-problem-in-python/, 2022. Accessed 21 May 2024.

[TV14] Paolo Toth and Daniele Vigo. *Vehicle Routing: Problems, Methods, and Applications*. SIAM, 2nd edition, 2014.

A OR-Tools Routing Code

```
// please go to: https://github.com/vincentyelpen/vrp
//output:

Vehicle 0 route: 0 -> 13 -> 10 -> 11 -> 6 -> 3 -> 14 -> 9 -> 12 -> 0
Distance of the route: 102.00m
Cost of the route: 153.00

Vehicle 1 route: 0 -> 8 -> 7 -> 2 -> 4 -> 5 -> 1 -> 0
Distance of the route: 109.00m
Cost of the route: 163.50

Total distance of routes: 211.00m
Total cost of routes: 316.50

from ortools.constraint_solver import routing_enums_pb2
from ortools.constraint_solver import pywrapcp

def create_data_model():
    data = {}
    data['distance_matrix'] = [
        [0, 16, 15, 13, 15, 38, 9, 29, 20, 28, 11, 8, 25, 4, 22],
        [16, 0, 15, 33, 9, 14, 35, 26, 26, 57, 11, 24, 66, 13, 35],
        [15, 15, 0, 43, 10, 27, 34, 15, 10, 46, 22, 20, 67, 16, 45],
        [13, 33, 43, 0, 37, 26, 7, 54, 53, 16, 25, 10, 28, 14, 9],
        [15, 9, 10, 37, 0, 21, 40, 21, 20, 40, 16, 28, 61, 17, 39],
        [38, 14, 27, 26, 21, 0, 28, 56, 67, 37, 9, 20, 52, 15, 28],
        [9, 35, 34, 7, 40, 28, 0, 45, 31, 24, 11, 4, 21, 8, 15],
        [29, 26, 15, 54, 21, 56, 45, 0, 13, 89, 33, 33, 102, 29, 56],
        [20, 26, 10, 53, 20, 67, 31, 13, 0, 99, 32, 29, 99, 25, 56],
        [28, 57, 46, 16, 40, 37, 24, 89, 99, 0, 29, 22, 31, 26, 7],
        [11, 11, 22, 25, 16, 9, 11, 33, 32, 29, 0, 8, 49, 7, 28],
        [8, 24, 20, 10, 28, 20, 4, 33, 29, 22, 8, 0, 34, 5, 18],
        [25, 66, 67, 28, 61, 52, 21, 102, 99, 31, 49, 34, 0, 28, 31],
        [4, 13, 16, 14, 17, 15, 8, 29, 25, 26, 7, 5, 28, 0, 14],
        [22, 35, 45, 9, 39, 28, 15, 56, 56, 7, 28, 18, 31, 14, 0],
    ]
    data['demands'] = [0, 389, 370, 359, 341, 327, 326, 290, 265, 262, 261, 261, 248,
        220, 220]
    data['vehicle_capacities'] = [2198, 2198]
    data['num_vehicles'] = 2
    data['depot'] = 0
    return data

def print_solution(data, manager, routing, solution):
    """Prints solution on console."""
    total_distance = 0.0
    total_cost = 0.0
    cost_per_meter = 1.5
    for vehicle_id in range(data['num_vehicles']):
        index = routing.Start(vehicle_id)
        plan_output = f'Vehicle {vehicle_id}\'s route: '
        route_distance = 0.0
        while not routing.IsEnd(index):
            plan_output += f'{manager.IndexToNode(index)} -> '
            previous_index = index
            index = solution.Value(routing.NextVar(index))
            route_distance += routing.GetArcCostForVehicle(previous_index, index,
                vehicle_id)
        route_cost = route_distance * cost_per_meter
        plan_output += f'{manager.IndexToNode(index)}'
        plan_output += f'\nDistance of the route: {route_distance:.2f}m'
        plan_output += f'\nCost of the route:   {route_cost:.2f}\n'
        print(plan_output)
        total_distance += route_distance
        total_cost += route_cost
    print(f'Total distance of all routes: {total_distance:.2f}m')
    print(f'Total cost of all routes:   {total_cost:.2f}')

def main():
```

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


    def distance_callback(from_index, to_index):

        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        return data['distance_matrix'][from_node][to_node]

    transit_callback_index = routing.RegisterTransitCallback(distance_callback)

    routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)


    def demand_callback(from_index):

        from_node = manager.IndexToNode(from_index)
        return data['demands'][from_node]
    demand_callback_index = routing.RegisterUnaryTransitCallback(demand_callback)

    routing.AddDimensionWithVehicleCapacity(
        demand_callback_index,
        0,
        data['vehicle_capacities'],
        True,
        'Capacity')

    search_parameters = pywrapcp.DefaultRoutingSearchParameters()
    search_parameters.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)

    solution = routing.SolveWithParameters(search_parameters)

    if solution:
        print_solution(data, manager, routing, solution)
    else:
        print('No solution found !')

if __name__ == '__main__':
    main()
```

Listing 1: Python code for solving a vehicle routing problem using OR-Tools

B Knapsack-problem Code

```
// listed is the processing codes
// for the raw data and whole project of the experiment
// please go to: https://github.com/ChartOn/Math_Project_Complexity

// data generating code
import random

n = 500
filename = "dataset.txt"


with open(filename, 'w') as file:
    for _ in range(n):
        value = random.randint(1, 100)
        weight = random.randint(1, 10)
        file.write(f"{value} {weight}\n")

// dynamic programming code
# modified from the originial code:
```

```
# https://www.analyticsvidhya.com/blog/2022/05/knapsack-problem-in-python/

from fractions import Fraction
import time

class KnapsackPackage(object):

    def __init__(self, value, weight):
        self.weight = weight
        self.value = value
        self.cost = value / weight

    def __lt__(self, other):
        return self.cost < other.cost

class FractionalKnapsack(object):
    def knapsackGreProc(self, W, V, M, n):
        packs = []
        for i in range(n):
            packs.append(KnapsackPackage(W[i], V[i]))
        packs.sort(reverse=True)
        remain = M
        result = 0
        for pack in packs:
            if remain >= pack.weight:
                result += pack.value
                remain -= pack.weight
            else:
                fraction = Fraction(remain, pack.weight)
                result += fraction * pack.value
                break
        return result

with open("dataset.txt", 'r') as file:
    lines = file.readlines()
    items = [KnapsackPackage(*map(int, line.strip().split())) for line in lines]

trial = 5
M = 50

with open("greedy_results.txt", 'w') as result_file:
    for n in range(1, 501):
        trial_times = []
        for _ in range(trial):
            start_time = time.time()
            FractionalKnapsack().knapsackGreProc([item.weight for item in items[:n]],
                [item.value for item in items[:n]], M, n)
            end_time = time.time()
            exe_time = end_time - start_time
            trial_times.append(str(exe_time))

        result_file.write(f"n={n}: {','.join(trial_times)}\n")

// regression and ploting code for Experiment 1
import matplotlib.pyplot as plt
import numpy as np

n_values = []
average_times = []

with open("dynamic_results.txt", 'r') as result_file:
    for line in result_file:
        parts = line.strip().split(': ')
        n_values.append(int(parts[0][2:]))
        times = parts[1].split(',')
        average_time = sum(map(float, times)) / len(times)
        average_times.append(average_time)

poly_coeffs = np.polyfit(n_values, average_times, 1)
poly_func = np.poly1d(poly_coeffs)
```

```
residuals = average_times - poly_func(n_values)
mse = np.mean(residuals**2)
rmse = np.sqrt(mse)
mae = np.mean(np.abs(residuals))

ss_total = np.sum((average_times - np.mean(average_times))**2)
ss_residual = np.sum(residuals**2)
r_squared = 1 - (ss_residual / ss_total)
r = np.sqrt(r_squared)

print("Linear Fit Parameters:", poly_coeffs)
print("MSE:", mse)
print("RMSE:", rmse)
print("MAE:", mae)
print("R^2", r_squared)
print("R:", r)

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.scatter(n_values, average_times, label='Average Runtime', color='black')
plt.plot(n_values, poly_func(n_values), color='red', label='Linear Fit')
plt.xlabel('Number of Items Input')
plt.ylabel('Average Runtime')
plt.title('Fit Plot of Experiment 1')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.scatter(n_values, residuals, label='Residuals', color='blue', marker='x')
plt.xlabel('Number of Items Input')
plt.ylabel('Residuals')
plt.title('Residuals Plot of Experiment 1')
plt.axhline(y=0, color='black', linestyle='--')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

// greedy algorithm code
# modified from the originial code:
# https://www.analyticsvidhya.com/blog/2022/05/knapsack-problem-in-python/

from fractions import Fraction
import time

class KnapsackPackage(object):

    def __init__(self, value, weight):
        self.weight = weight
        self.value = value
        self.cost = value / weight

    def __lt__(self, other):
        return self.cost < other.cost

class FractionalKnapsack(object):
    def knapsackGreProc(self, W, V, M, n):
        packs = []
        for i in range(n):
            packs.append(KnapsackPackage(W[i], V[i]))
        packs.sort(reverse=True)
        remain = M
        result = 0
        for pack in packs:
            if remain >= pack.weight:
                result += pack.value
                remain -= pack.weight
            else:
                fraction = Fraction(remain, pack.weight)
                result += fraction * pack.value
```

```
                break
        return result

with open("dataset.txt", 'r') as file:
    lines = file.readlines()
    items = [KnapsackPackage(*map(int, line.strip().split())) for line in lines]

trial = 5
M = 50

with open("greedy_results.txt", 'w') as result_file:
    for n in range(1, 501):
        trial_times = []
        for _ in range(trial):
            start_time = time.time()
            FractionalKnapsack().knapsackGreProc([item.weight for item in items[:n]],
                [item.value for item in items[:n]], M, n)
            end_time = time.time()
            exe_time = end_time - start_time
            trial_times.append(str(exe_time))

        result_file.write(f"n={n}: {','.join(trial_times)}\n")

// regression and ploting code for Experiment 2
import matplotlib.pyplot as plt
import numpy as np

n_values = []
average_times = []

with open("greedy_results.txt", 'r') as result_file:
    for line in result_file:
        parts = line.strip().split(': ')
        n_values.append(int(parts[0][2:]))
        times = parts[1].split(',')
        average_time = sum(map(float, times)) / len(times)
        average_times.append(average_time)

poly_coeffs = np.polyfit(n_values, average_times, 1)
poly_func = np.poly1d(poly_coeffs)

residuals = average_times - poly_func(n_values)
mse = np.mean(residuals**2)
rmse = np.sqrt(mse)
mae = np.mean(np.abs(residuals))

ss_total = np.sum((average_times - np.mean(average_times))**2)
ss_residual = np.sum(residuals**2)
r_squared = 1 - (ss_residual / ss_total)
r = np.sqrt(r_squared)

print("Linear Fit Parameters:", poly_coeffs)
print("MSE:", mse)
print("RMSE:", rmse)
print("MAE:", mae)
print("R^2", r_squared)
print("R:", r)

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.scatter(n_values, average_times, label='Average Runtime', color='black')
plt.plot(n_values, poly_func(n_values), color='red', label='Linear Fit')
plt.xlabel('Number of Items Input')
plt.ylabel('Average Runtime')
plt.title('Fit Plot of Experiment 1')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.scatter(n_values, residuals, label='Residuals', color='blue', marker='x')
plt.xlabel('Number of Items Input')
```

```
plt.ylabel('Residuals')
plt.title('Residuals Plot of Experiment 1')
plt.axhline(y=0, color='black', linestyle='--')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()

//outputs:
//Experiment 1
Linear Fit Parameters: [4.64679355e-07 1.93014974e-04]
MSE: 7.616803330937752e-11
RMSE: 8.727429937236821e-06
MAE: 6.072311032151858e-06
R^2 0.9833498927409517
R: 0.9916400015837158

//Experiment 2
Fit Parameters: [7.28640879e-08 2.05093203e-04]
MSE: 5.422379858547268e-11
RMSE: 7.363681048597412e-06
MAE: 4.960340516921796e-06
R^2: 0.988146837679607
R: 0.9940557517964508
```

Listing 2: Python code for solving Knapsack Problem and for empirical analysizing