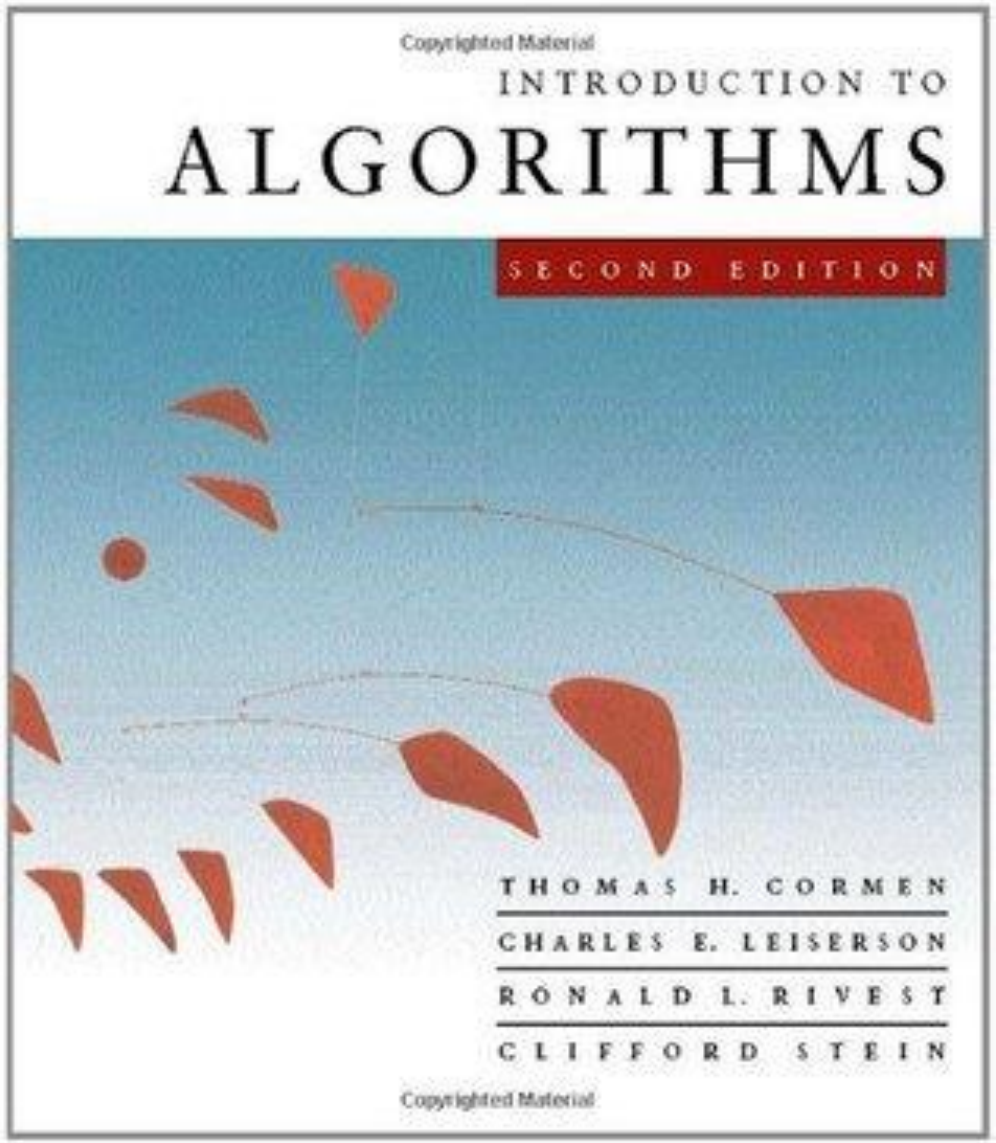
Fall 2024

Prof. Alaa Sheta

Algorithms



Exhaustive Search

- **Disadvantages:**
  - **Inefficient for Large Problems**: The method can become computationally expensive due to its exponential time complexity for larger input sizes.
  - **Memory Usage**: It can consume much memory for large datasets.

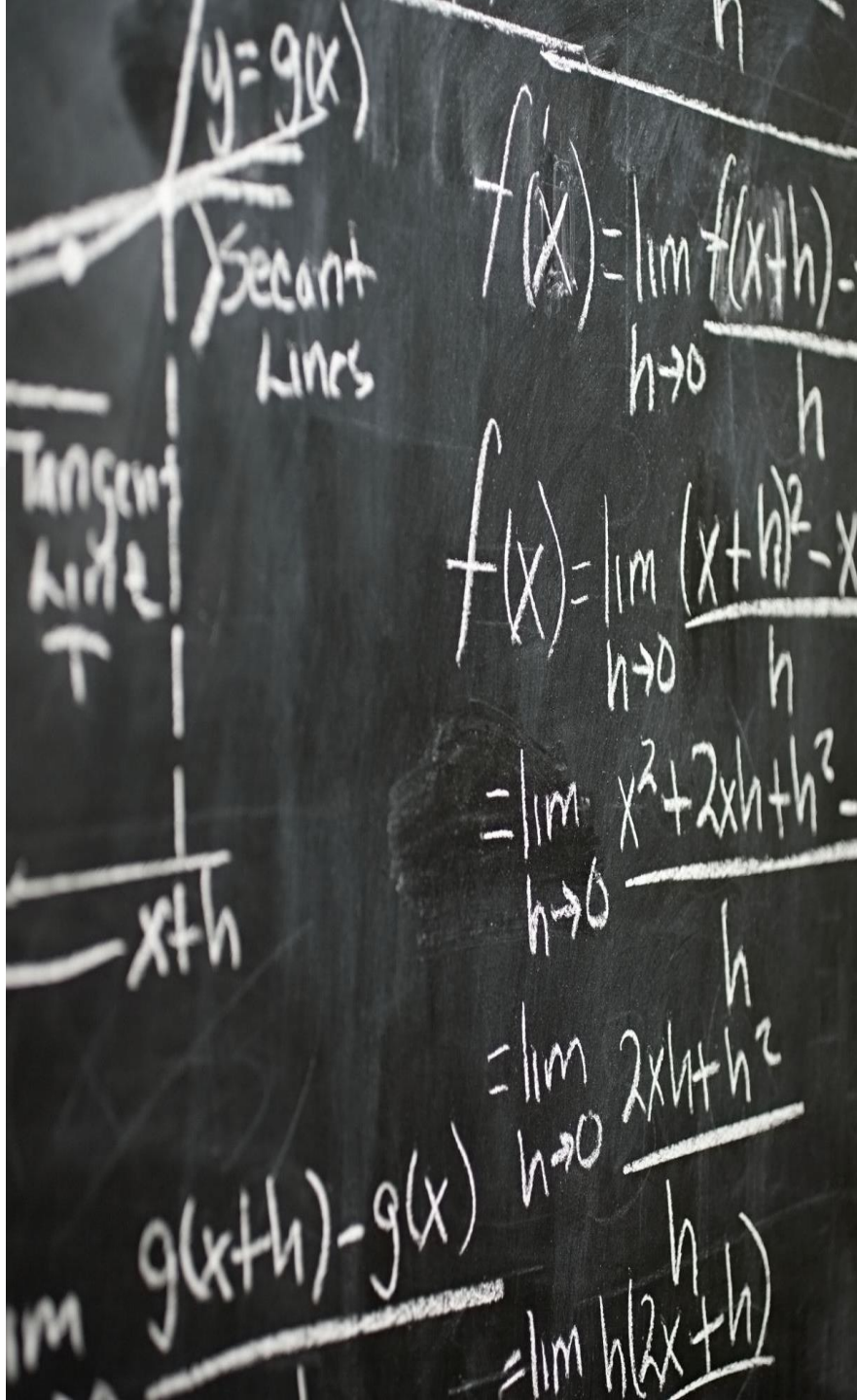
Traveling Salesman Problem (TSP)

- The **Traveling Salesman Problem (TSP)** is a well-known combinatorial optimization problem in which a salesman must visit a set of cities, starting from a home city, visiting each city exactly once, and then returning to the starting city. The goal is to find the shortest possible route that covers all cities.
- **Key Characteristics:**

1. **Input**: A list of cities and the distances (or costs) between each pair of cities.
2. **Objective**: Find the shortest route (or tour) that visits every city exactly once and returns to the starting city.
3. **Constraints**:
    1. Each city must be visited exactly once.
    2. The salesman must return to the starting city.

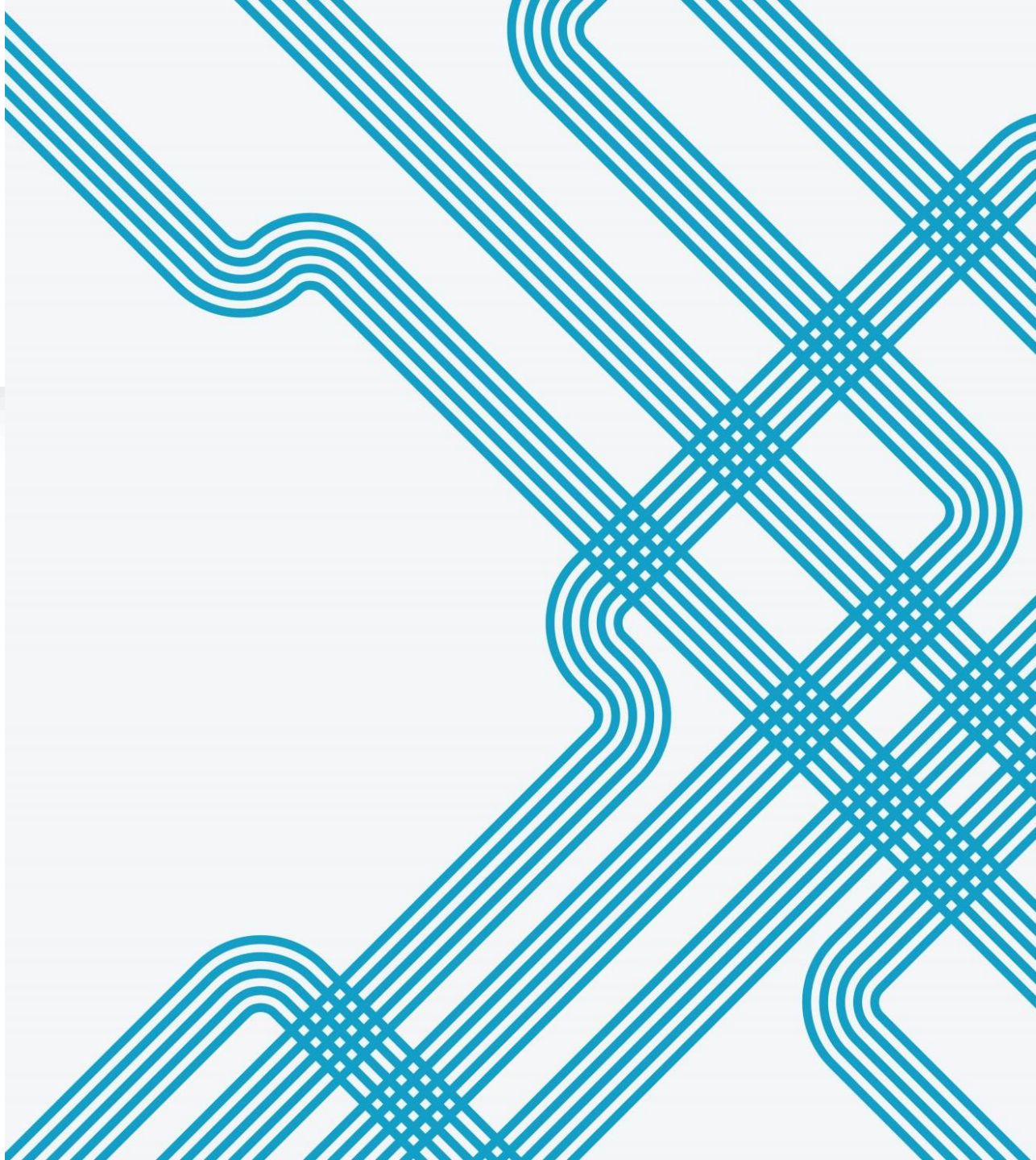
TSP as a Computational Problem

- **NP-Hard**: TSP belongs to the class of NP-hard problems, meaning that there is no known polynomial-time solution for the general case of TSP.
- **Brute Force Solution**: The brute force approach generates all possible routes (permutations) and selects the shortest one. This has a time complexity of **O(n!)**, where **n** is the number of cities.



Real-Life Applications of TSP

1. **Logistics and Delivery**: Optimizing delivery routes for trucks, drones, or couriers to minimize time and fuel.
2. **Manufacturing**: Reducing the time for machines in production lines to move between different points.
3. **Circuit Design**: Minimizing the length of wires in the circuit board layout.



Example of TSP

Consider a salesman who needs to visit 4 cities: A, B, C, and D. The distance between each pair of cities is given in the table below:

	A	B	C	D
A	0	10	15	20
B	10	0	35	25
C	15	35	0	30
D	20	25	30	0

Steps for Brute Force Approach:

1. List all possible routes (permutations of cities) starting and ending at city A:
    - Route 1: A → B → C → D → A
    - Route 2: A → B → D → C → A
    - Route 3: A → C → B → D → A
    - Route 4: A → C → D → B → A
    - Route 5: A → D → B → C → A
    - Route 6: A → D → C → B → A

2. Calculate the total distance for each route:

   - **Route 1**: A → B → C → D → A = 10 + 35 + 30 + 20 = 95
   - **Route 2**: A → B → D → C → A = 10 + 25 + 30 + 15 = 80
   - **Route 3**: A → C → B → D → A = 15 + 35 + 25 + 20 = 95
   - **Route 4**: A → C → D → B → A = 15 + 30 + 25 + 10 = 80
   - **Route 5**: A → D → B → C → A = 20 + 25 + 35 + 15 = 95
   - **Route 6**: A → D → C → B → A = 20 + 30 + 35 + 10 = 95

3. Identify the shortest route:

   - The shortest routes are **Route 2** and **Route 4**, both with a total distance of **80**.

Optimal Solution:

The optimal route for this instance of TSP is either:

- A → B → D → C → A, or
- A → C → D → B → A.

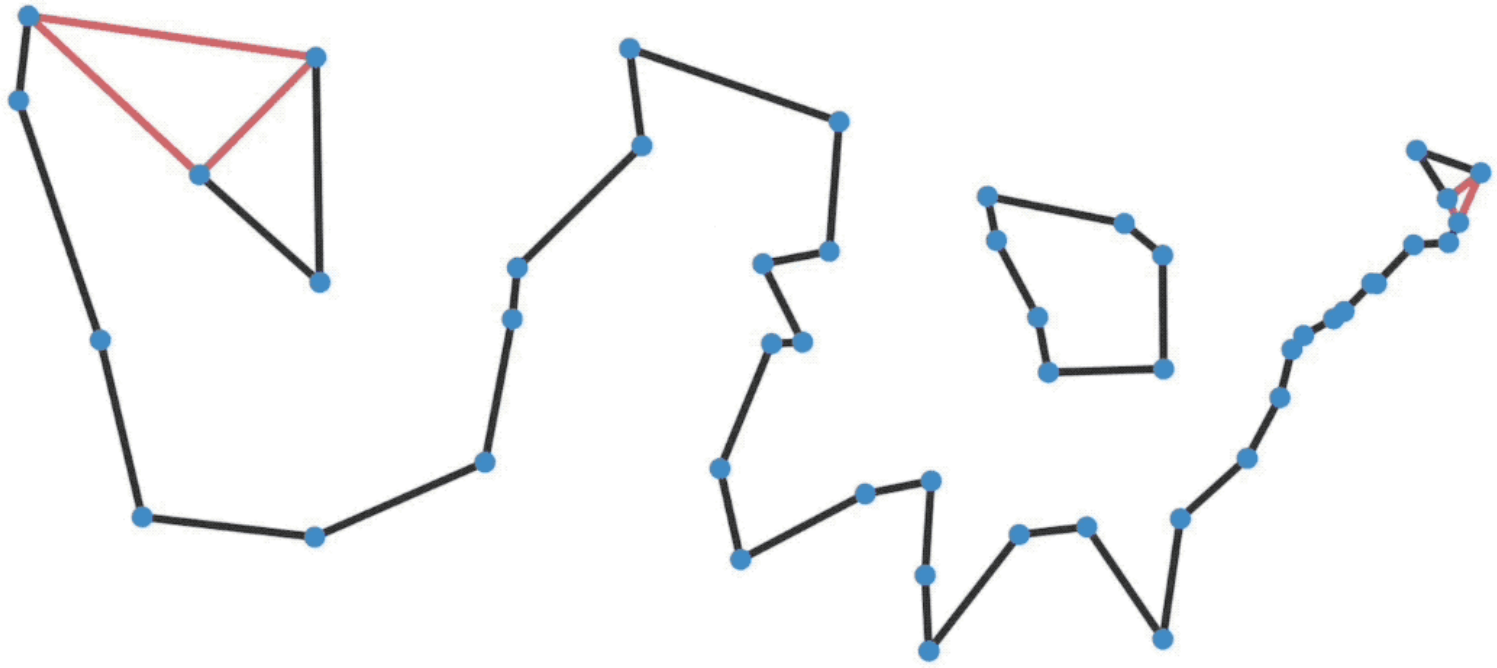
Solving TSP with More Efficient Methods

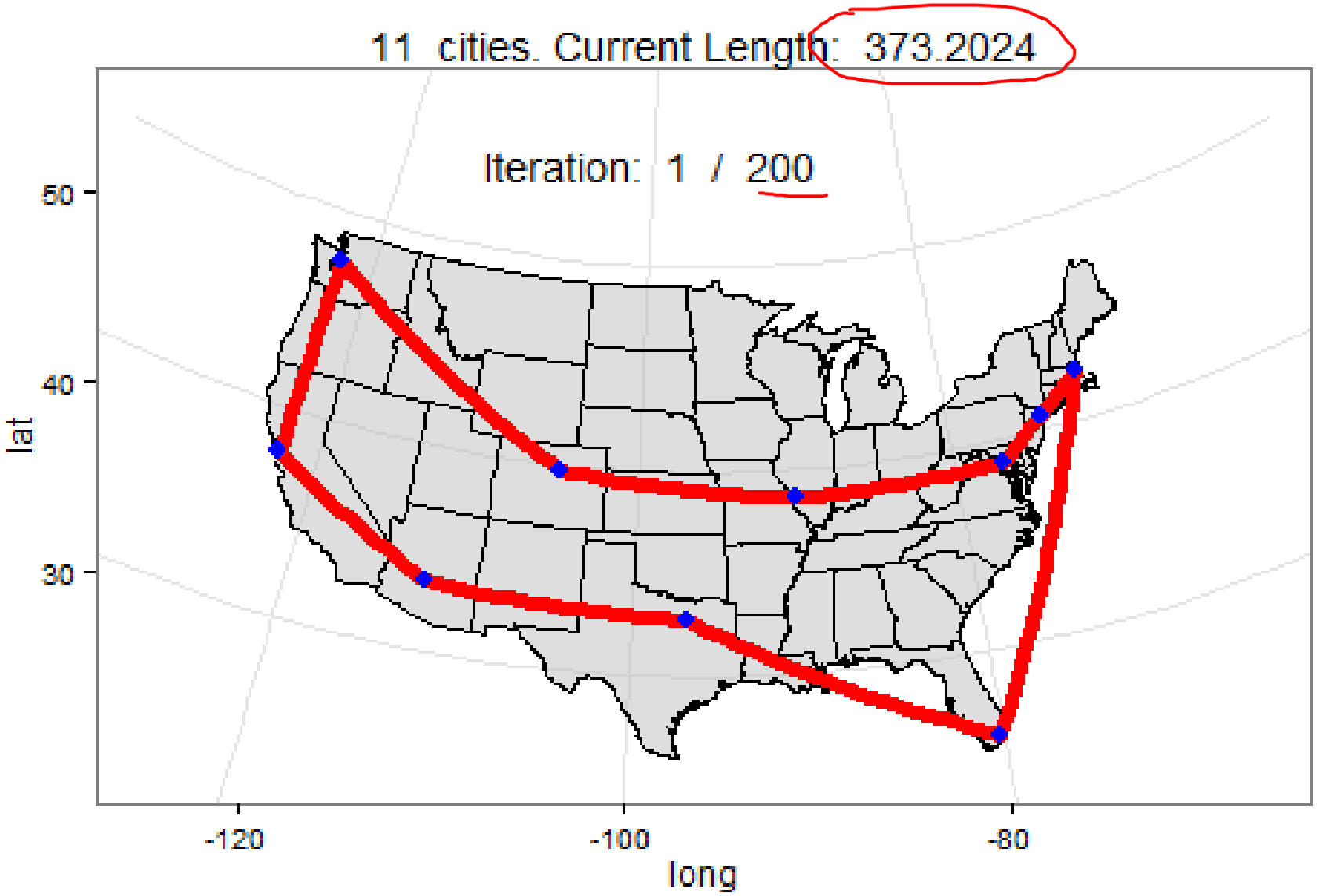
For larger instances of TSP, brute force becomes computationally impractical due to its factorial time complexity. Therefore, several heuristic and approximation algorithms are often used, including:

1. **Nearest Neighbor Algorithm**: Start at a random city and always visit the nearest unvisited city.
2. **Dynamic Programming (Held-Karp Algorithm)**: Breaks the problem down into subproblems to solve it in $O(n^2 * 2^n)$ time.
3. **Genetic Algorithms**: Use evolutionary techniques to search for good approximations.
4. **Simulated Annealing**: A probabilistic technique to find an approximate solution.

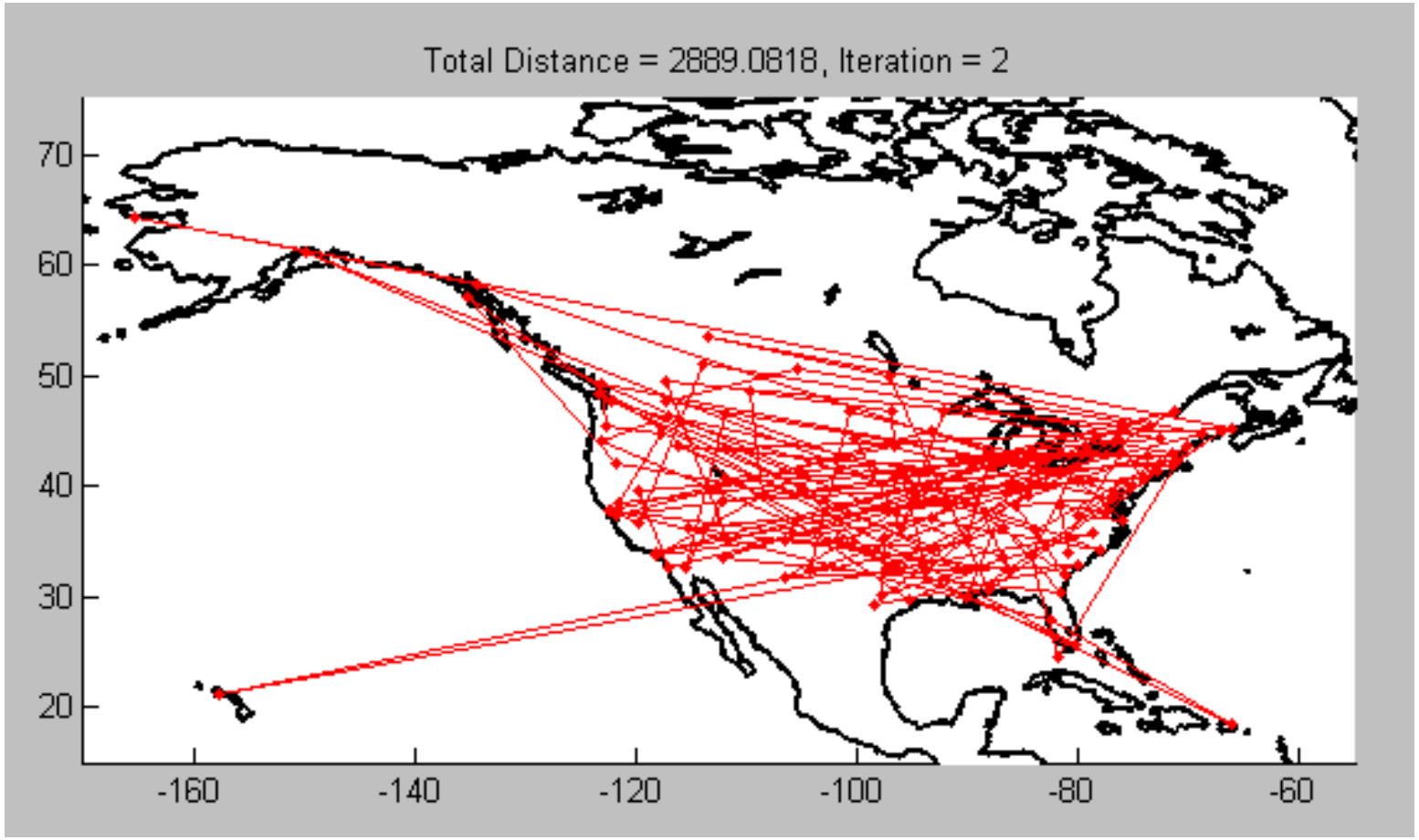
Animation

ANIMATED ALGORITHMS




11 cities. Current Length: 373.2024
Iteration: 1 / 200
50
40
30
lat
-120
-100
-80
long

Animation using Genetic Algorithms



Code for TSP

```python
from itertools import permutations

# Define a function to calculate the total distance of a given route
def calculate_total_distance(route, distance_matrix):
    total_distance = 0
    n = len(route)
    # Add up the distances for each pair of cities in the route
    for i in range(n - 1):
        total_distance += distance_matrix[route[i]][route[i + 1]]
    # Add the distance to return to the starting city
    total_distance += distance_matrix[route[-1]][route[0]]
    return total_distance
```

```python
# Brute-force TSP
def tsp_brute_force(distance_matrix):
    n = len(distance_matrix)
    cities = list(range(n))
    min_distance = float('inf')
    best_route = None

    # Generate all possible permutations of cities (excluding the starting city)
    for perm in permutations(cities[1:]):
        # Add the starting city (0) at the beginning and end of the permutation
        route = [0] + list(perm)
        print(f"Checking route: {route}")
        total_distance = calculate_total_distance(route, distance_matrix)
        print(f"Total distance: {total_distance}")
        print("")

        # Update the best route if a shorter route is found
        if total_distance < min_distance:
            min_distance = total_distance
            best_route = route
            print(f"New best route: {best_route} with distance {min_distance}")

    return best_route, min_distance
```

```python
# Example distance matrix (A, B, C, D)
distance_matrix = [
    [0, 10, 15, 20],  # Distances from city A
    [10, 0, 35, 25],  # Distances from city B
    [15, 35, 0, 30],  # Distances from city C
    [20, 25, 30, 0]   # Distances from city D
]

# Solve the TSP
best_route, min_distance = tsp_brute_force(distance_matrix)

print("\nOptimal route:", best_route)
print("Minimum distance:", min_distance)
```

Other methods for solving the TSP problem

- Other methods exist to solve the TSP problem. They include:

1. **Exact Methods**
   - These methods guarantee to find the optimal solution but may take exponential time for large instances.
- **Brute Force**
   - Explore all possible permutations of cities and choose the shortest one. This method has a time complexity of $O(n!)$, where n is the number of cities.
- **Dynamic Programming (Held-Karp Algorithm)**
   - Uses memorization to store intermediate results, reducing the time complexity to $O(n^2 * 2^n)$. However, this still becomes infeasible for large n.

Other methods for solving the TSP problem

2. Approximation Algorithms

- These provide near-optimal solutions in polynomial time, suitable for large TSP instances.
- **Nearest Neighbor Heuristic**
  - The algorithm starts at a random city and chooses the closest unvisited city until all cities are visited. It's simple but can produce suboptimal solutions.
- **Minimum Spanning Tree (MST) Approximation**
  - Creates a minimum spanning tree and performs a traversal to generate a tour. This can be effective but does not guarantee optimality.

Nearest Neighbor Heuristic for TSP

- The **Nearest Neighbor (NN) Heuristic** is one of the simplest algorithms for finding an approximate solution to the Traveling Salesman Problem (TSP). It builds a tour by always visiting the nearest unvisited city. While this heuristic is easy to implement, it doesn't guarantee an optimal solution but can often provide a good approximation in a reasonable time.

- **Steps of the Nearest Neighbor Heuristic**
  1. **Start at a Random City**: Choose any city as the starting point.
  2. **Find the Nearest Unvisited City**: From the current city, find the city that is closest and hasn't been visited yet.
  3. **Move to That City**: Travel to that city and mark it as visited.
  4. **Repeat**: Continue the process, always traveling to the nearest unvisited city.
  5. **Return to the Starting City**: Once all cities have been visited, return to the starting city to complete the tour.

Example

Let's illustrate the Nearest Neighbor Heuristic with an example involving 5 cities. The distance between the cities is given in the following distance matrix:

	A	B	C	D	E
A	0	10	15	20	25
B	10	0	35	25	30
C	15	35	0	30	20
D	20	25	30	0	15
E	25	30	20	15	0

Step-by-Step Solution

1. **Start at City A**:
   - City A is chosen as the starting point.
   - From City A, the distances to other cities are:
     B = 10, C = 15, D = 20, E = 25.
   - The nearest city is **B** (distance 10).
2. **Move to City B**:
   - From City B, the distances to other unvisited cities are:
     C = 35, D = 25, E = 30.
   - The nearest city is **D** (distance 25).
3. **Move to City D**:
   - From City D, the distances to other unvisited cities are:
     C = 30, E = 15.
   - The nearest city is **E** (distance 15).
4. **Move to City E**:
   - From City E, the only remaining unvisited city is **C** (distance 20
5. **Move to City C**:
   - Now all cities have been visited.
   - Finally, return to the starting city **A** (distance 15).

Final Tour:

The sequence of the tour is: **A → B → D → E → C → A.**

Total Distance:

- A → B = 10
- B → D = 25
- D → E = 15
- E → C = 20
- C → A = 15

Total distance = 10 + 25 + 15 + 20 + 15 = **85.**

Advantages and Disadvantages

- **Advantages:**
  - **Simplicity**: The nearest neighbor heuristic is straightforward to understand and implement.
  - **Speed**: It runs in $O(n^2)$ time faster than more complex exact algorithms.

- **Disadvantages:**
  - **Suboptimal Solutions**: The algorithm doesn't guarantee the optimal tour. It can sometimes lead to poor results, especially when the nearest city isn't part of the optimal tour.
  - **Greedy Nature**: It can make short-sighted decisions since it always chooses the closest city without considering future steps.

Performance

- The Nearest Neighbor Heuristic performs reasonably well for small to medium-sized problems but may perform poorly on larger datasets. It is often used as an initial solution that can be further refined by more sophisticated methods like **k-opt** or **genetic algorithms**.

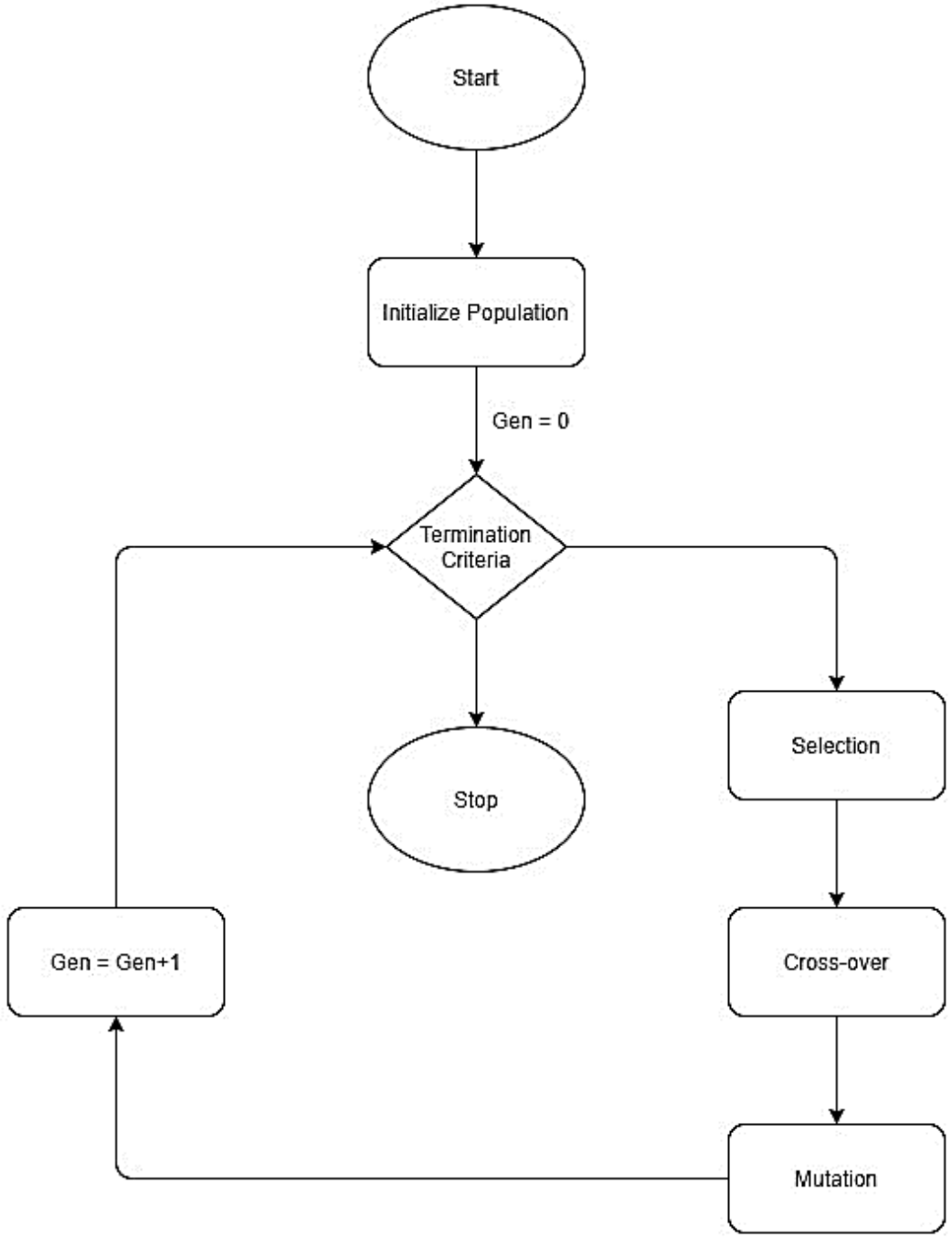
https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-in-python/

Traveling Salesman Problem (TSP) using Genetic Algorithm (Python)

Ramez Shendy · Follow

Published in **AI monks.io** · 15 min read · Aug 5, 2023

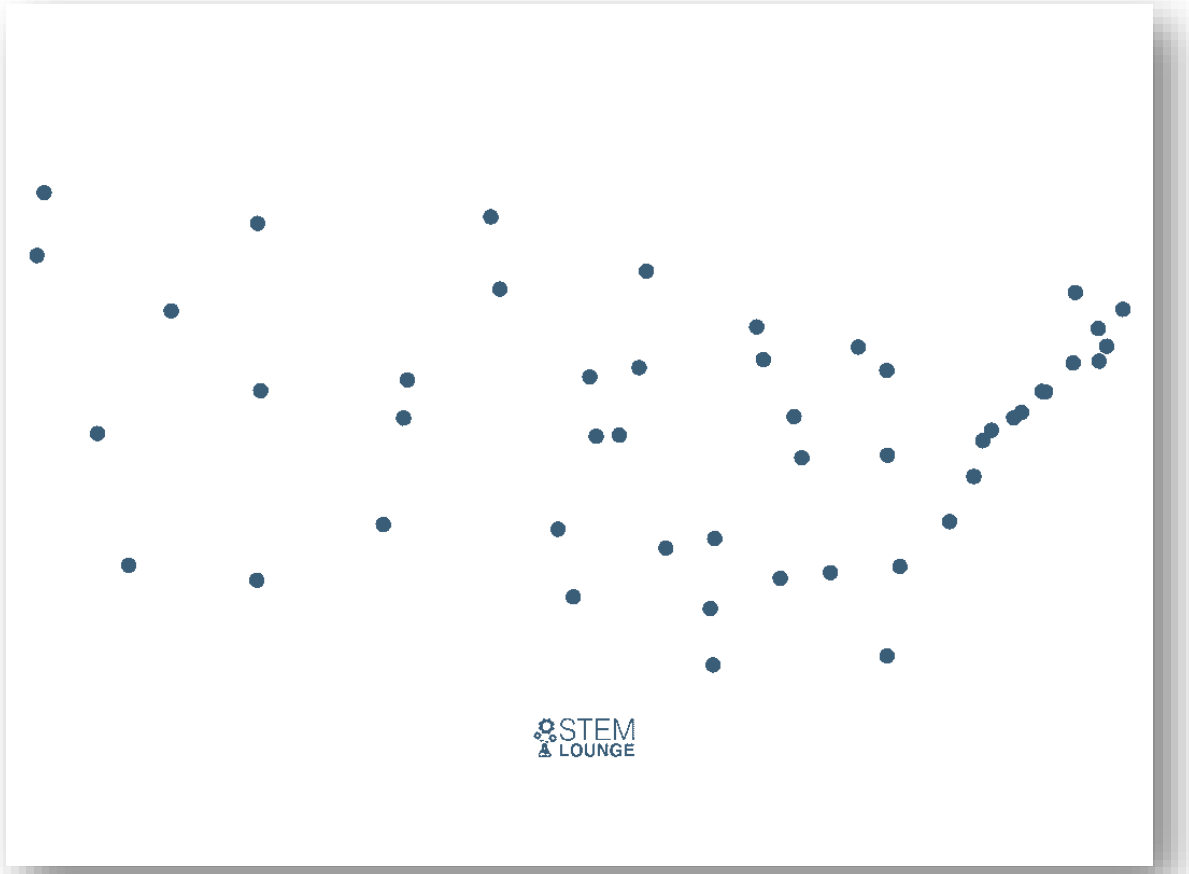
Genetic Algorithms

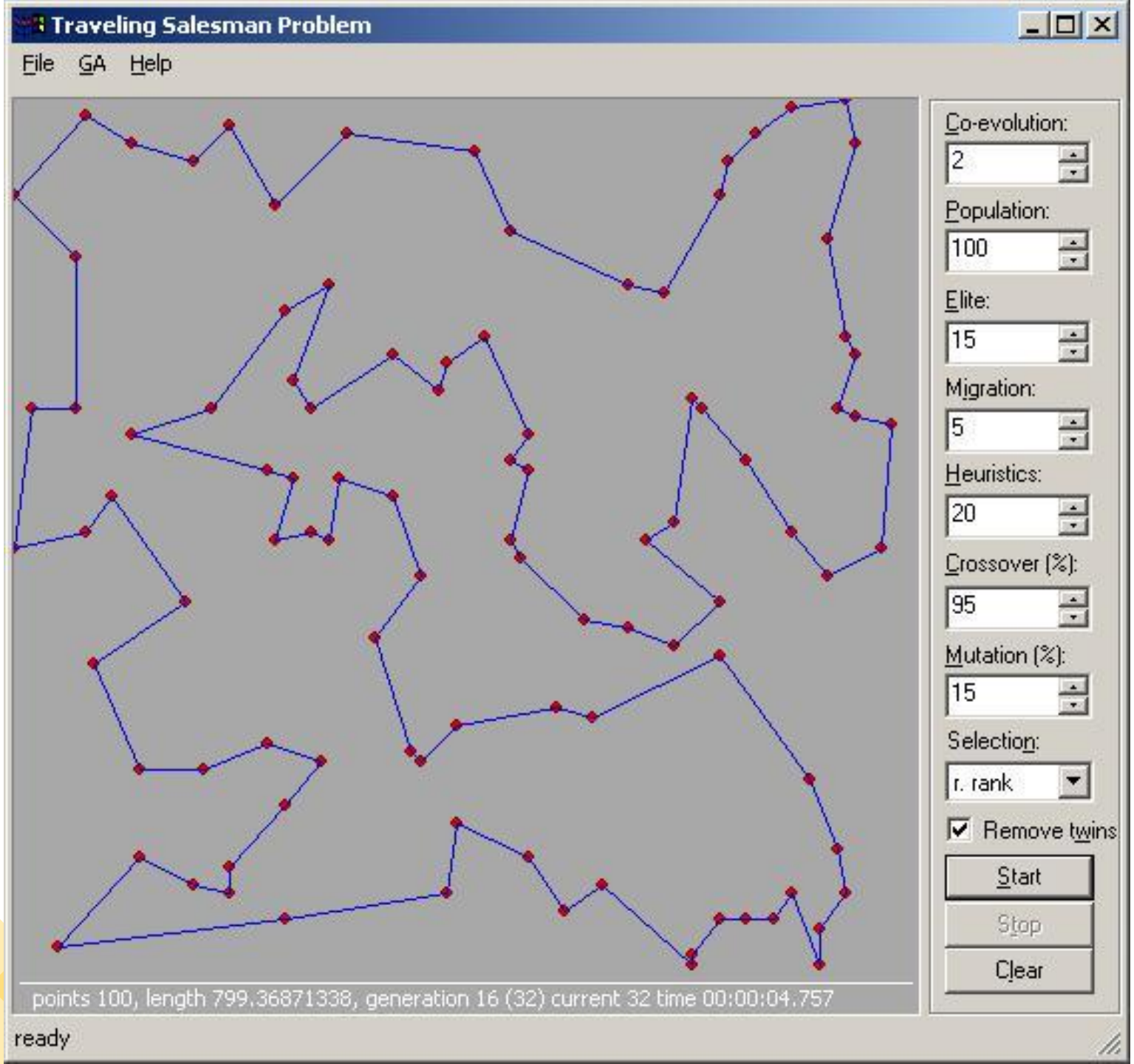


Other methods for solving the TSP problem

3. Metaheuristic Methods

- These methods are designed to explore large search spaces and often provide good approximations in reasonable time for large-scale problems.
  - **Genetic Algorithm (GA)**
  - **Crow Search Algorithm**
  - **Ant Colony Optimization (ACO)**
  - **Particle Swarm Optimization (PSO)**




Traveling Salesman Problem
File
GA
Help
Co-evolution:
2
Population:
100
Elite:
15
Migration:
5
Heuristics:
20
Crossover (%):
95
Mutation (%):
15
Selection:
r. rank
Remove twins
Start
Stop
Clear
points 100, length 799.36871338, generation 16 (32) current 32 time 00:00:04.757
ready

Knapsack Problem

- The **Knapsack Problem** is a classic optimization problem in combinatorial mathematics. It is about selecting a subset of items, each with a weight and a value, to maximize the total value without exceeding a given weight capacity.

- The problem comes in various forms, but the most common one is the **0/1 Knapsack Problem**, where each item can either be included or excluded from the knapsack.

Problem Setup

Suppose you have a knapsack with a weight capacity of **W = 7**, and you are given 4 items with the following weights and values:

Item	Weight	Value
1	2	10
2	3	14
3	4	16
4	5	30

Exhaustive Search Approach

With **4 items**, the number of possible subsets is $2^4 = 16$ (since each item can either be included or excluded). We will evaluate all possible subsets of items and calculate their total weight and value, discarding those that exceed the weight capacity of the knapsack (7 in this case).

Problem Definition

Given:

- A set of n items, where each item i has:
  - Weight w_i
  - Value v_i
- A knapsack with a maximum weight capacity W

The goal is to find the subset of items that maximizes the total value $\sum v_i$ while ensuring that the total weight $\sum w_i$ does not exceed the capacity W.

Subset	Items Included	Total Weight	Total Value	Feasible?
0000	None	0	0	Yes
0001	Item 4	5	30	Yes
0010	Item 3	4	16	Yes
0011	Item 3, Item 4	9	46	No
0100	Item 2	3	14	Yes
0101	Item 2, Item 4	8	44	No
0110	Item 2, Item 3	7	30	Yes

0111	Item 2, Item 3, Item 4	12	60	No
1000	Item 1	2	10	Yes
1001	Item 1, Item 4	7	40	Yes
1010	Item 1, Item 3	6	26	Yes
1011	Item 1, Item 3, Item 4	11	56	No
1100	Item 1, Item 2	5	24	Yes
1101	Item 1, Item 2, Item 4	10	54	No
1110	Item 1, Item 2, Item 3	9	40	No
1111	Item 1, Item 2, Item 3, Item 4	14	70	No

Evaluating the Results

Now that we have evaluated all subsets, we can select the subset with the highest total value that doesn't exceed the weight capacity.

- The highest feasible value is **40**, and it can be achieved by either:
  - Subset `1001` (Item 1 and Item 4): Total weight = 7, Total value = 40
  - Subset `0110` (Item 2 and Item 3): Total weight = 7, Total value = 30

Thus, the optimal solution using **exhaustive search** is to include **Item 1** and **Item 4**, resulting in a total value of **40** and a total weight of **7**.

Exhaustive Search Solution

In an exhaustive search (also called brute force), we try every possible subset of items to find the one that maximizes the total value without exceeding the weight limit. The time complexity of this approach is $O(2^n)$, as there are 2^n possible subsets to consider.

Steps in an Exhaustive Search Solution:

1. Generate all possible subsets of items.
2. For each subset, calculate the total weight and total value.
3. Keep track of the subset that has the highest value while satisfying the weight constraint.
4. Return that subset as the solution.

Conclusion

- In an exhaustive search, we evaluate every possible combination of items, ensuring we find the optimal solution. However, the time complexity is $O(2^n)$, where n is the number of items, making it impractical for large instances. For larger problems, more efficient algorithms like **Dynamic Programming** or **Greedy heuristics** are used.

NP-hard problems

- Thus, for both the **traveling salesman and knapsack problems** considered above, exhaustive search leads to highly inefficient algorithms on every input. These two problems are the best-known examples of so-called ***NP-hard problems***.

- No polynomial-time algorithm is known for any *NP-hard* problem. Moreover, most computer scientists believe such algorithms do not exist, although this very important conjecture has never been proven.

The Assignment Problem

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

- The **Assignment Problem** is a fundamental combinatorial optimization problem where the goal is to assign a set of tasks (or jobs) to a set of agents (or workers) such that the total cost (or time, distance, etc.) of the assignments is minimized (or the total profit is maximized). It is widely used in resource allocation, job scheduling, and logistics.

Problem Definition

Given:

- A set of n agents and n tasks.
- A cost matrix C of size $n \times n$, where C_{ij} represents the cost of assigning task j to agent i.

The goal is to find the optimal assignment of tasks to agents such that each agent is assigned exactly one task, each task is assigned to exactly one agent, and the total cost is minimized.

Mathematically, we aim to minimize the total cost:

$$\text{Total Cost} = \sum_{i=1}^{n} C_{i,\pi(i)}$$

where $\pi(i)$ is a permutation of the set $\{1, 2, \ldots, n\}$ representing the assignment of tasks to agents.

Exhaustive Search Solution

An **exhaustive search** solution to the Assignment Problem involves generating all possible assignments of tasks to agents and calculating the total cost for each one. Since there are $n!$ possible assignments (permutations of the tasks), the time complexity is $O(n!)$, which becomes computationally infeasible for large n.

Steps for Exhaustive Search:

1. **Generate all permutations** of the tasks.
2. **Calculate the total cost** for each permutation by summing the costs of assigning each task to a specific agent.
3. **Keep track of the minimum cost** and its corresponding assignment.
4. **Return the assignment** that yields the minimum cost.

Example

Let's consider a case with 3 agents and 3 tasks. The cost matrix is given as follows:

Agent \ Task	Task 1	Task 2	Task 3
Agent 1	9	2	7
Agent 2	6	4	3
Agent 3	5	8	1

Step 1: Generate all permutations of tasks

Since there are 3 agents and 3 tasks, the total number of possible assignments (permutations) is $3! = 6$. The possible assignments (permutations of tasks) are:

1. (Task 1, Task 2, Task 3)
2. (Task 1, Task 3, Task 2)
3. (Task 2, Task 1, Task 3)
4. (Task 2, Task 3, Task 1)
5. (Task 3, Task 1, Task 2)
6. (Task 3, Task 2, Task 1)

1. **(Task 1, Task 2, Task 3):**
   - Agent 1 → Task 1 (cost = 9)
   - Agent 2 → Task 2 (cost = 4)
   - Agent 3 → Task 3 (cost = 1)
   - Total cost = 9 + 4 + 1 = 14
2. **(Task 1, Task 3, Task 2):**
   - Agent 1 → Task 1 (cost = 9)
   - Agent 2 → Task 3 (cost = 3)
   - Agent 3 → Task 2 (cost = 8)
   - Total cost = 9 + 3 + 8 = 20
3. **(Task 2, Task 1, Task 3):**
   - Agent 1 → Task 2 (cost = 2)
   - Agent 2 → Task 1 (cost = 6)
   - Agent 3 → Task 3 (cost = 1)
   - Total cost = 2 + 6 + 1 = 9
4. **(Task 2, Task 3, Task 1):**
   - Agent 1 → Task 2 (cost = 2)
   - Agent 2 → Task 3 (cost = 3)
   - Agent 3 → Task 1 (cost = 5)
   - Total cost = 2 + 3 + 5 = 10
5. **(Task 3, Task 1, Task 2):**
   - Agent 1 → Task 3 (cost = 7)
   - Agent 2 → Task 1 (cost = 6)
   - Agent 3 → Task 2 (cost = 8)
   - Total cost = 7 + 6 + 8 = 21
6. **(Task 3, Task 2, Task 1):**
   - Agent 1 → Task 3 (cost = 7)
   - Agent 2 → Task 2 (cost = 4)
   - Agent 3 → Task 1 (cost = 5)
   - Total cost = 7 + 4 + 5 = 16

Step 3: Identify the assignment with the minimum cost

From the total costs calculated above, the minimum cost is **9**, corresponding to the assignment **(Task 2, Task 1, Task 3)**, where:

- Agent 1 → Task 2
- Agent 2 → Task 1
- Agent 3 → Task 3

Summary:

- The **Assignment Problem** involves finding the optimal way to assign tasks to agents to minimize total cost.
- **Exhaustive search** examines all possible assignments, making it computationally expensive with a time complexity of $O(n!)$.

Closest Pair Problem

- The **Closest Pair Problem** is a classic problem in computational geometry. The goal is **to find the two closest points in each set of points in a 2D plane.**
- The distance between points is typically calculated using **Euclidean distance**. This problem has several applications, including **clustering algorithms**, geographical mapping, and computer graphics.

Problem Definition

Given a set of n points $P = \{p_1, p_2, \ldots, p_n\}$ on a 2D plane, the objective is to find two distinct points p_i and p_j such that the distance $d(p_i, p_j)$ is minimized, where:

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Example

Consider the following set of points on a 2D plane:

$$P = \{(1, 3), (4, 5), (8, 9), (2, 2), (7, 8)\}$$

We calculate the Euclidean distance between each pair:

- $d((1, 3), (4, 5)) = \sqrt{(1-4)^2 + (3-5)^2} = \sqrt{9+4} = \sqrt{13}$
- $d((1, 3), (8, 9)) = \sqrt{(1-8)^2 + (3-9)^2} = \sqrt{49+36} = \sqrt{85}$
- $d((1, 3), (2, 2)) = \sqrt{(1-2)^2 + (3-2)^2} = \sqrt{1+1} = \sqrt{2}$
- $d((1, 3), (7, 8)) = \sqrt{(1-7)^2 + (3-8)^2} = \sqrt{36+25} = \sqrt{61}$

... and similarly for other pairs.

After calculating all the distances, we find that the pair $(1, 3)$ and $(2, 2)$ has the smallest distance of $\sqrt{2}$.

Brute Force Approach

The simplest approach is to compute the distance between every possible pair of points and return the pair with the minimum distance. This results in a time complexity of $O(n^2)$, since there are $\frac{n(n-1)}{2}$ pairs in a set of n points.

Optimized Approach: Divide and Conquer

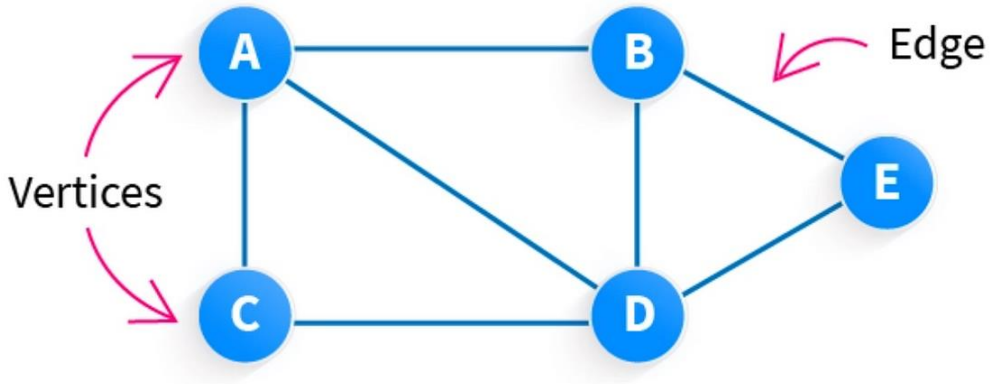
To improve the time complexity, we can use a **Divide and Conquer** strategy:

1. **Sort** the points by their x-coordinates.
2. **Divide** the points into two equal halves.
3. **Conquer**: Recursively find the closest pair in each half.
4. **Combine:** Find the closest pair across the two halves by checking the points near the dividing line.

This divide and conquer approach reduces the time complexity to $O(n \log n)$.

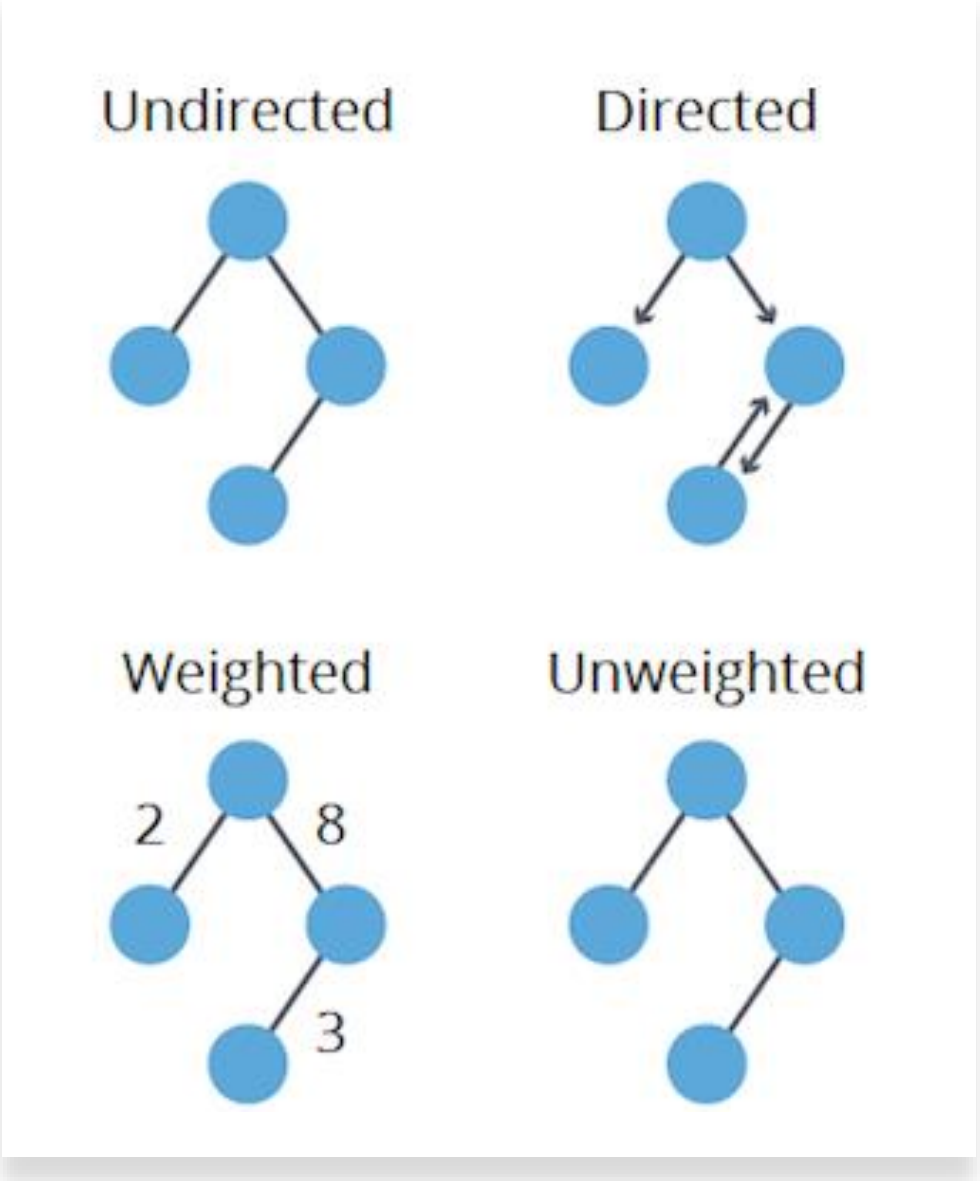
What is a Graph?

- A **graph** is a mathematical structure that models pairwise relationships between objects.
- It consists of:
  - **Vertices** (or nodes) that represent the objects.
  - **Edges** (or arcs) that represent the connections between the vertices.



Types of Graphs

1. **Undirected Graph**: The edges do not have a direction; the connection between two vertices is bidirectional.
2. **Directed Graph (Digraph)**: The edges have a direction, indicating the connection flows from one vertex to another.
3. **Weighted Graph**: Edges have associated weights or costs, representing some quantity like distance or time.
4. **Unweighted Graph**: Edges have no weights.



Advantages of graphs

1. Graphs can be used to model and analyze complex systems and relationships.
2. They are useful for visualizing and understanding data.
3. Graph algorithms are widely used in computer science and other fields, such as **social network analysis, logistics, and transportation**.

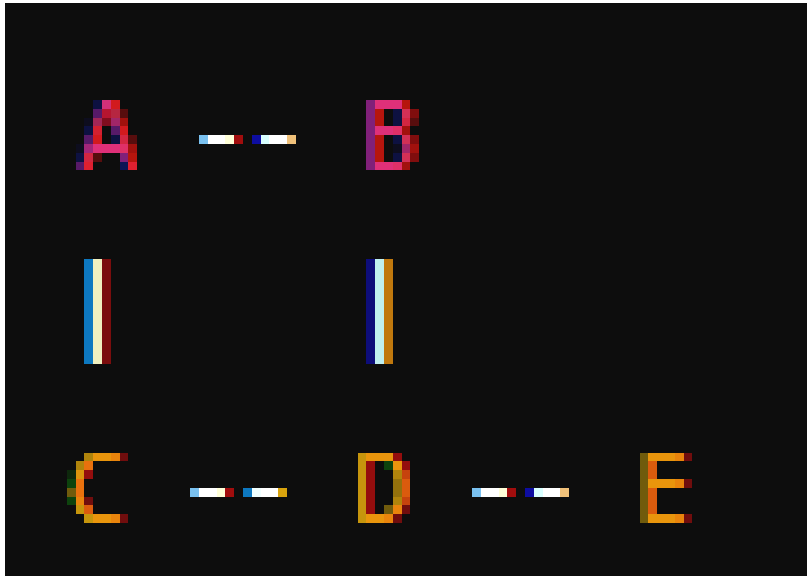
Disadvantages of graphs:

1. Large graphs can be complex to visualize and analyze.
2. Graph algorithms can be computationally expensive, especially for large graphs.
3. **The interpretation of graph results can be subjective and may require domain-specific knowledge.**
4. Graphs can be susceptible to **noise** and **outliers**, which can impact the accuracy of analysis results.

Example:

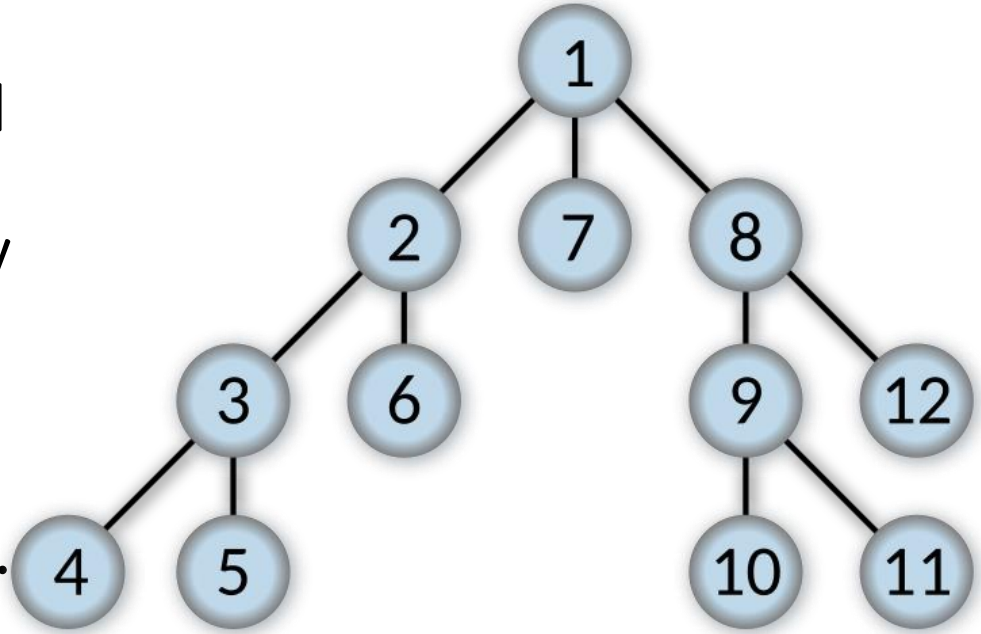
Consider the following graph with 5 vertices and 6 edges:

- Vertices: $V = \{A, B, C, D, E\}$
- Edges: $E = \{(A, B), (A, C), (B, D), (C, D), (C, E), (D, E)\}$



Depth First Search Algorithm

- **Depth First Search (DFS)** is a graph traversal algorithm used to explore vertices and edges of a graph by starting at a root (or any arbitrary node), **moving as far as possible along each branch before backtracking**.
- **It explores one branch of the graph as deeply as possible before moving to another.**

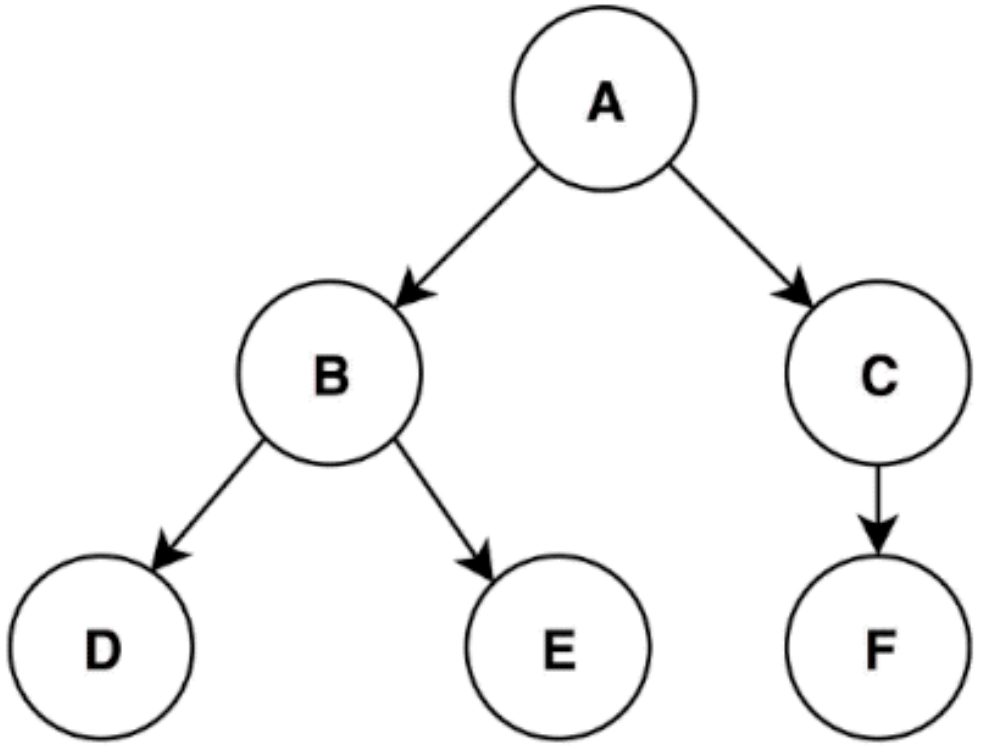


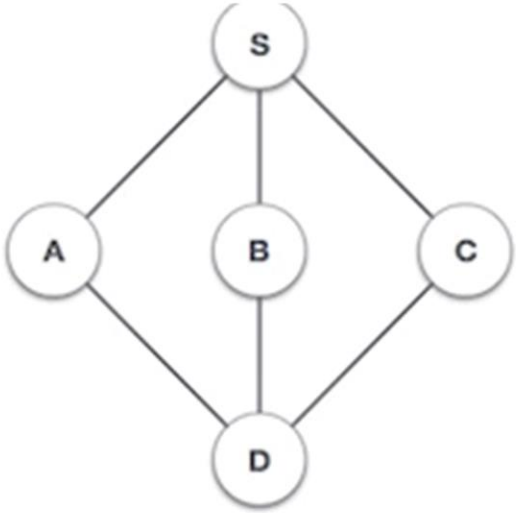
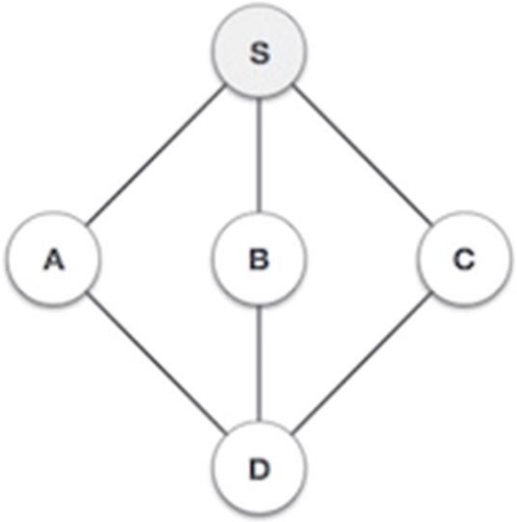
Characteristics of DFS

- It is called a **depth-first** search because it prioritizes going deep in the graph/tree before exploring other branches.
- It uses a **stack** (often implemented using recursion) to keep track of the vertices to visit.
- DFS can be applied to both **directed** and **undirected** graphs.

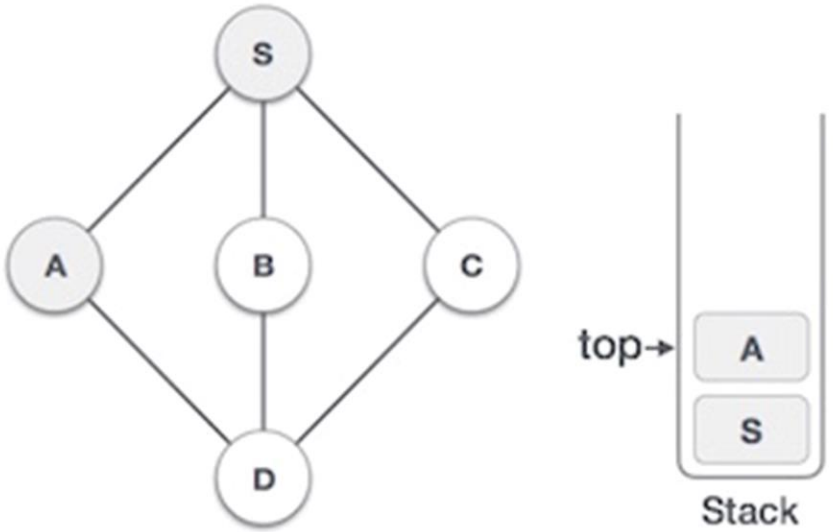
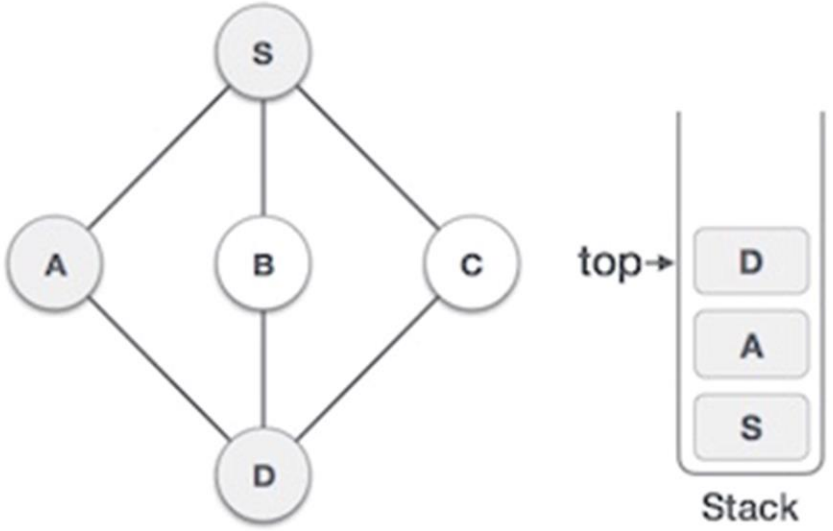
DFS Algorithm

1. **Start** at an initial vertex.
2. **Visit** the vertex and mark it as visited.
3. For each **unvisited neighbor** of the current vertex, recursively visit that neighbor.
4. If all neighbors are visited, **backtrack** to the previous vertex and explore other unvisited vertices.
5. Repeat this process until all vertices have been visited.

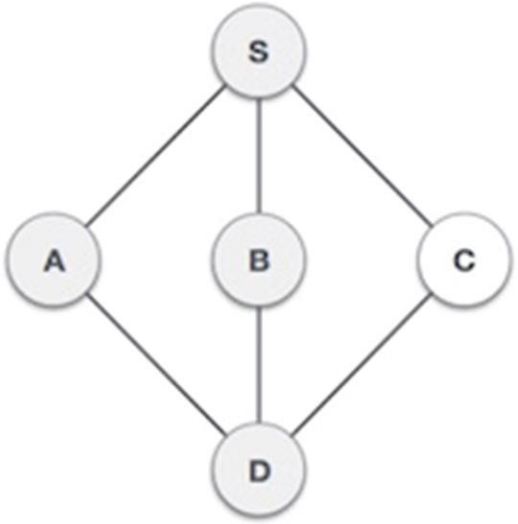
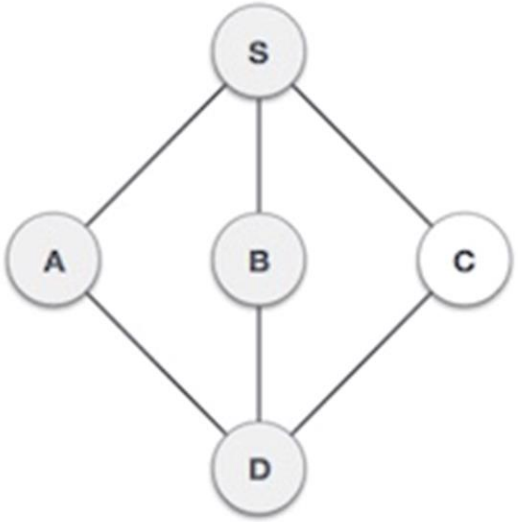


1			Initialize the stack.
2			Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

Output = S

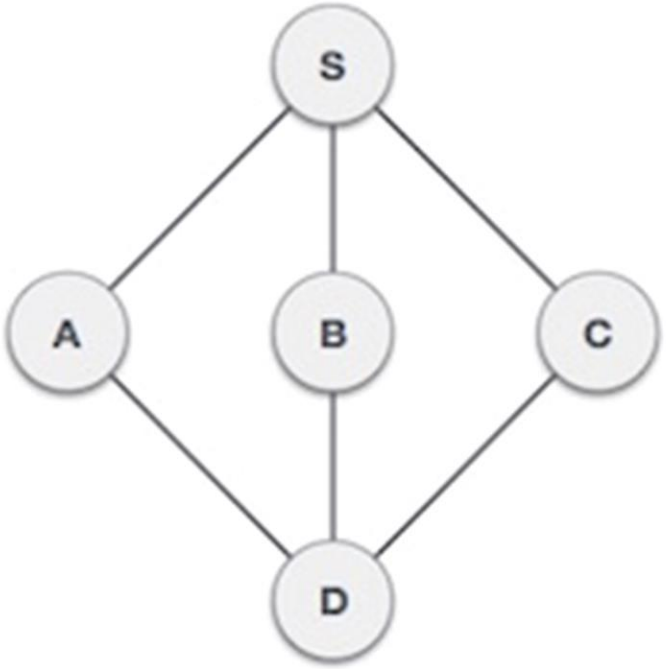
3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

Output = S, A, D

5			We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.
6			We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

Output = S, A, D, B

7



Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

Output = S, A, D, B, C

https://www.youtube.com/watch?v=iaBEKo5sM7w

Time Complexity of DFS

For a graph with V vertices and E edges:

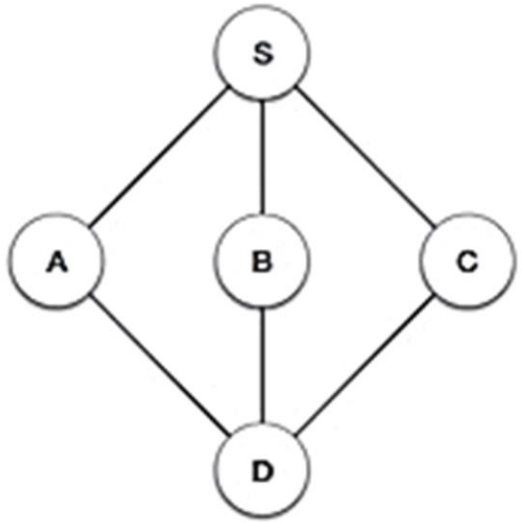
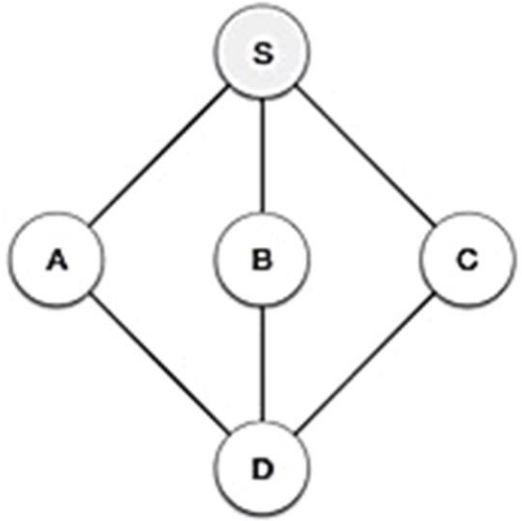
- **Time Complexity**: $O(V + E)$, because each vertex and each edge is processed once.
- **Space Complexity**: $O(V)$ for storing the recursion stack or the stack used in the iterative implementation.

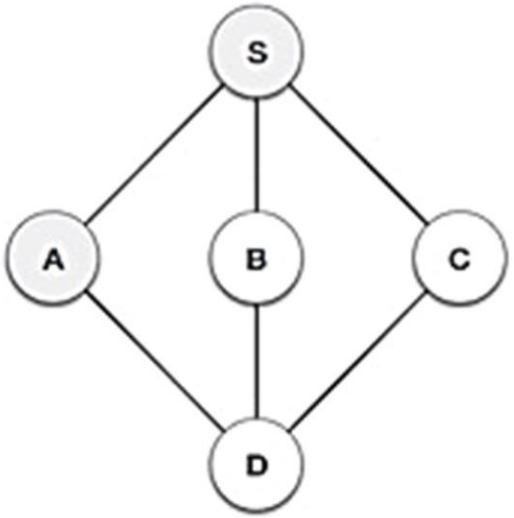
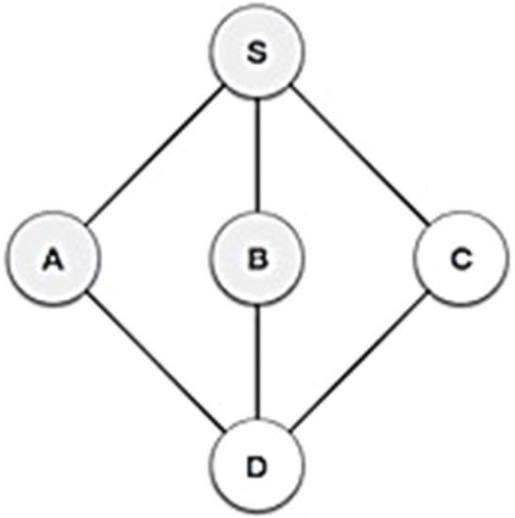
Breadth First Search Algorithm

- **Breadth First Search (BFS)** is a graph traversal algorithm used to explore nodes and edges of a graph level by level. Unlike **Depth First Search (DFS)**, which explores as deep as possible before backtracking, BFS explores all the neighbors of a node before moving on to their neighbors.

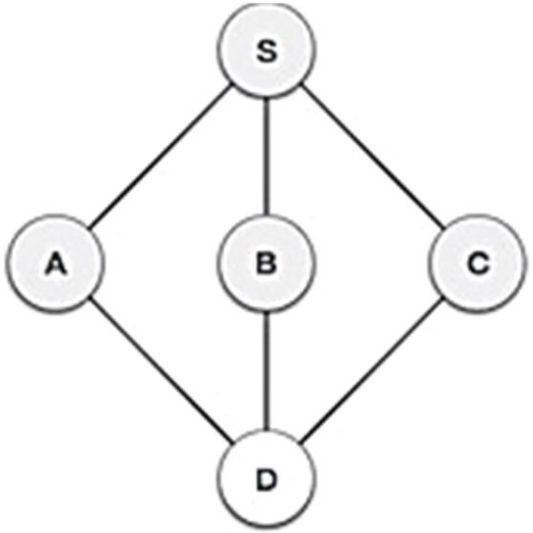
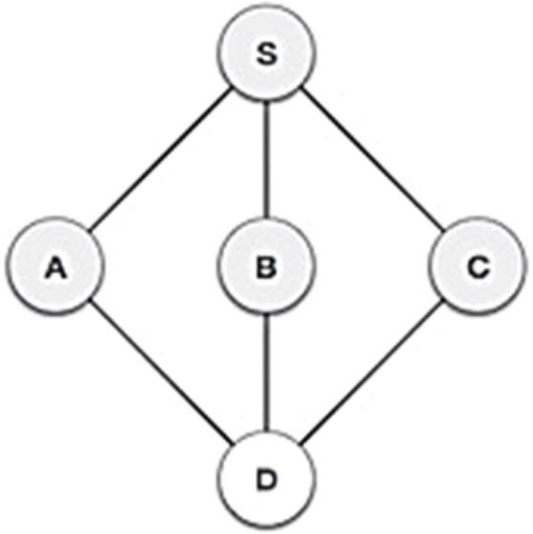
Key Characteristics of BFS

- BFS uses a **queue** data structure to keep track of the vertices to visit.
- It explores all nodes at the present depth (or "level") before moving on to nodes at the next level.
- BFS is commonly used to find the **shortest path** in unweighted graphs and is well-suited for **level-order traversal** in trees.

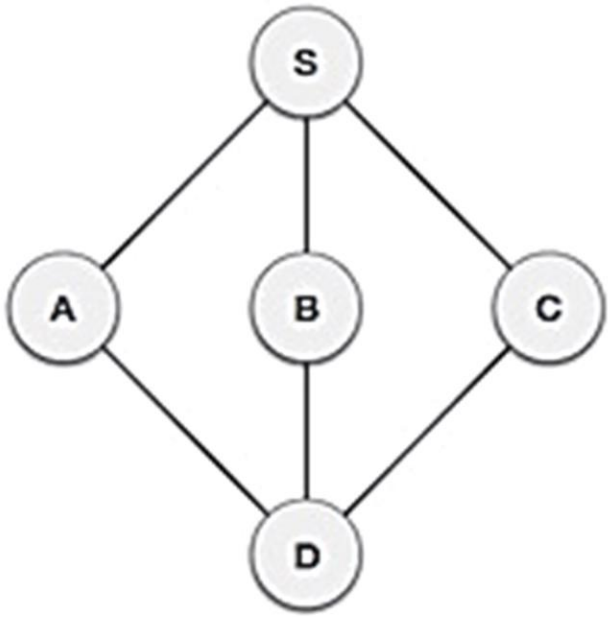
1		Queue	Initialize the queue.
2		Queue	We start from visiting **S** (starting node), and mark it as visited.

3		A Queue	We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4		B A Queue	Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.

Output = B, A

5		C B A Queue	Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.
6		C B Queue	Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.

Output = C, B, A

7		D C B Queue	From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

Output = D, C, B, A

https://www.youtube.com/watch?v=QRq6p9s8NVg

1. A
2. B, S
3. S
4. C
5. C, G
6. G, D, E, F
7. D, E, F, H
8. E, F, H
9. F, H
10. H
11. H

Thank You!