Efficient Single-Pass Pathfinding in Flow Fields with Dynamic Costs*

*Optimizing Grid-Based Navigation with a Single-Pass Flow Field Algorithm

1st Ali Hakan Orhan
Computer Engineering Akdeniz University
Antalya, Turkey
orhanalihakan@gmail.com

2nd Hüseyin Orhan
Computer Engineering Akdeniz University
Antalya, Turkey
huseyin.orhan@yahoo.com

***Abstract*—Pathfinding in grid-based environments is a critical component of AI-driven navigation systems, especially in applications like video games, robotics, and logistics. Traditional flow field pathfinding methods struggle with varying movement costs, leading to inefficiencies due to redundant calculations. This paper proposes a novel single-pass flow field algorithm that processes each grid cell only once, significantly improving efficiency without sacrificing accuracy. By leveraging bucket sorting for cost management, the proposed method minimizes computational overhead and ensures optimal path generation. Experimental results demonstrate that this approach achieves efficiency levels comparable to unified cost pathfinding while effectively handling variable-cost environments. The findings provide a scalable, real-time solution for AI-driven navigation in dynamic and cost-variable settings.**

***Index Terms*—Flow field pathfinding, single-pass pathfinding, grid-based navigation, bucket sort in pathfinding, cost-efficient pathfinding, varying cost navigation, real-time strategy game AI, pathfinding optimization algorithms.**

I. Introduction

Pathfinding is a fundamental challenge in artificial intelligence (AI) applications, particularly in grid-based environments where movement costs can vary. Traditional flow field pathfinding algorithms precompute directional vectors for navigation but suffer from inefficiencies when dealing with grids that have non-uniform movement costs. In such cases, cells with higher costs are often recalculated multiple times, leading to excessive computation times and reduced real-time applicability.

This research addresses these inefficiencies by introducing a single-pass flow field algorithm that eliminates redundant calculations. The proposed method ensures each cell is processed only once while maintaining accuracy and computational efficiency. By integrating bucket sorting, the algorithm optimizes cost management, reducing processing overhead and enhancing pathfinding performance.

The primary objectives of this study are:

- To develop a single-pass flow field algorithm capable of handling varying movement costs efficiently.
- To optimize pathfinding performance through improved queue management using bucket sorting.
- To evaluate the computational efficiency of the proposed method in comparison to traditional approaches.

The scope of this research includes grid-based pathfinding applications such as real-time strategy (RTS) games, robotic navigation, and traffic optimization. However, the study is limited to static grids with predefined cost variations and does not account for dynamically changing environments. The results are expected to contribute significantly to the field by improving efficiency in AI-driven navigation systems.

II. Literature Review

Pathfinding in grid-based environments has been extensively studied, with numerous algorithms developed to optimize computational efficiency, accuracy, and real-time applicability. This section systematically reviews relevant studies, highlighting existing approaches, their limitations, and the need for a single-pass flow field pathfinding algorithm with varying movement costs.

A. Flow Field Pathfinding and Traditional Approaches

Flow field pathfinding is widely used in real-time applications such as strategy games and robotics due to its ability to precompute navigation vectors. However, traditional methods assume uniform movement costs, leading to inefficiencies in environments with cost variations [1].

Dijkstra's algorithm [2], one of the most fundamental shortest-path algorithms, forms the basis for many grid-based navigation techniques. Despite its optimality, it recalculates nodes excessively when applied to varying cost grids. Similarly, A* [3] optimizes Dijkstra's approach with heuristic guidance but suffers from performance degradation in large-scale pathfinding.

Jump Point Search (JPS) [4] enhances A* by pruning unnecessary nodes, reducing search space. However, its applicability to flow fields is limited, as it does not precompute directional vectors efficiently. Fringe-based expansion methods [5] provide alternatives by prioritizing specific regions, yet scalability remains an issue.

B. Handling Varying Costs in Pathfinding

Grid-based pathfinding with varying movement costs presents unique challenges. Research by Rabin [6] emphasizes the impact of cost variations on computational overhead in game AI. Algorithms such as Lifelong Planning A* (LPA*) [7] and D* Lite [8] adapt to changing costs dynamically but require repeated recalculations, making them less efficient for static grid applications.

Recent advancements introduce hierarchical pathfinding strategies [9] to mitigate computation time by segmenting the grid. However, these methods do not eliminate redundant calculations entirely, necessitating the development of a single-pass algorithm that optimally processes each cell once.

C. Optimizing Efficiency with Single-Pass Approaches

Single-pass computation techniques aim to process each cell only once, reducing redundant calculations. The bucket sort method, commonly used in sorting algorithms [10], provides an effective means of managing priority queues in pathfinding. Studies by Cui and Shi [11] demonstrate the benefits of bucket sorting in navigation but do not specifically address its integration into flow field pathfinding.

This research builds upon these concepts by incorporating a single-pass approach with bucket sort optimization, ensuring efficient navigation in varying cost environments. The next section details the theoretical framework underpinning this methodology.

III. Theoretical Framework

The development of a high-performance, single-pass flow field pathfinding algorithm is grounded in principles from graph theory, algorithmic complexity, and discretized cost modeling. This framework establishes the foundational concepts underpinning the three algorithmic variants implemented and evaluated in this study.

A. Graph-Based Pathfinding Foundations

Flow field pathfinding operates over a grid modeled as a weighted graph, where each cell is treated as a node and edges correspond to possible movements between neighboring cells, weighted by movement cost. Classical algorithms like Dijkstra's and A* apply priority queues to ensure lowest-cost-first traversal. This study extends this foundation by proposing an optimized alternative using modular bucket sorting, which maintains cost order with reduced overhead.

B. Single-Pass and Multi-Pass Approaches

Efficiency in pathfinding is strongly influenced by whether an algorithm revisits nodes. Dijkstra's algorithm and the proposed bucket-based method guarantee a *single-pass* traversal: each cell is evaluated exactly once, ensuring predictable performance and avoiding redundant updates. In contrast, basic queue-based approaches may reprocess cells if better paths are discovered later, classifying them as *multi-pass* and potentially leading to inefficiencies in high-cost or obstacle-rich environments.

C. Cost-Ordered Processing via Modular Buckets

The core theoretical advancement of this study is the use of a modular bucket array to sort and manage cell processing by discretized cost values. Instead of maintaining a dynamic priority queue, the algorithm maps cost values to fixed-size bucket indices using a modulus operation. This structure supports a circular traversal of cost layers while preserving processing order due to the non-decreasing nature of cost propagation.

This design ensures:

- Each cell is processed exactly once.
- Lower-cost cells are always processed before higher-cost ones.
- Memory usage remains constant, independent of maximum cost values.

The constraint of this approach is that cost values must be discrete. Continuous cost functions are not directly compatible, whereas Dijkstra's algorithm supports them without modification. However, for discrete or quantized environments, bucket-based sorting achieves near-linear time complexity with minimal overhead.

D. Comparative Theoretical Characteristics

- **Dijkstra-Based Flow Field:** Theoretically optimal in all cases, supports continuous costs, but incurs logarithmic overhead due to priority queue operations.
- **Queue-Based Flow Field:** Simple and fast in sparse, low-variation environments, but prone to inefficiencies in dense or dynamic scenarios due to reprocessing.
- **Bucket-Based Flow Field (Proposed):** Combines the best of both—single-pass efficiency like Dijkstra, and speed comparable to the queue-based method on sparse maps, with consistent performance across different terrain complexities.

Together, these principles form the theoretical backbone of the study's proposed pathfinding solution. The following methodology section details the practical implementation and empirical evaluation of these algorithms.

IV. Methodology

This section outlines the methodology employed to implement and evaluate the proposed flow field pathfinding algorithms. Three variants were developed and tested under different grid sizes and obstacle configurations to compare their efficiency and computational performance.

A. Implemented Algorithms

- **Dijkstra-Based Flow Field:** Uses a priority queue to expand cells in order of increasing cumulative cost. It processes each cell once and supports continuous cost values.
- **Queue-Based Flow Field:** Uses a simple FIFO queue. It performs well in sparse maps with limited cost changes but may reprocess cells multiple times in complex environments.

- **Bucket-Based Flow Field (Proposed):** Uses a fixed-size modular bucket array to store and process cells based on discretized cost values. Each cell is processed exactly once, and cost values wrap around the array using modulus. This method maintains sorted cost order and consistent performance, but requires discrete cost values with a defined minimum step size.

Each algorithm supports movement in both orthogonal and diagonal directions. Orthogonal movement incurs a cost of 10 units, while diagonal movement costs 14 units. These are combined with each cell's base terrain cost, which may vary depending on obstacle presence.

B. Grid Setup and Obstacle Configurations

All experiments were conducted on square grids of varying sizes: $(64, 64)$, $(128, 128)$, $(256, 256)$, $(512, 512)$, $(1024, 1024)$, $(2048, 2048)$, and $(4096, 4096)$. The player's starting point was always positioned at the center of the grid.

Three types of obstacle configurations were used to evaluate performance in different scenarios:

- **No Walls:** An open grid environment with no obstructions.
- **Concentric Walls:** Multiple square layers of walls centered around the player. This configuration deletes the corners of each ring wall, leading to worse time and memory performance in queue-based algorithms due to reduced pruning and longer propagation paths.
- **Random Walls:** Randomly placed wall segments simulating irregular and unpredictable terrain features.

Obstacle cells were assigned a maximum movement cost of 50 units to simulate impassable or high-effort terrain, while all other cells retained a base cost of zero.

C. Benchmarking Procedure

Each combination of algorithm, grid size, and wall configuration was tested in four independent trials. For every trial:

- The grid was initialized with the selected obstacle configuration.
- The flow field algorithm was executed from the player's starting cell.
- Execution time was measured from start to finish using system timestamps.
- Memory usage was recorded, along with the normalized metric of memory usage per cell.

D. Performance Metrics

The performance of each method was evaluated using the following metrics:

- **Per-Cell Processing Time (μs/cell):** A normalized metric representing the average time required to process each individual grid cell.
- **Per-Cell Memory Usage (byte/cell):** A normalized metric representing memory consumption per grid cell.

No graphical rendering, path reconstruction, or directional vector generation was included in the benchmarks. The evaluation focuses strictly on the computational and memory efficiency of each flow field algorithm in propagating cost values across the grid.

V. Results

The following results evaluate the three flow field algorithms—Dijkstra-based, queue-based, and the proposed bucket-based method—across various grid sizes and obstacle configurations. Performance was measured in terms of per-cell execution time and memory usage, reflecting the theoretical principles outlined earlier.

A. *No-Walls Configuration:* Baseline Performance

In open-grid environments with no obstacles, where traversal cost remains uniform and minimal, the queue-based flow field algorithm demonstrated the highest efficiency. This is consistent with its theoretical strength in sparse, low-variation environments, where reprocessing is rare and overhead is minimal.

TABLE I
AVERAGE EXECUTION TIME AND PER-CELL PROCESSING TIME (NO WALLS)

Grid Size	Dijkstra (μs/cell)	Bucket (μs/cell)	Queue (μs/cell)
64x64	0.154	0.072	**0.033**
128x128	0.124	0.052	**0.029**
256x256	0.130	0.043	**0.028**
512x512	0.144	0.046	**0.032**
1024x1024	0.152	0.055	**0.036**
2048x2048	0.167	0.065	**0.063**
4096x4096	0.181	**0.072**	0.080
8192x8192	0.195	**0.080**	0.081

TABLE II
MEMORY USAGE AND PER-CELL MEMORY (NO WALLS)

Grid Size	Dijkstra (B/cell)	Bucket (B/cell)	Queue (B/cell)
64x64	0.978	0.786	**0.488**
128x128	0.468	0.383	**0.247**
256x256	0.251	0.190	**0.124**
512x512	0.120	0.094	**0.062**
1024x1024	0.064	0.047	**0.031**
2048x2048	0.031	0.023	**0.016**
4096x4096	0.016	0.012	**0.008**
8192x8192	0.008	0.006	**0.004**

As shown in Tables I and II, the queue-based method achieved the lowest per-cell processing time across most grid sizes, outperforming even the bucket-based approach at small to medium scales. The bucket-based method remained competitive, with stable and predictable performance, validating its single-pass, cost-ordered processing advantage. Dijkstra's algorithm, although robust and optimal, was consistently slower due to the overhead of priority queue management.

Memory usage followed a similar trend: the queue-based method was the most memory-efficient, especially for smaller grids, while the bucket-based algorithm used moderate memory but scaled consistently. Dijkstra's algorithm used the most memory due to its dynamic data structures.

B. ***Concentric Walls Configuration:*** *Structured High-Cost Regions*

In this obstacle-rich configuration designed to test propagation efficiency under constrained paths, the bucket-based algorithm significantly outperformed both Dijkstra and queue-based methods in both runtime and memory usage. This aligns with its theoretical advantage in single-pass traversal and fixed-cost layer propagation.

TABLE III
AVERAGE EXECUTION TIME AND PER-CELL PROCESSING TIME (CONCENTRIC WALLS)

Grid Size	Dijkstra (μs/cell)	Bucket (μs/cell)	Queue (μs/cell)
64x64	0.171	0.146	**0.099**
128x128	0.159	**0.056**	0.174
256x256	0.162	**0.045**	0.326
512x512	0.176	**0.051**	0.632
1024x1024	0.190	**0.060**	1.248
2048x2048	0.204	**0.073**	3.868
4096x4096	0.231	**0.087**	9.469
8192x8192	0.301	**0.122**	–

TABLE IV
MEMORY USAGE AND PER-CELL MEMORY (CONCENTRIC WALLS)

Grid Size	Dijkstra (B/cell)	Bucket (B/cell)	Queue (B/cell)
64x64	3.799	2.732	**1.367**
128x128	2.108	1.498	**1.186**
256x256	1.098	**0.782**	1.093
512x512	0.557	**0.399**	1.047
1024x1024	0.282	**0.202**	1.023
2048x2048	0.142	**0.101**	1.012

As Tables III and IV illustrate, the queue-based method's multi-pass nature led to substantial reprocessing overhead, causing exponential slowdowns as grid size increased. At 2048×2048, the queue method was over 50 times slower than the bucket-based approach. Memory usage also ballooned due to excessive queue operations and redundant cell updates.

Conversely, the bucket-based algorithm maintained stable, linear performance, reflecting its ability to process each cell exactly once and maintain sorted order without revisits. Dijkstra's method remained more consistent than the queue-based variant, but still lagged behind the bucket implementation in both speed and memory.

C. ***50% Random Wall Configuration:*** *Unstructured, High-Entropy Terrain*

To assess robustness in realistic, irregular environments, a 50% wall density grid was used with randomized player placement. This scenario introduced dynamic propagation challenges with non-uniform terrain and highly variable path options.

In this test, the bucket-based algorithm delivered the best overall performance, particularly for medium to large grid sizes (Table V). Its fixed-size modular bucket structure scaled well under the unpredictable obstacle layout, maintaining single-pass efficiency and avoiding the bottlenecks faced by the queue-based method.

TABLE V
EXECUTION TIME AND PER-CELL PROCESSING TIME (50% RANDOM WALLS)

Grid Size	Dijkstra (μs/cell)	Bucket (μs/cell)	Queue (μs/cell)
64x64	0.326	0.138	**0.067**
128x128	0.169	**0.063**	0.091
256x256	0.185	**0.056**	0.095
512x512	0.183	**0.058**	0.128
1024x1024	0.201	**0.069**	0.190
2048x2048	0.203	**0.075**	0.635
4096x4096	0.249	**0.114**	1.482
8192x8192	0.297	**0.148**	–

TABLE VI
PER-CELL MEMORY USAGE (50% RANDOM WALLS)

Grid Size	Dijkstra (B/cell)	Bucket (B/cell)	Queue (B/cell)
64x64	5.767	4.062	**0.986**
128x128	3.279	2.307	**0.531**
256x256	1.583	1.119	**0.243**
512x512	0.736	0.519	**0.194**
1024x1024	0.376	0.265	**0.122**
2048x2048	0.225	0.159	**0.155**
4096x4096	0.090	**0.063**	0.129

The queue-based approach again performed well at smaller scales but degraded rapidly beyond 2048×2048, where redundant processing led to exponential time increases and inconsistent memory usage (Table VI). Dijkstra's method remained predictable but was consistently slower than the bucket-based algorithm.

D. *Summary of Algorithm Suitability*

The comparative performance of the three flow field algorithms across configurations supports the theoretical framework presented earlier:

- **Queue-Based Flow Field:** Best suited for small, sparse, and obstacle-free maps. Its simplicity offers the lowest runtime and memory usage in ideal conditions but breaks down in dense or dynamic environments due to multi-pass inefficiency.
- **Dijkstra-Based Flow Field:** The most robust and flexible, supporting continuous cost values. It performs consistently but incurs higher computational and memory costs from priority queue operations.
- **Bucket-Based Flow Field (Proposed):** Offers a balance of performance and scalability. Its single-pass design and cost-ordered modular buckets enable near-linear performance across all tested scenarios—especially effective in large, dense, or complex terrains with discrete cost values.

These results confirm the proposed bucket-based method as a practical and efficient alternative for real-time pathfinding in discretized environments where consistent and scalable performance is critical.

DISCUSSION

The empirical results validate the theoretical strengths of the proposed bucket-based flow field algorithm, particularly in scenarios with discrete cost values and complex terrain.

Compared to the classical Dijkstra-based and queue-based approaches, the bucket-based method consistently achieved superior or comparable performance in terms of per-cell execution time and memory efficiency, especially in larger grids and high-obstacle environments.

In alignment with the literature, Dijkstra's algorithm maintained theoretical optimality and robustness across all scenarios but incurred significant overhead due to its reliance on dynamic priority queues. This overhead became particularly evident in larger grid sizes, where the logarithmic complexity of queue operations compounded. Prior work has acknowledged this limitation, often citing Dijkstra's stability at the expense of speed—findings mirrored in this study.

The queue-based method, while efficient on small and sparse maps, performed poorly in obstacle-rich or large-scale scenarios. As expected, its multi-pass nature led to substantial reprocessing, which amplified both execution time and memory usage. This aligns with previous observations that simple FIFO queues lack mechanisms to prioritize lower-cost paths or prevent redundant updates, making them unsuitable for high-complexity terrains.

The proposed bucket-based algorithm bridges the gap between the two existing methods. Its fixed-size modular bucket array ensures single-pass processing with constant memory overhead, provided that cost values are discrete and bounded. Notably, it achieved the best performance in the concentric wall and random wall configurations—scenarios that are traditionally challenging for multi-pass or priority-based methods. These outcomes are consistent with theoretical predictions and highlight the algorithm's ability to maintain order-preserving cost propagation without dynamic data structures.

However, the method is not without limitations. It relies on the discretization of cost values, making it incompatible with continuous or floating-point cost functions unless these are quantized. Moreover, tuning the bucket size and modulus parameters may require domain-specific calibration to optimize memory use and prevent cost collisions. While this study did not explore adaptive bucket sizing or hybrid methods, such enhancements may extend the applicability of this algorithm to broader pathfinding problems.

Future Research Directions

To build upon the findings of this study, several avenues merit further exploration:

- **Support for Continuous Costs:** Future work could investigate hybrid schemes that integrate priority queues within modular bucket structures, allowing for support of continuous cost domains without sacrificing performance.
- **Dynamic Environments:** Real-time updates, moving obstacles, or cost re-evaluation during runtime were not addressed. Extending the bucket-based model to dynamic grids would be valuable in real-world applications like robotics or games.
- **Parallelization:** The modular and structured nature of the bucket-based method makes it a candidate for parallel processing on GPUs or multi-core systems, potentially unlocking further performance gains for extremely large-scale maps.
- **Directional Vector Generation:** While this study focused purely on cost propagation, incorporating directional flow vector computation would bring the algorithm closer to practical deployment in pathfinding and AI navigation systems.

Conclusion

This study introduced and evaluated a bucket-based, single-pass flow field pathfinding algorithm, grounded in cost-ordered graph traversal and modular sorting structures. Compared to traditional Dijkstra and queue-based methods, the proposed algorithm demonstrated robust and scalable performance, especially on large, obstacle-dense grids with discrete cost values.

Key findings include:

- The queue-based method is fastest on small, sparse grids but deteriorates rapidly in complex scenarios due to reprocessing inefficiencies.
- Dijkstra's algorithm remains robust but incurs higher computational and memory costs due to priority queue overhead.
- The bucket-based algorithm achieves near-linear performance and stable memory usage across all tested scenarios, validating its theoretical efficiency.

Ultimately, the bucket-based approach offers a compelling trade-off between speed, memory, and complexity, making it well-suited for applications requiring predictable, efficient, and scalable pathfinding in discretized environments.

References

[1] Y. Björnsson and K. R. Halldórsson, "Enhanced real-time a*," *Journal of Artificial Intelligence Research*, vol. 24, pp. 851–887, 2005.

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[3] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[4] D. Harabor and A. Grastien, "Online Graph Pruning for Pathfinding on Grid Maps," in *Proc. of AAAI Conference on Artificial Intelligence*, vol. 25, no. 1, 2011.

[5] N. R. Sturtevant, "Grid-based pathfinding: Past, present, and future," *Artificial Intelligence Magazine*, vol. 40, no. 2, pp. 100–116, 2019.

[6] S. Rabin, "Heuristics for pathfinding in games," in *Game Developers Conference*, 2000.

[7] S. Koenig and M. Likhachev, "Lifelong planning a*," *Artificial Intelligence*, vol. 155, pp. 93–146, 2004.

[8] ——, "D* lite," *AAAI Conference on Artificial Intelligence*, 2002.

[9] V. Bulitko and Y. Björnsson, "Incremental heuristic search in games," in *Artificial Intelligence and Interactive Digital Entertainment Conference*, 2007.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[11] X. Cui and H. Shi, "Motion planning with dynamic obstacles using the extended potential field algorithm," *International Journal of Intelligent Computing and Cybernetics*, vol. 4, no. 1, pp. 144–159, 2011.