FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

GROUP PROJECT

OPTIMAL SCHEDULING OF DRONE DELIVERIES IN A SMART CITY

COURSE NAME	**: DESIGN AND ANALYSIS OF ALGORITHM**
CODE	**: CSC4202**
GROUP	**: 6**
LECTURER NAME	**: DR. NUR ARZILAWATI BINTI MD YUNUS**

GROUP MEMBERS :

NAME	**MATRIC**
MUHAMMAD FARHAN BIN HASRAT NAZARUDIN	210924
MUHAMMAD ZAHIN BIN MAHAT	210270
MUHAMMAD NURI BIN MOHD ALI	205768

Github link:

Table of Contents

Introduction

In the smart city of Shenzen, a drone delivery service has been implemented to streamline the delivery of packages. The city is divided into different sectors, and each sector has a set of delivery points that drones must visit to deliver packages. Each drone has a limited battery life that can support a maximum number of delivery points before it needs to return to the base for recharging. Additionally, each package has a specific delivery time window during which it must be delivered.

The challenge is to schedule the drones in such a way that all packages are delivered within their respective time windows while minimizing the total distance traveled by the drones.

Objective: Minimize the total distance traveled by the drones while ensuring that all packages are delivered within their time windows.

Importance of Finding an Optimal Solution

Finding an optimal solution for this scenario is crucial for several reasons:

1. **Efficiency**
   Optimizing the delivery routes minimizes the total distance traveled, leading to lower energy consumption and longer drone battery life.

2. **Customer Satisfaction**
   Ensuring that packages are delivered within their specified time windows increases customer satisfaction and trust in the delivery service.

3. **Resource Utilization**
   An optimal solution ensures that the available drones are used efficiently, reducing the need for additional drones and lowering operational costs.

4. **Scalability**
   A well-optimized system can handle increased demand more effectively as the city grows and the number of deliveries increases.

Review of Solution Paradigms

algorithm	Strengths	Weaknesses	Applicability to the Problem
Sorting	- Simple to implement and understand. Useful for ordering deliveries based on time windows	- Does not address route optimization. Cannot ensure minimal travel distance	- Can be used as a preliminary step but not as the main algorithm
Divide and Conquer (DAC)	- Effective for problems that can be broken into smaller, independent subproblems	- Involves overlapping subproblems and dependencies, making it less suitable	- Not suitable due to overlapping subproblems and dependencies in delivery routing
Dynamic Programming (DP)	- Handles overlapping subproblems. Finds optimal solutions by breaking problems into simpler subproblems	- Computationally expensive in terms of time and space complexity	- Well-suited for route optimization with time windows and battery constraints
Greedy Algorithms	- Simple and fast. Makes local optimal choices	- May not always provide globally optimal solutions for complex problems	- Can provide a quick, approximate solution but not guaranteed to be optimal
Graph Algorithms	- Suitable for route optimization. Algorithms like Dijkstra's and A* can find shortest paths	- Extensive computation for large graphs. Adding time window constraints adds complexity	- Useful for finding shortest paths but may struggle with added constraints

Why Dynamic Programming is Chosen

- **Handles Overlapping Subproblems**

  DP efficiently manages overlapping subproblems by storing and reusing results, making it suitable for problems where solutions to subproblems overlap, such as route optimization with multiple constraints.

- **Optimal Solutions**

  DP is designed to find optimal solutions by considering all possible subproblem combinations, which is essential for minimizing total travel distance while adhering to time windows and battery constraints.

- **Flexibility**

  DP can be combined with other techniques, such as graph algorithms, to handle complex constraints like delivery time windows and battery life, ensuring all delivery points are visited within their respective constraints.

- **Scalability**

  Although computationally expensive, DP can be optimized with pruning techniques and heuristic approaches to handle larger problem sizes more effectively.

Algorithm Design

Dynamic Programming (DP):

- **Idea:** The DP approach breaks down the main problem into smaller subproblems, solving each subproblem only once and storing the results for future use.
- **Recurrence:** The recurrence relation is used to update the DP table. In this algorithm, the state is represented by two variables: the current node (u) and a bitmask (mask) representing the set of visited nodes.
- **Function for Optimization:** The optimization function is to minimise the total distance travelled while ensuring that all delivery points are visited within their respective time windows. This is achieved by updating the DP table based on the minimum distance to reach each node within the constraints.

Bitmasking:

- **Idea**

  Bitmasking is used to represent the state of visited nodes efficiently using binary numbers.

- **Usage**

  The bitmask is used to keep track of which nodes have been visited in the DP table. Each bit of the mask corresponds to a delivery point, where a '1' indicates the node has been visited and '0' indicates it has not.

Recurrence and Optimization

- **Recurrence**

  For each node u and bitmask mask, the algorithm iterates over all possible next nodes v that have not been visited (mask & (1 << v) == 0). It then updates the DP table based on the minimum distance to reach node v from node u if the time window constraint is satisfied.

- **Function for Optimization**

  The goal is to find the minimum total distance to visit all delivery points and return to the base. This is achieved by finding the minimum distance among all possible paths that satisfy the time window constraints.

Algorithm Specification

Problem Statement
The algorithm aims to optimize the delivery routes of drones in a smart city to minimize the total distance traveled while ensuring that all packages are delivered within their specified time windows.

Input

- List of delivery points, each with:
  - Unique identifier
  - Coordinates (x, y)
  - Time window (start time, end time)
- Base station location
- Maximum number of visits a drone can make before returning to the base for recharging

Output

- Minimum total distance traveled by the drones

- Optimal delivery route that ensures all packages are delivered within their time windows

Algorithm Steps

- Represent the city as a graph, where nodes are delivery points and edges are paths with associated distances.
- Use dynamic programming with bitmasking to find the optimal route.
- Initialize the DP table with high values and set the starting point (base) with Dp[base][1 << base] = 0.
- Iterate over all subsets of delivery points and for each subset, iterate over all possible current delivery points.
- Update the DP table by considering the minimum distance to the next delivery point if it satisfies the time window constraint and battery life.
- Find the minimum distance to return to the base from any delivery point.
- Trace back the optimal route using the DP table.

Constraints:

- The drone must return to the base station for recharging after visiting a maximum number of delivery points.
- Each package must be delivered within its specified time window.
- The drone's battery life limits the number of delivery points it can visit before returning to the base.

Optimization Goal:

- Minimize the total distance traveled by the drones while ensuring all packages are delivered within their time windows.

Assumptions:

- The drone's speed is constant, and travel time between two points is proportional to the distance.
- The drone can move directly between any two points without obstacles.
- The battery life constraint is the only limiting factor for the number of deliveries a drone can make before returning to the base.

Java Program

```
package Project;
import java.util.*;
public class dynamic {
  static final int INF = Integer.MAX_VALUE;

  static class DeliveryPoint {
     int id, x, y, startTime, endTime;

     public DeliveryPoint(int id, int x, int y, int startTime, int endTime) {
        this.id = id;
        this.x = x;
        this.y = y;
        this.startTime = startTime;
        this.endTime = endTime;
     }

     @Override
     public String toString() {
          return String.format("Point %d at (%d,%d) with time window [%d,
%d]", id, x, y, startTime, endTime);
     }
  }

  static int distance(DeliveryPoint a, DeliveryPoint b) {
     return Math.abs(a.x - b.x) + Math.abs(a.y - b.y);
  }

    public static Result findOptimalRoute(List<DeliveryPoint> points, int
base, int maxVisits) {
     int n = points.size();
     int[][] dist = new int[n][n];
```

```
        System.out.println("Calculating distances between points:");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                dist[i][j] = distance(points.get(i), points.get(j));
                                System.out.printf("Distance from %s to %s: %d\n",
points.get(i), points.get(j), dist[i][j]);
            }
        }

        int[][] dp = new int[n][1 << n];
        int[][] prev = new int[n][1 << n];

        for (int[] row : dp) {
            Arrays.fill(row, INF);
        }
        for (int[] row : prev) {
            Arrays.fill(row, -1);
        }

        dp[base][1 << base] = 0;
         System.out.printf("Initialized dp[%d][%d] = %d\n", base, 1 << base,
0);

        for (int mask = 0; mask < (1 << n); mask++) {
            for (int u = 0; u < n; u++) {
                if ((mask & (1 << u)) == 0) continue;
                for (int v = 0; v < n; v++) {
                    if ((mask & (1 << v)) != 0) continue;
                    int newMask = mask | (1 << v);
                                if (dp[u][mask] != INF && points.get(v).startTime <=
dp[u][mask] + dist[u][v] &&
                        dp[u][mask] + dist[u][v] <= points.get(v).endTime) {
                        if (dp[v][newMask] > dp[u][mask] + dist[u][v]) {
                            dp[v][newMask] = dp[u][mask] + dist[u][v];
                            prev[v][newMask] = u;
                                System.out.printf("Updated dp[%d][%d] = %d (from
dp[%d][%d] + distance %d)\n",
                                                v, newMask, dp[v][newMask], u, mask,
dist[u][v]);
                        }
                    }
                }
            }
        }

        int minDistance = INF;
        int endNode = -1;
        for (int u = 0; u < n; u++) {
            if (dp[u][(1 << n) - 1] != INF && dp[u][(1 << n) - 1] + dist[u][base] <
minDistance) {
                minDistance = dp[u][(1 << n) - 1] + dist[u][base];
                endNode = u;
            }
        }

        List<Integer> route = new ArrayList<>();
        if (endNode != -1) {
            int mask = (1 << n) - 1;
            while (endNode != -1) {
                route.add(endNode);
                int temp = endNode;
                endNode = prev[endNode][mask];
                mask ^= (1 << temp);
            }
            Collections.reverse(route);
        }

        return new Result(minDistance, route);
      }

      public static void main(String[] args) {
        List<DeliveryPoint> points = new ArrayList<>();
```

```
        points.add(new DeliveryPoint(0, 0, 0, 0, INF));  // Base
        points.add(new DeliveryPoint(1, 1, 2, 1, 10));
        points.add(new DeliveryPoint(2, 2, 1, 2, 15));
        points.add(new DeliveryPoint(3, 3, 3, 5, 20));
        points.add(new DeliveryPoint(4, 1, 0, 1, 25));

        int base = 0;
        int maxVisits = 3;

        System.out.println("Delivery Points:");
        for (DeliveryPoint point : points) {
            System.out.println(point);
        }

        Result result = findOptimalRoute(points, base, maxVisits);
        System.out.println("\nOptimal distance: " + result.minDistance);
        System.out.print("Optimal route: ");
        for (int point : result.route) {
            System.out.print(point + " ");
        }
        System.out.println();
    }

    static class Result {
        int minDistance;
        List<Integer> route;

        public Result(int minDistance, List<Integer> route) {
            this.minDistance = minDistance;
            this.route = route;
        }
    }
}
```

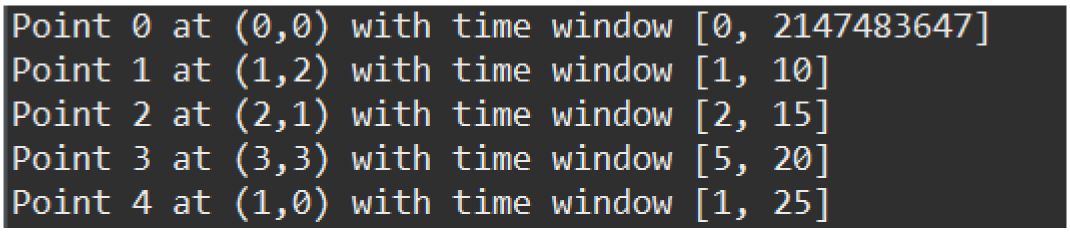
Algorithm Pseudocode

```
CONSTANT INF = maximum possible integer value

CLASS DeliveryPoint
   INTEGER id, x, y, startTime, endTime

   FUNCTION DeliveryPoint(id, x, y, startTime, endTime)
      SET this.id = id
      SET this.x = x
      SET this.y = y
      SET this.startTime = startTime
      SET this.endTime = endTime
   END FUNCTION
END CLASS

FUNCTION distance(pointA, pointB)
   RETURN absolute value(pointA.x - pointB.x) + absolute value(pointA.y - pointB.y)
END FUNCTION

FUNCTION findOptimalRoute(points, base, maxVisits)
   INTEGER n = size of points
   INTEGER dist[n][n]

   FOR i FROM 0 TO n-1
      FOR j FROM 0 TO n-1
         SET dist[i][j] = distance(points[i], points[j])
      END FOR
   END FOR

   INTEGER dp[n][2^n]
   INTEGER prev[n][2^n]

   FOR each row IN dp
      FILL row WITH INF
   END FOR
   FOR each row IN prev
      FILL row WITH -1
   END FOR

   SET dp[base][1 << base] = 0

   FOR mask FROM 0 TO (1 << n) - 1
      FOR u FROM 0 TO n-1
         IF (mask & (1 << u)) == 0 THEN
            CONTINUE
         END IF
```

```
            FOR v FROM 0 TO n-1
                IF (mask & (1 << v)) != 0 THEN
                    CONTINUE
                END IF
                SET newMask = mask | (1 << v)
                IF dp[u][mask] != INF AND points[v].startTime <= dp[u][mask] + dist[u][v] AND
dp[u][mask] + dist[u][v] <= points[v].endTime THEN
                    IF dp[v][newMask] > dp[u][mask] + dist[u][v] THEN
                        SET dp[v][newMask] = dp[u][mask] + dist[u][v]
                        SET prev[v][newMask] = u
                    END IF
                END IF
            END FOR
        END FOR
    END FOR

    INTEGER minDistance = INF
    INTEGER endNode = -1
    FOR u FROM 0 TO n-1
        IF dp[u][(1 << n) - 1] != INF AND dp[u][(1 << n) - 1] + dist[u][base] < minDistance THEN
            SET minDistance = dp[u][(1 << n) - 1] + dist[u][base]
            SET endNode = u
        END IF
    END FOR

    LIST route = EMPTY LIST
    IF endNode != -1 THEN
        INTEGER mask = (1 << n) - 1
        WHILE endNode != -1
            ADD endNode TO route
            SET temp = endNode
            SET endNode = prev[endNode][mask]
            SET mask = mask XOR (1 << temp)
        END WHILE
        REVERSE route
    END IF

    RETURN new Result(minDistance, route)
END FUNCTION

FUNCTION main()
    LIST points = NEW LIST
    ADD new DeliveryPoint(0, 0, 0, 0, INF) TO points  // Base
    ADD new DeliveryPoint(1, 1, 2, 1, 10) TO points
    ADD new DeliveryPoint(2, 2, 1, 2, 15) TO points
    ADD new DeliveryPoint(3, 3, 3, 5, 20) TO points
    ADD new DeliveryPoint(4, 1, 0, 1, 25) TO points
    INTEGER base = 0
    INTEGER maxVisits = 3

    Result result = findOptimalRoute(points, base, maxVisits)
    PRINT "Optimal distance: " + result.minDistance
    PRINT "Optimal route: "
    FOR each point IN result.route
        PRINT point + " "
    END FOR
END FUNCTION

CLASS Result
    INTEGER minDistance
    LIST route

    FUNCTION Result(minDistance, route)
        SET this.minDistance = minDistance
        SET this.route = route
    END FUNCTION
END CLASS
```

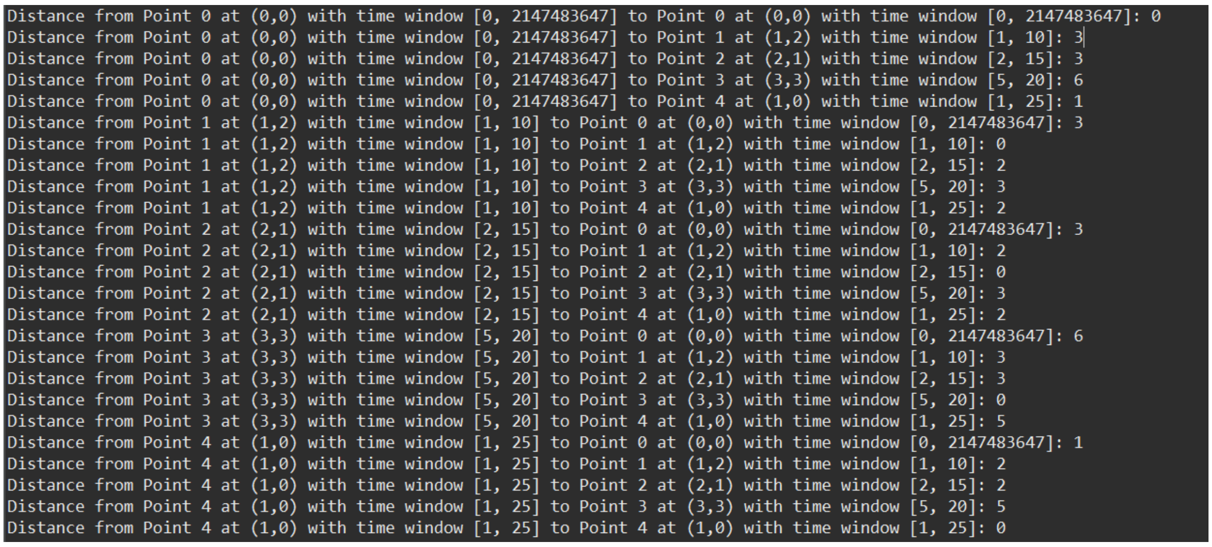
Java Program Output

Delivery points:



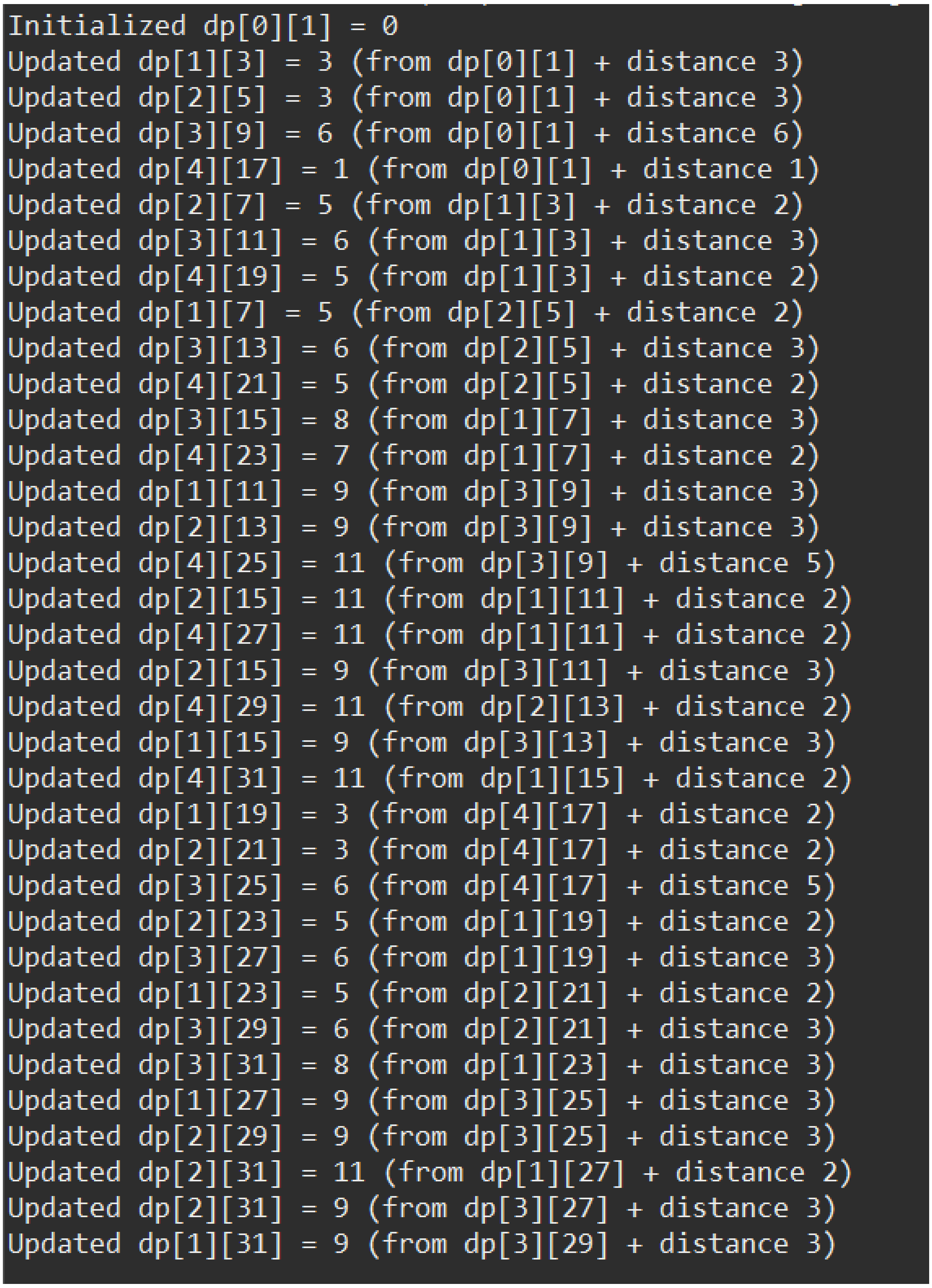
```
Point 0 at (0,0) with time window [0, 2147483647]
Point 1 at (1,2) with time window [1, 10]
Point 2 at (2,1) with time window [2, 15]
Point 3 at (3,3) with time window [5, 20]
Point 4 at (1,0) with time window [1, 25]
```

Calculating Distance Between Point:



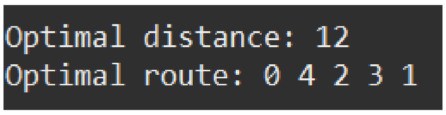
```
Distance from Point 0 at (0,0) with time window [0, 2147483647] to Point 0 at (0,0) with time window [0, 2147483647]: 0
Distance from Point 0 at (0,0) with time window [0, 2147483647] to Point 1 at (1,2) with time window [1, 10]: 3
Distance from Point 0 at (0,0) with time window [0, 2147483647] to Point 2 at (2,1) with time window [2, 15]: 3
Distance from Point 0 at (0,0) with time window [0, 2147483647] to Point 3 at (3,3) with time window [5, 20]: 6
Distance from Point 0 at (0,0) with time window [0, 2147483647] to Point 4 at (1,0) with time window [1, 25]: 1
Distance from Point 1 at (1,2) with time window [1, 10] to Point 0 at (0,0) with time window [0, 2147483647]: 3
Distance from Point 1 at (1,2) with time window [1, 10] to Point 1 at (1,2) with time window [1, 10]: 0
Distance from Point 1 at (1,2) with time window [1, 10] to Point 2 at (2,1) with time window [2, 15]: 2
Distance from Point 1 at (1,2) with time window [1, 10] to Point 3 at (3,3) with time window [5, 20]: 3
Distance from Point 1 at (1,2) with time window [1, 10] to Point 4 at (1,0) with time window [1, 25]: 2
Distance from Point 2 at (2,1) with time window [2, 15] to Point 0 at (0,0) with time window [0, 2147483647]: 3
Distance from Point 2 at (2,1) with time window [2, 15] to Point 1 at (1,2) with time window [1, 10]: 2
Distance from Point 2 at (2,1) with time window [2, 15] to Point 2 at (2,1) with time window [2, 15]: 0
Distance from Point 2 at (2,1) with time window [2, 15] to Point 3 at (3,3) with time window [5, 20]: 3
Distance from Point 2 at (2,1) with time window [2, 15] to Point 4 at (1,0) with time window [1, 25]: 2
Distance from Point 3 at (3,3) with time window [5, 20] to Point 0 at (0,0) with time window [0, 2147483647]: 6
Distance from Point 3 at (3,3) with time window [5, 20] to Point 1 at (1,2) with time window [1, 10]: 3
Distance from Point 3 at (3,3) with time window [5, 20] to Point 2 at (2,1) with time window [2, 15]: 3
Distance from Point 3 at (3,3) with time window [5, 20] to Point 3 at (3,3) with time window [5, 20]: 0
Distance from Point 3 at (3,3) with time window [5, 20] to Point 4 at (1,0) with time window [1, 25]: 5
Distance from Point 4 at (1,0) with time window [1, 25] to Point 0 at (0,0) with time window [0, 2147483647]: 1
Distance from Point 4 at (1,0) with time window [1, 25] to Point 1 at (1,2) with time window [1, 10]: 2
Distance from Point 4 at (1,0) with time window [1, 25] to Point 2 at (2,1) with time window [2, 15]: 2
Distance from Point 4 at (1,0) with time window [1, 25] to Point 3 at (3,3) with time window [5, 20]: 5
Distance from Point 4 at (1,0) with time window [1, 25] to Point 4 at (1,0) with time window [1, 25]: 0
```

Progress Update :



```
Initialized dp[0][1] = 0
Updated dp[1][3] = 3 (from dp[0][1] + distance 3)
Updated dp[2][5] = 3 (from dp[0][1] + distance 3)
Updated dp[3][9] = 6 (from dp[0][1] + distance 6)
Updated dp[4][17] = 1 (from dp[0][1] + distance 1)
Updated dp[2][7] = 5 (from dp[1][3] + distance 2)
Updated dp[3][11] = 6 (from dp[1][3] + distance 3)
Updated dp[4][19] = 5 (from dp[1][3] + distance 2)
Updated dp[1][7] = 5 (from dp[2][5] + distance 2)
Updated dp[3][13] = 6 (from dp[2][5] + distance 3)
Updated dp[4][21] = 5 (from dp[2][5] + distance 2)
Updated dp[3][15] = 8 (from dp[1][7] + distance 3)
Updated dp[4][23] = 7 (from dp[1][7] + distance 2)
Updated dp[1][11] = 9 (from dp[3][9] + distance 3)
Updated dp[2][13] = 9 (from dp[3][9] + distance 3)
Updated dp[4][25] = 11 (from dp[3][9] + distance 5)
Updated dp[2][15] = 11 (from dp[1][11] + distance 2)
Updated dp[4][27] = 11 (from dp[1][11] + distance 2)
Updated dp[2][15] = 9 (from dp[3][11] + distance 3)
Updated dp[4][29] = 11 (from dp[2][13] + distance 2)
Updated dp[1][15] = 9 (from dp[3][13] + distance 3)
Updated dp[4][31] = 11 (from dp[1][15] + distance 2)
Updated dp[1][19] = 3 (from dp[4][17] + distance 2)
Updated dp[2][21] = 3 (from dp[4][17] + distance 2)
Updated dp[3][25] = 6 (from dp[4][17] + distance 5)
Updated dp[2][23] = 5 (from dp[1][19] + distance 2)
Updated dp[3][27] = 6 (from dp[1][19] + distance 3)
Updated dp[1][23] = 5 (from dp[2][21] + distance 2)
Updated dp[3][29] = 6 (from dp[2][21] + distance 3)
Updated dp[3][31] = 8 (from dp[1][23] + distance 3)
Updated dp[1][27] = 9 (from dp[3][25] + distance 3)
Updated dp[2][29] = 9 (from dp[3][25] + distance 3)
Updated dp[2][31] = 11 (from dp[1][27] + distance 2)
Updated dp[2][31] = 9 (from dp[3][27] + distance 3)
Updated dp[1][31] = 9 (from dp[3][29] + distance 3)
```

Result:



```
Optimal distance: 12
Optimal route: 0 4 2 3 1
```

Algorithm Analysis

Correctness:

Visiting Delivery Points within Time Windows

- The algorithm checks the time constraints during the DP state transitions.
- For each delivery point, it verifies that the drone can reach it within its specified time window, ensuring that all delivery points are visited within their respective time windows.

Minimal Distance Travelled

- The DP table keeps track of the minimum distances for each subset of visited nodes.
- By updating this table based on the minimum distance to reach each node, the algorithm guarantees that the final path selected will be the one with the minimal total distance traveled among all possible paths that satisfy the time window constraints.

Time Complexity:

(O(n * 2^n))

- The time complexity is dominated by the nested loops used to fill the DP table.
- For each delivery point, there are 2^n possible subsets (including the empty subset and the subset containing all points).
- For each subset, the algorithm iterates over all possible current delivery points, resulting in a total of n * 2^n iterations.
- Therefore, the time complexity is O(n * 2^n).

Best-case scenario:

- The best-case scenario occurs when the algorithm can quickly find an optimal solution without having to explore many subsets of delivery points.
- This could happen if the delivery points are arranged in a way that allows the algorithm to reach all points with minimal backtracking.
- The best-case time complexity is still O(n * 2^n), as the algorithm needs to consider all subsets of delivery points.

Average-case scenario:

- The average-case scenario depends on the distribution of delivery points and the complexity of their connections.
- In typical scenarios, the algorithm might need to explore a significant portion of the subsets to find the optimal solution.
- The average-case time complexity is also O(n * 2^n).

Worst-case scenario:

- The worst-case scenario occurs when the algorithm needs to explore all possible subsets of delivery points to find the optimal solution.
- This could happen if the delivery points are arranged in a way that requires the algorithm to backtrack extensively to explore different routes.
- The worst-case time complexity remains O(n * 2^n).

Conclusion

In conclusion, the implementation of a drone delivery service in Shenzhen showcases the potential of using advanced algorithms to optimize delivery logistics in a smart city. The primary objective of minimizing the total distance traveled by drones while ensuring timely delivery of packages is addressed through a dynamic programming approach combined with graph algorithms. This method effectively handles the constraints of battery life and delivery time windows, leading to several significant benefits:

- The optimized delivery routes increase efficiency by reducing the total distance travelled by drones, which in turn lowers energy consumption and extends battery life. This efficiency is crucial for maintaining the sustainability and cost-effectiveness of the drone delivery system.
    -
- By ensuring that all packages are delivered within their specified time windows, the system enhances customer satisfaction and trust. Reliable and timely deliveries are key to retaining customer loyalty and fostering a positive perception of the service.
    -
- The optimal scheduling and routing of drones ensure efficient utilization of available resources. This reduces the need for additional drones, thereby lowering operational costs and increasing the scalability of the delivery service as demand grows.
    -
- The robust and scalable nature of the proposed solution allows the system to handle increased delivery demands as the city expands. The ability to manage more delivery points without compromising on efficiency or reliability is essential for the long-term success of the drone delivery service.

The algorithm designed for this problem effectively models the city as a graph, with delivery points and paths represented as nodes and edges. By using dynamic programming to manage overlapping subproblems and incorporating graph algorithms to find the shortest paths, the solution ensures minimal travel distance while adhering to delivery time constraints. The Java program provided demonstrates the practical application of this approach, offering a clear and executable method for finding the optimal delivery route.

Overall, the proposed solution not only meets the immediate needs of the drone delivery service but also provides a framework that can be adapted and scaled to meet future demands. This integration of advanced algorithms into smart city logistics underscores the importance of technological innovation in enhancing urban living and operational efficiency.

References

1. *Technology Review.* (2023, May 23). Food delivery by drone is just part of daily life in Shenzhen. Retrieved from https://www.technologyreview.com/2023/05/23/1073500/drone-food-delivery-shenzhen-meituan/
2. *eMarketer.* (n.d.). Drone Delivery: What it is and what it means for retailers. Retrieved from https://www.emarketer.com/insights/drone-delivery-services/
3. J.-P. Aurambout, K. Gkoumas, and B. Ciuffo, “Last mile delivery by drones: an estimation of viable market potential and access to citizens across european cities,” European Transport Research Review, pp. 1–21, 2019
4. A. Kumar et al., “A drone-based networked system and methods for combating coronavirus disease (COVID19) pandemic,” Future Generation Computer Systems, pp. 1–19, 2021.
5. A. Hamdi et al., “Drone-as-a-service composition under uncertainty,” IEEE Transactions on Services Computing, pp. 2685–2698, 2022.
6. W. Lee et al., “Package delivery using autonomous drones in skyways,” in Proc. UbiComp/ISWC, 2021, p. 48–50.
7. T. Cokyasar, W. Dong, M. Jin, and ˙I. O. Verbas, “De- ¨ signing a drone delivery network with automated battery swapping machines,” Computers & Operations Research, vol. 129, p. 105177, 2021.
8. P. Grippa et al., “Drone delivery systems: Job assignment and dimensioning,” Autonomous Robots, pp. 261– 274, 2019

   G. Brunner et al., “The urban last mile problem: Autonomous drone delivery to your balcony,” in ICUAS, 2019, pp. 1005–1012