Procedural Generation of Game Maps using Dynamic Programming

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— Procedural level generation offers immense replayability in video games but frequently struggles with creating reliable, playable maps. Relying purely on stochastic algorithms, such as random walks or cellular automata, often leads to broken navigation, impassable bottlenecks, or unintended pacing that severely diminishes the player experience. This paper explores the application of Dynamic Programming not merely as a validation tool, but as a primary, deterministic generator for procedurally generated environments. We propose a dynamic path carving methodology where the algorithmic space is initialized as a dense grid. Each cell is assigned a randomized topological weight representing traversal resistance based on designer constraints. By leveraging bottom up tabulation, the game engine calculates the contiguous path of least cumulative cost from the designated spawn point to the objective. Unlike traditional generative methods that build structures blindly, this approach utilizes optimal substructure to engineer the layout directly from a cost matrix. Consequently, the generated map is mathematically guaranteed to be both procedurally varied and entirely passable, eliminating the risk of disconnected zones or dead ends. Furthermore, because the execution time is strictly bounded by the grid dimensions, this technique prevents the performance spikes commonly associated with recursive validation algorithms. Ultimately, integrating dynamic path carving into procedural pipelines empowers developers to synthesize organic, unpredictable environments while maintaining absolute control over spatial integrity and game pacing.

***Keywords*— Dynamic Programming, Procedural Level Generation, Random Number Generation, Pathfinding**

I. Introduction

Video game designer has always been facing the problem of creating level designs that are both playable and also unique. Creating levels manually is always an option, but for games that are designed to be played over and over, this approach quickly falls due to the fact that a handcrafted map would never change in a major way each replay. Video game genres such as *Rogue-like* are fundamentally designed to be played over and over again. Unlike other genres of video games, where progression meaning accessing further stages and maps, the main design element of a *Rogue-like* game is for the player to be reset into the same stage over and over again. In such a game, a stage does not represent the map or playable area itself, but it represents the progression of the player in other ways, such as unlocking previously unnatained abilities or gear. Thus, the player will not progress the game through finishing a map and continuing to another map, instead the player will progress to the stages of the game through unlocking abilities, gear, and also the knowledge of the game itself. Thus, each time the player character dies in-game, they are reset to a specific starting point all over again, but instead of losing all their progress, instead the player retains the reward from previous run.

To maintain player engagement across countless iterations, Rogue-likes and similar genres heavily rely on Procedural Content Generation (PCG). PCG algorithms dynamically synthesize game environments at runtime, ensuring that no two playthroughs are exactly alike. Historically, developers have utilized stochastic approaches—such as cellular automata, random walks, BSP (Binary Space Partitioning) trees, or Perlin noise—to generate these spaces. While these techniques excel at creating visually chaotic and organic structures, they operate independently of the spatial logic required for a functional gameplay loop. They place tiles and walls based on mathematical noise rather than navigational intent.

This blindness introduces a significant vulnerability into the level design pipeline: the generation of unplayable or heavily flawed maps. Relying purely on randomized algorithms frequently yields topological anomalies, such as disconnected key zones, impassable bottlenecks, or frustratingly long dead ends. In a game loop where permanent death is a core mechanic, player trust in the fairness of the environment is paramount; an unwinnable map generated by a faulty algorithm severely damages the user experience. To mitigate this, developers typically employ secondary validation scripts—such as running a pathfinding search post-generation—to verify if the objective is reachable from the spawn point. If the validation fails, the map is either heavily mutated or discarded entirely to start anew. However, this recursive cycle of "guess-and-check" introduces severe performance spikes, unpredictable loading times, and high computational overhead, especially as grid complexity scales up.

To resolve the dichotomy between procedural unpredictability and guaranteed playability, this paper proposes a novel methodology: utilizing Dynamic

Programming (DP) not as a retrospective validation tool, but as the primary generation mechanic. Rather than building structures blindly and checking them later, we initialize the environment as a dense grid where each cell holds a randomized topological weight. By applying a bottom-up tabulation approach, the algorithm calculates the optimal substructure, the contiguous path of least cumulative cost, directly through the cost matrix. This dynamic path carving mathematically guarantees the existence of a continuous, playable route from its inception. Furthermore, because the execution time is strictly bounded by the grid dimensions to an O(N×M) time complexity, it entirely eliminates the recursive performance drops of traditional generation. The remainder of this paper explores the theoretical framework of this DP approach, details its algorithmic implementation, and analyzes its performance advantages over traditional stochastic map generation.

II. THEORITICAL FOUNDATIONS

To establish Dynamic Programming (DP) as a viable primary generator for game environments, it is necessary to reframe the map not as a collection of physical tiles, but as a mathematically quantifiable space. This section defines the theoretical shift from stochastic, generate-and-test methods to a constructive, deterministic methodology governed by optimal substructure.

A. Constructive vs. Generate-and-Test Approaches

Procedural Content Generation (PCG) in level design broadly falls into two algorithmic approaches, constructive, and generate-and-test approach. The generate-and-test approach produces algorithms (e.g., random walks, purely stochastic cellular automata) that build the space blindly and rely on a secondary heuristic or pathfinding script to evaluate playability. If the test fails, the generation is discarded or repaired. This is inherently recursive and computationally volatile. Meanwhile constructive approach produces algorithms that generate content that is mathematically guaranteed to meet all design constraints upon execution, requiring no post-generation validation.

The dynamic path carving methodology proposed in this paper is strictly a constructive approach. By engineering the path *before* solidifying the environmental geometry, the algorithm bypasses the need for secondary validation, fundamentally altering the computational flow of level generation.

B. The Map as a Weighted Topological Matrix

In this framework, the algorithmic space is not a blank canvas, but rather a pre-initialized grid representing traversal resistance. Let the game map be defined as a 2D topological matrix of dimensions $N \times M$. Instead of cells holding boolean values (e.g., walkable vs. non-walkable), each cell is assigned a scalar weight W(i,j) defined by a randomized distribution function. This weight signifies the "cost" or "resistance" of carving a path through that specific coordinate.

By utilizing a seeded Random Number Generator (RNG) to populate W(i,j), we can maintain the unpredictable variance required of Rogue-like games, while translating that chaos into a strictly quantifiable data structure.

C. Optimal Substructure and the Bellman Equation

Dynamic Programming excels at solving complex optimization problems by breaking them down into simpler, overlapping subproblems. For DP to be applicable to procedural generation, the problem of creating a continuous, playable map must exhibit optimal substructure, meaning the optimal path from the spawn point to the objective contains the optimal paths to all intermediary points along the way.

Assuming a generalized directional flow from a spawn point at the origin of (0,0) to an objective at (N-1, M-1), we can define the cumulative cost matrix using a bottom-up tabulation approach. The state transition function, derived from the Bellman equation, calculates the minimum cost to reach any cell (i,j):

$$C(i,j) = W(i,j) + min\begin{cases} C(i-1,j) \\ C(i,j-1) \end{cases}$$

Because the DP algorithm calculates sequentially based on previously solved adjacent cells, it evaluates the entire topological landscape in a single, systematic pass. Once the tabulation reaches the objective, the game engine simply backtracks from (N-1, M-1) to (0,0) following the steepest descent in the cumulative cost matrix C. This backtracking physically "carves" the guaranteed walkable path into the final level geometry.

D. Algorithmic Determinism and Time Complexity

Traditional stochastic generation paired with post-generation pathfinding (such as A* search algorithms) often faces unpredictable performance. A* searches expand nodes based on heuristics, and in highly complex or broken mazes generated by noise algorithms, worst-case scenarios can cause severe performance spikes.

Conversely, the DP approach enforces absolute algorithmic determinism. Because the bottom-up tabulation systematically processes every cell in the grid exactly once, the execution time is strictly bounded. The time complexity for generating the guaranteed critical path is consistently $O(N \times M)$.

This linear relationship to the grid's total area ensures that the computational overhead remains completely predictable, regardless of the chaotic nature of the randomized weights within . Consequently, dynamic path carving guarantees both spatial integrity (a continuous path) and computational integrity (no performance spikes), forming an ideal foundation for real-time, procedural pipelines.

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. Creating the Random Grid and Main Path

We can create the random grid by just creating a flat array.

```
int* generateTopologicalGrid(unsigned int
n, unsigned int m, unsigned int seed) {
    simple_srand(seed);

    unsigned int totalSize = n * m;
    int* grid =
(int*)simple_malloc(totalSize *
sizeof(int));

    for (unsigned int i = 0; i <
totalSize; ++i) {
        grid[i] = (simple_rand() % 100) +
1;
    }

    return grid;
}
```

This function will generate a board with the specified size and then fill each cell with a random value.

```
int* carveOptimalPath(int*
topologicalGrid, unsigned int n, unsigned
int m) {
    for(unsigned int i = 0; i < totalSize;
++i) {
        pathGrid[i] = 0;
    }
    costMatrix[0] = topologicalGrid[0];
    for (unsigned int x = 1; x < n; ++x) {
        costMatrix[x] = costMatrix[x - 1]
+ topologicalGrid[x];
    }
    for (unsigned int y = 1; y < m; ++y) {
        costMatrix[y * n] = costMatrix[(y
- 1) * n] + topologicalGrid[y * n];
    }

    for (unsigned int y = 1; y < m; ++y) {
        for (unsigned int x = 1; x < n;
++x) {
            int costFromLeft =
costMatrix[y * n + (x - 1)];
            int costFromTop =
costMatrix[(y - 1) * n + x];

            costMatrix[y * n + x] =
topologicalGrid[y * n + x] +
MIN(costFromLeft, costFromTop);
        }
    }
    int currentX = n - 1;
    int currentY = m - 1;
    pathGrid[currentY * n + currentX] = 1;
    while (currentX > 0 || currentY > 0) {
        if (currentX == 0) {
            currentY--;
        } else if (currentY == 0) {
            currentX--;
        } else {
            int costLeft =
costMatrix[currentY * n + (currentX - 1)];
            int costTop =
costMatrix[(currentY - 1) * n + currentX];

            if (costLeft < costTop) {
                currentX--;
            } else {
                currentY--;
            }
        }
        pathGrid[currentY * n + currentX]
= 1;
    }
    return pathGrid;
}
```

Note that the functiona above has been shortened and to only retain most of the algorithm and not show irrelevant technical details.

The is the main algorithm for this project, which is a standard Dynamic Programming path traversal based on optimized cost. However, even after using such approach, the generated path would still be somewhat close to a 45° angle. This is mathematically expected since the random value is uniformly distributed.

In a grid where you can only move down and right, every single valid path from the top-left to the bottom-right has the exact same number of steps. Let L be the total number of cells visited:

$$L = N + M - 1$$

Because every path is exactly the same length, the DP algorithm cannot save costs by taking a "shorter" route. It can only save costs by finding cells with lower random numbers.

Let μ be the average value of a random cell (e.g., 50.5). Because of the Law of Large Numbers, the total cost of any macroscopic path you take will naturally gravitate toward $L \times \mu$. Since the noise is uniform, there are no macroscopic "valleys" (large patches of cheap cells) or "mountains" (large patches of expensive cells). The entire grid looks statistically identical from a macro perspective.

This is where the 45-degree line emerges. Since the algorithm is looking for the optimal sequence of low-cost cells, it needs choices.

The number of possible paths passing through a specific coordinate follows a binomial distribution, calculated as

$$\binom{x+y}{x} \times \binom{(N-1-x)+(M-1-y)}{N-1-x}$$

Therefore, the DP algorithm statistically gravitates toward the 45-degree diagonal because that is where it has the highest mathematical probability of finding local bypasses around high random values.

The picture above shows one of many cases where this problem occurs. To solve this, we can add subgoals in between the start node and the end goal. While the nature of DP still restricts the path for an actual random path, this still creates more unpredictable path and turns.

B. Generating Non-linear Topology

While the primary dynamic programming algorithm excels at identifying the global minimum-cost route, it naturally produces a singular, strictly continuous trajectory. In uniform or pseudo-random topological grids, this primary path often exhibits a strong statistical bias toward a central diagonal to maximize its routing degrees of freedom. To disrupt this linearity and introduce complex macro-structures, we implement a stochastic post-processing phase. Rather than attempting to compute a multi-branch tree in a single pass, which would exponentially increase computational overhead. This secondary phase treats the established main path as an immutable, collective target state.

During this phase, the algorithm arbitrarily selects a set of unvisited coordinates to serve as "substart" nodes. These substarts act as isolated origins for independent branch generation. For each substart, the dynamic programming sequence is re-initialized to calculate the local minimum-cost route required to intersect the main path. By utilizing DP for these tertiary routes, the algorithm ensures that the overall environment's movement constraints and cost penalties are strictly respected, maintaining optimal local efficiency even within the randomized branches.

To maximize the procedural variety of the generated environments, the parameter dictating the total number of substarts is randomized per execution. This variable branching frequency guarantees that the final network topology remains unpredictable. Environments may range from sparsely branched, near-linear corridors to dense, highly dendritic networks with unpredictable 90-degree adjustments. Ultimately, this layered approach, combining a globally optimal backbone with stochastically driven, locally optimal branches, produces a more organic and structurally diverse pathing network.

```
void addRandomBranch(int* pathGrid,
unsigned int n, unsigned int m) {
    unsigned int totalSize = n * m;
    int emptyCount = 0;
    for (unsigned int i = 0; i <
totalSize; i++) {
        if (pathGrid[i] == 0)
emptyCount++;
    }

    if (emptyCount == 0) return;

    unsigned int startX, startY;
    do {
        unsigned int idx = (unsigned
int)simple_rand() % totalSize;
        startX = idx % n;
        startY = idx / n;
    } while (pathGrid[startY * n + startX]
!= 0);
    int minDistance = totalSize + 1;
    unsigned int targetX = startX, targetY
= startY;

    for (unsigned int y = 0; y < m; y++) {
        for (unsigned int x = 0; x < n;
x++) {
            if (pathGrid[y * n + x] == 1)
{
```

```
                int dist =
simple_abs((int)startX - (int)x) +
simple_abs((int)startY - (int)y);
                if (dist < minDistance) {
                    minDistance = dist;
                    targetX = x;
                    targetY = y;
                }
            }
        }
    }
    int currX = startX;
    int currY = startY;

    while (currX != (int)targetX || currY
!= (int)targetY) {
        pathGrid[currY * n + currX] = 1;

        if (simple_abs(currX -
(int)targetX) > simple_abs(currY -
(int)targetY)) {
            currX += (currX <
(int)targetX) ? 1 : -1;
        } else {
            currY += (currY <
(int)targetY) ? 1 : -1;
        }
    }
}
```

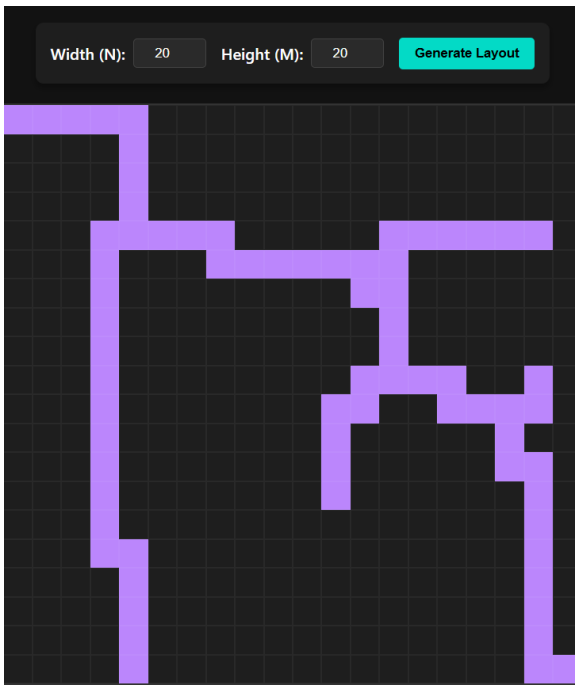
The code above shows the process of creating a new path. The main algorithm stays largely the same as they all are based on dynamic programming, but for this one it goes to the cell that is part of the main path.

C. Grid Representation

Because the grid itself is really only a flat array on the source code, all dynamic programming adjacent cell calculations are really just jumping a specific amount of steps that represent a row. In the end it matters not since all computation is exactly the same. If anything, it actually improves performance since it all resides on adjacent memory.

IV. Results and Evaluation

The final implementation of the procedural generation has sucessfully generated paths that are guaranteed to be different and passable at the same time. Our definition of a valid map is one that starts at (0,0) and ends at (M-1, N-1). Visually, this will mean that each map will start at the upper left corner of the map and end at the bottom right corner of the map. For different scenarios with different definitions of a valid map, it is fairly trivial to modify the algorithm to suit any scenarios needs.



As demonstrated in the generated visual outputs, the resulting paths are highly unique and unpredictable, even though the core dynamic programming approach is functionally "static" by being restricted to only moving down and right.

The raw mathematical expectation for a uniform noise grid limits the dynamic programming sequence, statistically pulling the optimal path toward a 45-degree central diagonal. Our results show that the implementation of the addRandomBranch subroutine successfully disrupts this linearity.

By selecting unvisited coordinates as isolated "substarts" and calculating their local minimum-cost route

to intersect the main path, the algorithm generates highly complex, dendritic networks. Importantly, the generated branches still strictly respect the environment's movement constraints and cost penalties, maintaining optimal local efficiency while looking entirely organic to the player.

V. Conclusion

For the purpose of this assignment, we have succesfully created a simple and lightweight algorithm that is guaranteed to produce maps that are both unique and valid.

By redefining procedural generation away from stochastic "guess-and-check" methods and toward a constructive, deterministic methodology, this project resolves a critical bottleneck in Rogue-like level design. Initializing the environment as a quantifiable topological matrix and applying a bottom-up dynamic programming tabulation ensures that the optimal substructure, the playable path, is engineered natively into the map's foundation.

Furthermore, the architectural choice to implement the topological grid as a flat array in C proved highly beneficial. It not only streamlined the mathematical calculation of row-stepping for adjacent cells but also maximized adjacent memory access, yielding exceptional performance. Coupled with a stochastic post-processing phase that injects random tertiary branches to break linear diagonal bias, the methodology offers a robust solution. Ultimately, this dynamic path carving technique empowers game developers to rapidly synthesize organic, visually unpredictable environments without sacrificing spatial integrity, performance consistency, or player trust.

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/StimaMazeGen

Working public demo for this project can be accessed through this webpage:

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

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, "Dynamic Programming or DP" [Online]. Available: https://www.geeksforgeeks.org/dsa/dynamic-programming/

[2] Kyzrati, "Procedural Map Generation" [Online]. Available: https://www.gridsagegames.com/blog/2014/06/procedural-map-generation/

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

Billie Bhaskara Wibawa
13524024