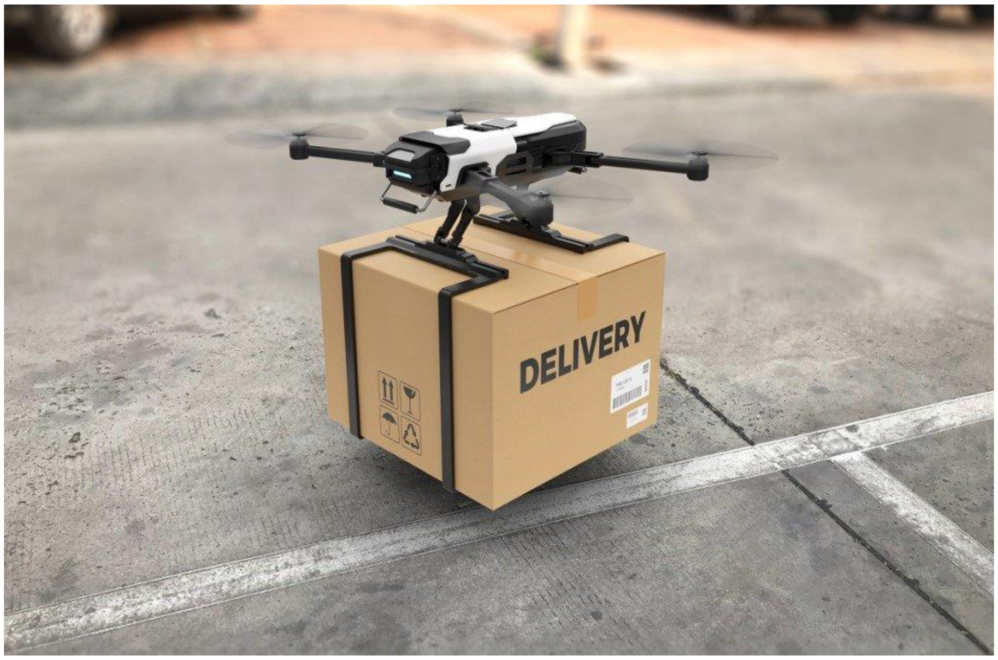
Delivery Drone Route Proposal



TEAM MEMBERS :

- شيماء عماد محمد محمود
- أميرة حسين احمد شخبه
- حمزة محمود فرج سليمان
- عبدالرحمن اليمني معوض
- اسماعيل وصال المصلحى عامر
- مريم جمعه حسين ضباش
- محمد محمود عبد الحميد الشويمى
- محمد عبد الغني الدسوقي

Project Proposal

Drone Delivery Route Optimization Using AI Search Algorithms

---

1. Project Title

Drone Delivery Route Optimization Using Multiple AI Search Algorithms

---

2. Problem Definition

The goal of this project is to solve the **Drone Delivery Route Optimization Problem**, where an Unmanned Aerial Vehicle (UAV) must find an efficient route to reach specific delivery points.

- **Representation:** The delivery environment is modeled as a **graph**, where each location is a **Node**, and each valid route between nodes is an **Edge** that may have a weight (distance, energy, or time).
- **Main Challenge:** Selecting and comparing multiple Artificial Intelligence (AI) search algorithms to determine which one can generate a valid and optimal route efficiently, considering factors such as reachability, cost, memory usage, and execution time.

---

3. Rationale (Why This Problem?)

This problem was selected due to its practical relevance and academic value:

- **Modern Application:** Drone delivery is increasingly used in logistics and smart cities.
- **Optimization Impact:** Enhanced route optimization reduces delivery time, energy consumption, and improves system efficiency.
- **Academic Suitability:** The problem perfectly matches the concepts taught in the AI course, especially search algorithms.

---

4. Algorithms to Be Implemented

Below is the list of algorithms the team will implement and compare.

Uninformed Search

- **Depth First Search (DFS)**
- **Breadth First Search (BFS)**
- **Uniform Cost Search (UCS)**

Informed Search

- **A* Search**
- **Greedy Best-First Search**

Evolutionary Algorithm

- **Genetic Algorithm (GA)**

---

5. Detailed Algorithm Roles and Implementation

#	Algorithm	Team Member	Purpose	Key Notes	Implementation
1	Depth First Search (DFS)	**Shimaa**	Check reachability and explore paths deeply.	Does not guarantee optimality. Useful for connectivity.	Python (Stack-based).
2	Breadth First Search (BFS)	**Hamza**	Find shortest path in terms of number of steps.	Guarantees shortest path in unweighted graphs.	Python (Queue-based).
3	Uniform Cost Search (UCS)	**Mariam**	Find the least-cost path in weighted graphs.	Guarantees optimal solution.	Python (Priority Queue).
4	A* Search	**Ismail**	Find an optimal and efficient path using heuristics.	Combines cost + heuristic. Optimal if heuristic is admissible.	Python (Priority Queue).
5	Greedy Best-First Search	**Amira**	Choose the next node closest to the goal.	Very fast but not optimal. Can get stuck in local minima.	Python or C++.

#	Algorithm	Team Member	Purpose	Key Notes	Implementation
6	Genetic Algorithm (GA)	**Abdelrahman**	Produce near-optimal routes via population evolution.	Useful for large or complex search spaces.	Python (Evolutionary approach).
7	Data Collection & File Organization	**Mohamed ElShemimy**	Collect all algorithm outputs and organize them into one structured file.	Ensures proper formatting and documentation of results.	Word
8	Final Comparison & Evaluation	**Mohamed Abdelghany**	Compare all algorithms using metrics provided by the team and create the final comparison tables.	Produces final performance summary.	Word

6. Comparison Measures

All algorithms will be evaluated and compared based on the following metrics:

- **Execution Time:** How fast each algorithm finds a route.
- **Memory Usage:** Total memory consumed during execution.
- **Success Rate:** Whether the algorithm can reach the goal or not.
- **Solution Optimality:** How close the solution is to the true optimal route.
- **Scalability:** Performance when the graph size increases.

7. Team Roles and Responsibilities

Team Member	Primary Role	Secondary Role	Tools Used
Shimaa	DFS Implementation	Testing & Documentation	Python
Hamza	BFS Implementation	Testing & Documentation	Python
Ismail	A* Search Implementation	Testing & Documentation	Python
Mariam	UCS Implementation	Testing & Documentation	Python
Amira	Greedy Search Implementation	Testing & Documentation	Python / C++
Abdelrahman	Genetic Algorithm Implementation	Testing & Documentation	Python
Mohamed ElShemimy	Data Collection & File Organization	Documentation	Word
Mohamed Abdelghany	Final Comparison & Evaluation	Documentation	Word
Team Leader(Shimaa)	GitHub Repo Management, Merging Code, Proposal Submission	—	GitHub

AI Search Algorithms for Drone Delivery

Depth First Search (DFS)

Depth First Search (DFS) for Drone Delivery Problem

What is DFS?

DFS (Depth First Search) is a graph search algorithm that explores one full branch deeply before moving to another.

Explores the left-most / deepest path first

Uses a stack

Performs backtracking when reaching a dead end

Gives a valid path, but not the shortest or optimal one

Why DFS in the Drone Delivery Problem?

In the drone-delivery route:

DFS helps check if the drone can reach all delivery points

Ensures the graph is connected

Finds a valid route, but not the best or shortest

Not suitable for finding optimal or cost-efficient routes

How DFS Works?

Start at the initial location

Mark it as visited

Move to the next unvisited neighbor

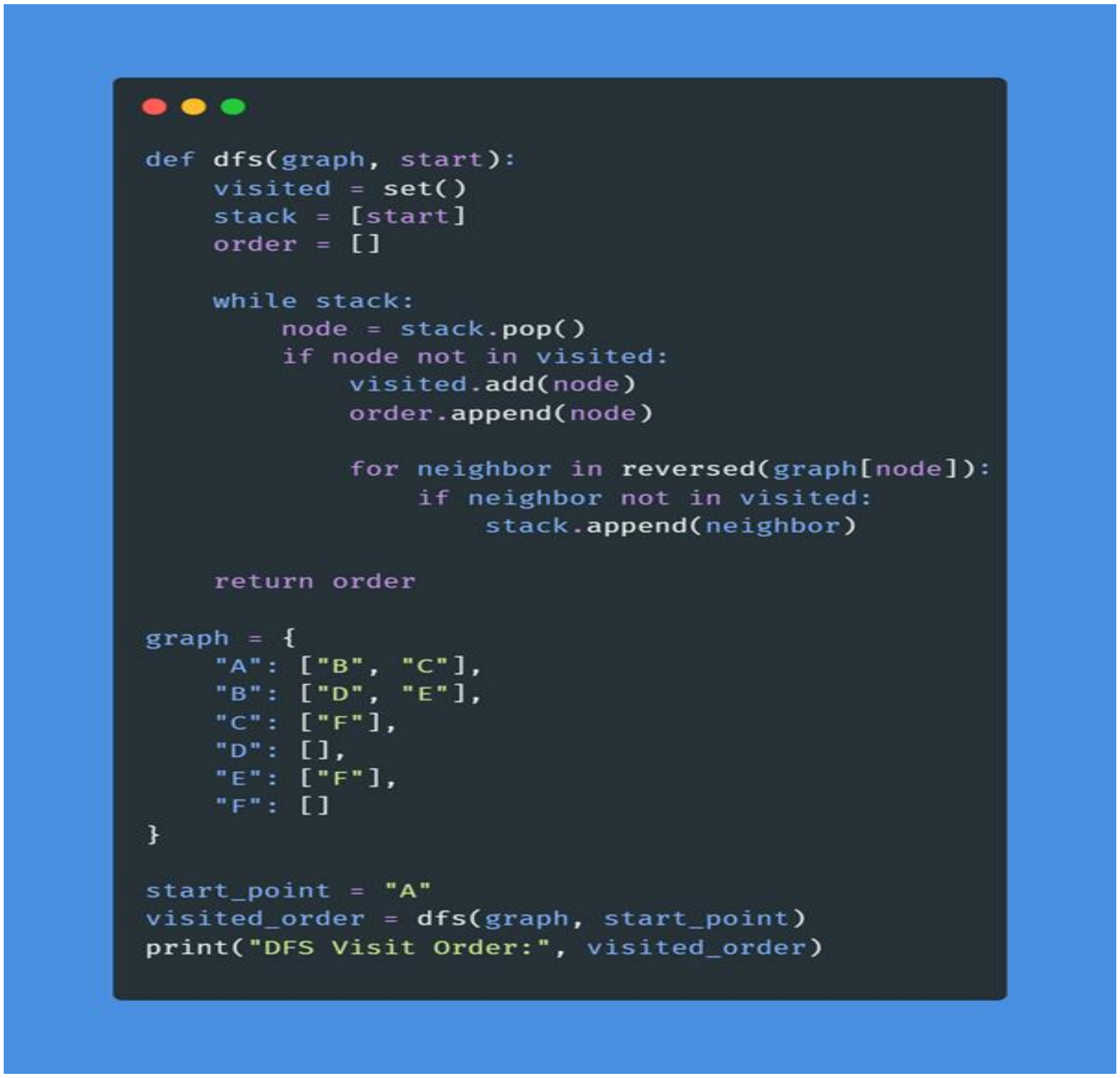
Continue going deeper

When reaching a dead end → Backtrack

Try another branch

Continue until all reachable nodes are visited or find the goal

Python Implementation



```python
def dfs(graph, start):
    visited = set()
    stack = [start]
    order = []

    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            order.append(node)

            for neighbor in reversed(graph[node]):
                if neighbor not in visited:
                    stack.append(neighbor)

    return order

graph = {
    "A": ["B", "C"],
    "B": ["D", "E"],
    "C": ["F"],
    "D": [],
    "E": ["F"],
    "F": []
}

start_point = "A"
visited_order = dfs(graph, start_point)
print("DFS Visit Order:", visited_order)
```

Example Output

DFS Visit Order: ['A', 'C', 'F', 'B', 'E', 'D']

Meaning:

DFS reached all delivery points

The path is not optimal

DFS confirms connectivity only

Advantages

Simple and easy to implement

Uses low memory

Good for checking connectivity

Works well in deep graphs

Disadvantages

Does NOT give shortest path

May explore deep wrong routes

Slower in large/deep graphs

Not suitable for weighted routes

Comparison Measures for DFS

1. **Execution Time**

DFS can be moderately fast because it goes deep quickly,

but in large graphs it may waste time exploring long incorrect paths.

2. **Memory Usage**

DFS uses very little memory because it only stores:

Stack

Visited nodes

3. **Success Rate**

DFS can reach the goal if a path exists.

It may take a long or wrong route, but it eventually finds a path.

4. **Solution Optimality**

DFS does not produce an optimal route.

It always returns the first path it finds, not the shortest or least-cost one.

5. **Scalability**

DFS does not scale well.

When the graph grows deeper or larger, DFS may explore unnecessary long paths before backtracking.

Breadth First Search (BFS)

Delivery Drone Route Problem:

Problem description:

Our project focuses on a Delivery Drone Navigation

Problem, where a drone must deliver a package from a Start Point (A) to a Target Point (T) while avoiding obstacles such as buildings, no-fly zones, or restricted paths.

We represent the city map as a graph, where:

•Nodes = possible drone positions

•Edges = valid movement paths between positions

The goal is to find the shortest path from A→ T to ensure the drone delivers the package efficiently and safely.

Breadth-First Search (BFS):

I prefer Breadth-First Search (BFS) because:

•It guarantees the shortest path in an unweighted graph

•It explores all neighbors level by level

•It is simple, fast, and works very well for grid maps

•Perfect for drone navigation where all movements cost the same

How BFS Works:

Start from the initial node A

Visit all neighbors (level 1)

Then visit neighbors of neighbors (level 2)

Continue expanding until reaching T

Stop when the target node is found

Reconstruct the shortest path using parent pointers BFS uses a queue to keep track of nodes to explore.

Example gird:

0 0 0 1 0

0 A 0 1 0

0 0 0 0

1 0 1 0 0 0 0 1 T Legend:

•0 → free path

•1 → obstacle

•A → start

•T → target

BFS Traversal on the Grid:

Step-by-step:

1.Start at A

2.Add neighbors to queue

3.Expand level by level

4.Avoid obstacles and already-visited nodes

5.Stop at T

Final Output From BFS:

BFS returns:

•Shortest Path (A →B →E →G → T)

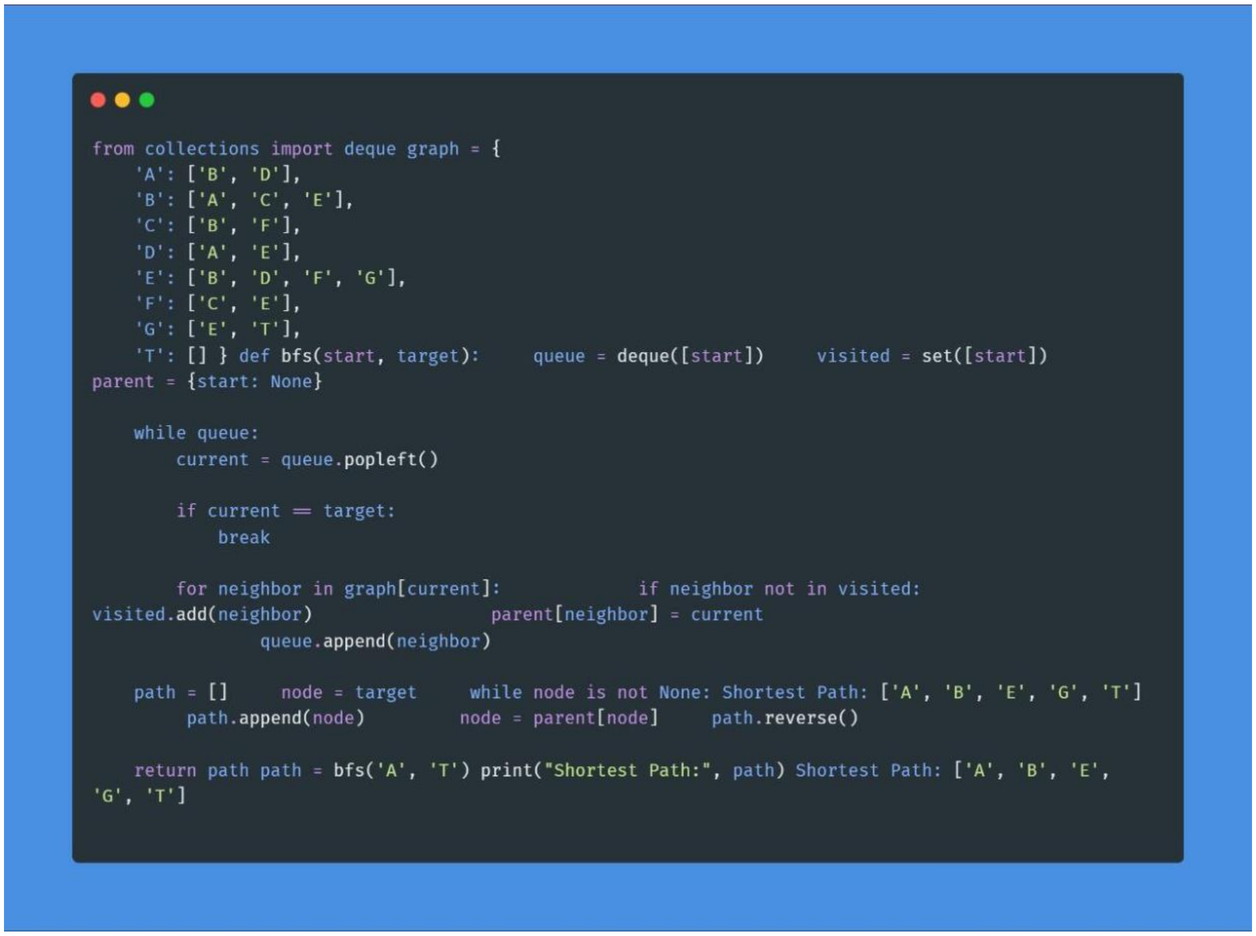
•Path length

•Visited nodes count

•Time complexity: O(V + E)

•Space complexity: O(V)

BFS Code (Python):



```
from collections import deque graph = {
    'A': ['B', 'D'],
    'B': ['A', 'C', 'E'],
    'C': ['B', 'F'],
    'D': ['A', 'E'],
    'E': ['B', 'D', 'F', 'G'],
    'F': ['C', 'E'],
    'G': ['E', 'T'],
    'T': [] } def bfs(start, target):     queue = deque([start])     visited = set([start])
parent = {start: None}

    while queue:
        current = queue.popleft()

        if current == target:
            break

        for neighbor in graph[current]:             if neighbor not in visited:
visited.add(neighbor)                 parent[neighbor] = current
                queue.append(neighbor)

    path = []     node = target     while node is not None: Shortest Path: ['A', 'B', 'E', 'G', 'T']
         path.append(node)          node = parent[node]     path.reverse()

    return path path = bfs('A', 'T') print("Shortest Path:", path) Shortest Path: ['A', 'B', 'E',
'G', 'T']
```

Comparison measures for BFS:

Execution Time:

Fast for small graphs, but grows quickly as the graph gets bigger (O(V + E)).

Memory Usage:

High memory usage because it stores many nodes in the queue.

Success Rate:

Always finds a solution if one exists (complete algorithm).

Solution Optimality:

Guarantees the shortest path in unweighted graphs.

Scalability:

Does not scale well for large graphs due to memory growth.

Advantages:

Guarantees the shortest path.

Complete algorithm.

Deterministic.

Simple to implement.

Expands level-by-level.

Great for unweighted maps / grids.

Disadvantages:

High memory usage.

Slow on large graphs.

Works only for unweighted graphs.

Not suitable for dynamic environments.

Can be expensive in branching factor.

Uniform Cost Search (UCS)

“Drone Delivery Route Problem with Uniform Cost Search Algorithm”

-What is Uniform Cost Search: -

.Uniform Cost Search is an uninformed search algorithm that expands the node with the lowest total path cost from the start node to the goal. The algorithm guarantees finding the optimal solution when all costs are non-negative.

-Why UCS in Drone Delivery Problem: -

.Uniform Cost Search is suitable for the Drone Driven Route problem because the costs between routes are not equal. UCS ensures that the drone always chooses the path with the lowest total cost, which is critical for minimizing battery usage or travel time.

-How UCS Work: -

Initialization: The algorithm starts by adding the start node to the priority queue with a cumulative cost of 0.

Expansion: In each iteration, it removes the node with the lowest cost from the priority queue.

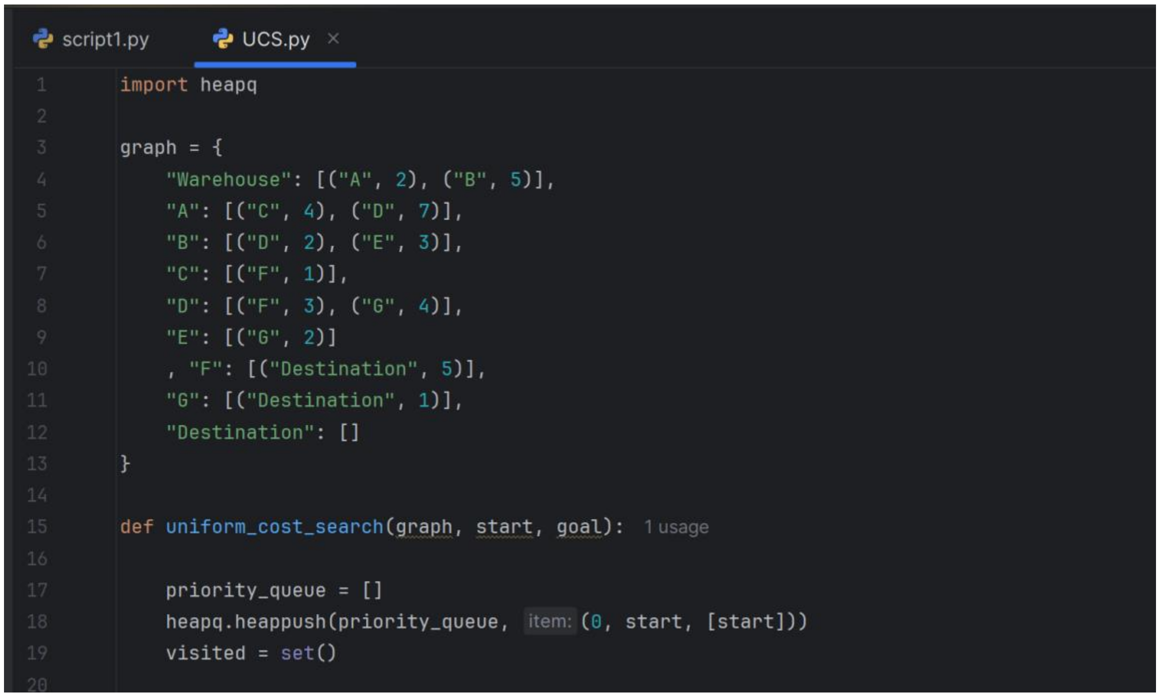
Goal Check: If the removed node is the goal node, the algorithm terminates and returns the path.

Neighbor Exploration: Otherwise, it expands the node and explores all its unvisited neighbors. It calculates the new cumulative cost for each neighbor by adding the current node's cost to the edge cost.

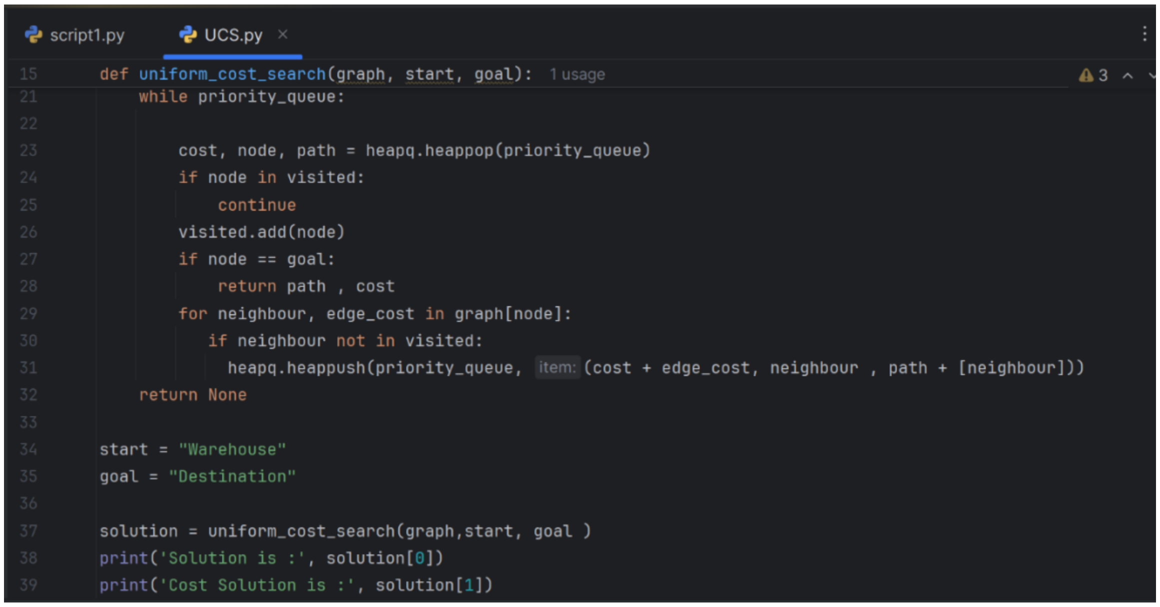
Queue Update: Neighbors are added to the priority queue. If a neighbor is already in the queue but a cheaper path to it is discovered, the algorithm updates its cost in the queue (a "decrease key" operation).

Repetition: Steps 2-5 are repeated until the goal is found, or the queue is empty.

-Python Implementation: -

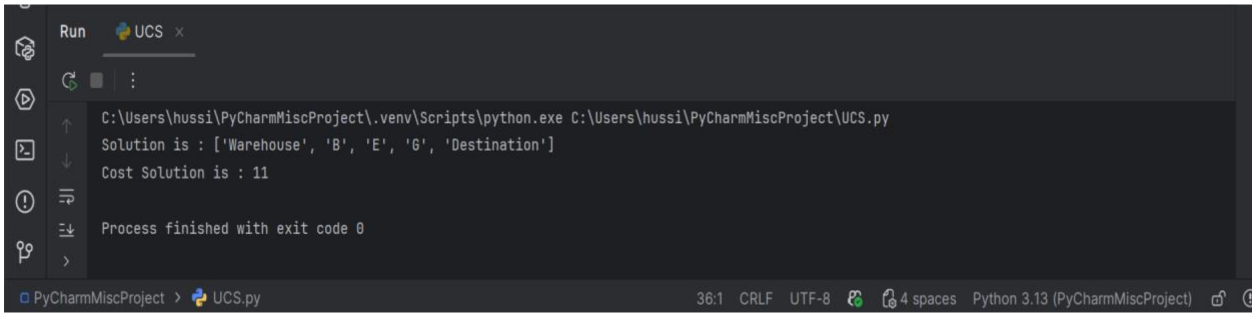


```python
import heapq

graph = {
    "Warehouse": [("A", 2), ("B", 5)],
    "A": [("C", 4), ("D", 7)],
    "B": [("D", 2), ("E", 3)],
    "C": [("F", 1)],
    "D": [("F", 3), ("G", 4)],
    "E": [("G", 2)]
    , "F": [("Destination", 5)],
    "G": [("Destination", 1)],
    "Destination": []
}

def uniform_cost_search(graph, start, goal):  1 usage

    priority_queue = []
    heapq.heappush(priority_queue,  item: (0, start, [start]))
    visited = set()
```



```python
def uniform_cost_search(graph, start, goal):  1 usage
    while priority_queue:

        cost, node, path = heapq.heappop(priority_queue)
        if node in visited:
            continue
        visited.add(node)
        if node == goal:
            return path , cost
        for neighbour, edge_cost in graph[node]:
           if neighbour not in visited:
             heapq.heappush(priority_queue,  item: (cost + edge_cost, neighbour , path + [neighbour]))
    return None

start = "Warehouse"
goal = "Destination"

solution = uniform_cost_search(graph,start, goal )
print('Solution is :', solution[0])
print('Cost Solution is :', solution[1])
```

-Code Output: -



```
C:\Users\hussi\PyCharmMiscProject\.venv\Scripts\python.exe C:\Users\hussi\PyCharmMiscProject\UCS.py
Solution is : ['Warehouse', 'B', 'E', 'G', 'Destination']
Cost Solution is : 11

Process finished with exit code 0
```

Run
UCS
PyCharmMiscProject > UCS.py
36:1 CRLF UTF-8 4 spaces Python 3.13 (PyCharmMiscProject)

-Advantages: -

1- Guarantees the optimal solution.

2- Works well when path costs vary.

3- Simple and reliable for cost-based path-finding problems.

-Disadvantages: -

1- Can be slow for large graphs.

2- Requires high memory usage.

3- Less efficient compared to heuristic-based algorithms such as A*.

-Comparation Measures For UCS: -

.Execution Time:

Time Complexity: O(E log V)

Where:

V is the number of vertices (locations)

E is the number of edges (routes)

In the Drone Driven Route problem, the execution time increases as the number of possible routes increases, making UCS suitable for small to medium-sized environments.

.Memory Usage:

Uniform Cost Search stores all generated nodes and paths in memory until the goal is reached.

Space Complexity: O(V)

For the drone routing problem, memory usage can become significant if the environment is large or highly connected.

.Success Rate:

Uniform Cost Search has a high success rate.

It always finds a solution if one exists.

In the Drone Driven Route problem, UCS reliably finds a valid delivery route whenever one exists.

.Completeness:

Uniform Cost Search is a complete algorithm.

It is guaranteed to find a solution if one exists

This holds as long as all path costs are non-negative

.Optimality:

Uniform Cost Search is optimal.

It always returns the path with the minimum total cost.

This is one of the main reasons it is used in cost-sensitive problems.

For the Drone Driven Route problem, this ensures minimal battery consumption, travel time, or distance.

.Scalability:

Uniform Cost Search has limited scalability.

Performance degrades as the size of the graph increases.

It explores many nodes without using heuristics.

For large-scale drone delivery systems, heuristic-based algorithms such as A* are more efficient and scalable.

A* Search Algorithm

A* Search Algorithm Definition:
A* is an intelligent search algorithm used to find the shortest path between a start node and a goal node in a graph or grid. It combines the actual cost from the start (g(n)) with a heuristic estimate of the remaining cost (h(n)) to prioritize which nodes to explore

Why A* in the Drone Delivery Problem?

Considers the actual travel cost between delivery locations.

Uses a heuristic function to guide the search toward promising routes.

Reduces unnecessary exploration compared to DFS and BFS.

Can guarantee an optimal solution when an admissible heuristic is used.

Helps minimize delivery distance, time, and energy consumption.

Provides a good balance between solution quality and computational efficiency.

How Does A* Work?

A* works by systematically exploring paths while always choosing the most promising option based on both the actual cost so far and an estimated remaining cost. At each step, A* evaluates nodes using the formula**: $f(n)=g(n)+h(n)$**

Step-by-step process:

Start node initialization

The start node is placed in a **priority queue.**

Its cost so far g(n) is 0.

Its heuristic h(n) estimates the remaining distance to the goal.

Node selection

A* always selects the node with the lowest f(n) value from the queue.

This node is considered the most promising to lead to an optimal solution.

Node expansion

The selected node is expanded by generating its neighboring nodes.

For each neighbor:

Update the actual cost g(n).

Compute the heuristic h(n).

Calculate the total score f(n).

Queue update

All newly generated nodes are added to the priority queue.

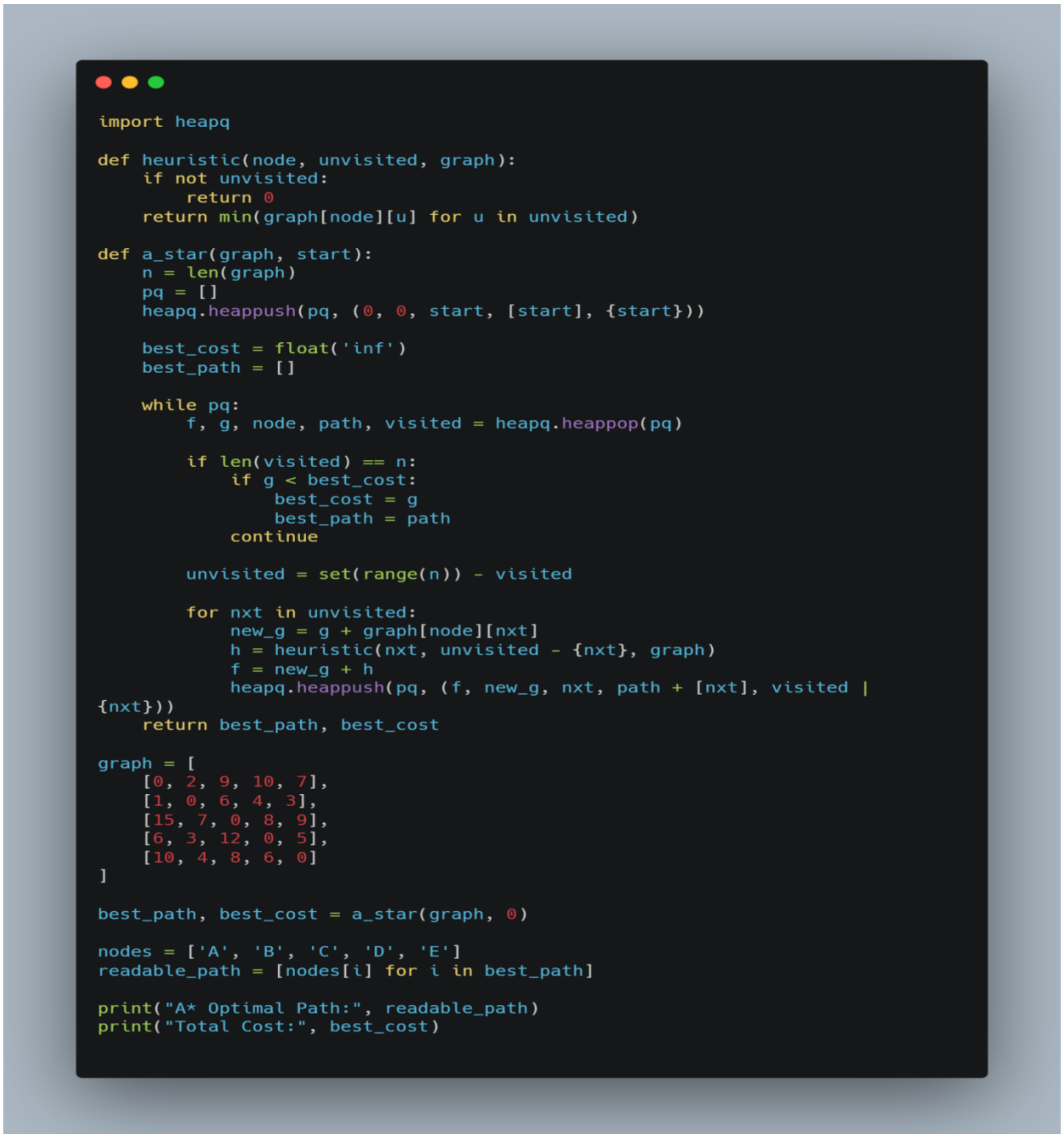
Nodes with lower f(n) values are explored first.

Goal check

If the goal node is reached (or all required nodes are visited), A* stops.

The path with the lowest total cost is returned

Python Implemenation



```python
import heapq

def heuristic(node, unvisited, graph):
    if not unvisited:
        return 0
    return min(graph[node][u] for u in unvisited)

def a_star(graph, start):
    n = len(graph)
    pq = []
    heapq.heappush(pq, (0, 0, start, [start], {start}))

    best_cost = float('inf')
    best_path = []

    while pq:
        f, g, node, path, visited = heapq.heappop(pq)

        if len(visited) == n:
            if g < best_cost:
                best_cost = g
                best_path = path
            continue

        unvisited = set(range(n)) - visited

        for nxt in unvisited:
            new_g = g + graph[node][nxt]
            h = heuristic(nxt, unvisited - {nxt}, graph)
            f = new_g + h
            heapq.heappush(pq, (f, new_g, nxt, path + [nxt], visited |
{nxt}))
    return best_path, best_cost

graph = [
    [0, 2, 9, 10, 7],
    [1, 0, 6, 4, 3],
    [15, 7, 0, 8, 9],
    [6, 3, 12, 0, 5],
    [10, 4, 8, 6, 0]
]

best_path, best_cost = a_star(graph, 0)

nodes = ['A', 'B', 'C', 'D', 'E']
readable_path = [nodes[i] for i in best_path]

print("A* Optimal Path:", readable_path)
print("Total Cost:", best_cost)
```

Advantages

Finds the shortest / least-cost path

Uses a heuristic to guide the search efficiently

Expands fewer nodes than DFS and BFS

Guarantees optimality with an admissible heuristic

Suitable for weighted graphs (real distances)

Disadvantages

Requires more memory than DFS

Performance depends heavily on heuristic quality

Can become slow for very large search spaces

Designing a good heuristic may be difficult

1.Execution Time

A* is generally faster than DFS and BFS because it explores only the most promising routes. However, with a weak heuristic, its performance may degrade

O(b^d)

2.Memory Usage

A* uses high memory because it stores:

Priority Queue

Multiple partial paths

Cost information for each node

O(b^d)

3.Success Rate

A* always finds a solution if one exists,

provided the graph is finite and connected.

4. Solution Optimality

A* produces an optimal solution when using an

admissible heuristic.

It guarantees the shortest or least-cost route.

5. Scalability

A* scales better than DFS and BFS,

but for very large problems (many delivery points),

its memory usage can become a limitation.

Greedy Search Algorithm

Greedy algorithms are a class of algorithms that make locally optimal choices at each step with the hope of finding a global optimum solution.

Why Greedy algorithms in the Drone Delivery Problem?

The Greedy algorithm was chosen due to its simplicity, low execution time, and suitability for real-time decision-making in drone delivery applications

How does the Greedy Algorithm work?

Start with the initial state of the problem. This is the starting point from where you begin making choices.

Evaluate all possible choices you can make from the current state. Consider all the options available at that specific moment.

Choose the option that seems best at that moment, regardless of future consequences. This is the "greedy" part - you take the best option available now, even if it might not be the best in the long run.

Move to the new state based on your chosen option. This becomes your new starting point for the next iteration.

Repeat steps 2-4 until you reach the goal state or no further progress is possible. Keep making the best local choices until you reach the end of the problem or get stuck.

Advantages

Simplicity & Ease of Implementation: They are straightforward to understand and code, requiring fewer lines of code.

Speed & Efficiency: Generally faster than other methods (like dynamic programming) because they make quick, local decisions without backtracking.

Resource-Efficient: Use minimal memory and computational power, ideal for mobile or embedded systems.

Near-Optimal Solutions: Can provide good enough results quickly when a perfect solution is impractical or too slow.

Disadvantages

Not Always Optimal: The biggest flaw; local best choices don't always lead to the global best, resulting in suboptimal outcomes (e.g., coin change problem).

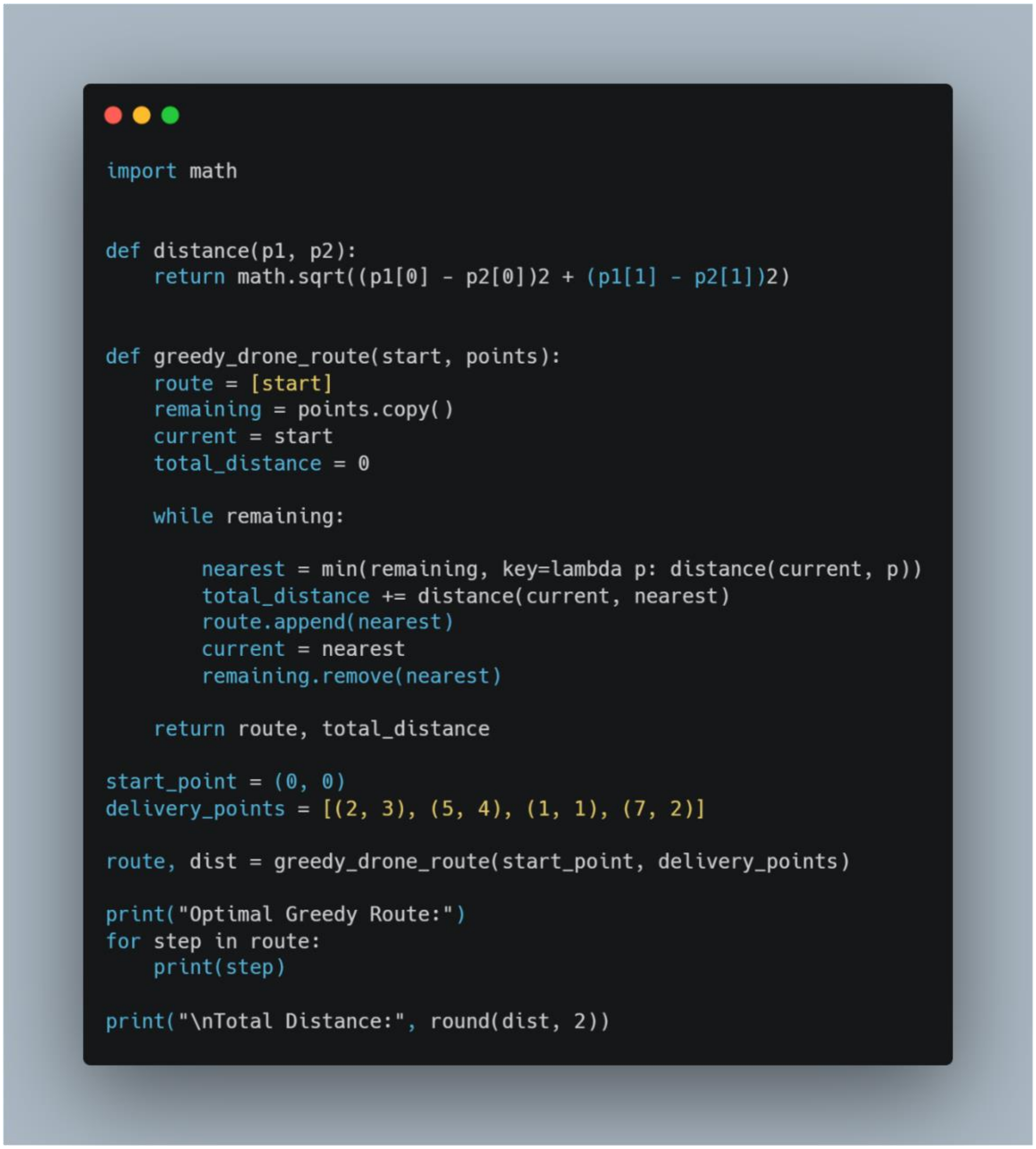
Limited Applicability: Only work for problems with specific properties (Greedy Choice Property and Optimal Substructure), failing on complex problems needing global foresight.

Difficult to Prove Correctness: Proving a greedy algorithm works perfectly can be mathematically challenging.

Sensitivity to Input: The order of data can sometimes influence the final result.

The idea: The drone starts from the starting point, then at each step it visits the nearest location that has not yet been visited.

Python Implementation



```python
import math


def distance(p1, p2):
    return math.sqrt((p1[0] - p2[0])2 + (p1[1] - p2[1])2)


def greedy_drone_route(start, points):
    route = [start]
    remaining = points.copy()
    current = start
    total_distance = 0

    while remaining:

        nearest = min(remaining, key=lambda p: distance(current, p))
        total_distance += distance(current, nearest)
        route.append(nearest)
        current = nearest
        remaining.remove(nearest)

    return route, total_distance

start_point = (0, 0)
delivery_points = [(2, 3), (5, 4), (1, 1), (7, 2)]

route, dist = greedy_drone_route(start_point, delivery_points)

print("Optimal Greedy Route:")
for step in route:
    print(step)

print("\nTotal Distance:", round(dist, 2))
```

Comparison measures for greedy algo:

Execution Time

The Greedy algorithm has low execution time, making it suitable for real-time drone routing decisions.

$O(n^2)$

Fast

Suitable for Real-Time

Slightly slower if the number of points is very large

Memory Usage

The Greedy approach requires minimal memory, which is suitable for resource-constrained drone systems

$O(n)$ Memory

Success Rate

The Greedy algorithm successfully visits all delivery points under standard conditions.

Success Rate $\approx$ 100%

Decreases if there are complex restrictions (Battery / No-Fly Zones)

Solution Optimality

While the Greedy algorithm does not guarantee an optimal solution, it often produces near-optimal routes.

Not Optimal

Near Optimal

located in Local Optimum

Scalability

The Greedy algorithm scales reasonably well for small to medium-sized delivery scenarios.

Genetic Algorithm (GA)

What is the Genetic Algorithm?

A GA is an optimization technique inspired by the principles of natural selection, and genetics.

It is particularly effective for complex search spaces where exhaustive search is impractical.

Why Use Genetic in the Drone Delivery Problem?

Efficiently finds near-optimal routes among a very large number of possible paths.

Handles complex and nonlinear constraints, such as distance limits or multiple delivery points.

Avoid local optima using mutation and crossover.

Scales better than classical search algorithms for large graphs.

GA Design Details

Solution Representation

Each solution is represented as a permutation of delivery points, ensuring:

Each location is visited exactly once.

No duplicate nodes exist in a route.

Fitness Function

It evaluates the total distance of a route.

The objective is to minimize total travel distance, meaning shorter routes have higher fitness.

Population Initialization

The algorithm starts with a random population of routes, ensuring diversity in the initial search space.

Elitism

The best solution from each generation is carried forward unchanged to ensure solution quality doesn’t degrade.

Selection Method

Tournament Selection is used to choose parent routes.

These balances:

Exploitation of good solutions.

Exploration of new areas in the search space.

Crossover Operator

Order Crossover is applied to combine two parent routes while preserving the relative order of nodes.

Mutation Operator

Swap Mutation randomly exchanges two delivery points in a route to maintain population diversity and prevent premature convergence by enabling exploration of new regions of the search space.

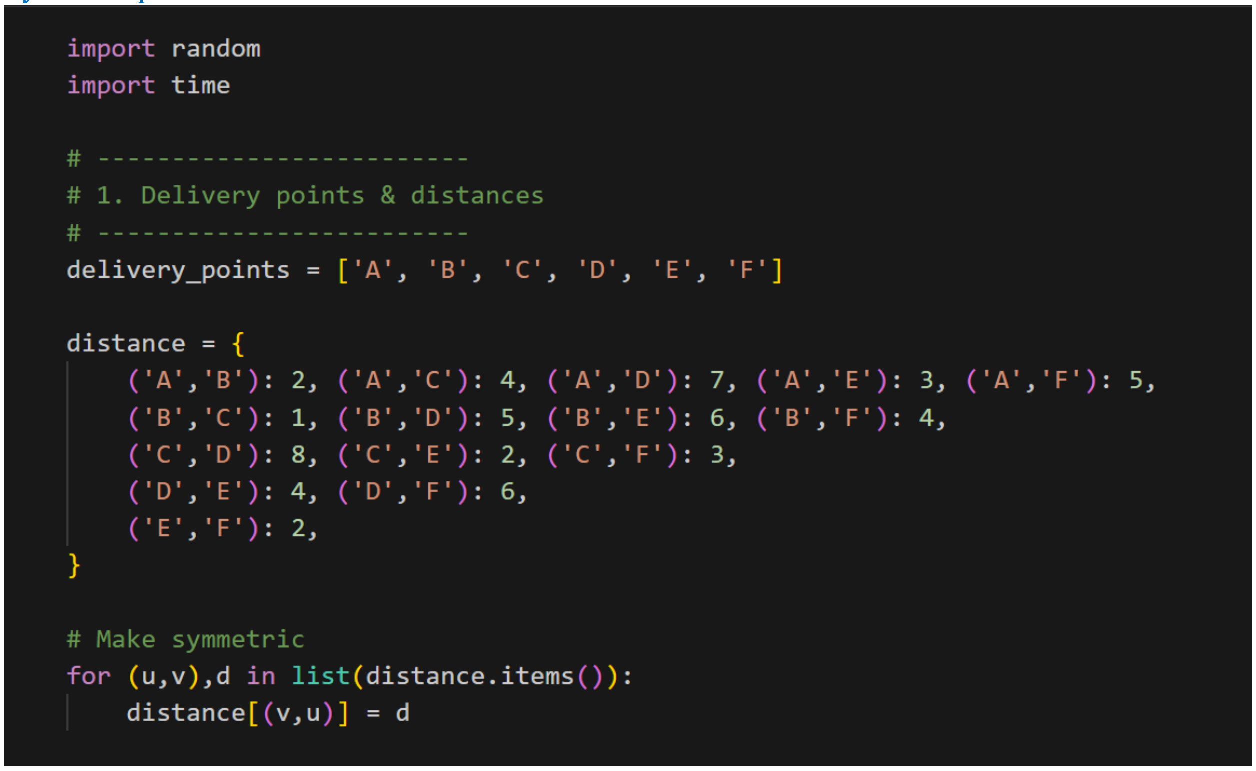
Termination Criteria

The algorithm stops when:

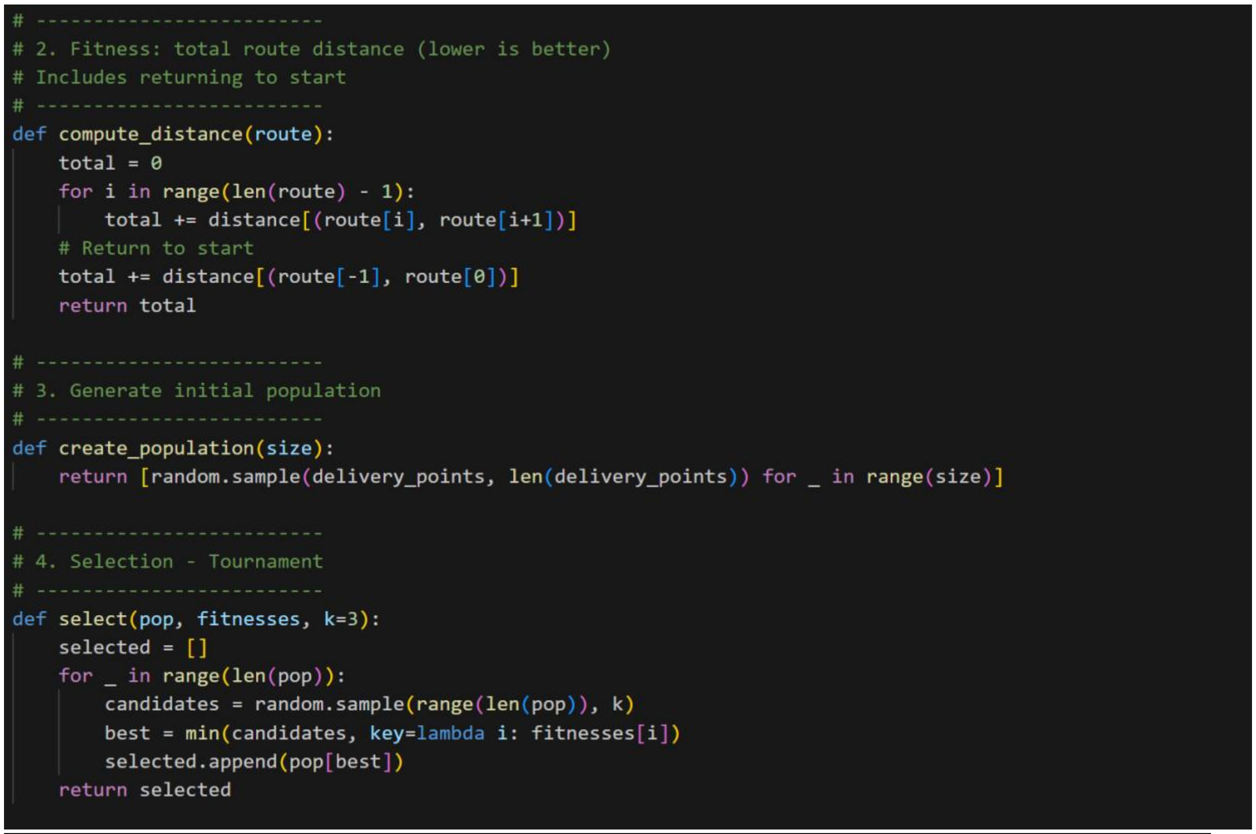
A fixed number of generations is reached.

No significant improvement is observed over multiple generations.

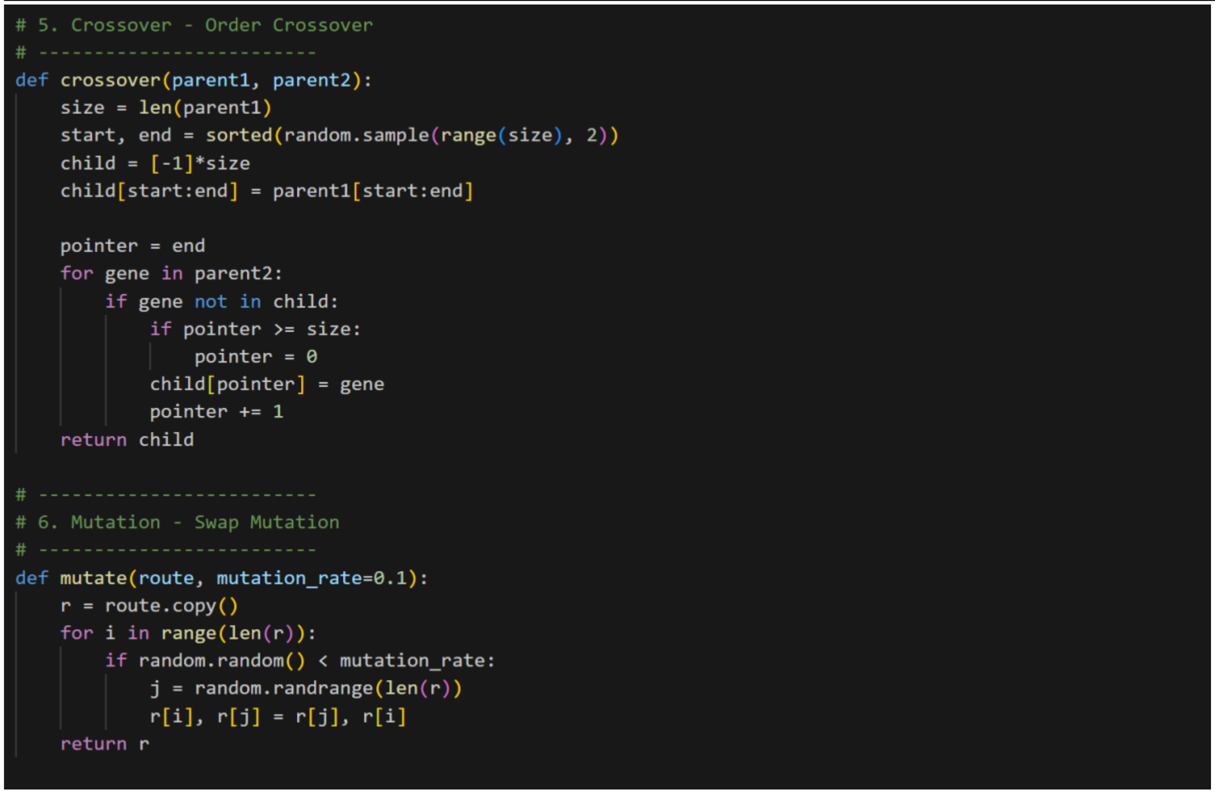
Python Implementation



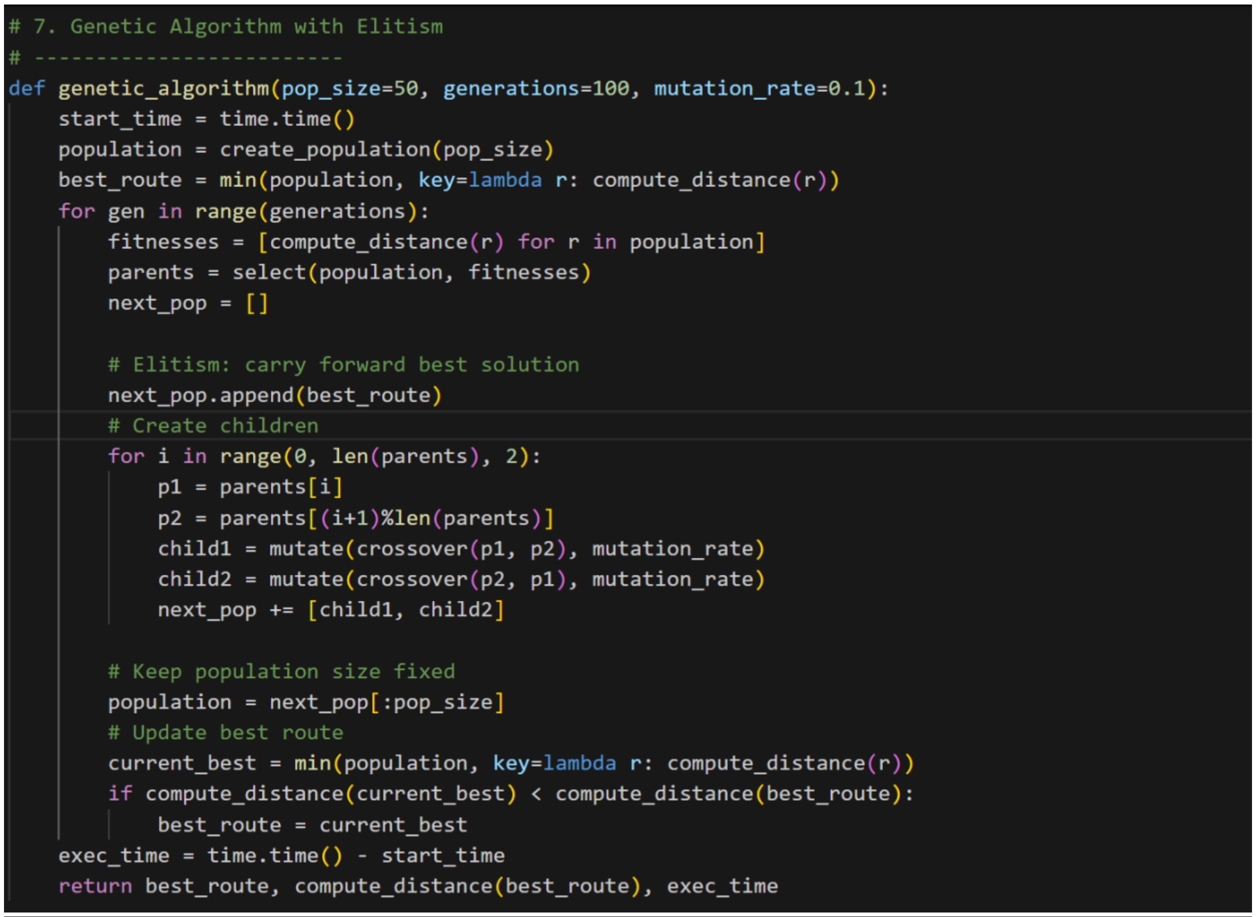
```python
import random
import time

# -------------------------
# 1. Delivery points & distances
# -------------------------
delivery_points = ['A', 'B', 'C', 'D', 'E', 'F']

distance = {
    ('A','B'): 2, ('A','C'): 4, ('A','D'): 7, ('A','E'): 3, ('A','F'): 5,
    ('B','C'): 1, ('B','D'): 5, ('B','E'): 6, ('B','F'): 4,
    ('C','D'): 8, ('C','E'): 2, ('C','F'): 3,
    ('D','E'): 4, ('D','F'): 6,
    ('E','F'): 2,
}

# Make symmetric
for (u,v),d in list(distance.items()):
    distance[(v,u)] = d
```



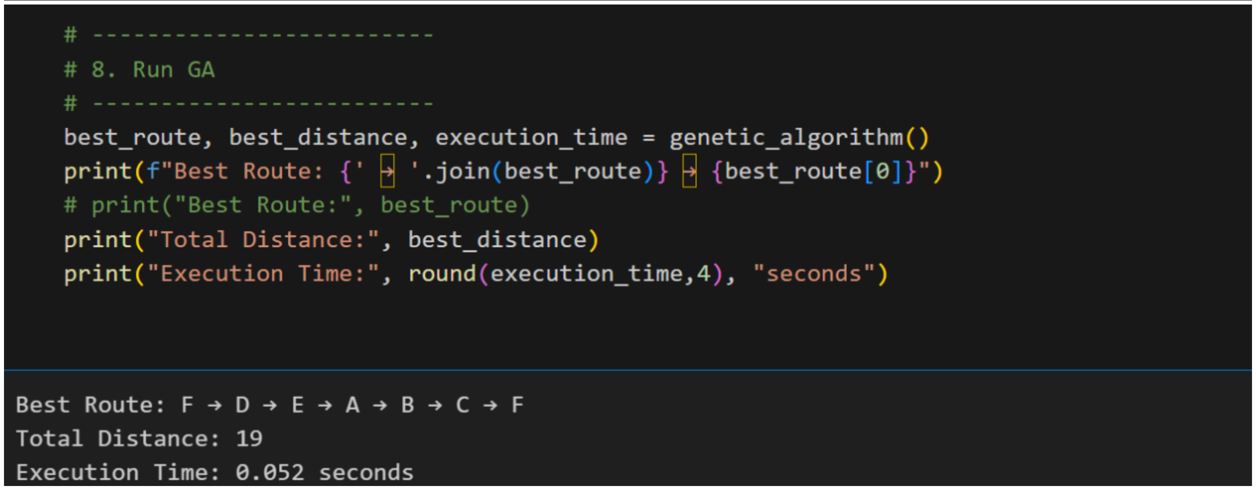
```python
# ------------------------
# 2. Fitness: total route distance (lower is better)
# Includes returning to start
# ------------------------
def compute_distance(route):
    total = 0
    for i in range(len(route) - 1):
        total += distance[(route[i], route[i+1])]
    # Return to start
    total += distance[(route[-1], route[0])]
    return total

# ------------------------
# 3. Generate initial population
# ------------------------
def create_population(size):
    return [random.sample(delivery_points, len(delivery_points)) for _ in range(size)]

# ------------------------
# 4. Selection - Tournament
# ------------------------
def select(pop, fitnesses, k=3):
    selected = []
    for _ in range(len(pop)):
        candidates = random.sample(range(len(pop)), k)
        best = min(candidates, key=lambda i: fitnesses[i])
        selected.append(pop[best])
    return selected
```



```python
# 5. Crossover - Order Crossover
# ------------------------
def crossover(parent1, parent2):
    size = len(parent1)
    start, end = sorted(random.sample(range(size), 2))
    child = [-1]*size
    child[start:end] = parent1[start:end]

    pointer = end
    for gene in parent2:
        if gene not in child:
            if pointer >= size:
                pointer = 0
            child[pointer] = gene
            pointer += 1
    return child

# ------------------------
# 6. Mutation - Swap Mutation
# ------------------------
def mutate(route, mutation_rate=0.1):
    r = route.copy()
    for i in range(len(r)):
        if random.random() < mutation_rate:
            j = random.randrange(len(r))
            r[i], r[j] = r[j], r[i]
    return r
```



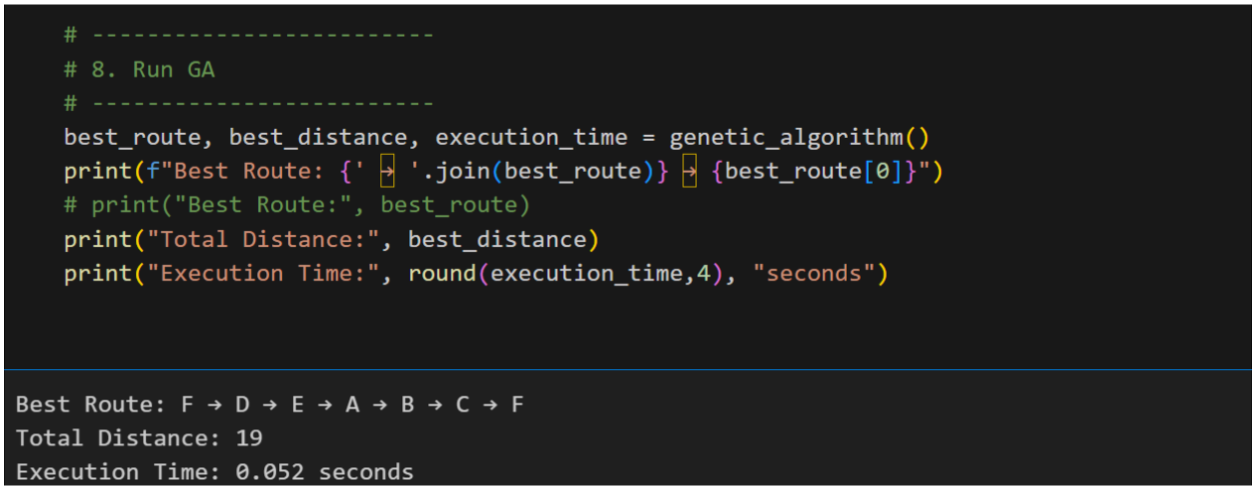
```python
# 7. Genetic Algorithm with Elitism
# -------------------------
def genetic_algorithm(pop_size=50, generations=100, mutation_rate=0.1):
    start_time = time.time()
    population = create_population(pop_size)
    best_route = min(population, key=lambda r: compute_distance(r))
    for gen in range(generations):
        fitnesses = [compute_distance(r) for r in population]
        parents = select(population, fitnesses)
        next_pop = []

        # Elitism: carry forward best solution
        next_pop.append(best_route)
        # Create children
        for i in range(0, len(parents), 2):
            p1 = parents[i]
            p2 = parents[(i+1)%len(parents)]
            child1 = mutate(crossover(p1, p2), mutation_rate)
            child2 = mutate(crossover(p2, p1), mutation_rate)
            next_pop += [child1, child2]

        # Keep population size fixed
        population = next_pop[:pop_size]
        # Update best route
        current_best = min(population, key=lambda r: compute_distance(r))
        if compute_distance(current_best) < compute_distance(best_route):
            best_route = current_best
    exec_time = time.time() - start_time
    return best_route, compute_distance(best_route), exec_time
```



```python
# -------------------------
# 8. Run GA
# -------------------------
best_route, best_distance, execution_time = genetic_algorithm()
print(f"Best Route: {' → '.join(best_route)} → {best_route[0]}")
# print("Best Route:", best_route)
print("Total Distance:", best_distance)
print("Execution Time:", round(execution_time,4), "seconds")
```

```
Best Route: F → D → E → A → B → C → F
Total Distance: 19
Execution Time: 0.052 seconds
```

Output



```
# ------------------------
# 8. Run GA
# ------------------------
best_route, best_distance, execution_time = genetic_algorithm()
print(f"Best Route: {' → '.join(best_route)} → {best_route[0]}")
# print("Best Route:", best_route)
print("Total Distance:", best_distance)
print("Execution Time:", round(execution_time,4), "seconds")
```

```
Best Route: F → D → E → A → B → C → F
Total Distance: 19
Execution Time: 0.052 seconds
```

GA finds a feasible, near-optimal route visiting all delivery points.

Minimizes total travel distance efficiently.

Produces practical paths ready for implementation.

Genetic Algorithm (GA) Evaluation Metrics

Execution Time

GA may take longer than simple search algorithms because it evaluates many solutions over multiple generations.

Memory Usage

It depends on population size and number of delivery points.

Success Rate

GA reliably finds feasible routes by visiting all delivery points.

Solution Optimality

GA produces near optimal or optimal routes, especially for larger complex graphs.

Scalability

By adjusting population size and number of generations, it can handle more delivery points without exploring unnecessary paths.

Hill climbing Algorithm

Hill climbing is a heuristic search algorithm that belongs to the family of local search methods. It is designed to solve problems where the goal is to find an optimal (or near-optimal) solution by iteratively moving from the current state to a better neighboring state, according to a heuristic or evaluation function.

Why in the Drone Delivery Problem?

Hill Climbing was selected because it enhances an initial solution by continuously searching for better neighboring solutions. It provides a balance between execution speed and solution quality, making it suitable for optimizing drone delivery routes without high computational cost.

How does the Hill climbing Algorithm work?

1. Initial State: Start with an arbitrary or random solution (initial state).
2. Neighboring States: Identify neighboring states of the current solution by making small adjustments (mutations or tweaks).
3. Move to Neighbor: If one of the neighboring states offers a better solution (according to some evaluation function), move to this new state.
4. Termination: Repeat this process until no neighboring state is better than the current one. At this point, we have reached a local maximum or minimum

Advantages

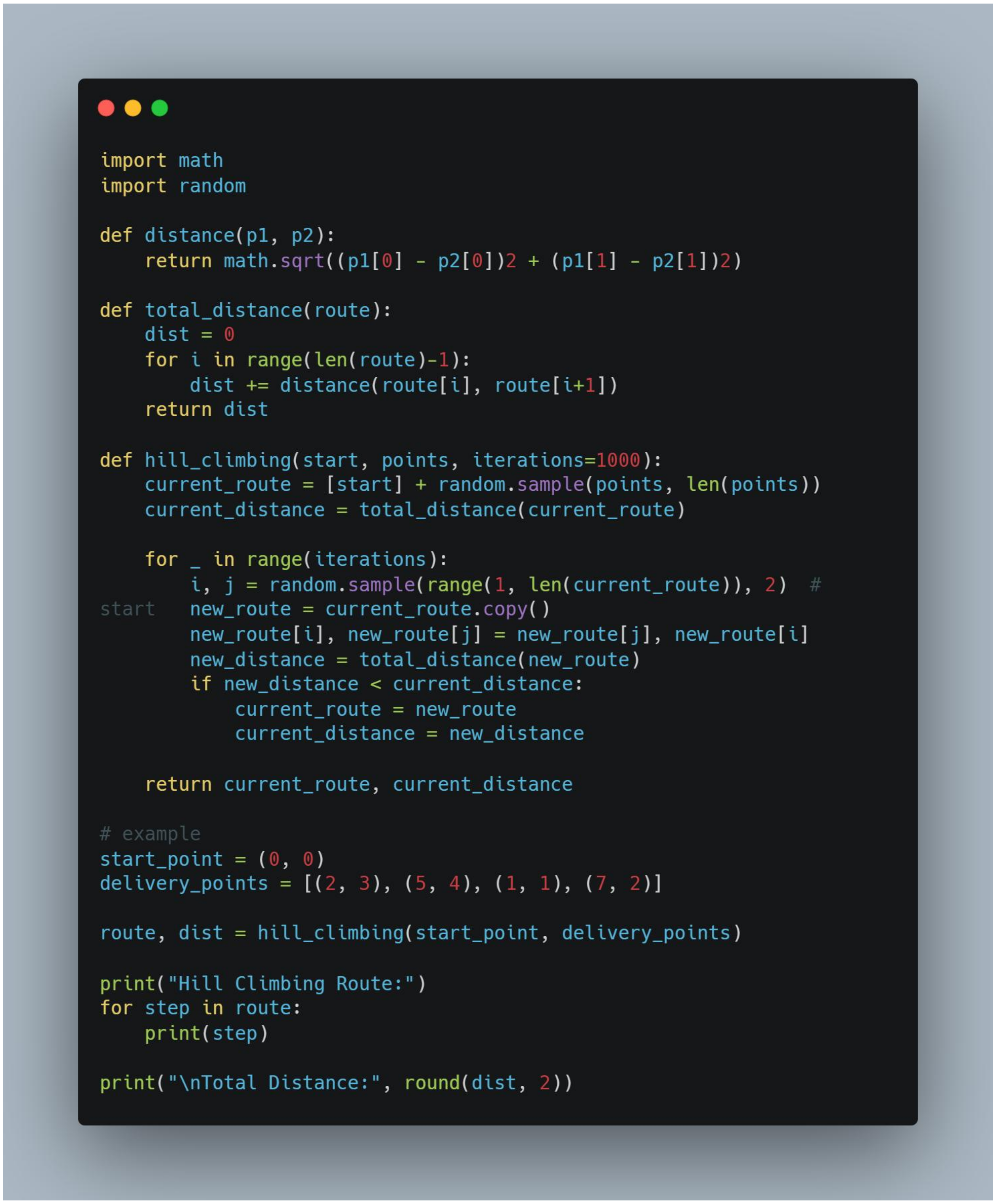
- **Simplicity & Intuitive:** Easy to understand, implement, and explain.
- **Memory Efficient:** Requires minimal memory as it only stores the current state.
- **Fast Convergence:** Quickly finds a good solution in time-sensitive scenarios.
- **Versatile:** Applicable to both continuous and discrete problems, including routing, circuit design, and scheduling.
- **Customizable:** Easily modified with heuristics to improve performance.

Disadvantages

- **Local Optima:** Most significant drawback; gets trapped in a peak that isn't the highest overall.
- **Plateaus & Ridges:** Stagnates on flat areas (plateaus) or narrow peaks (ridges) where improvement isn't clear.
- **No Guarantee of Global Optimum:** Fails to find the best possible solution, only a locally good one.
- **Start State Dependency:** The final result heavily relies on the initial starting point.

The idea: We start with a random path, then try to improve it by swapping two positions on the path if this swap reduces the total distance. We repeat the process until we find no further improvements.

Python Implementation



```python
import math
import random

def distance(p1, p2):
    return math.sqrt((p1[0] - p2[0])2 + (p1[1] - p2[1])2)

def total_distance(route):
    dist = 0
    for i in range(len(route)-1):
        dist += distance(route[i], route[i+1])
    return dist

def hill_climbing(start, points, iterations=1000):
    current_route = [start] + random.sample(points, len(points))
    current_distance = total_distance(current_route)

    for _ in range(iterations):
        i, j = random.sample(range(1, len(current_route)), 2)  #
start   new_route = current_route.copy()
        new_route[i], new_route[j] = new_route[j], new_route[i]
        new_distance = total_distance(new_route)
        if new_distance < current_distance:
            current_route = new_route
            current_distance = new_distance

    return current_route, current_distance

# example
start_point = (0, 0)
delivery_points = [(2, 3), (5, 4), (1, 1), (7, 2)]

route, dist = hill_climbing(start_point, delivery_points)

print("Hill Climbing Route:")
for step in route:
    print(step)

print("\nTotal Distance:", round(dist, 2))
```

Comparison Measures for Hill climbing Algorithm:

Execution Time

Hill Climbing requires more execution time than Greedy due to iterative solution improvement.

Time depends on: Number of attempts, Number of neighbors

Time Complexity O $(k \times n^2)$

Memory Usage

Hill Climbing uses moderate memory, storing only the current and neighboring solutions.

Memory Complexity: O(n)

Success Rate

Hill Climbing generally succeeds in producing a valid route but may get stuck in local optima.

The solution isn't always the best; it can be improved by:

Random Restart

Sideways Moves

Solution Optimality

Hill Climbing improves solution quality compared to Greedy but does not guarantee global optimality.

The solution improves gradually.

Scalability

Hill Climbing scales moderately and performs well on medium-sized problems.

To explain:

When the number of points increases: The number of neighbors increases The time increases

Algorithm	Execution time	Memory usage	Success Rate	Solution Optimality	Scalability
Depth First Search (DFS)	DFS can be moderately fast because it goes deep quickly, but in large graphs it may waste time exploring long incorrect paths.	DFS uses very little memory because it only stores: • Stack • Visited nodes	DFS can reach the goal if a path exists. It may take a long or wrong route, but it eventually finds a path.	DFS does not produce an optimal route. It always returns the **first** path it finds, not the shortest or least-cost one.	DFS does not scale well. When the graph grows deeper or larger, DFS may explore unnecessary long paths before backtracking.
Breadth First Search (BFS)	Fast for small graphs, but grows quickly as the graph gets bigger (O(V + E)).	High memory usage because it stores many nodes in the queue	Always finds a solution if one exists (complete algorithm)	Guarantees the shortest path in unweighted graphs.	Does not scale well for large graphs due to memory growth.
Uniform Cost Search (UCS)	**Time Complexity: O(E log V)**	Uniform Cost Search stores all generated nodes and paths in memory until the goal is reached. • **Space Complexity: O(V)**	Uniform Cost Search has a **high success rate**. It always finds a solution if one exists	Uniform Cost Search is optimal. For the Drone Driven Route problem, this ensures minimal battery consumption, travel time, or distance.	Uniform Cost Search has limited scalability. For large-scale drone delivery systems, heuristic-based algorithms such as A* are more efficient and scalable.
A* Search	A* is generally faster than DFS and BFS because it explores only the most promising routes. However, with a weak heuristic, its performance may degrade **O(b^d)**	**A* uses high memory because it stores:** • **Priority Queue** • **Multiple partial paths** • **Cost information for each node** **O(b^d)**	A* always finds a solution **if one exists**, provided the graph is finite and connected.	A* produces an **optimal solution** when using an **admissible heuristic**. It guarantees the shortest or least-cost route.	A* scales **better than DFS and BFS**, but for very large problems (many delivery points), its memory usage can become a limitation.

Greedy Best-First Search	Uniform Cost Search has limited scalability. For large-scale drone delivery systems, heuristic based algorithms such as A* are more efficient and scalable	The Greedy approach requires minimal memory, which is suitable for resource constrained drone systems O(n) Memory	The Greedy approach requires minimal memory, which is suitable for resource constrained drone systems O(n) Memory	While the Greedy algorithm does not guarantee an optimal solution, it often produces near optimal routes. Not Optimal Near Optimal located in Local Optimum	The Greedy algorithm scales reasonably well for small to medium sized delivery scenarios.
Genetic Algorithm (GA)	GA may take longer than simple search algorithms because it evaluates many solutions over multiple generations.	Stores the population and fitness values; memory grows with population size and number of delivery points.	GA reliable finds a feasible route that visits all delivery points. Mutation and crossover help explore diverse solutions and avoid getting stuck.	GA produces near optimal or optimal routes, minimizing total distance or cost.	By adjusting population size and number of generations, it can handle more delivery points without exploring unnecessary paths.
Hill climbing	Hill Climbing requires more execution time than Greedy due to iterative solution improvement. Time depends on: Number of attempts, Number of neighbors Time Complexity O $(k \times n^2)$	Hill Climbing uses moderate memory, storing only the current and neighboring solutions. Memory Complexity: O(n)	Hill Climbing generally succeeds in producing a valid route but may get stuck in local optima. The solution isn't always the best; it can be improved by: Random Restart Sideways Moves	Hill Climbing improves solution quality compared to Greedy but does not guarantee global optimality. The solution improves gradually	Hill Climbing scales moderately and performs well on medium-sized problems. To explain: When the number of points increases: The number of neighbors increases The time increases

The best algorithm is **A* Search** because it provides an optimal solution with better performance and scalability compared to other search algorithms.