

Optimal Heuristic Selection for A* Pathfinding in Variable-Cost Grid Environments

Darius-Ioan Vlășan

Technical University of
Cluj-Napoca
Cluj-Napoca, Romania

vlasan.na.darius@student.utcluj.ro

Sebastian Stăvar

Technical University of
Cluj-Napoca
Cluj-Napoca, Romania

sebastianstavar02@gmail.com

Dorian Gorgan

Technical University of
Cluj-Napoca
Cluj-Napoca, Romania

dorian.gorgan@cs.utcluj.ro

ABSTRACT

This paper performs an analysis and experimental evaluation of the A* (A-Star) navigation algorithm implemented in the Unity development environment using the C# language. The study focuses on the comparative analysis of three fundamental heuristic functions (Manhattan, Euclidean, and Diagonal) in grid-discretized environments, evaluating their impact on computational efficiency and the quality of the generated path. The experiments were conducted on maps of different sizes (15×17 and 50×50) under three cost scenarios: clear terrain, dense central penalty, and dispersed movement penalties (swamp). The experimental results demonstrate that the Diagonal heuristic provides an optimal balance between execution time and the number of nodes explored on grids with 8 movement directions, guaranteeing path optimality. In contrast, it was empirically demonstrated that the Manhattan heuristic becomes inadmissible in this context; although it significantly reduces the explored state space by up to 45% and execution time by approximately 32%, it generates suboptimal paths. Furthermore, introducing complex penalty zones causes severe runtime inversions, where Manhattan's node-count advantage fails to translate into faster execution. The paper also highlights the algorithm's ability to dynamically adapt to variable-cost environments, achieving an efficient compromise between the geometric length of the path and the total movement penalty.

AUTHOR KEYWORDS

A-Star Pathfinding; Heuristic Functions; Grid Navigation; Variable-Cost Environments; Artificial Intelligence in Games; Unity 3D

DOI: 10.37789/icusi.2026.29

INTRODUCTION

Domain Context and the Importance of Navigation in Games

In modern video game development, Artificial Intelligence (AI) plays a fundamental role in creating immersive and interactive experiences for users [9]. Among the various AI technologies applied in this space (navigation, perception, decision, group collaboration, and optimization), the navigation of autonomous agents represents one of the most complex and intensely studied problems.

In a dynamic virtual environment, an agent's ability to determine an optimal path from a starting point to a specified destination involves not only avoiding static geometric obstacles but also adapting to terrain properties, such as areas with variable travel costs. The A* (A-Star) algorithm remains the widely used industry standard for solving pathfinding

problems on grid-like structures, due to its mathematical efficiency and the flexibility offered through the use of heuristic functions [5].

Defining the Theme and Problematic

The theme of this paper consists of the analysis and experimental evaluation of artificial intelligence techniques in computer games. Focusing on the technological pillar of navigation, the study specifically addresses the behavior of the A* algorithm in grid-discretized environments.

The central issue analyzed is how the choice of the heuristic function directly influences the performance of the algorithm in two distinct macrostructural contexts: small-scale maps and large-scale maps. Additionally, the investigation extends the classical analysis by introducing zones with variable movement penalties, forcing the algorithm to balance the optimization of geometric distance with the optimization of the total traversal cost.

Objectives and Intentions of the Study

The main objective of the paper highlights the close correlation between the size of the game space, the movement restrictions of agents, and the chosen heuristic function.

The research intentions aim to collect precise performance data by instrumenting the source code developed in the Unity workspace. To validate the formulated hypotheses, the evaluation of the experiments will be done according to metrics proposed in the paper:

- **Computational execution time:** accurately measured in milliseconds through *warm-up* techniques to isolate the algorithm's behavior from initial compilation effects.
- **Explored state space volume:** determined by counting the total number of nodes added to the closed list before reaching the target.
- **Generated solution quality:** reflected through the comparative analysis between the physical length of the final path and the total accumulated cost depending on the applied terrain penalties.

Through these measurements, the paper aims to empirically highlight advanced theoretical phenomena, analyzing the trade-off between reducing computation time and the potential loss of path optimality in the case of specific heuristics.

Structure of the Paper

The paper is structured into six main chapters. Introduction establishes the general framework of the research, defining the domain context, theme, objectives, and intentions. Other Achievements reviews the current state of technology in the field, exploring other languages, technologies, experiments,

or applications related to the paper's theme. Theoretical Considerations defines the mathematical and architectural foundations of the algorithm. Technological Considerations describes the development ecosystem used, presenting the chosen technologies, workspaces, and programming languages. Experimental Evaluation represents the empirical core of the paper, focusing on the description and execution of the experiments within the project. Conclusions synthesizes the obtained results, offering critical and comparative assessments of the demonstrated concepts and performances.

RELATED WORKS

Alternative Technologies and Development Environments in Game Navigation

While the current paper uses the Unity ecosystem and the C# language, the video game industry and the academic research community employ a wide range of other languages, development environments, and tools related to the theme of determining optimal paths.

C++ and Custom Engines / Unreal Engine. C++ remains the standard for high-performance (AAA) games due to direct control over memory management and microprocessor-level execution. The Unreal Engine uses native navigation systems based on three-dimensional spatial recurrence and optimized data structures, providing a robust alternative to discrete grid-based implementations.

Python in Academic Research. For academic simulations and rapid artificial intelligence prototyping, Python is frequently used in conjunction with libraries like Pygame or Reinforcement Learning environments, such as OpenAI Gym. Although the raw computational performance is inferior to C# or C++, Python excels in integrating neural network algorithms with classical pathfinding systems.

Alternative Approaches and Pathfinding Algorithms

The paper's theme, centered on the experimental analysis of artificial intelligence techniques, requires contextualization in relation to other algorithmic solutions developed for navigation.

Dijkstra's Algorithm [2]. It represents the theoretical foundation upon which the A* algorithm was built. Unlike A*, Dijkstra does not use a heuristic function to guide the search, exploring the state space uniformly in all directions. Although it guarantees the shortest path in graphs with arbitrary costs (including mud or swamp areas), the number of evaluated nodes is significantly higher, making it inefficient for large maps in real-time.

Greedy Best-First Search. This algorithm exclusively uses the heuristic component $h(n)$ to select the next node, ignoring the historical accumulated cost $g(n)$. It is extremely fast because it heads directly towards the target, but it does not guarantee path optimality and often fails in the presence of complex obstacles (such as U-shaped traps), where it can get stuck in local minima.

Hierarchical Pathfinding (HPA) [1].* For massive maps, the standard single-level A* approach becomes computationally prohibitive. HPA* abstracts the grid into macro-regions

(groups of nodes), first calculating a coarse global path between these regions, followed by local refinements. This reduces search complexity by orders of magnitude, sacrificing a tiny percentage of the absolute path optimality.

Software Solutions and Integrated Industry Tools

In practical development, programmers often rely on mature tools to manage navigation.

Unity NavMesh System. It is the native solution in the Unity engine, based on generating a polygonal navigation mesh (*Navigation Mesh*) instead of a discrete grid. NavMesh allows agents to navigate complex three-dimensional surfaces and offers integrated support for dynamic obstacle avoidance (through *Reciprocal Velocity Obstacles* – RVO) [11], as well as defining specific costs for different surface types, similar to the movement penalty logic analyzed in this study.

A Pathfinding Project (Aron Granberg) [3].* It is one of the most popular commercial plugins for Unity, offering highly optimized implementations of grids, NavMesh graphs, and point graphs. It utilizes advanced memory optimization techniques, such as asynchronous execution and multithreading processing.

Research Collectives and Reference Projects

The scientific study of pathfinding algorithms is supported by academic collectives and dedicated projects that provide testing metrics and standards in the field.

Moving AI Lab (University of Denver). Led by Nathan Sturtevant [10], this research collective maintains one of the largest public databases of benchmark maps derived from real games, such as *Dragon Age: Origins* or *StarCraft*. The platform allows researchers to experimentally evaluate their algorithms on thousands of standardized scenarios, analyzing metrics similar to those used in this paper: execution time and the number of expanded nodes.

OpenRA Project [6]. OpenRA is an open-source reimplementation of classic Real-Time Strategy (RTS) games, where navigation management for hundreds of simultaneous units is a critical problem. The developer community tests and implements optimized variants of hierarchical A* and group collision avoidance systems, providing a relevant case study on the scalability of pathfinding algorithms on large-scale maps.

Positioning of the Present Study. While commercial tools like Unity NavMesh and A* Pathfinding Project offer highly optimized black-box solutions, and platforms like Moving AI provide extensive benchmarking maps, this study focuses on isolating and validating the raw algorithmic behavior of fundamental heuristics in a controlled, custom Unity grid. This bridges the gap between textbook theory and practical engine implementation.

THEORETICAL CONSIDERATIONS

This chapter defines the concepts, notions, architectures, algorithms, and mathematical models underlying the analyzed navigation mechanisms.

Fundamental Concepts in Grid Navigation

Navigation (or pathfinding) is the process of determining a continuous and valid route between a starting point and a destination within a virtual environment. In the context of this study, the game space is discretized as a two-dimensional grid composed of individual nodes.

Each node encapsulates spatial and cost properties, defined as follows:

- **Walkable State:** A boolean value indicating whether the node represents free space or a collidable obstacle.
- **Grid Position:** The discrete Cartesian coordinates (x, y) of the node within the matrix.
- **Movement Penalty:** An additional cost added when traversing the node, used to simulate difficult terrain types (e.g., swamp).

A* (A-Star) Algorithm Architecture

The A* algorithm is a graph search technique that determines the minimum-cost path between a start node and a target node. Its efficiency stems from combining two metrics to evaluate the order in which adjacent nodes are explored.

For any evaluated node n during the search, the total estimated cost is calculated using the fundamental function:

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

Where the equation components are:

- $g(n)$: represents the exact accumulated cost to reach from the start node to node n . This calculation incorporates the physical distance traveled and penalties specific to the traversed terrain.
- $h(n)$: represents the estimated heuristic cost from node n to the destination node.
- $f(n)$: represents the total evaluation function. The A* algorithm will always select from the open list the node with the lowest value of $f(n)$ to be explored in the next step.

When two nodes in the open list share the same $f(n)$ value, a tie-breaking mechanism is required. In this implementation, ties are resolved by the one added last.

Heuristic Functions and Admissibility

A heuristic [8] is considered admissible if it never overestimates the actual cost to reach the destination, meaning:

$$h(n) \leq C^*(n) \quad (2)$$

where $C^*(n)$ represents the minimum actual cost to the destination. If the heuristic is admissible, the A* algorithm mathematically guarantees finding the optimal path.

To model movement in two-dimensional space, the base costs used in implementation are scaled by a factor of 10 to avoid expensive calculations with floating-point numbers. Thus, a horizontal or vertical movement has a cost of 10, and a diagonal movement has an approximate cost of 14, because:

$$\sqrt{10^2 + 10^2} \approx 14.14 \quad (3)$$

For the mathematical definition of the used heuristics, let the current node be A with coordinates (x_A, y_A) and the destination node B with coordinates (x_B, y_B) . The absolute coordinate differences are defined as:

$$\Delta x = |x_A - x_B| \quad (4)$$

$$\Delta y = |y_A - y_B| \quad (5)$$

During the experimental study, three heuristic functions were evaluated [7].

Manhattan Heuristic. This heuristic calculates the distance assuming the agent can move exclusively in the four orthogonal directions: up, down, left, and right. The mathematical model is:

$$h_{\text{Manhattan}}(A, B) = 10 \cdot (\Delta x + \Delta y) \quad (6)$$

Theoretical Analysis. In a grid navigation system with eight movement directions, the Manhattan heuristic overestimates the real cost of the route, thus becoming an inadmissible heuristic. This forces the algorithm to prioritize search speed to the detriment of guaranteeing the optimal path.

Euclidean Heuristic. It calculates the straight-line distance between two points based on the Pythagorean theorem. The implemented mathematical model uses rounding to the nearest integer for computational optimization:

$$h_{\text{Euclidean}}(A, B) = \left\lceil 10 \cdot \sqrt{\Delta x^2 + \Delta y^2} \right\rceil \quad (7)$$

Theoretical Analysis. This heuristic is admissible on a grid with eight movement directions, but calculating the square root for each expanded node introduces additional processing overhead on the central processing unit (CPU).

Diagonal Heuristic (Octile Metric). This heuristic function is specifically optimized for grids allowing movement in eight directions. The algorithm assumes diagonal movement (cost 14) as long as possible along the shorter component, followed by linear movement (cost 10) for the remaining distance.

Let:

$$\Delta_{\min} = \min(\Delta x, \Delta y) \quad (8)$$

and

$$\Delta_{\max} = \max(\Delta x, \Delta y) \quad (9)$$

The mathematical model becomes:

$$h_{\text{Diagonal}}(A, B) = 14 \cdot \Delta_{\min} + 10 \cdot (\Delta_{\max} - \Delta_{\min}) \quad (10)$$

Theoretical Analysis. This function represents one of the best approximations of actual distance on a discrete two-dimensional grid with eight movement directions. It offers an excellent balance between maintaining admissibility and reducing the number of expanded nodes, without requiring costly mathematical operations like square root extraction.

TECHNOLOGICAL CONSIDERATIONS

This chapter details the technological ecosystem used for implementing the navigation system and executing the experimental benchmarks. The workspace, the chosen programming language, and the software architecture designed for component modularization are presented.

Workspace: Unity Graphic Engine

The experimental system was integrated and developed within the Unity 3D graphics engine, a platform widely used in the video game industry for its advanced physical simulation and real-time rendering capabilities. Unity provides a Component-Based Architecture, extended in the project by inheriting the base class *MonoBehaviour*.

Using the Unity engine offered major technological advantages for the experimental study:

Layer Masks System. It allows the efficient identification of geometric and cost properties of the terrain in real-time. Through the variables *obstacleMask* and *swampMask*, the system filters collisions and dynamically determines the type of traversed node.

Physics Programming Interface (Raycasting and Volumetrics). The native function *Physics.CheckSphere* is used in the initialization phase to perform volumetric intersection tests in three-dimensional space, mapping the presence of obstacles or cost zones based on coordinates in the virtual world.

Graphical Visualization Subsystem (Gizmos). The *OnDraw-Gizmos* method allows direct rendering in the Unity editor window of the internal state of the grid. This facility provides critical visual support for validating the algorithm, differentially coloring traversable nodes (white), obstacles (black), closed state space (*closedListVisual* – red), and the resulting optimal path (*finalPath* – green).

Programming Language: C# and Performance Correlation

All calculation modules, data structures, and algorithms were implemented in the C# language. C# is an object-oriented, strongly typed language managed through a virtual machine with JIT (*Just-In-Time*) compilation.

In the context of a scientific paper focused on experimental evaluation of execution time, the characteristics of the C# language impose strict technical considerations.

Memory Management and Garbage Collection (GC). Because the A* algorithm uses dynamic data structures re-instantiated at each call (open lists and closed sets), memory management becomes a critical factor. Repeated memory allocation on the heap can trigger the automatic Garbage Collection mechanism, introducing artificial latencies in measurements [4].

To eliminate this impact and ensure the scientific validity of the benchmarks, the experimental architecture forces controlled memory cleanup and the collection of pending finalizers through the *System.GC.Collect()* and *System.GC.WaitForPendingFinalizers()* instructions before initiating each actual measurement.

JIT (Just-In-Time) Compilation. Upon the first run of the *CalculatePath* method, the virtual machine translates the intermediate code into machine instructions specific to the processor, causing a significant latency spike. To isolate pure algorithmic performance from this technological penalty, a preliminary warm-up phase was implemented, running the algorithm once without recording experimental data. Thus, subsequent measurements accurately reflect the effective computational processing time.

Software Architecture and Component Modularization

The system was designed following the single responsibility principle, being decomposed into five independent classes that interact modularly.

Node Class. Represents the fundamental data model of a discretized space unit. It is a class intended to store the information necessary for navigation and contains the traversability state (*walkable*), the absolute position in the 3D world (*worldPos*), the matrix index (*gridX*, *gridY*), specific costs *g* and *h*, the terrain penalty (*movementPenalty*), and a reference to the

parent node (*parent*) used for path reconstruction. The calculated property *fCost* dynamically returns the sum of the costs associated with a node, optimizing the evaluations performed by the A* algorithm.

GridManager Class. It is responsible for abstracting and managing the two-dimensional space. In the initialization phase (*Awake*), it calculates the discrete dimensions of the grid based on the requested total size (*gridWorldSize*) and the radius of a node (*nodeRadius*).

The *CreateGrid* method instantiates the two-dimensional node matrix, evaluates the presence of obstacles, and applies the *swampPenalty*. The class also provides spatial query functionalities, such as getting adjacent neighbors (*GetNeighbors*) and converting a position from the continuous 3D world into a discrete node (*NodeFromWorldPoint*).

Pathfinding Class. Exclusively encapsulates the mathematical logic of the A* algorithm. It is completely decoupled from the physical control of the agent and focuses on managing the search lists (*openList* and *closedList*), as well as implementing the switch-case decision structure for the three types of heuristics defined in the *HeuristicType* enumeration: Manhattan, Euclidean, and Diagonal.

PathFollower Class. Manages the physical movement of the agent along the resulting path. The implementation uses specific Unity asynchronous routines (*Coroutine*), allowing the agent's position to be updated frame by frame via the *yield return null* instruction. The movement is achieved through linear interpolation using the *Vector3.MoveTowards* function, and the orientation is updated gradually through spherical interpolation using *Quaternion.Slerp*. This approach avoids blocking the main execution thread of the application.

AgentController Class. Represents the central point of coordination and orchestration of the experiments. It holds references to all system components and exposes the *MoveToTarget* and *RunExperiments* methods. The latter automates the scientific testing process by sequentially and controllably executing the three heuristics on the same map configuration. In this way, the uniformity of the experimental environment and the direct comparability of the obtained results are guaranteed.

EXPERIMENTAL EVALUATION

This chapter describes and executes the planned experiments within the project, evaluating the results according to the proposed metrics to validate the behavior of the applied artificial intelligence techniques.

Hypothesis, Assumption, and Prediction

The structure of the experiment starts from the following fundamental hypotheses:

- (1) **Admissibility Hypothesis:** We assume that the Manhattan heuristic, being inadmissible on a grid with movement in eight directions, will run the fastest and explore the fewest nodes, but will generate a suboptimal path compared to the Euclidean and Diagonal heuristics.
- (2) **Scalability Hypothesis:** On small maps, the performance differences between heuristics will be negligible, but they will grow significantly as the size of the search space expands.

- (3) **Cost Adaptability Hypothesis:** We assume that the presence of extensive zones with high penalties in the center of the map will force the algorithm to choose detour routes, with a greater spatial length, while dispersed penalties will cause the algorithm to partially traverse difficult zones to minimize the total movement cost.

Experiment Description: Tasks and Scenarios

The experiment consists of navigating an autonomous agent from a start point to a fixed destination.

Two main testing environments were created, each divided into three distinct scenarios:

- **Environment A (Small Map: 15×17)**
 - Scenario 1: No cost zones, Figure 1;
 - Scenario 2: Dense cost zone in the center of the map, Figure 2;
 - Scenario 3: Randomly dispersed cost zones, Figure 3.

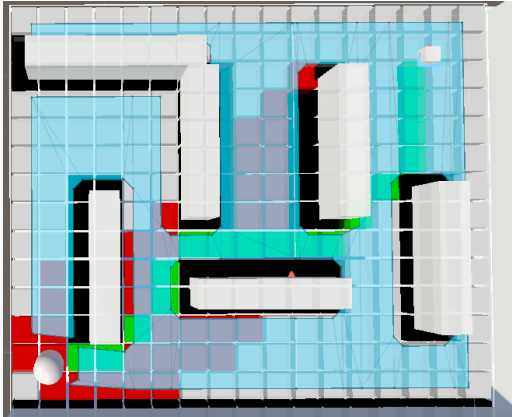


Figure 1: Map A without cost zones

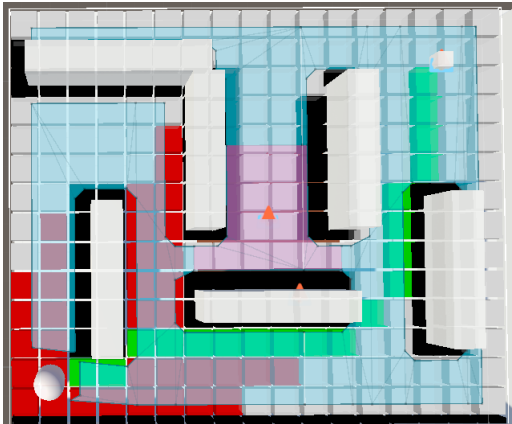


Figure 2: Map A with dense cost zone in the center of the map

- **Environment B (Large Map: 50×50)**
 - Scenario 1: No cost zones, Figure 4;
 - Scenario 2: Dense cost zone in the center of the map, Figure 5;
 - Scenario 3: Randomly dispersed cost zones, Figure 6.

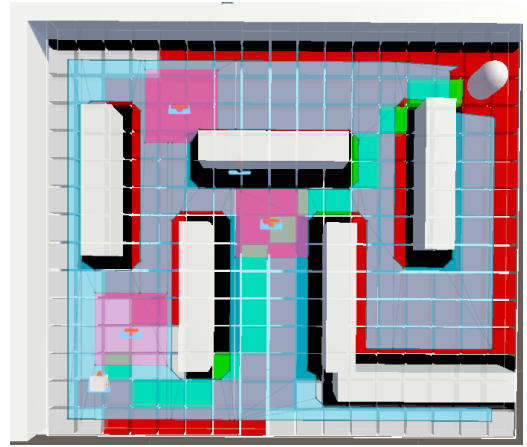


Figure 3: Map A with randomly dispersed cost zones

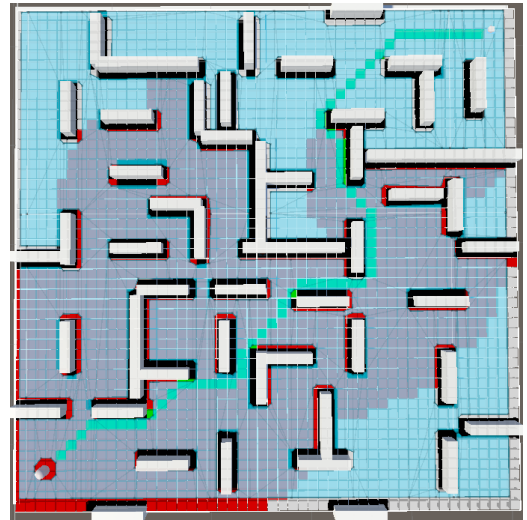


Figure 4: Map B without cost zones

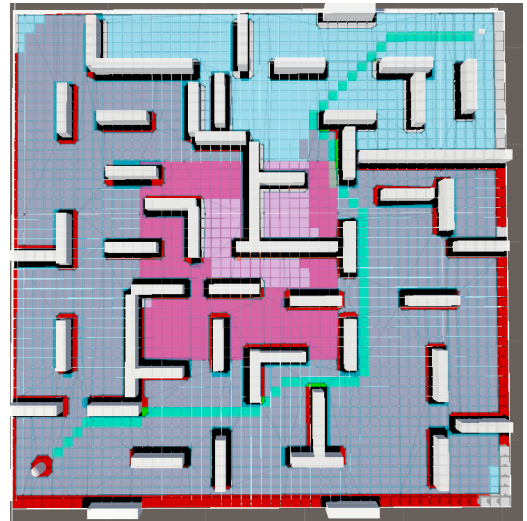


Figure 5: Map B with dense cost zone in the center of the map

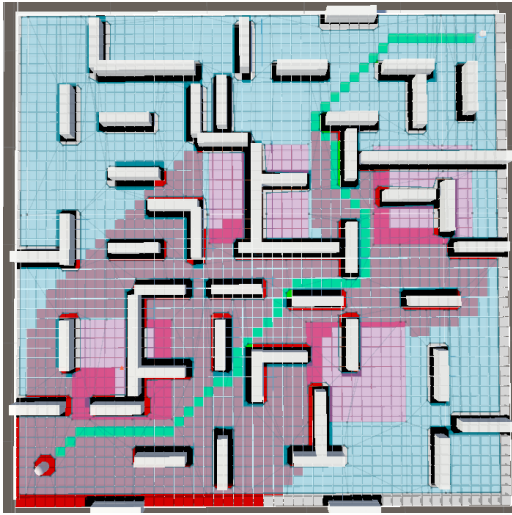


Figure 6: Map B with randomly dispersed cost zones

Experiment Participants

Because the paper analyzes the performance of artificial intelligence techniques, the participants in the experiment are not human subjects, but autonomous virtual agents instantiated in the Unity environment and controlled by the *AgentController* component.

System Configuration

To ensure deterministic evaluation, the measurements were run on the same computing system, using the Unity 3D editor. Data collection was carried out sequentially, including automated procedures for collecting unused memory (Garbage Collection) and warm-up cycles for JIT compilation, guaranteeing a controlled and reproducible execution environment. Each scenario configuration was executed exactly ten times.

Experiment Variables

The following categories of variables were defined within the experimental design.

Independent Variables.

- Heuristic function type (Manhattan, Euclidean, Diagonal);
- Grid size (15×17 and 50×50 , respectively);
- Penalty zone topology (no additional cost, central blockage, random dispersion).

Measurable Variables.

- **Execution time:** measured in milliseconds (ms);
- **Number of explored nodes:** final size of the closed list;
- **Path length:** total number of nodes in the final route;
- **Total path cost:** sum of movement costs and penalties associated with the resulting route.

Analyzed Measurement Data

The raw data were collected directly from the console and centralized for comparative analysis.

Environment A: Small Map (15×17). On a map with reduced dimensions, the algorithmic complexity remains low (Table 1).

Table 1: Experimental Results for Environment A

Scenario	Heuristic	Time (ms)	Nodes	Length	Cost
1	Manhattan	0.0602	31	12	120
1	Euclidean	0.0524	28	12	120
1	Diagonal	0.0408	20	12	120
2	Manhattan	0.1646	88	23	230
2	Euclidean	0.1729	96	23	230
2	Diagonal	0.1705	94	23	230
3	Manhattan	0.1250	71	12	210
3	Euclidean	0.1193	73	12	210
3	Diagonal	0.1253	73	12	210

Interpretation. In Scenario 1, the execution times are very low for all heuristics, with differences observable mainly in the number of explored nodes. The Diagonal heuristic explores only 20 nodes, demonstrating the best efficiency. Contrary to expectations, in this scenario, Manhattan recorded the longest execution time and expanded the most nodes. At this micro-scale, algorithmic advantages are overshadowed by measurement noise.

In Scenario 2, the presence of a central zone with a high cost causes the agents to choose a detour route. The path length increases to 23 nodes, but the total cost remains optimal.

In Scenario 3, the agents partially traverse the penalized zones. The path length remains unchanged, but the total cost increases from 120 to 210 units.

Environment B: Large Map (50×50). On an extended map, the differences between the heuristics become much more evident (Table 2).

Table 2: Experimental Results for Environment B

Scenario	Heuristic	Time (ms)	Nodes	Length	Cost
1	Manhattan	1.4295	600	68	680
1	Euclidean	2.3308	1229	67	670
1	Diagonal	2.1117	1086	67	670
2	Manhattan	3.4440	1417	74	890
2	Euclidean	2.9427	1513	74	890
2	Diagonal	2.9547	1472	74	890
3	Manhattan	1.0278	370	67	670
3	Euclidean	2.1437	942	67	670
3	Diagonal	1.7184	775	67	670

Interpretation. Scenario 1 clearly highlights the difference between an inadmissible and an admissible heuristic. The Manhattan heuristic explores only 600 nodes and achieves an execution time of approximately 1.43 ms, but produces a path of length 68. The Euclidean and Diagonal heuristics guarantee the optimal path, with a length of 67, but require exploring a much larger number of nodes and roughly double the execution time.

In Scenario 2, all heuristics are forced to bypass the central penalized zone, causing the path length to increase to 74 nodes. In this scenario, a runtime inversion occurs: Manhattan expands fewer nodes (1417) than the others but records the longest execution time (3.4440 ms). This contradicts the node-count-drives-runtime assumption and highlights the impact of cache state and memory allocation behavior during execution. In Scenario 3, the dispersed distribution of penalties does not significantly alter the shape of the optimal route, but

performance differences between heuristics remain visible in execution time and the number of explored nodes.

Conclusions on Hypotheses

The analysis of experimental data confirms the initially formulated hypotheses.

- (1) **Validating the Admissibility Hypothesis:** This hypothesis was only partially confirmed. The expected suboptimality of the Manhattan heuristic was observed in only one of the six configurations (Environment B, Scenario 1). In the other five scenarios, it returned paths identical in cost to the admissible heuristics. Furthermore, its theoretical advantage in node expansion did not strictly correlate with faster execution times across all tests, due to runtime environmental factors.
- (2) **Validating the Scalability Hypothesis:** On the 50×50 exploration surface, the average execution time increased approximately nineteen-fold compared to the 15×17 environment, demonstrating the strong dependence between search space size and computational cost.
- (3) **Validating the Cost Adaptability Hypothesis:** The implemented algorithm simultaneously optimizes distance and traversal cost. In scenarios with high central penalties, it prefers longer detour routes, avoiding traversal costs that would exceed the advantage of a geometrically shorter path.

CONCLUSIONS

Following the bibliographic study and the experimental evaluation of artificial intelligence techniques applied to computer game navigation, the present paper highlighted the ability of the A* algorithm to solve the optimal pathfinding problem in dynamic, two-dimensional environments. The analysis of the measured data allows the formulation of clear critical and comparative assessments regarding the performance of heuristic functions and the cost adaptation behavior.

First, the study practically demonstrated the inadmissibility phenomenon of the Manhattan heuristic on a grid allowing 8-direction movement. While in clear, large-scale terrain (Environment B, Scenario 1) it explored up to 45% fewer nodes and offered a superior execution time, it compromised the theoretical functionality of the A* algorithm by generating a suboptimal path (length 68 compared to the optimum 67). Furthermore, the data revealed that Manhattan's theoretical advantage is highly unstable: in environments with dense blockages, it suffered from runtime inversions, taking longer to execute despite expanding fewer nodes. Thus, while occasionally performant, the Manhattan heuristic should be used exclusively in grids restricted to 4 orthogonal movement directions, as its behavior in 8-way grids is unpredictable and often suboptimal.

Second, the comparative performance evaluation highlights the superiority of the Diagonal (Octile metric) heuristic for tile/grid-based games that allow diagonal movement. This offered the ideal computational balance, guaranteeing the optimal path, while consistently expanding fewer nodes than the Euclidean heuristic (1086 vs. 1229 in the obstacle-free environment), thereby reducing processor load without using costly mathematical operations (such as root extraction). From the perspective of functional characteristics, the experiments validated the robustness of the algorithm in managing variable cost zones (swamps/mud). The algorithm exhibited emergent resource optimization behavior: it chose to bypass dense blockages via longer spatial routes to avoid accumulating penalties (Scenario 2), but correctly calculated that directly crossing dispersed penalized zones (Scenario 3) is preferable to massive detouring. This compromise demonstrates that the spatial length of the path and the total movement cost are metrics that can evolve inversely proportionally, depending on the topography of the game level.

In conclusion, the performance of interactive navigation systems does not rely on a universally valid algorithmic solution, but on a close correlation between the size of the game space, the movement restrictions of agents, and the chosen heuristic function. The present study confirms that rigorous code instrumentation and empirical evaluation of work environments (Unity/C#) are mandatory steps for achieving an optimal balance between the quality of artificial intelligence and the millisecond budget available per frame in an interactive application.

ACKNOWLEDGMENTS

The research presented in this article has been funded by the national project 135PHE/2026, through the PN IV - European and International Cooperation Programme.

References

- [1] Adi Botea, Martin Müller, and Jonathan Schaeffer. 2004. Near optimal hierarchical path-finding (HPA*). *Journal of Game Development* 1 (01 2004).
- [2] Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1, 1 (1959), 269–271. doi:10.1007/BF01386390
- [3] Aron Granberg. [n. d.]. A* Pathfinding Project. <https://arongranberg.com/astar/>. Accessed: July 2026.
- [4] Jason Gregory. 2026. *Game Engine Architecture: Volume I, Foundations and Core Engine Systems* (4th ed.). CRC Press.
- [5] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107. doi:10.1109/TSSC.1968.300105
- [6] OpenRA Community. [n. d.]. OpenRA: A Libre/Free Real Time Strategy Game Engine. <https://www.openra.net/>. Accessed: July 2026.
- [7] Amit Patel. [n. d.]. Amit's Thoughts on Pathfinding. <http://theory.stanford.edu/~amitp/GameProgramming/>. Accessed: June 2026.
- [8] Judea Pearl. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.
- [9] Stuart Russell and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach* (4th. ed.). Pearson.
- [10] Nathan R. Sturtevant. 2012. Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 2 (2012), 144–148. doi:10.1109/TCIAIG.2012.2197681
- [11] Jur van den Berg, Ming Lin, and Dinesh Manocha. 2008. Reciprocal Velocity Obstacles for Real-Time Multi-agent Navigation. *ICRA*, 1928–1935. doi:10.1109/ROBOT.2008.4543489