A Comparative Performance Analysis of Unidirectional and Bidirectional A* Search for Global Flight Pathfinding

Buege Mahara Putra - 13523037
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: buege.putra@gmail.com , 13523037@std.stei.itb.ac.id

***Abstract*— The A* search algorithm is effective for finding optimal paths but can face performance issues in large-scale graphs. This paper investigates the performance of Bidirectional A* search as an optimization over the standard Unidirectional A*. Both algorithms were implemented and tested on a real-world flight dataset, with the objective of finding the path with the minimum total travel time. The heuristic function was calculated using the Haversine great-circle distance to estimate remaining travel time. The results show that bidirectional search offers a significant average speedup of 1.75x and reduces the number of nodes explored by 55.3%, with its advantage growing substantially on medium and long-haul routes. However, this speed increase came at a considerable cost. The bidirectional implementation only found the same optimal-cost path as the unidirectional search in 60% of cases, pointing to a common implementation flaw in its termination condition. The study concludes that while bidirectional A* is a powerful optimization for large-scale pathfinding, its practical application demands a correct implementation to ensure that gains in execution speed do not compromise path optimality.**

Keywords*— A* search; bidirectional search; pathfinding; flight network; graph theory; heuristic; Haversine formula.

I. Introduction

The appearance of complex, interconnected networks has been a defining feature of the 21st century, from social media graphs to global supply chains. Among the most tangible of these is the global air transportation network, a vast web connecting thousands of airports with millions of routes. For passengers, airlines, and logistics companies, navigating this network efficiently is a problem with practical and economic significance. The challenge does not only lie in finding a path between two cities, but in discovering the optimal path according to a specific metric, such as cost, distance, or, as is the focus of this paper, total travel time.

Pathfinding algorithms provide the computational framework for solving such problems. While foundational algorithms can guarantee a path, more advanced techniques are required for achieving optimality in a timely manner, especially as the scale of the network grows. The A* search algorithm stands out as a highly effective method for finding the lowest-cost path by intelligently using a heuristic to guide its search, significantly outperforming "blind" searches in complex graphs.

However, even A* can suffer from performance degradation in large state spaces. Application of bidirectional search can offer promising optimization, where two simultaneous searches, one forward from the start and one backward from the goal, are conducted with the aim of meeting in the middle. This paper hypothesizes that a Bidirectional A* search will offer a substantial performance improvement over the standard Unidirectional A* by reducing the total number of nodes that must be explored.

To validate this hypothesis, this study implements and compares both Unidirectional and Bidirectional A* algorithms. These algorithms are applied to a real-world global flight network dataset, where the objective is to find the path with the minimum total flight time. Performance will be quantitatively measured by analyzing the number of expanded nodes and the total execution time required to find the optimal path. This paper aims to provide an empirical basis for evaluating the practical benefits of bidirectional search in the context of large-scale, real-world pathfinding problems.

II. Theoretical Basis

A. Graph

A graph is a data structure that represents connections between objects. It consists of a set of nodes (or vertices) and a set of edges that connect pairs of nodes [1]. For example, a graph can be used to represent system of bridges and land separated by rivers, with nodes representing lands and edges representing bridges, shown in Fig. 1.

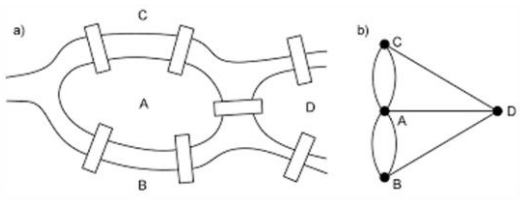

Fig. 1. Representation of Königsberg bridge problem in graph. Source: [1]

A fundamental property of a graph is its directionality. In an undirected graph, the edges are bidirectional, where a connection between node A and node B implies that one can travel from A to B and also from B to A along the same edge. In contrast, a directed graph (or digraph) uses edges that have a specific orientation. An edge from node A to node B does not imply the existence of a corresponding edge from B to A [1].

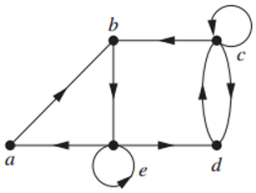

Fig. 2. A directed graph. Source: [1]

The global flight network is inherently a directed graph. For example, the existence of a scheduled flight from Jakarta (CGK) to Tokyo (NRT) is a distinct entity from a flight operating from Tokyo (NRT) to Jakarta (CGK). They are separate routes with different flight numbers, departure times, and potentially even different flight durations due to external factors such as jet streams.

A graph becomes a weighted graph when a numerical value, or weight, is associated with each edge. This weight quantifies a specific attribute of the connection, such as its cost, distance, capacity, or travel time. The choice of weight is critical as it defines the criteria for an "optimal" path. A path that is optimal for one weight (e.g., shortest distance) may be suboptimal for another (e.g., lowest price).

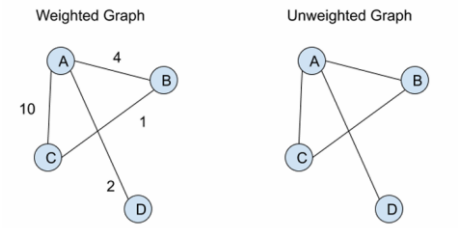

Fig. 3. Weighted vs. unweighted graph. Source: [1]

In the context of this paper, each directed edge (flight route) is assigned two weights:

- Time: The duration of the flight in minutes. This value serves as the cost to be minimized by the pathfinding algorithm. The optimal path will be the one with the lowest cumulative flight time.
- Distance: The physical distance of the flight route in kilometers. This value is not used as a direct cost but is used to calculate the heuristic for the A* search algorithm, providing an estimate of the remaining travel time.

A graph can be represented in various ways in computer memory. The choice of representation method has significant implications for memory usage and the performance of pathfinding operations. Some of the most common representation methods are [2]:

- Adjacency matrix: Each pair of node in the graph is represented as a matrix, with the value as the weight of the edges connecting the nodes in the case of weighted graph, or simply boolean 1/0 in the case of unweighted graph.

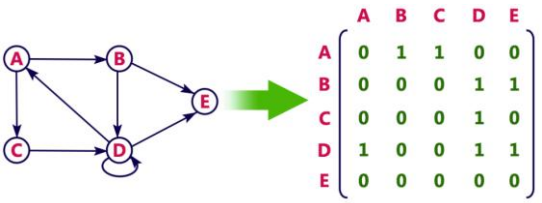

Fig. 4. Adjacency matrix.
Source: http://www.btechsmartclass.com/data_structures/graph-representations.html

- Incidency matrix: Nodes are represented in the rows, and edges are represented in the columns of the matrix. For a directed graph, the (weighted) value is positive if the direction of the edge is “out” from the node, and negative if the direction goes “in” to the node.

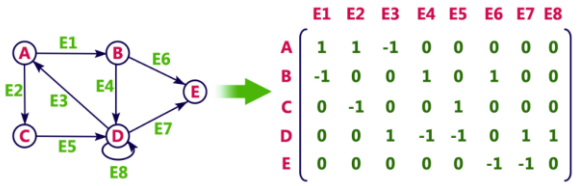

Fig. 5. Incidency matrix.
Source: http://www.btechsmartclass.com/data_structures/graph-representations.html

- Adjacency list: The graph is represented using linked list, with each node having elements containing pointer to nodes connected.

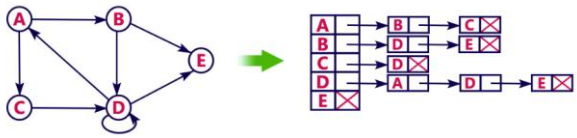

Fig. 6. Adjacency list using linked list.
Source: http://www.btechsmartclass.com/data_structures/graph-representations.html

In the context of this paper, given the sparse nature of of the global flight network, the adjacency list is the most appropriate and efficient representation.

B. *A* Search*

A* (pronounced "A-star") is an informed search algorithm, with the core idea of avoiding expanding paths that are already expensive based on actual costs and selected heuristics. It finds the least-cost path from a start node to a goal node by maintaining a priority queue of paths to explore. The priority of each path is determined by the evaluation function [3]:

$$f(n) = g(n) + h(n) \quad (1)$$

Where:

- n is the last node on the path.
- $g(n)$ is the actual cost of the path from the start node to node n.
- $h(n)$ is the heuristic, an estimated cost of the cheapest path from n to the goal node. This heuristic guides A* to prioritize paths that appear to be leading closer to the goal node.

For A* to guarantee an optimal solution, its heuristic $h(n)$ must be admissible, meaning it never overestimates the actual cost to reach the goal. In terms of complexity, A* has both time and space complexity of $O(b^m)$.

A* search can be performed from both start and goal node simultaneously. This type of search is called bidirectional search. It consists of two searches:

- Forward search: Starts from the start node towards the goal node. This is identical to unidirectional A* search.
- Backward search: Starts from the goal node towards the start node.

The backward search requires a reversed graph, where edge A → B is turned into B → A. Its cost function $g_{bwd}(n)$ calculates the cost from the goal node, and its heuristic $h_{bwd}(n)$ estimates the cost from node n to the start node.

The termination condition occurs when one search finds a node that has already been fully processed by the other, and the best path found through this intersection point is proven to be better than any other potential path in either priority queue. This strategy has the potential to drastically reduce the number of nodes explored, leading to significant performance gains.

C. *Haversine Formula*

When calculating the straight-line distance between two geographical points, one cannot use simple Euclidean distance due to the Earth's curvature. Haversine formula is a mathematical equation used in navigation to calculate the great-circle distance between two points on a sphere from their latitudes and longitudes.

Given the latitude (φ), longitude (λ), and the Earth's mean radius (r, approximated as 6,371 km), the Haversine formula calculates the distance d between point 1 (φ_1, λ_1) and point 2 (φ_2, λ_2) as follows [4]:

$$\text{hav}\,\theta = \text{hav}(\Delta\varphi) + cos\,\varphi_1\, cos\,\varphi_2\, \text{hav}(\Delta\lambda) \quad (2)$$

$$d = r\,\text{archav}(\text{hav}\,\theta) = 2r\,arcsin\sqrt{\text{hav}\,\theta} \quad (3)$$

Where:

- $\text{hav}\,\theta = sin^2\left(\frac{\theta}{2}\right)$
- $\Delta\varphi = \varphi_2 - \varphi_1$
- $\Delta\lambda = \lambda_2 - \lambda_1$
- All angular values in radian.

III. IMPLEMENTATION

The experiment uses Python as its language, chosen for its easy-to-use libraries. The dataset used is collected from https://www.flightsfrom.com, using scraper from https://github.com/Jonty/airline-route-data. Essential data for our use includes airports IATA code, its coordinates (latitude and longitude), and its routes that is directly connected to other airports. The data is saved into a JSON file.

The data is imported and converted into graph in the form of adjacency list, building both forward and backward adjacency to prepare for bidirectional A* search.

```
class AirportGraph:
  def __init__(self, json_file_path: str):
    # Store airport information
    self.airports: Dict[str, Dict] = {}

    # Adjacency list: {airport:
    # [(destination, distance, time), ...]}
    self.graph: Dict[str, List[Tuple[str, int, int]]]
                = {}

    # Reverse adjacency list for bidirectional search
    self.reverse_graph: Dict[str, List[Tuple[str, int,
                        int]]] = {}

    self.__load_from_json(json_file_path)
```

Fig. 7. AirportGraph class.

The cost function $g(n)$ is defined as the accumulation of time taken from the start node to node n.

The heuristic $h(n)$ is defined as the approximate time it takes to travel from node n to the goal node using great-circle distance d and the average speed of an average aircraft as follows:

$$h(n) = \frac{d}{v_{avg}} \quad (4)$$

The v_{avg} used for this purpose is 885 km/h [5]. We take the lower bound to ensure the heuristic is admissible. Another factor contributing to ensuring admissibility is using great-circle distance, which is the shortest path between two points on Earth. This ensures the distance used in heuristic is shorter than the actual distance traveled because an actual aircraft travels at certain altitudes, not at the surface of the Earth.

```
def haversine_distance(self, from_airport: str,
                       to_airport: str) -> float:
  from_info = self.get_airport_info(from_airport)
  to_info = self.get_airport_info(to_airport)

  if not from_info or not to_info:
    return float('inf')

  # Convert latitude and longitude from degrees to
  # radians
  lat1 = math.radians(float(from_info['latitude']))
  lon1 = math.radians(float(from_info['longitude']))
  lat2 = math.radians(float(to_info['latitude']))
  lon2 = math.radians(float(to_info['longitude']))
```

```
  r = 6371 # Radius of earth in kilometers

  # Haversine formula
  dlat = lat2 - lat1
  dlon = lon2 - lon1
  hav_theta = math.sin(dlat/2)**2 + math.cos(lat1) *
              math.cos(lat2) * math.sin(dlon/2)**2
  d = 2 * r * math.asin(math.sqrt(hav_theta))

  return d
```

Fig. 8. Haversine formula implementation.

```
def a_star_search(self, start: str, goal: str, avg_speed:
float = 885) -> Tuple[List[str], float, int]:
  if start not in self.airports or goal not in
    self.airports:
    return [], float('inf'), 0

  if start == goal:
    return [start], 0, 1

  # Priority queue:
  # (f_score, g_score, current_airport, path)
  open_set = [(0, 0, start, [start])]
  closed_set = set()
  g_scores = {start: 0}
  nodes_visited = 0

  while open_set:
    current_f_score, current_g_score, current, path =
      heapq.heappop(open_set)

    if current in closed_set:
      continue

    closed_set.add(current)
    nodes_visited += 1

    if current == goal:
      return path, current_g_score, nodes_visited

    # Explore neighbors
    for neighbor, distance, time in
      self.get_connections(current):
      if neighbor in closed_set:
        continue

      # Calculate actual cost (g_score) using time
      tentative_g_score = current_g_score + time

      if neighbor not in g_scores or tentative_g_score <
         g_scores[neighbor]:
        g_scores[neighbor] = tentative_g_score

        # Calculate heuristic cost (h_score) using
        # haversine distance / avg_speed
        h_distance = self.haversine_distance(neighbor,
                                             goal)
        h_score = (h_distance / avg_speed) * 60

        # Calculate total estimated cost
        # (f_score = g_score + h_score)
        f_score = tentative_g_score + h_score
        new_path = path + [neighbor]
        heapq.heappush(open_set, (f_score,
                  tentative_g_score, neighbor, new_path))

  return [], float('inf'), nodes_visited  # No path found
```

Fig. 9. Unidirectional A* search implementation.

```
def bidirectional_a_star_search(self, start: str, goal:
str, avg_speed: float = 885) -> Tuple[List[str], float,
int]:
  if start not in self.airports or goal not in
    self.airports:
    return [], float('inf'), 0

  if start == goal:
    return [start], 0, 1

  # Forward search data structures
  forward_open = [(0, 0, start, [start])]
  forward_closed = set()
  forward_g_scores = {start: 0}
  forward_parent = {start: None}

  # Backward search data structures
  backward_open = [(0, 0, goal, [goal])]
  backward_closed = set()
  backward_g_scores = {goal: 0}
  backward_parent = {goal: None}

  best_cost = float('inf')
  meeting_node = None
  nodes_visited = 0

  while forward_open and backward_open:
    # Forward search step
    if forward_open:
      current_f_score, current_g_score, current, path =
        heapq.heappop(forward_open)

      if current in forward_closed:
        continue

      forward_closed.add(current)
      nodes_visited += 1

      # Check if current node was reached by backward
      # search
      if current in backward_closed:
        total_cost = current_g_score +
          backward_g_scores.get(current, float('inf'))
        if total_cost < best_cost:
          best_cost = total_cost
          meeting_node = current

      # Explore forward neighbors
      for neighbor, distance, time in
        self.get_connections(current):
        if neighbor in forward_closed:
```

```
          continue

        tentative_g_score = current_g_score + time

        if neighbor not in forward_g_scores or
          tentative_g_score < forward_g_scores[neighbor]:
          forward_g_scores[neighbor] = tentative_g_score
          forward_parent[neighbor] = current

          # Calculate heuristic cost (h_score):
          # haversine distance to goal / avg_speed
          h_distance = self.haversine_distance(neighbor,
                                               goal)
          h_score = (h_distance / avg_speed) * 60

          # Calculate total estimated cost
          # (f_score = g_score + h_score)
          f_score = tentative_g_score + h_score
          new_path = path + [neighbor]
          heapq.heappush(forward_open, (f_score,
            tentative_g_score, neighbor, new_path))

    # Backward search step
    if backward_open:
      current_f_score, current_g_score, current, path =
        heapq.heappop(backward_open)

      if current in backward_closed:
        continue

      backward_closed.add(current)
      nodes_visited += 1

      # Check if current node was reached by forward
      # search
      if current in forward_closed:
        total_cost = forward_g_scores.get(current,
          float('inf')) + current_g_score
        if total_cost < best_cost:
          best_cost = total_cost
          meeting_node = current

      # Explore backward neighbors (using reverse graph)
      for neighbor, distance, time in
        self.get_reverse_connections(current):
        if neighbor in backward_closed:
          continue

        tentative_g_score = current_g_score + time

        if neighbor not in backward_g_scores or
           tentative_g_score <
           backward_g_scores[neighbor]:
          backward_g_scores[neighbor] = tentative_g_score
          backward_parent[neighbor] = current

          # Calculate heuristic cost (h_score): haversine
distance from start / avg_speed
          h_distance = self.haversine_distance(start,
                       neighbor)
          h_score = (h_distance / avg_speed) * 60
          f_score = tentative_g_score + h_score
          new_path = path + [neighbor]
          heapq.heappush(backward_open, (f_score,
            tentative_g_score, neighbor, new_path))

    # Early termination if we found a meeting point
    if meeting_node is not None:
      break

  # Reconstruct path if meeting point found
  if meeting_node is not None:
    # Build forward path from start to meeting node
    forward_path = []
    current = meeting_node
    while current is not None:
      forward_path.append(current)
      current = forward_parent.get(current)
    forward_path.reverse()

    # Build backward path from meeting node to goal
    backward_path = []
    current = backward_parent.get(meeting_node)
    while current is not None:
      backward_path.append(current)
      current = backward_parent.get(current)

    # Combine paths
    final_path = forward_path + backward_path
    return final_path, best_cost, nodes_visited

  return [], float('inf'), nodes_visited
```

Fig. 10. Bidirectional A* search implementation.

IV. Results and Analysis

To analyze the results, a test suite is created. The test suite generates test routes between different airport categories based on expected path length. List of airports are sorted based on their connectivity (i.e. how much airport they connect). The categories are divided into:

- Hub-to-Hub (Short)
- Hub-to-Large (Medium-Short)
- Large-to-Medium (Medium)
- Medium-to-Small (Medium-Long)
- Small-to-Small (Long)
- Cross-Continental (Very Long)

Then 5 pairs of airports from each category are selected randomly. The selected routes are subjected to both unidirectional and bidirectional A* search. Table I. and Fig. 11 shows the result of the test suite.

TABLE I. PER-CATEGORY RESULTS

Route Category	Expected Hops	Avg. Path Length	Nodes Visited (Avg)		Execution Time (ms) (Avg)		Performance (Bidirectional)	
			Uni	Bi	Uni	Bi	Speed up	% Fewer Nodes
Hub-to-Hub (Short)	1-2	2.4	57.3	48.6	4.99	6.75	0.79x	-3.40%
Hub-to-Large (Medium-Short)	2-3	3.2	123.5	29.2	6.62	4.77	1.44x	76.40%
Large-to-Medium (Medium)	2-4	5.0	479.9	82.0	13.5	5.95	2.27x	82.90%
Medium-to-Small (Medium-Long)	3-5	6.8	746.4	219.9	16.76	9.67	1.73x	70.50%
Small-to-Small (Long)	4-6+	6.5	426.4	103.4	9.86	3.84	2.57x	75.70%
Cross-Continental (Very Long)	3-7+	4.3	407.4	89.8	8.7	4.92	1.77x	78.00%

```
OVERALL PERFORMANCE ANALYSIS:
----------------------------------------
Success Rate: 60/60 (100.0%)

- Performance Improvements:
   Average Speedup: 1.75x
   Average Node Reduction: 55.3%

- Bidirectional Efficiency:
   Faster execution: 48/60 (80.0%)
   Fewer nodes visited: 53/60 (88.3%)

DETAILED ALGORITHM COMPARISON:
------------------------------------------------
- Optimal Path Quality: 36/60 (60.0%) found same
   optimal cost

- Performance by Path Length:
   Short (≤2 hops)  : 10 tests, 0.87x speedup,
                      -17.0% fewer nodes
   Medium (3-4 hops): 22 tests, 1.65x speedup,
                      68.2% fewer nodes
   Long (≥5 hops)   : 28 tests, 2.14x speedup,
                      71.0% fewer nodes
```

Fig. 11. Excerpt of test suite result.

The data shows a clear correlation between path length and the effectiveness of bidirectional search.

- Short Paths (≤ 2 hops): For short routes, such as the "Hub-to-Hub" category, bidirectional search performs worse than unidirectional search. It resulted in a 0.79x-0.87x speedup (i.e., it was 13-21% slower) and increased the number of nodes visited by 3-17%. This is due to the inherent overhead of managing two separate search frontiers (two open/closed lists), which outweighs the benefits when the goal is already close.
- Medium Paths (3-4 hops): This is where the benefits of bidirectional search become apparent. For routes like "Hub-to-Large" and "Large-to-Medium," it achieved an average speedup of 1.65x and a massive 68.2% reduction in nodes visited. The "Large-to-Medium" category saw the most dramatic node reduction in the entire test set (82.9%), leading to a 2.27x speedup. This indicates that as search complexity grows, the "meet-in-the-middle" strategy becomes highly effective.
- Long Paths (≥ 5 hops): The performance advantage is most noticeable for long and complex routes. Across these tests, bidirectional search achieved an average speedup of 2.14x while visiting 71.0% fewer nodes. The "Small-to-Small" category, with an average path length of 6.5 hops, registered the highest speedup of 2.57x. This confirms that the bidirectional algorithm scales exceptionally well for deep searches in a large graph.

While a correctly implemented bidirectional A* search should always find the optimal path, the results show that only 36 out of 60 tests (60.0%) found the same optimal cost path as the unidirectional search.

This suggests a potential flaw in the bidirectional algorithm's implementation, likely in its termination condition. A bidirectional search can find a path when its two frontiers first meet, but this initial path is not guaranteed to be the shortest. The algorithm must continue searching until the sum of the costs from the start and to the end for the meeting point is less than or equal to the length of the best path found so far. This presents a critical trade-off, that is, the current implementation often sacrifices path quality for execution speed.

V. CONCLUSION

The experiment in this paper was conducted to validate the hypothesis that a Bidirectional A* search offers a substantial performance improvement over a standard Unidirectional A* search by reducing the total nodes explored in a real-world global flight network. The results of the experiment largely confirm this hypothesis, but with critical caveats regarding path length and solution optimality.

While Bidirectional A* is a powerful optimization that can drastically accelerate pathfinding in large graphs, its practical application requires a robust implementation that correctly handles the termination condition. The experiment highlights that without this, significant gains in execution speed may come at the unacceptable cost of sacrificing path optimality.

APPENDIX

Source code:
https://github.com/buege-putra/AStarComparison

ACKNOWLEDGEMENT

The author of this paper would like to thank the lecturer of this course, Dr. Nur Ulfa Maulidevi for her guidance throughout the course, and to thank the author's family and friends for their support throughout the period of writing this paper.

REFERENCES

[1] R. Munir, "Graf (Bag. 1)," 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

[2] R. Munir, "Graf (Bag. 2)," 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/21-Graf-Bagian2-2024.pdf

[3] N. Ulfa Maulidevi and R. Munir, "Penentuan Rute (Route/Path Planning) Bagian 2: Algoritma A*," 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf

[4] G. Van Brummelen, Heavenly Mathematics: The Forgotten Art of Spherical Trigonometry. Princeton University Press, 2013.

[5] World Aviation, "What's the speed of an airplane? - World Aviation ATO," *World Aviation | Escuela de Pilotos de Referencia*, Sep. 13, 2024. https://worldaviationato.com/en/airplane-speed/

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 24 Juni 2025

Buege Mahara Putra
13523037