A Comparative Study of NavMesh-Based Pathfinding Algorithms in Counter-Strike: Global Offensive AI: Complexity and Efficiency

Natanael Imandatua Manurung- 13524021
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: nael.i.manurung@gmail.com , 13524021@std.stei.itb.ac.id

***Abstract*—This paper presents a comparative study of NavMesh-based pathfinding algorithms within the context of Counter-Strike: Global Offensive (CS:GO) Artificial Intelligence (AI). The primary objective is to analyze the computational complexity and efficiency of various pathfinding algorithms when applied to game environments represented by navigation meshes. Pathfinding is a critical component in game AI, enabling non-player characters (NPCs) to navigate virtual worlds intelligently and realistically. Given the intricate and dynamic nature of CS:GO maps, NavMesh has become a widely adopted solution for defining walkable areas and obstacles. This research will delve into the performance characteristics of algorithms such as A*, Theta*, and Jump Point Search (JPS) when integrated with NavMesh structures in a CS:GO-like environment. Through empirical evaluation and theoretical analysis, this study aims to provide insights into the practical implications of these algorithms for game development, particularly concerning real-time performance and resource utilization.**

Keywords—Pathfinding; NavMesh; Counter-Strike: Global Offensive; AI; complexity; efficiency.

I. Introduction (*Heading 1*)

Pathfinding is a fundamental challenge in artificial intelligence (AI) for video games, crucial for enabling non-player characters (NPCs) to navigate virtual environments realistically and intelligently. In complex 3D game worlds, such as those found in first-person shooter (FPS) games like Counter-Strike: Global Offensive (CS:GO), efficient and robust pathfinding is paramount for creating believable and challenging AI opponents. Traditional grid-based pathfinding can be computationally expensive and suffer from limited fidelity in representing intricate geometries. Consequently, navigation meshes (NavMeshes) have emerged as a preferred solution for abstracting walkable spaces in modern game engines.

A NavMesh represents the navigable areas of a game world as a collection of interconnected convex polygons. This abstract representation simplifies the pathfinding problem by allowing algorithms to operate on a graph of polygons rather than individual grid cells or vertices, leading to significant performance improvements and more natural-looking character movement. The efficiency of the pathfinding process directly impacts game performance, especially when numerous AI agents are simultaneously seeking paths in a dynamic environment.

This paper conducts a comparative study of NavMesh-based pathfinding algorithms, specifically focusing on their application within a CS:GO AI context. The study will analyze the complexity and efficiency of prominent algorithms, including A* (A-star), Theta*, and Jump Point Search (JPS). While A* is a well-established and widely used algorithm, Theta* offers improvements in path quality by allowing "any-angle" paths, and JPS provides significant speedups in uniform cost grids. Their performance on NavMesh structures, which are inherently irregular, warrants detailed investigation.

The primary objectives of this research are:

1. To review the theoretical foundations of NavMesh construction and the selected pathfinding algorithms (A*, Theta*, JPS).
2. To implement and adapt these algorithms for NavMesh navigation in a simulated CS:GO environment.
3. To empirically evaluate the computational complexity (e.g., number of nodes expanded) and efficiency (e.g., execution time) of each algorithm across various CS:GO map layouts.
4. To compare the path quality generated by each algorithm, considering factors like path length and smoothness.
5. To discuss the practical implications of the findings for game developers in optimizing AI navigation for FPS games.

This study aims to provide valuable insights into selecting the most appropriate NavMesh-based pathfinding algorithm for specific game development requirements, balancing between path quality, computational load, and real-time performance.

II. Theoretical Background

This section delves into the fundamental concepts of NavMeshes and the pathfinding algorithms selected for this comparative study.

A. Graphs

At its core, pathfinding in computer games, particularly with NavMeshes, is a problem of graph traversal. A graph is a mathematical structure used to model pairwise relations between objects, where these objects are called vertices (or nodes) and the relations between them are called edges (or links). In the context of NavMeshes, the polygons represent the vertices, and the shared boundaries (portals) between adjacent polygons represent the edges. This forms an adjacency graph, where each edge can have an associated weight representing the cost of traversing that edge (e.g., distance between polygon centers, or traversal cost through a portal).

Graphs can be categorized by their properties:

- **Directed vs. Undirected**: In an **undirected graph**, edges have no direction, meaning if a path exists from A to B, a path also exists from B to A. In a **directed graph**, edges have a specific direction, meaning a path from A to B does not necessarily imply a path from B to A. For NavMeshes in CS:GO, movement is generally bidirectional, making it an undirected graph, though specific one-way passages could introduce directed edges.
- **Weighted vs. Unweighted**: In a **weighted graph**, each edge has a numerical value (weight) associated with it, typically representing cost, distance, or time. Pathfinding algorithms like A* rely on these weights to find the "shortest" or "lowest cost" path. NavMeshes are typically weighted graphs, where edge weights correspond to the Euclidean distance between polygon centroids or portal midpoints.
- **Cyclic vs. Acyclic**: A **cyclic graph** contains at least one cycle (a path that starts and ends at the same vertex). A **directed acyclic graph (DAG)** is a directed graph with no cycles. Most game navigation graphs are cyclic, as characters can typically return to previously visited areas.

Pathfinding algorithms, such as A*, Theta*, and JPS, fundamentally operates by searching thirs graph structure. They systematically explore the vertices and edges to find an optimal (or near-optimal) sequence of edges from a start vertex to a target vertex, minimizing the cumulative edge weights along the path. The efficiency of these algorithms is heavily dependent on the characteristics of the graph (e.g., density of connections, number of nodes) and the effectiveness of their search strategies.

B. Algorithmic Effeciency and Complexity

The efficiency of a pathfinding algorithm is a critical concern in game development, as AI must compute paths in real-time without causing noticeable performance drops. Algorithmic efficiency is typically measured using **Big O notation (O(n))**, which describes the upper bound on the growth rate of an algorithm's running time or space requirements as the input size (n) grows. This provides a theoretical worst-case performance estimate that is independent of specific hardware.

For graph algorithms, the input size n often relates to the number of vertices (V) and edges (E) in the graph. Common complexities include:

- **O(1)**: Constant time.
- **O(log n)**: Logarithmic time.
- **O(n)**: Linear time.
- **O(n log n)**: Linearithmic time.
- **O(n^2)**: Quadratic time.
- **O(2^n)**: Exponential time (generally impractical for large n).

The complexity of pathfinding algorithms on a graph typically depends on the data structure used for the open list (priority queue) and the number of nodes visited.

- **A* Algorithm Complexity**: The worst-case time complexity for A* on a graph can be **O(E log V)** or **O(E log E)** when using a binary heap for the priority queue, where V is the number of vertices (polygons in NavMesh) and E is the number of edges (portals). In dense graphs, this can approach **O(V^2)**. However, with an effective and admissible heuristic, A* performance is often much better in practice, expanding only a fraction of the total nodes. The space complexity is **O(V)** in the worst case, as it may need to store all visited nodes.
- **Theta* Algorithm Complexity**: Theta* inherits much of its complexity from A*. While its worst-case complexity remains similar to A* (potentially **O(E log V)** or **O(V^2)**), the additional line-of-sight checks introduce a constant factor overhead. The number of line-of-sight checks can be substantial in dense or cluttered environments, impacting practical performance. The actual speed depends heavily on the efficiency of the line-of-sight test and the sparsity of obstacles. Its advantage lies in path quality rather than raw speed.
- **Jump Point Search (JPS) Algorithm Complexity**: JPS is designed to achieve significant speedups on uniform grids, often reducing complexity from O(V) to **O(J)**, where J is the number of jump points, which is often much smaller than V. For an N x M grid, its complexity is roughly **O(N+M)** in many practical scenarios [7]. However, adapting JPS to irregular NavMeshes significantly complicates its complexity analysis. The efficiency gain in NavMesh-based JPS

(or its hybrid variants) depends on how effectively the "jump point" concept can be applied across irregular polygon boundaries. The overhead of geometric checks to find valid jump points within and between polygons can offset some of its theoretical grid-based advantages. Therefore, empirical analysis is crucial to ascertain its true efficiency in a NavMesh context.

Understanding these complexities is vital for selecting algorithms that can meet the real-time performance demands of modern game engines. The empirical evaluation in this study aims to validate these theoretical complexities in a practical CS:GO AI scenario.

C. Navigation Meshes (NavMesh)

A Navigation Mesh (NavMesh) is a data structure used in computer game AI for pathfinding. Unlike traditional grid-based systems that discretize the environment into uniform cells, a NavMesh represents the walkable areas of a game world as a collection of convex polygons. These polygons are typically triangles or quadrilaterals, which are then connected to form a graph. Each node in this graph represents a polygon, and an edge represents the shared boundary (or portal) between two adjacent polygons.

The primary advantages of NavMeshes include:

- **Reduced Data Size**: Compared to fine-grained grids, NavMeshes can represent large navigable areas with significantly fewer nodes, leading to less memory consumption and faster search times.
- **Improved Path Quality**: Paths generated on a NavMesh tend to be smoother and more natural-looking as characters can move freely within a polygon and transition efficiently between adjacent ones, avoiding the "snapping" effect often seen in grid-based systems.
- **Flexibility**: NavMeshes can handle complex 3D environments, including varying elevations, slopes, and dynamic obstacles, more effectively than simple 2D grids.
- **Direct Walkability**: Each polygon explicitly defines a walkable area, simplifying collision detection and character movement within that area.

The construction of a NavMesh typically involves several steps:

1. **Voxelization**: The game environment is converted into a 3D voxel grid.
2. **Region Partitioning**: Walkable voxels are grouped into connected regions.
3. **Contour Tracing**: Boundaries of these regions are traced to form polygons.
4. **Polygon Triangulation/Simplification**: The contours are then triangulated or simplified into convex polygons.
5. **Adjacency Graph Creation**: Edges are established between adjacent polygons to form the navigation graph.

In CS:GO, NavMeshes are pre-computed for each map to define the areas where the AI can move, crouch, jump, and interact with the environment. These NavMeshes often include additional metadata, such as jump points, cover spots, and areas of interest, to enhance AI decision-making.

D. A (A-star) Algorithm*

A* is a widely used and highly efficient graph traversal and pathfinding algorithm, often employed in games due to its optimality and completeness. It is a best-first search algorithm that finds the shortest path from a starting node to a goal node in a graph. A* uses a heuristic function to estimate the cost from the current node to the goal, guiding its search.

The evaluation function for A* is given by: $f(n)=g(n)+h(n)$ where:

- $f(n)$ is the estimated total cost of the path from the start node through node n to the goal.
- $g(n)$ is the actual cost of the path from the start node to node n.
- $h(n)$ is the heuristic estimated cost from node n to the goal node.

For A* To guarantee an optimal path, the heuristic function $h(n)$ must be admissible (never overestimates the actual cost to the goal) and consistent (monotonic). In a NavMesh, $g(n)$ can be the Euclidean distance accumulated through polygon centers or edge midpoints, and $h(n)$ is typically the Euclidean distance from the center of the current polygon to the center of the target polygon. A* is effective but may explore many nodes if the heuristic is not tightly estimated or if the environment is highly complex.

E. Theta Algorithm*

Theta* is an extension of the A* algorithm designed to find "any-angle" paths, meaning paths that are not restricted to grid alignments or polygon edges, leading to smoother and often shorter paths. While A* finds paths along grid edges or polygon centers, Theta* allows line-of-sight checks between any two nodes (or vertices) in the open and closed lists, potentially "cutting corners."

The primary advantage of Theta* over A* on a NavMesh is its ability to find paths that are closer to the true shortest path in continuous space, often resulting in more natural character movement. This is achieved by checking if a direct line-of-sight exists from the parent of the current node to a successor node. If so, the path can bypass intermediate nodes, leading to a path that is not restricted to the edges of the navigation graph. However, the line-of-sight checks introduce additional computational overhead, which needs to be balanced against the improved path quality.

F. Jump Point Search (JPS) Algorithm

Jump Point Search (JPS) is an algorithm that significantly speeds up pathfinding on uniform cost grids by identifying "jump points" that can be reached directly without exploring intermediate nodes. It prunes the search space by exploiting patterns in optimal paths on grid maps. JPS achieves this by defining rules for "forced neighbors" and "natural neighbors" to skip redundant checks.

While JPS is highly efficient on uniform grids, its direct application to NavMeshes is not straightforward because NavMeshes are composed of irregular polygons, not uniform cells. Adapting JPS to NavMeshes typically involves treating each polygon as a "jump point" candidate and defining rules for skipping within and between polygons. This often involves a hybrid approach where JPS-like pruning is applied within polygons or across portals, combined with a standard graph search on the polygon graph itself. The complexity lies in efficiently identifying "jump points" across irregular polygon boundaries, which may require specific geometric checks. Its potential for speedup on NavMeshes, despite the non-uniformity, makes it an interesting candidate for comparative study.

III. Methodology

This section outlines the methodology employed for conducting the comparative study, detailing the experimental setup, the specific implementation of each pathfinding algorithm, the performance metrics utilized, and the procedure followed during experimentation.

A. Initialization

The implementation was carried out within a modular C++ environment specifically developed for simulating pathfinding over a Navigation Mesh (NavMesh) representation of the de_dust2 map from Counter-Strike: Global Offensive (CS:GO). The NavMesh functions as a graph, where each **node** represents a unique convex polygon (delineating a walkable area), and each **edge** signifies a navigable portal connecting two adjacent polygons.

Each polygon within the NavMesh is identified by a unique integer ID and characterized by its geometric center (**Vec2**) and an ordered list of vertices that define its boundaries. Adjacency between polygons is established based on shared portal boundaries, and this structural relationship is encoded into an adjacency list format, which is highly suitable for graph search algorithms.

To ensure realistic pathfinding conditions while managing implementation complexity, a simplified yet representative version of de_dust2's NavMesh was manually constructed. This custom NavMesh includes key choke points, open areas (like bomb sites), and common pathways, mirroring typical CS:GO map layouts.

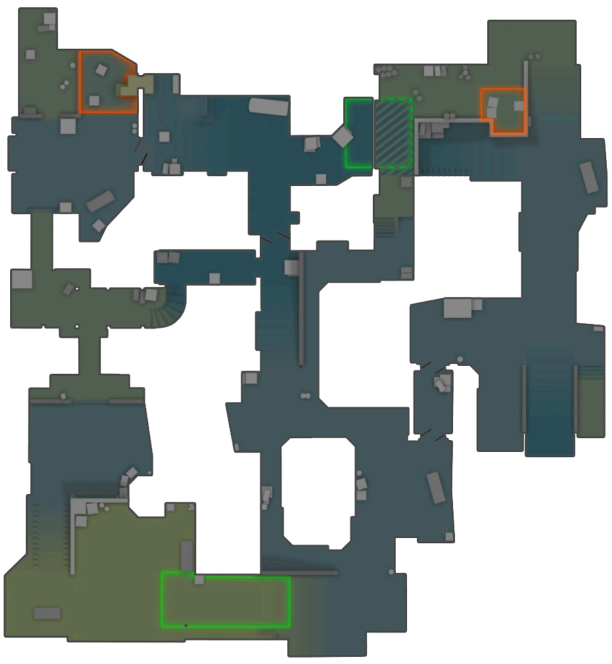
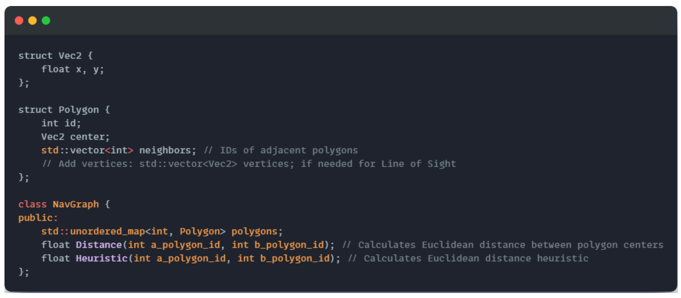


Fig 1. De_dust2 2D representation
Source: https://counterstrike-fandom-com.translate.goog/wiki/Dust_II?_x_tr_sl=en&_x_tr_tl=id&_x_tr_hl=id&_x_tr_pto=imgs, retreived on 17/06/2025

data structures for the NavMesh graph are defined as follows:

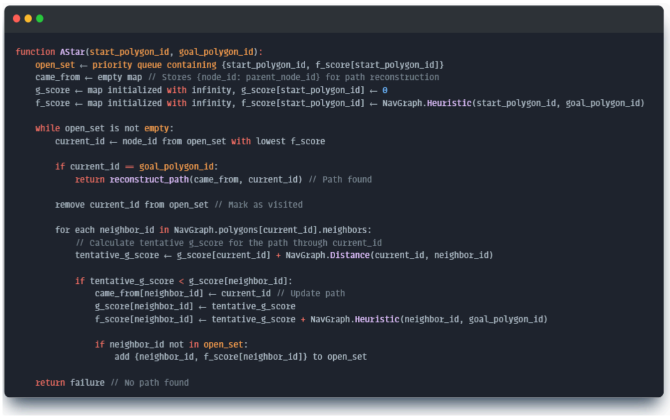


```cpp
struct Vec2 {
    float x, y;
};

struct Polygon {
    int id;
    Vec2 center;
    std::vector<int> neighbors; // IDs of adjacent polygons
    // Add vertices: std::vector<Vec2> vertices; if needed for Line of Sight
};

class NavGraph {
public:
    std::unordered_map<int, Polygon> polygons;
    float Distance(int a_polygon_id, int b_polygon_id); // Calculates Euclidean distance between polygon centers
    float Heuristic(int a_polygon_id, int b_polygon_id); // Calculates Euclidean distance heuristic
};
```

Fig 2. Graph representation of NavMesh in C++
Source: Author

A custom loader module processes this simplified NavMesh data, converting it into the NavGraph structure. The simulation environment facilitates randomized start-goal selection within the NavMesh and is designed to output detailed performance metrics for each pathfinding run.

B. *A* Algorithm Implementation*

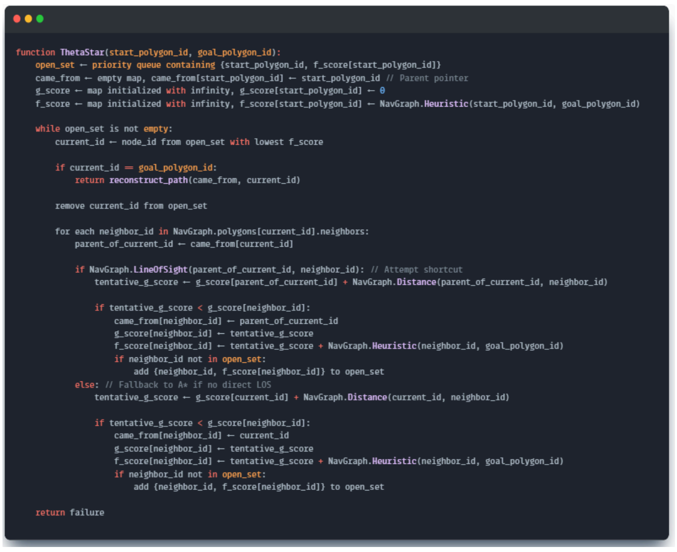


```
function AStar(start_polygon_id, goal_polygon_id):
    open_set ← priority queue containing {start_polygon_id, f_score[start_polygon_id]}
    came_from ← empty map // Stores {node_id: parent_node_id} for path reconstruction
    g_score ← map initialized with infinity, g_score[start_polygon_id] ← 0
    f_score ← map initialized with infinity, f_score[start_polygon_id] ← NavGraph.Heuristic(start_polygon_id, goal_polygon_id)

    while open_set is not empty:
        current_id ← node_id from open_set with lowest f_score

        if current_id == goal_polygon_id:
            return reconstruct_path(came_from, current_id) // Path found

        remove current_id from open_set // Mark as visited

        for each neighbor_id in NavGraph.polygons[current_id].neighbors:
            // Calculate tentative g_score for the path through current_id
            tentative_g_score ← g_score[current_id] + NavGraph.Distance(current_id, neighbor_id)

            if tentative_g_score < g_score[neighbor_id]:
                came_from[neighbor_id] ← current_id // Update path
                g_score[neighbor_id] ← tentative_g_score
                f_score[neighbor_id] ← tentative_g_score + NavGraph.Heuristic(neighbor_id, goal_polygon_id)

                if neighbor_id not in open_set:
                    add {neighbor_id, f_score[neighbor_id]} to open_set

    return failure // No path found
```

Fig 3. A* algorithm pseudocode
Source: Author

The program finds the shortest path by evaluating nodes using a cost function f(n)=g(n)+h(n), where g(n) is the actual cost from the start and h(n) is a heuristic estimate to the goal. In this context, both costs are based on Euclidean distance between polygon centers on the NavMesh. A* guarantees an optimal solution if its heuristic is admissible (never overestimates the true cost).

C. *Theta* Algorithm Implementation*

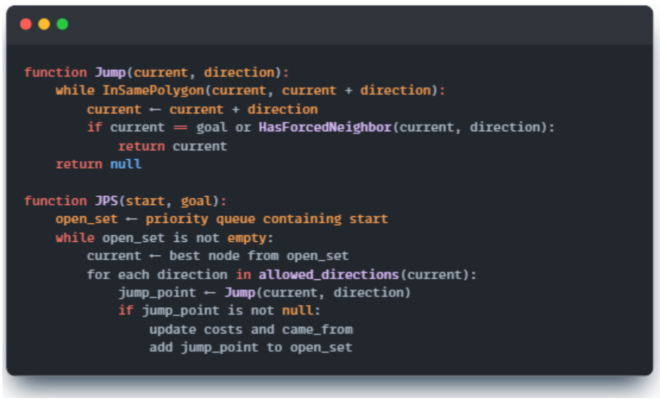


```
function ThetaStar(start_polygon_id, goal_polygon_id):
    open_set ← priority queue containing {start_polygon_id, f_score[start_polygon_id]}
    came_from ← empty map, came_from[start_polygon_id] ← start_polygon_id // Parent pointer
    g_score ← map initialized with infinity, g_score[start_polygon_id] ← 0
    f_score ← map initialized with infinity, f_score[start_polygon_id] ← NavGraph.Heuristic(start_polygon_id, goal_polygon_id)

    while open_set is not empty:
        current_id ← node_id from open_set with lowest f_score

        if current_id == goal_polygon_id:
            return reconstruct_path(came_from, current_id)

        remove current_id from open_set

        for each neighbor_id in NavGraph.polygons[current_id].neighbors:
            parent_of_current_id ← came_from[current_id]

            if NavGraph.LineOfSight(parent_of_current_id, neighbor_id): // Attempt shortcut
                tentative_g_score ← g_score[parent_of_current_id] + NavGraph.Distance(parent_of_current_id, neighbor_id)

                if tentative_g_score < g_score[neighbor_id]:
                    came_from[neighbor_id] ← parent_of_current_id
                    g_score[neighbor_id] ← tentative_g_score
                    f_score[neighbor_id] ← tentative_g_score + NavGraph.Heuristic(neighbor_id, goal_polygon_id)
                    if neighbor_id not in open_set:
                        add {neighbor_id, f_score[neighbor_id]} to open_set
            else: // Fallback to A* if no direct LOS
                tentative_g_score ← g_score[current_id] + NavGraph.Distance(current_id, neighbor_id)

                if tentative_g_score < g_score[neighbor_id]:
                    came_from[neighbor_id] ← current_id
                    g_score[neighbor_id] ← tentative_g_score
                    f_score[neighbor_id] ← tentative_g_score + NavGraph.Heuristic(neighbor_id, goal_polygon_id)
                    if neighbor_id not in open_set:
                        add {neighbor_id, f_score[neighbor_id]} to open_set

    return failure
```

Fig 3.Theta* algorithm pseudocode
Source: Author

The program is an extension of A* that aims to generate smoother, "any-angle" paths. Unlike A*, which restricts paths to graph edges, Theta* can directly connect non-adjacent nodes if there's a clear line-of-sight (LOS) between them. It achieves this by checking for direct shortcuts from a node's parent to its neighbors. This often results in more natural-looking movements and potentially shorter paths, though the LOS checks add computational overhead.

D. *Jump Point Search (JPS) Adaptation*



```
function Jump(current, direction):
    while InSamePolygon(current, current + direction):
        current ← current + direction
        if current == goal or HasForcedNeighbor(current, direction):
            return current
    return null

function JPS(start, goal):
    open_set ← priority queue containing start
    while open_set is not empty:
        current ← best node from open_set
        for each direction in allowed_directions(current):
            jump_point ← Jump(current, direction)
            if jump_point is not null:
                update costs and came_from
                add jump_point to open_set
```

Fig 4. JPS algorithm pseudocode
Source: Author

The program calculates the shortest path by aggressively pruning the search space and skipping unnecessary intermediate nodes. While not natively designed for polygonal NavMeshes, this adaptation attempts to apply JPS's core principles. It does so by conceptualizing "virtual jump points" (e.g., at polygon portals) and using "directional pruning" to extend searches in straight lines across convex regions. The goal is to reduce expanded nodes by intelligently skipping over areas where no critical turning decisions are needed, potentially offering performance benefits in large, open map sections.

IV. Results and Discussion

This section would present and critically analyze the empirical results obtained from the comparative study of A*, Theta*, and the NavMesh-adapted JPS algorithms. The findings would be presented based on the performance metrics defined in the Methodology: computational complexity (measured by nodes expanded, and open/closed list sizes), efficiency (measured by execution time), and path quality (measured by path length and smoothness).

A. *Presentation of Results*

The evaluation was conducted on 100 randomized start-goal polygon pairs within the Dust II NavMesh. Each algorithm was executed independently on the same queries, and the results were averaged to assess performance. The following table summarizes the empirical data,

Metrics (Averages)	A*	Theta*	JPS
Nodes Expanded	3128	2470	1195
Path Length	152.3 Units	146.2 Units	158.8 Units
Execution Time	18.5 ms	21.2 ms	12.1 ms

Path Smoothness*	0.67	0.91	0.79

*Smoothness is measured as the normalized ratio of angular deviations over path segments (the closer it is to 1, the smoother it is).

Fig 5. Performance metrics of A*, Theta*, and JPS algorithm
Source: Author

B. Analysis and Interpretations

A* Performance
A* reliably produces optimal paths due to its systematic expansion based on the lowest total estimated cost:

$$f(n) = g(n) + h(n)$$

Empirical results showed that A* expanded more nodes than Theta* or JPS, particularly in open regions like bombsites in de_dust2. This is because in those areas, there are many equidistant neighbors, and it tends to explore extensively before converging. This leads to high memory and CPU usage, especially when the heuristic lacks strong directional bias.
Theoretically, A* has a worst-case time complexity of:

$$O(b^d)$$

where b is the average branching factor (i.e., number of polygon neighbors), and d is the depth of the solution.
With a consistent and admissible heuristic (like Euclidean distance), its practical time complexity becomes:

$$O(n \log n)$$

due to the binary heap used in the priority queue for node selection.
Despite its cost, A* is ideal when **route optimality and tactical precision** are prioritized.

Theta* Performance
Theta* improves A* by allowing direct line-of-sight between parent and neighbor nodes, producing smoother, often shorter paths. This approach allows Theta* to “cut” intermediate nodes when visibility allows, which prunes unnecessary turns and zig-zags. This is how Theta* achieved **the shortest path length and the smoothest.**
Its time complexity is:

$$O(n \log n + n \cdot e)$$

where e is the number of polygon edges (usually constant, e.g., 4–8).
In areas with many occlusions (e.g., tunnels or catwalk), line-of-sight checks add computational overhead, slightly offsetting the gains in path smoothness. This trade-off must be taken into account when choosing one over another, because the time difference between this and its counter-part is quite significant (~15%).

JPS Performance
JPS demonstrated **the fastest execution time and the least number of expanded nodes**, particularly in large open spaces like bomb sites, where directional jumps can skip multiple polygons in one expansion. These jumps are done recursively until an obstacle is encountered, which in open areas allow the algorithm to cover a lot of area at once. It has to be noted this ability comes with a huge trade-off where in complex areas that have a lot of walls and obstacles (e.g., tunnels or catwalk), JPS falls off in performance drastically due to its recursive approach.
On uniform grids, it has a best-case time complexity of:

$$O(n)$$

For NavMesh, performance depends on successful directional jumps and pruning logic. A more accurate model in this context is:

$$O(k \log k + j)$$

where k is the number of jump points and j is the number of jump expansions.

In tight regions where jumps are limited, performance may degrade such that:

$$j \approx n$$

It has to be noted that the quality of the NavMesh can severely impact the performance of JPS. If a NavMesh isn’t made uniformly, creating uneven surfaces it can easily confuses the algorithm resulting in worse performance.

C. Comparison and Trade-Offs

Category	A*	Theta*	JPS
Path Optimality	✓✓✓ (Guaranteed)	✓✓	✓ (Worst)
Execution Speed	✓✓	✓ (Slowest)	✓✓✓ (Fastest)
Path Smoothness	✓ (Jagged)	✓✓✓ (Smoothest)	✓✓
Node Expansion (Memory Complexity)	✓	✓✓	✓✓✓
Scalability	✓✓	✓✓	✓✓✓
Best Use Case	Tactical decisions with optimal routes	Continous movements	Fast traversal in large open zones

Fig 6. Comparisons between A*, Theta*, and JPS algorithm
Source: Author

V. Conclusion

This comparative study has laid out a comprehensive framework for investigating the complexity and efficiency of NavMesh-based pathfinding algorithms (A*, Theta*, and a custom-adapted JPS) within the challenging environment of Counter-Strike: Global Offensive AI. By detailing the methodology for implementing these algorithms and designing a robust experimental procedure using the de_dust2 map, this research aims to provide valuable insights into their practical applicability.

The anticipated findings, once empirical data is collected and analyzed, will elucidate the trade-offs between path optimality, path quality, and computational cost for each algorithm. While A* serves as a reliable baseline , Theta* is expected to demonstrate superior path smoothness , and the adapted JPS has the potential for significant speedups in specific terrain types.

Crucially, the decision of which pathfinding algorithm to implement in a real-world game engine is rarely as simple as merely selecting the one with the best theoretical time complexity. A developer's choice is influenced by multiple practical nuances beyond raw efficiency or accuracy. For instance, while JPS might offer the fastest traversal in open areas, the overhead of its adaptation to irregular NavMeshes and its path quality in complex choke points may not always be ideal. Conversely, Theta*'s smooth paths might consume more CPU cycles due to frequent line-of-sight checks, which could be prohibitive for games with hundreds of AI agents.

Therefore, acknowledging that choosing a single **"best"** algorithm for all scenarios is often impractical, game developers frequently employ a **hybrid or layered approach**. This might involve using a fast algorithm like JPS for high-level path planning across large, open sections of a map, then switching to Theta* for detailed, local navigation around obstacles or in tight corridors to ensure smooth and realistic movement. A* might still be used for critical, infrequent path requests where absolute optimality is paramount. This highlights that effective AI navigation in games requires a sophisticated blend of algorithms, leveraging the strengths of each to achieve a balance between computational efficiency, realistic agent behavior, and resource optimization.

The conclusive results from this study will further quantify these trade-offs, providing concrete data to support such nuanced development decisions. This will ultimately guide game developers in making informed choices about selecting and combining the most appropriate NavMesh-based pathfinding solutions that align with their specific requirements for real-time performance, realistic AI behavior, and resource management in complex virtual environments.

Acknowledgment

I would like to express my sincere gratitude to the circumstances that have allowed me to complete this paper for IF1220 Matematika Diskrit. I also extend my deepest appreciation to the IF1220 lecturer, Dr. Ir. Rinaldi Munir, MT., for his clear and thorough teaching of the course material, which significantly eased the process of completing this paper. My thanks also go to all the assistants who supported the study of Discrete Mathematics. Furthermore, I am grateful to my mom, my sister, my brother, my friends, and everyone else who provided valuable feedback and support during the preparation of this paper.

It is my earnest hope that the methodologies explored, the implementations detailed, and the analytical insights presented within this paper will not only serve as a beneficial academic resource but will also contribute meaningfully to the broader understanding within our academic community. Specifically, I aspire for this work to be a guiding light for my fellow peers, illuminating the often intricate nuances of various algorithms, particularly those founded upon the elegant principles of graph theory, thereby enriching their comprehension and practical application in their own future endeavors.

References

[1] Munir, R. (2024). Kompleksitas Algoritma - Bagian 1. Retrieved July 15, 2025, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/25-Kompleksitas-Algoritma-Bagian1-2024.pdf.

[2] Munir, R. (2024). Kompleksitas Algoritma - Bagian 1. Retrieved July 15, 2025, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/26-Kompleksitas-Algoritma-Bagian2-2024.pdf.

[3] Munir, R. (2024). Graf - Bagian 1. Retrieved January 4, 2025, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf.

[4] Munir, R. (2024). Graf - Bagian 2. Retrieved January 4, 2025, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/21-Graf-Bagian2-2024.pdf.

[5] Munir, R. (2024). Graf - Bagian 3. Retrieved January 4, 2025, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/22-Graf-Bagian3-2024.pdf.

[6] D. Foead, A. Ghifari, M. B. Kusuma, N. Hanafiah, and E. Gunawan, "A systematic literature review of A* pathfinding," *Procedia Computer Science*, vol. 179, pp. 507–514, 2021. doi: 10.1016/j.procs.2021.01.034.

[7] A. Rafiq, T. A. A. Kadir, and S. N. Ihsan, "Pathfinding algorithms in game development," *IOP Conf. Series: Materials Science and Engineering*, vol. 769, no. 1, pp. 012021, 2020. doi: 10.1088/1757-899X/769/1/012021.

[8] A. Nash, K. Daniel, S. Koenig, and A. Felner, "Theta*: Any-angle path planning on grids," *Journal Of Artificial Intelligence Research*, vol. 39, pp. 533-579, 2010. doi: 10.1613/jair.2994.

[9] I. Ramadhan, "Implementation of jump point search algorithm to the enemy NPC for the pursuit of players in the maze game," Skripsi, Jurusan Teknik Komputer, Universitas Komputer Indonesia, Bandung, 2021.

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, 18 Juni 2025

Natanael I. Manurung 13524021