Physics-Aware A*: Sequential Filtering for Jump Feasibility in 2D Platformers

Gabriel Barbosa de Oliveira[1], **Leonardo Tortoro Pereira**[2], **Maria Victória Brandão Barros**[1]

[1]Institute of Mathematical and Computer Sciences, University of São Paulo, São Carlos, São Paulo, Brazil

[2]São Paulo State University (UNESP), Institute of Geosciences and Exact Sciences (IGCE)

gabrieloliveira00@outlook.com, leonardo.t.pereira@unesp.br, mariabrandaob@gmail.com

***Abstract. Introduction**: In 2D platform games, movement is governed by physics (gravity, limited jumps, air momentum). Traditional pathfinding algorithms like A* assume free movement, generating geometrically valid but physically impossible paths. **Objective**: We propose an incremental physical validation approach integrated into A*, using augmented states and a modular filter chain to find traversable paths in platformer levels. **Methodology**: The algorithm augments states with movement phases (grounded/jumping/falling), jump origin, and initial velocity. A sequential filter chain validates collisions, reachability (via UAM equations), momentum conservation, and trajectory blocking. Experiments on four classic maps compare our method against standard A*. **Results**: Our method finds 3–13× more physically valid paths than A*, but at the median consumes 10–35× more memory and runs 21–97× slower. The trade-off is acceptable for offline pre-computation and individual NPCs.*

***Keywords** Pathfinding, Platform Games, A* Algorithm, Physical Simulation, Game AI.*

1. Introduction

In 2D platform games, character movement is governed by physics simulation: gravity, jumps with limited range, and the inability to change direction mid-air determine which transitions between grid cells are executable. Traditional pathfinding algorithms such as A* [Hart et al. 1968, Russell e Norvig 2021] assume free movement between adjacent valid cells, an assumption that does not hold in this domain. As a result, they generate geometrically correct paths that are often impossible to execute, as illustrated in Figure 1.

This gap is critical for several practical applications: autonomous agents (NPCs) require truly traversable paths; level design tools need to identify unreachable areas; and procedural level validation depends on verifying physical completability. Previous work [Tremblay et al. 2014, Iskandar et al. 2020] indicates that research on pathfinding in platform games rarely integrates physical constraints such as gravity and jumps. Recently, [Athaillah et al. 2025] proposed CheckJump, which pre-computes a connectivity graph (world mapping), concentrating the cost upfront and amortizing it over multiple queries, an effective approach for static maps, but one that requires full reconstruction when the environment or mechanics change.

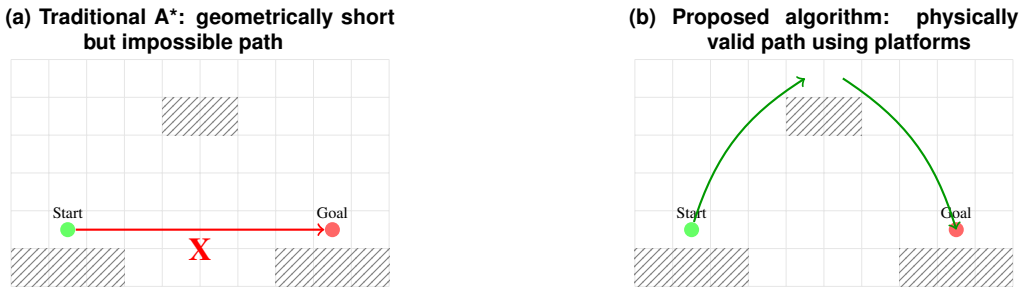

Figure 1. Comparison between geometric and physical paths

This work explores a complementary strategy: incremental physical validation during the search. We propose a sequential filter architecture integrated into A*, which uses augmented states containing position, movement state (grounded/jumping/falling), jump origin, and initial velocity to decide, at each node expansion, which neighbors are physically reachable. Thus, only filter-validated states compose the search space, eliminating the need for global pre-processing. This physics-aware A* is called PAA*.

Experiments on four representative maps (Super Mario Bros., The Lion King, and two areas of Castlevania: Symphony of the Night) show that the proposed algorithm finds 3–13 times more physically validated paths than traditional A* (see Section 4). Figure 2 shows paths from A* and PAA*. On the other hand, the computational costs are significantly higher: at the median, it consumes 10–35 times more memory and runs 21–97 times slower (Section 4.2). We discuss the practical viability of this trade-off and the limitations of the current modeling.

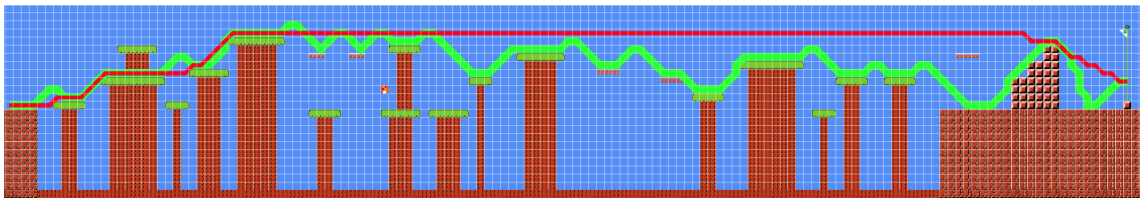

Figure 2. Super Mario Bros. Level 1-3 adapted in Godot and solved by a classic A* (red) vs. PAA* (green). Our algorithm shows a more natural path for a platforming agent

2. Related Works

A* [Hart et al. 1968] is widely recognized as one of the most efficient strategies for finding minimal cost paths on graphs or grids. It combines the actual cost from the start to the current node with an admissible heuristic, guaranteeing optimality [Russell e Norvig 2021]. However, this theory assumes that all transitions allowed by the graph topology are practically executable. In platform games, this premise is invalid: a diagonal move may be geometrically adjacent but unreachable if the agent is in the air, has insufficient horizontal velocity, or is blocked by a ceiling.

A vast body of literature addresses A* optimization on grids (online pruning [Harabor e Grana 2011], benchmarks [Sturtevant 2012], dynamic search [Moghadam et al. 2024]), but all operate under the same free movement assumption, not

tackling physical constraints like gravity and jumps. Iskandar et al. [Iskandar et al. 2020] highlight that previous research on pathfinding in platform games often ignored vertical movements influenced by gravity, jumps, and falls, pointing to a significant gap in integrating physics with path planning.

Among the works that effectively handle movement physics, CheckJump by [Athaillah et al. 2025] stands out. Their method performs pre-processing called *world mapping*: for each ledge cell of the map, jumps are simulated to identify all reachable platforms, building a connectivity graph. This pre-processing costs 36–146 ms, but afterwards path queries are fast (plain A* on the sparse graph). The approach is advantageous for static maps with many queries, but requires full reconstruction when the environment or mechanics change (e.g., moving platforms, *power-ups*). Their evaluation uses three custom maps from a single unreleased game, all with fixed height (10 rows) and at most 720 cells.

3. Method

This work explores a complementary strategy: incremental physical validation during the expansion of A* states. The main differences are: no physics-aware pre-processing, with jump feasibility validated on demand during the search and a lightweight geometric pre-computation of agent-body collisions performed once per map/agent-size pair, independent of physics parameters; augmented states, where each node stores position, movement state (grounded/jumping/falling), last on-ground position, and initial velocity; and a modular filter architecture, where checks (collisions, reachability, momentum, trajectory) are encapsulated in pluggable filters, extensible without changing the A* core (Figures 3 and 4).

While CheckJump concentrates the cost upfront and amortizes it over multiple searches, our approach distributes the cost across the search, making it more suitable for dynamic contexts or one-shot queries. Additionally, our experiments (Section 3.2) use four maps from published commercial titles spanning diverse topologies and ranging from $\sim$3,900 to $\sim$21,700 cells. A direct empirical comparison between the two strategies is left for future work.

A 2D platform game map is a *grid* $w \times h$ where each cell (x, y) is either solid (blocks movement) or non-solid (walkable). Suspended cells are non-solid cells with no solid cell directly below them. Agent movement is governed by gravity g, constant horizontal velocity v_{0x}, initial vertical jump velocity ($v_{0y} < 0$), and collisions. The trajectory follows uniformly accelerated motion (UAM) on the y-axis and uniform motion (UM) on the x-axis [Millington e Funge 2016, Eberly 2003]. We adopt the game convention where y increases downward: dy = -1 is upward (jumping) and dy = +1 is downward (falling).

We define:

Ground cell: a non-solid cell (x, y) such that $(x, y + 1)$ is solid or outside the *grid*.
Jump: a path whose start and end are ground cells and all intermediate cells are suspended.
Impossible jump: a jump whose target is unreachable given the origin and v_0.
Traversable path: a path with no impossible jumps.

The problem: find the shortest traversable path between two ground cells. Figure 1 (above) contrasts a geometrically short (A*) and a feasible (proposed) path.

To eliminate impossible states, we define an **augmented state**:

$$s = \langle (x, y), \text{state}, (x_g, y_g), \mathbf{v}_0 \rangle$$

where (x, y) is the current cell; state $\in \{\text{Grounded}, \text{Jumping}, \text{Falling}\}$; (x_g, y_g) is the last ground cell; $\mathbf{v}_0 = (v_{0x}, v_{0y})$ with $v_{0y} = 0$ when grounded or falling from a ledge; $v_{0y} < 0$ is preserved when the fall originates from a jump. This information allows us to decide, during expansion, whether a transition is possible (e.g., if falling, only downward moves are allowed).

Each candidate transition is validated by a modular filter chain (Figure 3). Each filter may **accept** (possibly transforming the state), **reject** (discard), or **transform** the candidate state. Rejection by any filter stops the chain.

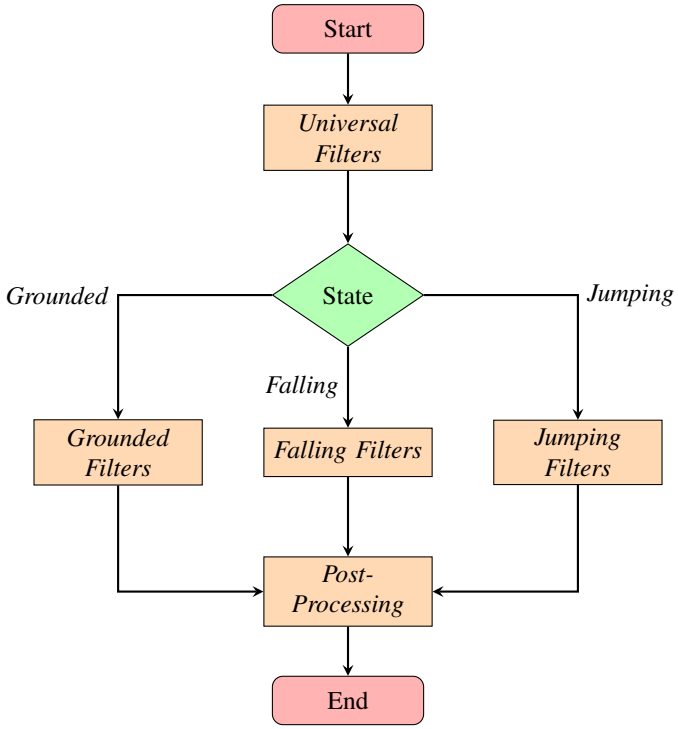

Figure 3. Filter chain flowchart

The universal filters consist of two rejection rules. First, the valid cell filter discards any position where the agent's multi-cell footprint (the agent might not fit in a single cell) would overlap a solid cell or extend beyond the grid boundaries. Second, the blocked diagonal filter rejects diagonal moves whenever both cardinal neighbors are obstructed relative to the agent's body.

Each movement state applies a specific set of filters:

When grounded (Figure 4a): The *Grounded Movement* filter transforms according to direction (d_x, d_y):

- $d_y = -1 \rightarrow$ *Jumping* with $\mathbf{v}_0 = (v_{0x}, v_{0y})$;

- $d_y = 0$: stays *Grounded* if the destination is on the ground; otherwise, it becomes *Falling* with $(v_{0x}, 0)$;
- $d_y = 1 \rightarrow$ *Falling* with $(v_{0x}, 0)$.

When jumping (Figure 4b): Four filters in sequence:

1. *Jumping Movement*: if $d_y = -1$ stays *Jumping*, it otherwise becomes *Falling*.
2. *Momentum Preservation*: rejects horizontal reversal ($\text{sign}(x - x_g) \times \text{sign}(d_x) = -1$).
3. *Jump Reachability*: using UAM/UM, checks whether there exists $t > 0$ such that $\Delta y = v_{0y}t + \frac{1}{2}gt^2$ and $|v_{0x}t| \geq |\Delta x_{\text{adjusted}}|$.
4. *Jump Trajectory*: approximates the parabola by two line segments (start→peak, peak→goal) and detects collisions; if a ceiling is hit during the ascent, the fall from the collision point is evaluated for reachability.

When falling (Figure 4c): Five filters in sequence:

1. *Falling Movement*: transforms to *Falling*, preserving origin and $\mathbf{v}_0$.
2. *Downward Only*: rejects if $d_y \neq 1$ (only downward and diagonal-downward moves allowed).
3. *Momentum Preservation*: same rule as in jumping.
4. *Jump Reachability*: same kinematic check as in the jumping state. Note that v_{0y} may be negative when the fall originates from a jump (the original jump velocity is preserved).
5. *Jump Trajectory*: validates the trajectory from the original jump origin to the current falling position, detecting collisions along the approximated arc.

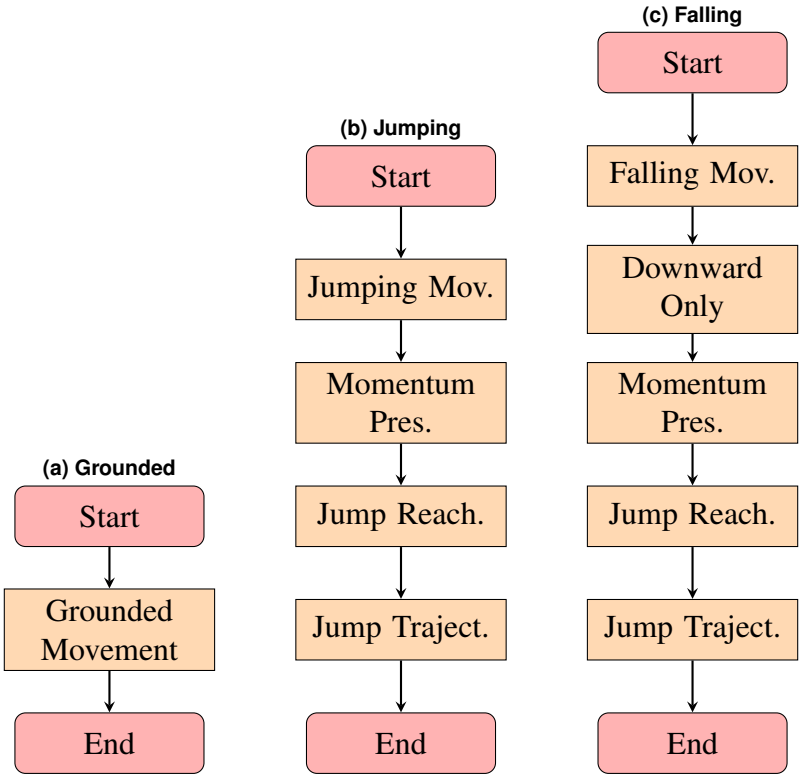

Figure 4. Motion filter flowcharts

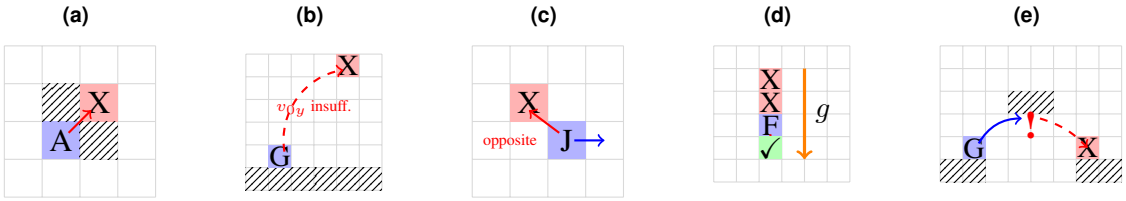

Figure 5. Examples of rejections: (a) blocked diagonal (A=any state, X=rejected); (b) unreachable height (G=grounded); (c) in-air direction change (J=jumping); (d) upward move while falling (F=falling); (e) trajectory blocked by ceiling

Figure 5 illustrates typical rejections.

Then, we have a post-processing filter, *Transition to Ground*: if the cell is on the ground, it updates the state to *Grounded*, resets the jump origin, and sets $\mathbf{v}_0 = (0, 0)$.

3.1. Integration with A*

The traditional A* algorithm [Russell e Norvig 2021] is preserved. The only modification is in **neighbor generation**: for each direction, a candidate state is created and passed through the filter chain corresponding to the current movement state. Only accepted states (after possible transformations) are enqueued. The heuristic applied is Euclidean distance, which is admissible. Both the proposed algorithm and the baseline A* share the same geometric neighbor generation, pre-computing valid positions by filtering out collisions and locations the agent cannot fit in. The cost of this pre-computation is presented in Table 1.

3.2. Experimental Setup

We re-created and tested four maps from different classic games: Super Mario Bros. (SMB) (jumps at the limit), Lion King (high verticality), Symphony of the Night (SOTN) Gallery (ground-based), SOTN Stairs (staircase with many possible paths). Each map is used with cell granularity factors 1 and 2 (2×2 subdivision), totaling eight configurations. Factor 2 multiplies the cell count by 4, testing the algorithm with state explosion. Dimensions are given in Table 1.

All ledge cells in each map are identified to generate our test cases. For each ordered pair (l_i, l_j) with $i \neq j$, both traditional A* and the proposed algorithm are executed.

We collected the following metrics: number of validated paths (indirect validation, reusing the filters to identify impossible jumps), estimated memory (MB, analytically computed from data structure sizes), and time (s). Experiments were performed in Godot 4.5 (C#/GDScript). The implementation is openly available on GitHub[1].

4. Results

The experiments evaluated the proposed approach across eight configurations, varying three independent variables: map topology, granularity factor, and algorithm type. The dependent variables are the number of validated paths (our core effectiveness

[1] https://github.com/bdogabriel/paastar-sbgames

claim), estimated memory, and execution time, which captures the practical trade-off. Complementary results, such as datasets and images showing generated paths, are available in our OSF project [2] (Section A).

4.1. Effectiveness: validated paths

Table 1 compares the number of paths found by traditional A* and the proposed algorithm (with physical validation through filters). The reported values refer to paths that passed all reachability and collision checks implemented in the filters.

Table 1. Validated paths per configuration. Precomp. is the one-time agent-body collision grid computation (shared by both algorithms).

Map	Dimensions	Factor	A*	Proposed	Precomp. (ms)
SMB	176×23	1	173	2010	3.4
SMB	352×46	2	193	2428	14.3
Lion King	63×86	1	1010	6757	5.1
Lion King	126×172	2	1144	6757	37.6
SOTN Gallery	96×41	1	137	841	3.4
SOTN Gallery	192×82	2	145	870	24.2
SOTN Stairs	45×105	1	2005	6476	4.2
SOTN Stairs	90×210	2	949	6480	29.0

The proposed algorithm consistently finds 3–13 times more paths that satisfy the implemented physical constraints. Important observations:

1. **Stability**: for most maps, the number of valid paths of the proposed algorithm remains stable across both granularities (e.g., Lion King: 6757 in both; SOTN Stairs: 6476 vs 6480), indicating that physical validation captures intrinsic level properties largely independent of discretization. SMB shows an increase (2010 vs 2428), likely due to finer resolution enabling additional edge-case jumps.
2. **Inconsistency of traditional A***: A* varies significantly with granularity in some cases (SOTN Stairs: 2005 $\rightarrow$ 949), revealing that its paths may reflect discretization artifacts.
3. **Variation across maps**: the ratio between algorithms ranges from approximately 3× (SOTN Stairs, factor 1) to 13× (SMB), reflecting the topological characteristics.

4.2. Computational costs

The additional costs arise from the use of augmented states and the filter chain. Table 2 summarizes estimated memory (in MB, computed analytically from data structure sizes and hash table overhead constants) and execution time (in seconds) statistics for the Lion King map, the most complex scenario tested. The values shown are quartiles extracted from the distributions obtained in the experiments.

The proposed algorithm consumes, at the median, approximately 10–35 times more memory and runs 21–97 times slower than traditional A*. In the worst case (Lion King, factor 2), the median memory of the proposed method is $\sim$6.7 MB versus $\sim$0.19

[2] https://osf.io/mcpb4/

Table 2. Performance on the Lion King map (per-execution values)

Factor	Algorithm	Metric	Min	Q1	Median	Q3	Max
1	A*	Memory (MB)	0.0011	0.0202	0.0526	0.1290	0.4724
	Proposed	Memory (MB)	0.0022	0.1801	0.9008	4.7037	13.3171
	A*	Time (s)	6e-6	0.000084	0.000230	0.000610	0.00396
	Proposed	Time (s)	1.2e-5	0.001285	0.007855	0.04710	0.16093
2	A*	Memory (MB)	0.0016	0.0655	0.1895	0.4917	1.8695
	Proposed	Memory (MB)	0.0041	1.2111	6.7021	34.0760	96.8234
	A*	Time (s)	9e-6	0.000311	0.000908	0.002427	0.01103
	Proposed	Time (s)	1.9e-5	0.01385	0.088417	0.48596	1.47687

MB for A* ($\sim$35× larger), and the median time is $\sim$88 ms versus $\sim$0.9 ms ($\sim$97× slower). These costs are inherent to the expansion of the state space (each cell can generate multiple states depending on movement, origin, and velocity) and the execution of physics calculations (UAM equations, trajectory tracing) for each neighbor.

Increasing granularity (factor $2 \rightarrow 2\times2$ subdivision, quadrupling the number of cells) affects the algorithms differently. While traditional A* exhibits roughly quadratic growth, the proposed algorithm shows superlinear growth. On the Lion King map, when doubling the factor, the median time of the proposed method grows from $\sim$7.9 ms to $\sim$88 ms ($\sim$11×). Then, the median memory of the proposed method grows from $\sim$0.90 MB to $\sim$6.7 MB ($\sim$7.4×).

This behavior occurs because the state space expands combinatorially: more cells generate more combinations of movement states, jump origins, and initial velocities.

Memory was estimated analytically from data structure sizes; actual C# runtime allocation may differ due to allocator overhead.

The overhead was consistent across all maps and granularities; the full per-map quartile distributions are in the supplementary material (Section A). At factor 1, median memory ratios ranged from 10× (SMB) to 19× (SOTN Gallery), and median time slowdowns from 21× (SMB) to 38× (SOTN Gallery). At factor 2, these grew to 23–33× and 58–81× respectively. Nevertheless, absolute medians remained below 11 MB and 120 ms across all three maps, acceptable for offline pre-computation, level validation, and individual NPCs.

5. Discussion

The results show that the proposed algorithm is effective at finding paths that respect physical constraints, while traditional A* fails by up to an order of magnitude (Table 1). However, this effectiveness comes with significantly higher computational costs (Table 2). This section discusses the trade-offs, compares with related approaches, assesses practical viability, and acknowledges limitations.

Traditional A* assumes that all transitions between non-solid cells are executable, ignoring gravity and momentum. This generates false positives: geometrically valid but impossible paths. Our method addresses this by expanding the state to include kinematics and validating each transition with physical filters. The gain in precision (3–13× more valid paths) has two fundamental costs:

- **Combinatorial expansion**: each cell can originate multiple states (different

movements, origins, velocities), explaining the 10–35× higher memory consumption (median).
- **Per-node overhead**: each neighbor requires solving quadratic equations, line tracing, and collision checks, justifying the 21–97× time difference (median).

[Athaillah et al. 2025] proposed *CheckJump*, which builds a connectivity graph via *world mapping* (pre-computing jumps between ledges). The central difference is when the cost is incurred:

- **CheckJump**: upfront cost of 36–146 ms, then fast queries. Advantageous for static maps with many queries.
- **Physics-Aware A***: minimal upfront cost (geometric agent-body collision grid, independent of physics parameters; 3.4–37.6 ms in our experiments, Table 1), with each search paying for physical validation on demand. Advantageous for one-shot queries or dynamic environments (e.g., *power-ups* that alter physics).

Both approaches share limitations (geometric approximations, empirical parameters). A direct empirical comparison on the same benchmarks is left as future work.

5.1. Practical viability

Despite the costs, viable scenarios exist. **(1) Offline pre-computation**: running the algorithm during development or level loading (tens to hundreds of ms per pair) is acceptable. **(2) Procedural level validation**: checking reachability of areas without real-time navigation. **(3) Individual NPCs**: latencies of hundreds of ms are tolerable for one or a few agents, especially when run in the background.

Challenging scenarios (multiple real-time agents, maps larger than 200×200 cells) will require optimizations such as caching, parallelization, or more informed heuristics.

5.2. Limitations

We acknowledge the following limitations:

1. **Physical approximations**: parabolic trajectory approximated by two linear segments; collision detection using Bresenham (does not capture sub-cell collisions); agent-body collision grid is precomputed once per map/agent-size combination (independent of physics parameters such as gravity and jump velocity).
2. **Simplified modeling**: binary *grid* excludes one-way collisions, moving platforms, *hazards*. Advanced mechanics (double jump, *dash*, *wall jump*) are not supported, though the modular filter architecture allows extensions.
3. **Indirect validation**: effectiveness metrics use the same functions as the filters, consistent but not definitive proof of traversability. Manual tests on selected cases were performed, but full autonomous navigation was not implemented. Therefore, false positives may exist, despite the conservative design.
4. **False negatives**: conservatism may reject executable paths, reducing the coverage of viable solutions.

6. Conclusion

This work developed a pathfinding algorithm for 2D platform games that finds traversable paths while respecting physical constraints through incremental validation during the

search, a problem lacking adequate solutions in the literature. The main contributions are: **augmented states** with state $s = \langle (x, y), \text{state}, (x_g, y_g), \mathbf{v}_0 \rangle$ capturing the kinematic context necessary to validate transitions; a **modular sequential filter architecture**: a validation chain composed of universal filters, movement-state-specific filters, and post-processing, allowing flexible composition and extension to new mechanics; **integration with A* without pre-processing**, where neighbor generation is submitted to the filter chain and the Euclidean heuristic remains admissible and optimal within the physically feasible state space; and an **empirical evaluation** on four classic maps with two granularities, showing that the proposed algorithm finds 3–13 times more physically valid paths than traditional A*, albeit with 10–35$\times$ more memory and 21–97$\times$ more time (median per configuration).

The experiments also reveal that traditional A* may be sensitive to grid discretization and accepts many geometrically short but impossible paths. Our approach produces largely stable results across different granularities because its validation reflects the intrinsic physical properties of the level.

Several directions can extend this work: performance optimizations (partial pre-computation of jumps between ledges, combining incremental and CheckJump approaches; reachability caching; parallelization of the filter chain; more informed heuristics, such as estimating remaining jumps based on vertical distance); rigorous validation by implementing full autonomous navigation to empirically measure false positive and negative rates; advanced mechanics, adding filters for double jump, dash, wall jump, and wall slide, with the modular architecture facilitating these extensions; dynamic environments, integrating moving platforms (including cycle phase in the state), a scalability challenge; and an empirical comparison with CheckJump, implementing both approaches in the same environment to identify the crossover point where pre-computation becomes more advantageous.

The proposed algorithm offers a viable solution to a practical problem: finding paths that respect physics in platform games, where traditional A* systematically fails. Although the computational costs are high, they are tolerable in typical scenarios (pre-computation, level validation, individual NPCs) and can be mitigated with targeted optimizations. The modular nature of the filter architecture provides a solid foundation for both practical applications and future research on motion planning in environments with complex dynamics.

Declaration of Generative AI and AI-assisted technologies

We used LLMs such as DeepSeek and ChatGPT to improve the article's overall quality. We take full responsibility for all content produced with AI assistance.

References

Athaillah, I., Kholil, M., Akhsani, R., Ismanto, I., Waspada, H. P., e Muluk, M. S. (2025). Checkjump: An approach to real-time pathfinding for 2d grid-based platformer games. In *AIP Conference Proceedings*, volume 3250, pages 020008–1–020008–8.

Eberly, D. H. (2003). *Game Physics*. Morgan Kaufmann, Burlington.

Harabor, D. D. e Grana, A. (2011). Online graph pruning for pathfinding on grid maps. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 25, pages 1114–1119.

Hart, P. E., Nilsson, N. J., e Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107.

Iskandar, U. A. S., Diah, N. M., e Ismail, M. (2020). Identifying artificial intelligence pathfinding algorithms for platformer games. In *2020 IEEE International Conference on Automatic Control and Intelligent Systems (I2CACIS)*, Shah Alam. IEEE.

Millington, I. e Funge, J. (2016). *Artificial Intelligence for Games*. CRC Press, Burlington, 3 edition.

Moghadam, S. K., Ebrahimi, M., e Harabor, D. D. (2024). Guards: Benchmarks for weighted grid-based pathfinding. *Expert Systems with Applications*, 247:123719.

Russell, S. J. e Norvig, P. (2021). *Artificial Intelligence: A Modern Approach*. Pearson, Upper Saddle River, 4 edition.

Sturtevant, N. R. (2012). Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):144–148.

Tremblay, J., Borodovski, A., e Verbrugge, C. (2014). I can jump! exploring search algorithms for simulating platformer players. In *AIIDE Workshop: Artificial Intelligence in the Game Design Process*.

A. Supplementary Material

Data from all levels' execution, screenshots, and other supplementary material can be found at our OSF project's page [3].

[3] `https://osf.io/mcpb4/`