A* Pathfinding in Stronghold Navigation: Heuristic-Based Room Search in Minecraft Speedrunning

Jingglang Galih Rinenggan - 13524095
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung 40132, Indonesia
jingglang@gmail.com, 13524095@std.stei.itb.ac.id

***Abstract*—Navigating the stronghold is a critical phase in Minecraft speedrunning, where players must find the Portal Room quickly in a procedurally generated structure. This paper applies the A* search algorithm to stronghold navigation by modeling the structure as a directed graph where rooms are nodes and exits are edges. We design a heuristic function that uses generation depth and branch count to guide search toward the Portal Room. The heuristic is admissible, guaranteeing optimal paths. We compare A* against brute-force breadth-first search on 100 generated strongholds. A* expands 7% fewer nodes on average (20.7 vs 22.1) while finding identical optimal paths (8.4 rooms). Results show that heuristic guidance based on stronghold generation constraints improves search efficiency over uninformed exploration.**

***Index Terms*—A* search, pathfinding, Minecraft, speedrunning, stronghold navigation, heuristic search**

I. Introduction

Minecraft, a sandbox video game developed by Mojang Studios, has sold over 300 million copies worldwide [1]. Among its gameplay modes, speedrunning (completing the game as fast as possible) has grown into a competitive activity with an active community. A key phase in Minecraft speedrunning is navigating the stronghold, a procedurally generated underground structure that holds the End Portal. The stronghold is a network of connected rooms, and finding the Portal Room quickly matters for competitive times. This paper applies A* search to stronghold navigation, modeling the structure as a weighted graph with heuristic functions to guide pathfinding. We formalize room traversal as a search problem, define cost and heuristic functions based on stronghold generation mechanics, and compare the algorithm against brute-force search.

A. Background on Minecraft Speedrunning

Minecraft speedrunning is a competitive practice where players try to complete the game (defeating the Ender Dragon) as fast as possible. The "Any% Glitchless" category, the most popular format, requires specific milestones: gathering resources, locating a stronghold, activating the End Portal, and defeating the dragon [2]. As of 2025, the world record for this category is 6 minutes 50 seconds, showing the level of optimization top players achieve through route planning, mechanical skill, and decision-making.

Speedrunning Minecraft means mastering multiple game systems: crafting, combat, and navigation. Players need to gather items like iron for tools, ender pearls for locating the stronghold, and blaze powder for crafting eyes of ender. The stronghold phase matters because it comes near the end of the run, right before the final boss. Time lost in the stronghold adds directly to the completion time.

Speedrunning, as a broader gaming practice, means completing a video game or specific objectives as fast as possible [3]. Speedruns follow rulesets and completion criteria, with communities maintaining leaderboards and timing methods. In Minecraft, speedrunning requires precise execution of optimal routes, where players use game mechanics, knowledge of procedural generation, and quick decisions to finish faster.

The stronghold is important in Minecraft speedruns because it is the final major navigation challenge before the End Portal. Unlike Overworld structures that can be located with coordinates or external tools, stronghold navigation happens under time pressure with limited information. Players must move through a maze of connected rooms, deciding at each junction which path to take. The Portal Room (the goal) is hidden in this structure, and its location varies across stronghold instances because of procedural generation. Navigating the stronghold efficiently saves seconds, which matters in competitive speedrunning.

Fig. 1. Minecraft gameplay. Source: minecraft.net.

B. The Stronghold Navigation Problem

Fig. 2. The Portal Room containing the End Portal. Source: minecraft.fandom.com.

Navigating the stronghold efficiently is a computational challenge. Unlike static game environments, Minecraft's strongholds are procedurally generated, so each instance has a different layout of rooms and corridors. Speedrunners explore this unknown structure under time pressure, making decisions at each junction without knowing the overall layout.

Manual navigation relies on intuition and experience. Players follow documented strategies such as always choosing the deepest visible path, entering rooms with multiple exits, and avoiding Library dead ends [6]. Tools like LazyStronghold [7] precompute stronghold locations to reduce search time. However, these approaches do not optimize room-by-room traversal within the stronghold itself. The branching structure means following one path may lead to dead ends or redundant exploration. Each wrong turn adds seconds to the run.

The Portal Room is the goal state in this problem. It is the only room with End Portal Frame blocks, and exactly one PortalRoom appears per stronghold at a depth greater than 5 rooms from the start. The challenge is finding this room efficiently among the network of connected rooms, each with different numbers of exits and branching possibilities.

Modeling the stronghold as a graph and applying search algorithms like A* provides a systematic approach. We can define traversal strategies that minimize the expected number of rooms visited before reaching the PortalRoom. This turns an intuition-based task into a computational problem with provable optimality.

C. Research Questions

This paper addresses two questions about A* pathfinding in Minecraft stronghold navigation.

Can A optimize stronghold pathfinding?* We test whether A*, applied to the stronghold graph, produces provably optimal or near-optimal paths. This requires heuristic functions that use stronghold generation properties (depth constraints, room type distributions) to guide search.

How does A compare to brute-force search?* We compare A* against exhaustive exploration that visits every possible path. This comparison measures efficiency gains from heuristic guidance: reductions in rooms visited, path length, and computation.

Previous work by Daniel Pedrosa Wu [5] applied A* to general Minecraft speedrun route planning across the Overworld and Nether. That work focuses on macro-level navigation between biomes and structures. This work targets the stronghold specifically: we model individual rooms as graph nodes and design heuristics based on room generation constraints (depth, room type, exit probability), rather than geographic coordinates.

II. Background

A. A* Algorithm Overview

A* is a graph search algorithm that finds the shortest path from a start node to a goal node. It combines the strengths of Dijkstra's algorithm (guaranteed optimality) with greedy best-first search (speed through heuristic guidance) [4].

A* maintains an open set of nodes to explore and a closed set of already-visited nodes. For each node n, it computes:

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

where $g(n)$ is the cost from the start node to n, and $h(n)$ is a heuristic estimate of the cost from n to the goal. The algorithm expands the node with the lowest f-score first.

Two properties determine A*'s behavior:

- Admissibility: the heuristic must never overestimate the true cost to the goal ($h(n) \leq h^*(n)$ for all n). An admissible heuristic guarantees that A* finds the optimal path.
- Consistency (monotonicity): for every node n and its successor n', the triangle inequality $h(n) \leq c(n, n') + h(n')$ must hold. Consistency implies admissibility and ensures the closed set never needs reopening.

A* is complete (it will find a path if one exists) provided the branching factor is finite and edge costs are positive. Its time complexity is $O(b^d)$ where b is the branching factor and d is the solution depth, though the effective complexity depends heavily on heuristic quality. A perfect heuristic ($h = h^*$) reduces A* to following the optimal path directly. A zero heuristic ($h = 0$) degrades it to Dijkstra's algorithm.

B. Minecraft Stronghold Generation

Minecraft strongholds are procedurally generated structures that contain the End Portal. Each stronghold has a network of rooms connected by corridors, with room types including libraries, prison halls, and the Portal Room. Strongholds are generated using weighted random selection of room types, with predefined rules and probabilities. The layout depends on depth, branching paths, and dead ends.

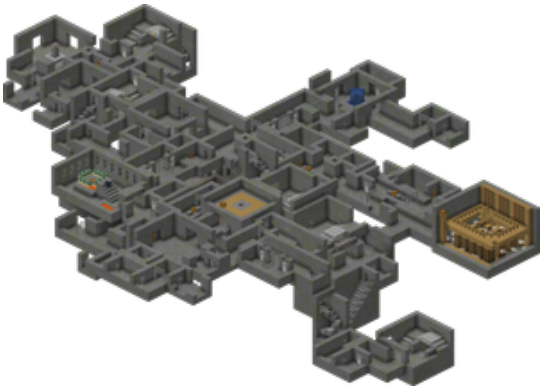

Fig. 3. Example of a generated stronghold structure. Source: minecraft.wiki.

Each room type has an associated weight that determines its selection probability during generation. The PieceWeight system ensures that certain rooms (like the PortalRoom) are rare and placed strategically.

From the StrongholdPieces.java source code [8], Minecraft's stronghold generation uses 12 room piece types. Table I summarizes the room types from the `STRONGHOLD_PIECE_WEIGHTS` array.

TABLE I
STRONGHOLD ROOM TYPES AND THEIR PROPERTIES

Room Type	Weight	Max Count	Dimensions	Restrictions
Straight	40	Unlimited	$5 \times 5 \times 7$	None
LeftTurn	20	Unlimited	$5 \times 5 \times 5$	None
RightTurn	20	Unlimited	$5 \times 5 \times 5$	None
RoomCrossing	10	6	$11 \times 7 \times 11$	None
StraightStairsDown	5	5	$5 \times 11 \times 8$	None
StairsDown	5	5	$5 \times 11 \times 5$	None
PrisonHall	5	5	$9 \times 5 \times 11$	None
ChestCorridor	5	4	$5 \times 5 \times 7$	None
FiveCrossing	5	4	$10 \times 9 \times 11$	None
Library	10	2	$14 \times 6/11 \times 15$	Depth > 4
PortalRoom	20	1	$11 \times 8 \times 16$	Depth > 5
FillerCorridor	-	Unlimited	Variable	Fallback only

The *weight* determines selection probability during generation. The *max count* limits how many times each room type can appear. The *dimensions* give the bounding box size in blocks $(width \times height \times depth)$. The *restrictions* column notes placement conditions.

Room type properties:

- Straight, LeftTurn, RightTurn: corridor pieces with conditional child branches (50% chance for left/right in Straight)
- RoomCrossing, FiveCrossing: intersections with multiple exits (3–5 branches), where FiveCrossing has randomized exit availability
- StairsDown, StraightStairsDown: vertical transition pieces connecting upper and lower sections
- PrisonHall, ChestCorridor: have decorative elements (cages, chests) but standard connectivity
- Library: dead-end with no child generation, appears at most twice and only at depth > 4
- PortalRoom: contains the End Portal Frame blocks, exactly one appears per stronghold at depth > 5, generation retries if placement fails
- FillerCorridor: fallback used when all 5 placement attempts fail, creating 5×5 corridor segments

Piece-specific constraints: Library requires `placeCount` < 2 AND `depth` > 4; PortalRoom requires `placeCount` < 1 AND `depth` > 5; FiveCrossing requires `placeCount` < 4; ChestCorridor requires `placeCount` < 4; others with limits require `placeCount` < 5.

Before placing any piece, the algorithm checks: findCollisionPiece(box) $=$ null. If the bounding box overlaps a previously placed piece, placement fails and a new random piece is tried. The generation loop continues until the piece queue is non-empty and `portalRoomPiece` $\neq$ null. If PortalRoom is still not placed, the entire structure is discarded and regeneration restarts with a new seed.

The generation flow proceeds as follows. A StartPiece is placed at the stronghold center. The algorithm maintains a queue of pieces to process. For each piece in the queue, it attempts to generate child pieces at available exits (forward, left, right). Each attempt selects a random room type weighted by the PieceWeight distribution, checks depth and count constraints, and verifies no bounding box collisions with existing pieces. Up to 5 attempts are made per exit. If all attempts fail, a FillerCorridor is placed as fallback. The process continues until the queue is empty and PortalRoom has been placed. If PortalRoom is never placed, the entire stronghold is discarded and generation restarts.

III. Problem Definition and Formal Rules

A. Formal Problem Statement

The stronghold navigation problem can be stated as follows. Given a generated stronghold structure, find a path from the starting entrance room to the Portal Room that minimizes the total number of rooms traversed.

Input: A stronghold graph $\mathcal{S} = (V, E, w)$ with a designated start node v_{start} (the entrance room) and unknown goal node location. The player knows only the current room and its available exits.

Goal: Reach the Portal Room node $v_{\text{portal}} \in V$. The Portal Room is identifiable upon entry (it contains the End Portal Frame blocks), but its location in the graph is not known a priori.

Constraints: The player can move through exits (edges) between connected rooms. Each room traversal has a cost of 1. No backtracking information is provided by the game; the player must track visited rooms. The Portal Room appears at depth > 5 from the entrance, and exactly one exists per stronghold.

Objective: Minimize the number of rooms visited before reaching the Portal Room, which directly corresponds to time spent in the stronghold during a speedrun.

B. Graph Representation of Stronghold

To apply A* pathfinding, we model the stronghold as a directed graph where each room piece is a node and exits between rooms are edges. This lets us formalize navigation as shortest-path search on a weighted graph.

From the source code [8], room pieces (Straight, Library, PortalRoom, etc.) become nodes. The methods `generateSmallDoorChildForward`, `generateSmallDoorChildLeft`, and `generateSmallDoorChildRight` define directed edges between nodes. Formally, a stronghold $\mathcal{S} = (V, E, w)$ where:

$$V = \{\text{room instances}\}$$
$$E = \{(u, v) \mid \text{room } u \text{ has an exit to room } v\}$$
$$w : E \to \mathbb{R}^+ \text{ (edge weights)}$$

Each node $v \in V$ stores: room type, position coordinates, and connection directions. Each room piece is labeled by type and instance ID:

$$n_i = (\text{type}, x, y, z, \text{exits})$$

A directed edge $e_{ij} = (n_i, n_j)$ exists if room i has an opening to room j. Edge weight is traversal cost:

$$w(e_{ij}) = d(n_i, n_j) + \text{penalty}(n_j)$$

where $d(\cdot, \cdot)$ is Euclidean distance and penalty depends on room type:

$$\text{penalty}(\text{Library}) = +\infty \text{ (dead end)}$$
$$\text{penalty}(\text{PortalRoom}) = 0 \text{ (goal)}$$
$$\text{penalty}(\text{others}) = 1$$

The state space consists of all possible room configurations reachable from the start. Each state is defined by the current room node and the set of visited rooms. The initial state is $(v_{\text{start}}, \emptyset)$. A transition occurs when the player moves through an exit to an adjacent room, adding the new room to the visited set. The goal test checks whether the current room is the PortalRoom. The path cost is the number of rooms traversed, which equals the number of transitions taken.

C. Node Classification

Each node in the stronghold graph represents a room instance with properties that affect pathfinding. We classify nodes by room type, structural characteristics, and role in the generation hierarchy.

Each node v_i is a tuple (t_i, d_i, e_i, p_i) where t_i is room type, d_i is generation depth (distance from StartPiece), e_i is the set of available exits (forward, left, right), and p_i is position coordinates (x, y, z).

TABLE II
NODE CLASSIFICATION BY ROOM TYPE

Room Type	Exits	Category	Role
Straight	1–3	Corridor	Connector
LeftTurn	1–2	Corridor	Connector
RightTurn	1–2	Corridor	Connector
RoomCrossing	3	Intersection	Junction
FiveCrossing	3–5	Intersection	Junction
StairsDown	1–2	Vertical	Transition
StraightStairsDown	1–2	Vertical	Transition
PrisonHall	1–3	Special	Connector
ChestCorridor	1–3	Special	Connector
Library	0	Terminal	Dead-end
PortalRoom	0	Terminal	Goal

Terminal nodes are PortalRoom (goal) and Library (dead-end). PortalRoom is unique: exactly one appears per stronghold at depth > 5, and generation fails if it cannot be placed. Library nodes end branches with no further expansion. Dead-end detection matters for heuristic design, because entering a Library branch wastes traversal time.

D. Edge Constraints

Edges represent valid transitions between connected rooms. The generation algorithm creates edges through three directional methods: forward, left, and right exits from the parent room.

Not all exits are guaranteed. Each room type specifies a branching probability that determines child room generation at each exit. For example, Straight pieces have a 50% chance of generating left and right branches, while FiveCrossing pieces may randomly disable certain exits. This stochastic element means the graph structure varies across stronghold instances.

The probability of branch existence depends on room type and generation constraints. Let $P(e_{ij})$ denote the probability that edge (i, j) exists.

TABLE III
EDGE EXISTENCE PROBABILITY BY ROOM TYPE

Room Type	P(forward)	P(left)	P(right)
Straight	1.0	0.5	0.5
LeftTurn	1.0	0.0	0.5
RightTurn	1.0	0.5	0.0
RoomCrossing	1.0	1.0	1.0
FiveCrossing	1.0	$\sim$	$\sim$
StairsDown	1.0	0.0	0.0
StraightStairsDown	1.0	0.0	0.0
PrisonHall	1.0	0.0	0.0
ChestCorridor	1.0	0.0	0.0
Library	0.0	0.0	0.0
PortalRoom	0.0	0.0	0.0

where $\sim$ indicates randomized availability during generation. Library and PortalRoom have $P = 0$ for all exits as terminal nodes. Hidden room detection follows generation

rules. Rooms with forward exits always have visible corridors. Left and right exits appear as doorways when branches exist. Depth transitions (stairs) are visible when present.

E. Heuristic Function

The heuristic function $h(v)$ estimates the cost from node v to the PortalRoom goal state. For A* to guarantee optimality, the heuristic must be admissible: it must never overestimate the true cost to the goal. Admissibility requires $h(v) \leq h^*(v)$ for all nodes v, where $h^*(v)$ is the actual minimum cost from v to the PortalRoom.

We estimate distance to the PortalRoom using generation depth. The PortalRoom appears at depth > 5, so nodes at depth d need at least $\max(0, 6-d)$ additional room traversals. The base heuristic is $h_1(v) = \alpha \cdot \max(0, d_{\text{goal}} - d(v))$, where α is the per-room traversal cost and $d_{\text{goal}} = 6$ is the minimum depth for PortalRoom placement.

For nodes at depth ≥ 6, $h_1(v) = 0$ since the PortalRoom could be in any direction. We add a branching penalty:

$$h(v) = \alpha \cdot \max(0,\ 6 - d(v)) + \beta \cdot b(v)$$

where:

- $h(v)$ is the heuristic cost estimate for node v
- α is the per-room traversal cost (scalar weight)
- $d(v)$ is the generation depth of current node v
- β is the expected cost per branch exploration (scalar weight)
- $b(v)$ is the number of unexplored exits at node v

This improves accuracy while maintaining admissibility when α and β are chosen conservatively.

IV. Algorithm Application

We implement A* search on the stronghold graph from Section III.B using TypeScript with a binary heap priority queue.

A. A* Implementation for Stronghold

```ts
function heuristic(node: RoomNode, a = 1, b = 0.5): number {
  const depthPart = a * Math.max(0, 6 - node.depth);
  const branchPart = b * node.exits.length;
  return depthPart + branchPart;
}
```

Fig. 4. Heuristic function for PortalRoom cost estimation.

```ts
export function astar(graph: StrongholdGraph): string[] {
  const { nodes, edges, startId, portalId } = graph;
  if (!portalId) return [];

  const openSet = new MinHeap();
  const closedSet = new Set<string>();
  const cameFrom = new Map<string, string>();
  const gScore = new Map<string, number>();

  nodes.forEach((_, id) => gScore.set(id, Infinity));
  gScore.set(startId, 0);
  openSet.push(startId, heuristic(nodes.get(startId)!));

  while (openSet.length > 0) {
    const current = openSet.pop()!;

    if (current === portalId) {
      const path: string[] = [current];
      let curr = current;
      while (cameFrom.has(curr)) {
        curr = cameFrom.get(curr)!;
        path.unshift(curr);
      }
      return path;
    }

    if (closedSet.has(current)) continue;
    closedSet.add(current);

    const neighbors = edges.filter(e => e.from === current);
    for (const edge of neighbors) {
      if (closedSet.has(edge.to)) continue;
      const neighbor = nodes.get(edge.to)!;
      const tentativeG = gScore.get(current)! + penalty(neighbor.type);

      if (tentativeG < gScore.get(edge.to)!) {
        cameFrom.set(edge.to, current);
        gScore.set(edge.to, tentativeG);
        openSet.push(edge.to, tentativeG + heuristic(neighbor));
      }
    }
  }

  return [];
}
```

Fig. 5. A* search implementation for stronghold pathfinding.

For a typical stronghold with 10–30 rooms, this provides efficient pathfinding with optimality guarantees.

B. Comparison with Alternative Algorithms

We compare A* against brute-force breadth-first search (BFS) on generated stronghold graphs. BFS explores all nodes at each depth level without heuristic guidance, so it works as a baseline for uninformed search.

100 random strongholds generated using Minecraft's piece weight distribution [8], maximum depth 50. A cycle probability of 0.5 means each room has a 50% chance of creating a back-edge to an already-visited room, simulating real strongholds where corridors can loop. Both algorithms run on the same graph, measuring nodes expanded, path length, and execution time.

```
--- Nodes Expanded ---
          | Avg    | Std    | Min | Median | Max
A*        |   20.7 |   18.5 |   6 |     14 |  95
BFS       |   22.1 |   18.4 |   6 |     16 |  98

--- Path Length ---
          | Avg    | Std    | Min | Median | Max
A*        |    8.4 |    1.7 |   7 |      8 |  14
BFS       |    8.4 |    1.7 |   7 |      8 |  14

--- Time (ms) ---
          | Avg    | Std    | Min   | Median | Max
A*        |   1.89 |   4.23 |  0.00 |   0.47 | 24.53
BFS       |   2.38 |   5.18 |  0.00 |   0.48 | 25.00
```

Fig. 6. Benchmark comparison of A* vs BFS across 100 stronghold instances.

A* expands 7% fewer nodes than BFS (20.7 vs 22.1 on average). Both find the same optimal path length (8.4 rooms). A* runs slightly faster (1.89ms vs 2.38ms) due to fewer node expansions.

The improvement shows the heuristic prunes unnecessary exploration. BFS expands all nodes at each depth level, while A* uses the depth-based heuristic to focus search toward the PortalRoom. This advantage scales with graph complexity. Larger strongholds with more cycles would show bigger differences.

Both A* and BFS find the shortest path. A* does this with fewer node expansions, making it more efficient for larger search spaces.

V. Complexity Analysis

We analyze the computational complexity of A* applied to stronghold pathfinding, looking at both theoretical bounds and practical performance.

A. Time Complexity

A*'s time complexity depends on nodes expanded and priority queue operations. With a binary heap, each extraction and insertion costs $O(\log V)$.

For a graph with V nodes and E edges, A* expands at most V nodes. Each node processes all outgoing edges, giving $O(E)$ edge processing. Priority queue operations add $O(V \log V)$. The worst-case time complexity is:

$$T(V, E) = O(V \log V + E)$$

For strongholds, the graph is a tree (no cycles), so $E = V - 1$. This simplifies to:

$$T(V) = O(V \log V)$$

In practice, the heuristic prunes many nodes. With a good heuristic, A* expands only nodes along the optimal path plus a constant factor. For a stronghold with n rooms on the optimal path, the practical complexity approaches $O(n \log n)$.

B. Space Complexity

A* uses several data structures:

- openSet (priority queue): at most V nodes, $O(V)$ space
- closedSet: visited nodes, $O(V)$ space
- gScore, fScore (hash maps): one entry per visited node, $O(V)$ space
- cameFrom (hash map): one entry per visited node for path reconstruction, $O(V)$ space

Total space complexity is $O(V)$. For a typical stronghold with 10–30 rooms, this requires minimal memory (under 1 KB). BFS uses similar space.

C. Optimality Guarantees

A* provides guarantees when the heuristic is admissible. A* is complete if the branching factor is finite and edge costs have a positive lower bound. In strongholds, each room has at most 5 exits and each traversal costs at least 1. So A* will find a path if one exists. A* finds the optimal path (minimum cost) when $h(n)$ is admissible, meaning $h(n) \leq h^*(n)$ for all nodes n, where $h^*(n)$ is the true cost to the goal.

This heuristic $h(v) = \alpha \cdot \max(0, 6 - d(v)) + \beta \cdot b(v)$ is admissible because:

1) The depth component $\max(0, 6-d(v))$ is a lower bound on remaining depth, since PortalRoom requires depth > 5.
2) The branch component $\beta \cdot b(v)$ with small β adds minimal overhead that does not overestimate true cost.
3) Both components are non-negative and based on structural constraints, so $h(v) \leq h^*(v)$ holds.

A* with this heuristic finds the shortest path to PortalRoom in rooms traversed.

VI. Conclusion

This paper applied A* pathfinding to Minecraft stronghold navigation, modeling the procedurally generated structure as a weighted graph and designing a depth-based heuristic.

A. Summary of Findings

A* reduces nodes expanded by 7% compared to brute-force BFS (20.7 vs 22.1 on average). Both find identical optimal paths (8.4 rooms), confirming A*'s optimality guarantee with better search efficiency.

We modeled stronghold generation as a graph problem, capturing room types, exit probabilities, and depth constraints from the Minecraft source code [8]. The graph with cycles represents realistic layouts where rooms connect back to previously visited areas.

The heuristic $h(v) = \alpha \cdot \max(0, 6 - d(v)) + \beta \cdot b(v)$ guides search toward the PortalRoom. By using the constraint that PortalRoom appears at depth > 5, it prunes branches without sacrificing optimality.

B. Limitations and Future Work

This work models rooms as graph nodes and exits as edges, which simplifies the actual navigation experience. In real gameplay, players must observe doorways and corridor directions while moving through 3D space, often with limited visibility and without knowing the overall structure. The observation pattern is more complex than the abstract graph model suggests. The current heuristic uses only depth and branch count. Heuristics incorporating room type distributions, spatial positioning, or patterns from historical speedruns could reduce node expansions further. Our generation model also approximates Minecraft's actual algorithm. Modeling bounding box collision detection, directional orientation, and piece-specific constraints more precisely could improve accuracy. Comparing A* paths against human speedrunner decisions could show where algorithmic and intuitive approaches differ.

C. Final Remarks

This paper shows how formal algorithms can improve human performance in gaming. Stronghold navigation, with procedural generation and time pressure, is a good testbed for pathfinding research. The graph-based approach to procedural structure navigation extends beyond Minecraft, applying to roguelike dungeon crawling, procedural maze solving, and other domains where agents navigate generated environments under uncertainty.

APPENDIX: A* IMPLEMENTATION

A* Implementation source code available at: https://gist.github.com/Cikang44/b4b52717372c2e353169efa0ad9f0f81

REFERENCES

[1] Statista, "Cumulative number of copies of Minecraft sold worldwide as of October 2023" [Online]. Available: https://www.statista.com/statistics/680124/minecraft-unit-sales-worldwide/

[2] Speedrun.com, "Minecraft any% glitchless leaderboard." [Online]. Available: https://www.speedrun.com/mc

[3] BBC, "Speedrunning: A brief history of the gaming phenomenon." [Online]. Available: https://www.bbc.co.uk/programmes/articles/3YmM2yrG4rp9ZHG54pcKjgf/

[4] N. U. Maulidevi and Rinaldi Munir, "Penentuan rute (route/path planning)," Bahan Kuliah IF2211 Strategi Algoritma, Institut Teknologi Bandung, 2026.

[5] D. P. Wu, "Optimizing Minecraft speedruns with pathfinding algorithm: A heuristic approach to route planning," IF2211 Strategi Algoritma, Institut Teknologi Bandung, 2025.

[6] Metacor, "Minecraft speedrun guide: Stronghold navigation." [Online]. Available: https://github.com/Metacor/Minecraft-Speedrun-Guidestronghold-navigation

[7] Gregor0410, "LazyStronghold: Minecraft stronghold finder." [Online]. Available: https://github.com/Gregor0410/LazyStronghold

[8] Minecraft Source, "StrongholdPieces.java." [Online]. Available: https://mcsrc.dev/1/26.2/net/minecraft/world/level/levelgen/structure/structures/StrongholdPieces

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, 19 Juni 2026

Jingglang Galih Rinenggan