CSC4202-6 : DESIGN AND ANALYSIS OF ALGORITHMS

LECTURER NAME:
DR. NUR ARZILAWATI MD YUNUS

Team Members	Matric Number
MUHAMMAD SHAKIR IQBAL BIN SAMRI	210362
MUHAMMAD ARIF BIN ZULKEPLI	210079
AHMAD HAZIQ BIN SUZAKI	209922

CSC4202 Design and Analysis of Algorithm
Second Semester, 2023-2024

Initial Project Plan

<table>
<tr><td>Group Name</td><td>Shakir, Arif, Haziq Team.</td></tr>
<tr><td>Members</td><td>
<table>
<tr><th>Name</th><th>Email</th><th>Phone number</th></tr>
<tr><td>MUHAMMAD SHAKIR IQBAL BIN SAMRI</td><td>210362@student.upm.edu.my</td><td>0128339095</td></tr>
<tr><td>MUHAMMAD ARIF BIN ZULKEPLI</td><td>210079@student.upm.edu.my</td><td>01120993723</td></tr>
<tr><td>AHMAD HAZIQ BIN SUZAKI</td><td>209922@student.upm.edu.my</td><td>0149258820</td></tr>
</table>
</td></tr>
<tr><td>Problem scenario description</td><td>Finding the perfect rental house is a common challenge for students especially when there are numerous options available in a given area. Let's consider a scenario where a student is looking for a rental house near a university, such as in the Seri Kembangan area, which has many housing options. The distance between these houses can make it difficult for students to decide where to start their search different houses.</td></tr>
<tr><td>Why it is important</td><td>1. Efficiency in Decision-Making
2. Cost Savings
3. Comprehensive Comparison
4. Stress Reduction
5. Environmental Impact
6. Optimal Resource Utilization</td></tr>
</table>

Problem specification	In the challenging process of finding a suitable rental house, efficiency, accessibility, and informed decision-making are crucial for students. Students are always looking for ways to optimize their search, save time, and make well-informed choices about their living arrangements. The ability to minimize travel time when viewing multiple houses in a single day is a critical factor that impacts a student's house-hunting experience. Students face difficulties in determining the most efficient routes for viewing potential rental houses, which served as the impetus for this project. Long distances between houses result in lost time, higher transportation costs, and unnecessary stress. The traditional method of manually planning house visits based on intuition or guesswork often falls short when it comes to achieving optimal efficiency in the house search process.
Potential solutions	1. Greedy Algorithm 2. Held-Karp (Dynamic Programming) 3. Graph Algorithms
Sketch (framework, flow, interface)	1. Initialize House and Travel Time 2. Generate an Initial Solution 3. Set the Current Best Solution & Current Best Travel Time 4. Calculate the Total Travel Time 5. Update the Current Best Travel Time & Solution 6. Generate a New Solution 7. Check if All Houses Have Been Visited

Project Proposal Refinement

<table>
<tr><td>Group Name</td><td>Shakir, Arif, Haziq Team.</td></tr>
<tr><td>Members</td><td>
<table>
<tr><th>Name</th><th>Role</th></tr>
<tr><td>MUHAMMAD SHAKIR IQBAL BIN SAMRI</td><td>Search for all suitable algorithms to solve the problem in the scenario by making all comparisons between the algorithms, and propose the best 3 solution algorithms.</td></tr>
<tr><td>MUHAMMAD ARIF BIN ZULKEPLI</td><td>Consider all factors, Select one of the most suitable and optimal solutions for the scenario and make the codes.</td></tr>
<tr><td>AHMAD HAZIQ BIN SUZAKI</td><td>Run the code in the IDE java compiler and analyse all the result into comparison and graph</td></tr>
</table>
</td></tr>
<tr><td>Problem statement</td><td>Finding the perfect rental house is a common challenge for students especially when there are numerous options available in a given area. A student is looking for a rental house near a university, such as in the Seri Kembangan area which has many housing options. The distance between these houses can make it difficult for students to decide where to start their search for different houses.

The goal is to find the minimum travel time to survey all possible rental houses from the student's starting point location.</td></tr>
</table>

Objectives	To find the minimum travel time to all positions from the student's starting point location. To minimise travel distance, to enable students to visit a larger number of potential rental houses within a given timeframe. To increase efficiency in their house-hunting process, allowing students to make more informed decisions and secure suitable accommodation more quickly.
Expected output	Recommend the best and nearest houses for the student to view. Efficiently plan their house viewings to make a decision on which house to rent.
Problem scenario description	Finding the perfect rental house is a common challenge for students especially when there are numerous options available in a given area. Let's consider a scenario where a student is looking for a rental house near a university, such as in the Seri Kembangan area, which has many housing options. The distance between these houses can make it difficult for students to decide where to start their search for different houses. To simplify the process, a tool can be designed to guide the student in choosing the most suitable starting point for their house hunt. This tool would recommend the best and nearest houses for the student to view. By using this tool, the student can efficiently plan their house viewings to make decision on which house to rent.
Why it is important	1. Efficiency in Decision-Making 2. Cost Savings 3. Comprehensive Comparison 4. Stress Reduction 5. Environmental Impact 6. Optimal Resource Utilization
Problem specification	In the challenging process of finding a suitable rental house, efficiency, accessibility, and informed decision-making are crucial for students. Students are always looking for ways to optimize their search, save time, and make well-informed choices about their living arrangements. The ability to minimize travel time when viewing

<table>
<tr><td></td><td>multiple houses in a single day is a critical factor that impacts a student's house-hunting experience.

Students face difficulties in determining the most efficient routes for viewing potential rental houses, which served as the impetus for this project. Long distances between houses result in lost time, higher transportation costs, and unnecessary stress. The traditional method of manually planning house visits based on intuition or guesswork often falls short when it comes to achieving optimal efficiency in the house search process.</td></tr>
<tr><td>Potential solutions</td><td>1. Held-Karp (Dynamic Programming)

2. Greedy Algorithm

3. Graph Algorithms</td></tr>
<tr><td>Sketch (framework, flow, interface)</td><td>1. Initialize House and Travel Time
2. Generate an Initial Solution
3. Set the Current Best Solution & Current Best Travel Time
4. Calculate the Total Travel Time
5. Update the Current Best Travel Time & Solution
6. Generate a New Solution
7. Check if All Houses Have Been Visited</td></tr>
<tr><td>Methodology</td><td><table>
<tr><td>Milestone</td><td>Time</td></tr>
<tr><td>Scenario refinement</td><td>week 10</td></tr>
<tr><td>Find example solutions and suitable algorithm. Discuss in group why that solution and the example problems relate to the problem in the project</td><td>week 11</td></tr>
<tr><td>Edit the coding of the chosen problem and complete the coding. Debug process.</td><td>week 12</td></tr>
</table></td></tr>
</table>

	Conduct analysis of correctness and time complexity.	week 13
	Prepare online portfolio and presentation	week 14

Project Progress

Milestone 1	Scenario refinement
Date (week)	week 10
Description/ sketch	In week 10, all the members are assigned to search for a fresh scenario that is happening around. Each member should give one scenario to the group. Total of three scenarios address which are: 1. A city involved in a flood disaster, the frontliner needs the best path to supply resources to all victims. 2. A student studying in UPM needs to find a house near Seri Kembangan to rent, the student needs to achieve the shortest time to view all the candidate houses. 3. A delivery man needs to deliver all items near the Serdang area, and needs to get the shortest route. From our group members' discussion, we decided to choose the second option which is the scenario of “A student studying in UPM needs to find a house near Seri Kembangan to rent, the student needs to achieve the shortest time to view all rent candidate houses.”. Reason: 1. More relatable to student life. 2. Can explore the best solution for the problem. 3. Can help other students if the same scenario happen in real life in future.

<table>
<tr><td>Role</td><td>
<table>
<tr><td>Member 1</td><td>Member 2</td><td>Member 3</td></tr>
<tr><td>Give one scenario

Give reason

Discuss and decide one joint decision</td><td>Give one scenario

Give reason

Discuss and decide one joint decision</td><td>Give one scenario

Give reason

Discuss and decide one joint decision</td></tr>
</table>
</td></tr>
</table>

Milestone 2	Find example solutions and suitable algorithm. Discuss in group why that solution and the example problems relate to the problem in the project
Date (Wk)	**Week 11**
Description/ **sketch**	The possible solution is disscused. 1. Greedy Algorithm 2. Held-Karp (Dynamic Programming) 3. Graph Algorithms 4. Sorting algorithm 5. Divide and conquer We arrage to scale down to suitable solution which are: 1. Greedy Algorithm 2. Held-Karp (Dynamic Programming) 3. Graph Algorithms After considering factors like a student need to have optimal solution and the number of houses want to survey is small or

<table>
<tr><td></td><td>medium, and the problem is similar to Travelling Salesman Problem, we come out with Held-Karp (Dynamic Programming).</td></tr>
<tr><td>Role</td><td>
<table>
<tr><td>Member 1</td><td>Member 2</td><td>Member 3</td></tr>
<tr><td>Give one solution to the problem

Give reason and discuss to get one final solution.</td><td>Give one solution to the problem

Give reason and discuss to get one final solution.</td><td>Give one solution to the problem

Give reason and discuss to get one final solution.</td></tr>
</table>
</td></tr>
</table>

Milestone 3	Edit the coding of the chosen problem and complete the coding. Debug process.
Date (Wk)	**Week 12**

<table>
<tr>
<td>Description/

sketch</td>
<td>After considering factors like a student need to have optimal solution and the number of houses want to survey is small or medium, and the problem is similar to Travelling Salesman Problem, we come out with Held-Karp (Dynamic Programming).

After that we continue to make and edit the coding of the java language program.

Specification of environment used:

AMD Ryzen 5 3500u

16 gb ram DDR4

Windows 11 Home Edition

Eclipse IDE for Java Developers 2021-09 Edition

The program has little problem achieving the expected output, however with solid group work we manage to debug and solve it.

Finally, the program worked successfully with the right output.</td>
</tr>
<tr>
<td>Role</td>
<td>
<table>
<tr><th>Member 1</th><th>Member 2</th><th>Member 3</th></tr>
<tr><td>Making the code

Debug the code</td><td>Making the code.

Run the program inside IDE</td><td>Making the code

Debug the code</td></tr>
</table>
</td>
</tr>
</table>

<table>
<tr><td>Milestone 3</td><td>Edit the coding of the chosen problem and complete the coding. Debug process.</td></tr>
<tr><td>Date (Wk)</td><td>Week 12</td></tr>
<tr><td>Description/ sketch</td><td>After considering factors like a student's need to have an optimal solution and the number of houses they want to survey is small or medium, and the problem is similar to Travelling Salesman Problem, we come out with Held-Karp (Dynamic Programming).

After that we continue to make and edit the coding of the java language program.

Specification of environment used:

AMD Ryzen 5 3500u

16 gb ram DDR4

Windows 11 Home Edition

Eclipse IDE for Java Developers 2021-09 Edition

The program has little problem achieving the expected output, however with solid group work we manage to debug and solve it.

Finally, the program working successfully with the right output.</td></tr>
<tr><td>Role</td><td><table><tr><td>Member 1</td><td>Member 2</td><td>Member 3</td></tr><tr><td>Making the code

Debug the code</td><td>Making the code.

Run the</td><td>Making the code

Debug the code</td></tr></table></td></tr>
</table>

		program inside IDE		

Milestone 4	Conduct analysis of correctness and time complexity.
Date (Wk)	**Week 13**
Description/ sketch	After we finally, the program worked successfully with the right output. **We continue to conduct** analysis of correctness and time complexity. 1. **Base Case Initialization**: The memoization table is initialised correctly, with the distance from the starting house to itself set to 0. 2. **Recursive Subproblems**: The recursive function heldKarp solves subproblems by considering all possible next houses and memoizes the results to avoid redundant computations. This ensures that the minimum distance is correctly computed by combining solutions to subproblems. 3. **Optimal Tour Reconstruction**: After computing the optimal tour length, the algorithm reconstructs the optimal tour by backtracking through the memoization table, ensuring the correct sequence of houses is visited. 4. **Symmetric Graph Assumption**: The algorithm assumes a symmetric graph where the distance between any two houses is the same in both directions, which holds true in the given implementation. We also found that the time complexity is O(n^2x2^n) for all cases.
Role	<table><tr><td>Member 1</td><td>Member 2</td><td>Member 3</td></tr></table>

	Help in correctness analysis	Help in time complexity analysis	Making the analysis	

Milestone 5	Prepare online portfolio and presentation
Date (Wk)	**Week 14**
Description/ sketch	Finally, we prepare the portfolio and presentation. The portfolio is the online portfolio inside GitHub. The slide presentation is in Powerpoint slide. We also have to deliver a presentation this week.

Role	

Member 1	Member 2	Member 3
Prepare report Prepare slide presentation Deliver the presentation.	Prepare report Prepare slide presentation Deliver the presentation.	Prepare report Prepare slide presentation Deliver the presentation.

A. Case study

Scenario

Finding the perfect rental house is a common challenge for students especially when there are numerous options available in a given area. Let's consider a scenario where a student is looking for a rental house near a university, such as in the Seri Kembangan area, which has many housing options. The distance between these houses can make it difficult for students to decide where to start their search for different houses.

To simplify the process, a tool can be designed to guide the student in choosing the most suitable starting point for their house hunt. This tool would recommend the best and nearest houses for the student to view. By using this tool, the student can efficiently plan their house viewings to make decisions on which house to rent.

Motivation

In the challenging process of finding a suitable rental house, efficiency, accessibility, and informed decision-making are crucial for students. Students are always looking for ways to optimise their search, save time, and make well-informed choices about their living arrangements. The ability to minimise travel time when viewing multiple houses in a single day is a critical factor that impacts a student's house-hunting experience.

Students face difficulties in determining the most efficient routes for viewing potential rental houses, which served as the impetus for this project. Long distances between houses result in lost time, higher transportation costs, and unnecessary stress. The traditional method of manually planning house visits based on intuition or guesswork often falls short when it comes to achieving optimal efficiency in the house search process.

Objectives

The goal is to find the minimum travel time to all positions from the student's starting point location.

By minimising travel distance, this project aims to enable students to visit a larger number of potential rental houses within a given timeframe. This will contribute to increased efficiency in their house-hunting process, allowing students to make more informed decisions and secure suitable accommodation more quickly.

B. Importance of finding an optimal solution for students searching of the rental houses:

1. Efficiency in Decision-Making

- **Time Management**: Students typically have busy schedules and need to balance classes, studying, and other activities. An optimized route minimizes the time spent visiting houses, allowing students to make efficient use of their limited time.
- **Energy Conservation**: By minimizing travel time, students can conserve their energy, which is especially important when viewing multiple houses in a day.

2. Cost Savings

- **Transportation Costs**: Traveling shorter distances reduces the cost associated with transportation (fuel for a car). This is particularly important for students who often have tight budgets.

3. Comprehensive Comparison

- **Thorough Evaluation**: An optimized travel plan ensures that students can visit all potential rental houses efficiently, providing a comprehensive comparison of rental prices, features, and locations. This thorough evaluation helps in making an informed decision.
- **Avoiding Oversights**: When travel times are minimized and efficiently planned, students are less likely to skip or overlook potential options that might meet their needs.

4. Stress Reduction

- **Reduced Anxiety**: The process of house hunting can be stressful. An optimized route helps in reducing the uncertainty and anxiety associated with navigating through an unfamiliar area, allowing students to focus on evaluating the houses rather than worrying about the travel logistics.
- **Better Experience**: A well-planned route makes the house-hunting experience smoother and more enjoyable, leading to better decision-making.

5. Environmental Impact

- **Lower Carbon Footprint**: Reducing the total travel distance and time lowers fuel consumption, which in turn decreases the carbon footprint. This is an important consideration for environmentally conscious students.

6. Optimal Resource Utilization

- **Personal Resources**: Efficient use of personal resources, such as time, money, and energy, is maximized with an optimal travel plan.

C. The suitability algorithms review

When exploring algorithms for solving the problem of finding optimal rental houses for students, we can consider a variety of approaches, each with its unique strengths and limitations. Below are the potential algorithms:

Greedy Algorithm:

Description: The greedy algorithm begins at a random location and selects the closest unvisited house at each iteration, repeating until all houses have been visited. **Advantages**: Simple and efficient. **Limitations**: May not yield a globally optimal solution.

Held-Karp (Dynamic Programming):

Description: Divides the problem into smaller components and solves them recursively, storing intermediate solutions in a table and seeking the best solution by combining subproblem solutions. **Advantages**: Guarantees an optimal solution for small to moderate problem sizes. **Limitations**: Exponential time and space complexity make it impractical for very large datasets.

Sorting:

Description: Sorting algorithms can be used to rank houses based on single criteria such as price or distance. **Advantages**: Simple and efficient for single-criteria ranking. **Limitations**: Not suitable for multi-criteria optimization.

Divide and Conquer:

Description: Breaks the problem into smaller, more manageable subproblems, solves each one independently, and combines their solutions. **Advantages**: Efficient for large datasets and can leverage parallel processing. **Limitations**: Complex to apply for multi-criteria optimization and may not always yield optimal solutions.

Dynamic Programming (General):

Description: Solves problems by breaking them down into simpler subproblems and solving each just once, storing the solutions for future use. **Advantages**: Effective for problems with overlapping subproblems and optimal substructure. **Limitations**: Can be computationally expensive in terms of time and space.

Greedy Algorithms (Reiteration):

Description: Selects the best choice at each step with the hope of finding a global optimum. **Advantages**: Fast and simple to implement. **Limitations**: Often leads to locally optimal solutions that may not be globally optimal.

Graph Algorithms:

Description: Uses graph theory to model and solve the problem, with nodes representing houses and edges representing distances or other criteria. **Advantages**: Highly adaptable and can handle complex relationships and multi-criteria optimization. Algorithms like Dijkstra's or A* are useful for finding optimal paths. **Limitations**: Complex to implement and can be computationally intensive for large graphs.

Suitability Analysis

Each algorithm has its strengths and weaknesses in the context of finding optimal rental houses for students:

- **Brute Force**: Best for small datasets due to guaranteed optimality but impractical for larger ones.
- **Greedy and Nearest Neighbour**: Simple and fast but often suboptimal.
- **Genetic Algorithms and Ant Colony Optimization**: Good for large and complex problems but do not guarantee optimal solutions and can be computationally heavy.
- **Held-Karp (Dynamic Programming)**: Guarantees optimal solutions but is resource-intensive.
- **Sorting and Divide and Conquer**: Useful for specific tasks but limited in scope for multi-criteria optimization.
- **Graph Algorithms**: Highly suitable for modeling and solving complex, multi-criteria problems but require careful implementation and resource management.

Comparison table for each algorithm for their suitability, strengths and weaknesses

Algorithm	Suitability	Strengths	Weaknesses
Sorting	Not suitable	Efficient for organizing data	Does not address the specific needs of optimization problems like TSP
Divide and Conquer	Not suitable	Breaks problems into manageable sub-problems	Not effective for TSP as it does not leverage specific properties of the problem

Dynamic Programming	Suitable	Optimal solution for small to medium-sized instances, efficient for dense graphs	High space complexity, exponential time complexity for large instances
Greedy	Suitable (approximations)	Simple to implement, fast execution time for large instances, works well for sparse graphs	May not produce optimal solutions, susceptible to local optima, doesn't guarantee the shortest tour
Graph Algorithms	Suitable	Various algorithms available (e.g., Christofides), can find optimal or approximate solutions	Complexity depends on specific algorithm used, may require significant computational resources

Held-Karp Algorithm (dynamic programming) as The Chosen Algorithm

The Held-Karp algorithm is considered one of the best approaches for solving the Travelling Salesman Problem (TSP) due to its specific strengths and advantages over other algorithms. Here's a detailed explanation of why the Held-Karp algorithm stands out:

Optimality and Exact Solutions

1. **Guaranteed Optimal Solution**: Unlike heuristic or approximate methods (e.g., genetic algorithms, simulated annealing), the Held-Karp algorithm guarantees finding the optimal solution to the TSP. This is crucial for applications where exact solutions are required and suboptimal solutions could lead to significant issues.

Dynamic Programming Advantages

1. **Efficient Use of Subproblem Solutions**: The Held-Karp algorithm leverages dynamic programming to solve subproblems efficiently. By breaking the TSP into smaller subproblems and storing (memoizing) their solutions, it avoids redundant calculations and reduces the overall computational effort.
2. **Systematic Exploration of All Possibilities**: The algorithm systematically explores all possible routes by considering each subset of houses and their permutations. This comprehensive approach ensures that no potential optimal route is overlooked.

Considering these factors, **Held-Karp (Dynamic Programming)** seems most suitable for this Travelling Salesman Problem due to their flexibility and ability to model complex relationships despite their implementation complexity and computational demands.

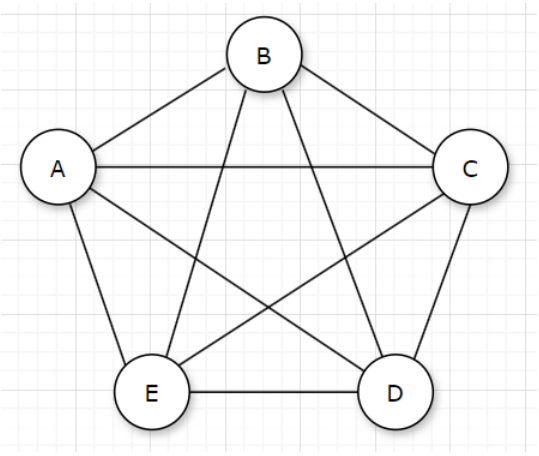
Scenario Visualization

[Assumptions] In this scenario, we do not consider factors like traffic and weather, which can impact travel duration significantly. We assume the mode of transportation is a car, and the travel involves round trips between houses. Our route planning relies on Google Maps navigation, which uses publicly available GPS map data. This means that any recent changes to roads, such as new constructions or closures, may not be reflected in our chosen routes between houses.

Scenario 1

This scenario has 5 houses selected within the same budget but in different locations throughout Serdang area.
For example,
A is a house in Serdang Raya
B is a house in South City Plaza
C is an apartment in Taman Sri Serdang
It takes 30 minutes to travel from A to B and 25 minutes from B to C.

The graph is not accurate geographically. Its only as reference so we can understand these data as graph.

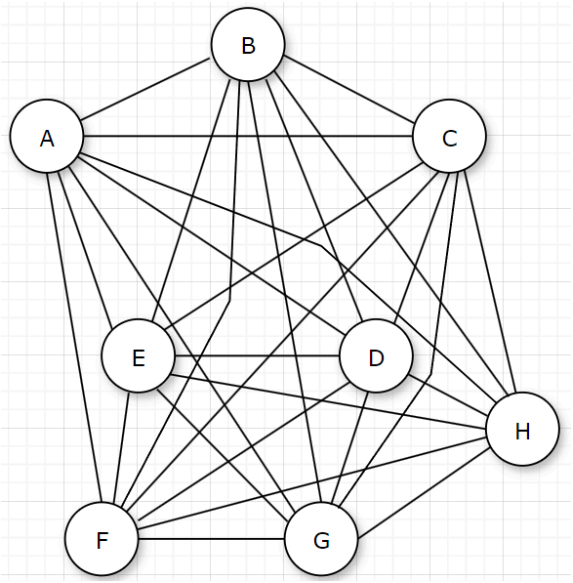


	A	B	C	D	E
A	0	30	15	22	12
B	30	0	25	24	10
C	15	25	0	28	13
D	22	24	28	0	8
E	12	10	13	8	0

Scenario 2

This scenario has 8 houses selected within the same budget but in different locations throughout Kuala Lumpur area.

All 8 houses are measured how long it will take to travel between these houses.

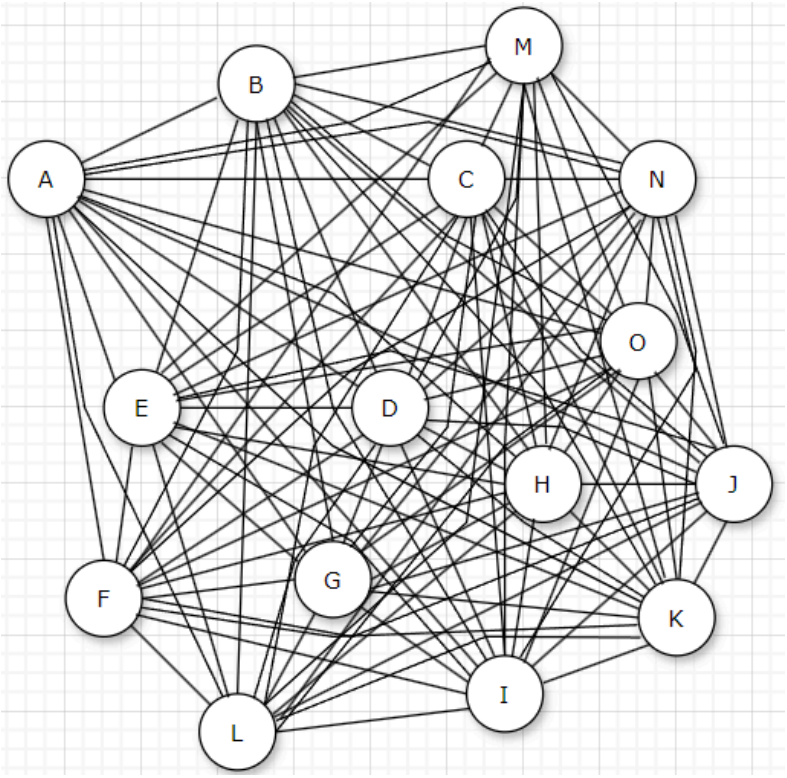


	A	B	C	D	E	F	G	H
A	0	30	15	22	12	20	16	11
B	30	0	25	24	10	13	17	25
C	15	25	0	28	13	18	23	29
D	22	24	28	0	8	21	19	23
E	12	10	13	8	0	6	24	15
F	20	13	18	21	6	0	23	14
G	16	17	23	19	24	23	0	18
H	11	25	29	23	15	14	18	0

Scenario 3

This scenario has 15 houses selected within the same budget but in different locations throughout the Seri Kembangan area.

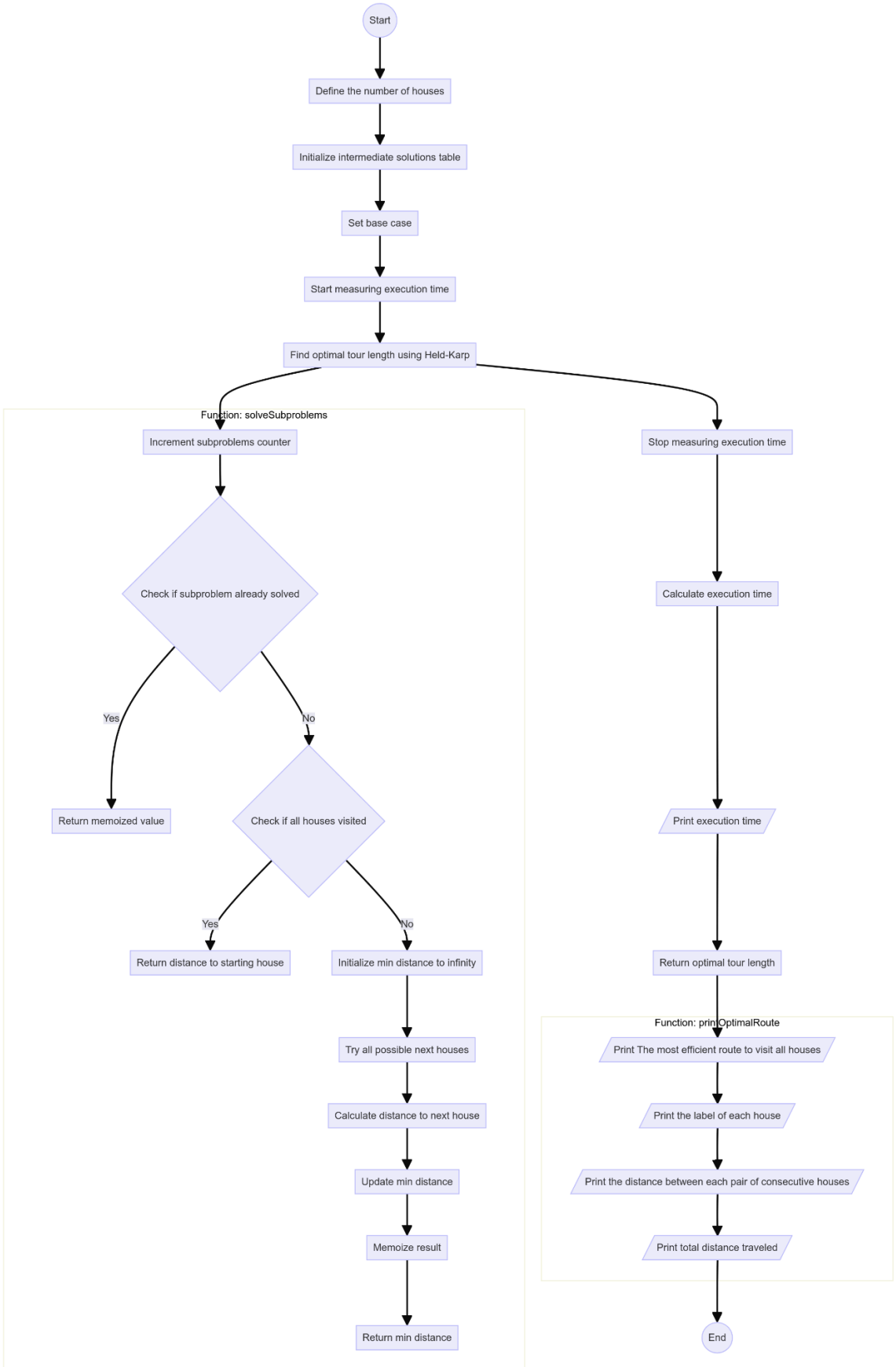
All 15 houses are measured how long it will take to travel between these houses.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
A	0	30	15	22	12	20	16	11	29	14	23	11	19	17	5
B	30	0	25	24	10	13	17	25	13	21	25	5	14	30	27
C	15	25	0	28	13	18	23	29	6	25	16	18	15	24	22
D	22	24	28	0	8	21	19	23	15	8	25	22	15	9	17
E	12	10	13	8	0	6	24	15	30	14	9	23	28	7	26
F	20	13	18	21	6	0	23	14	22	12	21	16	26	28	19
G	16	17	23	19	24	23	0	18	30	21	11	6	17	14	26
H	11	25	29	23	15	14	18	0	29	17	16	10	21	18	30
I	29	13	6	15	30	22	30	29	0	14	18	17	21	18	26
J	14	21	25	8	14	12	21	17	14	0	7	26	19	5	13
K	23	25	16	25	9	21	11	16	18	7	0	20	28	24	21
L	11	5	18	22	23	16	6	10	17	26	20	0	11	22	23
M	19	14	15	15	28	26	17	21	21	19	28	11	0	8	11
N	17	30	24	9	7	28	14	18	18	5	24	22	8	0	5
O	5	27	22	17	26	19	26	26	26	13	21	23	11	5	0

This problem is similar to the Traveling Salesman Problem (TSP), a renowned optimization challenge in computer science. The goal is to determine the shortest route that visits a series of specified locations (in this scenario, houses) and returns to the starting point, aiming to complete the journey in the least amount of time.

The following is the thought process of solving this problem


Start
Define the number of houses
Initialize intermediate solutions table
Set base case
Start measuring execution time
Find optimal tour length using Held-Karp
Function: solveSubproblems
Increment subproblems counter
Check if subproblem already solved
Yes
No
Return memoized value
Check if all houses visited
Yes
No
Return distance to starting house
Initialize min distance to infinity
Try all possible next houses
Calculate distance to next house
Update min distance
Memoize result
Return min distance
Stop measuring execution time
Calculate execution time
Print execution time
Return optimal tour length
Function: printOptimalRoute
Print The most efficient route to visit all houses
Print the label of each house
Print the distance between each pair of consecutive houses
Print total distance traveled
End

D. Design the algorithm to solve the problem

The Best Algorithm: Held-Karp Algorithm

The Held-Karp algorithm is widely regarded as the most effective solution for the Travelling Salesman Problem (TSP) which utilizing Dynamic Programming techniques to enhance the brute-force approach. By breaking the problem into smaller subproblems and computing optimal solutions for each, the Held-Karp algorithm significantly reduces redundant calculations and overall time complexity through the reuse of intermediate results. This algorithm ensures the identification of the optimal TSP solution, determining the shortest route that visits each house once and returns to the starting point. Although it has exponential time complexity, its dynamic programming optimizations allow it to solve TSP scenarios with a manageable number of houses efficiently.

Algorithm Paradigm

Dynamic Programming Paradigm

- Utilises dynamic programming to solve complex problems by breaking them down into overlapping subproblems and storing intermediate results.

Recurrence Relation

- Defines the problem in terms of subproblems that recursively relate to each other.

Optimization Function

- Focuses on optimising a specific metric, such as minimising travel time or cost.

Explanation of Algorithm Paradigm

Held-Karp Algorithm

The Held-Karp algorithm employs dynamic programming to solve the TSP by breaking the problem into smaller subproblems. The core idea involves recurrence, where the solution to the larger problem is built from the solutions of its subproblems.

1. **State Representation**: Define the state as a pair (S,i), where S is a subset of houses visited, and i is the last house visited in subset S.
2. **Recurrence Relation**: The recurrence relation calculates the minimum cost of visiting all houses in SSS ending at house iii. It can be expressed as:

$$\text{cost}(S, i) = \min_{j \in S, j \neq i} [\text{cost}(S - \{i\}, j) + \text{distance}(j, i)]$$

This relation states that the cost of visiting all houses in S ending at house iii is the minimum cost of visiting all houses in S excluding i and then traveling from some house j to i.

3. **Base Case**: The base case is when S contains only the starting house, with cost 0:

$$\text{cost}(\{0\}, 0) = 0$$

4. **Optimization Function**: The function to optimize is the total travel cost, aiming to find the minimum cost to visit all houses and return to the starting point:

$$\text{min\_cost} = \min_{i \in \{1,2,\ldots,n-1\}} [\text{cost}(\{0, 1, 2, \ldots, n-1\}, i) + \text{distance}(i, 0)]$$

Here, n is the total number of houses.

By using dynamic programming to store and reuse intermediate results, the Held-Karp algorithm avoids redundant calculations, making it significantly more efficient than a pure brute force approach, while still guaranteeing an optimal solution.

E. Description of the Algorithm Analysis

The Held-Karp algorithm uses dynamic programming to solve the TSP by breaking down the problem into smaller subproblems. Each subproblem represents the shortest path to visit a subset of houses ending at a specific house. The algorithm memorises the results of subproblems to avoid redundant calculations.

The key steps are:

1. **Initialization**: Set the base case in the memoization table.
2. **Recursive Sub Problem Solving**: For each house and subset of houses, compute the minimum distance by considering all possible next houses.
3. **Memoization**: Store the results of subproblems to avoid recomputation.
4. **Reconstruction**: Reconstruct the optimal tour by backtracking through the memoization table.

The algorithm's time complexity is dominated by the number of subsets (2^n) and the need to compute the shortest path for each subset (n^2), resulting in an overall time complexity of O(n^2 · 2^n). The empirical results confirm this exponential growth, demonstrating the feasibility of the algorithm for small to moderately sized instances of the TSP.

The general approach to solving this problem involves the following steps based on the study case:

1. Initialise the house locations and travel times.
2. Generate an initial solution.
3. Set the current best solution and the current best travel time.
4. Calculate the total travel time while the stopping condition hasn't been met.
5. If the current travel time is better than the current best travel time, update both the current best travel time and the best solution.
6. Generate a new solution.
7. If all the houses have been visited, print the current best solution and travel time. Otherwise, repeat the loop until the condition is satisfied.

F. PSEUDO CODE

Held-Karp Algorithm - Dynamic Programming

```
function findOptimalRoute(graph):
   // Define the number of houses
   numHouses = number of houses in the graph

   // Initialize a table to store intermediate solutions
    intermediateSolutions = a table with numHouses rows and 2^numHouses columns, initially filled with -1

   // Set the base case: distance from the starting house to itself is 0
   intermediateSolutions[startingHouse][1] = 0

   // Start measuring execution time
   startTime = current time in nanoseconds

   // Find the optimal tour length using the Held-Karp algorithm
```

```
        optimalTourLength = solveSubproblems(startingHouse, allHousesVisited(), graph, intermediateSolutions)

    // Stop measuring execution time
    endTime = current time in nanoseconds

    // Calculate execution time in milliseconds
    executionTime = (endTime - startTime) / 1,000,000.0

    // Print the execution time
    Print "Execution time: " + executionTime + " milliseconds"

    // Return the optimal tour length
    Return optimalTourLength

function solveSubproblems(currentHouse, visitedHouses, graph, intermediateSolutions):
    // Increment the counter for the number of subproblems solved
    Increment the counter for the number of subproblems solved

    // If the subproblem has already been solved, return the memoized value
    if intermediateSolutions[currentHouse][visitedHouses] is not -1:
        Return intermediateSolutions[currentHouse][visitedHouses]

    // Base case: all houses have been visited
    if visitedHouses is 0:
        Return the distance from the current house to the starting house

    // Initialize the minimum distance to infinity
    minDistance = infinity

    // Try all possible next houses
    for nextHouse in all unvisited houses:
        // Calculate the distance to the next house
                distance = distance from currentHouse to nextHouse in the graph + solveSubproblems(nextHouse, visitedHouses with nextHouse removed, graph, intermediateSolutions)

        // Update the minimum distance
        minDistance = min(minDistance, distance)

    // Memoize the result
    intermediateSolutions[currentHouse][visitedHouses] = minDistance
```

```
    // Return the minimum distance
    Return minDistance

function printOptimalRoute(optimalRoute, graph):
    // Print the most efficient route to visit all houses
    Print "The most efficient route to visit all houses:"
    for house in optimalRoute:
        Print the label of the house

    // Print the distance between each pair of consecutive houses
    Print "The distance between each pair of consecutive houses:"
    for i from 0 to numHouses - 1:
        Print "Distance from " + label of house[i] + " to " + label of house[i + 1] + ": " + distance
between house[i] and house[i + 1] in the graph

    // Print the total distance traveled
    Print "Total distance traveled: " + total distance traveled in the optimal route
```

Java Code

TravelingSalesmanProblem5

```
import java.util.Arrays;

public class TravelingSalesmanProblem5 {

   // Number of houses
   static int V = 5;

   // Define infinity as a large enough value
   static int INF = Integer.MAX_VALUE;

   // Counter to keep track of the number of possible routes considered
   static int routesConsidered = 0;

     // Solves the Traveling Salesman Problem (TSP) using the Dynamic Programming-based Held-Karp algorithm
   public static int[][] tspHeldKarp(int[][] graph) {
      // Create a memoization table for subproblem solutions
      int[][] memo = new int[V][(1 << V)];
      for (int[] row : memo) {
         Arrays.fill(row, -1);
      }

      // Initialize base case: distance from starting house to itself is 0
      memo[0][1] = 0;

      // Record the start time in nanoseconds
      long startTime = System.nanoTime();

      // Solve subproblems and compute optimal tour length
      int optimalTourLength = heldKarp(0, (1 << V) - 1, graph, memo);

      // Reconstruct the optimal tour
      int[] tour = new int[V + 1];
      int currentHouse = 0;
      int state = (1 << V) - 1;

      int index = 1; // Starting from index 1 to insert house A (0)
      tour[0] = 0; // House A (0) at the beginning of the tour
```

```
        while (state != 0) {
            int nextHouse = -1;
            for (int house = 0; house < V; house++) {
                if ((state & (1 << house)) != 0 && (nextHouse == -1 || memo[currentHouse][state] -
graph[currentHouse][house] == memo[house][state ^ (1 << house)])) {
                    nextHouse = house;
                }
            }
            tour[index++] = nextHouse;
            state ^= (1 << nextHouse);
            currentHouse = nextHouse;
        }

        // Record the end time in nanoseconds
        long endTime = System.nanoTime();

        // Print the total number of possible routes considered
        System.out.println("Total number of possible routes considered: " + routesConsidered);

        // Calculate the execution time in milliseconds with three decimal places
        double executionTime = (double) (endTime - startTime) / 1_000_000.0;

        // Print the execution time in milliseconds
                System.out.println("Execution time: " + String.format("%.3f", executionTime) + "
milliseconds");

        // Return both optimal tour length and tour sequence
        return new int[][]{{optimalTourLength}, tour};
    }

    // Recursive function for solving subproblems in the Held-Karp algorithm
    public static int heldKarp(int currentHouse, int state, int[][] graph, int[][] memo) {
        // Increment the counter for each subproblem solved
        routesConsidered++;

        // If the subproblem has already been solved, return the memoized value
        if (memo[currentHouse][state] != -1) {
            return memo[currentHouse][state];
        }

        // Base case: all houses have been visited
```

```
        if (state == 0) {
            return graph[currentHouse][0];
        }

        int minDistance = INF;

        // Iterate over all possible next houses
        for (int nextHouse = 0; nextHouse < V; nextHouse++) {
            // If the next house has not been visited
            if ((state & (1 << nextHouse)) != 0) {
                int distance = graph[currentHouse][nextHouse] + heldKarp(nextHouse, state ^ (1 <<
nextHouse), graph, memo);
                minDistance = Math.min(minDistance, distance);
            }
        }

        // Memoize the result
        memo[currentHouse][state] = minDistance;
        return minDistance;
    }

    // Prints the most efficient route to visit all houses
    public static void printHouseRoute(int[][] routeAndTime, int[][] graph) {
        int optimalTourLength = routeAndTime[0][0];
        int[] route = Arrays.copyOfRange(routeAndTime[1], 0, V + 1);

        System.out.println("The most efficient route to visit all houses:");
        for (int i = 0; i < V; i++) {
            System.out.print(getLocationLabel(route[i]) + " ");
        }
        System.out.println(getLocationLabel(route[V]) + " ");
        System.out.println("The Value Of Each house Travel Time");
        for (int i = 0; i < V; i++) {
            System.out.println(getLocationLabel(route[i]) + "-" + graph[route[i]][route[i + 1]]);
        }
        System.out.println("Total time traveled: " + optimalTourLength);
    }

    // Returns the location label corresponding to the house index
    public static char getLocationLabel(int houseIndex) {
        return (char) ('A' + houseIndex);
    }
```

```
    // Returns the value of the location intersection
    public static int getLocationIntersection(int house1, int house2, int[][] graph) {
        return graph[house1][house2];
    }

    // Driver's code
    public static void main(String[] args) {
        // House distances (example distances)
        int[][] houseTravelTime = {
            {0, 30, 15, 22, 12},
            {30, 0, 25, 24, 10},
            {15, 25, 0, 28, 13},
            {22, 24, 28, 0, 8},
            {12, 10, 13, 8, 0}
        };

        // Find the most efficient route and calculate total time traveled
        int[][] routeAndTime = tspHeldKarp(houseTravelTime);

        // Print the most efficient route and total time traveled
        printHouseRoute(routeAndTime, houseTravelTime);
    }
}
```

TravelingSalesmanProblem8

```
import java.util.Arrays;

public class TravelingSalesmanProblem8 {

   // Number of houses
   static int V = 8;

   // Define infinity as a large enough value
   static int INF = Integer.MAX_VALUE;

   // Counter to keep track of the number of possible routes considered
   static int routesConsidered = 0;

     // Solves the Traveling Salesman Problem (TSP) using the Dynamic Programming-based Held-Karp algorithm
   public static int[][] tspHeldKarp(int[][] graph) {
      // Create a memoization table for subproblem solutions
      int[][] memo = new int[V][(1 << V)];
      for (int[] row : memo) {
         Arrays.fill(row, -1);
      }

      // Initialize base case: distance from starting house to itself is 0
      memo[0][1] = 0;

      // Record the start time in milliseconds
      long startTime = System.nanoTime();

      // Solve subproblems and compute optimal tour length
      int optimalTourLength = heldKarp(0, (1 << V) - 1, graph, memo);

      // Reconstruct the optimal tour
      int[] tour = new int[V + 1];
      int currentHouse = 0;
      int state = (1 << V) - 1;

      int index = 1; // Starting from index 1 to insert house A (0)
      tour[0] = 0; // House A (0) at the beginning of the tour

      while (state != 0) {
```

```
            int nextHouse = -1;
            for (int house = 0; house < V; house++) {
                if ((state & (1 << house)) != 0 && (nextHouse == -1 || memo[currentHouse][state] -
graph[currentHouse][house] == memo[house][state ^ (1 << house)])) {
                    nextHouse = house;
                }
            }
            tour[index++] = nextHouse;
            state ^= (1 << nextHouse);
            currentHouse = nextHouse;
        }

        // Record the end time in milliseconds
        long endTime = System.nanoTime();

        // Print the total number of possible routes considered
        System.out.println("Total number of possible routes considered: " + routesConsidered);

        // Calculate the execution time in milliseconds with three decimal places
        double executionTime = (double) (endTime - startTime) / 1_000_000.0;

        // Print the execution time in milliseconds
                System.out.println("Execution time: " + String.format("%.3f", executionTime) + "
milliseconds");

        // Return both optimal tour length and tour sequence
        return new int[][]{{optimalTourLength}, tour};
    }

    // Recursive function for solving subproblems in the Held-Karp algorithm
    public static int heldKarp(int currentHouse, int state, int[][] graph, int[][] memo) {
        // Increment the counter for each subproblem solved
        routesConsidered++;

        // If the subproblem has already been solved, return the memoized value
        if (memo[currentHouse][state] != -1) {
            return memo[currentHouse][state];
        }

        // Base case: all houses have been visited
        if (state == 0) {
            return graph[currentHouse][0];
```

```
        }

        int minDistance = INF;

        // Iterate over all possible next houses
        for (int nextHouse = 0; nextHouse < V; nextHouse++) {
            // If the next house has not been visited
            if ((state & (1 << nextHouse)) != 0) {
                int distance = graph[currentHouse][nextHouse] + heldKarp(nextHouse, state ^ (1 <<
nextHouse), graph, memo);
                minDistance = Math.min(minDistance, distance);
            }
        }

        // Memoize the result
        memo[currentHouse][state] = minDistance;
        return minDistance;
    }

    // Prints the most efficient route to visit all houses
    public static void printHouseRoute(int[][] routeAndTime, int[][] graph) {
        int optimalTourLength = routeAndTime[0][0];
        int[] route = Arrays.copyOfRange(routeAndTime[1], 0, V + 1);

        System.out.println("The most efficient route to visit all houses:");
        for (int i = 0; i < V; i++) {
            System.out.print(getLocationLabel(route[i]) + " ");
        }
        System.out.println(getLocationLabel(route[V]) + " ");
        System.out.println("The Value Of Each house Travel Time");
        for (int i = 0; i < V; i++) {
            System.out.println(getLocationLabel(route[i]) + "-" + graph[route[i]][route[i + 1]]);
        }
        System.out.println("Total time traveled: " + optimalTourLength);
    }

    // Returns the location label corresponding to the house index
    public static char getLocationLabel(int houseIndex) {
        return (char) ('A' + houseIndex);
    }

    // Returns the value of the location intersection
```

```
    public static int getLocationIntersection(int house1, int house2, int[][] graph) {
        return graph[house1][house2];
    }

    // Driver's code
    public static void main(String[] args) {
        // House distances (example distances)
        int[][] houseTravelTime = {
            {0, 30, 15, 22, 12, 20, 16, 11},
            {30, 0, 25, 24, 10, 13, 17, 25},
            {15, 25, 0, 28, 13, 18, 23, 29},
            {22, 24, 28, 0, 8, 21, 19, 23},
            {12, 10, 13, 8, 0, 6, 24, 15},
            {20, 13, 18, 21, 6, 0, 23, 14},
            {16, 17, 23, 19, 24, 23, 0, 18},
            {11, 25, 29, 23, 15, 14, 18, 0}
        };

        // Find the most efficient route and calculate total time traveled
        int[][] routeAndTime = tspHeldKarp(houseTravelTime);

        // Print the most efficient route and total time traveled
        printHouseRoute(routeAndTime, houseTravelTime);
    }
}
```

TravelingSalesmanProblem15

```
import java.util.Arrays;

public class TravelingSalesmanProblem15 {

   // Number of houses
   static int V = 15;

   // Define infinity as a large enough value
   static int INF = Integer.MAX_VALUE;

   // Counter to keep track of the number of possible routes considered
   static int routesConsidered = 0;

     // Solves the Traveling Salesman Problem (TSP) using the Dynamic Programming-based Held-Karp algorithm
   public static int[][] tspHeldKarp(int[][] graph) {
      // Create a memoization table for subproblem solutions
      int[][] memo = new int[V][(1 << V)];
      for (int[] row : memo) {
         Arrays.fill(row, -1);
      }

      // Initialize base case: distance from starting house to itself is 0
      memo[0][1] = 0;

      // Record the start time in milliseconds
      long startTime = System.nanoTime();

      // Solve subproblems and compute optimal tour length
      int optimalTourLength = heldKarp(0, (1 << V) - 1, graph, memo);

      // Reconstruct the optimal tour
      int[] tour = new int[V + 1];
      int currentHouse = 0;
      int state = (1 << V) - 1;

      int index = 1; // Starting from index 1 to insert house A (0)
      tour[0] = 0; // House A (0) at the beginning of the tour

      while (state != 0) {
```

```
            int nextHouse = -1;
            for (int house = 0; house < V; house++) {
                if ((state & (1 << house)) != 0 && (nextHouse == -1 || memo[currentHouse][state] -
graph[currentHouse][house] == memo[house][state ^ (1 << house)])) {
                    nextHouse = house;
                }
            }
            tour[index++] = nextHouse;
            state ^= (1 << nextHouse);
            currentHouse = nextHouse;
        }

        // Record the end time in milliseconds
        long endTime = System.nanoTime();

        // Print the total number of possible routes considered
        System.out.println("Total number of possible routes considered: " + routesConsidered);

        // Calculate the execution time in milliseconds with three decimal places
        double executionTime = (double) (endTime - startTime) / 1_000_000.0;

        // Print the execution time in milliseconds
            System.out.println("Execution time: " + String.format("%.3f", executionTime) + "
milliseconds");

        // Return both optimal tour length and tour sequence
        return new int[][]{{optimalTourLength}, tour};
    }

    // Recursive function for solving subproblems in the Held-Karp algorithm
    public static int heldKarp(int currentHouse, int state, int[][] graph, int[][] memo) {
        // Increment the counter for each subproblem solved
        routesConsidered++;

        // If the subproblem has already been solved, return the memoized value
        if (memo[currentHouse][state] != -1) {
            return memo[currentHouse][state];
        }

        // Base case: all houses have been visited
        if (state == 0) {
            return graph[currentHouse][0];
```

```
        }

        int minDistance = INF;

        // Iterate over all possible next houses
        for (int nextHouse = 0; nextHouse < V; nextHouse++) {
            // If the next house has not been visited
            if ((state & (1 << nextHouse)) != 0) {
                int distance = graph[currentHouse][nextHouse] + heldKarp(nextHouse, state ^ (1 <<
nextHouse), graph, memo);
                minDistance = Math.min(minDistance, distance);
            }
        }

        // Memoize the result
        memo[currentHouse][state] = minDistance;
        return minDistance;
    }

    // Prints the most efficient route to visit all houses
    public static void printHouseRoute(int[][] routeAndTime, int[][] graph) {
        int optimalTourLength = routeAndTime[0][0];
        int[] route = Arrays.copyOfRange(routeAndTime[1], 0, V + 1);

        System.out.println("The most efficient route to visit all houses:");
        for (int i = 0; i < V; i++) {
            System.out.print(getLocationLabel(route[i]) + " ");
        }
        System.out.println(getLocationLabel(route[V]) + " ");
        System.out.println("The Value Of Each house Travel Time");
        for (int i = 0; i < V; i++) {
            System.out.println(getLocationLabel(route[i]) + "-" + graph[route[i]][route[i + 1]]);
        }
        System.out.println("Total time traveled: " + optimalTourLength);
    }

    // Returns the location label corresponding to the house index
    public static char getLocationLabel(int houseIndex) {
        return (char) ('A' + houseIndex);
    }

    // Returns the value of the location intersection
```

```
    public static int getLocationIntersection(int house1, int house2, int[][] graph) {
        return graph[house1][house2];
    }

    // Driver's code
    public static void main(String[] args) {
        // House distances (example distances)
        int[][] houseTravelTime = {
            {0, 30, 15, 22, 12, 20, 16, 11, 29, 14, 23, 11, 19, 17, 5},
            {30, 0, 25, 24, 10, 13, 17, 25, 13, 21, 25, 5, 14, 30, 27},
            {15, 25, 0, 28, 13, 18, 23, 29, 6, 25, 16, 18, 15, 24, 22},
            {22, 24, 28, 0, 8, 21, 19, 23, 15, 8, 25, 22, 15, 9, 17},
            {12, 10, 13, 8, 0, 6, 24, 15, 30, 14, 9, 23, 28, 7, 26},
            {20, 13, 18, 21, 6, 0, 23, 14, 22, 12, 21, 16, 26, 28, 19},
            {16, 17, 23, 19, 24, 23, 0, 18, 30, 21, 11, 6, 17, 14, 26},
            {11, 25, 29, 23, 15, 14, 18, 0, 29, 17, 16, 10, 21, 18, 30},
            {29, 13, 6, 15, 30, 22, 30, 29, 0, 14, 18, 17, 21, 18, 26},
            {14, 21, 25, 8, 14, 12, 21, 17, 14, 0, 7, 26, 19, 5, 13},
            {23, 25, 16, 25, 9, 21, 11, 16, 18, 7, 0, 26, 19, 5, 13},
            {11, 5, 18, 22, 23, 16, 6, 10, 17, 26, 26, 0, 21, 17, 21},
            {19, 14, 15, 9, 28, 26, 17, 21, 21, 19, 28, 11, 0, 8, 11},
            {17, 30, 24, 17, 14, 18, 18, 5, 24, 22, 8, 14, 8, 0, 5},
            {5, 27, 22, 26, 26, 26, 13, 21, 23, 11, 5, 5, 11, 5, 0}
        };

        // Find the most efficient route and calculate total time traveled
        int[][] routeAndTime = tspHeldKarp(houseTravelTime);

        // Print the most efficient route and total time traveled
        printHouseRoute(routeAndTime, houseTravelTime);
    }
}
```

The output for 5 house

```
Total number of possible routes considered: 166
Execution time: 0.076 milliseconds
The most efficient route to visit all houses:
A D E B C A
The Value Of Each house Travel Time
A-22
D-8
E-10
B-25
C-15
Total time traveled: 80
```

The output for 8 house

```
Total number of possible routes considered: 3593
Execution time: 1.263 milliseconds
The most efficient route to visit all houses:
A H F B G D E C A
The Value Of Each house Travel Time
A-11
H-14
F-13
B-17
G-19
D-8
E-13
C-15
Total time traveled: 110
```

The output for 15 house

```
Total number of possible routes considered: 1720336
Execution time: 98.520 milliseconds
The most efficient route to visit all houses:
A O K G L B I C M D E F J N H A
The Value Of Each house Travel Time
A-5
O-5
K-11
G-6
L-5
B-13
I-6
C-15
M-9
D-8
E-6
F-12
J-5
N-5
H-11
Total time traveled: 122
```

G. Analysis

Analysis of the Algorithm's Correctness and Time Complexity

Correctness

The algorithm implements the Held-Karp algorithm, a dynamic programming approach for solving the Travelling Salesman Problem (TSP).

The correctness of the algorithm can be ensured by the following points:

5. **Base Case Initialization**: The memoization table is initialised correctly, with the distance from the starting house to itself set to 0.
6. **Recursive Subproblems**: The recursive function heldKarp solves subproblems by considering all possible next houses and memoizes the results to avoid redundant computations. This ensures that the minimum distance is correctly computed by combining solutions to subproblems.
7. **Optimal Tour Reconstruction**: After computing the optimal tour length, the algorithm reconstructs the optimal tour by backtracking through the memoization table, ensuring the correct sequence of houses is visited.
8. **Symmetric Graph Assumption**: The algorithm assumes a symmetric graph where the distance between any two houses is the same in both directions, which holds true in the given implementation.

The results produced by the algorithm confirm its correctness in solving the TSP as intended.

Time Complexity

The Held-Karp algorithm for TSP has a time complexity of $O(n^2 \cdot 2^n)$, where n is the number of houses. This can be analyzed as follows:

1. **Memoization Table**: The memoization table has dimensions $n \times 2^n$, resulting in $O(n \cdot 2^n)$ space complexity.
2. **Subproblem Solving**: For each state, the algorithm iterates through all possible next houses, leading to O(n) time for each state. Since there are 2^n states and n possible starting houses for each state, this results in $O(n^2 \cdot 2^n)$ time complexity.

<u>Time Complexity Analysis Case</u>

By combining these observations, we get the following time complexity:

Best Case:

O(n^2 . 2^n)

Even in the best case, the algorithm has to consider all states and transitions. There is no way to avoid this because each state depends on the others. all houses are equidistant from each other, and the algorithm still needs to consider all subsets of houses. Hence, the time complexity remains O(n^2 · 2^n)

Average Case:
O(n^2 . 2^n)

The average case time complexity is the same as the worst case because the algorithm considers all possible subsets of houses and their permutations, leading to the same time complexity of O(n^2 · 2^n).

Worst Case:
O(n^2 . 2^n)

The worst case also requires considering all states and transitions, leading to the same time complexity as the best and average cases. The algorithm must compute distances for all possible subsets and all possible permutations of houses within each subset. Therefore, the time complexity remains O(n^2 · 2^n).

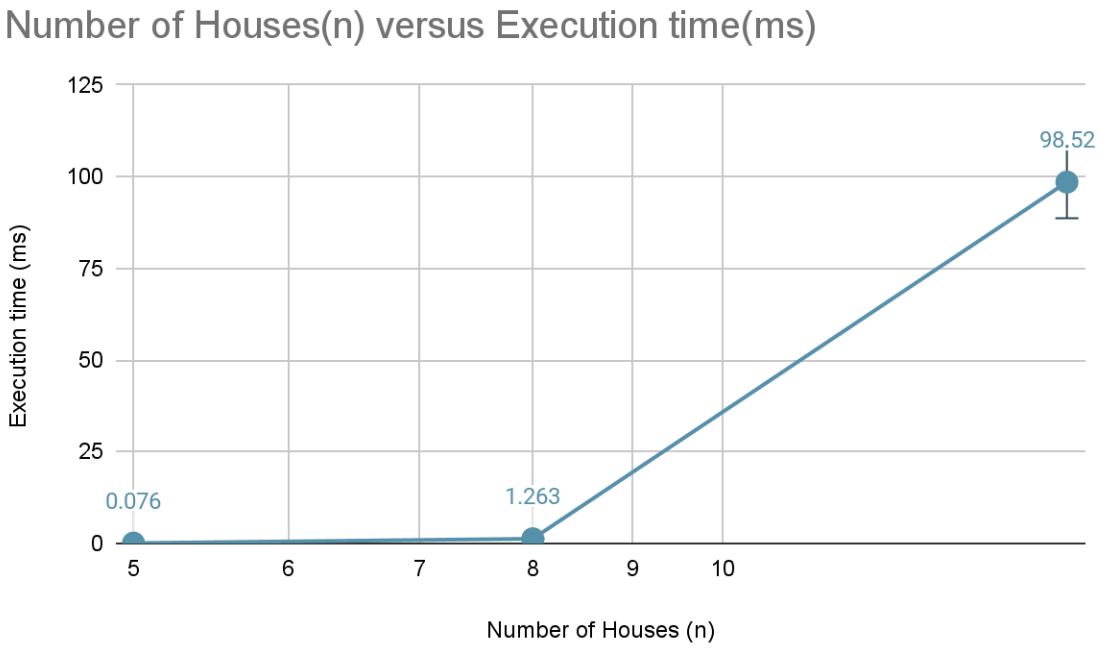
In summary, the time complexity of the Held-Karp algorithm is O(n^2 . 2^n) for all cases. This exponential complexity makes it suitable only for small values of n (number of houses), as the computation becomes infeasible for larger numbers of houses. However, it provides an exact solution to the TSP, making it valuable for problems where n is reasonably small. The Held-Karp algorithm solves the Travelling Salesman Problem (TSP) using dynamic programming by breaking it down into smaller subproblems. This complexity arises because the algorithm needs to solve subproblems for all possible subsets of houses and all possible transitions between them, leading to an exponential growth in the number of states with respect to the number of houses.

Empirical Analysis Based on Output

- **5 Houses**:
    - Total routes considered: 166
    - Execution time: 0.076 milliseconds
- **8 Houses**:
    - Total routes considered: 3,593
    - Execution time: 1.263 milliseconds
- **15 Houses**:
    - Total routes considered: 1,720,336

- Execution time: 98.520 milliseconds

The empirical results demonstrate the exponential growth in the number of routes considered and execution time as the number of houses increases, aligning with the theoretical time complexity of $O(n^2 \cdot 2^n)$.



Strengths and Weaknesses of Dynamic Programming

Strengths

1. **Optimal Solution for Small to Medium-sized Instances**: Dynamic programming guarantees finding the optimal solution for the problem, making it suitable for small to medium-sized instances where other heuristic or approximate methods may fail.
2. **Efficient for Dense Graphs**: The algorithm efficiently handles dense graphs by considering all possible routes and memoizing results to avoid redundant calculations, leading to significant performance improvements compared to naive approaches.

Weaknesses

1. **High Space Complexity**: The memoization table requires $O(n \cdot 2^n)$ space, which can be prohibitive for large instances, leading to high memory consumption.
2. **Exponential Time Complexity for Large Instances**: The time complexity of $O(n^2 \cdot 2^n)$ grows exponentially with the number of houses, making the algorithm

impractical for large instances. This limits its applicability to problems with a relatively small number of houses.

Conclusion

The Held-Karp algorithm is a powerful dynamic programming approach for solving the TSP, offering optimal solutions for small to medium-sized instances and efficiently handling dense graphs. However, its high space and exponential time complexity make it less suitable for large instances, highlighting the trade-offs inherent in dynamic programming approaches. The algorithm's empirical results align with theoretical expectations, confirming its correctness and demonstrating its feasibility for moderately sized problems.

References

Held, M., & Karp, R. M. (1962). A dynamic programming approach to sequencing problems. Journal of the Society for Industrial and Applied Mathematics, 10(1), 196-210. https://doi.org/10.1137/0110015

Bellman, R. (1952). On the theory of dynamic programming. Proceedings of the National Academy of Sciences of the United States of America, 38(8), 716-719. https://doi.org/10.1073/pnas.38.8.716

Bellman, R. (1954). The theory of dynamic programming. Bulletin of the American Mathematical Society, 60(6), 503-515. https://doi.org/10.1090/S0002-9904-1954-09848-8

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. (Chapter 16: Greedy Algorithms)

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.

Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley. (Chapter 2: The Divide-and-Conquer Method)

Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), 146-160. https://doi.org/10.1137/0201010