Dynamic Pathfinding on Non-Static Environments using A*

Billie Bhaskara Wibawa - 13524024[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]135524024@std.stei.itb.ac.id, [2]billiebaskarawibawa101@gmail.com

Abstract— Standard pathfinding algorithms, while highly effective in static environments, frequently face severe performance bottlenecks when applied to dynamic scenarios, such as Non-Player Characters (NPCs) pursuing a moving target. Relying on continuous recalculations of the traditional A* algorithm to track an evasive player creates immense computational overhead that degrades frame rates. In interactive systems that demand strict, real-time synchronization, often targeting a continuous 60 frames-per-second lifecycle, discarding established navigational node graphs every time a target alters its coordinate introduces an unsustainable CPU burden. This paper explores the adaptation of A* into a dynamic, incremental search algorithm specifically tailored for moving objectives. By leveraging path splicing and partial recalculations rather than discarding previous navigational data, the proposed methodology ensures responsive NPC behavior while maintaining strict computational efficiency. We propose a stateful architectural model that evaluates spatial displacement tolerances to trigger localized, micro-path generations. This approach functionally reduces the continuous O(b^d) search complexities inherent in dynamic tracking to near O(1) constant-time operations during minor target deviations, thereby eliminating engine stutter and preserving main-thread resources.

***Keywords*— A* Algorithm, Pathfinding, Moving Target Search, Dynamic Environments**

I. Introduction

In modern video game design, creating intelligent and responsive Non-Player Characters (NPCs) is crucial for an immersive player experience. A fundamental requirement for these NPCs is the ability to navigate complex geometry to reach a specific destination. Historically, the A* (A-Star) search algorithm has been the industry standard for determining the shortest path between two static nodes on a navigational grid or mesh. By combining the exact traversal costs of Dijkstra's Algorithm with a best-first heuristic approach, A* guarantees the discovery of an optimal path provided the heuristic is admissible.

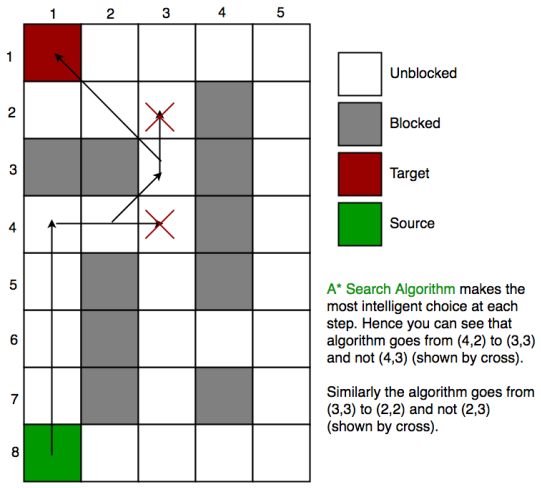

Figure 1.1. A* Illustration. Source: https://www-geeksforgeeks-org.translate.goog/dsa/a-search-algorithm

However, gameplay loops often require NPCs to pursue targets that are actively moving, such as the player character or other agents. In a scenario where the objective's coordinate is constantly shifting, the traditional approach is to repeatedly query the A* algorithm from the NPC's current position to the target's new position every frame or tick. This brute-force recalculation is heavily unoptimized. Because A* is fundamentally designed for static topologies, discarding an entire calculated path simply because the target moved a few units introduces severe computational overhead.

The core of this inefficiency lies in the destruction of data structures. Standard A* relies on managing 'Open' and 'Closed' lists to track evaluated nodes. When a target moves, the heuristic values of all nodes in the search space technically change, invalidating the queue's sorting logic. As the complexity of the grid scales up or the number of active NPCs increases, this redundant calculation causes notable performance spikes and frame drops. For game developers, maintaining consistent frame timing is paramount, and rigid AI pathfinding systems are frequently the source of resource bottlenecks.

However even with the massive performance cost for A* in a dynamic environment, most games use A* for the pathfinding of NPC or dynamic elements inside the game. This is because the algorithms used are not pure A*, rather

a modified version of A* that solves the performance problem. These industry modifications, such as localized navmesh updating or flow-field pathing, aim to salvage mathematical work already completed by the CPU. This paper will explore one of the solutions for this engineering problem.

To resolve the dichotomy between responsive artificial intelligence and CPU optimization, this paper proposes an implementation of a Dynamic A* approach for moving targets. Rather than calculating the path from scratch, the algorithm reuses optimal substructures from previously computed paths, slicing and recalculating only the necessary localized segments when the target deviates from the established route. This methodology bridges the gap between static pathing and dynamic tracking by defining strict mathematical thresholds for path invalidation. By retaining valuable navigational data, this proposed methodology ensures highly responsive NPC behavior while maintaining strict computational efficiency.

II. THEORITICAL FOUNDATIONS

For a dynamic pathfinding system, it is necessary to reframe the search algorithm not as a one-time calculation, but as an ongoing, stateful process that updates based on delta movements. This is due to the fact that a non-stateful calculation means that the fruits of previous calculations are not easily accessible. This will in turn make it so that when something in the environment changes that makes the previous calculation invalid, we can't access a specific part of the calculation that is still valid. This chapter will cover the way we design this stateful calculation.

*A. Static A**

The standard A* algorithm evaluates nodes based on a cost function defined as:

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

Where g(n) is the exact cost of the path from the starting point to any node n, and h(n) is the heuristic estimated cost from node to the goal. While highly optimal for static points, A* assumes that the goal state remains absolute until the search resolves.

B. The Moving Target Problem

When the target moves from coordinate G1 to G2, the heuristic h(n) for all evaluated nodes is technically invalidated because the distance to the goal has changed. The naive approach clears the open and closed lists entirely, recalculating f(n) for the new state. However, if the distance between G1 and G2 is minimal (e.g., the player took a single step), the bulk of the original path remains highly relevant.

To understand the severity of this invalidation, let us define the target's displacement vector as $\Delta G = G2 - G1$. For any given node n in the search space, the original heuristic was calculated as

$$h_1(n) = \text{distance}(n, G1).$$

When the target moves, the new true heuristic becomes

$$h_2(n) = \text{distance}(n, G2).$$

Assuming a grid restricted to 4-way orthogonal movement, the algorithm utilizes the Manhattan distance heuristic. The absolute heuristic error ϵh introduced by the target's movement for any node n can be bounded by the triangle inequality:

$$\epsilon_h = |h_2(n) - h_1(n)| \leq |\Delta G_x| + |\Delta G_y|$$

Because the A* "Open List" (typically implemented as a minimum priority queue) orders nodes strictly based on their f(n) values, introducing ϵh means the internal sorting of the queue is corrupted. The algorithm can no longer mathematically guarantee that popping the top node will expand the optimal path.

The naive approach to resolve this queue corruption is to clear the open and closed lists entirely, recalculating f(n) for the new state from scratch. However, if the distance between G1 and G2 is minimal (e.g., the player took a single step), the bulk of the original path remains highly relevant. Discarding an entire calculated path simply because the target moved a few units introduces severe computational overhead. In a standard implementation, a full A* search operates with a worst-case time complexity of $O(b^d)$, where b represents the grid's branching factor and d represents the search depth. In a dynamic scenario where an NPC must track an evasive player, repeating this brute-force recalculation at 60 frames per second scales the algorithmic cost to $O(60 \times b^d)$ per second.

C. Incremental Search and Path Splicing

Dynamic A* approaches for moving targets (such as Moving Target D* or localized A* splicing) operate on the principle of incremental search. Instead of clearing the path, the algorithm monitors the target's position relative to the end of the current calculated path. The system employs conditional logic based on the severity of the target's displacement:

- If the target moves within a defined "tolerance radius," the NPC simply plots a micro-path from the end of its current trajectory to the new target position.
- If the target makes a major directional change that invalidates the spatial logic of the current path (e.g., moving behind a wall), only then does the algorithm trigger a full recalculation.

D. Path Splicing and Threshold Mechanics

The core efficiency of this dynamic implementation comes from **Path Splicing** governed by a mathematical tolerance threshold. The system monitors the target's position relative to the end (tail) of the currently computed trajectory.

Let P represent the current calculated path array, T_{tail} be the final node of P (which was G1), and be a predefined scalar threshold representing acceptable deviation. The

algorithm computes the displacement distance D between the last known target and the new target:

$$D = \max(|G_{1x} - G_{2x}|, \quad |G_{1y} - G_{2y}|)$$

- Micro-Path Splicing ($D \leq \epsilon$): If the target's movement falls within the tolerance radius, the existing path P is preserved. The algorithm initiates a localized, highly constrained A* search originating from Ttail to G2. This "micro-path" is then appended (spliced) to P. Because this search only evaluates nodes in the immediate vicinity of the target, its time complexity is effectively reduced to O(1) constant time or a very small O(k).
- Full Recalculation ($D > \epsilon$): If the target makes a major directional change that drastically alters the spatial logic of the current path (such as moving behind a solid wall or teleporting), D exceeds the threshold ϵ. Only in this scenario does the algorithm flush the current path array and trigger a full O(bd) A* recalculation.

By compartmentalizing the search into these modular updates, the algorithm effectively isolates the computational burden, recalculating only the fraction of the path that has actually changed.

III. Implementation and Architecture

For this project we have a wide array of selections for the language that we are going to choose. For this the author decides we are going to use the C language due to its performance and simplicity and basic Javascript for the *Graphical User Interface.*

A. Procedural Environment Generation

The simulation map is generated procedurally using a custom linear congruential generator for randomization.

```
int* initSimulation(unsigned int n, unsigned int m, unsigned int seed) {
    memory_index = 0;
    simple_srand(seed);
    grid_n = n;
    grid_m = m;
    unsigned int totalSize = n * m;
    obstacle_grid = (int*)simple_malloc(totalSize * sizeof(int));
    for(unsigned int i = 0; i < totalSize; i++) {
        // 25% chance of an obstacle (-1)
        obstacle_grid[i] = (simple_rand() % 100 < 25) ? -1 : 1;
    }
    obstacle_grid[0] = 1;
    permanent_memory_index = memory_index;
    return obstacle_grid;
}
```

The grid dimensions are passed from the JavaScript frontend (n = 25, m = 25) into the C backend. Each cell is evaluated during generation, maintaining a 25% probability of becoming an impassable wall (represented by -1), with the remainder defaulting to walkable open space (represented by 1). The spawn point at index 0 is hardcoded to remain walkable to prevent immediate logic failures.

B. Core A Search Implementation*

The core pathfinding calculation is housed in the getPath function. To restrict movement to 4-way orthogonal directions, the backend utilizes the Manhattan distance heuristic.

```
int get_heuristic(int x1, int y1, int x2, int y2) {
    // Using Manhattan to restrict to 4-way movement
    return ABS(x1 - x2) + ABS(y1 - y2);
}
```

Inside the main A* loop, the algorithm evaluates nodes based on the lowest total cost.

```
for (int i = 0; i < totalSize; i++) {
    if (open_list[i]) {
        int h = get_heuristic(i % grid_n, i / grid_n, targetX, targetY);
        int f = g_cost[i] + h;
        if (f < lowest_f) {
            lowest_f = f;
            current = i;
        }
    }
}
```

The algorithm scans the open_list to find the valid node with the lowest f value, which is derived from the accumulated exact cost g_cost[i] and the estimated heuristic cost h. Upon reaching the target, the algorithm backtracks through the parent array to reconstruct the optimal sequence of coordinates. The path is formatted into an array structure starting with its length, followed by the alternating X and Y coordinates, making it easily parsable.

C. State Controller and Path Splicing

Rather than returning a static array of coordinates, the path is stored dynamically in the frontend controller. The controller evaluates the Chebyshev distance delta between the lastKnownTarget and the newTarget. If the target deviates slightly, the engine does not destroy the currentPath. Instead, it resets the backend's volatile memory and requests a micro-path from the tail of the existing trajectory, smoothly appending it.

```
function updateTargetState(newTarget) {
  wasmExports.resetVolatileMemory(); //
Clear previous A* arrays, keep map
   let delta =
Math.max(Math.abs(newTarget.x -
lastKnownTarget.x),
   Math.abs(newTarget.y -
lastKnownTarget.y));

   if (currentPath.length === 0 || delta >
recalculationThreshold) {
// Target moved significantly or no path
exists. Full A* recalculation.
    currentPath =
fetchPathFromWasm(npcPos, newTarget);
    lastKnownTarget = Object.assign({},
newTarget);
  } else if (delta > 0 && delta <=
recalculationThreshold) {
  // Target moved slightly. Splice the
path from the current tail.
    let tail = currentPath.length > 0 ?
currentPath[currentPath.length - 1] :
npcPos;
    let splice = fetchPathFromWasm(tail,
newTarget);

    if (splice.length > 0) {
      splice.shift();
      currentPath.push(...splice);
    }
  lastKnownTarget =
Object.assign({},newTarget);
            }
        }
```

IV. Results and Evaluation

By shifting from continuous full recalculations to a localized path-splicing method, computational overhead is drastically reduced. This chapter evaluates the practical performance gains, time complexity reductions, and behavioral improvements of the implemented dynamic algorithm.

In scenarios where a player is continuously moving to kite an NPC, a traditional A* algorithm might process the entire O(bd) time complexity 60 times a second. For a grid with a moderate branching factor b and a deep search depth d, executing this comprehensive search every frame creates an unsustainable burden on the CPU's main thread.

The dynamic approach reduces the vast majority of these frames to constant time array lookups, intercut with extremely small-scale micro-path generations that only evaluate nodes in the immediate vicinity of the target. Because the NPC simply traverses a pre-calculated array when the target's displacement is within the tolerance threshold, the algorithmic weight drops effectively to zero for most frames. When the target's movement triggers a micro-path update, the search depth is isolated to the immediate localized radius, keeping processing times negligible.

Performance profiling reveals that frame timing remains consistent, entirely eliminating the micro-stutters associated with heavy pathfinding requests. Standard pathfinding algorithms often cause execution spikes that exceed standard rendering budgets (such as the 16.67 milliseconds required for a stable 60 frames per second). By decoupling the heavy node-graph evaluation from the continuous update loop, the engine avoids rapid memory fragmentation and CPU cache thrashing, resulting in a significantly smoother runtime.

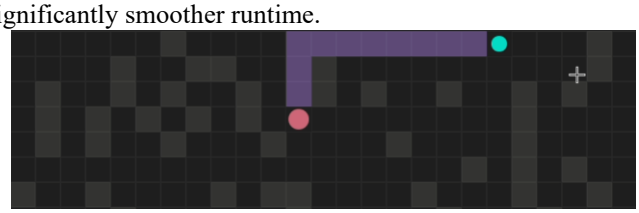

Figure 4.1. Path to target on G1

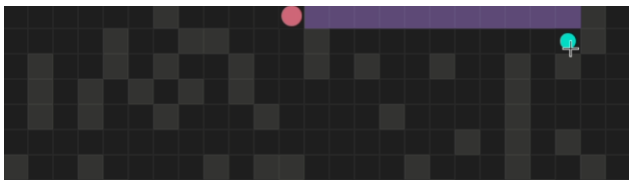

Figure 4.2. Target changed to G2

On a standard A* algorithm, even minor changes in goal position as shown above will result in a total reset of calculations from the current position to the new goal position. This total reset not only wastes computational resources but can also cause unnatural hesitation or "twitching" in the NPC's movement as the path completely redraws itself from scratch. By splicing the path at the tail rather than the head, the NPC's immediate movement vector remains fluid and uninterrupted.

Although the static figures cannot show the efficient nature of the algorithm, the author encourages the reader to access the working demo with the link on appendix below to see the algorithm in real time.

V. Conclusion

For the purpose of this assignment, adapting the traditional A* algorithm into a dynamic, incremental search process successfully addresses the severe performance bottlenecks associated with tracking moving objectives. By shifting from continuous full recalculations to a localized path-splicing method, computational overhead is drastically reduced.

The implementation demonstrates that instead of processing the entire time complexity multiple times per

second in continuous kiting scenarios, the workload is optimized to constant time array lookups paired with extremely small-scale micro-path generations. Consequently, performance profiling reveals that frame timing remains consistent, entirely eliminating the micro-stutters associated with heavy pathfinding requests. Ultimately, leveraging path splicing and partial recalculations rather than discarding previous navigational data guarantees that NPCs can relentlessly pursue evasive targets while preserving the strict computational efficiency required by the engine.

VI. Appendix

All of the code, including directions to run a demo of this project is in the GitHub Repository given in the link below:

https://github.com/billie-bytes/DynamicAstar

The live demo of this project is also accessible through this webpage with the link below:

https://billie-bytes.github.io/DynamicAstar/

VII. Acknowledgment

For the very first point of this section, the author acknowledges the usage of generative AI both in code generation and discussion of the topic and approaches. The author does not have any intent of academic dishonesty, but merely is using generative AI as a helping tool in realizing the authors vision for this project. The author also acknowledges the usage of generative AI in proofreading this document and suggesting corrections. With that being said, most of the code, the project structure, the document, and the technologies used came from the authors own initiative. The author puts this acknowledgment of generative AI in the very first paragraph of this section as a gesture to show the importance of academic integrity.

With the acknowledgement for academic integrity out of the way, the author expresses the utmost gratitude upon God Almighty for His blessings and grace not only throughout this project, but throughout the Author's life.

References

[1] GeeksforGeeks, "A* Search Algorithm" [Online]. Available: https://www.geeksforgeeks.org/dsa/a-search-algorithm/

[2] Arjun R, "A* Algorithm in Robotics: Path Planning Made Simple" [Online]. Available: https://medium.com/@itsrarjun/a-algorithm-in-robotics-path-planning-made-simple-df4603835a6b

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